DEPOSIT : une approche pour exprimer et déployer des politiques de collecte sur des infrastructures de capteurs heterogènes et partagées

Cyril Cecchinel

UNIVERSITÉ CÔTE D'AZUR
ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

Thèse de doctorat

Présentée en vue de l'obtention du grade de

docteur en Sciences

de l'Université Côte d'Azur

Mention : Informatique

Présentée et soutenue par
Cyril CECCHINEL

DEPOSIT :
Une approche pour exprimer et déployer des politiques de collecte de données sur des infrastructures de capteurs hétérogènes et partagées

Dirigée par Philippe Collet

Soutenue prévue le 08/11/2017
Devant le jury composé de :

François	Baude	Professeur des Universités, Côte d'Azur	Examinateur
Philippe	Collet	Professeur des Universités, Côte d'Azur	Directeur
Benoît	Combemale	Professeur des Universités, Toulouse	Examinateur
Sébastien	Mosser	Maître de conférence, Côte d'Azur	Encadrant
Romain	Rouvoy	Professeur des Universités, Lille I	Rapporteur
Houari	Sahraoui	Professeur, Université de Montréal	Rapporteur

Remerciements

En premier lieu, merci à Philippe d'avoir accepté de diriger mes travaux de recherche, toujours avec une note d'humour et de bonne humeur qui te ressemble bien. L'ensemble de tes retours ainsi que tes nombreux conseils de rédaction scientifique m'ont permis d'avancer et de rester sur les rails de ce long voyage. Sébastien, cette thèse représente les cours de snowboard que tu as pu me donner pour gagner le droit de rester dans l'équipe. L'ensemble de tes conseils m'a permis de passer d'une piste verte balisée sans danger à une descente hors-piste où l'on doit être autonome et faire sa propre trace (et sans faute de carre si possible). C'est grâce à toutes nos discussions sur un télésiège ou devant un tableau blanc que les idées défendues dans cette thèse ont pris racine.

Merci au *labo* (laboratoire I3S à Sophia-Antipolis) de m'avoir accueilli durant ces trois années. Un merci à Mireille pour ses conseils dans mon parcours d'enseignant à l'IUT Nice Côte d'Azur et à Anne-Marie pour avoir pris en tout temps « température » du bureau. Je n'oublie pas également les *thésards* qui ont suivi des chemins parallèles. Ivan, en plus d'avoir été mon *parrain* et ami en école d'ingénieurs, a été un super co-bureau qui a su partager son environnement malgré nos divergences iséro-savoyardes. Je n'oublie pas non plus Benjamin pour son Nerf automatique, Stéphanie pour son chocolat noir 99 % et Sami pour toutes les capsules de café d'une certaine marque suisse échangées.

Ma thèse a également été rythmée par le programme EIT Digital qui m'a donné l'opportunité de passer 6 mois au sein du laboratoire SnT de l'université du Luxembourg. Je remercie vivement Yves Le Traon pour l'accueil fait au sein de son équipe. Un *villmools Merci* également à François et Grégory pour leur collaboration et de m'accueillir dans les prochaines semaines dans leur startup récemment fondée. Merci également aux thésards, néo-docteurs et staff luxembourgeois, Élise, Ludovic, Assaad, Thomas et Niklas pour toutes nos discussions (pas toujours scientifiques) et bon moments partagés.

Je tiens ensuite à remercier les professeurs Rouvoy et Sahraoui pour avoir pris le temps de lire et de rapporter ce manuscrit. Les rapports m'ont été précieux pour préparer la soutenance et obtenir un avis extérieur sur mes travaux. Merci également aux professeurs Baude et Combemale d'avoir accepté de participer à ce comité de thèse.

Il y a également les amis qui doivent être remerciés pour leur soutien indéfectible. Dima et Lydie merci d'avoir supporté, et sans l'avoir demandé, les hauts et les bas de la thèse et pour toujours avoir été présents. Matthieu, frère de thèse expatrié, merci d'avoir passé de longues heures à échanger sur nos thèses respectives (dont la relecture de ce manuscrit) et pour m'avoir accueilli (et supporté ma collocation !) à Luxembourg. Tu as également survécu à ma conduite à gauche (et quelque peu dynamique) sur les routes de Nouvelle-Zélande ! Je n'oublie pas mes amis du Sud : Endy, Sam, Anne-Cécile, Jérôme, Thibaut et Mathilde.

Je n'oublie pas ma Savoie natale et de cœur. En premier lieu mes parents, Bruno et Sylvie, qui ont su m'encourager à aller toujours le plus loin possible dans mes études et qui ont toujours été bienveillants. Un merci également à mon petit frère Kévin, qui en plus d'avoir toujours été curieux vis-à-vis de mes travaux, m'a sauvé un nombre incalculable de fois en me rappelant les fêtes familiales lorsque j'étais complètement noyé par mon travail. Il y a aussi les grands et arrières grands-parents. Un merci spécial à mon grand-père Daniel qui a toujours porté de l'attention et qui a pris le temps de lire une bonne dizaine de fois ce manuscrit. (*aparté*) Papy, je sais que tu vas lire très

attentivement ces remerciements, donc merci pour le soutien scolaire lorsque j'étais plus jeune et pour m'avoir ensuite guidé dans le choix de mon école d'ingénieur et pour m'avoir encouragé à continuer en thèse (mais bon, il va falloir penser à s'arrêter maintenant). Merci également à mon cousin Fabien d'avoir accepté de parcourir l'Europe (notamment la Finlande et la Hongrie), après avoir compris que le moyen le plus simple pour passer du temps avec moi était de se joindre à mes déplacements.

Pour conclure, on a toujours peur lors de la rédaction de cette section d'oublier des personnes. Ainsi, pour éviter toute omission involontaire de ma part, cher lecteur, merci pour votre attention !

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

1.1 Contexte et problématiques

L'Union internationale des télécommunications (UIT) définit l'Internet des objets comme une « infrastructure mondiale pour la société de l'information, qui permet de disposer de services évolués en interconnectant des objets (physiques ou virtuels) grâce aux technologies de l'information et de la communication existantes ou en évolution » [SU05]. Ces objets peuvent aller du gadget portable comme une montre intelligente, à la dernière voiture connectée tout en passant par des dispositifs intermédiaires comme une station météo. Les objets connectés à Internet sont souvent qualifiés de *smart* (intelligent) par leur capacité à remonter, à un utilisateur, des informations sur leur environnement.

Interconnectés au sein d'infrastructures à grande échelle telles que des bâtiments ou des villes, les objets connectés participent à la réalisation d'applications consommant des données en provenance de l'environnement physique. Par exemple, les applications d'aide au stationnement au sein des villes de Nice et de Westminster reposent respectivement sur 1000 et 3000 capteurs répartis au niveau des différents emplacements de stationnement. Au sein du Grand-Duché de Luxembourg, 95% des compteurs d'électricité et de gaz sont connectés à Internet afin d'offrir une facturation plus précise de la consommation énergétique. Cependant, ces différentes infrastructures ont un coût d'installation et de maintenance non négligeable (plusieurs millions d'euros pour les infrastructures équipant la ville de Nice et le G.D. du Luxembourg). Ainsi, il est primordial qu'elles puissent être réutilisées lorsque de nouveaux besoins en collecte de données sont identifiés au fil du temps. Toutefois, les infrastructures actuelles ont le défaut d'être spécifiques à un domaine et d'être configurées pour une tâche de collecte particulière avec peu ou aucune possibilité de réutilisation menant à des déploiements redondants [KBG+15]. En plus de l'aspect financier, des infrastructures redondantes créent des silos isolant les données. Ce constat fait apparaître la première problématique de cette thèse *(P_1, partage)* « **comment une infrastructure de capteurs peut-elle être partagée entre plusieurs collectes de données afin d'éviter des déploiements redondants ?** »

Les prévisions du groupe Gartner annoncent que le nombre d'objets connectés va passer de 8.4 milliards (2017) à plus de 20 milliards en 2020. En plus de l'explosion du nombre d'applications interagissant avec ces objets, le nombre de constructeurs va également augmenter pour atteindre un marché estimé à 3000 milliards de dollars US [Gar17]. Ces constructeurs vont apporter de nouveaux types de matériels et de nouvelles technologies. Réciproquement, des constructeurs présents aujourd'hui sur le marché risquent de disparaître, ou de voir leur technologie absorbée par un concurrent. Les collectes de données développées reposent sur une synergie entre matériel et logiciel : le *matériel* permet de collecter des informations sur l'environnement ou d'interagir physiquement sur ce dernier, alors que le *logiciel* permet d'exposer, à travers une application, ces informations à un utilisateur afin qu'il puisse comprendre et/ou prendre des décisions sur son environnement.

Actuellement, la programmation d'une plateforme au sein d'une infrastructure de capteurs s'effectue directement au niveau de son contrôleur, en utilisant des langages dits de bas-niveau et en interagissant directement avec le matériel de la plateforme. Les développements sont par conséquent spécifiques à une plateforme ou à une même architecture des plateformes. Toutefois, l'augmentation du nombre de plateformes matérielles disponibles sur le marché rend les infrastructures hétérogènes. Ainsi, l'expression des collectes de données alimentant des applications devient une tâche complexe

puisque les langages de programmation bas-niveau n'offrent que peu d'abstractions logicielles et doivent prendre en compte les spécificités de chacune des plateformes composant l'infrastructure. Cette situation conduit à la seconde problématique *(P_2, adaptation)* « **comment une collecte de données peut-elle être exprimée sur une infrastructure hétérogène de capteurs ?** »

Ce rapport étroit entre matériel et logiciel force deux communautés distinctes à travailler de concert. D'un côté, on retrouve les ingénieurs logiciels dont la responsabilité est de traduire sous forme d'application des spécifications reposant sur des objets, et d'autre part des experts réseau dont la responsabilité est de configurer les différents objets et d'assurer la maintenance du réseau. Malgré ce besoin d'interactions, ces deux communautés restent imperméables [Pic10] : les ingénieurs logiciels attendent des experts réseau que le réseau envoie automatiquement des données au format supporté par l'application développée et les experts réseau attendent des ingénieurs logiciels qu'ils expriment des collectes de données optimisées en fonction des contraintes matérielles des plateformes présentes dans l'infrastructure. Cependant, les problématiques propres aux plateformes présentes dans les infrastructures, *par ex.*, gestion de la consommation énergétique, sont éloignées du champ de compétences d'un ingénieur logiciel. De manière similaire, la compréhension des spécifications auxquelles doit répondre un ingénieur logiciel est hors des préoccupations métiers de l'expert réseau. Ce constat nous amène à la troisième dernière problématique *(P_3, séparation)* « **comment permettre à un ingénieur logiciel et un expert réseau de travailler ensemble à l'expression de collecte de données tout en restant concentré sur leurs compétences respectives ?** »

1.2 Contribution : Exploitation par des ingénieurs logiciels d'une infrastructure de capteurs partagée

L'Internet des Objets repose sur des capteurs, *i.e.*, un dispositif physique permettant de transformer une grandeur physique observée en une autre grandeur physique, souvent de nature électrique, utilisable à des fins de mesures et sur des actionneurs, *i.e.*, un dispositif transformant une énergie en un phénomène physique afin de modifier l'environnement dans lequel il est déployé. Dans le cadre de cette thèse, nous nous intéressons à la *collecte de données* à partir d'une infrastructure composée de capteurs (appelée par la suite *infrastructure de capteurs*) permettant de fournir des données à une application.

Les infrastructures de capteurs sont elles-mêmes composées de réseaux de capteurs (permettant l'acquisition des données de l'environnement) acheminant des données, à travers un ensemble de plateformes relais des données, jusqu'à un routeur frontal où elles sont ensuite envoyées sur un réseau IP traditionnel (*par ex.*, Internet). De l'autre côté, se trouvent des applications consommant en temps réel les données produites par le réseau de capteur ou des serveurs stockant ces données pour des analyses et usages futurs [FRL07]. La figure Fig. 1.1 illustre une infrastructure hétérogène composée de plateformes hébergeant des capteurs, de plateformes relayant les données de capteurs, d'un routeur frontal et de plusieurs applications recevant les données de capteurs.

Les plateformes embarquées dans le réseau de capteurs ont des capacités de calcul variables en fonction du matériel. Par exemple, il est possible d'effectuer des tâches complexes sur des plateformes de type nano-ordinateur, *par ex.*, RaspberryPi, alors que

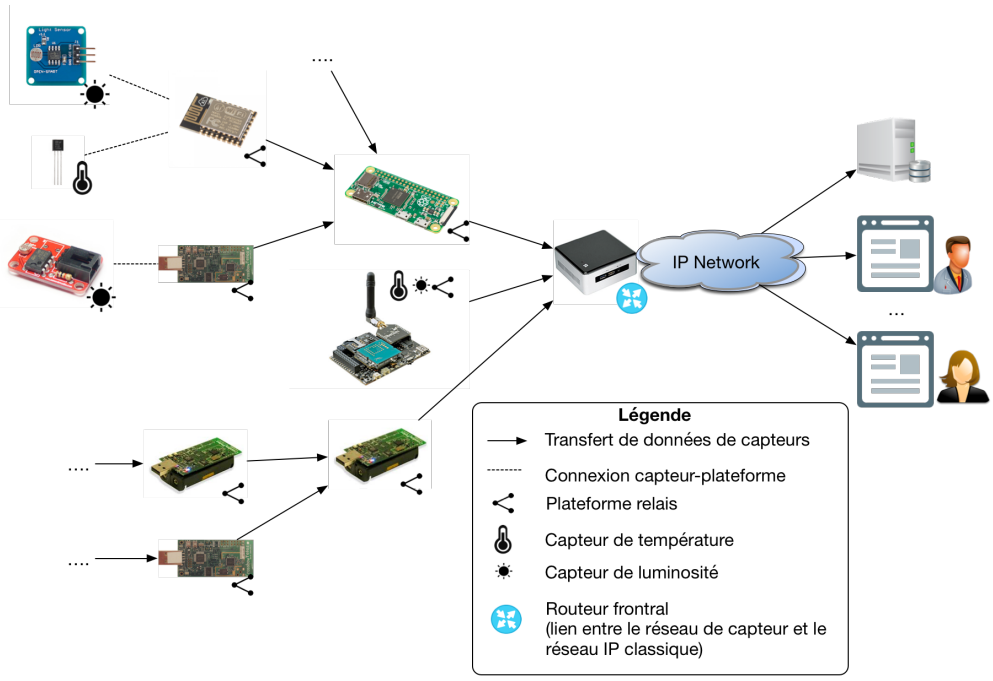


FIGURE 1.1 – Illustration d'une infrastructure hétérogène de capteurs

des plateformes basées sur un micro-contrôleur, *par ex.*, TI MSP430, n'effectueront que des tâches simples. Le récent paradigme de l'*edge computing* [SCZ+16] (« informatique sur les bords ») encourage par ailleurs à placer directement les éléments de logique métier sur les plateformes du réseau de capteurs, notamment dans l'objectif de réduire le trafic réseau, et donc l'énergie électrique consommée.

La contribution globale de cette thèse a pour objectif d'exploiter les ressources d'un réseau de capteurs en permettant à un ingénieur logiciel d'exprimer une collecte de données (sous forme de **politique de collecte de données**) sur une infrastructure hétérogène à grande échelle, tout en restant focalisé sur ses préoccupations métiers. L'infrastructure devra être également réutilisable entre plusieurs politiques de collecte de données grâce à un mécanisme de partage automatique de l'infrastructure.

Afin de répondre à la problématique *(P_1, partage)*, nous introduisons un opérateur de composition et un proxy de composition. L'opérateur de composition a pour objectif de composer deux politiques de collecte de données et de renvoyer une nouvelle politique contenant le résultat de la composition des intentions métier. Cette nouvelle politique composée pourra alors être déployée sur l'infrastructure de capteurs. Sur les infrastructures partagées entre plusieurs ingénieurs logiciels, un ingénieur n'a pas connaissance des autres politiques déployées et ne peut, ainsi, pas utiliser l'opérateur de composition. Nous définissons alors un proxy de composition, placé entre les ingénieurs logiciels et l'infrastructure de capteurs et ayant la charge de collecter les nouvelles politiques de collecte de données et de les composer *automatiquement* avec les politiques précédemment déployées. Grâce à ce mécanisme de partage, plusieurs politiques peuvent être exécutées simultanément sur l'infrastructure. Ce mécanisme permet ainsi de ne plus avoir à effectuer de déploiement redondant lorsqu'une nouvelle application sera identifiée et que les déploiements actuels seront suffisants.

Afin de répondre à la problématique *(P_2, adaptation)*, nous présentons une contribution visant à adapter une politique de collecte de données, exprimée indépendamment des considérations matérielles, sur une infrastructure cible. Nous présentons notamment

un opérateur visant à placer chaque action d'une politique de collecte de données sur une plateforme de l'infrastructure de capteurs en fonction de critères de déploiement établis par un expert réseau. Un second opérateur a ensuite la charge de créer des sous-politiques spécifiques aux plateformes et contenant l'ensemble des actions à effectuer sur une plateforme de l'infrastructure. Enfin, dans l'objectif de produire du code exécutable, nous introduisons des générateurs de code. Ces générateurs de code utilisent des modèles de variabilité décrivant la diversité des plateformes afin de produire du code exploitant au mieux les capacités matérielles disponibles.

Afin de répondre à la problématique *(P_3, séparation)*, nous proposons d'utiliser le principe de séparation des préoccupations. Nous fournissons à l'ingénieur logiciel un méta-modèle permettant d'exprimer des politiques de collecte de données indépendantes du matériel sous forme de processus métiers (*i.e.*, un enchainement d'actions à effectuer sur les données). Nous fournissons également différents méta-modèles à l'expert réseau afin qu'il puisse modéliser l'infrastructure de capteur dont il a la supervision. in de permettre un développement réutilisable, nous introduisons également des opérateurs permettant de réutiliser totalement ou partiellement une politique de collecte de données.

Les contributions présentées dans ce manuscrit ont été mises en œuvre dans un logiciel appelé DEPOSIT [1] et disponible en open source sur GitHub [2].

1.3 Fil rouge

Afin d'illustrer les contributions relatives à la définition et au déploiement d'une politique de collecte de données, nous introduisons une politique de collecte de données servant de fil rouge déroulé au cours des différents chapitres.

Les bâtiments intelligents permettent « *l'intégration de solutions actives et passives de gestion énergétique, visant à optimiser la consommation, mais également à favoriser le confort et la sécurité des utilisateurs tout en respectant les réglementations en vigueur* » [DRI14]. D'après l'institut d'études de marché Zion, le segment des bâtiments intelligents va croître annuellement de 30 % jusqu'en 2020 pour atteindre un marché de 36 milliards de dollars US.

Fil rouge : Description

L'organisme est particulièrement vulnérable face aux brusques changements de température. Ainsi, une bonne pratique veut que la différence de température entre un bâtiment climatisé et l'environnement extérieur ne soit pas supérieure à 8°C.

Au sein d'un bâtiment intelligent, on considère que chaque bureau est équipé de 2 capteurs de température ambiante (T1_x et T2_x où x est le numéro du bureau) ainsi que d'un capteur de température dans le bloc climatiseur réversible (TAC_x). Un capteur situé à l'extérieur du bâtiment (T_OUTSIDE) permet d'obtenir la température extérieure.

Une application doit être capable d'identifier des situations où, lorsque la climatisation est en fonctionnement, la différence entre la température moyenne du bureau et la température extérieure est supérieure au seuil de 8°C. Pour cela, cette

1. Data collEction POlicies for Sensing InfrasTructures
2. https://github.com/ace-design/DEPOSIT

application se connectera à un collecteur (COLLECTOR) où les données issues du réseau de capteur seront accessibles.

Cet exemple permet d'illustrer les problématiques soulevées en SEC. 1.1 :

— *(P_1, partage)* : L'infrastructure du bâtiment intelligent est composée de nombreux capteurs répartis à travers les bureaux et pièces communes. De nombreuses autres politiques seront définies pour alimenter en données d'autres applications, *cf.* CHAP. 7. L'infrastructure devra être partagée entre cet exemple fil rouge et des politiques de collecte de données tierces.

— *(P_2, adaptation)* : L'infrastructure du bâtiment intelligent est composée de plateformes de types et constructeurs différents. Le fil rouge devra être adapté automatiquement aux spécificités des différentes plateformes.

— *(P_3, séparation)* : Les préoccupations transverses liées aux plateformes de l'infrastructure sont hors du domaine métier d'un ingénieur logiciel. La définition d'une politique de collecte de données par un ingénieur logiciel doit alors uniquement faire intervenir des concepts propres à son expertise.

1.4 Plan du document

Cette thèse est structurée en huit chapitres répartis comme suit :

CHAPITRE 1 Ce chapitre introduit les problématiques liées à l'exploitation par des ingénieurs logiciels d'une infrastructure de capteurs partagée. Nous présentons la contribution apportée par cette thèse et nous définissons le fil rouge qui sera déroulé au cours des différents chapitres.

CHAPITRE 2 Ce chapitre dresse un état de l'art des différents travaux se rapprochant des thématiques abordées par cette thèse. Nous mettons en avant les contributions liées à l'expression de besoin métiers sur une infrastructure de capteurs, à leur déploiement automatique et au partage de l'infrastructure entre plusieurs applications.

CHAPITRE 3 Afin de répondre aux problématiques de cette thèse, nous avons catégorisé, au sein de l'état de l'art, un ensemble de travaux s'y rapprochant. À partir des éléments manquants de l'état de l'art, nous explicitons les motivations de cette thèse et nous définissons les objectifs des contributions présentées.

CHAPITRE 4 Ce chapitre décrit notre contribution qui vise à offrir à un ingénieur logiciel la possibilité d'exprimer des politiques de collecte de données réutilisables et indépendantes du matériel embarqué au sein d'une infrastructure de capteurs. Nous procédons à la formalisation d'une politique de collecte de données et nous définissons des opérateurs de réutilisation partielle ou totale.

CHAPITRE 5 Afin de pouvoir être mises en œuvre sur une infrastructure, les politiques de collecte de données doivent s'adapter aux spécificités des plateformes. Nous proposons dans ce chapitre une contribution visant à adapter automatiquement une politique de collecte de données à une infrastructure et de générer automatiquement les fichiers sources décrivant sa mise en œuvre. Nous présentons ainsi des modèles permettant de représenter la diversité d'une infrastructure de capteurs et des opérateurs permettant l'adaptation de la politique sur une infrastructure cible. Une fois la politique adaptée pour les plateformes de l'infrastructure cible, nous présentons un ensemble de générateurs de code traduisant la politique en code prêt à être transféré sur les différentes plateformes.

CHAPITRE 6 Une infrastructure de capteurs peut héberger une politique de collecte de données précédemment déployée. Afin de limiter la redondance des infrastructures, cette infrastructure doit pouvoir être partagée entre plusieurs applications et utilisateurs. Dans ce chapitre, nous présentons une contribution visant à partager automatiquement une infrastructure de capteurs. Nous définissons un opérateur de composition permettant la composition de politiques de collecte de données et un service de composition permettant de partager automatiquement l'infrastructure lorsque de nouvelles politiques sont identifiées.

CHAPITRE 7 Ce chapitre valide la contribution par rapport aux objectifs de cette thèse. Nous montrons également que l'approche est applicable sur des simulations d'infrastructures de bâtiments et villes intelligentes. Enfin, nous discutons des limites de notre validation.

CHAPITRE 8 Ce dernier chapitre dresse une conclusion et présente les perspectives de cette thèse.

Chapitre 2

État de l'art

Sommaire

L'objectif de ce chapitre est d'évaluer les solutions proposées au sein de la littérature permettant d'exprimer des besoins métiers sur une infrastructure de capteurs (Sec. 2.1), le déploiement automatisé d'applications sur une infrastructure de capteurs (Sec. 2.2) et le partage de l'infrastructure entre plusieurs applications (Sec. 2.3). Pour chacun de ces domaines de contribution, nous utilisons des critères de l'état de l'art afin de les comparer et de montrer qu'aucune solution ne répond à leur ensemble. Enfin, dans une dernière partie (Sec. 2.4), nous chercherons à identifier des approches intégrées permettant le développement de bout-en-bout sur une infrastructure de capteurs. La figure Fig. 2.1 présente l'organisation de notre étude de l'état de l'art :

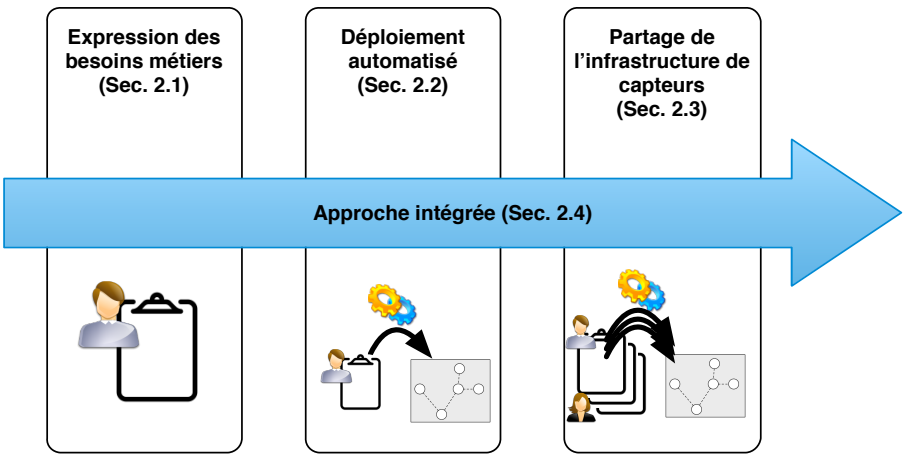


Figure 2.1 – Organisation de l'état de l'art

2.1 Expression des besoins métiers

Nous nous intéressons dans cette section à l'identification de travaux permettant d'exprimer des besoins métiers et se rapprochant de la problématique *(P_2, adaptation)* « comment une politique de collecte de données peut-elle être développée sur une infrastructure hétérogène ? ».

La programmation d'une infrastructure de capteurs demande à un utilisateur d'adopter une expertise bas-niveau pour la mise en oeuvre de ses intentions métier et gérer des problématiques transverses telles que les aspects énergétiques ou l'empreinte mémoire de son programme [Pic10]. Pour simplifier ces tâches, un ensemble d'approches permettent de masquer partiellement ou totalement ces aspects [MP11].

▶ ***Critère 1 : Niveau d'abstraction***. Nous classons les approches identifiées selon leur niveau d'abstraction, *i.e.*, leur capacité à fournir des primitives masquant les problématiques bas-niveau.

Ces approches sont, de plus, spécifiques à la programmation d'une plateforme individuelle (*par ex.*, une plateforme TMote Sky), *i.e.*, définition du comportement local de la plateforme ou à la programmation d'une infrastructure entière, *i.e.*, définition du comportement global de l'infrastructure [MP11].

▶ ***Critère 2 : Cible des abstractions***. Nous classons les approches identifiées selon leur cible (locale ou globale) au sein de l'infrastructure de capteurs.

L'hétérogénéité des infrastructures se rencontre aussi bien au niveau du matériel utilisé (plateformes d'architectures et de constructeurs différents) que sur le type de communication à l'intérieur du réseau de capteurs [Bor14].

▶ ***Critère 3 : Hétérogénéité du matériel***. Nous classons les approches par rapport à leur applicabilité sur du matériel hétérogène, *i.e.*, le code mettant en œuvre des intentions métier pourra être transféré sur des plateformes de constructeurs ou d'architectures différentes.
▶ ***Critère 4 : Transparence des communications***. De manière similaire, nous classons les approches par rapport à leur transparence vis-à-vis des technologies et protocoles de communications différents : une approche implicite permettra de s'abstraire du type de communication, et donc de leur hétérogénéité, alors qu'une approche explicite demandera de prendre en compte le type de communication.

Enfin, nous portons également attention à la notion de réutilisabilité logicielle. En effet, elle constitue une bonne pratique de développement puisqu'elle permet de réduire les temps de développement et d'encourager la production d'applications contenant des composants standardisés lorsque le besoin courant est identique à un besoin précédemment exprimé [Kru92].

▶ ***Critère 5 : Réutilisabilité d'artefacts logiciels***. Nous classons les approches par leur capacité à offrir des mécanismes de réutilisation logicielle afin d'intégrer, lors de l'expression des besoins métier, des artefacts standardisés ou développés dans un autre contexte, *par ex.*, un composant ou une bibliothèque logicielle.

2.1.1 Systèmes d'exploitation pour l'Internet des Objets

Les systèmes d'exploitation pour l'Internet des Objets ont pour objectif de fournir un ensemble d'abstractions permettant de s'affranchir des problématiques bas-niveau des plateformes de capteurs. Ils se présentent sous la forme d'un ensemble de bibliothèques offrant des fonctions utilisables par un ingénieur logiciel afin d'interagir avec les composants physiques de la plateforme. En supportant un ensemble de plateformes cibles, ils permettent également à une application d'être mise en œuvre sur différents types de matériel sans avoir à adapter le code (**critère 3**).

TinyOS [GLVB+03] est un système d'exploitation basé sur une architecture à composants et destiné à construire des applications embarquées sans-fil à faible consommation énergétique. En août 2017, TinyOS supportait 14 types de plateformes différents [1]. Le système d'exploitation est basé sur un ordonnanceur de tâches et une collection de bibliothèques logicielles permettant d'interagir avec les différents éléments de la plateforme (**critère 3**). Un ingénieur logiciel peut créer de nouveaux composants ou réutiliser des composants de la bibliothèque TinyOS afin de concevoir des applications (**critère 5**). Dans l'éventualité où l'ingénieur logiciel souhaite créer un nouveau composant TinyOS, il utilisera le langage *nesC* (network embedded systems C), dérivé de C et permettant de concevoir des applications pour des systèmes à ressources contraintes et mis en réseau (**critère 4**). Le langage *nesC* offre notamment un paradigme de développement dirigé par les évènements. La cross compilation d'un programme nesC est ensuite effectuée en quatre étapes : *(i)* les composants et interfaces composant l'application sont analysés et vérifiés sémantiquement, *(ii)* les composants sont instanciés (si génériques) et connectés les uns aux autres à partir des liaisons déclarées entre les interfaces des composants, *(iii)* les expressions constantes sont remplacées par leur va-

1. http://tinyos.stanford.edu/tinyos-wiki/index.php/Platform_Hardware

leur calculée statiquement par le compilateur (*constant folding*), et *(iv)* le code C est généré pour la plateforme considérée.

Le système d'exploitation Contiki [DGV04] réutilise le langage C pour définir des applications et supporte, en août 2017, 17 types de plateformes différentes [2]. Le système d'exploitation se présente sous la forme d'un ensemble d'abstractions matérielles permettant de masquer l'hétérogénéité des composants présents sur une plateforme de capteurs (**critère 3**). Cependant, le langage masque uniquement les interactions matérielles. Il n'offre en particulier, pas d'abstraction permettant de décrire des intentions métier. Un compilateur C natif pour l'architecture matérielle de la plateforme cible est chargé de traduire le code source en programme exécutable. De plus, en utilisant le langage C, Contiki possède les mêmes mécanismes de réutilisation que ceux apportés par le langage (**critère 5**). Un ingénieur logiciel pourra alors développer et réutiliser des bibliothèques et des macros réutilisables à travers différentes applications, en plus des bibliothèques C standards.

En s'appuyant sur le langage C++, le système d'exploitation RIOT [BHG+13] permet de réutiliser les concepts de la programmation par objet, où un objet peut représenter un sujet du monde réel (**critère 1**). En août 2017, il supporte 52 types différents de plateformes cibles [3]. Ainsi, la conception de l'application peut se faire à un plus haut niveau d'abstraction. De manière similaire à Contiki, RIOT offre un ensemble d'abstractions matérielles permettant de masquer l'hétérogénéité des composants présents sur une plateforme de capteurs (**critère 3**). Pour chacune des plateformes supportées par Contiki, un compilateur C++ spécifique à l'architecture de la plateforme permet de traduire les intentions métier en programme exécutable. Enfin, en utilisant le langage C++, RIOT bénéficie des mécanismes de réutilisation logicielle nativement présents dans le langage (**critère 5**).

FreeRTOS [DGM09], bien que n'étant pas spécifique à l'Internet des Objets, permet à un ingénieur logiciel d'exprimer des besoins métiers sur des plateformes présentes au sein d'une infrastructure de capteurs. Les applications FreeRTOS sont développées en C et traduites en langage machine grâce à un compilateur C spécifique à la plateforme cible. En utilisant le langage C, ce système d'exploitation possède les mécanismes de réutilisation logicielle (**critère 5**).

Analyse

▶ ***1. Niveau d'abstraction***. Les approches proposées augmentent le niveau d'abstraction en proposant des bibliothèques d'abstractions matérielles. Les approches TinyOS, Contiki offrent un support architectural (composant ou objets) pour la définition des concepts métiers.

▶ ***2. Cible des abstractions***. L'ensemble des approches présentées cible une plateforme individuelle au sein de l'infrastructure de capteurs. Cette contrainte impose à l'ingénieur logiciel d'exprimer autant de programmes que de plateformes dont il a besoin d'exploiter pour ses besoins métiers. Dès lors, il devra gérer les problématiques transverses liées au réseau de capteurs (*par ex.*, la synchronisation des transmissions de données entre différentes plateformes) qui sont hors de son domaine d'expertise.

▶ ***3. Hétérogénéité du matériel***. Les systèmes d'exploitation pour les infrastructures de capteurs offrent une couche d'abstraction matérielle permettant de masquer la mise en œuvre concrète du matériel sur la plateforme. Ainsi, le développement

2. http://www.contiki-os.org/hardware.html
3. https://github.com/RIOT-OS/RIOT/wiki/RIOT-Platforms

d'un programme utilise des primitives offertes par la couche d'abstraction et reste indépendant de toute considération matérielle. Cependant, nous nuançons cette indépendance par le fait que la couche d'abstraction ne supporte qu'un ensemble de matériel compatible avec le système d'exploitation considéré.

▶ ***4. Transparence des communications***. Seul TinyOS, à travers le langage nesC (network embedded systems C) permet un développement avec des communications réseau implicites grâce à la réutilisation de composants.

▶ ***5. Réutilisabilité d'artefacts logiciels***. Le développement d'applications TinyOS suit le paradigme de la programmation orientée composants. Ainsi, ces composants sont directement réutilisables au sein d'autres programmes. Contiki, RIOT et FreeRTOS, en utilisant les langages C/C++ permettent une réutilisation indirecte des programmes développés sous la forme de bibliothèques C/C++.

2.1.2 L'infrastructure comme base de données

En marge des systèmes d'exploitation déployés sur l'infrastructure de capteurs, plusieurs approches visent à considérer le réseau de capteurs comme une base de données distribuée [DGMS07]. Cette approche permet notamment d'économiser l'énergie des plateformes en privilégiant un stockage des données au plus près des capteurs plutôt que leur envoi massif vers un serveur distant [TD11].

Cougar [YG02] a pour objectif d'envoyer des tâches à réaliser par les capteurs grâce à des requêtes distribuées (**critère 2**). Ces requêtes permettent notamment d'agréger des données au sein de l'infrastructure et d'appliquer des filtres sur les résultats agrégés avant leur envoi effectif hors du réseau de capteurs. La base de données formée par l'infrastructure de capteurs est requêtable grâce à un langage proche de SQL (**critère 1**), *par ex.*, `SELECT` permet de sélectionner un ensemble de capteurs et `AVG` permet de calculer la moyenne entre plusieurs valeurs.

De manière similaire, TinyDB [MFHH05] permet d'envoyer des requêtes (**critère 1**) et d'effectuer des opérations directement au sein des plateformes de capteurs (**critère 2**). Les valeurs de capteurs appartiennent à une table *sensors* où chaque instant de mesure est traduit par une ligne et chaque attribut (*par ex.*, température) par une colonne. Le même schéma de données est imposé pour toutes les données produites par chaque plateforme dans le réseau et certaines colonnes peuvent avoir l'enregistrement `NULL` si la plateforme ne contient pas le capteur spécifié. Les auteurs reprennent la syntaxe SQL (`SELECT`, `FROM`, `WHERE`, `GROUP BY`) pour effectuer des requêtes (Lst. 2.1). Le langage de requêtes permet également d'effectuer des agrégations directement au sein des plateformes de capteurs afin de réduire le volume de données transmis à travers le réseau. Enfin, il offre également la possibilité de définir des requêtes basées sur des évènements.

Listing 2.1 – Identification des noeuds ayant une tension critique (*i.e.*, batterie presque déchargée)

```
1 SELECT nodeid, voltage
2 WHERE voltage < 1.5
3 FROM sensors
4 SAMPLE PERIOD 10 minutes
```

Analyse

▶ ***1. Niveau d'abstraction***. L'utilisation d'un langage de requêtes proche de SQL permet d'obtenir une syntaxe indépendante des plateformes et décrivant le traite-

ment à appliquer sur les données. Les différentes requêtes sont ensuite distribuées sur les différentes plateformes composant l'infrastructure.

▶ ***2. Cible des abstractions***. Ces approches visent à programmer l'ensemble de l'infrastructure à partir d'un point central. L'ingénieur logiciel n'a ainsi pas besoin d'exprimer des intentions métier pour chacune des plateformes contenues dans l'infrastructure.

▶ ***3. Hétérogénéité du matériel***. Ces approches ne supportent pas une vraie hétérogénéité des architectures de plateformes puisque l'ensemble des plateformes doit adopter le même schéma de données.

▶ ***4. Transparence des communications***. L'ensemble des communications réseau à l'intérieur de l'infrastructure de capteurs sont implicites.

▶ ***5. Réutilisabilité d'artefacts logiciels***. Nous n'avons pas identifié de support réel à la réutilisation de requêtes précédemment exprimées au sein de nouvelles requêtes.

2.1.3 Macro-programming

A la différence de programmer chaque plateforme du réseau de capteurs indépendamment des autres, les techniques de *macro-programming* visent à définir des besoins métiers à l'échelle d'une infrastructure à partir d'un point central.

Définition 2.1.1 (Macro-programming [BPRL05])**.** « Macroprogramming of sensor networks (...) means moving beyond node-centric programming and instead specifying aggregate behaviors which are then automatically translated by a compilation framework into node-level specifications. »

Kairos [GGG05] a été une des premières approches visant à programmer non plus le réseau de capteurs au niveau de chaque plateforme, mais au niveau de l'infrastructure (**critère 2**). Elle fournit un ensemble de primitives permettant d'abstraire les notions liées à la génération de code distribuée et aux interactions entre les différentes plateformes (**critères 1 et 4**). Un compilateur génère ensuite un fichier binaire qui pourra être exécuté sur les différentes plateformes. Toutefois, le code Kairos reste dépendant de l'infrastructure sur lequel il sera déployé et n'offre pas de mécanismes de réutilisation logicielle.

Le langage fonctionnel Regiment [NW04] offre la possibilité d'exprimer des applications complexes à destination des réseaux de capteurs en quelques lignes de code. Il utilise une syntaxe proche du langage Haskell afin de programmer des régions (**critère 1 et 2**), *i.e.*, un ensemble de plateformes distribuées géographiquement. Chaque région du réseau de capteurs effectue une tâche spécifique (*par ex.*, agrégation) et peut communiquer avec d'autres régions (**critère 4**). En connectant les différentes tâches les unes avec les autres, l'ingénieur logiciel réalise son application. Cependant, la mise en œuvre d'une tâche reste spécifique à l'infrastructure sur laquelle elle sera déployée. De plus, les auteurs ne font pas mention de mécanisme de réutilisation.

Le langage Atag [BPRL05] permet de définir un graphe de tâches abstraites afin de programmer le comportement du réseau de capteurs (**critères 1 et 2**). Ils introduisent les notions de *tâches abstraites* (représentant le type de processus appliqué sur les données), de *données abstraites* (représentant le type de données échangé entre les tâches) et de *canaux abstraits* (associant chaque tâche abstraite avec un type de données abstrait, **critère 4**). Chaque notion abstraite possède un certain nombre d'annotations qui préciseront leur mise en œuvre sur le réseau de capteurs (*cf.* FIG. 2.2). De plus, en

étant abstraites, les tâches symbolisent une action à exécuter et ne contiennent pas de mise en œuvre du traitement : elles sont donc indépendantes du matériel (**critère 3**). Toutefois, les auteurs ne font pas mention de mécanisme de réutilisation.

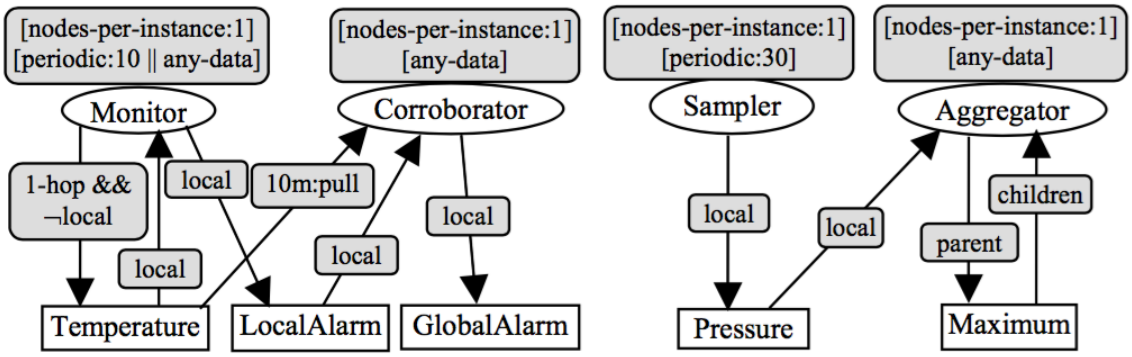


FIGURE 2.2 – Exemple d'application Atag (tâches abstraites sur fond blanc, canaux abstraits fléchés, annotations sur fond gris) [BPRL05]

Pleiades [KGMG07] vise à étendre le langage nesC [GLVB+03] afin de permettre l'expression d'applications TinyOS au niveau de l'infrastructure (**critère 2**). Ce programme *central* aura accès aux différentes plateformes du réseau, *par ex.*, l'instruction de boucle *cfor* permet d'itérer sur toutes les plateformes du réseau ou sur tous les voisins d'une plateforme particulière. Le compilateur Pleiades traduit ensuite ce code *central* en code spécifique aux plateformes. Par la réutilisation du langage nesC, Pleiades bénéficie des principes de réutilisabilité du système d'exploitation TinyOS (**critère 5**).

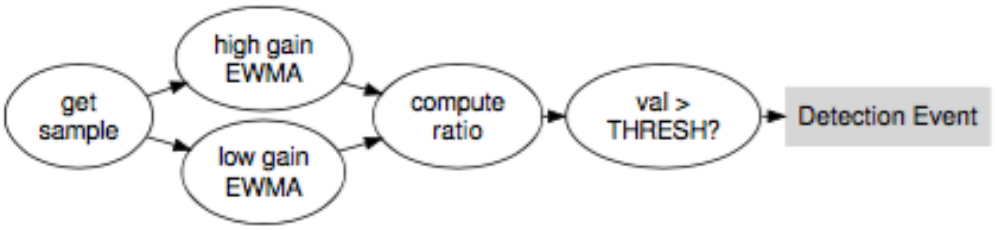


FIGURE 2.3 – Exemple de flux de données permettant de détecter un séisme (extrait de [MMW08])

Au sein de l'approche Flask [MMW08], les auteurs constatent que le code nesC implique pour un ingénieur logiciel d'écrire du code bas-niveau dirigé par les évènements, et ainsi, de comprendre l'architecture matérielle de la plateforme sur laquelle ils souhaitent déployer leur code. Afin de programmer de manière centrale (**critère 2**) et à plus haut niveau d'abstraction, ils introduisent le langage Flask (**critère 1**). Flask est une extension du langage Haskell permettant d'exprimer des programmes sous la forme de flux de données (*cf.* FIG. 2.3). Les programmes écrits en Haskell sont ensuite compilés en Red puis en nesC afin d'être exécutés sur les différentes plateformes. Ce langage a pour avantage d'abstraire les différentes communications réseau à travers les flux de données (**critère 4**) et de permettre de réutiliser les tâches entre différents

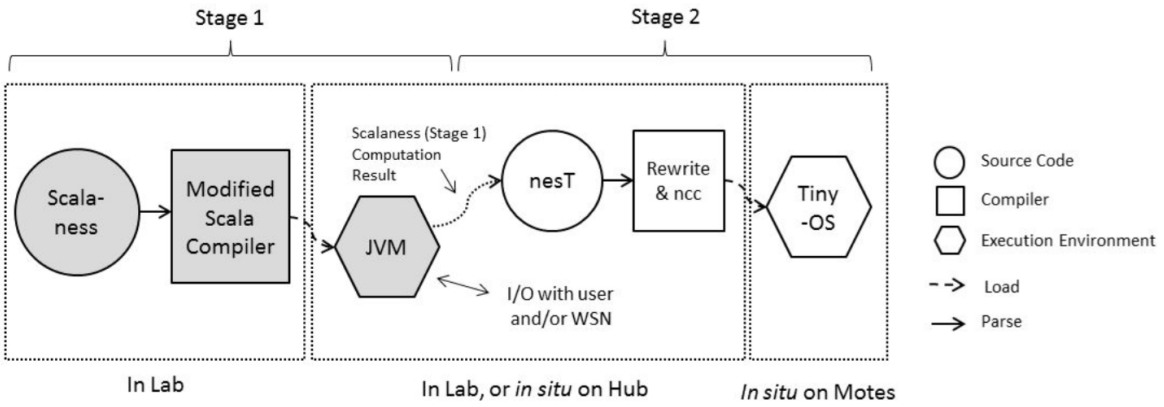


FIGURE 2.4 – Compilation et exécution Scalaness/nesT (extrait de [CSSW13])

programmes . Toutefois, la mise en œuvre d'une tâche reste spécifique à l'infrastructure sur laquelle elle sera déployée. De plus, les auteurs ne discutent pas de mécanisme permettant la réutilisation des programmes Flask précédemment exprimés.

L'approche Scanaless/nesT [CSSW13] vise à offrir une technique de macro-programmation avec sureté du typage *i.e.*, le compilateur vérifie les types de données. Dans un premier temps, un ingénieur logiciel utilise le langage Scalaness (une variante de Scala) afin d'exprimer à haut niveau d'abstraction le programme qu'il souhaite déployer sur le réseau de capteurs (**critères 1 et 2**). Dans un second temps, le programme Scalaness est traduit automatiquement en nesT (variante typée du langage nesC) avant d'être projeté, sous forme d'images TinyOS, sur les différentes plateformes composant le réseau de capteurs. Le code exprimé est indépendant du matériel : l'adaptation aux caractéristiques des plateformes sera gérée par les différents compilateurs TinyOS. Il est également intéressant de souligner que les auteurs proposent la réutilisation de composants nesT à travers d'autres programmes. Concernant les aspects réseau, l'approche laisse la responsabilité aux ingénieurs logiciels de gérer les communications réseaux entre les différentes plateformes. Enfin, en ne produisant que des images TinyOS, la réutilisabilité de l'approche est limitée aux plateformes compatibles avec le système d'exploitation TinyOS.

Analyse

▶ ***1. Niveau d'abstraction***. À l'exception de Pleides, ces approches n'abstraient pas les opérations liées au matériel embarqué sur les plateformes, obligeant ainsi un ingénieur logiciel à exprimer la mise en œuvre de chacune de ses intentions métier en fonction des caractéristiques matérielles de chaque plateforme.

▶ ***2. Cible des abstractions***. Les techniques de *macro-programming* permettent de programmer un ensemble de plateformes de capteurs à partir d'un point central. Elles contribuent ainsi à réduire le nombre de fichiers sources décrivant les intentions métier. Un compilateur a ensuite la responsabilité de produire les fichiers sources prêts à être transférés sur les différentes plateformes.

▶ ***3. Hétérogénéité du matériel***. Ces techniques imposent une infrastructure homogène puisqu'elles reposent sur un unique compilateur générant les fichiers sources à destination des plateformes impliquées dans la mise en œuvre des besoins métiers.

▶ ***4. Transparence des communications***. Les langages de macro-programming permettent également d'abstraire la complexité liée à la topologie et aux communications réseaux.

▶ ***5. Réutilisabilité d'artefacts logiciels***. Nous n'avons pas identifié de mécanisme de réutilisation au sein de l'approche Kairos. L'approche Regiment permet de réutiliser la notion de tâche, mais uniquement au sein de la même infrastructure de capteurs. L'approche Pleides, en étant basée sur TinyOS, bénéficie de l'approche de réutilisation des composants TinyOS. Les approches Flask et Scalaness/nesT permettent de réutiliser la notion de programmes entre différentes plateformes supportant le système d'exploitation TinyOS. Enfin, bien que l'approche Atag repose sur des tâches indépendantes du matériel, et donc réutilisables au sein de modules à destination d'infrastructures différentes, il n'est pas fait mention de mécanisme de réutilisation logicielle.

2.1.4 Langages spécifiques au domaine

Dans le cadre du développement de systèmes embarqués, l'utilisation d'un langage spécifique au domaine permet d'offrir un plus haut niveau d'abstraction par rapport à l'utilisation d'un langage générique [MEP17]. Par sa nature, le langage spécifique au domaine contient uniquement des instructions relatives au domaine dans lequel il s'applique. Ainsi, la vérification de la mise en œuvre du programme par rapport aux spécifications métiers sera proche du langage naturel de l'expert métier et pourra être effectuée à un cout de maintenance réduit.

Définition 2.1.2 (Langages spécifiques au domaine)**.** « A Domain Specific Language (DSL) is a concise, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are tailored to the stakeholders who specify that particular concern. » [Fow10]

À travers l'approche SALT [CGDLR13], les auteurs font le constat que le manque de standards et de plateformes de développement universelles pose problème au développement des interactions entre ces différentes plateformes. En effet, chaque plateforme apporte son paradigme faisant que les applications sont fortement liées au matériel. Ils proposent une solution où une application est bâtie autour d'une *chorégraphie*, *i.e.*, un ensemble d'éléments exécutés indépendamment sur des plateformes différentes, mais dont la collaboration (grâce à des échanges de messages) permet la réalisation de l'application souhaitée (**critère 2**). Les *chorégraphies* sont construites à partir de transducteurs à états finis, *i.e.*, un automate à états finis avec sorties. Afin de définir ces transducteurs, les auteurs proposent le langage spécifique au domaine de l'interaction entre plateformes, `SALT` (**critère 1**). Ce langage contient uniquement des instructions permettant de définir des états, des actions à effectuer au sein des états, des transitions et des messages. Le transducteur représentant la chorégraphie est ensuite traduit, grâce à un *framework* issu de leurs précédents travaux, en un exécutable prêt à être déployé sur des plateformes de capteurs. Le langage SALT est accompagné d'un éditeur graphique permettant d'exprimer les chorégraphies. Les préoccupations liées au réseau sont automatiquement gérées lors de l'échange des messages de synchronisation (**critère 4**). De plus, ce langage permet la réutilisation logicielle puisque chacun des éléments d'une chorégraphie est réutilisable au sein d'une autre chorégraphie. Toutefois, l'approche ne présente pas de mécanisme de réutilisation. Enfin, les primitives constituant les différentes actions d'une chorégraphie sont indépendantes des plateformes et peuvent être déployées sur différentes architectures (**critère 3**).

À travers l'approche Sminco [Sad07], les auteurs soulignent également que le développement d'applications pour les réseaux de capteurs sans-fils est une tâche complexe qui implique de longs développements sans un support possible des experts métiers. Pour impliquer les experts métiers dans le développement, ils suggèrent l'utilisation de langages spécifiques au domaine ayant une syntaxe simple et permettant l'expression de simulations (**critère 1**). Ils présentent un langage spécifique au domaine orienté flux de données. La définition des flux de données repose sur un modèle propre au métier visé décrivant des actions à valeur ajoutée à effectuer sur les données. Ce langage est ensuite traduit automatiquement en Scheme, puis en code C utilisable pour les plateformes cibles. L'adaptation du programme aux ressources des différentes plateformes est gérée automatiquement lors de la phase de compilation (**critère 3**). En revanche, les auteurs ne donnent pas d'information quant à la cible de déploiement (plateforme ou infrastructure) et à la réutilisabilité des flux de données au sein d'autres applications.

Analyse

▶ ***1. Niveau d'abstraction***. En masquant les problématiques liées aux infrastructures de capteurs, les langages spécifiques au domaine permettent à des ingénieurs logiciels d'exprimer à haut niveau d'abstraction des intentions métier. Le code produit est ainsi plus facilement lisible (en étant proche d'un domaine métier, ils sont facilement compréhensibles par des experts du métier) et maintenable.

▶ ***2. Cible des abstractions***. L'approche SALT vise la programmation d'une infrastructure entière grâce aux orchestrations. L'approche Sminco a été validée par le déploiement de flux de données sur une plateforme, mais les auteurs ne donnent pas d'informations quant à son application pour programmer une infrastructure entière.

▶ ***3. Hétérogénéité du matériel***. Les approches présentées reposent sur des primitives ou sur un modèle indépendant du matériel sous-jacent. L'adaptation à l'hétérogénéité du matériel est gérée automatiquement lors de la compilation du programme en code transférable sur les plateformes de l'infrastructure.

▶ ***4. Transparence des communications***. Les problématiques liées aux communications réseau sont gérées automatiquement lors de la compilation.

▶ ***5. Réutilisabilité d'artefacts logiciels***. Les éléments de chorégraphie de SALT sont réutilisables sur du matériel différent. L'approche Sminco ne donne pas d'éléments quant à la réutilisabilité des tâches décrivant les flux de données au sein d'autres applications.

2.1.5 Approches par composants

Les approches par composants visent à augmenter le niveau d'abstraction en offrant la possibilité de concevoir des traitements de données en utilisant une approche modulaire et réutilisable (**critère 5**).

L'approche REMORA [TLA+10] vise à offrir un paradigme de programmation par composants à destination des plateformes de capteurs (**critère 2**). L'approche présentée repose sur un modèle de composants où chaque composant repose sur une description et une mise en œuvre. La description du composant est conforme à SCA (*Service Component Architecture*) et spécifie les interfaces sous forme de services (opérations apportées), références (opérations requises), évènements produits et consommés et propriétés. La mise en œuvre est effectuée indépendamment du système d'exploitation cible en utilisant un langage proche de C (*C-Like*) et repose sur les évènements (produits par l'application ou le système d'exploitation). La composition de ces composants est

effectuée au sein d'un fichier XML (**critère 1**) référençant l'ensemble des composants de l'application ainsi que les liens entre les références et services. Les artefacts décrivant les composants, leur mise en œuvre *C-Like* ainsi que leur composition sont ensuite transformés en langage C prêt à être déployé sur une plateforme de capteurs. Afin de gérer l'hétérogénéité des plateformes (**critère 3**), les auteurs introduisent une couche d'abstraction (*OS Abstraction Layer*) masquant les préoccupations propres à chaque plateforme. Cette couche d'abstraction peut être développée à destination de n'importe quel système d'exploitation supportant le langage C.

L'approche COSMOS [CRS07] apporte des éléments d'architecture réutilisables permettant la construction d'un intergiciel spécialisé au contexte local et piloté par une chaine de remontée d'informations intelligente. Cette chaine de remontée est constituée de trois couches : une couche de collection où sont récoltées des informations brutes et de bas niveau (évènements systèmes, capteurs et périphériques connectés – **critère 2**), une couche intermédiaire où les données sont traitées en vue de leur donner une information à valeur ajoutée pour piloter une couche de niveau supérieur, où des décisions métier sont effectuées. Ces couches sont composées de nœuds de contexte définis sous forme de composants FRACTAL [BCL+06]. La construction de cette chaine de remontée est effectuée de manière déclarative grâce à la définition d'un langage de description d'architecture FractalADL [LOQS07] ou d'annotations Fraclet [RM09] (**critère 1**). Les auteurs proposent également un langage spécifique au domaine afin de faciliter la manipulation du langage de description [RCS08]. Des mécanismes de réutilisation et de partage des composants sont directement intégrés au sein du langage de description d'architecture sous la forme de quatre patrons de conceptions (composite, patron de méthodes, poids-mouche et singleton). Au niveau de la couche de collection, l'hétérogénéité du matériel (**critère 3**) est gérée par le canevas logiciel SAJE [CGLSM03] permettant de représenter sous forme d'objets les ressources systèmes physiques (*par ex.*, batterie ou mémoire vive) que logiques (*par ex.*, fichier ou socket réseau). Afin de permettre la composition de composants, COSMOS supporte la définition explicite de mode pull (observations) et push (notifications) pour le transfert d'informations. Ces technologies permettent indirectement de rentre transparent les communications inter-composants puisque la logique de transmission des messages leur est déléguée (**critère 4**).

Analyse

▶ ***1. Niveau d'abstraction***. Les approches apportent un paradigme de programmation par composants afin d'augmenter le niveau d'abstraction : REMORA permet la définition indépendante du système d'exploitation de composants et la composition de ces derniers au sein d'un fichier d'associations tandis que COSMOS offre un support architectural sous forme de composants FRACTAL à la définition de contexte local piloté par une chaine de remontée d'informations.

▶ ***2. Cible des abstractions***. L'approche REMORA est destinée à programmer individuellement une plateforme de l'infrastructure de capteurs. L'approche COSMOS permet de définir un intergiciel spécialisé au contexte et piloté par une remontée d'informations intelligente au niveau de plateformes de type PCs (Windows, GNU/Linux) ou PDA mais n'est pas destinée à générer du code pour des plateformes de capteurs.

▶ ***3. Hétérogénéité du matériel***. REMORA permet la gestion de l'hétérogénéité grâce à une couche d'abstraction au dessus du système d'exploitation de la plateforme cible. COSMOS gère l'hétérogénéité des cibles de déploiement à travers le

canevas logiciel SAJE (gestion des plateformes Windows XP, 2000, Mobile 2003 et GNU/Linux au moment de la publication de l'approche).

▶ ***4. Transparence des communications***. COSMOS permet de gérer indirectement les problématiques liées aux communications réseaux en permettant aux composants d'être configurés pour utiliser les protocoles push et pull.

▶ ***5. Réutilisabilité d'artefacts logiciels***. Ces approches permettent, par construction, la réutilisation d'artefacts logiciels en permettant aux composants d'être partagés. COSMOS apporte de plus un mécanisme de tissage d'aspects à travers la mise en œuvre FRACTAL.

2.1.6 Approches pour le *crowdsourcing*

Les approches présentées précédemment visent à exprimer des besoins métiers sur des réseaux de capteurs fixes. Avec la prolifération des téléphones intelligents (*Smartphones*), des approches visent à assimiler ces dispositifs à des plateformes de capteurs mobiles [YJS12]. Ces plateformes mobiles sont particulièrement utilisées au sein du paradigme du *crowdsourcing* visant à cibler des possesseurs de smartphone afin de permettre l'exécution de tâches de collecte de données à grande échelle qui seraient coûteuses et consommatrices de ressources avec les méthodes traditionnelles [CKLZY12].

L'approche Pogo [BL12] permet l'expression de collecte de données à grande échelle sur des smartphones Android, grâce au langage Javascript. Ces politiques sont ensuite envoyées sur les smartphones d'utilisateurs participants (**critère 2**) et ayant installés un client local pouvant interpréter le code Javascript. Les auteurs justifient le choix d'utiliser un langage générique par rapport à un langage spécifique au domaine (*cf.* Def.2.1.2) afin de supporter des applications variées. De plus, ce langage est largement utilisé par les développeurs d'applications. Afin de masquer l'hétérogénéité caractéristique des smartphones Android et de ne pas écrire du code spécifique à une plateforme donnée, ils proposent une couche d'abstraction matérielle contenant 11 méthodes (**critère 3**). Ainsi, les développeurs n'ont pas besoin de connaître les détails du fonctionnement du téléphone ou du système d'exploitation Android. De plus, les abstractions liées au réseaux permettent de gérer automatiquement les changements de la connexion de données (WiFi ou cellulaire) inhérentes à la mobilité des smartphones (**critère 4**).

L'approche APISENSE [HRS13] permet de déployer des tâches de collecte de données à large échelle en utilisant les smartphones Android et iOS d'utilisateurs participants. A la différence de Pogo, APISENSE propose un « marché aux politiques » où les utilisateurs peuvent, à partir d'une description des intentions du développeur, décider d'exécuter ou non les tâches de collecte leurs téléphones. L'utilisation d'un « marché aux collectes » permet de cibler, à partir d'un point central, plusieurs smartphones à condition que les utilisateurs donnent leur accord explicite (**critère 2**). De manière similaire à Pogo, les intentions métier décrites au sein des tâches de collecte utilisent le langage Javascript. Un ensemble de « *stings* » (23 *stings* dans la version APISENSE 1.10) décrit les capteurs du téléphone et agit comme une abstraction matérielle masquant l'hétérogénéité du matériel. Concernant les caractéristiques réseau, la gestion de l'utilisation de la connexion WiFi ou de la connexion cellulaire est laissée au choix du développeur. Ainsi, ce dernier peut préférer utiliser un type particulier de connexion pour la réalisation des tâches de collecte de données, *par ex.*, une connexion WiFi est souvent préférable, car gratuite alors qu'une connexion de données est décomptée du forfait mobile.

Analyse

▶ ***1. Niveau d'abstraction***. Les approches visant à effectuer des tâches de collecte de données à grande échelle utilisent un langage générique, *par ex.*, Javascript, plutôt qu'un langage spécifique au domaine afin de supporter une grande variété d'applications.

▶ ***2. Cible des abstractions***. Les tâches de collecte de données sont définies au niveau d'un téléphone intelligent puis envoyées à plusieurs téléphones d'utilisateurs volontaires ou mises à disposition sur une place de marché. Ainsi, elles ont la finalité de cibler une flotte de téléphones (équivalent à une infrastructure de capteurs).

▶ ***3. Hétérogénéité du matériel***. La grande variété de téléphones intelligents entraine *de facto* une grande hétérogénéité du matériel. Des couches d'abstractions logicielles, *par ex.*, les stings APISENSE, permettent de supporter cette hétérogénéité.

▶ ***4. Transparence des communications***. L'approche Pago permet d'utiliser les connexions de données mobiles et WiFi de manière transparente. L'approche APISENSE laisse la liberté au développeur de choisir la connexion à utiliser.

▶ ***5. Réutilisabilité d'artefacts logiciels***. Les approches ne présentent pas de mécanisme de réutilisation d'une collecte de données ou d'éléments d'une collecte de données au sein d'une nouvelle campagne.

Conclusion

Dans cette section nous avons présenté un état de l'art permettant d'exprimer l'expression de besoins métiers sur une infrastructure de capteurs. Nous comparons au sein du tableau Tab. 2.1, les différentes approches de cette section par rapport aux critères identifiés.

Étant donné le nombre important de dispositifs pouvant être utilisés au sein d'une politique de collecte de données, *par ex.*, scénarios de villes ou bâtiments intelligents, une approche permettant l'expression d'intentions sur l'infrastructure à partir d'un point central est nécessaire afin de ne pas avoir à identifier et programmer manuellement l'ensemble des plateformes impliquées dans la réalisation des besoins métiers. De plus, les ingénieurs logiciels n'ayant pas de connaissances préalables sur les infrastructures de capteurs, l'approche proposée doit masquer les problématiques bas niveau propres aux systèmes en offrant une approche à haut niveau d'abstraction et simplifiant les communications réseaux. Enfin, face à la diversification des plateformes présentes sur le marché et à la multiplication d'applications utilisant des infrastructures de capteurs, l'approche proposée doit être applicable sur un matériel hétérogène et offrir des mécanismes de réutilisation logicielle afin de réécrire un minimum de code lorsqu'un besoin courant est similaire à un besoin précédemment exprimé.

Nous pouvons remarquer que les langages spécifiques au domaine et les approches de *crowdsourcing* permettent l'expression à haut niveau d'abstraction des besoins métier : les premiers grâce à un langage contenant uniquement des instructions relatives au domaine ciblé, les seconds grâce à une couche d'abstraction matérielle. Nous préférerons utiliser les langages spécifiques au domaine, car ils fournissent un langage ne contenant que des instructions pertinentes au domaine ciblé. Dans cette classe de langage, les approches proposées n'apportent pas de mécanismes de réutilisation logicielle.

Ainsi, un premier point d'action de cette thèse est de fournir une approche d'expression de besoins métiers à haut niveau d'abstraction à destination des ingénieurs logiciels et supportant une infrastructure de capteurs hétérogènes tout en offrant des mécanismes de réutilisation.

TABLEAU 2.1 – Comparatif des approches d'expression de besoins métier

	Approche	Niveau d'abstraction	Cibles des abstractions	Hétérogénéité du matériel	Transparence des communications	Réutilisabilité d'artefacts logiciels
Systèmes d'exploitation	TinyOS	Intermédiaire (langage nesC)	Plateforme individuelle	Partiellement (uniquement entre plateformes compatibles TinyOS)	Oui	Oui (composant TinyOS)
	Contiki	Bas (langage C)	Plateforme individuelle	Partiellement~ (uniquement entre plateformes compatibles Contiki)	Non	Partiellement (réutilisation de bibliothèques C)
	RIOT	Intermédiaire (langage C++)	Plateforme individuelle	Partiellement (uniquement entre plateformes compatibles RIOT)	Non	Partiellement (réutilisation de bibliothèques C++)
	FreeRTOS	Bas (langage C)	Plateforme individuelle	~ (uniquement entre les plateformes FreeRTOS)	Non	Partiellement (réutilisation de bibliothèques C)
B.D.D.	Cougar	Intermédiaire (requêtes)	Infrastructure	Non	Oui	Non
	TinyDB	Intermédiaire (requêtes)	Infrastructure	Non	Oui	Non
Méta-Programmation	Kairos	Intermédiaire (langage procédural)	Infrastructure	Non	Oui	Non
	Regiment	Intermédiaire (langage Haskell)	Infrastructure	Non	Oui	Non
	Atag	Intermédiaire (langage Atag)	Infrastructure	Oui	Oui	Non
	Pleides	Intermédiaire (langage nesC)	Infrastructure	Partiellement (uniquement entre plateformes compatibles TinyOS)	Non	Oui (composant TinyOS)
	Flask	Intermédiaire (langage Haskell)	Infrastructure	Partiellement (uniquement entre plateformes compatibles TinyOS)	Oui	Non
	Scalaness/nesT	Intermédiaire (langage Scala)	Infrastructure	Partiellement (uniquement entre plateformes compatibles TinyOS)	Non	Oui (modules nesT)
Langages spécifiques au domaine	SALT	Haut (définition de chorégraphies)	Infrastructure	Oui	Oui	Non
	Sminco	Haut (définition de flux de données)	?	Oui	?	Non
Approches composants	REMORA	Intermédiaire (composants - C-like)	Plateforme individuelle	Oui (couche d'abstraction OS)	Indirectement	Oui (composant)
	COSMOS	Intermédiaire (composants - Java)	Plateforme individuelle (PC/PDA)	Oui (réutilisation de SAJE)	Indirectement (Pull/Push)	Oui (composant FRACTAL)
Approches *crowdsourcing*	ApiSense	Haut (couche d'abstraction matérielle)	Infrastructure	Oui	Non	Non
	Pogo	Haut (couche d'abstraction matérielle)	Infrastructure	Oui	Oui	Non

2.2 Déploiement d'une application sur une infrastructure de capteurs

En 2011, Pilloni et al. constatent que le déploiement de scénarios sur des infrastructures à grande échelle, *par ex.*, des bâtiments ou villes intelligentes, peut impliquer plusieurs centaines de plateformes différentes [PA11]. La difficulté réside alors dans l'identification des plateformes sur lesquelles les actions devront être déployées tout en réduisant l'impact du déploiement par rapport à des critères non fonctionnels, *par ex.*, minimisation de l'impact sur les batteries des plateformes de capteurs. Ce problème se rencontre également dans le contexte du *cloud computing* à travers les étapes de planification, *planner*, *i.e.*, identification de la séquence d'actions de déploiement à effectuer sur une infrastructure *cloud* et de génération de plan de déploiement, *i.e.*, mise en œuvre des actions de déploiement [CDCE+13, DCLT+14]. Dans le cadre du *cloud computing* et après le déploiement des applications, des services de *monitoring* permettent de mesurer en continu des paramètres liés à l'infrastructure ou aux applications, *par ex.*, consommation électrique ou qualité de service courante, afin d'améliorer l'opération des systèmes et des applications [KEW+10]. Dans le contexte des infrastructures de capteurs, le suivi de la consommation énergétique est une préoccupation principale, car ces infrastructures contiennent des plateformes fonctionnant sur batteries [KW07].

À partir de ce constat, nous identifierons dans une première partie des travaux permettant le déploiement d'applications sur une infrastructure de capteurs et nous utiliserons les critères suivants afin de comparer les approches :

▶ ***Critère 6 : Identification des plateformes cibles***. Nous classons les approches par rapport à leur capacité à identifier automatiquement l'ensemble des plateformes cibles pour le déploiement d'une application.

▶ ***Critère 7 : Utilisation d'une fonction d'optimisation de déploiement***. Nous classons les approches par rapport à leur capacité à adapter le déploiement par rapport à une fonction d'optimisation, *par ex.*, une fonction de minimisation de l'impact énergétique du déploiement.

▶ ***Critère 8 : Suivi des applications déployées et/ou de l'infrastructure***. Nous classons les approches par leur capacité à offrir un service de *monitoring* au niveau des applications déployées et/ou de l'infrastructure de capteurs.

Dans une seconde partie, nous présenterons des travaux liés à l'estimation ou à l'obtention de la consommation énergétique d'une application déployée sur une infrastructure distribuée.

2.2.1 Automatisation du déploiement

L'approche DIANE [VSID15] permet de générer une topologie de déploiement pour une application IoT en fonction de la disponibilité de l'infrastructure sous-jacente. Une application IoT est décrite en termes d'unités techniques (TU), d'unités de déploiement (DU) et d'instances de déploiement (DI). Les TUs permettent de décrire, grâce au langage JSON-LD, les différents composants logiciels de l'application. Pour chaque TU, un ou plusieurs DUs décrivent comment il doit être déployé sur l'infrastructure physique grâce à une liste de contraintes matérielles et logicielles (*par ex.*, un composant logiciel Java doit être déployé sur un système possédant une machine virtuelle Java 1.8). Les DIs représentent enfin le déploiement concret d'un TU sur un DU. La résolution des contraintes pour associer un TU à un DU est effectuée grâce à un générateur de déploiement, contenant lui-même un solveur de contraintes identifiant les plateformes

cibles (**critère 6**). DIANE contient également un registre de déploiement ayant pour responsabilité de maintenir à jour une structure de données associant un TU sur un DU. Il est ainsi possible de savoir, à tout instant t, quels sont DUs libres ou impliqués dans une application. Toutefois, cette approche ne présente pas un mécanisme de discrimination de la plateforme résultant du générateur de déploiement en fonction de critères d'optimisation du déploiement. Enfin, nous notons la présence d'un *profiler* au sein des passerelles vers les plateformes IoT permettant de surveiller le statut de l'infrastructure (**critère 8**).

Au sein de l'approche définie par Patel et al. [PC15], le déploiement des applications IoT est décrit à travers des préoccupations fonctionnelles, des préoccupations de déploiement et des préoccupations de plateformes. Les préoccupations fonctionnelles sont associées (grâce à un opérateur d'association *Map*) avec les préoccupations de déploiement afin d'obtenir, une cible de déploiement pour chaque préoccupation fonctionnelle (**critère 6**). L'ensemble des cibles de déploiement, exprimées sous la forme de *mapping files*, *i.e.*, décrivant chacun une association d'un service à une plateforme de l'infrastructure, sont ensuite associées aux préoccupations des plateformes (grâce à un opérateur de liaison *Linker*) afin d'obtenir du code prêt à mis en œuvre sur l'infrastructure. L'approche ne présente toutefois pas un moyen d'intégrer une fonction d'optimisation du déploiement en fonction de critères fonctionnels de l'infrastructure de capteurs. De plus, elle ne présente pas de mécanismes permettant le suivi des applications déployées ou de l'état de l'infrastructure.

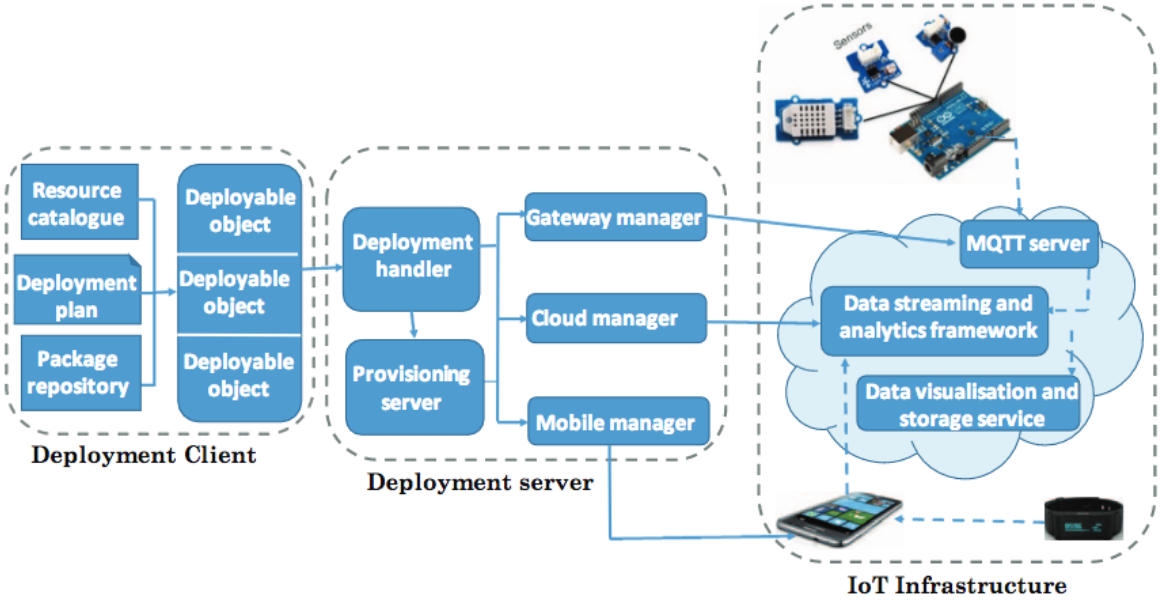


FIGURE 2.5 – Architecture haut niveau du système de déploiement (extrait de [MFT17])

L'approche présentée par Mohamed et al. [MFT17], permet le déploiement simultané de plusieurs applications IoT sur différentes plateformes. Le déploiement des applications est effectué grâce à un client et à un serveur (*cf.* FIG. 2.5). Le client de déploiement est le point d'entrée du système et a pour responsabilité de générer des objets déployables grâce à différents artefacts (catalogue de ressources, plan de déploiement, dépôt de paquets). Le client de déploiement contient un *optimiser* chargé de générer le plan de déploiement. Le plan de déploiement contient l'ensemble des opérations requises et spécifie sur quelle plateforme chaque opération devrait être exécutée.

Les auteurs précisent que cet *optimiser* devra, au sein de travaux futurs, permettre l'expression d'éléments non-fonctionnels pour guider le déploiement (**critère 7** planifié). Au sein du serveur de déploiement, les objets déployables sont réceptionnés par un gestionnaire de déploiement qui associera, grâce à un serveur de provision, la plateforme cible (**critère 6**). Différents *managers* ont pour responsabilité de déployer et de gérer les applications sur les plateformes cibles. Le catalogue de ressources permet ici de gérer l'hétérogénéité du matériel. Enfin, ils mentionnent la présence d'un mécanisme de surveillance en temps réel de l'infrastructure permettant de mettre à jour le catalogue de ressources lorsque l'état d'une plateforme est modifié, *par ex.*, la plateforme devient active/inactive (**critère 8**).

2.2.2 Suivi de la consommation énergétique

L'informatique durable (*green computing*) est une notion visant principalement à réduire l'empreinte énergétique des systèmes d'information [Hoo08].

Orgerie et al. présentent un article de revue de techniques permettant d'augmenter l'efficacité énergétique de systèmes distribués à large échelle [OAL14]. Les solutions identifiées au sein de cet article permettent d'optimiser la consommation énergétique à différents niveaux, du nœud de calcul individuel à une infrastructure entière. Toutefois, au niveau de l'infrastructure, l'hétérogénéité du matériel et la diversité des applications déployées rendent cette tâche complexe.

Afin de répondre à ce problème, l'approche WattsKit [CFRS17] permet de créer des sondes agissant comme des « multi-mètres logiciels » et remontant la consommation électrique d'un service à partir de la charge processeur (37 % de la consommation électrique d'un serveur traditionnel [OAL14]). Pour une application déployée sur plusieurs nœuds de calcul sous forme de services, un agrégateur de plus haut niveau somme les consommations électriques de chacun de ces services pour obtenir la consommation électrique globale de l'application.

Les auteurs établissent le profil de consommation électrique d'un processeur grâce à un ensemble d'applications *workload* permettant de testant les différentes fonctionnalités d'un processeur, *par ex.*, hyper-threading ou TurboBoost. Ces *workloads* permettent de déclencher des évènements HPC (*Hardware Performance Counters*) caractérisant le type d'opération effectué par le processeur. Simultanément, un multi-mètre physique permet de mesurer la variation de la consommation électrique induite par chaque évènement. Une régression de type *robust ridge* entre un ensemble d'évènements HPC pertinents (*i.e.*, ayant une corrélation significative avec la consommation électrique mesurée) et leur consommation électrique permet de construire un modèle de consommation spécifique à l'architecture du processeur considéré.

Ce modèle de consommation est ensuite utilisé sous forme de module au sein de `PowerModule`. La formule associe des évènements à une estimation de consommation énergétique. Un ensemble de capteur collecte les évènements du processeur et les transmet à la formule de calcul. Les valeurs de consommations obtenues sont ensuite transmises à un système externe (*par ex.*, une base de données time-series comme InfluxDB). La figure Fig. 2.6 et le code Lst. 2.2 présentent respectivement l'architecture du multi-mètre logiciel formé et une mise en œuvre Scala chargée de surveiller, toutes les secondes pendant une heure, la consommation du service Apache2 et de reporter les valeurs mesurées dans une console logicielle.

Listing 2.2 – Mise en œuvre d'un multi-mètre logiciel permettant d'obtenir la consommation électrique du service `apache2`

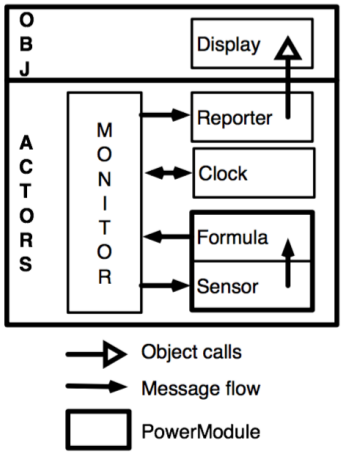


FIGURE 2.6 – Architecture d'un multi-mètre logiciel défini par Wattskit (extrait de [CFRS17])

```
1  object SDPowerMeter extends App {
2      val sdpm = PM.loadModule(LibpfmCoreProcessModule()) //
3      val display = val console = new ConsoleDisplay
4
5      val apachePower = sdpm.monitor("apache2")
6                            .every(1.seconds)
7                            .to(display)
8
9      apachePower(1.hour)
10     apachePower()
11     sdpm.shutdown()
12 }
```

La consommation électrique est la principale préoccupation lors du développement d'applications à destination de réseaux de capteurs sans-fil. Selon la mise en œuvre d'une application, l'autonomie de la batterie peut varier de quelques semaines à plusieurs années [DFR+13].

Afin de prédire l'autonomie des batteries par rapport à une application, Dâmaso et al. [DFR+13] proposent une approche évaluant la consommation des réseaux de capteurs sans-fil grâce à des modèles de simulation et des outils permettant d'automatiser l'approche. Ils ciblent les plateformes fonctionnant sur le système d'exploitation TinyOS (*cf.* SEC. 2.1.1). Afin d'évaluer la consommation électrique des applications pour réseaux de capteurs sans-fil, ils définissent trois activités : *(i)* création de modèles de consommation basiques à partir des valeurs de consommation de chaque instruction et opérateur nesC (estimation du coût énergétique à l'échelle de la ligne de code), *(ii)* création de modèles de fonctions en associant et connectant plusieurs modèles basiques (estimation du coût énergétique à l'échelle de la fonction) et, *(iii)* création du modèle d'application en associant et connectant plusieurs modèles de fonction (estimation du coût énergétique à l'échelle de l'application). Un traducteur permet à partir d'un code nesC de générer automatiquement les modèles fonctionnels et d'application. Un service web externe est ensuite chargé de procéder à la simulation.

Ces deux approches permettent d'obtenir la consommation énergétique d'un programme déployé sur une infrastructure distribuée (en temps réel pour une infrastructure *cloud computing* grâce à WattsKit, et par simulation pour une infrastructure de réseau de capteur sans-fil grâce à l'approche définie par Dâmaso et al.). Cependant, elles ne permettent pas d'agir directement sur la consommation énergétique. Au niveau des infrastructures de capteurs sans-fil, les communications réseaux constituent la principale source de dépense énergétique. De nombreux travaux ont ainsi permis de définir

des protocoles d'accès au réseau (Media Access Control protocols, ou MAC protocols) réduisant la consommation énergétique grâce à des techniques de réveil adaptatif ou de prévention des collisions [MV05, YHE02, VDL03, ROGLA06]. Toutefois, ces protocoles ne revoient pas d'information contextuelle à un expert réseau, pouvant guider d'éventuelles opérations de maintenance, *par ex.*, la connaissance du niveau de batterie courant permet d'anticiper les opérations de maintenance.

Conclusion

Dans cette section nous avons présenté un état de l'art relatif au déploiement automatisé d'applications sur des infrastructures de capteurs, domaine de recherche relativement récent. Nous comparons au sein du tableau Tab. 2.2, les différentes approches de cette section par rapport aux critères identifiés.

Les approches de déploiement d'applications sur des infrastructures de capteurs répondent au critère d'identification des plateformes cibles : l'approche DIANE répond à ce critère à travers un générateur de déploiement, l'approche définie par [PC15] grâce à l'opérateur Map, et l'approche définie par [MFT17] par le serveur de provision. Cependant les approches de déploiement automatisées n'intègrent pas de mécanisme permettant l'optimisation du déploiement en fonction de critères non fonctionnels, *par ex.*, charge d'une plateforme ou niveau de batterie. Par rapport à ce dernier point, l'approche définie par [MFT17] a prévu d'intégrer au sein de travaux futurs une gestion de ces critères non fonctionnels au sein de leur *optimiser*. Un second point d'action de cette thèse sera de permettre un déploiement optimisé par rapport à des critères non fonctionnels de l'infrastructure de capteurs.

Nous avons également identifié des approches permettant de connaître la consommation énergétique des applications déployées. L'approche Wattskit [CFRS17], bien qu'étant destinée aux infrastructures de *cloud computing*, permet de définir des multimètres logiciels pour obtenir la consommation instantanée d'un service. L'approche définie par Dâmaso et al. [DFR+13] permet quant à elle d'estimer la consommation d'une application TinyOS avant son déploiement. Toutefois, elles ne permettent pas d'agir sur la consommation de l'application déployée, *par ex.*, en réduisant la fréquence d'envoi des données sur le réseau. Afin d'agir sur la consommation énergétique, des protocoles d'accès au réseau et optimisés pour l'économie d'énergie ont été établis. Toutefois, ces protocoles ne donnent pas d'informations contextuelles sur le niveau de la batterie, qui peuvent se révéler utiles aux experts réseaux pour des tâches de maintenance. Dans le cadre de cette thèse, nous nous focalisons du côté de l'ingénieur logiciel souhaitant exprimer et déployer des intentions métier. Nous adresserons les problématiques énergétiques en perspective de nos contributions.

TABLEAU 2.2 – Comparatif des approches de déploiement automatisé

		Identification des plateformes cibles	Utilisation d'une fonction d'optimisation de déploiement	Suivi des applications déployées et/ou de l'infrastructure
Déploiement d'applications	Diane	Oui (solveur de contraintes)	Non	Oui (*profiler*)
	Patel et al. [2015]	Oui (opérateur *Map*)	Non	Non
	Mohamed et al. [2017]	Oui (serveur de provision)	Planifié (*optimiser*)	Non
Suivi énergétique	WattsKit	*Non pertinent*	*Non pertinent*	Oui* (consommation énergétique d'un service)
	Dâmaso et al. [2013]	*Non pertinent*	*Non pertinent*	Partiellement (simulation avant déploiement de l'impact énergétique)
Protocoles radio	Miller et al. [2005]	*Non pertinent*	*Non pertinent*	Oui (protocoles radios adaptatifs pour réduire l'accès au réseau)
	Ye et al. [2002]			
	Van Dam et al. [2003]			
	Rajendra et al. [2006]			

* Approche pour les infrastructures distribuées, par ex. *cloud computing*

2.3 Partage de l'infrastructure de capteurs

Nous nous intéressons ici aux travaux relatifs à la problématique *(P_1, partage)* « comment une infrastructure peut être partagée entre plusieurs politiques de collecte de données afin d'éviter des déploiements redondants ? ». Le partage d'une infrastructure de capteurs permet à une même infrastructure d'être utilisée par plusieurs applications et par plusieurs utilisateurs.

En 2015, Khan et al. [KBG+15] font le constat que les infrastructures de capteurs sont « spécifiques à un domaine, spécifiées pour une application avec pas ou peu de possibilités de les réutiliser pour de nouvelles applications ». Compte tenu du coût de déploiement d'une infrastructure, la réutilisation de l'infrastructure est une nécessité. En 2016, les mêmes auteurs présentent une revue de littérature [KBG+16] comparant les techniques de virtualisation pour les infrastructures de capteurs et introduisent des critères de comparaison pouvant également s'appliquer à d'autres techniques de partage :

▶ ***Critère 9 : Partage de la plateforme***. Nous classons les approches par leur capacité à exécuter plusieurs applications sur la même plateforme.
▶ ***Critère 10 : Partage du réseau (isolation)***. Nous classons les approches par leur capacité à former des groupes de plateformes isolés du reste de l'infrastructure afin d'isoler l'exécution d'une application et de s'assurer que chaque groupe est dédié à l'exécution d'une unique application.
▶ ***Critère 11 : Approche indépendante du type de plateforme***. Nous classons les approches par leur capacité à être indépendante d'une solution matérielle et/ou logicielle spécifique.
▶ ***Critère 12 : Applicabilité aux plateformes à ressources contraintes***. Nous classons les approches par leur capacité à intégrer des plateformes à ressources contraintes, *i.e.*, ayant une faible mémoire et des capacités de calcul réduites.
▶ ***Critère 13 : Support de l'hétérogénéité de l'infrastructure***. Nous classons les approches par leur capacité à contenir des plateformes ayant des caractéristiques différentes.
▶ ***Critère 14 : Sélection des plateformes (similaire à 6)***. Nous classons les approches par leur capacité à identifier les plateformes sur lesquelles les différentes tâches composant une application sont exécutées

2.3.1 Bancs d'essai

Les bancs d'essai permettent le déploiement de prototypes d'applications dans des conditions expérimentales proches des caractéristiques du monde réel, *par ex.*, infrastructure en milieu urbain [ISH10]. Ils ont la particularité d'être organisés autour d'une architecture permettant le partage de l'infrastructure entre plusieurs applications et utilisateurs. Cette architecture repose sur deux couches : *(i)* un réseau de capteurs reconfigurable à distance et *(ii)* une couche logicielle permettant de gérer les différentes expérimentations et utilisateurs.

Au cours des précédentes années, plusieurs bancs d'essai à grande échelle et partagés ont été mis à disposition afin de supporter les expérimentations liées aux villes et bâtiments intelligents. Nous pouvons notamment citer les projets :

— EQUIPEX `FIT/IoT-Lab` [BDRCD+11] contenant 2700 plateformes de capteurs réparties à travers 6 sites. Cette infrastructure est principalement utilisée pour expérimenter de nouveaux protocoles de communication sans-fil.

— SmartSantander [SMG+14] contenant 2000 plateformes de capteurs réparties à travers la ville espagnole de Santander. Cette infrastructure permet d'expérimenter des scénarios liés aux villes intelligentes.

Afin d'utiliser ces infrastructures, l'ingénieur logiciel doit préalablement réserver un ensemble de ressources grâce à un service de réservation. Ce service associe une ou plusieurs plateformes, à un utilisateur pour une durée définie. L'ingénieur logiciel se connectera ensuite à ces plateformes pour déployer ses applications.

Analyse

▶ ***9. Partage de la plateforme***. Les bancs d'essai dédient l'utilisation d'une plateforme à une application donnée. Il n'y a donc pas de partage de la plateforme entre plusieurs applications.

▶ ***10. Partage du réseau***. Lors de la réservation, le banc d'essai attribue un ensemble de plateformes à un utilisateur donné durant un temps imparti. Ces approches permettent ainsi le partage du réseau.

▶ ***11. Approche indépendante du type de plateforme***. Ces approches ne posent pas de condition sur la configuration matérielle des plateformes. D'un point de vue logiciel, elles doivent pouvoir être reprogrammées à distance.

▶ ***12. Applicabilité aux plateformes à ressources contraintes***. Les plateformes à ressources contraintes sont intégrables au sein de bancs d'essai. Nous pouvons par exemple identifier des plateformes *WSN430* (10 ko de mémoire vive et 48 ko de mémoire pour le stockage du programme) au sein de l'infrastructure EQUIPEX `FIT/IoT-Lab`.

▶ ***13. Support de l'hétérogénéité de l'infrastructure***. Les bancs d'essai contiennent plusieurs types de plateformes de capteurs. Par exemple, l'EQUIPEX `FIT/IOT-Lab` contient 3 types et architectures de plateformes différents (*WSN430*, *M3* et *A8*).

▶ ***14. Sélection des plateformes***. La sélection des plateformes est effectuée à la réservation soit par une sélection manuelle de la part de l'utilisateur, soit par attribution automatique d'un ensemble de plateformes libres.

Les bancs d'essai sont construits pour le partage de l'infrastructure entre plusieurs applications et utilisateurs. Ils reposent sur un service de réservation attribuant un ensemble de plateformes à une application. Toutefois, cette solution ne permet pas le déploiement en production d'applications puisque la réservation est limitée dans le temps. De plus, en dédiant les plateformes à une seule application, le passage à l'échelle de l'infrastructure ne peut se faire qu'en augmentant le nombre de plateformes disponibles au sein de l'infrastructure.

2.3.2 Virtualisation

En ingénierie système, les techniques de virtualisation ont pour but principal d'exécuter plusieurs systèmes d'exploitation simultanément sur un même serveur grâce à une abstraction matérielle [BDF+03]. Ainsi, le nombre de machines physiques à acquérir est moindre, réduisant en même temps les différentes tâches de maintenance et d'exploitation. Dans le cadre des infrastructures de capteurs, la virtualisation a été conçue afin de permettre la fédération de plusieurs réseaux de capteurs hétérogènes à travers une même abstraction [IHLH12].

SenShare [LEMC12] est une abstraction logicielle visant à virtualiser une plateforme de capteurs (**critère 9**). Cette virtualisation repose sur une couche d'abstraction logicielle (`HAL Layer` sur FIG. 2.7) en amont du système d'exploitation multi-tâches de la plateforme de capteurs. Cette abstraction définit notamment un ensemble d'interfaces d'accès aux ressources permettant le développement d'applications indépendantes du matériel. Cette approche est toutefois dépendante du type de plateforme puisqu'elle nécessite une plateforme capable d'exécuter un système d'exploitation multi-tâches. De ce fait, elle n'est également pas adaptée aux systèmes à ressources contraintes. Les critères 10, 13 et 14 ne sont pas applicables puisque cette approche ne cible qu'une plateforme individuelle de l'infrastructure.

Khan et al. [KBG+15] présentent une architecture visant à virtualiser un réseau de capteurs afin de le partager entre plusieurs applications. Cette virtualisation du réseau repose sur trois couches (*cf.* FIG. 2.8) : *(i)* la couche physique contenant l'ensemble des plateformes et capteurs physiques, *(ii)* la couche des capteurs virtuels permettant de

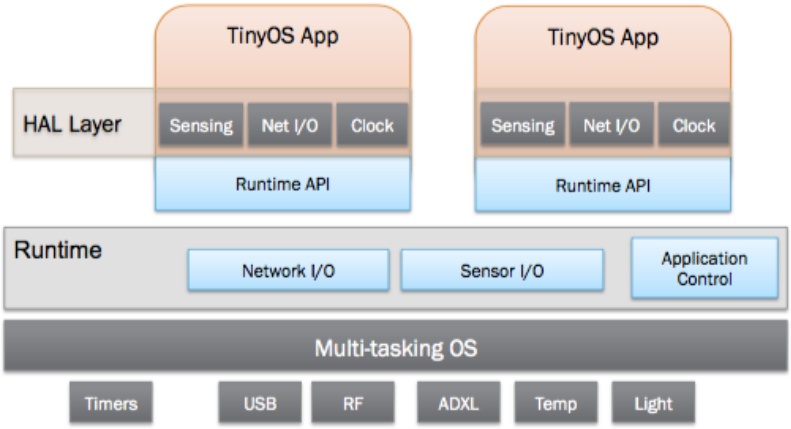


FIGURE 2.7 – Virtualisation d'une plateforme de capteurs (extrait de [LEMC12])

créer des instances virtuelles de plateformes contenant les capteurs (pouvant se trouver sur différentes plateformes physiques) souhaités par l'utilisateur (**critère 9**) et *(iii)* la couche de réseau superposée créant un réseau virtuel entre les instances de plateformes virtuelles (**critère 10**). Cette approche supporte des plateformes hétérogènes et laisse à l'utilisateur le choix des plateformes (virtuelles) à utiliser. Cette approche est toutefois dépendante du type de plateforme puisqu'elle nécessite une plateforme capable d'exécuter un système d'exploitation multi-tâches. De ce fait, elle n'est également pas adaptée aux systèmes à ressources contraintes.

Analyse

▶ ***9. Partage de la plateforme***. Les approches présentées permettent le partage de la plateforme de capteurs.

▶ ***10. Partage du réseau***. Seule l'approche définie par Khan et al. [KBG+15] permet la virtualisation du réseau.

▶ ***11. Approche indépendante du type de plateforme***. Les approches présentées nécessitent un système d'exploitation multi-tâches. Elles ne sont donc pas indépendantes du type de plateforme.

▶ ***12. Applicabilité aux plateformes à ressources contraintes***. Les approches présentées nécessitent des plateformes capables de supporter plusieurs flux d'exécution simultanés (un flux par application virtualisée). Elles ne sont donc pas applicables aux plateformes à ressources contraintes.

▶ ***13. Support de l'hétérogénéité de l'infrastructure***. L'approche définie par Khan et al. [KBG+15] supporte des plateformes hétérogènes.

▶ ***14. Sélection des plateformes***. L'approche définie par Khan et al. [KBG+15] offre une sélection libre par l'ingénieur logiciel des plateformes virtuelles à utiliser.

2.3.3 Planification de tâches

A la différence de la virtualisation, la planification de tâches ne nécessite pas d'exécuter simultanément plusieurs applications sur une même plateforme de capteurs. Elle est donc particulièrement adaptée pour les plateformes ne supportant pas des programmes multi-tâches.

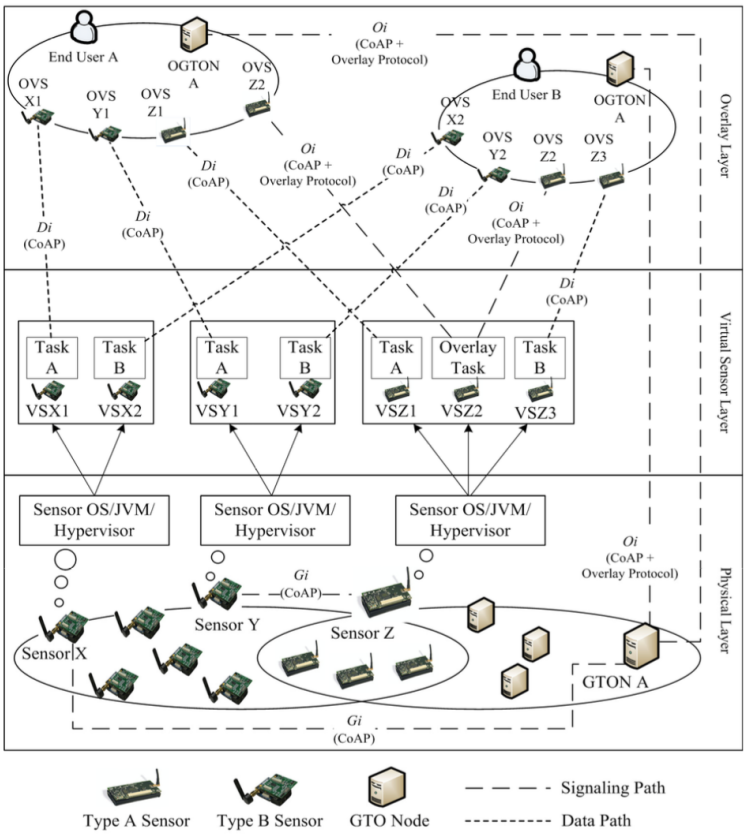


FIGURE 2.8 – Architecture de virtualisation multicouches pour des réseaux de capteurs sans-fil (extrait de [KBGC13])

Définition 2.3.1 (Tâches)**.** Une tâche est une unité d'exécution non divisible d'une application. [dFPD+13]

De Farias et al. présentent un algorithme de planification de tâches pour les réseaux de capteurs et d'actionneurs partagés. Les applications sont représentées comme des graphes orientés acycliques $T = (N, A)$ où N représente les tâches à être exécutées et A les dépendances entre les tâches. Le réseau de capteurs est, quant à lui, représenté sous la forme d'un graphe non orienté $G = (V, E)$ où V représente l'ensemble des plateformes de capteurs et E les liens de communication entre les différentes plateformes. L'algorithme proposé consiste à associer une file Q de tâches, provenant d'applications différentes, à chacune des plateformes de capteurs. Ensuite, pour toute tâche $n \in Q$, l'algorithme associe à n, une liste de capteurs disponibles CS pouvant réaliser n. Enfin, dans le but d'optimiser la consommation énergétique de l'infrastructure, l'algorithme choisit parmi CS le capteur c consommant le moins d'énergie, et qui sera utilisé pour exécuter la tâche. La validation de leur approche a été effectuée sur des plateformes SUN SPOT physiques et simulées.

Majeed et al. [MZ10] présentent une architecture visant à exécuter plusieurs applications sur un réseau de capteurs. Les auteurs ont pour objectifs *(i)* d'exécuter plusieurs applications dans une séquence préprogrammée et *(ii)* de reconfigurer dynamiquement l'infrastructure. Pour satisfaire ces objectifs, deux agents, *Switching agent* et *Configuration agent*, ont respectivement pour responsabilité de changer l'application en cours d'exécution et de reconfigurer chacune des plateformes de l'infrastructure. L'ingénieur en charge du réseau de capteurs défini une séquence prioritisée d'exécution d'applica-

tions. Après la terminaison d'une application n ou lors de la réception d'un évènement spécifique, *par ex.*, une valeur seuil, l'application n sera stoppée et l'application $n+1$ sera déployée (grâce à l'agent de configuration) puis exécutée. La validation de leur approche a été effectuée sur des plateformes Mica.

Analyse

▶ ***9. Partage de la plateforme***. Les algorithmes de planification de tâches visent à attribuer un ensemble de tâches, pouvant provenir d'applications différentes, puis à ordonnancer l'exécution des tâches et la récupération des valeurs produites. Chaque plateforme étant associée à une unique tâche, ces approches ne permettent pas le partage de la plateforme.

▶ ***10. Partage du réseau***. Seules les plateformes ayant besoin d'exécuter une tâche à un instant t sont actives. Dès lors, le groupe de plateformes est isolé des plateformes *dormantes*, validant le critère du partage du réseau.

▶ ***11. Approche indépendante du type de plateforme***. Ces approches ne posent pas de condition sur la configuration matérielle des plateformes. D'un point de vue logiciel, l'infrastructure de capteurs doit pouvoir être reconfigurée à distance.

▶ ***12. Applicabilité aux plateformes à ressources contraintes***. La planification de tâches ne nécessite pas de plateformes capables de supporter plusieurs flux d'exécution. Elles sont donc particulièrement adaptées pour les plateformes à ressources contraintes.

▶ ***13. Support de l'hétérogénéité de l'infrastructure***. Les approches par planification de tâches ont été validées sur un matériel homogène.

▶ ***14. Sélection des plateformes***. La sélection des plateformes est effectuée par l'algorithme de planification de tâches (association d'une tâche à une plateforme).

2.3.4 Dissémination de code

L'approche de dissémination de code vise à distribuer du code stocké dans un élément central aux différentes plateformes de capteurs.

Yu et al. [YRBL06] critiquent les approches de composition d'un ensemble d'applications au sein d'un unique fichier source au motif que le fichier exécutable pourra dépasser la mémoire de stockage et d'exécution des plateformes de capteurs. Ils proposent, Melete, une technique de dissémination de code au sein du réseau de capteurs en s'appuyant sur la machine virtuelle Maté [4] [LC02]. Leur approche repose sur trois points : *(i)* la modification de Maté pour supporter différentes applications au niveau de la plateforme de capteurs, *(ii)* une technique de groupage permettant de déployer différentes applications au niveau réseau (**critère 10**), et *(iii)* une technique réactive de transport de code uniquement à destination des plateformes à reconfigurer. Ces redéploiements dynamiques peuvent être déclenchés par une variation dans l'environnement ou par la mise à jour du logiciel par un administrateur réseau. Ils définissent un ensemble d'états et de transitions décrivant, au niveau de la plateforme, l'acquisition du code à exécuter. Durant l'état `MAINTAIN`, la plateforme publie périodiquement la version du code quelle héberge. Si elle détecte une modification du code hébergé par les autres plateformes voisines ,*i.e.*, la version de code diffère sur les autres plateformes directement connectées, alors la plateforme passe en état `REQUEST` afin de demander

4. permet le support du langage TinyScript et la reprogrammation des plateformes de capteurs

la dernière version de code. Les plateformes qui recevront la demande de code passeront en état `RESPONSE` où elles transmettront la dernière version du code. Enfin, pour propager le code à des plateformes localisées plus loin dans l'infrastructure de capteurs, *i.e.*, à plus d'une connexion de la plateforme considérée, l'état `FORWARD` permet à une plateforme de transmettre du code à d'autres plateformes. La validation de leur approche a été effectuée sur des plateformes TelosB (10 ko RAM, 48 ko ROM et 1 Mo Flash), mais supporte n'importe quelle plateforme capable d'exécuter la machine virtuelle Maté. L'empreinte mémoire de Melete mesurée est de l'ordre de 36.7 ko en mémoire ROM et 3 ko en mémoire RAM, ce qui est supportable pour des systèmes à ressources contraintes (**critère 12**).

Analyse

▶ ***9. Partage de la plateforme***. La modification de Maté permet l'exécution simultanée de plusieurs applications sur une même plateforme.

▶ ***10. Partage du réseau***. Au niveau réseau, l'application peut être déployée sur différentes portions du réseau (pouvant se chevaucher).

▶ ***11. Approche indépendante du type de plateforme***. L'approche présentée repose sur le déploiement préalable d'une machine virtuelle sur les plateformes de l'infrastructure de capteurs.

▶ ***12. Applicabilité aux plateformes à ressources contraintes***. La faible empreinte mémoire de Melete rend son utilisation adaptée aux plateformes à ressources contraintes. Son utilisation a été testée sur des plateformes TelosB.

▶ ***13. Support de l'hétérogénéité de l'infrastructure***. L'utilisation d'une machine virtuelle permet d'abstraire l'hétérogénéité des plateformes.

▶ ***14. Sélection des plateformes***. L'approche n'effectue pas de sélection des plateformes. Le code transite d'une plateforme à l'autre en fonction des variations dans l'environnement ou de mises à jour logicielles.

2.3.5 Approches « cloud »

Les approches *cloud* visent à traiter et stocker les valeurs de capteurs au sein de serveurs localisés hors du réseau de capteurs. Elles ont pour avantage d'abstraire la complexité de l'infrastructure à travers un ensemble de services et sont particulièrement adaptées aux scénarios nécessitant de consommer un nombre important de données. Les applications concernées peuvent ainsi reposer sur des algorithmes de fouille de données pour identifier et extraire les données appropriées à un besoin spécifique. Ainsi, les préoccupations de collecte et d'exploitation sont séparées pour l'utilisateur.

Dans l'approche *Sensor Cloud* [MKD14], les auteurs introduisent la notion de *capteurs virtuels en nuage*. Un capteur virtuel est une émulation d'un capteur physique qui reçoit ses données depuis les capteurs physiques composant le réseau de capteurs. Ils permettent notamment d'obtenir une vue personnalisée des données ou d'appliquer une fonction de transformation sur les données. L'architecture présentée dans leurs travaux repose sur trois couches : *(i)* une couche physique contenant les capteurs, *(ii)* un *cloud* contenant la configuration des capteurs virtuels et recevant les données depuis la couche physique, et *(iii)* la couche applicative contenant les différentes applications métiers. Dans ce type d'architecture, le réseau de capteurs n'est pas reconfiguré lors du déploiement d'application et n'héberge pas de code métier. L'ensemble des transformations à appliquer sur les données est effectué dans le *cloud*.

SenaaS [ACN10] est une architecture *Sensor-as-a-Service*, exposant uniquement les aspects fonctionnels des plateformes de capteurs et masquant les aspects techniques. L'architecture expose ensuite les capteurs sous forme de services Web sur lesquels différentes applications viendront se connecter.

Analyse

▶ ***9. Partage de la plateforme***. Les plateformes de l'infrastructure sont *indirectement* partagées entre différentes applications grâce à l'utilisation de services. En effet, plusieurs applications peuvent se connecter au même service pour récupérer les données de l'infrastructure de capteurs.

▶ ***10. Partage du réseau***. Dans les approches *cloud*, le réseau fonctionne en boite noire et ne permet la récupération de valeurs qu'à travers un ensemble de services. Ces approches ne permettent alors pas la segmentation du réseau en vue d'isoler des plateformes pour une application donnée.

▶ ***11. Approche indépendante du type de plateforme***. Le réseau de capteurs doit être connecté à Internet ce qui est une caractéristique des infrastructures de capteurs (*cf.* FIG. 1.1). L'approche est donc indépendante du type de plateformes.

▶ ***12. Applicabilité aux plateformes à ressources contraintes***. Ces approches sont applicables aux plateformes à ressources contraintes puisque ces dernières ont uniquement pour tâche de lire une valeur de capteur et de l'envoyer sur l'infrastructure de capteurs.

▶ ***13. Support de l'hétérogénéité de l'infrastructure***. L'hétérogénéité de l'infrastructure est masquée à travers les services exposés aux applications.

▶ ***14. Sélection des plateformes***. Les approches *cloud* n'exécutent pas de tâches applicatives sur les plateformes de capteurs. Ainsi, elles ne possèdent pas de mécanisme de sélection des plateformes.

Conclusion

Dans cette section nous avons présenté un état de l'art permettant de partager une infrastructure de capteurs entre plusieurs applications.

Nous comparons au sein du tableau FIG. 2.3 les différentes approches de cette section par rapport aux critères identifiés. Nous pouvons notamment remarquer que les approches supportant l'hétérogénéité de l'infrastructure ne permettent pas le partage d'une même plateforme entre plusieurs applications (à l'exception notable de Melete, mais cette dernière pose une condition forte sur le type de plateforme utilisé). Dans cette configuration, le passage à l'échelle de l'infrastructure de capteurs lors du déploiement de nouvelles applications ne pourra s'effectuer qu'en augmentant le nombre de plateformes.

Ainsi, un troisième point d'action de cette thèse est d'adresser le partage des plateformes de capteurs au sein d'une infrastructure de capteurs hétérogènes.

TABLEAU 2.3 – Comparatif des approches de partage de l'infrastructure

		Partage de la plateforme	Partage du réseau (isolation)	Approche indépendante du type de plateforme	Applicabilité aux plateformes à ressources contraintes	Support de l'hétérogénéité	Sélection de plateformes
Bancs d'essais	FIT IOT-LAB	Non	Oui (attribution d'un ensemble de plateformes par application)	Partiellement (reprogrammation à distance)	Oui	Oui	Oui (au moment de la réservation)
	SmartSantander	Non	Oui (attribution d'un ensemble de plateformes par application)	Partiellement (reprogrammation à distance)	Oui	Oui	Oui (au moment de la réservation)
Virtualisation	SenShare	Oui	N/A	Non (nécessite un système d'exploitation multi-tâche)	Non	N/A	N/A
	Khan et collab. [2015]	Oui	Oui (réseau virtuel)	Non (nécessite un système d'exploitation multi-tâche)	Non	Oui	Oui (choix des plateformes virtuelles laissé à l'utilisateur)
Planification de tâches	De Farias et collab, [2013]	Non	Oui	Partiellement (reprogrammation à distance)	Oui	? (validation sur un matériel homogène)	Oui (par l'algorithme de planification)
	Majeed et Zia [2010]	Non	Oui	Partiellement (reprogrammation à distance)	Oui	? (validation sur un matériel homogène)	Oui (par l'algorithme de planification)
Dissemination	Melete	Oui	Oui	Non (nécessite le déploiement préalable d'une machine virtuelle)	Oui	Oui (grâce à la machine virtuelle déployée)	Non
Cloud	Sensor Cloud	Partiellement (vue personnalisée)	Non	Oui	Oui	Oui	Non
	SenaaS	Partiellement (service web)	Non	Oui	Oui	Oui	Non

2.4 Approches intégrées

Nous nous intéressons maintenant aux travaux relatifs à la problématique *(P_3, séparation)* « comment permettre à un ingénieur logiciel et un expert réseau de travailler ensemble à l'expression de collecte de données tout en restant concentré sur leurs compétences respectives ? ». Les tâches développement incluent l'ensemble des activités depuis l'expression des besoins métier sous forme d'une collecte de données jusqu'au déploiement de la collecte de données sur une infrastructure de capteurs hétérogène.

Nous rappelons que le rôle de l'ingénieur logiciel est d'exprimer, sous forme de collecte de données, des besoins métiers et que le rôle d'un expert réseau est de maintenir un fonctionnement optimal de l'infrastructure. En particulier, l'ingénieur logiciel ne doit pas manipuler des notions bas niveau propres aux infrastructures de capteurs et la compréhension des spécifications auxquelles doit répondre l'ingénieur logiciel est hors des préoccupations métiers de l'expert réseau.

Dans les sections précédentes, nous avons adressé, de manière indépendante, les travaux relatifs à l'expression des besoins métiers sur une infrastructure de capteurs (SEC. 2.1), à leur déploiement automatisé (SEC. 2.2) et au partage des infrastructures (SEC. 2.3). Nous nous focalisons dans cette section sur une approche intégrée.

Cette approche étant orthogonale aux travaux précédemment identifiés, nous introduisons les critères de comparaison suivants :

▶ ***Critère A : Expression de besoins métiers***. Nous classons les approches par leur capacité à fournir des abstractions permettant d'exprimer des besoins métiers.
▶ ***Critère B : Déploiement automatisé***. Nous classons les approches par leur capacité à déployer automatiquement les besoins métiers sur une infrastructure de capteurs.
▶ ***Critère C : Partage de l'infrastructure***. Nous classons les approches par leur capacité à partager une infrastructure lors du déploiement de multiples besoins métiers.

Le nouveau standard d'informatique en nuage introduit par [LVCD13] permet la création d'applications IoT en décrivant la topologie interne des composants de l'application ainsi que le processus de déploiement. Ces spécifications sont ensuite manipulées par un environnement chargé de produire des fichiers sources prêts à être mis en œuvre sur un ensemble de plateformes identifiées automatiquement (**critère B**). En séparant la description de l'application du déploiement, ils offrent un support envers l'hétérogénéité des plateformes. Toutefois, cette chaîne de développement ne supporte pas le partage de l'infrastructure et n'offre pas de langage spécifique au domaine permettant à un ingénieur logiciel de développer ; sans connaissances préalables, des plateformes contenues dans l'infrastructure de capteurs.

Patel et al. définissent une approche de développement dirigée par les modèles et mettant l'accent sur l'identification et la séparation des différentes parties prenantes dans le processus de développement : *expert métier*, *concepteur d'applications*, *développeur d'applications*, *expert réseau* et *développeur embarqué*. Chacun de ces acteurs fournit une spécification en lien avec son champ d'expertise respectif (**critère A**). Plusieurs opérateurs viennent ensuite, à différentes étapes, manipuler ces spécifications afin de produire des fichiers sources prêts à être transférés sur des plateformes identifiées automatiquement (**critère B**). Cette approche définit de bonnes pratiques dans le développement d'applications IoT en laissant chaque partie prenante focalisée sur sa propre expertise métier. De plus, les spécifications sont interchangeables entre plusieurs projets, *par ex.*, la spécification de l'application est réutilisable sur une nouvelle infrastructure ou la spécification de l'expert réseau est réutilisable lors du déploiement d'une nouvelle application sur la même infrastructure de capteurs. Cette approche permet ainsi le déploiement d'applications réutilisables et indépendantes du matériel à haut niveau d'abstraction. Cependant, elle ne supporte pas le partage de l'infrastructure lorsque de nouvelles applications sont identifiées.

ThingML [FMS11] est également un exemple d'approche de développements par les modèles. L'approche repose sur un langage spécifique au domaine permettant à un développeur d'exprimer des *Things* et des *Configurations*, cf Lst. 2.3. Un *thing* correspond à un composant logiciel. Sa structure interne met en œuvre des procédures (exprimées en utilisant des actions indépendantes des plateformes, ou en utilisant le langage supporté par la plateforme cible), des structures Evenement-Condition-Action et des machines à états afin de réagir par rapport à des évènements. Chacun des *things* peut envoyer et recevoir des messages à travers des *ports*. Les *Configurations* décrivent les interconnexions des *things*. Une famille de compilateur transforme ensuite le modèle ThingML en code supporté par la plateforme cible (**critère B**). Cette approche permet ainsi le développement d'applications dépendantes ou indépendantes du matériel en fonction de la structure interne des *things*. L'utilisation d'un langage spécifique au domaine permet également d'augmenter le niveau d'abstraction (**critère A**). Cependant, cette approche est centrée sur une plateforme et ne permet pas la programmation

d'une infrastructure depuis un unique modèle ThingML. De plus, l'approche ne supporte pas le partage de la plateforme lorsque de nouvelles applications sont identifiées.

Listing 2.3 – Exemple de *thing* mettant en œuvre une machine à états décrivant le clignotement périodique d'une LED sur une plateforme et sa configuration associée pour une plateforme Arduino (source : https://github.com/TelluIoT/ThingML/blob/master/org.thingml.samples/src/main/thingml/samples/blink.thingml)

```
thing Blink includes LedMsgs, TimerMsgs
{
        required port HW
    {
                sends led_toggle, timer_start
        receives timer_timeout
        }

    statechart BlinkImpl init Blinking
    {
        state Blinking
        {
            on entry HW!timer_start (1000)

            transition -> Blinking
            event HW?timer_timeout
            action HW!led_toggle ()
        }
    }
}

configuration BlinkArduino
{
        group led : LedArduino
    set led.io.digital_output.pin = DigitalPin:PIN_13
        // The timer
        instance timer : TimerArduino
        // The blink application
        instance app : Blink
        connector app.HW => led.led.Led
        connector app.HW => timer.timer
}
```

L'approche Kevoree [FNM+14] permet l'exécution et la reconfiguration (**critère B**) de plateformes à partir de l'analyse de modèles à temps d'exécution (paradigme *models@run.time* [BBF09]). Le paradigme models@run.time permet l'utilisation des techniques de manipulation de modèle directement au sein de l'environnement d'exécution. Ainsi, des éléments du modèle peuvent être mis à jour en fonction du contexte d'exécution sans avoir à arrêter le système ou à remplacer l'ensemble du modèle. Ce paradigme est ainsi plébiscité pour l'adaptation de systèmes critiques ne pouvant être arrêtés pour des tâches de maintenance. Au sein de Kevoree, ce paradigme permet d'appliquer aux modèles une approche Observateur/Observable : un modèle de référence surveillé (l'observable) est observé par des plateformes de capteurs (les observateurs). Lors de la mise à jour d'un élément du modèle observé, les plateformes observatrices sont notifiées du changement et déclenchent une action adéquate en fonction de l'information reçue, *par ex.*, changer leur configuration. Il est possible d'utiliser cette approche pour exécuter plusieurs besoins métiers sur une infrastructure de capteurs en composant le modèle observé avec de nouveaux modèles décrivant de nouveaux besoins métiers (**critères A et, indirectement, critère C**). Les plateformes observatrices seront alors automatiquement configurées pour exécuter le nouveau modèle composé.

Analyse

Dans cette section, nous avons présenté des approches intégrées pour le développement d'applications à destination des infrastructures de capteurs.

Nous comparons au sein du tableau Tab. 2.4 les différentes approches de cette section par rapport aux critères identifiés. Nous pouvons remarquer que les approches permettent l'expression de besoins métier (avec toutefois une expertise bas-niveau pour l'approche présentée par Li et al. [LVCD13]) et le déploiement de ces besoins des plateformes de capteurs. Toutefois, ces approches ne permettent pas le partage des ressources de l'infrastructure lorsque de nouveaux besoins sont identifiés.

Ainsi, un quatrième point d'action de cette thèse est de fournir une approche intégrée permettant le partage de l'infrastructure lorsque de nouvelles applications sont identifiées.

Tableau 2.4 – Comparatif des approches intégrées

	Besoins métiers	**Déploiement automatisé**	**Partage de l'infrastructure de capteurs**
Li et collab. [2013]	Partiellement (nécessite une expertise bas-niveau)	Oui	Non
Patel et Cassou [2015]	Oui (modèles d'application)	Oui	Non
ThingML	Oui (langage spécifique au domaine)	Partiellement (uniquement génération de code pour une unique plateforme)	Non
Kevoree	Oui (modèle de référence)	Oui (mise à jour des plateformes observatrices)	Indirectement (composition du modèle de référence)

2.5 Conclusion

Dans ce chapitre, nous nous sommes intéressés aux problèmes de l'expression des besoins métiers, du déploiement automatisé et du partage de l'infrastructure de capteurs. Pour chacun de ces problèmes, nous avons identifié un certain nombre de critères et des motivations à partir d'éléments manquants de l'état de l'art :

— *Un développement par les ingénieurs logiciels :* proposer une approche d'expression de besoins métiers à haut niveau d'abstraction sur une infrastructure de capteurs hétérogènes et offrant des mécanismes de réutilisation logicielle.

— *Un déploiement optimisé :* proposer un mécanisme de déploiement optimisé par rapport à des critères fonctionnels de l'infrastructure de capteurs.

— *Partage d'une infrastructure hétérogène :* proposer une approche rendant possible le déploiement de plusieurs besoins métiers sur une même infrastructure de capteurs.

— *Approche intégrée supportant le partage :* proposer une approche permettant l'expression de besoins métier, leur déploiement automatique ainsi que le partage de l'infrastructure lorsque de nouveaux besoins sont identifiés.

Chapitre 3

Motivations

Sommaire

3.1 Introduction

Afin de répondre aux problématiques soulevées par cette thèse, nous avons identifié au sein de l'état de l'art Chap. 2 un certain nombre de travaux adressant les problématiques individuellement. Toutefois, nous avons montré qu'aucune approche ne recouvrait l'ensemble des critères identifiés pour chaque problématique. De plus, nous n'avons pas trouvé d'approche **intégrée** permettant de répondre à l'ensemble des problématiques. Nous présentons ainsi dans ce chapitre les motivations qui animent cette thèse et nous fixons les objectifs que nous allons atteindre.

3.2 Un développement par les ingénieurs logiciels

Nous visons ici à répondre à la problématique « comment permettre à un ingénieur logiciel et un expert réseau de travailler ensemble à l'expression de collecte de données tout en restant concentré sur leurs compétences respectives ? ». Nous avons présenté en Sec. 2.1 des travaux permettant à un ingénieur logiciel de programmer un réseau de capteurs. Les contributions de cette thèse étant destinées pour ce dernier, nous nous fixons l'objectif « (O_1) un ingénieur logiciel ne manipule que des concepts métiers ».

Hypothèse H_1 (Utilisateurs ciblés) : Les utilisateurs ciblés par cette thèse sont les ingénieurs logiciels ayant des compétences en développement d'applications, mais ne possédant pas de connaissances par rapport aux concepts propres aux réseaux de capteurs.

Pour répondre à cette problématique, nous adresserons les points présentés ci-après.

3.2.1 Permettre un développement indépendant du matériel

Si la mise en œuvre d'une politique de collecte de données traduisant des spécifications d'expert métier est dans le domaine métier de l'ingénieur logiciel, la gestion des problématiques inhérentes aux réseaux de capteurs ne l'est pas. Pourtant, le code mettant en œuvre la politique de collecte de données est fortement lié au matériel. La figure Fig. 3.1 illustre la récupération de la température sur des plateformes différentes équipées de capteurs de température différents : à gauche, la plateforme considérée est de type TMote Sky intégrant un capteur de température SHT11 et, à droite, la plateforme considérée est de type WSN430. Ainsi, si l'intention reste identique (« récupérer la valeur de température »), le code à produire est différent selon la plateforme considérée.

Ce premier constat soulève deux problèmes :

— La compréhension des architectures matérielles des plateformes de capteurs est hors du périmètre métier de l'ingénieur logiciel.

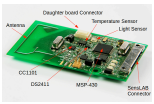
— Dans le cadre d'une infrastructure hétérogène, le code utilisé pour récupérer une valeur de température sur une plateforme n'est pas utilisable sur une plateforme d'architecture matérielle différente.

L'Ingénierie Dirigée par les Modèles (IDM) [Sch06] est une technique d'ingénierie logicielle permettant de décrire, sous forme de modèles, un besoin et sa solution. Les approches d'Architecture Dirigée par les Modèles (ADM) sont une variante de l'IDM ayant pour objectif de proposer un modèle indépendant de la plateforme, puis de le transformer en modèle spécifique à la plateforme cible pour décrire sa mise en œuvre effective.

```c
#include "contiki.h"
#include "dev/sht11-sensor.h"
#include "dev/leds.h"
#include <stdio.h>

//Declare the process
PROCESS(send_sensor_info_process, "Print the Sensor Information");
//Make the process start when the module is loaded
AUTOSTART_PROCESSES(&send_sensor_info_process);
/*---------------------------------------------------------------------------*/
static int
get_temp(void)
{
  return ((sht11_sensor.value(SHT11_SENSOR_TEMP) / 10) - 396) / 10;
}
/*---------------------------------------------------------------------------*/

//Define the process code
PROCESS_THREAD(send_sensor_info_process, ev, data)
{
  PROCESS_BEGIN();
  SENSORS_ACTIVATE(sht11_sensor);
  printf("Temperature: %d \n", get_temp());
  PROCESS_END();
}
```

Capteur de température SHT11
Plateforme : TMote Sky
OS: Contiki

```c
/* ... */
static int temperature_sensor()
{
   int16_t value_0, value_1;
   char value[sizeof(value_0) + sizeof(".") +sizeof(value_1)];
   value_0 = ds1722_read_MSB();
   value_1 = ds1722_read_LSB();
   value_1 >>= 5;
   value_1 *= 125;

   sprintf(value, "%i%i", value_0, value_1);

   return atoi(value);
}
/* ... */
static void hardware_init()
{
   // Initialize the temperature sensor
   ds1722_init();
   ds1722_set_res(12);
   ds1722_sample_cont();
}
/* ... */
int main( void )
{
   hardware_init();
   while (1) {
      printf("Temperature: %i\n", temperature_sensor());
      delay(DELAY >> 2);
   }
   return 0;
}
```



Capteur de température DS1722
Plateforme : WSN430 (Equipex FIT IoT-Lab)
OS: Aucun (programmation du micro-contrôleur)

FIGURE 3.1 – Récupération de la valeur de température sur deux plateformes de capteurs différentes

Une définition d'un modèle indépendant par rapport aux plateformes peut être trouvée dans [AVSPQ03] : « un modèle qui abstrait les technologies et les plateformes qui vont être utilisées pour mettre en œuvre l'application ».

Ainsi, appliquée à notre contexte, l'utilisation des techniques d'IDM/ADM est une bonne pratique permettant à un ingénieur logiciel d'exprimer des politiques de collecte de données indépendamment des plateformes de l'infrastructure.

3.2.2 Permettre un développement à haut niveau d'abstraction

Le développement d'une politique de collecte de données nécessite d'intégrer, au sein du code métier, des éléments spécifiques au réseau de capteurs considéré. Principalement, l'ingénieur logiciel doit manipuler des connexions réseau avec des réseaux propres à l'Internet des objets, *par ex.*, 6LoWPAN ou IEEE 802.15.4, gérer finement la consommation énergétique de son programme sur des plateformes alimentées par batterie et vérifier que la taille du programme exécutable obtenue n'est pas supérieure à la capacité de stockage de la plateforme, usuellement de l'ordre de quelques kilo-octets.

Mais pour réaliser ces différentes tâches, les langages de programmation pour les plateformes de capteurs nécessitent d'utiliser des éléments de bas niveau pour interagir avec les matériels.

Parameters	Typical	Unit
Tx802.11b, CCK 11Mbps, P OUT=+17dBm	170	mA
Tx 802.11g, OFDM 54Mbps, P OUT =+15dBm	140	mA
Tx 802.11n, MCS7, P OUT =+13dBm	120	mA
Rx 802.11b, 1024 bytes packet length , -80dBm	50	mA
Rx 802.11g, 1024 bytes packet length, -70dBm	56	mA
Rx 802.11n, 1024 bytes packet length, -65dBm	56	mA
Modem-Sleep①	15	mA
Light-Sleep②	0.9	mA
Deep-Sleep③	10	uA
Power Off	0.5	uA

FIGURE 3.2 – Extrait de la documentation technique de la plateforme ESP8266 relatif à la consommation énergétique

Par exemple, considérons que la cible de déploiement de la politique fil rouge soit une plateforme ESP8266, que l'on peut retrouver dans l'industrie. En fonction de l'utilisation faite des ressources réseau et du choix du mode d'économie d'énergie (Modem-sleep, Light-sleep ou Deep-sleep), la consommation énergétique de la plateforme peut varier d'un facteur 17000 (*cf.* FIG. 3.2) ! De plus, le choix du mode d'économie d'énergie contraint les ressources matérielles : en Modem-sleep, la plateforme peut lire et stocker localement des valeurs à partir des capteurs qui lui sont physiquement connectés alors qu'en Deep-sleep, la plateforme ne pourra pas effectuer d'opération. Les instructions permettant de choisir le mode d'économie énergétique sont directement explicitées, à l'aide des instructions présentées sur le listing LST. 3.1, au sein du code transféré sur la plateforme. Pourtant, ces instructions ne participent pas à la logique métier et changent le comportement de la plateforme. De plus, leur utilisation suppose une connaissance avancée des plateformes de l'infrastructure de capteurs. Au niveau du langage fourni à l'ingénieur logiciel, seules des instructions participant à la logique métier devraient être disponibles.

Listing 3.1 – Prototypes de fonctions configurant le mode d'économie d'énergie pour une plateforme ESP8266

```
wifi_set_sleep_type(LIGHT_SLEEP_T)
void system_deep_sleep(uint32 time_in_us)
void system_deep_sleep_instant(uint32 time_in_us)
```

Des abstractions existent pour le développement d'une politique de collecte de données (*cf.* SEC. 2.1), mais elles restent spécifiques à une plateforme ou à une infrastructure homogène. Le développement doit ainsi s'effectuer de manière indépendante du matériel comme le permettent les approches IDM. Pour supporter l'expression des

problématiques métiers, les langages spécifiques au domaine, ou langages dédiés sont une bonne pratique IDM pour monter le niveau d'abstraction [Fow10, MEP17]. Ainsi, appliqués à notre contexte, ils permettent la définition d'un langage ne proposant que des concepts liés à l'expression d'une politique de collecte de données.

3.2.3 Permettre un développement réutilisable

La réutilisabilité est une pratique d'ingénierie logicielle permettant de réduire les temps de développement et d'encourager la production d'applications contenant des éléments standardisés. Les techniques actuelles ne permettent aucune réutilisation ou uniquement une réutilisation envers des plateformes ou infrastructures architecturalement identiques.

Au niveau de l'exemple fil rouge, un ingénieur logiciel met en œuvre la politique de collecte de données à l'échelle d'un bureau. Lors du passage à l'échelle du bâtiment, il doit pouvoir réutiliser cette politique sans avoir à écrire le code pour l'ensemble des bureaux. Cependant, l'infrastructure hétérogène fait que le code écrit pour une plateforme n'est pas directement réutilisable sur une plateforme différente, *par ex.*, le code de récupération de la valeur de température n'est pas le même entre une plateforme TMote Sky et une plateforme WSN430, *cf.* Fig. 3.1.

De plus, certains éléments de la politique peuvent être pertinents et être réutilisés en tant que tels pour une politique différente. Par exemple, au sein du fil rouge, le traitement identifiant si le bloc climatiseur réversible est en mode de chauffage ou de refroidissement peut être pertinent pour une autre politique, *par ex.*, détecter si la climatisation est en marche alors que la fenêtre du bureau est ouverte. Un ingénieur logiciel doit pouvoir isoler ce traitement afin de le réutiliser, *par ex.*, sous forme de composant, au sein d'une autre politique. Cependant, le code isolé reste spécifique à l'architecture de la plateforme sur laquelle il a été extrait.

Le problème de la réutilisabilité est directement lié au fait que l'artefact repris est dépendant de l'architecture de la plateforme ou de l'infrastructure originelle. En proposant une approche dirigée par les modèles, la politique de collecte de données sera décrite indépendamment de la plateforme, *cf.* Sec. 3.2.1. Au sein des techniques IDM, la réutilisation d'un modèle au sein d'un autre modèle peut être effectuée grâce au concept de hiérarchie. La notion de réutilisation hiérarchique est également utilisée au sein du modèle de composant FRACTAL [BCL+06]. Appliquée à notre contexte, une politique de collecte de données peut encapsuler une autre politique de collecte de données. Par exemple, dans le cadre d'une ville intelligente où des politiques permettent de compter le nombre de places de stationnement libres par quartier, une politique définie au niveau de la ville et souhaitant avoir le nombre total d'emplacements libres encapsulerait les politiques définies pour chacun des quartiers et effectuerait une somme de leur résultat. Dans ce contexte, la réutilisation restera indépendante vis-à-vis du matériel. Un ingénieur logiciel peut également souhaiter uniquement réutiliser une sous-partie de politique. Dans ce cas, il ne devrait pas devoir réécrire le code correspondant. Des opérateurs de manipulation de modèle existent pour supporter l'extraction et l'intégration de fragments de modèles au sein d'autres modèles, *par ex.*, les techniques de *pruning* de méta-modèle [SMBJ09]. Par exemple, si un ingénieur logiciel a à sa disposition une politique comptant le nombre d'emplacements de stationnement libres par quartier et qu'il souhaite obtenir le nombre d'emplacements libres pour une rue donnée, il doit pouvoir tailler (*prunning*) la politique du quartier pour isoler les actions liées à la rue considérée.

3.3 Déploiement d'une politique sur une infrastructure

Nous visons ici à répondre à la problématique « comment une collecte de données peut-elle être exprimée sur une infrastructure hétérogène ? ». Nous avons présenté en Sec. 2.2 un des travaux permettant le déploiement automatisé d'applications sur un réseau de capteurs. Pour valider la réponse à cette problématique, nous nous fixons les objectifs suivants : « (O_2) une politique est adaptée aux capacités matérielles » et « (O_3) une politique est projetée sur l'infrastructure ». Pour mener à bien le déploiement d'une politique, nous faisons l'hypothèse que les plateformes de capteurs sont reprogrammables :

Hypothèse H_2 (Plateformes reprogrammables) : Les plateformes constituant l'infrastructure de capteurs sont reprogrammables. Dans le cadre de cette thèse, nous considérons leur reprogrammation comme étant l'écriture d'un nouveau programme embarqué.

Pour répondre à cette problématique, nous adresserons les points présentés ci-après.

3.3.1 Permettre la projection automatique des intentions métier sur l'infrastructure

Les politiques de collecte de données sont définies au niveau de l'infrastructure et peuvent ainsi faire intervenir différentes plateformes pour leur réalisation. Par exemple, la définition de la politique fil rouge fait intervenir explicitement une plateforme située à l'extérieur du bâtiment intelligent en plus de la/les plateforme(s) déployée(s) dans chaque bureau. Les actions liées à la récupération des valeurs de température extérieure doivent ainsi être projetées sur cette plateforme distante.

De plus, une projection est issue d'un choix parmi plusieurs possibilités de projection. Par exemple, la vérification d'un seuil de température peut s'effectuer aussi bien sur la plateforme produisant la valeur de température ou sur une plateforme relayant ces données de température. Nous considérons que la connaissance de la topologie réseau est hors de l'expertise d'un ingénieur logiciel et dépend de choix de conception fait par l'expert réseau ayant déployé l'infrastructure.

Ensuite, le choix des plateformes cibles pour l'exécution d'une politique de collecte de données est un problème compliqué. En effet, un choix de projection non trivial peut être préférable en fonction de caractéristiques propres aux plateformes, *par ex.*, soient deux plateformes pouvant supporter l'exécution de politiques de collecte de données dont l'une est alimentée par un courant secteur et l'autre par une batterie, alors le choix de la plateforme se portera sur la plateforme alimentée par le secteur afin d'économiser l'énergie de l'infrastructure.

Nous considérons que ces choix de projection en fonction de caractéristiques propres à l'infrastructure de capteurs sont également hors du domaine d'expertise de l'ingénieur logiciel. Cependant, la compréhension des actions métiers voulues par un ingénieur logiciel est également hors du domaine d'activité d'un expert réseau. Ce dernier n'a donc pas de visibilité sur l'ordonnancement des actions pour les répartir sur les différentes plateformes. Ainsi, bien que nécessaire, la séparation des préoccupations entre un ingénieur logiciel et un expert réseau est nécessaire afin que chacun reste concentré sur son domaine d'expertise, elle rend la projection des politiques compliquée, car l'un n'a pas de visibilité sur les besoins de l'autre. Une approche de projection automatique des politiques devrait alors être proposée afin de résoudre ce problème.

3.3.2 Permettre l'adaptation automatique aux spécificités matérielles

La politique de collecte de donnée exprimée par l'ingénieur logiciel est indépendante du matériel, *cf.* Sec. 3.2.1. Avant d'être mise en œuvre sur une infrastructure de capteurs, les intentions métier doivent être projetées sur des plateformes, *cf.* Sec. 3.3.1. Toutefois, les actions représentées restent descriptives et ne spécifient pas la façon d'être mise en œuvre. Elles doivent alors être adaptées aux spécificités matérielles de la plateforme.

Par exemple, au sein de la politique fil rouge, il est nécessaire de récupérer la température à partir d'un capteur d'un bureau. Selon le type de capteur de température, la fonction de transformation de la donnée mesurée en température Celsius n'est pas la même. Chaque type de capteur renvoie une tension électrique en fonction de la température ambiante. L'étude de la documentation du capteur nous donne ensuite la fonction de conversion traduisant la valeur mesurée en température. Les listings Lst. 3.2, Lst. 3.3 et Lst. 3.4 présentent, dans un même langage, la mise en œuvre de la fonction de conversion pour trois capteurs différents.

Listing 3.2 – Fonction C permettant de lire la température à partir d'un capteur de température Grove

```
double readTemperature(int _port) {
    int a = analogRead(_port);
    float R = 1023.0/((float)a)-1.0;
    R = 100000.0*R;
    float temperature=1.0/(log(R/100000.0)/4275+1/298.15)-273.15;
    return (double) temperature;
}
```

Listing 3.3 – Fonction C permettant de lire la température à partir d'un capteur de température Electronic Bricks

```
double readTemperature(int _port) {
    int a =analogRead(_port);
    float resistance=(float)(1023-a)*10000/a;
    float temperature=1/(log(resistance/10000)/3975+1/298.15)-273.15;
    return (double) temperature;
}
```

Listing 3.4 – Fonction C permettant de lire la température à partir d'un capteur de température DF Robot

```
double readTemperature(int _port) {
    int a = analogRead(_port);
    return (500 * a)/1024;
}
```

Ici encore, nous considérons que la connaissance du type de matériel embarqué sur les plateformes de capteurs est hors du domaine d'expertise d'un ingénieur logiciel et la connaissance des intentions métier voulues par un ingénieur logiciel est hors du domaine d'expertise d'un expert réseau. De manière similaire à la projection, afin de séparer les préoccupations, une approche automatisée adaptant chaque action aux caractéristiques physiques de la plateforme serait une bonne solution. En ingénierie logicielle, et plus spécifiquement au sein des lignes de produits logiciels, une bonne pratique vise à modéliser les éléments variables du logiciel au sein de modèles de variabilité. Étendre cette notion à la variabilité physique serait ainsi une bonne solution puisqu'un processus de génération de code pourrait utiliser ce modèle de variabilité afin de produire du code adapté à la plateforme considérée.

3.3.3 Adapter le déploiement en fonction des spécifications de l'expert réseau

Le déploiement d'une politique de collecte de données peut impacter le fonctionnement de l'infrastructure de capteurs. Par exemple, une action d'agrégation nécessite la récupération de larges quantités de données à partir des moyens de communication de la plateforme. Le déploiement de cette activité sur une plateforme alimentée par batterie réduira fortement l'autonomie. Nous considérons qu'un ingénieur logiciel n'a pas connaissance des problématiques liées au déploiement de sa politique. La sélection de la plateforme appropriée pour une activité doit alors être effectuée automatiquement. Pour guider ce choix, il est possible de définir une stratégie de déploiement où un expert réseau spécifie des critères d'optimisation (*par ex.*, par rapport au type d'alimentation). Cette stratégie est ensuite utilisée lors de la phase de déploiement pour identifier la plateforme cible satisfaisant au mieux les critères spécifiés.

3.4 Partage de l'infrastructure de capteurs

Nous visons ici à répondre à la problématique « *comment une infrastructure peut-elle être partagée entre plusieurs collectes de données afin d'éviter des déploiements redondants ?* ». Nous avons présenté en Sec. 2.3 des travaux permettant à une infrastructure de capteurs d'être partagée entre plusieurs applications. Pour valider cette problématique, nous nous fixons l'objectif suivant : « (O_4) Une infrastructure est partagée entre différentes politiques de collecte de données ».

Pour répondre à cette problématique, nous adresserons les points présentés ci-après.

3.4.1 Permettre à l'infrastructure de mettre en œuvre plusieurs applications de l'Internet des objets

Nous avons vu en introduction que le coût d'installation et de maintenance d'une infrastructure de capteurs à grande échelle encourage à sa réutilisation lorsque de nouvelles applications sont identifiées. Les techniques de partage d'infrastructure à grande échelle sont aujourd'hui basées sur un principe de réservation de ressources dans le temps [SMG+14, ABF+15], limitant la mise en production d'applications ou sur des techniques de partage de plateforme, mais posant des contraintes sur le type de matériel utilisé, *par ex.*, des plateformes multi-tâches pour la virtualisation ou sur l'homogénéité de l'infrastructure [KBG+16].

Pourtant, un réseau hétérogène doit pouvoir être partagé entre plusieurs applications, indépendamment du type de matériel utilisé. Par exemple, dans le cadre du domaine des bâtiments intelligents, le traitement du fil rouge de prévention de chocs thermiques peut être amené à coexister sur l'infrastructure avec d'autres applications de cartographie thermique ou de surveillance des déperditions énergétiques. De plus, les plateformes doivent être partagées entre plusieurs politiques de collecte de données afin de supporter un passage à l'échelle de l'infrastructure.

En ingénierie logicielle, la composition logicielle est définie comme la « construction d'applications logicielles à partir de composants mettant en œuvre des abstractions pertinentes pour un domaine de problème particulier » [NM95]. Ainsi, en faisait l'hypothèse que la composition de chaque politique de collecte de données (assimilées à une abstraction pertinente pour un problème particulier) forme une nouvelle politique de collecte de données comprenant l'ensemble des intentions métier exprimées dans

chacune des politiques, la politique composée est déployable comme n'importe quelle autre politique de collecte de données.

Hypothèse H_3 (Résultat de la composition) : Le résultat de la composition de deux politiques forme une nouvelle politique de données.

Ce principe offre une bonne solution face aux limitations des travaux de l'état de l'art puisqu'il permet, à travers un redéploiement, de remplacer la politique de collecte de données courante avec une politique de collecte de données contenant des intentions métier pour plusieurs applications.

3.4.2 Permettre le déploiement de nouvelles politiques de collecte de données sur une infrastructure partagée

Les approches de déploiement automatisé permettent à une application d'être déployée sur une infrastructure de capteurs tout en respectant le principe de séparation des préoccupations [PC15]. Cependant, les approches identifiées dans l'état de l'art ne permettent pas le partage automatique de l'infrastructure en *plusieurs* applications *cf.* FIG. 2.4, ou dans notre contexte, plusieurs politiques de collecte de données. Pour répondre à cette limitation, l'ingénieur logiciel devrait fournir à l'approche automatisée le résultat de la composition, *cf.* SEC. 3.4.1, de sa politique avec les autres politiques. Cependant, si les autres politiques ont été mises en œuvre par d'autres ingénieurs logiciels, il ne possède pas leur définition afin de les composer.

Avec les techniques actuelles, le déploiement de la politique fil rouge sur une infrastructure partagée, entrainera le remplacement des politiques précédemment déployées par d'autres ingénieurs logiciels pouvant *in fine* bloquer la collecte de données pour des applications tierces.

Un système intermédiaire *proxy* entre les ingénieurs logiciels et l'infrastructure de capteurs constitue une solution pour répondre à ce problème. En effet, par sa position centrale, il peut conserver une copie de l'ensemble des politiques mis en œuvre sur l'infrastructure. Ainsi, lors du déploiement d'une nouvelle politique, il pourrait composer cette dernière avec l'ensemble des politiques mises en œuvre et déployer le résultat de la composition. Par ailleurs, cette solution n'est pas incompatible avec le passage à l'échelle de l'infrastructure puisqu'elle ne stocke que des politiques et non des données.

3.5 Synthèse

Dans ce chapitre, nous avons explicité les motivations animant cette thèse. Pour chacune d'entre elles, nous avons présenté les limites des techniques actuelles et proposé une piste de solution. Nous récapitulons ci-dessous les différentes pistes que nous allons aborder dans les chapitres CHAP. 4, CHAP. 5 et CHAP. 6 :

▶ ***Séparation des préoccupations***. Afin de laisser l'ingénieur logiciel et l'expert réseau respectivement concentrés sur leurs domaines d'expertises, nous nous attacherons au principe de séparation des préoccupations.

▶ ***Ingénierie dirigée par les modèles***. Les techniques IDM permettent d'exprimer des intentions métier indépendamment des plateformes de l'infrastructure. Ces intentions sont alors réutilisables à travers différentes infrastructures sont avoir à réécrire le code les caractérisant. Ces techniques sont également dotées de langages spécifiques au domaine pour permettre la manipulation de modèles à haut niveau d'abstraction.

▶ ***Lignes de produits logicielles***. Les lignes de produits logiciels permettent, à travers les modèles de variabilité, de modéliser les points variables d'un logiciel. L'extension de cette approche aux plateformes physiques de l'infrastructure, *i.e.*, modéliser les points variables d'une plateforme, permet à des générateurs de code d'adapter le code mettant en œuvre une politique de collecte de données à l'infrastructure cible.

▶ ***Composition logicielle***. La composition logicielle permet d'assembler plusieurs artefacts pour former un logiciel. Les techniques de composition sont une bonne pratique puisqu'elles permettent à n'importe quelle infrastructure reprogrammable d'être partagée entre plusieurs politiques de collecte de données. En effet, d'après l'hypothèse H_3, le résultat de la composition de politique reste une politique de collecte de données.

L'étude des travaux de l'état de l'art nous a montré qu'il n'existe pas d'approche intégrée offrant à la fois un mécanisme d'expression de besoins métiers, de déploiement automatisé et de partage de l'infrastructure de capteurs. A travers les chapitres Chap. 4, Chap. 5 et Chap. 6, nous proposons un ensemble de contributions construisant une approche intégrée (de la définition de la politique à son déploiement sur une infrastructure partagée) et qui répond aux problématiques identifiées. En Chap. 7, nous présentons comment ces contributions valident les objectifs présentés.

Chapitre 4

Modélisation et mise en œuvre de politiques de collecte de données

Sommaire

4.1 Introduction

Lors de la conception d'une application basée sur l'Internet des Objets, l'ingénieur logiciel est amené à définir la procédure de collecte de données ainsi que les traitements à leur appliquer afin de les rendre utilisables dans son contexte métier. Il peut notamment déléguer certains traitements aux plateformes déployées dans le réseau de capteurs afin d'effectuer des opérations sur les données avant même qu'elles soient collectées par l'application. Pour cela, de nombreux langages de programmation génériques permettent de concevoir un *micrologiciel*, *par ex.*, C ou Lua, embarqués sur ces plateformes et traduisant les opérations souhaitées.

4.1.1 Problèmes identifiés

En manipulant des langages bas-niveau, les ingénieurs logiciels doivent manipuler des concepts propres aux infrastructures de capteurs qui sont hors de leur domaine d'expertise. Ils doivent en particulier gérer les problématiques liées au réseau, à l'énergie et aux faibles capacités de calcul des plateformes. De plus, en utilisant de tels langages, les programmes présentent l'inconvénient d'être spécifiques à une plateforme, empêchant leur réutilisation dans un autre contexte ou sur une infrastructure différente. Bien que certaines approches permettent de monter en abstraction, elles restent spécifiques à une architecture et ne permettent pas la réutilisation partielle et/ou totale des applications développées.

4.1.2 Objectifs de ce chapitre

Ce chapitre cible l'objectif (O_1) :

(O_1) Un ingénieur logiciel ne manipule que des concepts métiers

Les travaux d'état de l'art, *cf.* Sec. 2.1, nous montrent que l'infrastructure de capteurs doit être configurée au niveau matériel en fonction des besoins des ingénieurs logiciels. Ainsi, ils doivent comprendre l'infrastructure sous-jacente et utiliser des langages de programmation de bas niveau. Si des abstractions existent, elles restent toutefois limitées à une même architecture de plateforme ou d'infrastructure, réduisant leur applicabilité à des infrastructures hétérogènes. Afin que les ingénieurs logiciels restent focalisés sur leur domaine d'expertise, ils doivent utiliser une approche à *haut-niveau d'abstraction* et *indépendante du matériel* pour exprimer des politiques de collecte de données. Une telle abstraction permet également à un ingénieur logiciel de réutiliser sa politique dans différentes infrastructures de capteurs, car son code n'est plus couplé à un réseau spécifique de capteurs.

4.1.3 Plan du chapitre

Dans une première partie Sec. 4.2, nous présentons un méta-modèle permettant l'expression à haut-niveau d'abstraction de politiques de collecte de données. Plus particulièrement, nous effectuons une formalisation des politiques de collecte de données et présentons un support architectural à leur définition. Ensuite, dans une seconde partie Sec. 4.3, nous présentons des mécanismes permettant la réutilisation entière ou partielle d'une politique de collecte de données. Ces mécanismes sont formalisés et mis en

œuvre au sein d'un langage spécifique au domaine afin d'être directement utilisables par des ingénieurs logiciels. Enfin, une dernière partie Sec. 4.4 conclue ce chapitre.

Contributions présentées

Ce chapitre présente les contributions suivantes :

- Un méta-modèle permettant de représenter, indépendamment du matériel sous-jacent, des transformations à appliquer sur les données issues de capteurs ;
- Un langage spécifique au domaine permettant d'exprimer à haut-niveau d'abstraction des politiques de collecte de données ;
- Des mécanismes de réutilisation partielle ou totale de politiques de collecte de données directement utilisables par un ingénieur logiciel.

4.2 Expression des besoins de l'ingénieur logiciel

4.2.1 Problèmes

Les infrastructures de capteurs sont aujourd'hui programmées en utilisant des langages bas-niveau mélangeant le code mettant en œuvre la logique métier et le code propre au fonctionnement de la plateforme et des interactions réseau, tâches qui sont hors du domaine métier de l'ingénieur logiciel. De plus, l'utilisation d'un langage bas niveau produit un code difficile à appréhender pour une personne tierce et compliqué à maintenir. Afin de permettre aux ingénieurs logiciels d'exprimer des politiques de collecte de données, un langage à haut niveau doit leur être proposé. Ce langage doit proposer uniquement des instructions permettant de définir la collecte de données.

Le nombre de plateformes de capteurs et de technologies disponibles sur le marché font que les codes produits à ce jour sont spécifiques à un environnement d'exécution. Les abstractions proposées par les systèmes d'exploitation embarqués sont, de plus, uniquement compatibles avec les plateformes ayant la possibilité d'exécuter le système d'exploitation. Afin de permettre à un ingénieur logiciel d'exprimer un programme sur tout type d'infrastructure de capteurs, le langage proposé le langage devrait donc être indépendant du matériel.

4.2.2 Les politiques de collecte de données comme abstraction à la complexité

Chaque capteur présent dans une infrastructure de capteurs collecte des informations brutes par rapport à l'environnement dans lequel il est déployé. Afin de répondre aux problématiques métiers des ingénieurs logiciels, ces données doivent être traitées au sein de **politiques de collecte de données**.

Définition 4.2.1 (Politique de collecte de données)**.** Un ensemble d'opérations effectuées sur des données brutes afin de les convertir en données à valeur ajoutée [GBMP13]

4.2.3 Expression d'une politique de collecte de données

Au sein des infrastructures de capteurs, deux représentations sont couramment utilisées pour modéliser des transformations sur les données : les *machines à états finis* et les *processus métiers*.

▶ ***Machines à états finis***. Les machines à états finis sont couramment utilisées pour modéliser des protocoles de communication ou pour définir les interfaces de connexion de composants dans le cadre d'applications développées en utilisant une approche par composants [VMD+05].

▶ ***Processus métier***. Les flux de travaux [VDAVH04], assimilés à des processus métiers [vdAtHW03], se focalisent sur des activités orientées métier plutôt que sur des tâches opérationnelles. Parmi de nombreux champs d'application, ils ont été largement adoptés par la communauté scientifique [TDGS14]. Les processus métiers permettent à des scientifiques de construire des séquences de taches permettant d'effectuer des transformations sur de grands jeux de données. Par exemple, en juillet 2017, la plateforme myExpermiment comptait plus de 3800 processus métiers et 10500 utilisateurs référencés.

Les processus métiers sont ainsi particulièrement adaptés pour définir une politique de collecte de données. De plus, ils permettent de définir une séquence d'activités permettant de réaliser un besoin métier [Kru04].

Définitions et formalisation d'une politique

À partir de la définition d'un processus métier, nous formalisons une politique de collecte de données comme étant un ensemble d'**activités** et de **flux de données**.

▶ ***Activité***. Une activité $a \in A$ représente une action effectuée sur les données. Les actions peuvent être liées à de l'acquisition/émission de données aux interfaces de données ou des opérations de transformation sur les données. Nous regroupons donc les activités en deux catégories : *(i)* des interfaces de données (capteur ou collecteur) et *(ii)* opérations.

Définition 4.2.2 (Capteur)**.** Dispositif électronique traduisant un phénomène physique en une mesure électrique. Un capteur peut avoir un fonctionnement périodique ou évènementiel.

Définition 4.2.3 (Collecteur)**.** Système distant où sont envoyées les données pour stockage.

Une activité modélise une fonction à effectuer sur données de capteurs. Elle consomme $n_i \in \mathbb{N}^\star$ données de capteurs et produit $n_o \in \mathbb{N}^\star$ données de capteurs à travers des ports. Les ports réutilisent la notion d'interface introduite dans la programmation par composants [SBW99], *cf.* exemple gauche de la Fig. 4.1.

Définition 4.2.4 (Port)**.** Un port d'entrée (resp. sortie) d'activité est une interface permettant à l'activité de consommer (resp. de produire) des données de capteurs.

Définition 4.2.5 (Donnée de capteurs)**.** Une donnée de capteur d est définie comme une paire $\langle measure, timestamp \rangle$ où $measure$ correspond à la valeur électrique du phénomène physique mesurée et $timestamp$ l'horodatage associé à l'acquisition de cette mesure. Cette notion est abordée plus en détail en Sec. 4.2.3.

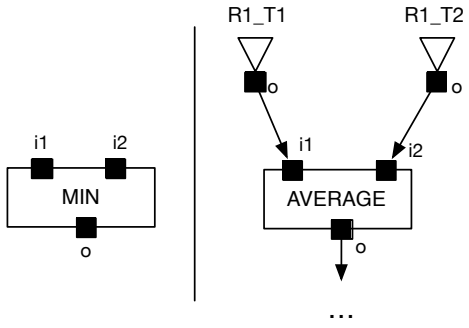


FIGURE 4.1 – Exemple d'activité avec 2 ports d'entrée $i1$ et $i2$ et 1 port de sortie o *(gauche)* et exemple de flux entre deux capteurs `R1_T1`, `R1_T2` et une activité `AVERAGE` *(droite)*

Pour une activité a, nous utiliserons les opérateurs $inputs(a)$ et $outputs(a)$ renvoyant respectivement un ensemble contenant les ports d'entrée et un ensemble contenant les portes de sortie de l'activité. Nous faisons un choix de conception faisant que chaque port d'activité ne supporte qu'une donnée de capteur à la fois, *i.e.*, il n'y a pas de stockage temporaire au niveau des ports d'activité. Ainsi, pour un port p, l'opérateur $data(p)$ désignera la donnée de capteurs disponible sur le port p.

▶ ***Flux de données***. Un flux de données $f \in F$ représente le flux de données entre deux activités *différentes*. Si a et b sont deux activités différentes et o_a, i_b respectivement un port de sortie de a et un port d'entrée de b alors $flow(o_a, i_b)$ traduit le flux de données entre ces deux ports. Nous utiliserons les notations $flow.source$ et $flow.destination$ comme pointeurs respectifs vers les interfaces de sortie et d'entrée des activités connectées par le flux $flow$. L'exemple présenté à droite sur la figure FIG. 4.1 illustre deux flux de données entre des activités différentes.

Hypothèse : Absence de cycle

Les politiques de collecte de données n'ont pas vocation à réguler une plateforme de capteur ou un système extérieur à travers des boucles de rétroaction, *i.e.*, une politique ne définit qu'une collecte de données. Ainsi, les politiques ne contiennent pas de cycle.

Définition 4.2.6 (Politique de collecte de données)**.** Nous définissions une politique de collecte de données $\pi \in \Pi$ comme un triplet $\langle name, activities, flows \rangle$ où $name$ est le nom de la politique, $activities$ un ensemble d'activités et $flows$ un ensemble de flux de données.

Nous introduisons un certain nombre d'opérateurs pour accéder aux différents composants d'une politique de collecte de données π :

— $name(\pi)$ retourne le nom de la politique π ;
— $activities(\pi)$ retourne la liste des activités présentes dans π ;
— $operations(\pi)$ retourne la liste des opérations présentes dans π ;
— $sensors(\pi)$ retourne la liste des capteurs présents dans π ;
— $collectors(\pi)$ retourne la liste des collecteurs présents dans π ;
— $joinpoints(\pi)$ retourne la liste des points d'extension présents dans π. Ils seront introduits en SEC. 4.3.3 ;
— $flows(\pi)$ retourne les liste des flux de données présents dans π.

D'après les définitions énoncées précédemment, on remarquera les relations ensemblistes : $operations(\pi) \subset activities(\pi)$, $sensors(\pi) \subset activities(\pi)$, $collectors(\pi) \subset activities(\pi)$, $joinpoints(\pi) \subset activities(\pi)$. À partir de ces définitions, nous pouvons également déduire la relation générale Eq. 4.1 entre les activités :

$$operations(\pi) \cup sensors(\pi) \cup collectors(\pi) \cup joinpoints(\pi) = activities(\pi) \quad (4.1)$$

Données de capteurs

Hypothèse : Temps courant

Le temps courant est accessible sur l'ensemble des plateformes de capteurs

► ***Synchronisme***. Les activités contenant au moins deux ports d'entrée mettent en relation au moins deux données de capteurs en vue de prendre une décision ou d'effectuer une action de transformation sur ces données.

Définition 4.2.7 (Synchronisme)**.** Des données de capteurs synchronisées sont des données de capteurs temporellement corrélées, *i.e.*, produites dans un même intervalle de temps

Si la politique de collecte de données contient n capteurs périodiques de périodes $P_1, \ldots, P_n$ alors un ensemble de m données de capteurs sera considéré comme synchronisé si la différence en valeur absolue de chacun des horodatages $t_1, \ldots, t_m$ est strictement inférieure au $PPCM(P_1, \ldots, P_n)$. La valeur du PPCM des périodes nous indiquera la période de décalage maximal autorisée, notée π_{offset}. Cette notion de synchronisme nous permet notamment de ne pas exécuter les activités avec des données qui ne seraient plus à jour suite à une panne du réseau de capteur par exemple avec un capteur défectueux ou une plateforme hors couverture réseau.

Synchronisme

Prenons l'exemple de trois capteurs périodiques $C1$ (période = 60 secondes), $C2$ (période = 90 secondes), $C3$ (période = 120 secondes) et un capteur évènementiel C'. Le PPCM des périodes est égal à $\pi_{\text{offset}} = 360$.

Tableau 4.1 – Exemple d'horodatages de données de capteur

	C1	**C2**	**C3**	**C'**	**Syn**
t_0	40	30	100	10	✓
t_1	100	120	220	10	✓
t_2	160	210	340	10	✓
t_3	220	300	460	10	✓
t_4	**220**	390	580	300	✗
t_5	**220**	480	700	300	✗

Le tableau ci-dessus présente l'horodatage associé à des données de capteurs reçues en entrée d'une politique de collecte de données. Entre les instants t_0 et t_3 les données sont synchronisées puisque l'écart entre chaque horodatage reste strictement inférieur à la valeur du PPCM des périodes. Nous simulons après l'instant t_3 une panne du capteur $C1$. Dès lors, nous détectons en t_4 et t_5 que les

données ne sont plus synchronisées ($|580 - 220| \geq \pi_{\text{offset}}$ et $|700 - 220| \geq \pi_{\text{offset}}$) puisque l'activité est alimentée avec une valeur obsolète de $C1$.

Nous **excluons** les valeurs de capteurs évènementiels du calcul du synchronisme puisque ces données restent valides tant qu'un nouvel évènement n'a pas été produit. Ainsi, la valeur renvoyée par un capteur évènementiel correspondra à la dernière valeur produite par ce capteur horodatée du temps courant.

▶ ***Types***. Au sein d'une politique, chaque activité gère un unique type de données. Les types de données peuvent être simples ou complexes. Les types simples représentent un format de donnée contenant un unique champ de valeur, *par ex.*, *Double*, *Integer*, *Long* ou *String*. Un type de données complexe contient un ensemble de types simples référencés par un identifiant.

Définition 4.2.8 (Type de données)**.** Un type de données est une représentation atomique ou structurée des données émises par le capteur.

Les types de données complexes contiennent un champ de *valeur* sur lequel les opérations seront appliquées. Dans la suite, nous utiliserons la notation $T.valeur$ pour référencer le champ de *valeur* du type T et la notation $T.k$ fera référence au champ k du type de données T. Les formats de données standardisés peuvent ainsi être modélisés avec un type complexe comme effectué dans l'approche SenML [JAS12]. Par exemple, le type de données utilisés dans le projet SmartCampus [CJMR14] est modélisé comme un type composé ($n \rightarrow$ StringType, $v \rightarrow$ IntegerType, $t \rightarrow$ LongType) où n correspond à l'identifiant du capteur, v à la valeur produite par le capteur et t à la date d'émission de la valeur.

Fil rouge : Données reçues et attendues

Les valeurs produites par les capteurs de température utilisent une adaptation du format de données SenML [JAS12] où le nom du capteur et la valeur mesurée sont encodés respectivement dans un champ nommé `n` de type *StringType* et un champ nommé `v` et de type *DoubleType*. Nous représentons ainsi ce type de donnée sous forme composée ($n \rightarrow$ StringType, $v \rightarrow$ DoubleType) où la première paire correspond au nom du capteur et la seconde paire à la valeur en degrés Celsius mesurée par le capteur, *i.e.*, `("T_OUTSIDE", 23.4)`. Nous définissons v comme champ de valeur.

Au sein de la politique de collecte de donnée, ces valeurs sont sous le format d'une *donnée de capteur* grâce à l'ajout de l'horodatage associé à l'acquisition de la mesure, *i.e.*, `(("T_OUTSIDE", 23.4), 1488190224)`.

Lorsque le seuil recommandé est dépassé pour un bureau, le collecteur doit recevoir des valeurs de capteur de type `AlertMessageType` qui est un type composé ($alert \rightarrow$ StringType, $room \rightarrow$ StringType), *par ex.*, `(("ALERT_TEMP", "R_1"), 1488190224)` où `"R_1"` correspond à un identifiant de bureau.

Validité d'une politique

En vue de traduire les politiques de collecte de données sur un réseau de capteurs (*cf.* Chap. 5), elles doivent satisfaire certaines propriétés pour être considérées comme **valides**.

▶ ***Validité structurelle***. Une politique de collecte de données est nourrie à partir de données provenant de capteurs. Ainsi, elle doit contenir au minimum un capteur :

$$sensors(\pi) \neq \emptyset \tag{4.2}$$

Après l'application des opérations sur les données, la politique envoie le résultat à un collecteur distant. Dès lors, la politique doit contenir au minimum un collecteur :

$$collectors(\pi) \neq \emptyset \tag{4.3}$$

Les activités utilisent les données présentes sur leur port d'entrée en vue de fournir un résultat. Les données étant transmises par les flux, chaque port d'entrée d'une activité a contenue dans une politique π doit être connecté à un flux de données :

$$\forall a \in activities(\pi), \forall i \in inputs(a), \exists f \in flows(\pi), f.destination = i \tag{4.4}$$

▶ ***Validité comportementale***. Les ports d'entrée d'activités ne supportent qu'une unique donnée de capteur à la fois. Ainsi, un port d'entrée d'une activité a contenue dans une politique π doit être connecté à un **unique** flux de données :

$$\forall a \in activities(\pi), \forall i \in inputs(a), \exists! f \in flows(\pi) f.destination = i \tag{4.5}$$

Les politiques de collecte de données n'ayant pas vocation à réguler une plateforme de capteur à travers des boucles de rétroaction intégrées, elles sont **a-cycliques**, *i.e.*, il n'existe pas de chemin d'une activité à elle-même et les flux acheminent les données entre des activités différentes :

$$\forall f \in \pi.flows, f.destination \neq f.source \tag{4.6}$$

Activités proposées à l'ingénieur logiciel

Classiquement, les activités contenues dans un processus métier contiennent du code spécifique à la plateforme sur laquelle il sera exécuté. Afin d'être indépendants du matériel, nous proposons un ensemble d'activités génériques, *c.à.d*, décrivant l'opération à effectuer et non la manière de l'effectuer.

Afin de proposer un ensemble d'activités cohérent pour l'expression de besoins métiers, nous avons défini et catégorisé, à partir de différents langages d'expression de transformation sur les données, un ensemble d'activités nécessaires à la définition de politiques de collecte de données :

▶ ***Interfaces d'entrée de données***. Ces activités possèdent une sortie o et alimentent la politique avec des données en provenance de capteurs périodiques ou évènementiels ou de politiques tierces.

▶ ***Interface de sortie de données***. Ces activités possèdent une entrée i et permettent d'envoyer les données de capteurs traitées par la politique vers un collecteur ou de politiques tierces.

Les interfaces d'entrée et sortie de données s'inspirent des `IOProcessor` du langage GWENDIA [GMB+09]

▶ ***Opérations fonctionnelles***. Ces opérations possèdent n entrées i_n $(n > 0)$ et une sortie o. Soit f la fonction modélisée par l'opération et I l'ensemble des entrées

(*i.e.*, les opérandes) alors le résultat de l'opération correspondra à l'application de la fonction f sur les entrées telles que $o = f(i_1, \ldots, i_n)_{i \in I}$.

Les opérations fonctionnelles s'inspirent des *Opérateurs Math* du langage graphique ArduBlock et des *functions* du langage de flux de données Node-RED.

▶ ***Opération conditionnelle***. Certaines données produites par les capteurs peuvent ne pas répondre aux besoins de la politique de collecte de données. L'opération `Condition` permet de rediriger la valeur produite par le capteur sur une sortie spécifique, déterminée par la vérification d'un prédicat. Cette opération possède une entrée i, deux sorties o_{then} et o_{else} et un prédicat p. Cette opération redirige les données présentées en entrée sur l'une des deux sorties en fonction de la satisfiabilité du prédicat p : $p(i) \implies o_{then} = i$ et $\neg p(i) \implies o_{else} = i$. Cette opération s'inspire de l'activité `Conditional` du langage GWENDIA.

▶ ***Opération d'extraction***. L'opération `Extract` permet d'extraire un champ d'un type de données composé. Cette opération a une entrée i (de type T) et une sortie o (de type T') et est paramétrée par la clé k du champ $T.k$ à sélectionner. La sortie o sera du type de $T.k$ et $T.k$ sera recopiée sur o. Cette opération s'inspire de l'opération d'extraction du moteur YAWL [VDATH05] et de la fonction *change* de Node-RED.

▶ ***Opération de production***. L'opération `Produce` permet de produire une donnée de capteur sous le format de données T' lorsqu'une donnée est reçue sur chacun des ports d'entrée. Cette opération a une entrée i (de type T), et une sorties o de type T' correspondant au message produit.

Nous présentons sur la figure FIG. 4.2 (page 62) les prototypes des activités offertes à l'ingénieur logiciel.

Fil rouge : Politique de collecte de données

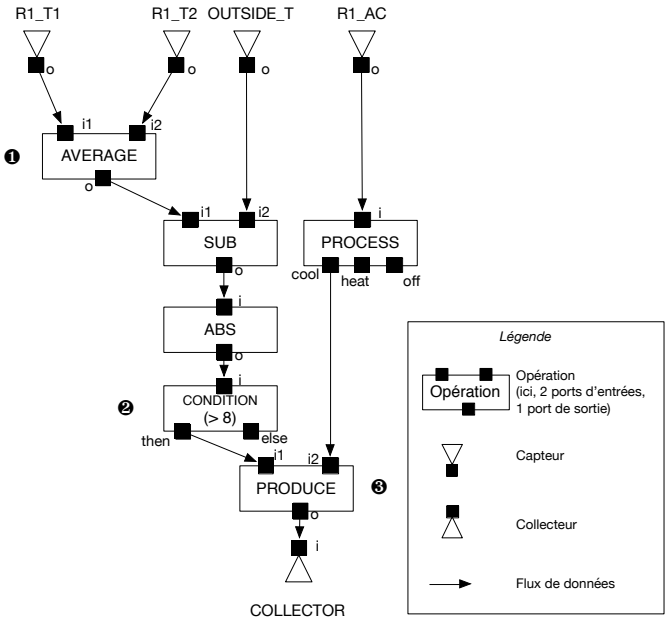


FIGURE 4.3 – Détection de possibles chocs thermiques

Dans cet exemple, nous illustrons la mise en œuvre de l'exemple fil rouge au sein d'un bureau du bâtiment intelligent. Dans un premier temps, la politique effectue une moyenne de la température du bureau à partir des deux capteurs de température ambiante ❶. La différence en valeur absolue par rapport à la tem-

pérature extérieure est ensuite comparée par rapport au seuil recommandé ❷ (8 degrés). Enfin si le seuil est dépassé et que la climatisation est en fonctionnement, un message d'alerte ❸ sera envoyé au collecteur sur lequel l'application est connectée.

Dans cet exemple, l'opération `PRODUCE` produit une donnée de capteur avec un message de type composé correspondant à l'alerte attendue par l'application, *par ex.*, `((ALERT_TEMP, R_2), 1488190224)`. L'opération `PROCESS` permet quant à elle d'encapsuler une politique déjà existante et sera présentée plus en détail en Sec. 4.3.2.

4.2.4 Un langage spécifique au domaine comme support à l'expression des politiques

Au sein du chapitre Chap. 2, nous avons identifié les langages spécifiques au domaine comme étant adaptés à l'expression de besoin métiers à haut-niveau. Ainsi, afin de faciliter l'expression par un ingénieur logiciel de politiques de collecte de données, nous proposons une approche d'ingénierie dirigée par les modèles en définissant un méta-modèle permettant de construire une politique de collecte de données utilisant les activités et les flux de données décrits en Sec. 4.2.3 et un langage spécifique au domaine permettant de manipuler à haut-niveau d'abstraction les éléments du méta-modèle.

La définition d'une politique de collecte de données s'effectue en quatre temps.

Dans un premier temps, l'ingénieur logiciel renseigne les propriétés de la politique telles que son nom (*hasForName*), le type de données utilisé (*handles* ainsi que la valeur offset acceptée (*offset*). Si aucune valeur *offset* n'est spécifiée, le PPCM de la période des capteurs périodiques déclarés sera utilisé.

```
(this PROPERTY)*

PROPERTY: hasForName STRING |
          handles DATATYPE  |
          offset  INT
```

Listing 4.1 – Déclaration des propriétés de la politique

Dans un second temps, les capteurs et collecteurs (`PeriodicSensor`, `EventSensor` et `Collector` sur Fig. 4.4) utilisés doivent être déclarés selon la grammaire présentée en Lst. 4.2.

```
declare DATAINTERFACE
DATAINTERFACE: (aPeriodicSensor() withPeriod INT | anEventSensor() |
                aCollector()) named STRING
```

Listing 4.2 – Déclaration de capteurs et collecteurs

Dans un troisième temps, l'ingénieur logiciel défini (*define*) les opérations (`Operation` sur Fig. 4.4) souhaitées selon la grammaire présentée en Lst. 4.6. Il est également possible de renommer le résultat d'une opération grâce à l'instruction *andRenameTo* (*par ex.*, le résultat de l'addition de deux données en provenance de capteurs `C1` et `C2` peut être renommé en `ADD_C1_C2` pour plus de lisibilité)

```
define OPERATION | PROCESS

OPERATION: UNARY | ((ARITHMETIC | CONDITION | COMPARING | EXTRACTING |
           PRODUCE) PORTS) | RENAME?

UNARY: anIncrementBy(INT) | aDecrementBy(INT) | anAbsoluteValue()
```

```
ARITHMETIC: anAdder() | aSubstractor() | aMultiplier() | aDivider() | anAvg()

CONDITION: aCondition(STRING)

COMPARING: anEqual() | aDifferent() | aGreatherEq() | aGreather() |
                   aLowerEq() | aLower() | aMin() | aMax()

EXTRACTING: anExtractor(STRING)

PRODUCE: aProducer(DATATYPE)

PORTS: (withInputs(STRING*) | withOutputs(STRING*))
       (handlingOnInputs(DATATYPE) handlingOnOutputs(DATATYPE) |
       handling(DATATYPE))?

RENAME: andRenameTo(String)

PROCESS: aProcess(OTHER_POLICY)
```

Listing 4.3 – Définition des opérations à appliquer sur les données

Enfin, dans un quatrième et dernier temps, les flux de données entre les différentes activités sont déclarés (`Flow` sur FIG. 4.4). Un flux de données s'exprime sous la forme $activity_a(output\_port) \rightarrow activity_b(input\_port)$. Dans le cas où l'activité ne possède qu'un unique port d'entrée ou de sortie, nous permettons l'omission des paramètres *output_port* et *input_port*.

Fil rouge : Mise en œuvre utilisant le DSL

Le code ci-dessous représente la mise en œuvre de la politique fil rouge. Le type `TemperatureSensorDataType` représente des données de capteur de type composé ($n \rightarrow$ StringType, $v \rightarrow$ DoubleType).

Listing 4.4 – Déclaration des propriétés du fil rouge

```
1  this hasForName "ThermalShockPrevention"
2  this handles classOf[TemperatureSensorDataType]
3
4  val r1_t1 = declare aPeriodicSensor() withPeriod 300 named "R1_T1"
5  val r1_t2 = declare aPeriodicSensor() withPeriod 300 named "R1_T2"
6  val outside_t = declare aPeriodicSensor() withPeriod 3600 named "OUTSIDE_T"
7  val r1_ac = declare aPeriodicSensor() withPeriod 60 named "R1_AC"
8  val collector = declare aCollector() named "COLLECTOR"
9
10 val avg = define anAvg() withInputs("i1", "i2")
11 val sub = define aSubstractor() withInputs("i1", "i2")
12 val abs = define anAbsoluteValue()
13 val condiiton = define aCondition("v␣>␣8")
14 val produce = define aProducer(new AlertMessageType("ALERT_TEMP", "R_1"))
15                  withInputs("i1", "i2")
16 val process = define aProcess(ACStatusPolicy())
17
18 flows {
19     r1_t1() -> avg("i1"); r1_t2() -> avg("i2")
20     avg() -> sub("i1"); outside_t() -> sub("i2")
21     sub() -> abs()
22     abs() -> condition()
23     r1_ac() -> process()
24     condition("then") -> produce("i1"); process("cool") -> produce("i2")
25     produce() -> collector()
26 }
```

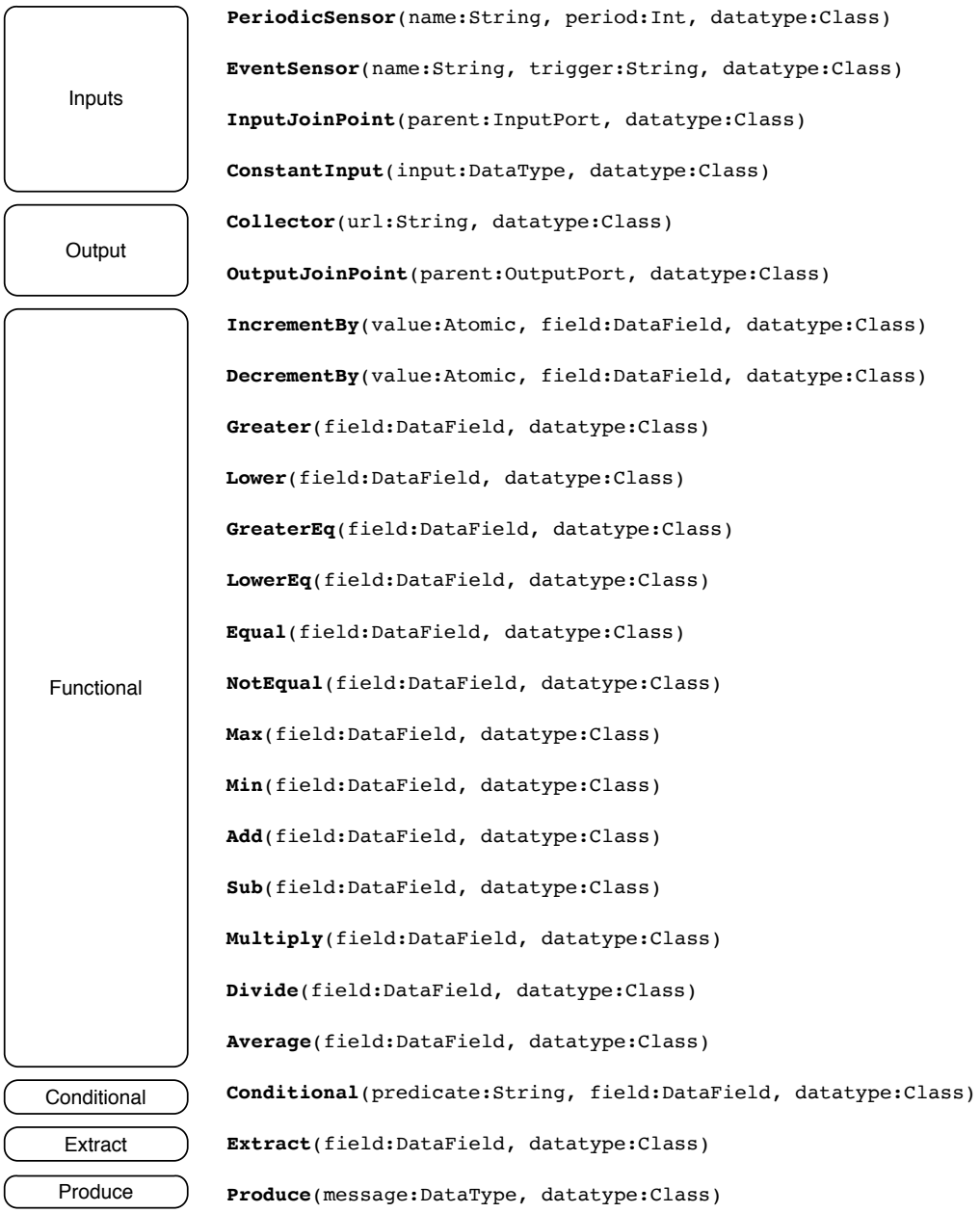


FIGURE 4.2 – Prototypes des activités proposées à l'ingénieur logiciel

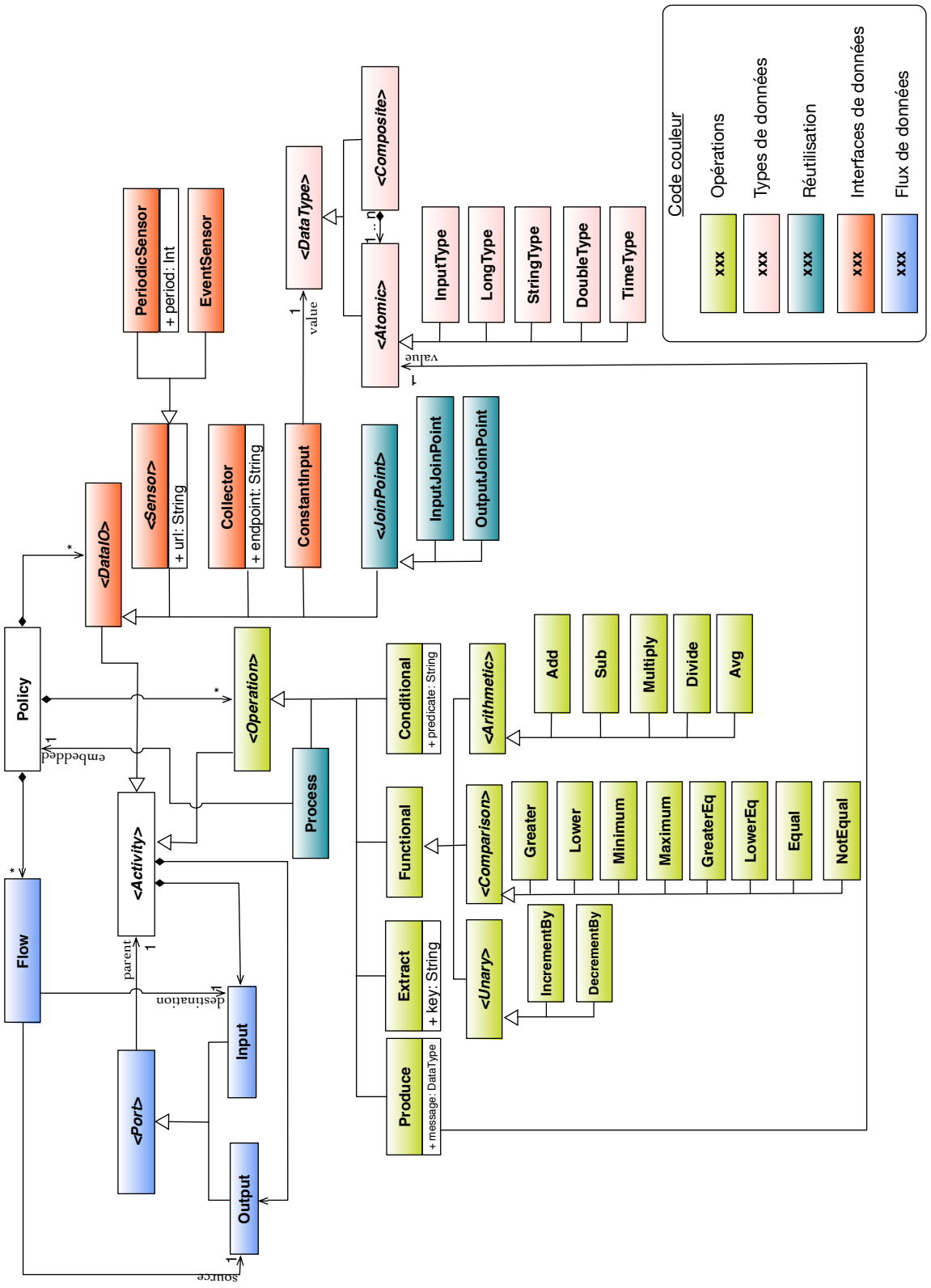


FIGURE 4.4 – Organisation du méta-modèle permettant la construction d'une politique de collecte de données

4.3 Réutilisation à haut niveau des politiques de collecte de données

Un ingénieur logiciel est susceptible de définir une politique de collecte de données à grande échelle qui doit réutiliser une politique définie précédemment dans un autre contexte ou à plus petite échelle, *par ex.*, le traitement appliqué sur les données d'un bureau doit être réutilisable à l'ensemble des bureaux similaires dans un bâtiment intelligent. Afin de respecter les bonnes pratiques d'ingénierie logicielle, l'ingénieur logiciel ne devrait pas à avoir à réécrire toute une politique pour répondre à ses exigences. De plus, une politique définie précédemment peut intégrer un traitement de données qui satisfait la plupart des besoins d'un ingénieur logiciel. Encore une fois, il devrait pouvoir isoler ce traitement pour l'intégrer dans sa politique sans avoir besoin de le redéfinir.

4.3.1 Problèmes liés à la réutilisation

L'état de l'art (Chap. 2) nous montre que la réutilisation au niveau de l'infrastructure est dépendant du matériel embarqué sur les plateformes de capteurs. Ainsi, une politique de collecte de données définie pour un type de plateforme ne pourra être réutilisée que sur une plateforme de même type. De plus, le support de la réutilisation à haut niveau est absent des approches étudiées : l'ingénieur logiciel doit fournir lui-même un composant réutilisable ou une bibliothèque logicielle à un autre utilisateur.

4.3.2 Réutilisation d'une politique entière

Principe de la hiérarchie

La hiérarchie a pour objectif d'offrir à l'ingénieur logiciel un mécamisme supportant la réutilisation d'une politique de collecte de données entière. Elle est notamment utilisée dans les processus métiers exécutés par les moteurs Kepler [ABJ+04] et YAWL [VDATH05]. Ils permettent l'encapsulation et le fonctionnement en boite noire d'une processus métier à l'intérieur d'un autre processus métier.

Introduction de l'activité `PROCESS`

Afin de permettre la formulation d'une politique de collecte de données réutilisant une politique précédemment exprimée dans un autre contexte ou à une plus petite échelle, nous introduisons l'activité `PROCESS`.

Une politique de collecte π est automatiquement transformée en activité `PROCESS` en deux étapes :

1. Une politique de collecte de données intermédiaire π' est créée à partir de la politique π tout en supprimant les capteurs et collecteurs ainsi que les flux de données rattachés à ces derniers (afin d'isoler le traitement des données de leurs sources et destinations initiales).
2. La politique nouvellement créée est encapsulée au sein d'une activité `PROCESS`. Cette activité a le même nombre de ports d'entrée que de capteurs différents présents dans π et le même nombre de ports de sortie que le nombre de collecteurs différents dans π.

4.3.3 Réutilisation partielle d'une politique

Principe de la sélection d'activités

La sélection d'activités a pour objectif d'offrir à l'ingénieur logiciel un mécanisme supportant l'extraction d'un traitement de données pertinent depuis une politique tierce et son intégration dans une politique de collecte de données en cours de construction. De tels mécanismes sont présents dans la manipulation de modèles, *par ex.*, l'élagage du méta-modèle. Cette classe d'algorithme identifie le sous-ensemble d'éléments liés à un élément donné et procède à l'extraction de l'élément et des éléments connexes.

Introduction de l'opérateur de sélection σ

Nous introduisons un opérateur de sélection σ permettant l'extraction d'activités dans une politique de collecte de données déjà existante selon les principes de *pruning* [SMBJ09].

$$\sigma : A \times \Pi \rightarrow \Pi \tag{4.7}$$

Ainsi, à partir d'une liste d'activités, cet opérateur construit automatiquement une nouvelle politique de données ne contenant que les activités souhaitées par l'ingénieur logiciel.

Considérons une politique de collecte de données π et $A' \subset activities(\pi)$ l'ensemble des activités sélectionnées dans π. Si deux activités $(a_1, a_2) \in A'$ sont reliées par un flux de données f dans π, alors l'opérateur σ conservera ce flux. Soit F' l'ensemble des flux de données qui seront conservés par la réutilisation sélective $F' = \forall f \in flows(\pi), f.source \in A' \wedge f.destination \in A'$.

Ainsi, la politique de collecte de donnée résultante de la sélection est formalisée comme suit :

$$\sigma(A', \pi) = \langle name, A', \forall f \in flows(\pi), f.source \in A' \wedge f.destination \in A' \rangle$$

L'opérateur de sélection ne contraint pas l'ingénieur logiciel à choisir des activités de façon à ce que la politique résultante soit valide au sens des propriétés définies en Sec. 4.2.3. Par exemple, si l'ingénieur logiciel sélectionne deux activités non reliées par un flux, alors la politique de collecte de données ne sera pas connexe. Ce choix de conception permet de supporter les étapes intermédiaires de création d'une nouvelle politique de collecte de données. Pour rendre cette politique valide, l'ingénieur logiciel devra la compléter avec des activités et flux de données.

Principe de l'intégration du résultat de la sélection dans une autre politique

La sélection permet d'extraire à une politique de collecte de données des traitements standardisés ou pertinents pour une utilisation dans un contexte différent. Dans le cadre de la programmation par aspects, les techniques de tissage (*weaving*) permettent, au sein d'un code métier, d'insérer des greffons (*advices*). Ces greffons s'insèrent au niveau de points de jonction. Nous proposons de reprendre ces concepts apportés par la programmation par aspects afin de permettre à un ingénieur logiciel de *tisser* le résultat d'une sélection au niveau de points d'extensions d'une politique de collecte de données.

Points d'extension d'une politique

Les points d'extension d'une politique de collecte de données sont rattachés aux ports d'une activité et représentent les endroits sur lesquelles il est possible d'effectuer une opération de tissage.

Définition 4.3.1 (Point d'extension)**.** Un point d'extension $\epsilon \in E$ agit comme une source ou destination d'une politique de collecte de données où il est possible de connecter une autre politique de collecte de données grâce à une opération de tissage. Un point d'extension de sortie peut se connecter sur un point d'extension d'entrée et réciproquement.

Nous utilisons la notation pointée *e.origin* afin de faire référence au port de l'activité étendu par le point d'extension.

La création des points d'extension aux ports des activités est automatisé grâce à la procédure d'extension : elle ajoute automatiquement un *point d'extension* **entrant** à chaque port d'entrée non connecté par un flux de donnée (par égard à la propriété de validité 4.5 relative à l'unicité du flux de donnée par port d'activité d'entrée) et un *point d'extension* **sortant** à chaque port de sortie. De manière réciproque, la procédure de *factorisation* supprime l'ensemble des points d'extension.

Le code Lst. 4.5 présente le mécanisme interne à la procédure d'extension :

Listing 4.5 – Mécanisme interne à l'extension d'une politique de collecte de données

```
def extend(p:DataCollectionPolicy){
    for (activity: activities(p)){
        for(input: inputs(activity)){
            val joinPointInput = new JoinPointInput()
            p.addActivity(joinPointInput)
            p.addFlow(new Flow(jointPointInput.output, input))
        }
        for(output : outputs(activity)){
            if (flows(p).find(f.source == output).isEmpty())
            {
                val joinPointOutput = new JoinPointOutput()
                p.addActivity(joinPointOutput)
                p.addFlow(new Flow(output, joinPointOutput.input))
            }
        }
    }
}
```

Fil rouge : Extension

La figure ci-dessous illustre l'application de la procédure d'extension sur la politique de collecte de données résultante de la sélection. Tous les ports de sorties sont étendus par un point d'extension de sortie (JO_x) et tous les ports d'entrées non connectés à un flux de données sont étendus par un point d'extension d'entrée (JI_x).

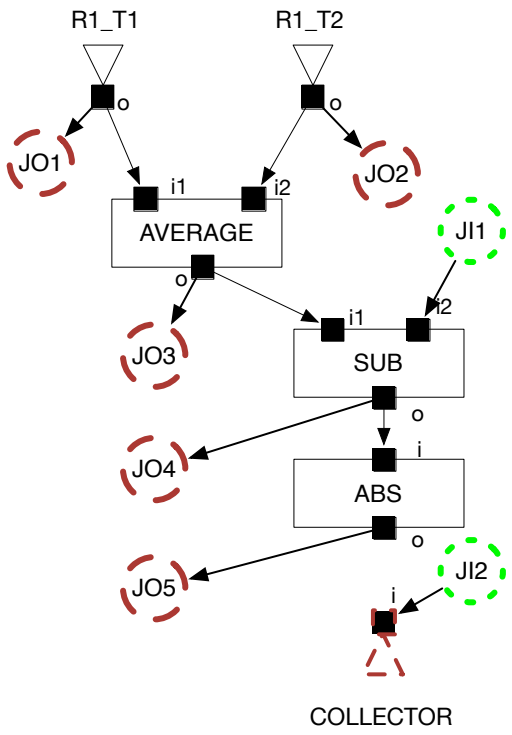


FIGURE 4.5 – Illustration de l'opérateur *extend*

Introduction de l'opérateur de tissage ω

Afin d'effectuer un tissage entre points d'extensions de politiques de collecte de données, nous introduisons l'opérateur de tissage ω.

$$\omega : \Pi \times \Pi \times L \rightarrow \Pi \tag{4.8}$$

Cet opérateur assure le tissage des politiques en créant, à partir d'une liste d'association entre points d'extension, des flux de données entre le port de sortie de l'activité sur laquelle est connecté le point d'extension sortant et le port d'entrée de l'activité sur laquelle est connecté le point d'extension entrant.

L'opérateur de tissage ω crée une nouvelle politique de collecte de donnée π_ω résultante du tissage de deux politiques de collecte de données $(\pi_1, \pi_2) \in \Pi^2$ selon une liste l d'associations $(\epsilon_1, \epsilon_2) \in L$ associant un point d'extension ϵ_1 sortant de π_1 à un point d'extension ϵ_2 entrant de π_2 :

$$\pi_\omega = \langle name, activities(\pi_1) \cup activities(\pi_2), \\ flows(\pi_1) \cup flows(\pi_2) \bigcup_{\forall u \in l} flow(\pi_1.origin, \pi_2.origin) \rangle$$

Fil rouge : Associations

La figure ci-dessous présente le tissage du résultat de la sélection avec deux autres politiques de collecte de données précédemment sélectionnées et étendues. Nous faisons les associations suivantes : `JO6` → `JI1`, `JO3` → `JI3`, `JO5` → `JI2`.

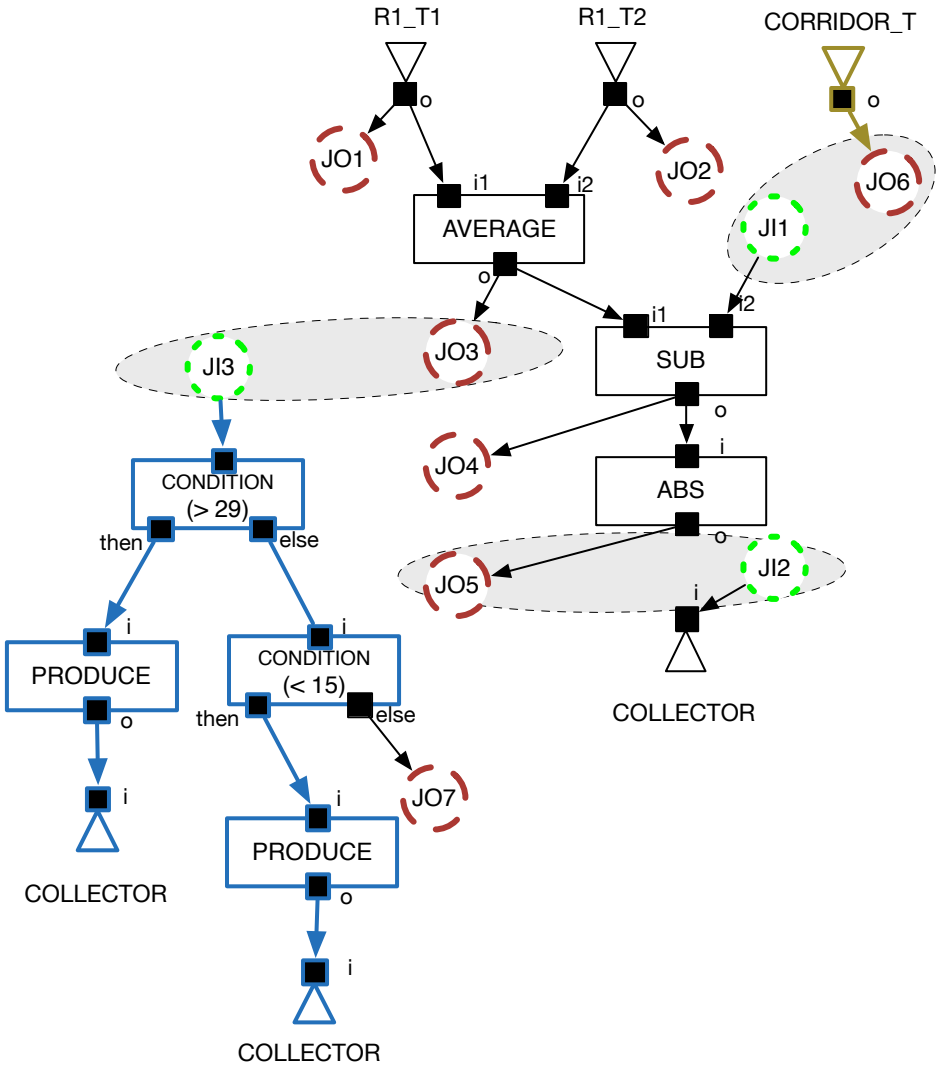


FIGURE 4.6 – Illustration des associations entre points d'extension

Après l'opération de tissage et si la politique de collecte de données est valide, mais que des points d'extension de sortie demeurent inutilisés, l'ingénieur logiciel appliquera la procédure de factorisation afin de supprimer ces points d'extensions non-pertinents.

Fil rouge : Factorisation

Le tissage d'une politique de collecte de données peut se conclure par un appel à la procédure de factorisation si la politique obtenue est valide, mais que des points d'extensions sortant restent inutilisés.

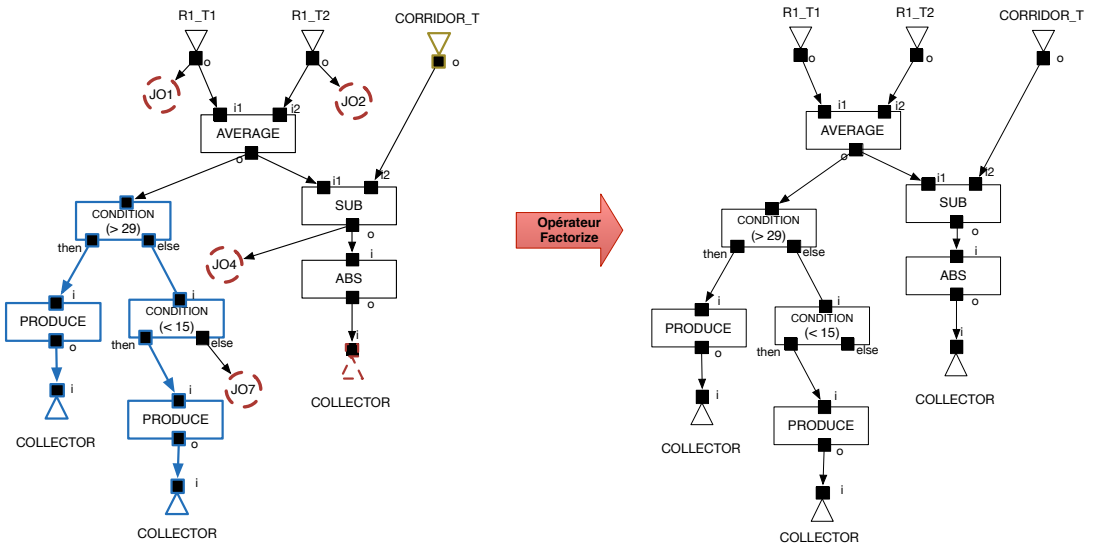


FIGURE 4.7 – Tissage et factorisation

4.3.4 Intégration dans le langage spécifique au domaine des supports à la réutilisation

Au sein de l'état de l'art *cf.* CHAP. 2, seules les approches par composant intègrent directement la notion de réutilisabilité par un ingénieur logiciel : le composant défini est utilisable comme une boite noire au sein de nouveaux programmes. Nous reprenons ce concept de réutilisation en boite noire à travers le procédé de réutilisation hiérarchique et est intégré au langage spécifique au domaine à travers l'instruction `define aProcess(policy)`, *cf.* SEC. 4.2.4. Afin d'utiliser à haut-niveau d'abstraction les opérations de sélection et de tissage, nous étendons le langage spécifique au domaine en ajoutant les instructions suivantes :

```
1 selectIn(POLICY_A) activities(ACTIVITY*)
2 selectIn(POLICY_B) activities(ACTIVITY*)
3 weaveBetween(POLICY_A, POLICY_B) andAssociates(ASSOCIATIONS*)
```

Listing 4.6 – Instructions de sélection et de tissage

En L.1 et L.2, les instructions `selectIn` et `activities` permettent à un ingénieur logiciel de procéder à une sélection d'activités entre deux politiques différentes. La procédure d'extension permettant d'ajouter les points d'extension est automatiquement réalisée sur ces deux politiques lors de l'appel aux instructions de sélection. Enfin, en L.3, l'ingénieur logiciel utilise les instructions `waveBetween` et `andAssociate` afin de procéder au tissage des politiques en créant des flux de données entre les points d'extensions associés.

4.4 Conclusion

Dans ce chapitre, nous avons présenté l'utilisation des processus métiers comme support à l'expression de besoins métiers sous forme de politiques de collecte de données à destination d'une infrastructure de capteurs. Dans l'objectif de garantir un code indépendant de toute considération matérielle, les activités proposées à l'ingénieur logiciel décrivent uniquement une opération à appliquer sur les données et non une mise en œuvre. Les activités proposées sont inspirées de projets tiers afin de proposer un ensemble cohérent par rapport aux besoins métier des ingénieurs logiciels. L'expression de ces politiques est outillée par un langage spécifique au domaine abstrayant les préoccupations réseau et permettant leur définition à un haut niveau d'abstraction.

Les politiques sont également réutilisables totalement ou partiellement grâce aux concepts de réutilisation hiérarchiques ou par tissage. Ces opérations de réutilisations sont intégrées au sein du langage spécifique au domaine afin de pouvoir être directement manipulables par un ingénieur logiciel. Grâce à ces opérations, une politique pourra intégrer des éléments d'une politique tierce afin d'utiliser dans un autre contexte des traitements spécifiques sur les données.

Chapitre 5

Adaptation automatique des politiques à la variabilité du matériel

Sommaire

5.1 Introduction

Une politique de collecte de données définie par un ingénieur logiciel est amenée à être déployée sur une infrastructure de capteurs. De telles infrastructures contiennent par essence différentes plateformes interconnectées et étant chacune spécialisée dans un ensemble de tâches telles que, *par ex.*, mesurer une valeur de capteur ou assurer la connectivité avec un cloud distant.

Définition 5.1.1 (Déploiement)**.** Nous définissons le déploiement d'une politique de collecte de données comme étant la production d'un ensemble de fichiers sources prêts à être exécutés sur une infrastructure de capteurs.

Au cours de ce chapitre, nous faisons l'hypothèse que les politiques de collecte de données à déployer sur une infrastructure de capteurs sont valides au sens des propriétés définies en Sec. 4.2.3.

5.1.1 Problèmes identifiés

À ce jour, un ingénieur logiciel doit tenir compte de l'infrastructure sur laquelle il compte déployer sa politique de collecte de données. Cela l'amène donc à effectuer des tâches en dehors de son champ de compétences comme comprendre la topologie réseau, identifier les plateformes cibles et utiliser des concepts propres au développement de systèmes embarqués. De ce fait, la mise en œuvre d'une politique de collecte de données est généralement laissée à la charge d'un expert réseau qui se basera sur des spécifications établies en amont. Cette étape supplémentaire est couteuse et peut conduire à des erreurs si l'ingénieur logiciel et l'expert réseau ont une interprétation différente des spécifications.

5.1.2 Objectifs de ce chapitre

Ce chapitre cible les objectifs (O_2) et (O_3) :

(O_2) Une politique est adaptée aux capacités matérielles

Le grand nombre de plateformes disponibles conduit à une grande diversité de logiciels et de matériels, ce qui rend chaque déploiement spécifique à une seule infrastructure. L'état de l'art offre déjà quelques abstractions, mais elles doivent être supportées par les différentes plateformes et restent dépendantes de l'infrastructure sous-jacente. Pour aider les ingénieurs logiciels dans l'exploitation d'une infrastructure de capteurs, une politique de collecte de données doit automatiquement être adaptée aux spécificités des plateformes.

(O_3) Une politique est projetée sur l'infrastructure

Par égard aux principes de séparation des préoccupations, une politique de collecte de données ne devraient pas être couplée à une infrastructure de capteurs. Les infrastructures de capteurs à grande échelle mettent à disposition une multitude de plateformes réparties dans des topologies réseaux complexes. Un ingénieur logiciel ne devrait pas avoir à tenir compte de cette complexité : un mécanisme attribuant un ensemble de plateformes pour la mise en œuvre de la politique serait nécessaire.

5.1.3 Plan du chapitre

Dans une première partie Sec. 5.2, nous introduisons un modèle d'infrastructure permettant de décrire l'infrastructure de capteurs et qui sera utilisé pour le déploiement des politiques de collecte de données. Ensuite, en Sec. 5.3, nous indiquerons comment le positionnement des activités sur une infrastructure peut être optimisé grâce aux stratégies de déploiement. En Sec. 5.4, nous présentons différents opérateurs permettant de déployer une politique de collecte de données. Enfin, la Sec. 5.5 présente les générateurs de codes nécessaires à la mise en œuvre des politiques et Sec. 5.6 conclut ce chapitre.

Contributions présentées

Ce chapitre présente les contributions suivantes :

- Un modèle des fonctionnalités permettant de représenter la diversité des plateformes réparties dans l'infrastructure de capteurs et un modèle topologique permettant de décrire les connexions entre les plateformes ;
- Une stratégie de déploiement permettant à un expert réseau d'optimiser le déploiement des politiques de collecte de données par rapport à des critères fonctionnels ;
- Un mécanisme d'adaptation aux spécificités des plateformes de l'infrastructure et un opérateur de décomposition permettant de répartir les activités sur les différentes plateformes ;
- Des générateurs de code produisant du code adapté aux spécificités des plateformes grâce à des modèles de variabilité.

5.2 Gérer la variabilité de l'infrastructure

5.2.1 Problèmes liés à la représentation logicielle d'une infrastructure de capteurs

Une plateforme $p \in P$ au sein d'une infrastructure de capteurs est constituée de cinq composants physiques principaux [KW07] *cf.* Fig. 5.1 :

- un composant *Contrôleur* assurant l'exécution du code embarqué
- un composant *Mémoire* stockant la mise en œuvre de la politique ainsi que les variables temporaires

— un composant *Capteur* regroupant l'ensemble des périphériques visant à mesurer une valeur physique

— un composant *Communication* assurant la connectivité de la plateforme au sein d'un réseau

— un composant *Énergie* assurant l'alimentation énergétique de la plateforme.

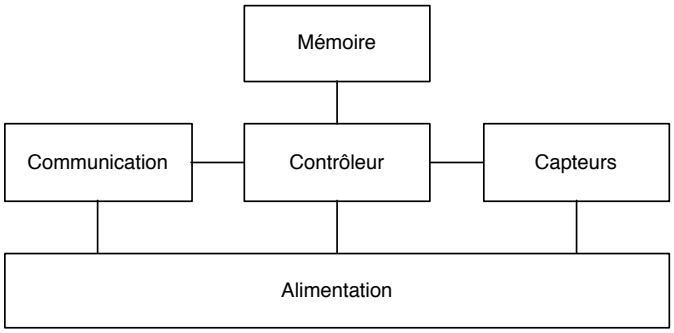


FIGURE 5.1 – Représentation structurelle des composants physiques d'une plateforme selon [KW07]

La diversité des fournisseurs en plateformes de capteurs et des technologies disponibles sur le marché font que chacun de ces composants peut varier d'une plateforme à une autre conduisant à des infrastructures de capteurs hétérogènes [KBG+16]. La mise en œuvre d'une politique de collecte de données doit prendre en compte les capacités fonctionnelles de la plateforme, *par ex.*, un générateur de code Python ne doit pas être utilisé pour une plateforme ne supportant qu'un contrôleur programmable en C. Toutefois, identifier les capacités techniques de chacune des plateformes pour adapter la politique est hors du domaine de compétences d'un ingénieur logiciel. Nous devons donc proposer un modèle regroupant l'ensemble des caractéristiques du matériel sous-jacent. De plus, une politique de collecte de données décrit, de manière centrale, les intentions à effectuer sur plusieurs plateformes de l'infrastructure de capteurs. L'ingénieur logiciel n'a pas la connaissance de la topologie réseau de l'infrastructure requise pour répartir au mieux ses intentions sur les différentes plateformes de l'infrastructure. Nous devons donc également proposer un modèle qui permet de représenter la topologie réseau.

5.2.2 Gérer la variabilité du matériel

Afin de gérer la variabilité des différentes plateformes composant l'infrastructure de capteurs, nous devons avoir une représentation de la variabilité de chacun des composants physiques. En génie logiciel, la modélisation de la variabilité doit permettre de capturer, organiser et représenter la variabilité [CB11]. Les lignes de produits complexes [UBFC14a], grâce à la combinaison d'un *modèle du domaine* et de *modèles de variabilité* permettent de gérer la variabilité pour chacun des éléments composant un modèle du domaine. Afin d'étendre cette approche aux infrastructures de capteurs, nous considérons un modèle du domaine décrivant la structure, *i.e.*, les composants physiques, d'une plateforme de capteurs. Pour chaque composant physique, nous capturons les différentes configurations possibles au sein d'un modèle de variabilité. Ce modèle a ainsi pour but d'organiser et représenter la variabilité.

▶ ***Modèle du domaine***. Le modèle du domaine décrit la structure physique d'une plateforme. En plus des composants physiques représentés FIG. 5.1, nous devons ajouter au modèle du domaine un élément *langage de programmation* représentant le ou les langages de programmation supportés par la plateforme. À partir de l'étude des plateformes disponibles sur le marché, nous introduisons un modèle du domaine tel que représenté en FIG. 5.2. Une plateforme possède un contrôleur pouvant être programmé à l'aide d'un ou plusieurs langages de programmation. Elle peut également posséder des capteurs. De plus, selon le type de mémoire utilisé, elle possède une certaine capacité de calcul. Ensuite, étant donné qu'une plateforme est déployée au sein d'un réseau, nous considérons qu'elle possède un ou plusieurs moyens de communication. Enfin la plateforme est alimentée par une source d'énergie.

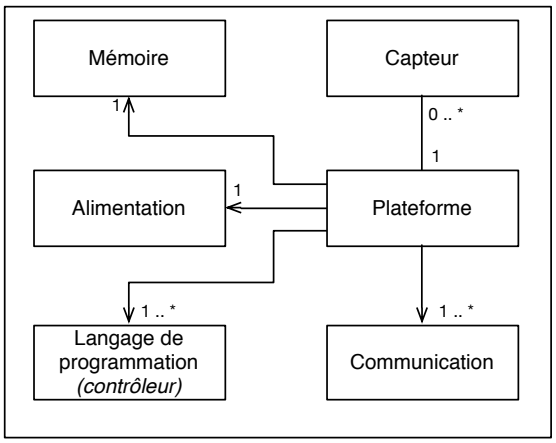


FIGURE 5.2 – Modèle du domaine décrivant une plateforme de capteurs

▶ ***Modèles de variabilité***. Les modèles de variabilité permettent de modéliser la diversité des composants structurels d'une plateforme. Dans le cadre de cette approche, nous nous reposons sur les *feature models* (FM) [KCH+90] [KLD02] afin de modéliser chaque composant d'une plateforme comme étant un arbre dans lequel chaque nœd représente une caractéristique. Les arbres sont assimilés à une équation logique où chaque solution donne une configuration possible pour la plateforme.

▶ ***Variabilité des contrôleurs***. Le contrôleur d'une plateforme de capteurs peut être programmé grâce à un langage de programmation :

$$CONTROLEUR = (nesC|Contiki|Python|Wiring|Java|Groovy|\ldots);$$

▶ ***Variabilité de la mémoire***. Nous considérons la mémoire comme étant la capacité d'une plateforme à stocker en mémoire des données. Nous simplifions la variabilité de la mémoire comme $MEMOIRE = (HIGH|LOW|CLOUD)$; selon si la plateforme a beaucoup de mémoire (*par ex.*, un RaspberryPi), peu de mémoire (*par ex.*, un ATmega328P [1]), ou est située dans un nuage (*par ex.*, un serveur distant dans une infrastructure extensible). Nous laissons à l'expert réseau déterminer la capacité de chacune des plateformes consistant son infrastructure de capteurs.

▶ ***Variabilité des capteurs***. Nous considérons un capteur $c \in S$ comme étant un composant transformant une grandeur physique en une tension électrique mesurable.

1. micro-contrôleur présent notamment sur les Arduino Uno et ayant 2 ko de mémoire vive

La *Sensor Manufacters Association* recense 81 types de capteurs différents. Pour chacun de ces 81 types de capteurs, nous pouvons également identifier des dizaines de constructeurs différents. Nous proposons une représentation de la variabilité des capteurs sous la forme d'un arbre où l'expert réseau sélectionne, dans un premier temps, le type de capteur puis, dans un second temps, le constructeur du capteur :

$$CAPTEUR = (TEMPERATURE|HUMIDITE|\ldots);$$
$$TEMPERATURE = (GROVE|ELECTRONICBRICKS|\ldots);$$

Cette dernière sélection permettra aux générateurs de code (*cf.* Sec. 5.5) d'utiliser les bibliothèques logicielles adaptées.

► ***Variabilité des communications***. Une plateforme de capteurs possède un ou plusieurs moyens de communication caractérisés par une technologie de communication filaire ou sans-fils et un sens de communication. L'expert réseau, à travers une seconde sélection, précise également le type de technologie de communication afin que les générateurs de code (*cf.* Sec. 5.5) utilisent les bibliothèques logicielles adaptées :

$$COMMUNICATION = (SENS(SANSFILS|FILAIRE));$$
$$SENS = (IN|OUT);$$
$$SANSFILS = (WIFI|BLE|XBEE|ZWAVE|LORA|SIGFOX;\ldots);$$
$$FILAIRE = (ETHERNET|SERIAL|I2C|WAN\ldots);$$

► ***Variabilité de l'énergie***. L'alimentation énergétique d'une plateforme de capteurs peut provenir soit de batteries, soit de courant secteur :

$$ENERGIE = (BATTERIE|SECTEUR)$$

La figure Fig. 5.3 illustre l'application du modèle du domaine et des modèles de variabilité afin de décrire une plateforme de capteurs.

5.2.3 Gérer la topologie réseau

Les infrastructures de capteurs sont constituées de plateformes reliées les unes aux autres par un moyen de communication pouvant être filaire ou sans-fil et à coût variable. Une topologie réseau est assimilée à un graphe où les noeuds représentent les plateformes et où les arcs représentent la connectivité entre deux plateformes [SW08].

Hypothèse H_4 (Topologie statique) : Nous considérons une infrastructure de capteurs fixe, *i.e.*, où les plateformes ne sont pas mobiles. Nous retrouvons notamment ce type d'infrastructure au niveau des bâtiments intelligents où les plateformes ne sont pas déplacées.

Dans notre approche, nous considérons des infrastructures remontant, via des liaisons à coûts variables, des valeurs de capteurs depuis les plateformes vers une application métier. Ainsi, nous assimilons la topologie réseau à un graphe orienté pondéré $T = (P, C)$ où P représentent les plateformes et C représentent la connectivité entre deux plateformes $(p_1, p_2) \in P^2$. Le poids associé à chaque arc représente le coût de la connectivité selon le tableau Tab. 5.1.

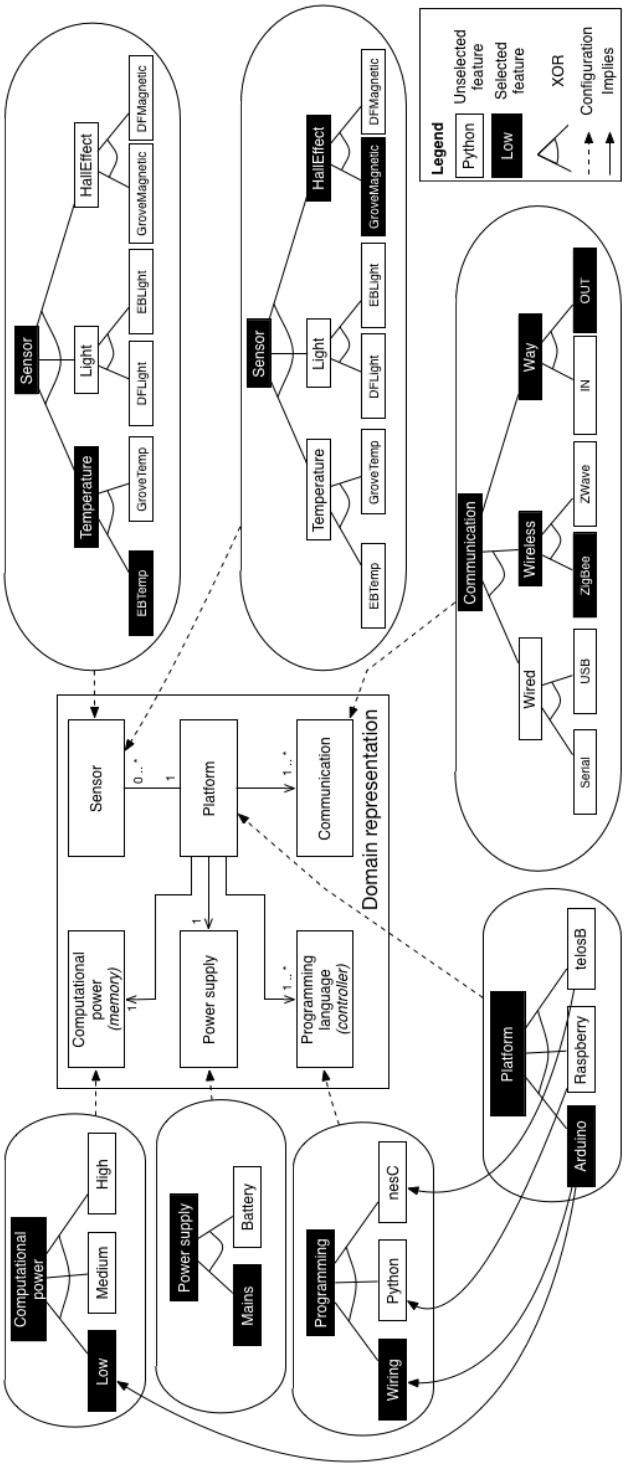


FIGURE 5.3 – Exemple d'une configuration décrivant une plateforme de capteurs

TABLEAU 5.1 – Pondération des communications réseau

Poids	*Type de communication*	*Examples*
1	Filaire	USB, Série
2	Sans-fils (local)	ZigBee, Z-Wave
3	Sans-fils (large échelle ou média payant)	Sigfox, GSM

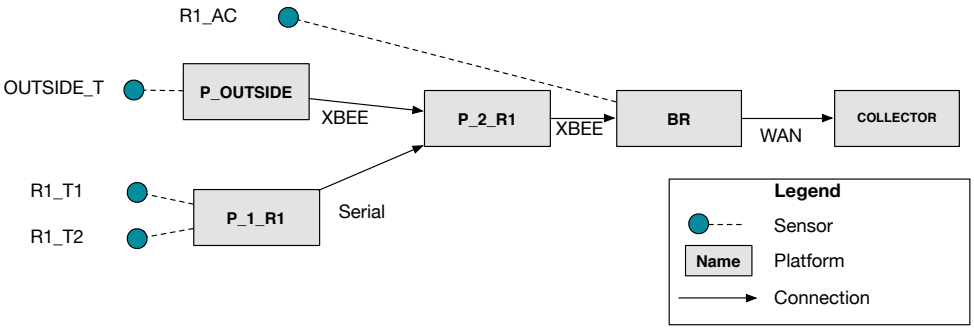


FIGURE 5.4 – Topologie décrivant une infrastructure de capteurs

5.2.4 Synthèse

Les modèles du domaine et le modèle de la topologie réseau sont définis par un expert réseau et permettent de décrire structurellement une infrastructure de capteurs. Les différentes sélections effectuées au sein des modèles de variabilité permettent d'obtenir, pour chaque plateforme, un modèle du domaine utilisable lors des phases de déploiement.

Fil rouge : Plateformes et topologie

La politique fil rouge est amenée à être déployée sur une infrastructure de capteurs hétérogène. Parmi l'ensemble des plateformes déployées, il contient un Arduino Uno (`P_1_R1`) sur lequel est connecté le capteur `R1_T1` et une plateforme dites *border router* (`BR`) faisant l'interconnexion du réseau de capteur avec Internet. Les choix de configuration résultant de la sélection des produits sont présentés ci-dessous :

TABLEAU 5.2 – Exemple de plateformes

	Arduino Uno `P_1_R1`	**Raspberry Pi 3** `BR`
Platform	Arduino	Rasbperry
Sensor(s)	Grove Temperature (R1_T1) Grove Temperature (R1_T2)	N/A
Communication(s)	Serial (OUT)	XBee (IN) WAN (OUT)
Prog. language	Wiring	Python
Power supply	Main	Main
Comp. power	Low	High

La figure Fig. 5.4 illustre la topologie réseau sur laquelle la politique fil rouge sera amenée à être déployée. L'expert réseau utilise une représentation XML pour exprimer les relations entre les différentes plateformes :

Listing 5.1 – Description de la topologie réseau

```
<connections>
    <connection from="P_1_R1" to="P_2_R1" weight="1"/>
    <connection from="P_OUTSIDE" to="P_2_R1" weight="2"/>
    <connection from="P_2_R1" to="BR" weight="2"/>
    <connection from="BR" to="COLLECTOR" weight="3" />
</connections>
```

5.3 Positionnement des activités sur l'infrastructure de capteurs

5.3.1 Besoin d'optimisation du déploiement

Le positionnement des activités vise à attribuer une plateforme cible de déploiement à chaque activité de la politique de collecte de données. Selon le type d'infrastructure de capteurs, le déploiement d'une collecte de données doit satisfaire un certain nombre de critères définis par un expert réseau. Par exemple, dans le cas d'une infrastructure fonctionnant sur batterie et où les communications réseau consomment beaucoup d'énergie, un expert réseau souhaitera que les activités de la politique de collecte de données soient effectuées au plus près des capteurs afin de limiter les communications inter-plateformes. Les critères d'optimisation peuvent être statiques, *i.e.*, n'évoluent pas au cours du temps, ou dynamiques.

5.3.2 Optimisation du déploiement

Afin de permettre à un expert réseau de traduire ses critères d'optimisation, nous introduisons la notion de *stratégie de déploiement* pouvant être statique ou dynamique :

Définition 5.3.1 (Stratégie de déploiement statique)**.** Une stratégie de déploiement *statique* vise à orienter le déploiement des activités contenues au sein d'une politique de collecte de données à partir d'une représentation statique de l'infrastructure de capteurs.

Définition 5.3.2 (Stratégie de déploiement dynamique)**.** Une stratégie de déploiement *dynamique* vise à orienter le déploiement des activités contenues au sein d'une politique de collecte de données en prenant en compte l'état courant de l'infrastructure de capteurs.

▶ ***Opérateur de positionnement***. Le but de la stratégie de déploiement est de retourner la plateforme satisfaisant au mieux les critères d'optimisation définis par l'expert réseau pour une activité. Pour cela, l'expert réseau défini un opérateur de positionnement *place* pour chacune des stratégies. Cet opérateur prend en entrée une activité, une liste de plateformes candidates au déploiement et le modèle topologique. Il retourne la plateforme cible qui satisfait au mieux la stratégie. Des exemples de mise en œuvre d'opérateur de positionnement à travers différentes stratégies seront illustrés en Sec. 5.4.2.

$$place : A \times P^{+} \times T \rightarrow P \tag{5.1}$$

5.3.3 Exemple illustratif

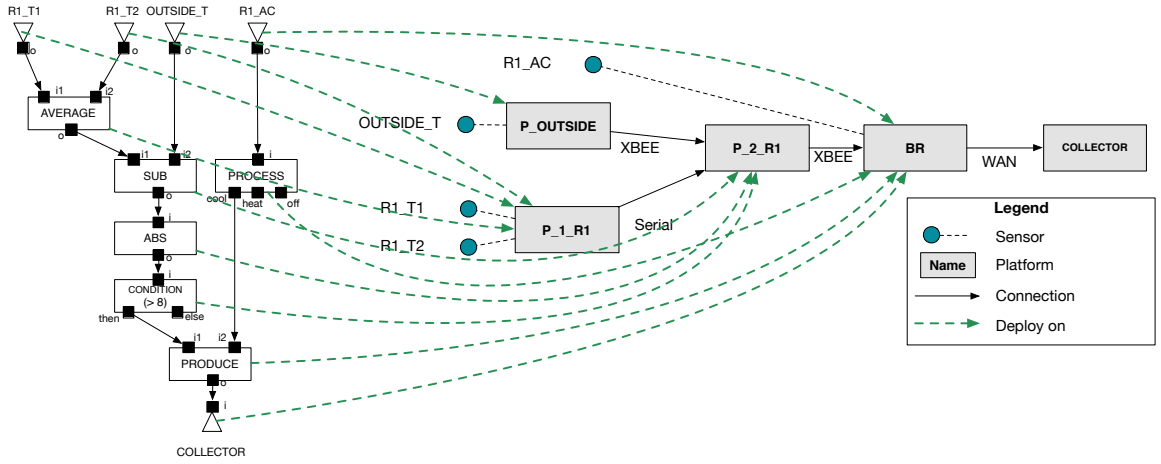


FIGURE 5.5 – Illustration du positionnement des activités de la politique fil rouge avec la stratégie *au plus près des capteurs*

La figure FIG. 5.5 présente le positionnement des activités de la politique fil rouge sur l'infrastructure de capteurs en suivant la stratégie *au plus près des capteurs* (décrite ultérieurement en SEC. 5.4.2).

5.3.4 Synthèse

Dans les deux sections précédentes, nous avons introduit les modèles du domaine (décrivant les fonctionnalités des plateformes), topologique (décrivant la configuration structurelle du réseau) ainsi que les stratégies de déploiement permettant d'optimiser le placement des activités en fonction de critères définis par un expert réseau. Nous regroupons ces trois notions sous la forme d'un **modèle d'infrastructure** spécifique à l'infrastructure de capteurs considérée.

Définition 5.3.3 (Modèle d'infrastructure)**.** Nous définissons un modèle d'infrastructure $i \in I$ comme un triplet $\langle f, t, s \rangle$ où f correspond à un modèle $f \in F$ des fonctionnalités des plateformes, t à un modèle $t \in T$ de la topologie réseau et s à une stratégie de déploiement $s \in \Sigma$.

Ce modèle topologique permettra de guider le déploiement de la politique de collecte de données sur l'infrastructure de capteurs.

5.4 Déploiement d'une politique de collecte de données

5.4.1 Problèmes liés au déploiement d'une politique de collecte de données

Lors du déploiement d'une politique de collecte de données, un expert réseau a besoin d'identifier quelles plateformes de capteurs seront impliquées dans sa réalisation.

En effet, l'infrastructure de capteur contenant un ensemble de plateformes connectées, elles n'émettent ou ne reçoivent pas toutes les mêmes valeurs de capteurs.

De plus, les intentions de l'ingénieur logiciel exprimées au sein de la politique sont indépendantes de l'infrastructure sous-jacente, par égard au principe de séparation des préoccupations. Chaque activité doit alors être implémentée en fonction de la plateforme sur laquelle elle sera déployée.

Si ces deux tâches sont relativement aisées sur des infrastructures de l'ordre de quelques plateformes, le passage à l'échelle de l'infrastructure induit une complexité croissante sur le nombre de plateformes à identifier. Nous proposons ainsi d'automatiser la tâche de déploiement qui devra respecter les deux propriétés suivantes :

► ***Adaptation de la politique***. Une politique de collecte de données doit être adaptée aux caractéristiques matérielles des plateformes composant l'infrastructure cible du déploiement.

► ***Identification des plateformes cibles***. Le processus de déploiement doit automatiquement identifier les plateformes cibles au sein de l'infrastructure de capteurs.

5.4.2 Adaptation de la politique de collecte de données (phase de pré-déploiement)

Chaque activité impliquée dans une politique de collecte de données *valide* repose sur des données provenant d'un ou plusieurs capteurs et peut n'être déployée que sur un type particulier de plateforme. Par exemple, une activité représentant une interface de sortie vers un collecteur distant ne peut être déployée que sur une plateforme ayant une connectivité Internet. De plus, en fonction de la topologie réseau, les plateformes ne reçoivent pas l'ensemble des valeurs de capteurs. Pour supporter l'adaptation de la politique, nous avons ainsi besoin d'identifier les potentielles plateformes cibles pour chaque activité. Ces plateformes cibles doivent assurer **l'accessibilité des valeurs de capteurs** requises par l'activité et la **compatibilité de l'activité**.

Accessibilité des valeurs de capteurs

Définition 5.4.1 (Valeur de capteur accessible)**.** Une valeur de capteur v_c est accessible sur une plateforme p si p peut recevoir v_c

Afin d'assurer l'accessibilité des valeurs de capteurs requis par l'activité, nous devons identifier quels capteurs sont accessibles depuis une plateforme p et quels capteurs sont requis pour une activité a. Nous introduisons ainsi les opérateurs *accessible* et *req*.

► ***Opérateur*** *accessible*. L'opérateur *accessible* est défini au niveau de la **topologie réseau** et utilise le modèle du domaine [2] afin d'identifier les capteurs accessibles depuis une plateforme. Cet opérateur est défini comme suit :

$$accessible : P \rightarrow S \tag{5.2}$$

Il effectue un parcours en profondeur du graphe $T^R = (P, C^R)$ où C^R correspond à l'ensemble des arcs orientés dans le sens inverse afin de déterminer l'ensemble des capteurs accessible depuis une plateforme $p \in P$.

► ***Opérateur*** *req*. L'opérateur *req* est défini au niveau de la **politique de collecte de données** et identifie les capteurs requis pour la réalisation d'une activité. Cet opérateur est défini comme suit :

2. Nous rappelons que les capteurs sont un élément du modèle du domaine

TABLEAU 5.3 – Activités ayant des contraintes de déploiement

	Activity	Constraint
Inputs	s :PeriodicSensor	s in target
	s :EventSensor	s in target
	i :InputJoinPoint	target has a communication IN
Outputs	c :Collector	target has a communication WAN OUT
	o :OutputJoinPoint	target has a communication OUT

$$req : A \rightarrow S \tag{5.3}$$

Il effectue un parcours en profondeur de la politique de collecte de données en renversant les flux de données afin de déterminer l'ensemble des capteurs requis depuis une activité.

Les opérateurs *req* et *accessible* permettent respectivement d'identifier les capteurs requis pour la réalisation d'une activité et d'identifier les capteurs accessibles depuis une plateforme de capteurs. Pour finaliser la sélection des plateformes pouvant mettre en œuvre une activité, il est nécessaire de déterminer si l'activité est compatible avec la cible de déploiement.

Vérification de la compatibilité de la cible de déploiement

Des activités peuvent posséder un certain nombre de contraintes sur leur déploiement : les activités récupérant une valeur de capteur doivent être déployées sur la plateforme contenant le capteur en question, les activités transmettant les données de capteurs à un collecteur distant doivent avoir une connexion internet sortante et les activités de point d'extension entrant et sortant ont respectivement une connexion entrante et sortante. Ces contraintes sont exprimées sous la forme d'une équation logique, *cf.* TAB. 5.3, et devront être satisfaites par les capacités fonctionnelles de la plateforme. Afin de vérifier la satisfaisabilité de cette équation logique, nous introduisons un opérateur $isDeployable$ défini comme suit :

$$isDeployable : A \times P \rightarrow \{vrai; faux\} \tag{5.4}$$

► ***Identification des plateformes candidates***.

Préalablement à l'identification des plateformes candidates, les activités `PROCESS` sont remplacées par leur politique de collecte de données embarquée. En effet, il n'est pas possible d'affecter à l'activité `PROCESS` un ensemble de plateformes candidates puisque la politique embarquée peut être amenée à être déployée sur plusieurs plateformes. Ensuite, à partir des opérateurs *req*, *accessible* et *isDeployable*, nous pouvons identifier les plateformes candidates au déploiement d'une activité, *i.e.*, répondant aux critères d'accessibilité des données de capteurs et de compatibilité.

Soient a une activité et p une plateforme de l'infrastructure de capteur, les critères d'accessibilité et de compatibilité sont respectivement définis comme $req(a) \subset accessible(p)$ et $isDeployable(a, p)$.

Pour chacune des activités a contenues dans une politique de collecte de données π, nous associons une propriété *targets* contenant l'ensemble des plateformes candidates au déploiement de l'activité sur une infrastructure i :

$$\forall a \in activities(\pi), \forall p \in platforms(i), a.targets = \{(req(a) \subseteq accessible(p)) \wedge isDeployable(a,p)\} \tag{5.5}$$

Déploiement de la politique de collecte de données

▶ ***Sélection de la plateforme cible***. Chaque activité possède, à travers la propriété associée *targets*, un certain nombre de plateformes candidates à son déploiement. La sélection de la plateforme cible pour chacune des activités doit être effectuée en assurant la **continuité de la politique** et en fonction de la **stratégie de déploiement**.

La topologie de l'infrastructure de capteurs fait que la sélection au hasard de plateformes candidates comme cible de déploiement peut entrainer une rupture de la continuité de la politique. Par exemple, soient deux activités A et B reliées par un flux de données, si A est déployée sur une plateforme P_A et B est déployée sur une plateforme, P_B mais que la topologie réseau n'assure pas de connectivité entre P_A et P_B alors il y a rupture de la continuité de la politique.

▶ ***Continuité de la politique***. Pour assurer la continuité de la politique, nous introduisons un algorithme de filtrage. Cet algorithme, présenté en Lst. 5.2, prend en entrée une liste **ordonnée** d'activités associées à leurs plateformes de déploiement candidates et une liste **ordonnée** de plateformes [3]. Il a pour responsabilité de ré-écrire la propriété *targets* de chaque activité en supprimant les plateformes responsables de la rupture de la continuité. L'algorithme considère deux activités a et b successives. Il compare les plateformes candidates de b par rapport aux plateformes candidates de a. Si des plateformes candidates de b sont topologiquement situées en amont des plateformes candidates de a, alors ces plateformes sont filtrées.

Listing 5.2 – Filtrage des plateformes candidates

```
1  def filter(activites:List[Activity, Set[Platform]], topology[ListPlatform]) = {
2    activites match {
3      case a :: b :: tail =>
4        val filtered = a._2.filterNot(e => { //a._2 = plateformes candidates de a
5          val indexA = topology.index(e) //index plateforme candidate e
6          val indexesB = b._2.map { topology(_) } //b._2 = plateformes candidates de b
7                  indexesB.map { i <= i < indexA }.nonEmpty}) //filtrage en fonction
8                                                              //de l'index
9        a._1.addProperty("targets", filtered) //re-ecriture
10       filter(b :: tail) //recursion sur la prochaine activite
11     case a :: Nil => () //une seule activite dans la liste, stop
12     case Nil => () //pas d'activite dans la liste, stop
13     }
14 }
```

▶ ***Utilisation de la stratégie de déploiement***. Après le filtrage des plateformes candidates, l'application de l'opérateur *place* de la stratégie de déploiement permet de sélectionner la plateforme cible pour le déploiement de l'activité. Cette plateforme cible est enregistrée dans la propriété *target*. Dans l'éventualité où l'opérateur *place* retourne $\emptyset$, alors un message d'erreur est retourné à l'ingénieur logiciel spécifiant que l'activité a ne peut pas être déployée à cause de la stratégie choisie par l'expert réseau. Dans la suite, nous utiliserons la dénomination de politique *pré-déployée* π_{pre}, pour se référer à une politique où les différentes activités ont une propriété *target* associée. Nous présentons ci-dessous trois stratégies de déploiement possibles sur les infrastructures de capteurs afin d'illustrer le fonctionnement de l'opérateur *place*.

3. Les politiques de collecte de données ainsi que les topologies réseau sont assimilées à des graphes orientés. Il est possible d'obtenir une liste ordonnée en effectuant un tri topologique des sommets.

Stratégie : Au plus près des capteurs (statique)

▶ ***But de la stratégie***. Cette stratégie a pour but de déployer les activités d'une politique au plus près des capteurs afin d'exploiter au maximum les ressources de l'infrastructure de capteurs et diminuer la consommation énergétique de l'infrastructure.

▶ ***Mise en œuvre de l'opérateur place***. Le code Lst. 5.3 présente la mise en œuvre de l'opérateur *place* pour la stratégie *au plus près des capteurs*. Dans un premier temps, l'opérateur calcule la distance minimale (grâce à un algorithme de parcours de graphe) entre chacune des plateformes candidates et leurs capteurs accessibles. Dans un second temps, l'opérateur calcule la distance moyenne de la plateforme par rapport à l'ensemble des capteurs. La plateforme ayant la distance moyenne minimale est désignée comme plateforme cible pour le déploiement de l'activité.

Listing 5.3 – Opérateur *place* pour la stratégie au plus près des capteurs

```
def place(a:Activity,
          candidates:Set[Platform], infra:InfrastructureModel):Platform={

    val distanceFromSensors = (for (p <- candidates) yield {
        (p, for (s <- accessible(p, infra.topology) yield {
            (s shortestPathTo p).edges.map(_.weight).sum))
        }
    }

    val avgDistanceFromSensor = distanceFromSensors.map { e ->
        (e._1, e._2.toMap.values.mean)
    }

    avgDistanceFromSensor.minBy(_._1)._1
}
```

Stratégie : Privilégier les plateformes alimentées sur le secteur (statique)

▶ ***But de la stratégie***. La consommation énergétique au sein d'un réseau de capteur est un problème récurrent pour les experts réseau [ACDFP09]. Ainsi, cette stratégie maximise l'utilisation de plateformes alimentées sur secteur au sein d'un réseau composé de plateformes alimentées par batteries et d'autres par secteur.

▶ ***Mise en œuvre de l'opérateur place***. Le code Lst. 5.4 présente la mise en œuvre de l'opérateur *place* pour la stratégie *plateformes alimentées sur le secteur*. Grâce au modèle des fonctionnalités des plateformes, l'opérateur identifie les plateformes alimentées par le secteur au sein de la liste des plateformes candidates au déploiement. Il retourne ensuite alealéatoirement une plateforme alimentée par le secteur comme cible de déploiement. Dans le cas où aucune plateforme n'est alimentée par le secteur, l'opérateur retourne une plateforme choisie aléatoirement parmi les plateformes candidates.

Listing 5.4 – Opérateur *place* pour la stratégie plateformes alimentées sur le secteur

```
def place(a:Activity,
          candidates:Set[Platform], infra:InfrastructureModel):Platform={

    val mains_powered = candidates.filter { p ->
        infra.facilities.get(p).energy == ENERGY.MAINS}.toList

    if (mains_powered.length > 0)
        Random.shuffle(mains_powered).head
    else Random.shuffle(candidates).head
}
```

Stratégie : Plateformes libres uniquement (dynamique)

▶ ***But de la stratégie***. Cette stratégie permet de déployer des activités **uniquement** sur des plateformes qui n'ont pas été impliquées dans un précédent déploiement. Elle repose sur un service de déploiement `InfrastructureManager` (introduit en CHAP. 6) contenant la liste des politiques exécutées sur chaque des plateformes.

▶ ***Mise en œuvre de l'opérateur place***. Le code LST. 5.5 présente la mise en œuvre de l'opérateur *place* pour la stratégie *plateformes libres*. L'opérateur utilise un service de déploiement pour identifier, parmi la liste des plateformes candidates, celles n'étant pas déjà impliquées dans la réalisation d'une politique de collecte de données tierces. Il retourne ensuite aléatoirement une plateforme libre comme cible de déploiement. Dans l'éventualité où toutes les plateformes supportent déjà une politique de collecte de données, l'opérateur renvoie $\emptyset$.

Listing 5.5 – Opérateur *place* pour la stratégie plateformes libres uniquement

```
def place(a:Activity,
          candidates:Set[Platform], infra:InfrastructureModel):Platform={

    val empty_platforms = candidates.filter(p ->
        InfrastructureManager(infra).subPolicies(p) == null)

    if (empty_platforms > 0)
        Random.shuffle(empty_platforms).head
    else null
}
```

Fil rouge : Identification des plateformes cibles

La politique fil rouge est amenée à être déployée sur une infrastructure de capteurs représentée par le modèle d'infrastructure $i \in I = \langle f, t, s \rangle$ où f correspond à l'ensemble des configurations de plateformes (*cf.* exemple illustré par FIG. 5.2), t à la topologie réseau présentée en FIG. 5.4 et s à la stratégie *au plus près des capteurs*.

TABLEAU 5.4 – Application de l'opérateur *req* sur les activités de la politique fil rouge

Activity	***req-operator***	***constraint***
R1_T1	{R1_T1}	R1_T1 ∈ target
R1_T2	{R1_T2}	R1_T2 ∈ target
OUTSIDE_T	{OUTSIDE_T}	OUTSIDE_T ∈ target
R1_AC	{R1_AC}	R1_AC ∈ target
AVERAGE	{R1_T1, R1_T2}	
SUB	{R1_T1, R1_T2, OUTSIDE_T}	
ABS	{R1_T1, R1_T2, OUTSIDE_T}	
CONDITION (> 8)	{R1_T1, R1_T2, OUTSIDE_T}	
PRODUCE	{R1_T1, R1_T2, OUTSIDE_T, R1_AC}	
CONDITION (< 16)	{R1_AC}	
COLLECTOR	{R1_T1, R1_T2, OUTSIDE_T, R1_AC}	target has internet access

TABLEAU 5.5 – Application de l'opérateur *accessible* sur les plateformes de la topologie réseau

Platform	***accessible-operator***
P_OUTSIDE	{OUTSIDE_T}
P_1_R1	{R1_T1, R1_T2}
P_2_R1	{OUTSIDE_T, R1_T1, R1_T2}
BR	{OUTSIDE_T, R1_T1, R1_T2, R1_AC}
COLLECTOR	{OUTSIDE_T, R1_T1, R1_T2, R1_AC}

À partir de l'application de l'opérateur *req* sur les activités de la politique fil rouge TAB. 5.4 et de l'opérateur *accessible* sur la topologie réseau TAB. 5.5, nous obtenons les plateformes candidates pour le déploiement des activités grâce à l'équation EQ. 5.5 :

TABLEAU 5.6 – Identification des plateformes candidates

Activity	***targets***
R1_T1	{P_1_R1}
R1_T2	{P_1_R1}
OUTSIDE_T	{P_OUTSIDE}
R1_AC	{BR}
AVERAGE	{P_1_R1, P_2_R1, BR, COLLECTOR}
SUB	{P_2_R1, BR, COLLECTOR}
ABS	{P_2_R1, BR, COLLECTOR}
CONDITION(> 8)	{P_2_R1, BR, COLLECTOR}
PRODUCE	{BR, COLLECTOR}
CONDITION(< 16)	{BR, COLLECTOR}
COLLECTOR	{BR}

L'application de l'opérateur *place* associe chaque activité à une plateforme cible (TAB. 5.7) :

TABLEAU 5.7 – Identification des plateformes cibles

Activity	***target***
R1_T1	P_1_R1
R1_T2	P_1_R1
OUTSIDE_T	P_OUTSIDE
R1_AC	BR
AVERAGE	P_1_R1
SUB	P_2_R1
ABS	P_2_R1
CONDITION(> 8)	P_2_R1
PRODUCE	BR
CONDITION(< 16)	BR
COLLECTOR	BR

5.4.3 Création de sous-politiques spécifiques aux plateformes (phase de déploiement)

Principe de l'opérateur de déploiement

La politique π_{pre} pré-déployée pour une infrastructure de capteurs I contient un ensemble d'activités associées à une plateforme de I. Nous identifions l'ensemble des plateformes P_{inv} impliquées dans la réalisation de la politique comme suit :

$$P_{inv} = \bigcup_{\forall a \in activities(\pi_{pre})} a.target$$

Chacune des plateformes $p \in P_{inv}$ aura pour responsabilité d'effectuer un certain nombre d'activités exprimées par l'ingénieur logiciel. Nous proposons de reformer de nouvelles politiques de collecte de données, spécifiques à une plateforme, regroupant uniquement les actions à effectuer sur cette plateforme. Nous introduisons ainsi l'opérateur *deploy* défini comme suit :

$$deploy : \Pi \times I \rightarrow \Pi^{+} \tag{5.6}$$

Cet opérateur consomme une politique de collecte de données *pré-déployée* et le modèle d'infrastructure et retourne, pour chaque plateforme de l'infrastructure de capteurs, une nouvelle politique de collecte de données.

Fonctionnement de l'opérateur de déploiement

À partir de π_{pre}, nous procédons à des sélections (*cf.* Eq. 4.7) d'ensembles d'activités ayant la même plateforme de déploiement. Le résultat de la sélection renvoie ainsi, pour chaque plateforme $p \in P_{inv}$, une nouvelle politique contenant l'ensemble des activités et flux de données spécifiques à p. Chaque flux de donnée f qui a été coupé par la sélection est remplacé par un *point d'extension sortant* JO au niveau du port de sortie de l'activité $f.source$ et par un *point d'extension entrant* JI au niveau du port d'entrée de l'activité $f.destination$. Le code traduisant la mise en œuvre de ces points d'extensions et obtenu par un générateur de sorte permettra aux données de capteurs arrivant sur que les données de capteurs arrivant sur JO soient transférées à travers le réseau sur JI.

Cette politique contient également une propriété *platform* contenant la plateforme de déploiement. Nous utilisons le terme de **sous-politique de collecte de données** afin de dénommer une politique spécifique à la plateforme.

Définition 5.4.2 (Sous-politique de collecte de données)**.** Une sous-politique de collecte de données est un ensemble d'activités et de flux résultant du découpage d'une politique de collecte de données. Cette sous-politique est spécifique à une plateforme et contient des *join-points* assurant la continuité depuis/vers d'autres sous-politiques.

Enfin, dans l'objectif d'assurer la connectivité du réseau, pour toutes les plateformes $p \in P_{not} = platforms(I)$ P_{inv} non-impliquées dans la réalisation de π_{pre}, une politique relais est créée.

Définition 5.4.3 (Politique relais)**.** Une politique relais est composée d'un *join-point* d'entrée relié à un *join-point* de sortie. Son rôle est d'assurer la connectivité du réseau en transférant, sans effectuer d'opération, l'ensemble des valeurs reçues aux plateformes qui sont directement connectées à la plateforme exécutant cette politique.

Fil rouge : Sous-politiques de collecte de données

Les sous-politiques de collecte de données résultant du découpage de la politique de collecte de données à partir du positionnement des activités sont illustrées en FIG. 5.4.3. Nous remarquons que chaque plateforme impliquée dans la politique de collecte de données contient uniquement les activités dont elle est cible. Une politique relais est également créée pour les plateformes non-impliqués afin de garantir la connectivité du réseau.

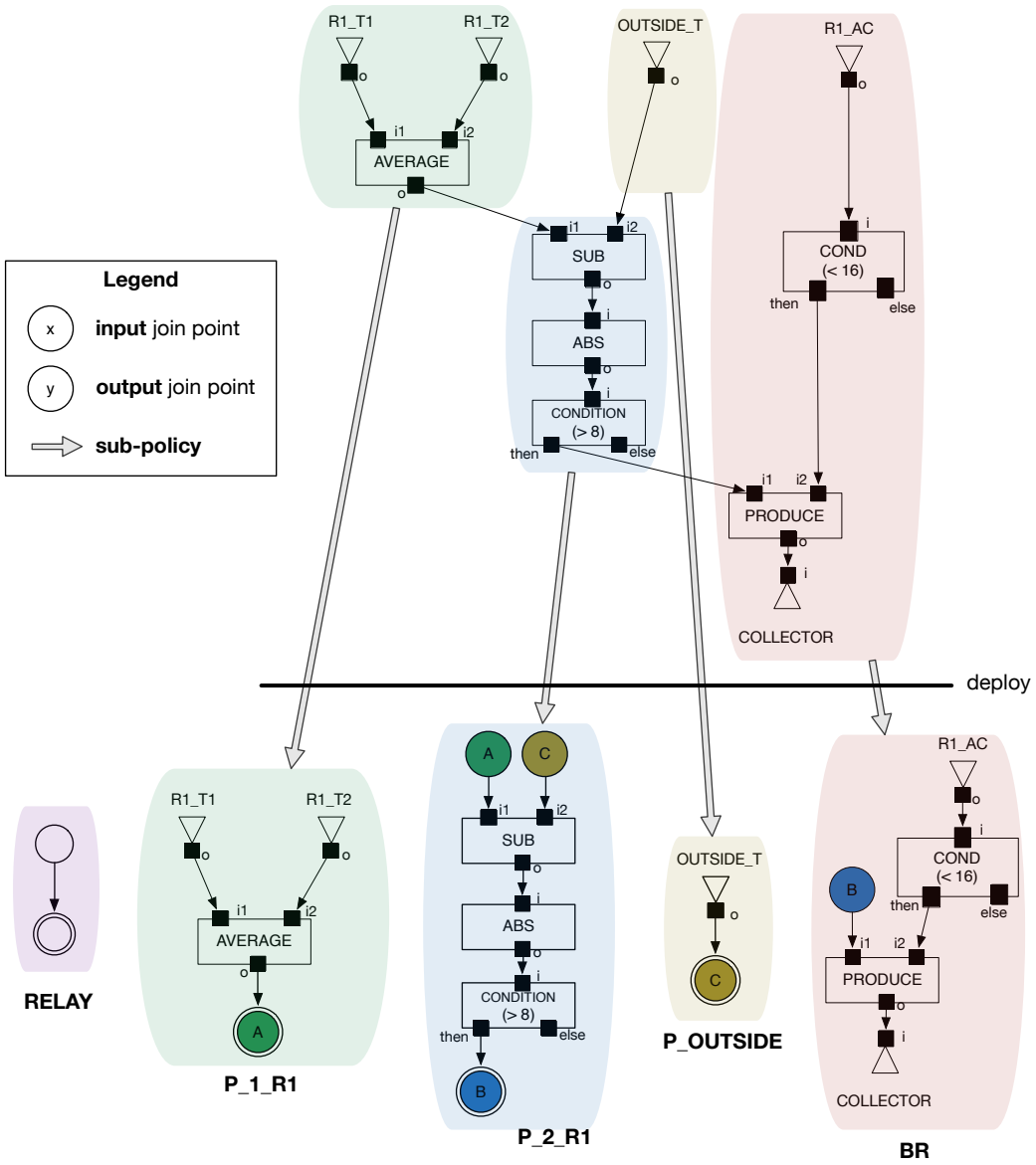


FIGURE 5.6 – Sous-politiques de collecte de données résultant du déploiement de la politique fil rouge

5.4.4 Synthèse

La politique de collecte de données, définie par un ingénieur logiciel, est indépendante de toute infrastructure sous-jacente. Tout comme les langages de macro-programmation, elle définit de manière centralisée comment l'infrastructure de capteur

doit traiter les données. En vue de déployer cette politique de collecte de données, nous avons introduit plusieurs opérateurs permettant d'associer à chaque activité, une plateforme de déploiement cible. Ensuite, pour chacune de ces plateformes, un opérateur *deploy*, génère, à partir du découpage de la politique de collecte de données, des sous-politiques de collecte de données regroupant les activités et flux de données à effectuer sur la plateforme. Ces sous-politiques de collecte de données pourront ensuite être traduites en code exécutable.

5.5 Génération de code

Les sous-politiques de collecte de données définies dans la section précédente contiennent l'ensemble des activités et flux de données spécifiques à une plateforme. Afin de traduire cette sous-politique en code exécutable, nous devons utiliser des générateurs de code prenant en compte la variabilité matérielle des différentes plateformes.

5.5.1 Problèmes liés à la génération de code

La génération du code traduisant une politique de collecte de données doit fournir un exécutable traduisant les intentions de l'ingénieur logiciel. Pour cela, le générateur de code doit respecter une sémantique d'exécution des activités et flux de données.

► ***Sémantique d'exécution***. Les générateurs de code doivent implémenter une sémantique d'exécution permettant d'obtenir du code conforme aux intentions de l'ingénieur logiciel.

De plus, la génération des activités et flux de données doit se faire dans un ordre tel que toutes les activités reçoivent des valeurs de capteur en provenance d'activités situées en amont dans la politique et puisse envoyer des valeurs de capteurs aux activités en aval.

► ***Ordre de génération***. Un générateur de code doit générer les activités dans un ordre assurant que pour une activité a, l'ensemble des activités en amont de a dans la politique soit généré avant a et que l'ensemble des activités en aval de a dans la politique soit généré après a.

Enfin, l'hétérogénéité des infrastructures de capteurs et le grand nombre de constructeurs présents sur le marché font que la génération de code est liée au matériel présent sur la plateforme. Par exemple, Lst. 5.6 illustre la différence du code mettant en œuvre une récupération de valeur de température entre deux capteurs de même type (température), mais de constructeurs différents.

Listing 5.6 – Récupération de la valeur de température à partir d'une thermo-résistance et d'un capteur LM35

```
// Code Grove - Temperature (thermo-resistance)
const int R0 = 100000;
int a = analogRead(pinTempSensor);
double R = 1023.0/a-1.0;
R = R0*R;
double temperature = 1.0/(log(R/R0)/B+1/298.15)-273.15; //Valeur de temperature en C

// Code ElectronicBricks - Temperature (capteur LM35)
uint16_t val=analogRead(A0);
double temperature = (double) val * 5.0 / 10.24 / 3.0; //Valeur de temperature en C
```

► ***Gestion de la variabilité matérielle***. Le générateur de code doit tenir compte de la variabilité matérielle d'une plateforme

5.5.2 Sémantique d'exécution

Nous produisons du code exécutable en nous basant sur les concepts d'exécution définis au sein de MOTEUR2 [GMLP08]. Cette sémantique d'exécution est particulièrement adaptée à notre contexte puisque MOTEUR2 est dédié à exécuter des processus métiers à forte consommation de données, notamment dans le traitement de données d'imagerie médicale.

Sémantique d'exécution d'une activité

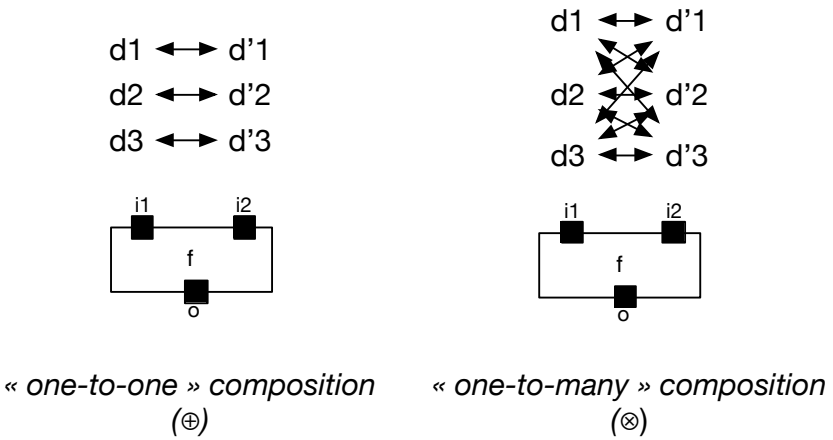


FIGURE 5.7 – Composition des données de capteur (*synchronisées*) au niveau des ports d'entrée

Une activité définie au sein d'une politique de collecte de données est caractérisée par des ports d'entrée, des ports de sortie et une fonction f à appliquer sur les données. Lorsqu'une activité présente plusieurs ports d'entrée, les données arrivant sur chacun des ports d'entrée doivent être composées en vue d'être traitées par la fonction f. Les deux types de composition de données les plus fréquemment rencontrés sont la composition *one-to-one* (dénotée $\oplus$) et *one-to-many* (dénotée $\otimes$) et illustrés dans FIG. 5.7. Dans le cas d'une politique de collecte de données, les données produites par les capteurs sont directement traitées à travers les activités. De ce fait, puisque nous ne disposons pas de liste de données de capteurs, nous faisons le choix d'utiliser uniquement la composition *one-to-one*.

Pour chacune des activités, lorsqu'une donnée de capteur est présente sur chacun des ports d'entrée, le code généré doit récupérer ces données, exécuter la fonction souhaitée par l'ingénieur logiciel et écrire le résultat de la fonction sur un ou plusieurs ports de sortie. Ces différentes étapes peuvent être représentées par un automate fini sans état terminal et reposant sur 4 états (FIG. 5.8) décrits ci-après : **IDLE**, **ACQUIRE**, **EXECUTE** et **SEND**.

▶ ***IDLE***. L'activité n'est pas exécuté tant que le signal *ready* n'est pas reçu.

▶ ***ACQUIRE***. Pour chacun des ports d'entrée et si et seulement si tous les ports d'entrée ont une donnée à leur disposition, alors l'activité produit le signal *begin*. Dans le cas contraire, l'activité produit le signal *stop*. Si l'activité ne possède pas de port d'entrée, *par ex.*, représente un capteur, alors le signal *begin* est immédiatement produit.

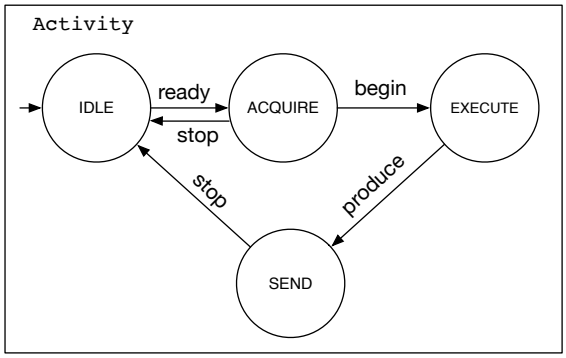


FIGURE 5.8 – Sémantique d'exécution d'une *activité*

▶ ***EXECUTE***. L'action représentée par l'activité est exécutée. Après l'exécution de l'action, l'activité produit le signal *produce*.

▶ ***SEND***. L'activité recopie le/les résultat(s) sur le/les port(s) de sortie puis produit le signal *stop* afin de retourner en état **IDLE**. Si l'activité ne possède pas de port de sortie, alors le signal *stop* est immédiatement produit.

Sémantique d'exécution d'une politique de collecte de données

Lorsque la politique de collecte de données est démarrée, *i.e.*, la plateforme est opérationnelle, la politique propage les données, à travers les flux de données, entre les différentes activités puis exécute ces dernières. Ces différentes étapes peuvent être représentées par un automate fini sans état terminal et reposant sur 3 états (FIG. 5.9) décris ci-après : **IDLE**, **PROPAGATE** et **ACTIVITIES**.

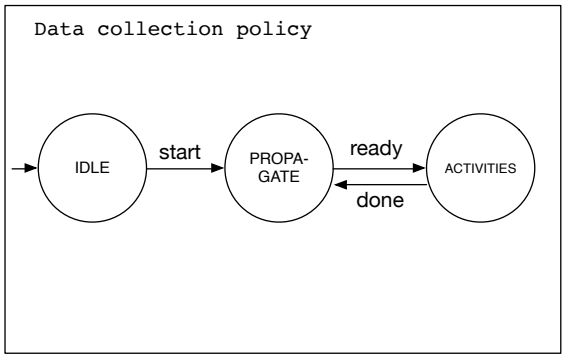


FIGURE 5.9 – Sémantique d'exécution d'une *politique*

▶ ***État IDLE***. La politique n'est pas exécutée tant que le signal *start* n'est pas produit.

▶ ***État PROPAGATE***. Pour chacun des flux de données, le moteur recopie la donnée présente en source sur la destination, *cf.* EQ. 5.7.

$$\forall f \in F, f.destination.data = f.source.data \tag{5.7}$$

Lorsque tous les flux de données ont été traités, le signal *ready* est produit.

▶ ***État ACTIVITIES***. Pour chacune des activités, la politique initie son exécution selon la sémantique présentée précédemment. Lorsque toutes les activités ont été traitées, le moteur génère le signal *done* afin de retourner en état **PROPAGATE**.

5.5.3 Production du code exécutable

Ordre de génération

En théorie des graphes, un tri topologique permet d'obtenir un ordre de visite des sommets de façon à ce qu'un sommet soit toujours visité avant ses successeurs. Grâce à un résolveur de contraintes, nous proposons d'étendre le tri topologique aux politiques de collecte de données (assimilée à des graphes orientés acycliques à multiples sources et puits). Pour une politique de collecte de données π, nous cherchons à associer un nombre à chaque activité correspondant à son ordre de génération.

Les activités correspondant aux sources de données (`INPUT`) doivent être générées en premier, nous leur attribuons donc la valeur 0. Inversement, les activités correspondant aux sorties de données (`OUTPUT`) doivent être générées en dernier, nous leur attribuons donc la valeur ∞. Nous générons ensuite une liste de contraintes à partir des flux de données f : $f.source < f.destination$. Cette contrainte permet de spécifier que l'activité source d'un flux de données soit toujours générée avant l'activité destination. Une solution satisfaisant ces contraintes est ensuite trouvée grâce à un outil tiers, *par ex.*, Choco [PFL16].

Gestion des capacités matérielles

La génération du code est liée au modèle des capacités afin d'adapter le code source produit aux éléments variables de la plateforme.

▶ ***Choix du générateur de code***. Le générateur de code γ approprié pour traduire une sous-politique de collecte de données sur une plateforme p est sélectionné selon le langage de programmation sélectionné dans le modèle du domaine décrivant la plateforme.

▶ ***Gestion de la variabilité des capteurs***. À partir du type de capteur `TYPE` et du constructeur `BRAND` renseignés dans le modèle du domaine, le générateur de code utilise une bibliothèque logicielle `TYPE_BRAND` exposant une méthode *acquire()*. Cette méthode met en œuvre la logique constructeur permettant de récupérer la valeur courante du capteur.

▶ ***Gestion de la variabilité des communications***. À partir du sens de communication et de la technologie de communication `TECHNOLOGY`, le générateur de code utilise une bibliothèque logicielle `TECHNOLOGY` exposant deux méthodes *send(sensor_value)* (utilisée pour les sens de communication sortant) et *receive(&sensor_value)* (utilisée pour les sens de communication entrant).

5.6 Conclusion

Dans ce chapitre, nous avons montré comment une politique de collecte de données définie par un ingénieur logiciel est déployée sur une infrastructure de capteurs. Afin de supporter le principe de séparation des préoccupations, l'ensemble des aspects fonctionnels de l'infrastructure de capteurs sont décrits par un expert réseau au sein d'un modèle d'infrastructure contenant *(i)* un modèle de topologie réseau, *(ii)* un modèle des fonctionnalités des plateformes et *(iii)* une stratégie de déploiement.

Différents opérateurs viennent à préparer et effectuer le déploiement d'une politique de collecte de données sur l'infrastructure de capteurs. Le déploiement s'effectue en deux temps : dans un premier temps, chaque activité est associée à une plateforme cible grâce à la stratégie de déploiement, puis dans un second temps, des sous-politiques spécifiques aux plateformes sont créées par sélection des activités et flux de données à déployer sur la plateforme.

Enfin, un ensemble de générateurs de code construit, en s'appuyant sur le modèle des fonctionnalités des plateformes et une sémantique d'exécution des activités et de la politique, un ensemble de fichiers sources prêts à être transférés sur les plateformes de l'infrastructure de capteurs.

Chapitre 6

Composition de politiques

Sommaire

6.1 Introduction

Une infrastructure de capteurs partagée permet l'exploitation des ressources sensorielles entre plusieurs applications et/ou plusieurs utilisateurs. Une telle infrastructure permet notamment de réutiliser le matériel afin d'éviter des redéploiements redondants lorsque de nouvelles applications sont identifiées [KBG+15].

Définition 6.1.1 (Infrastructure de capteurs partagée)**.** Une infrastructure de capteurs partagée est une infrastructure à accès ouvert capable de supporter l'exécution simultanée de plusieurs applications [LEMC12], et dans notre contexte, pouvant provenir de plusieurs ingénieurs logiciels.

6.1.1 Problèmes identifiés

L'étude de l'état de l'art a soulevé que les approches actuelles de partage de l'infrastructure ne permettaient pas le partage au niveau de la plateforme de capteurs au sein d'une infrastructure hétérogène.

Le partage des plateformes est pourtant un point crucial dans le passage à l'échelle des infrastructures de capteurs puisqu'il permet de réutiliser une même plateforme pour exécuter différentes applications. De plus, la gestion de l'hétérogénéité permet de partager des plateformes de constructeurs et technologies différentes.

Enfin, les approches actuelles visent à déployer l'ensemble des programmes en une seule fois sur les plateformes alors que de nouveaux besoins peuvent être identifiés après le déploiement de politiques de collecte de données.

6.1.2 Objectifs de ce chapitre

Ce chapitre cible l'objectif (O_4) :

(O_4) Une infrastructure est partagée entre différentes politiques

Lors du déploiement d'une nouvelle politique, plusieurs autres politiques peuvent déjà être en cours d'exécution sur l'infrastructure de capteurs, *cf.* (C_2). En raison de la séparation des préoccupations entre les points de vue du logiciel et du réseau, un ingénieur logiciel n'a pas à connaitre les différentes politiques déjà déployées. Un mécanisme de partage devrait donc gérer le partage de l'infrastructure entre plusieurs politiques de manière transparente et automatique pour l'ingénieur logiciel.

6.1.3 Plan du chapitre

Dans une première partie Sec. 6.2, nous introduisons un opérateur de composition agissant au niveau des politiques de collecte de données. Cet opérateur permet de composer deux politiques de collecte de données en une nouvelle politique. Ensuite, dans une seconde partie Sec. 6.3, nous présentons un proxy de composition placé entre les ingénieurs logiciels et l'infrastructure de capteurs et ayant pour but d'assurer le partage de cette infrastructure et de permettre le déploiement de nouvelles politiques de collecte de données. Enfin, dans une dernière partie Sec. 6.4, nous concluons ce chapitre.

Contributions présentées

Ce chapitre présente les contributions suivantes :

— Un opérateur de composition $\oplus$ agissant au niveau des politiques de collecte de données

— Un proxy de composition permettant le partage d'une infrastructure de capteurs en collectant des politiques de collecte de données pouvant provenir de plusieurs ingénieurs logiciels, en les composant avec la politique couramment exécutée sur l'infrastructure, et en procédant au déploiement de la politique composée.

6.2 Déploiement de plusieurs politiques de collecte de données sur une même infrastructure

6.2.1 Problème liés à l'exécution simultanée de politiques sur une infrastructure hétérogène

Le problème du partage des ressources se rencontre aussi bien au niveau logiciel que matériel.

Au sein des approches de *cloud computing*, le problème du partage d'un serveur entre plusieurs applications a été résolu grâce aux techniques d'isolation [AFG+10] : un ensemble de machines virtuelles est exécuté sur le serveur et chaque machine virtuelle est dédiée à une application.

Au niveau matériel, les processeurs peuvent exécuter de manière séparée et concurrente plusieurs flux d'exécution grâce aux techniques de *mutli-threading* [TEL95].

Les infrastructures de capteurs sont principalement composées de plateformes hétérogènes et potentiellement à ressources contraintes ne pouvant reprendre les concepts de virtualisation ou de *multi-threading*.

Les protothreads introduits en 2006 et utilisés au sein du système d'exploitation Contiki [DSVA06] ont introduit un paradigme permettant une exécution pseudo-parallèle de flux d'exécution grâce à un mécanisme d'ordonnancement dirigé par les évènements. Le code Lst. 6.1 illustre l'utilisation de protothreads permettant l'exécution simultanée de deux flux d'exécution : *Process #1* affichant le message `Process #1` toutes les secondes et *Process #2* affichant le message `Process #2` toutes les 5 secondes. Si cette approche permet de répondre à la problématique du partage d'une plateforme, elle impose l'utilisation de matériel supportant Contiki, rendant alors l'approche dépendante du type de plateforme utilise (contraire au critère 11 de l'état de l'art, *cf.* Sec. 2.3).

Listing 6.1 – Exemple d'exécution simultanée de deux flux d'exécution à l'aide de protothreads Contiki

```
#include "contiki.h"
#include <stdio.h> // Only for using printf

PROCESS(process1, "Process␣#1");
PROCESS(process2, "Process␣#2");

AUTOSTART_PROCESSES(&process1, &process2); // Start the processes

PROCESS_THREAD(process1, ev, data)
{
        static struct etimer timer;
```

```
        PROCESS_BEGIN();
        etimer_set(&timer, CLOCK_SECOND); //Triggers an event every second
        while(1)
        {
                PROCESS_WAIT_EVENT();
                if(ev == PROCESS_EVENT_TIMER)
        {
                printf("Process␣#1␣\n");
            etimer_reset(&timer); // Reset the timer
        }
        }
        PROCESS_END();
}

PROCESS_THREAD(process2, ev, data)
{
        static struct etimer timer;
        PROCESS_BEGIN();
        etimer_set(&timer, CLOCK_SECOND * 5); //Triggers an event every 5s
        while(1)
        {
                PROCESS_WAIT_EVENT();
                if(ev == PROCESS_EVENT_TIMER)
        {
                printf("Process␣#2␣\n");
            etimer_reset(&timer); // Reset the timer
        }
        }
        PROCESS_END();
}
```

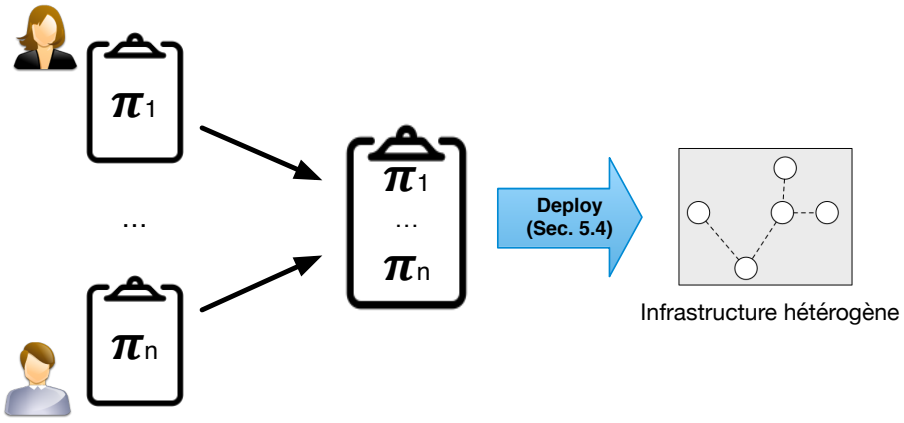


FIGURE 6.1 – Illustration à haut-niveau de la composition de politiques avant leur déploiement sur une infrastructure hétérogène

Au cours du chapitre précédent (CHAP. 5), nous avons présenté comment une politique de collecte de données, exprimée indépendamment du matériel, était déployée sur une infrastructure de capteurs hétérogène. Nous proposons de réutiliser les techniques d'adaptation et de projection définies dans ce précédent chapitre afin de déployer une politique composée (nous rappelons que notre hypothèse H_3 définit le résultat de la composition comme étant une nouvelle politique de collecte de données à part entière). Pour cela, nous procéderons à l'étape de composition en amont du déploiement comme illustré sur la figure FIG. 6.1.

6.2.2 Principe de la composition de politiques

Le déploiement d'une politique sur une infrastructure hétérogène est effectué grâce à l'opérateur *deploy* (*cf.* Eq. 5.6) prenant comme opérandes une politique de collecte de données et un modèle d'infrastructure. Nous rappelons que cet opérateur a pour objectif de décomposer une politique de collecte de données globale et indépendante du matériel en sous-politiques de collecte de données spécifiques aux plateformes.

Afin de réutiliser cet opérateur pour déployer plusieurs politiques de collecte de données, il est nécessaire de les composer en une nouvelle politique. Nous introduisons l'opérateur de composition noté $\oplus$ (*cf.* Eq. 6.1), composant deux politiques de collecte de données π_a et π_b et retournant une nouvelle politique $\pi_\oplus = \pi_a \oplus \pi_b$.

$$\oplus : \Pi \times \Pi \rightarrow \Pi \tag{6.1}$$

Ainsi, soient $i \in I$ un modèle d'infrastructure et $\pi_\oplus$, une politique résultante de l'opérateur $\oplus$ et adaptée pour i, le déploiement de $\pi_\oplus$ sur i s'effectuera de la manière suivante : $deploy(\pi_\oplus, i)$

Nous définissons, ci-dessous, les propriétés auxquelles doit répondre cet opérateur :

▶ ***Indépendance des opérations***. Afin de respecter les principes d'isolation, la composition de deux politiques de collecte de données π_a et π_b doit assurer que les opérations de π_a restent indépendantes des opérations de π_b, *i.e.*, aucun flux de donnée de π_b interagit avec des opérations de π_a, et que les opérations de π_b restent indépendantes des opérations de π_a. Cette propriété se traduit à l'aide de l'invariant suivant :

$$\begin{aligned}(\pi_a, \pi_b) \in \Pi^2, & O_a = \text{operations}(\pi_a), O_b = \text{operations}(\pi_b),\\ & F_a = \text{flows}(\pi_a), F_b = \text{flows}(\pi_b),\\ & (\nexists f \in F_a, f.\text{source} \in O_b \vee f.\text{destination} \in O_b) \wedge\\ & (\nexists f \in F_b, f.\text{source} \in O_a \vee f.\text{destination} \in O_a)\end{aligned} \tag{6.2}$$

▶ ***Fusion des capteurs identiques***. Différentes politiques de collecte de données peuvent faire appel aux mêmes capteurs. Si π_a et π_b contiennent des capteurs identiques (Def.6.2.1), alors ils ne pourront être appelés simultanément sur une plateforme à unique flux d'exécution. Ces capteurs identiques doivent alors être fusionnés en un unique capteur lors de la composition comme illustrée en Fig. 6.2.

Définition 6.2.1 (Capteurs identiques)**.** Soient s_1 et s_2 deux capteurs. $s_1 = s_2$ s.s.i. s_1 et s_2 sont deux capteurs ayant le même nom, le même type de données et *(i)* dans le cas de deux capteurs périodiques, la même période d'échantillonnage, *(ii)* dans le cas de deux capteurs évènementiels, un échantillonnage déclenché par le même évènement.

▶ ***Commutativité***. Étant donné que les politiques sont exprimées indépendamment les unes des autres, l'ordre dans lequel elles sont composées ne doit pas avoir d'impact sur le résultat de la composition. Ainsi, l'opérateur de composition doit être commutatif :

$$\forall(\pi_a, \pi_b) \in \Pi^2, \pi_a \oplus \pi_b = \pi_b \oplus \pi_a \tag{6.3}$$

6.2.3 L'opérateur de composition $\oplus$

Les politiques de collecte de données peuvent être assimilées à des graphes orientés [VVLM08] où les activités représentent les sommets, et les flux de données représentent les arcs. Dans le domaine de la théorie des graphes, les graphes à double

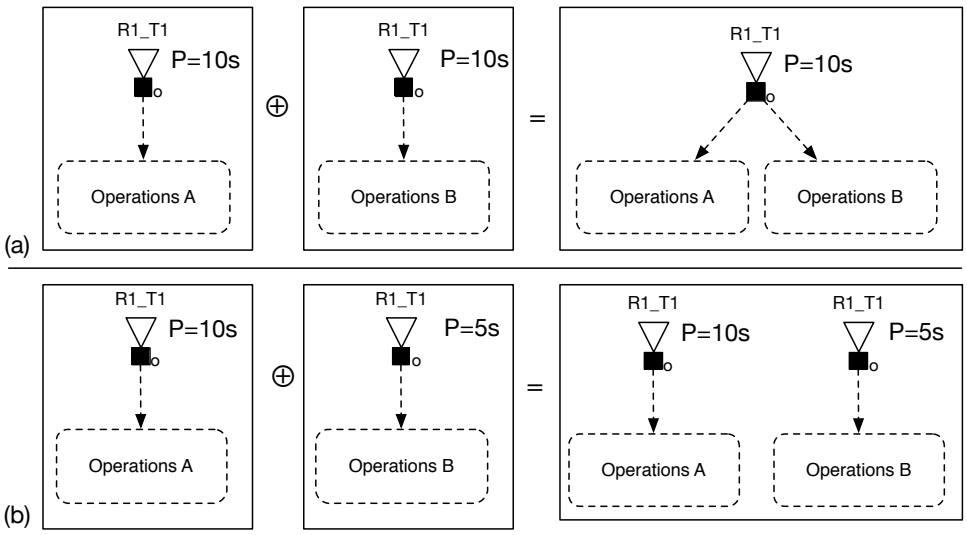


FIGURE 6.2 – Illustration de la fusion de capteurs. *(a)* fusion de deux capteurs identiques. *(b)* aucune fusion, car les capteurs ne sont pas identiques.

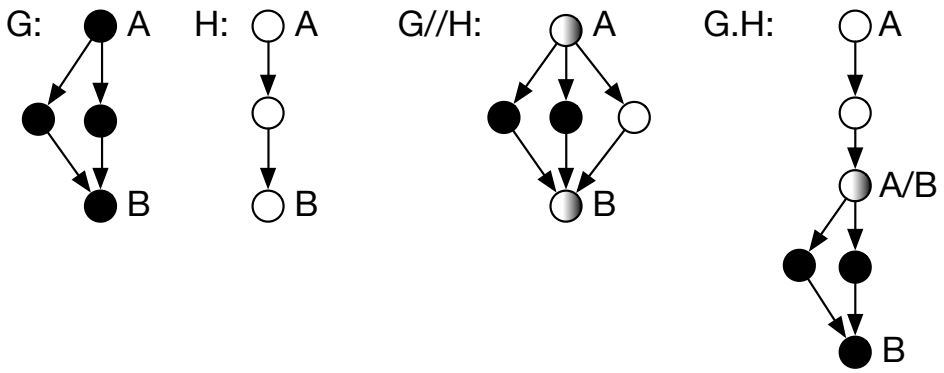


FIGURE 6.3 – Illustration des compositions *mise en parallèle* et *mise en série*

terminaux (*Two-terminal graph* ou TTG) sont une classe de graphes où chaque graphe possède deux sommets particuliers : s et t, respectivement *source* et *puit* [Epp92]. Ces graphes supportent deux types d'opérations de composition : *mise en parallèle (//)* et *mise en série (.)*, illustrés sur la figure FIG. 6.3. Soient deux graphes G et H :

— La composition *mise en parallèle* (opération P) fusionne deux TTG X et Y ayant les sommets s_x, t_x, s_y et t_y en formant un nouveau graphe $G = P(X, Y)$ et en identifiant $s = s_x = s_y$ et $t = t_x = t_y$.

— La composition *mise en série* (opération S) fusionne deux TTG X et Y ayant les sommets s_x, t_x, s_y et t_y en formant un nouveau graphe $G = S(X, Y)$ et en identifiant $s = s_x$, $t_x = s_y$ et $t = t_y$.

Dans le cadre des politiques de collecte de données, les deux types de composition assurent l'indépendance des opérations puisque la composition s'effectue au niveau des sources ou puits de données. Cependant, seule la composition *mise en parallèle* assure la fusion des capteurs identiques. De plus, l'utilisation d'une composition *mise en série*

entre deux politiques de collecte de données fusionnerait un puit de données avec une source de données ce qui est, par définition, *invalide*.

Il est nécessaire d'étendre la composition *mise en parallèle* aux politiques de collecte de données puisque ces dernières peuvent posséder plusieurs capteurs, *i.e.*, sources du graphe TTG. Enfin, la composition ne doit pas fusionner les collecteurs, *i.e.*, puits du graphe TTG, qui, le cas échéant, provoquerait l'arrivée de plusieurs flux de données sur l'unique port d'entrée du collecteur fusionné, contraire à la règle de validité 4.5.

Nous effectuons cette extension de la composition *mise en parallèle* en deux étapes détaillées ci-après.

Union des politiques

Soient deux politiques de collecte de données π_a (assimilée au graphe A) et π_b (assimilée au graphe B), l'union de $A \cup B$ crée la politique $\pi_{a \cup b}$ (*cf.* FIG. 6.4) :

$$\pi_{a \cup b} = \langle name, activities(\pi_a) \cup activities(\pi_b), flows(\pi_a) \cup flows(\pi_b) \rangle \tag{6.4}$$

La propriété de commutativité de l'union donne les relations EQ. 6.5 et EQ. 6.6 validant la propriété de commutativité de l'opérateur $\oplus$.

$$activities(\pi_a) \cup activities(\pi_b) = activities(\pi_b) \cup activities(\pi_a) \tag{6.5}$$

$$flows(\pi_a) \cup flows(\pi_b) = flows(\pi_b) \cup flows(\pi_a) \tag{6.6}$$

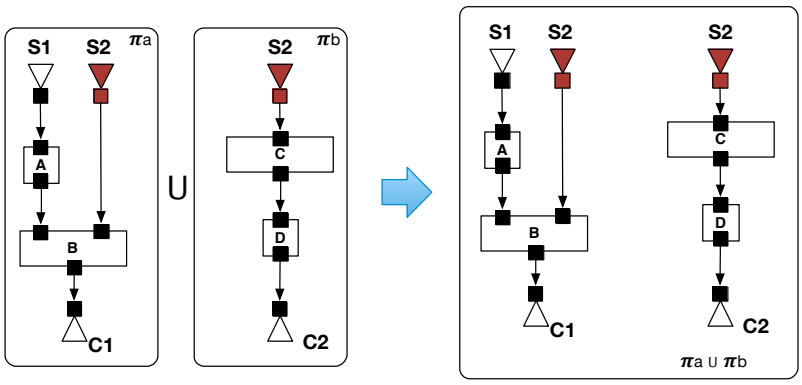


FIGURE 6.4 – Union de π_a avec π_b (le capteur $S2$ est identique entre les politiques)

Fusion des capteurs identiques

Au sein de la politique $\pi_{a \cup b}$, les capteurs périodiques et événementiels identiques (*cf.* DEF.6.2.1) sont fusionnés.

Soient deux politiques de collecte de données π_a et π_b et deux capteurs $s_a \in sensors(\pi_a)$ et $s_b \in sensors(\pi_b)$ tels que $s_a = s_b$, alors la fusion de ces capteurs s'effectue par *(i)* création d'un capteur s_c tel que $s_c = s_b = s_a$, *(ii)* par la création de flux de données entre s_c et les ports des activités étant destination de flux de données ayant eux-mêmes pour source s_a ou s_b et *(iii)* suppression des capteurs s_a et s_b ainsi que les flux de données ayant ces capteurs comme source (*cf.* FIG. 6.5).

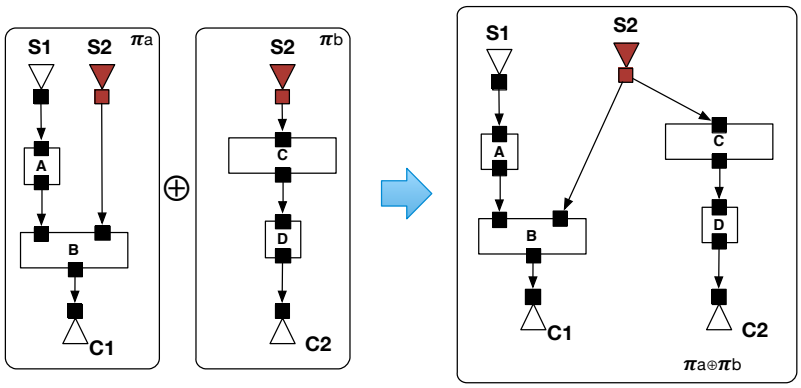


FIGURE 6.5 – Politique $\pi_{a \oplus b}$ obtenue après fusion du capteur $S2$

Les codes LST. 6.2 et LST. 6.3 présentent une mise en œuvre Scala de la méthode « `=` » (permettant d'identifier un capteur identique) au sein des classes `PeriodicSensor` et `EventSensor` du méta-modèle illustré en FIG. 4.4 et le code LST. 6.4 détaille la mise en œuvre Scala de l'opérateur de composition. Une méthode utilitaire `findIdentical` identifie à partir d'une liste de capteurs, les capteurs identiques (à partir des fonctions `=` présentées en LST. 6.2 et LST. 6.3) et retourne une liste d'ensemble de capteurs identiques, *par ex.*, `findIdentical(List(`s_a`,` s_a`,` s_b`,` s_c`,` s_c`) = List((`s_a`,` s_a`), (`s_c`,` s_c`))`.

Listing 6.2 – Mise en œuvre Scala de la vérification de l'identité entre deux capteurs périodiques (méthode au sein de la classe `PeriodicSensor` du méta-modèle)

```
override def =(x: Any): Boolean = x.isInstanceOf[PeriodicSensor[T]] &&
    (x.asInstanceOf[PeriodicSensor[T]].dataType equals this.dataType) &&
    (x.asInstanceOf[PeriodicSensor[T]].name equals this.name) &&
    (x.asInstanceOf[PeriodicSensor[T]].period equals this.period)
```

Listing 6.3 – Mise en œuvre Scala de la vérification de l'identité entre deux capteurs évènementiels (méthode au sein de la classe `EventSensor` du méta-modèle)

```
override def =(x: Any): Boolean = x.isInstanceOf[EventSensor[T]] &&
    (x.asInstanceOf[EventSensor[T]].dataType equals this.dataType) &&
    (x.asInstanceOf[PeriodicSensor[T]].name equals this.name) &&
    (x.asInstanceOf[EventSensor[T]].trigger equals this.trigger)
```

TABLEAU 6.1 – Périodes d'échantillonnage pour les capteurs impliqués dans $\pi_{\text{thermalshock}}$ et $\pi_{\text{energylosses}}$ (en gras : périodes identiques)

	R1_T1	R1_T2	R1_AC	OUTSIDE_T
$\pi_{\text{thermalshock}}$	300	300	**60**	**3600**
$\pi_{\text{energylosses}}$	60	60	**60**	**3600**

Listing 6.4 – Code Scala mettant en œuvre l'opérateur $\oplus$

```
def compose(p1:Policy, p2:Policy): Policy = {
    // new composed policy definition
    val p = new Policy()
    p.activities = p1.activities union p2.activities
    p.flows = p1.flows union p2.flows

    // Merging
    val toMerge = findIdentical(p.sensors)
    toMerge.forEach { e => {
              val newSensor = e.head.duplicate
              val newFlows = p.flows.filter(f => e contains f.source).map{f =>
                 new Flow(newSensor.output, f.destination_input)
              }

             p.add(newSensor)
             newFlows.foreach(p.add)
             e.forEach(p.delete)
         }
    }

    p //Return the composed policy
}
```

6.2.4 Exemple illustratif de la composition entre deux politiques

Pour illustrer l'application de l'opérateur $\oplus$, nous allons composer la politique fil rouge $\pi_{\text{thermalshock}}$ avec une nouvelle politique $\pi_{\text{energylosses}}$ permettant de détecter les déperditions énergétiques d'un bureau *cf.* FIG. 6.6.

La politique $\pi_{\text{energylosses}}$ repose sur les capteurs `R1_T1`, `R1_T2`, `R1_AC` et `OUTSIDE_T`. Elle a pour objectif de retourner une alerte si la température extérieure est inférieure à la température moyenne du bureau alors que la climatisation est en fonctionnement (il est donc préférable d'ouvrir la fenêtre pour refroidir le bureau).

Les périodes d'échantillonnage des capteurs impliqués dans ces deux politiques sont présentées en TAB. 6.1 et tous les capteurs produisent le même type de données. Le code Scala LST. 6.5 illustre l'utilisation de l'opérateur $\oplus$ au sein du langage spécifique au domaine.

Listing 6.5 – Utilisation de l'opérateur $\oplus$

```
    this hasForName "Composition_Example"
    EnergyLosses() + ThermalShockPrevention()
```

La politique composée obtenue est illustrée en FIG. 6.7. Cette nouvelle politique ne contient pas de capteurs identiques (`R1_AC` et `T_OUTSIDE` ont été fusionnés) et les opérations restent indépendantes puisqu'il n'y a pas de flux de données partagé entre les opérations initialement contenues dans $\pi_{\text{thermalshock}}$ et les opérations initialement contenues dans $\pi_{\text{energylosses}}$. Cette politique est ensuite déployable sur une infrastructure de capteurs en identifiant les plateformes candidates à la mise en œuvre des activités (*cf.* EQ. 5.5) et en obtenant les sous-politiques spécifiques aux plateformes cibles du

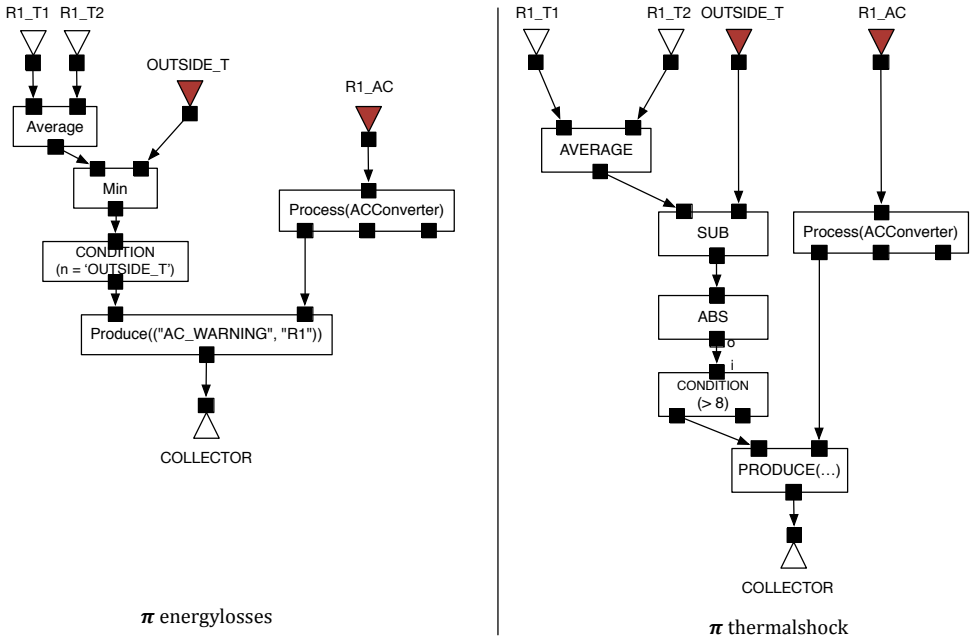


FIGURE 6.6 – Politiques $\pi_{\text{energylosses}}$ et $\pi_{\text{thermalshock}}$ (en rouge : capteurs identiques)

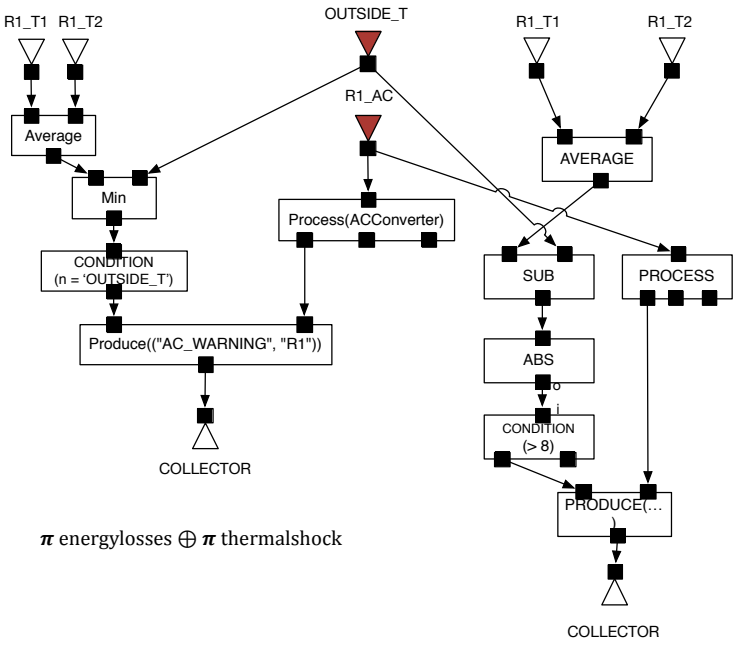


FIGURE 6.7 – Résultat de la composition $\pi_{\text{thermalshock}} \oplus \pi_{\text{energylosses}}$ (en rouge : capteurs fusionnés)

déploiement (*cf.* EQ. 5.6). Par décomposition de la politique en sous-politiques, les plateformes cibles mettent en œuvre des activités initialement (*i.e.*, avant composition) contenues dans des politiques différentes. Les plateformes de l'infrastructure de capteurs hétérogène sont ainsi partagées entre plusieurs politiques sans avoir eu la nécessité d'introduire une complexité supplémentaire au déploiement.

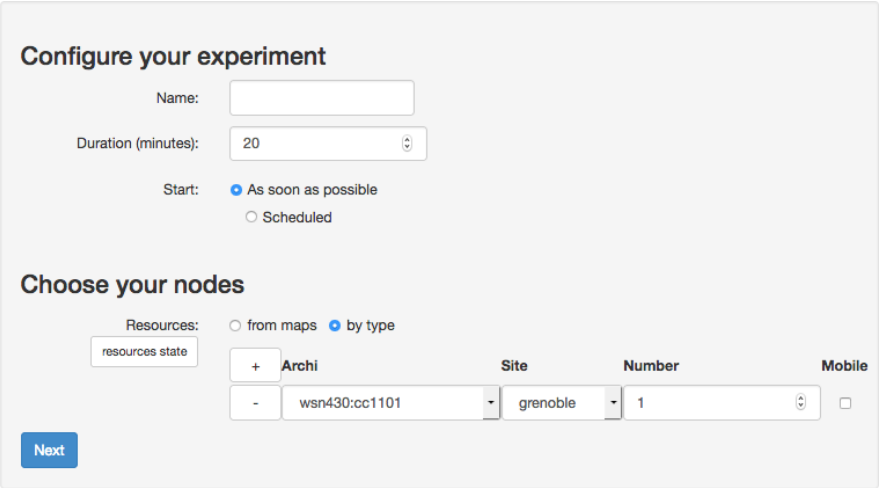


FIGURE 6.8 – Interface du service de réservation de l'infrastructure de capteurs partagée *FIT IoT-Lab* [ABF+15]

6.3 Déploiement de plusieurs politiques de collecte de données sur une infrastructure partagée

6.3.1 Problèmes liés au déploiement sur des infrastructures partagées

Nous avons vu au sein de la partie précédente que l'application de l'opérateur $\oplus$ (*cf.* EQ. 6.1) sur des politiques de collecte de données avant leur déploiement permet la création d'une nouvelle politique composée. Cette politique composée reprend les besoins métiers exprimés au sein de politiques individuelles et permet, après création des sous-politiques grâce à l'opérateur *deploy*, aux différentes plateformes de l'infrastructure hétérogène d'être partagées entre plusieurs applications.

Cependant, dans le cadre d'infrastructures partagées comme les projets SmartSantander [SMG+14] ou FIT IoT-Lab [ABF+15], un ingénieur logiciel n'a pas connaissance des politiques préalablement déployées. Il ne peut donc pas composer sa politique avec les politiques déployées par d'autres ingénieurs logiciels.

Actuellement, le problème du partage des ressources entre plusieurs utilisateurs dans les infrastructures de capteurs ou infrastructures type *IaaS*[1] est pris en charge à travers un **service de réservation**.

Le service de réservation a pour responsabilité de réserver un certain nombre de ressources dédiées à un utilisateur durant une période donnée, *par ex.*, FIG. 6.8. Toutefois, son utilisation soulève deux problèmes :

— *Passage à l'échelle :* la limitation d'un utilisateur par plateforme implique que lorsque le nombre d'utilisateurs augmente, le passage à l'échelle ne peut se faire qu'en augmentant le nombre de plateformes physiques disponibles et donc le coût financier et les opérations de maintenance.

— *Limitation dans le temps :* l'utilisation des plateformes affectées à un utilisateur est limitée dans le temps pour ensuite rendre ces plateformes de nouveau disponibles auprès d'autres utilisateurs. Une politique de collecte de données déployée sur ces plateformes aura donc une durée de vie limitée.

1. Infrastructure as a Service

Afin de pallier ces limitations, nous proposons une approche de partage automatique de l'infrastructure.

6.3.2 Partage automatique de l'infrastructure

Une infrastructure partagée doit permettre le partage des plateformes de capteurs afin de faciliter son passage à l'échelle lors du déploiement de nouvelles politiques de collecte de données. De plus, étant donné qu'un ingénieur logiciel ne peut pas manipuler directement l'opérateur $\oplus$ sur de telles infrastructures (il ne possède pas la définition de l'ensemble des politiques déployées), un mécanisme tiers doit composer automatiquement sa politique avec celles précédemment déployées.

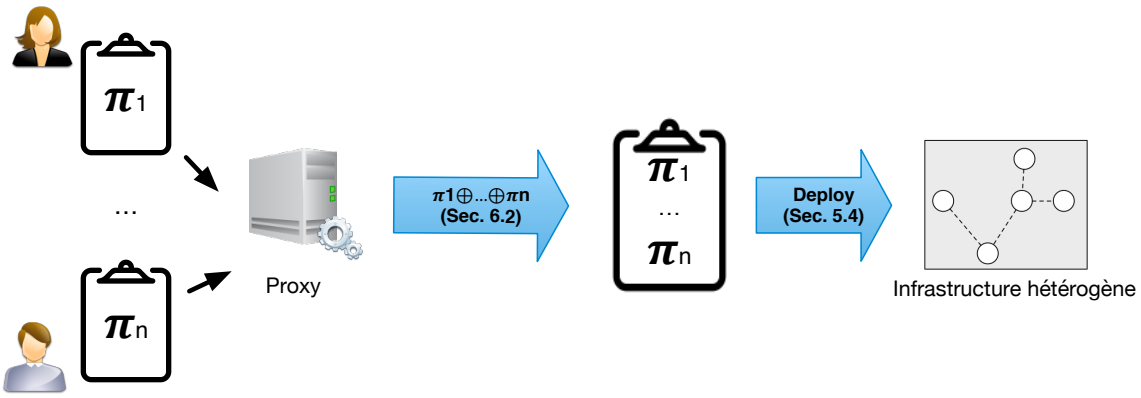


FIGURE 6.9 – Proxy entre les ingénieurs logiciels et l'infrastructure de capteurs hétérogènes (les étapes de composition et de déploiement sont initiées par le proxy)

Un proxy est défini comme étant un composant informatique se plaçant entre deux systèmes et interceptant leurs échanges. La délégation des opérations de composition à un proxy est ainsi une bonne solution puisque, par sa position centrale, il connait l'ensemble des politiques qui ont été déployées sur l'infrastructure de capteurs *cf.* FIG. 6.9. De plus, en gérant automatiquement la récupération des politiques et leur composition, ce proxy permet à n'importe quelle infrastructure de capteurs d'être partagée entre plusieurs utilisateurs. Pour mener à bien ces rôles, le proxy doit répondre aux propriétés suivantes :

▶ ***Vision globale de l'infrastructure***. Le proxy doit posséder une représentation du modèle d'infrastructure (*cf.* DEF.5.3.3) afin de connaitre la topologie, les plateformes déployées, ainsi que la stratégie de déploiement choisie par l'expert réseau.

▶ ***Historique***. Le proxy doit connaitre les politiques globales déployées ainsi que les sous-politiques déployées sur les plateformes de l'infrastructure.

▶ ***Composition automatique***. Lorsqu'un ingénieur logiciel souhaite déployer une nouvelle politique de collecte de données sur une infrastructure partagée, le proxy doit composer cette politique automatiquement avec les politiques globales précédemment déployées.

6.3.3 Définition du service de composition

Afin de répondre aux propriétés énoncées précédemment, nous définissons un proxy `InfrastructureManager` (FIG. 6.10). Pour répondre au besoin *Vision globale de l'in-*

frastructure, le proxy contient une représentation du modèle d'infrastructure. Ensuite, pour répondre au besoin *Historique*, le proxy maintient une représentation de la politique couramment exécutée sur l'infrastructure. Enfin, pour répondre au besoin *Composition automatique*, le proxy compose automatiquement la nouvelle politique avec la politique couramment exécutée et régénère un ensemble de fichiers sources prêts à être déployés par un expert réseau. Nous permettons également de récupérer les différentes sous-politiques déployées sur les plateformes de l'infrastructure de capteurs afin que les stratégies de déploiement dynamiques (*cf.* SEC. 5.3) puissent avoir une vision des politiques exécutées sur chaque plateforme.

Le proxy ainsi défini supporte l'ensemble des déploiements de politiques de collecte de données pour une infrastructure de capteurs donnée et expose trois méthodes :

— `registerPolicy(policy)` : ajout de la politique de collecte de données passée en paramètre sur l'infrastructure de capteurs ;

— `getPolicy()` : retour de la politique *composée* actuellement exécutée sur l'infrastructure ;

— `getPolicy(platform)` : retour de la sous-politique déployée sur une plateforme passée en paramètre.

Le code LST. 6.6 présente la mise en œuvre Scala du proxy `InfrastructureManager`. Il contient une représentation de la politique de collecte de données actuellement exécutée sur l'infrastructure de capteurs (L.3), une représentation de chacune des politiques déployées (L.4) et une structure de données associant une plateforme à une sous-politique (L.5). Nous décrivons ci-après la succession d'actions effectuée automatiquement par le proxy lors de l'ajout d'une nouvelle politique :

Lorsqu'un ingénieur logiciel souhaite déployer une nouvelle politique de collecte de données, il procède à son enregistrement grâce à la méthode `registerPolicy`. Cette méthode ajoute la nouvelle politique à la liste des politiques préalablement composées (L.8) et redéfinit la politique actuellement exécutée comme étant le résultat de la composition de la nouvelle politique avec la politique courante (L.9). Cette méthode applique ensuite la procédure de *prédéploiement* (L.11) puis l'opérateur *deploy* afin de décomposer la politique en sous-politiques spécifiques aux plateformes (L.13). Étant donné que ce dernier opérateur reconstruit les sous-politiques, la structure de données *subPolicies* est réinitialisée (L.12). Enfin, chacune des sous-politiques est associée à une plateforme (L.15) puis est traduite en code source prêt à être transféré sur les plateformes de l'infrastructure de capteurs (L.16).

Ainsi, lors du déploiement d'une nouvelle politique de collecte de données, cette dernière est automatiquement composée avec les politiques précédemment déployées

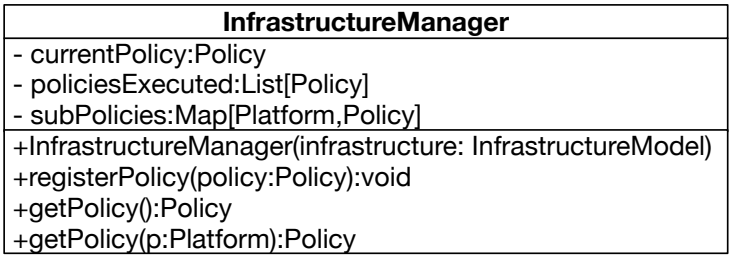


FIGURE 6.10 – Proxy `InfrastructureManager`

sans que l'ingénieur logiciel ait à connaitre l'ensemble des politiques exécutées sur l'infrastructure de capteurs.

Listing 6.6 – Mise en œuvre Scala du proxy `InfrastructureManager`

```
class InfrastructureManager(infrastructure:InfrastructureModel) {

  private var currentPolicy:Policy = _
  private var policiesExecuted:List[Policy] = List()
  private var subPolicies:Map[Platform, Policy] = Map()

  def registerPolicy(policy: Policy):Unit = {
    policiesExecuted = policy :: policiesExecuted
    currentPolicy = currentPolicy + policy

    val preDeployed = PreDeploy(currentPolicy, infrastructure.topology)
    subPolicies = Map()
    Deploy(preDeployed, infrastructure.topology, infrastructure.strategy).
        foreach{ policy =>
            subPolicies = subPolicies + (policy.target(), policy)
            policy.generate()
        }
  }
  def getPolicy():Policy = currentPolicy
  def getPolicy(p:Platform) = subPolicies(p)
}
```

6.4 Conclusion

Au cours de ce chapitre, nous avons adressé le problème de déploiement multiple de plusieurs politiques de collecte de données sur une même infrastructure contenant des plateformes hétérogènes. Afin de permettre l'exécution simultanée de plusieurs politiques sur une infrastructure contenant des plateformes hétérogènes et des systèmes contraints, nous avons défini un opérateur de composition $\oplus$ agissant au même niveau des politiques de collecte de données. Cet opérateur fusionne les capteurs identiques entre plusieurs politiques et assure l'indépendance des opérations en se basant sur une composition de graphe parallèle.

Nous nous sommes également intéressés au problème de déploiement de politiques sur des infrastructures de capteurs partagées. La solution classique de réserver un ensemble de plateformes pour un ingénieur logiciel soulève des problématiques de passage à l'échelle (une plateforme est dédiée à un unique ingénieur logiciel) et de limitation temporelle des activités. Pour pallier ces limitations, nous offrons la possibilité de partager une infrastructure de capteurs grâce à un proxy de composition. Ce proxy a pour but de collecter les différentes politiques, de les composer et de régénérer des fichiers sources prêts à être déployés sur l'infrastructure de capteurs.

Chapitre 7

Validation et évaluation à l'échelle de villes et bâtiments intelligents

Sommaire

7.1 Introduction

Nous présentons dans ce chapitre une validation des objectifs identifiés en Chap. 3 grâce à la définition et au déploiement de scénarios liés aux *bâtiments intelligents*. Ensuite, nous procédons à une évaluation à l'échelle de villes et bâtiments intelligents de l'approche afin de vérifier son applicabilité sur des scénarios liés aux larges infrastructures de capteurs.

▶ ***Prototype DEPOSIT***. L'ensemble des travaux présentés dans les chapitres précédents a été mis en œuvre au sein d'un logiciel appelé DEPOSIT [1]. Ce logiciel a été développé en Scala 2.11 (environ 8500 lignes de code) durant les trois années de recherche qui ont conduit à ces résultats. Afin de s'assurer de la non-régression du logiciel lorsque de nouvelles fonctions étaient ajoutées, 92 tests de spécifications ont été saisis et un mécanisme d'intégration continue assure le maintien du fonctionnement du logiciel au fur et à mesure du développement.

DEPOSIT repose sur différents *packages* :

— `deposit.core` : ce *package* contient une mise en œuvre du méta-modèle présenté en Fig. 4.4.

— `deposit.dsl` : ce *package* contient la définition du langage spécifique au domaine permettant de manipuler les éléments du méta-modèle.

— `deployment.infrastructure` : ce *package* contient les classes permettant de créer le modèle d'infrastructure (modèle des fonctionnalités des plateformes, modèle topologique et stratégie de déploiement).

— `deployment.generator` : ce *package* contient les différents générateurs de code traduisant les politiques de collecte de données en code exécutable.

— `deposit.runtime` : ce *package* contient la définition du proxy de composition permettant le déploiement de politiques de collecte de données sur une infrastructure partagée.

Ce logiciel et son code source sont disponibles sur GitHub à l'adresse https://github.com/ace-design/DEPOSIT

▶ ***Contexte logiciel***. Nous utilisons une machine virtuelle dédiée et équipée de 16 Go de mémoire vive pour conduire les différentes expérimentations. Chacune des expérimentations a été reproduite au minimum 5 fois afin de considérer les artefacts de mesures comme négligeables.

▶ ***Contexte matériel***. Dans le cadre de la validation, nous utilisons une infrastructure reprenant la topologie décrite en Fig. 7.1. Cette infrastructure est construire sur du matériel de prototypage (plateformes Arduino et Raspberry Pi) et nous permet ainsi d'avoir des plateformes évolutives grâce à l'ajout de *shields* [2]. Par exemple, l'ajout d'un shield Grove sur une plateforme Arduino nous permet d'utiliser des capteurs du constructeur Grove. De manière similaire, l'ajout d'un shield sans-fil nous permet de bénéficier d'une connectivité sans-fil sur une plateforme n'en contenant initialement pas. Ces plateformes nous ont permis de travailler dans un contexte où chaque bureau possède 6 capteurs : 3 de température et 3 de puissance électrique ; un capteur de température extérieur est également accessible. La figure Fig. 7.2 présente la plateforme `P_OUTSIDE` utilisée au sein de notre infrastructure. Cette plateforme est un

1. Data collEction POlicies for Sensing InfrasTructures
2. un ensemble de cartes électroniques se connectant sans soudure afin d'ajouter de nouvelles fonctionnalités

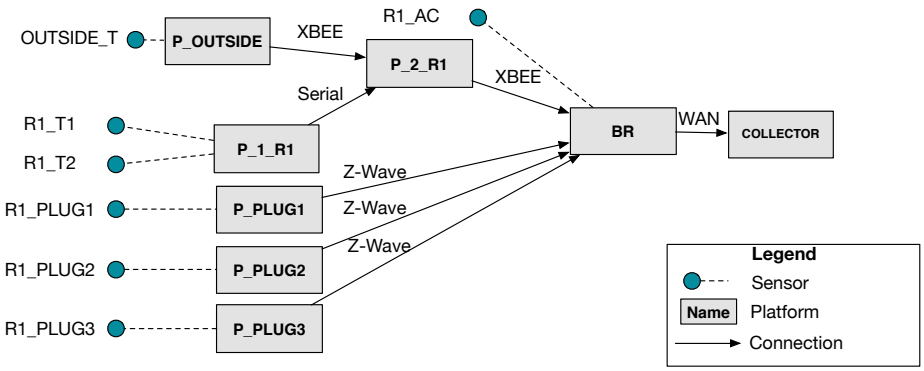


FIGURE 7.1 – Topologie de l'infrastructure de capteurs

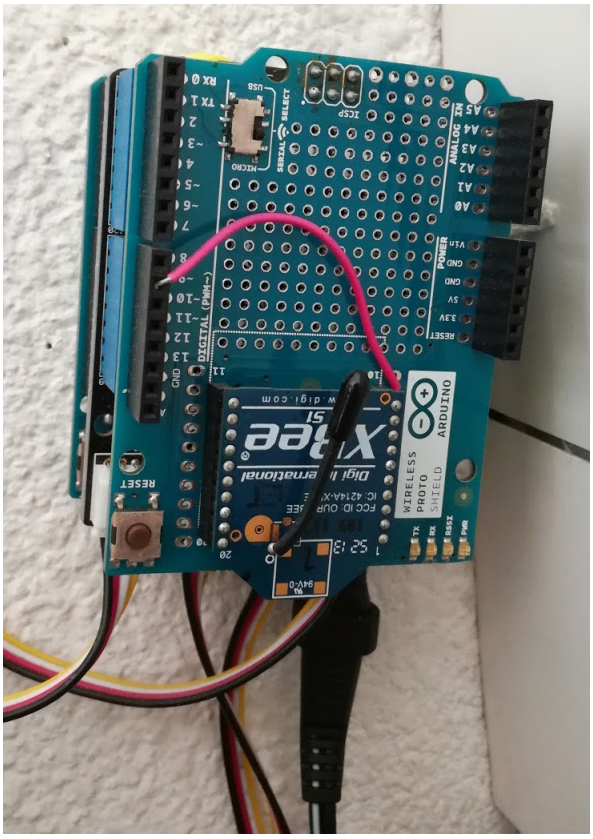


FIGURE 7.2 – Plateforme `P_OUTSIDE` équipée d'un shield Grove et d'un shield sans-fil

Arduino Uno, équipé d'un shield Grove (sur lequel est connecte le capteur de température `OUTSIDE_T`, non apparent sur l'illustration) et d'un shield sans-fil pour permettre une communication XBEE avec la plateforme `P_2_R1`. Nous étendons ensuite cette infrastructure par simulation pour cibler un bâtiment entier puis effectuer une évaluation à l'échelle de villes et bâtiments intelligents afin de raisonner sur plusieurs milliers de capteurs.

Ce chapitre est organisé de la manière suivante : Dans une première partie, nous utiliserons le scénario fil rouge afin d'atteindre les objectifs identifiés en CHAP. 3. Ensuite, nous procéderons à une évaluation des différentes contributions à plus grande échelle

en ciblant les villes et bâtiments intelligents. Enfin, nous discuterons des obstacles à la validité avant de conclure.

7.2 Déploiement d'une politique de collecte de données *« bâtiment intelligent »*

7.2.1 Objectifs de validation

À partir de critères d'acceptation (C_x), nous allons valider les différents objectifs (O_x) identifiés en CHAP. 3 et résumés ci-dessous.

(O_1) Un ingénieur logiciel ne manipule que des concepts métiers

Les travaux d'état de l'art, *cf.* SEC. 2.1, nous montrent que l'infrastructure de capteurs doit être configurée au niveau matériel en fonction des besoins des ingénieurs logiciels. Ainsi, ils doivent comprendre l'infrastructure sous-jacente et utiliser des langages de programmation de bas niveau. Si des abstractions existent, elles restent toutefois limitées à une même architecture de plateforme ou d'infrastructure, réduisant leur applicabilité à des infrastructures hétérogènes. Afin que les ingénieurs logiciels restent focalisés sur leur domaine d'expertise, ils doivent utiliser une approche à *haut-niveau d'abstraction* et *indépendante du matériel* pour exprimer des politiques de collecte de données. Une telle abstraction permet également à un ingénieur logiciel de réutiliser sa politique dans différentes infrastructures de capteurs, car son code n'est plus couplé à un réseau spécifique de capteurs. Afin de valider cet objectif, les critères (C_1) et (C_2) devront être satisfaits :

— (C_1) Une politique de collecte de données est construite en utilisant uniquement des concepts métiers.

— (C_2) Une politique de collecte de données est déployée sur une infrastructure cible sans la nécessité de connaissances préalables relatives au fonctionnement de cette infrastructure.

(O_2) Une politique est adaptée aux capacités matérielles

Le grand nombre de plateformes disponibles conduit à une grande diversité de logiciels et de matériels, ce qui rend chaque déploiement spécifique à une seule infrastructure. L'état de l'art offre déjà quelques abstractions, mais elles doivent être supportées par les différentes plateformes et restent dépendantes de l'infrastructure sous-jacente. Pour aider les ingénieurs logiciels dans l'exploitation d'une infrastructure de capteurs, une politique de collecte de données doit être adaptée automatiquement aux spécificités des plateformes. Afin de valider cet objectif, le critère (C_3) devra être satisfait :

— (C_3) Le code source généré utilise des bibliothèques logicielles adaptées aux plateformes de l'infrastructure de capteurs

(O_3) Une politique est projetée sur l'infrastructure

Par égard aux principes de séparation des préoccupations, une politique de collecte de données ne devraient pas être couplée à une infrastructure de capteurs. Les infrastructures de capteurs à grande échelle mettent à disposition une multitude de plateformes réparties dans des topologies réseaux complexes. Un ingénieur logiciel ne

devrait pas avoir à tenir compte de cette complexité : un mécanisme attribuant un ensemble de plateformes pour la mise en œuvre de la politique serait nécessaire. Afin de valider cet objectif, les critères (C_4) et (C_5) devront être satisfaits :

— (C_4) Chaque activité de la politique de collecte de donnée est projetée automatiquement vers une plateforme cible adaptée.

— (C_5) La procédure de génération de code produit du code prêt à être transféré sur les plateformes de l'infrastructure de capteurs.

(O_4) Une infrastructure est partagée entre différentes politiques

Lors du déploiement d'une nouvelle politique, plusieurs autres politiques peuvent déjà être en cours d'exécution sur l'infrastructure de capteurs, *cf.* (C_2). En raison de la séparation des préoccupations entre les points de vue du logiciel et du réseau, un ingénieur logiciel n'a pas à connaitre les différentes politiques déjà déployées. Un mécanisme de partage devrait donc gérer le partage de l'infrastructure entre plusieurs politiques de manière transparente et automatique pour l'ingénieur logiciel. Afin de valider cet objectif, les critères (C_6) et (C_7) devront être satisfaits :

— (C_6) Plusieurs politiques sont exécutables sur la même infrastructure (partage du réseau).

— (C_7) Une même plateforme supporte simultanément plusieurs politiques (partage de la plateforme).

7.2.2 Définition de la politique de collecte de données

Dans cette sous-section, nous allons procéder à la validation de (O_1) en montrant que la politique fil rouge ne s'exprime qu'en utilisant des concepts métiers propres à la définition de politiques de collecte de données. Nous rappelons qu'une politique de collecte de données est un « ensemble d'opérations effectuées sur des données brutes afin de les convertir en données à valeur ajoutée » [GBMP13]. Nous complétons ensuite cette politique afin de couvrir l'ensemble des bureaux du bâtiment intelligent. À partir du langage spécifique au domaine et des éléments du langage générique dans lequel ce dernier a été mis en oeuvre, l'ingénieur logiciel définit une politique *modèle* qui sera instanciée pour chacun des bureaux : par exemple `template("10")` construit la politique de collecte de données pour le bureau `R10`. Dans le code Lst. 7.1, les éléments variables de la politique, *par ex.*, L.5 - identifiant du capteur, seront renseignés lors de l'instanciation de la politique.

Listing 7.1 – Politique *template*

```
1
2  object ThermalShockPreventionTemplate extends DEPOSIT_TEMPLATE{
3
4    def template(args:Any*): Unit = {
5      this hasForName s"ThermalShockPrevention_${args.head}"
6      this handles classOf[SmartCampusType]
7
8      val r1_t1 = declare aPeriodicSensor() withPeriod 300 named s"R${args.head}_T1"
9      val r1_t2 = declare aPeriodicSensor() withPeriod 300 named s"R${args.head}_T2"
10     val outside_t = declare aPeriodicSensor() withPeriod 3600 named "OUTSIDE_T"
11     val r1_ac = declare aPeriodicSensor() withPeriod 60 named s"R${args.head}_AC"
12     val collector = declare aCollector() named "COLLECTOR"
13
14     val avg = define anAvg() withInputs("i1", "i2") andRenameData "AVG_TEMP"
15     val sub = define aSubstractor() withInputs("i1", "i2")
16     val abs = define anAbsoluteValue()
```

```
17      val condition = define aCondition "v␣>␣8"
18      val produce = define aProducer
19                    new AlertMessageType("ALERT_TEMP", s"R_${args.head}")
20                        withInputs("i1", "i2")
21      val process = define aProcess ACStatusPolicy()
22
23      flows {
24        r1_t1() -> avg("i1")
25        r1_t2() -> avg("i2")
26        avg() -> sub("i1")
27        outside_t() -> sub("i2")
28        sub () -> abs ()
29        abs() -> condition()
30        r1_ac () -> process ("AC")
31        condition("then") -> produce("i1")
32        process("COOL") -> produce("i2")
33        produce() -> collector()
34      }
35    }
36  }
```

Validation de (C_1)

Le code Lst. 7.1 illustre la définition de la politique de collecte *template* qui sera mise en œuvre pour chacun des bureaux composant le bâtiment intelligent. Grâce au langage spécifique au domaine (présenté en Sec. 4.2.4), l'ingénieur logiciel utilise les éléments du méta-modèle, *cf.* Fig. 4.4, pour définir la politique de collecte de données :

- L'activité `PERIODIC_SENSOR` (L.8-L.11) est utilisée pour déclarer les capteurs nourrissant la politique avec des valeurs environnementales.
- L'opération `AVERAGE` (*cf.* L.14) est utilisée pour calculer la température moyenne du bureau. La valeur de capteur résultante est nommée `AVG_TEMP`.
- Les opérations `SUBSTRACTOR` et `ABS` (L.15-L.16) sont utilisées pour calculer la différence en valeur absolue entre la température moyenne du bureau et la température extérieure.
- L'opération `CONDITION` (L.17) permet de vérifier si la différence précédemment calculée est supérieure au seuil de 8 degrés Celsius.
- L'opération `PRODUCE` (L.18) construit une valeur de capteur d'alerte lorsque ses ports d'entrée ont chacun une valeur de capteur synchronisée.
- L'activité `PROCESS` (L.21) permet la réutilisation d'une politique de collecte renseignant l'état de fonctionnement d'un bloc climatiseur réversible (états `COOL`, `HEAT` ou `OFF`).
- Les flux de données (L.23 – L.34) définissent les connexions entre les ports des différentes activités.

Plus particulièrement, le code mettant en œuvre la politique de collecte de données *template* ne fait intervenir aucun élément lié aux communications réseau ou à l'économie d'énergie. L'ingénieur logiciel n'ayant, de ce fait, pas besoin de manipuler des notions propres aux réseaux de capteurs, nous considérons le premier critère (C_1) satisfait.

Validation de (C_2)

Les préoccupations liées à l'infrastructure de capteurs sont regroupées au sein du modèle d'infrastructure présenté en annexe Ann.A. Dans le cadre de l'infrastructure du bâtiment intelligent, *cf.* Fig. 7.1, le modèle d'infrastructure contient :

- Un modèle topologique décrivant les connexions entre les différentes plateformes ;

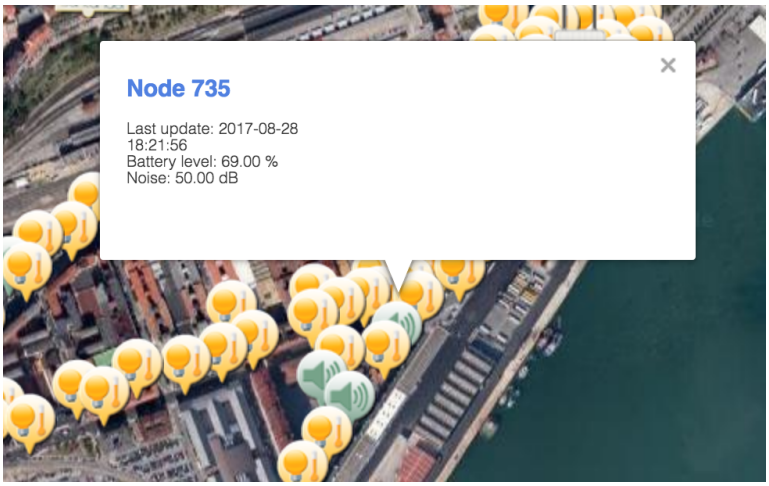


FIGURE 7.3 – Cartographie des capteurs déployés dans le projet SmartSantander (source : http://maps.smartsantander.eu/)

— Un modèle de fonctionnalités décrivant physiquement les plateformes ;

— Une stratégie de déploiement : *au plus près des capteurs* (*cf.* SEC. 5.3).

La procédure de pré-déploiement remplace l'activité `PROCESS` par sa politique embarquée puis associe, à chacune des activités, un ensemble de plateformes candidates. Ensuite, l'application de l'opérateur *deploy* identifie les plateformes cibles grâce à la stratégie de déploiement puis crée des sous-politiques de collecte de données pour chacune des plateformes impliquées dans la mise en œuvre de la politique, *cf.* TAB. 5.7. Enfin, pour chacune des sous-politiques de collecte de données, un générateur de code produit du code adapté pour la plateforme cible, *cf.* SEC. 5.5.

La complexité du déploiement est gérée grâce à une procédure de pré-déploiement et un opérateur de déploiement pilotés par un modèle d'infrastructure fourni par l'expert réseau. Cette séparation des préoccupations satisfait le critère (C_2).

(C_1) et (C_2) étant satisfaits, nous considérons l'objectif (O_1) atteint.

Limites

(C_1) **:** La validation de (C_1) a été effectuée en utilisant les activités du méta-modèle présentées en SEC. 4.2.3. Les actions proposées au sein de ce méta-modèle sont inspirées d'autres travaux présentant des langages de manipulation de données. Nous pensons cet ensemble est suffisant pour l'expression des besoins métier d'un ingénieur logiciel. En revanche, nous n'avons pas conduit de tests utilisateurs pour le valider. Il est dans nos perspectives (transfert industriel, *cf.* SEC. 8.2.1) d'évaluer l'utilisation des activités auprès d'experts métier dans le domaine de l'Industrie 4.0 et des grilles énergétiques.

(C_2) **:** Si le déploiement d'une politique ne nécessite pas de connaissances sous-jacentes vis-à-vis de l'infrastructure de capteurs, la définition de la politique nécessite de connaitre l'identifiant des capteurs utilisés, nuançant la validation de (C_2). Cependant, il est d'usage de mettre à disposition la liste des capteurs disponibles au sein d'une infrastructure de capteurs (*par ex.*, FIG. 7.3). Nous pouvons ainsi supposer que l'ingénieur logiciel connait les capteurs nécessaires pour la réalisation de sa politique de collecte de données.

7.2.3 Déploiement de la politique de collecte de données

Dans cette sous-section, nous allons procéder à la validation de (O_2) en déployant la politique fil rouge sur une infrastructure de capteurs vierges (la validation du partage aura lieu en Sec. 7.2.4). La topologie de cette infrastructure est rappelée en Fig. 7.1.

Validation de (C_3)

Les lignes de produits logiciels permettent de créer des logiciels à partir d'un ensemble d'éléments partagés dans une famille. Ces lignes de produits sont bâties autour d'un modèle de variabilité gérant la sélection cohérente des éléments à intégrer dans le logiciel. Nous avons réutilisé ces modèles de variabilité afin de modéliser l'hétérogénéité de l'infrastructure de capteurs. L'expert réseau peut ainsi décrire, au sein du modèle d'infrastructure, la configuration complexe de chacune des plateformes. Ce modèle de variabilité est ensuite utilisé pour identifier les plateformes candidates au déploiement d'activité (grâce à la vérification de contraintes de déploiement *cf.* Sec. 5.4.2) et lors de la génération de code pour produire du code adapté à la plateforme cible (*cf.* Sec. 5.5). Cette production de code adaptée à la variabilité des plateformes est possible grâce à l'utilisation d'un ensemble de bibliothèques de décrivant la mise en œuvre de chacune des fonctionnalités possibles pour une plateforme.

Listing 7.2 – Gestion des bibliothèques de code au sein du générateur de code `Wiring`

```
override val sensorTypeHandling: HashMap[SensorType.Value, (String, String, String)] =
HashMap(
    SensorType.Temperature -> ("acquire()",
                               generateConcreteDataTypeName(classOf[DoubleType]),
                               "TemperatureSensor")
)

override val sensorBrandHandling: HashMap[SensorBrand.Value, (String, String)] =
HashMap(
    SensorBrand.GroveTemperature -> ("temperature_grove.h", "GroveTemperatureSensor"),
    SensorBrand.EBTemperature -> ("temperature_eb.h", "EBTemperatureSensor"),
    SensorBrand.DFTemperature -> ("temperature_df.h", "DFTemperatureSensor")
)
```

La plateforme `P_OUTSIDE` étant une plateforme de type Arduino, le modèle de variabilité spécifie que le générateur `Wiring` doit être utilisé pour produire le code décrivant la mise en œuvre de la politique fil-rouge (*cf.* Fig. 5.3 page 77). Ce générateur de code connait les bibliothèques logicielles à utiliser en fonction des caractéristiques matérielles de la plateforme. Par exemple, le listing présenté en Lst. 7.2 illustre le choix de la bibliothèque en fonction du type et du constructeur de capteurs. Dans le cas présent, la plateforme est équipée d'un capteur de température de type `Grove`. Le générateur utilisera ainsi la bibliothèque logicielle `temperature_grove.h` (abstrayant les problématiques de conversion de la valeur mesurée en température) et la méthode `acquire()` pour récupérer les valeurs depuis le capteur. L'utilisation de la configuration complexe des plateformes associée aux générateurs de code permet ainsi de satisfaire le critère (C_3), et donc de remplir l'objectif (O_2).

Validation de (C_4)

La stratégie de déploiement définie par l'expert réseau Sec. 5.3 a pour objectif d'associer une plateforme cible à chacune des activités composant la politique de collecte de données et de construire les sous-politiques. Les générateurs de code ont pour finalité de produire le code source prêt à être transféré sur les plateformes composant l'infrastructure de capteurs.

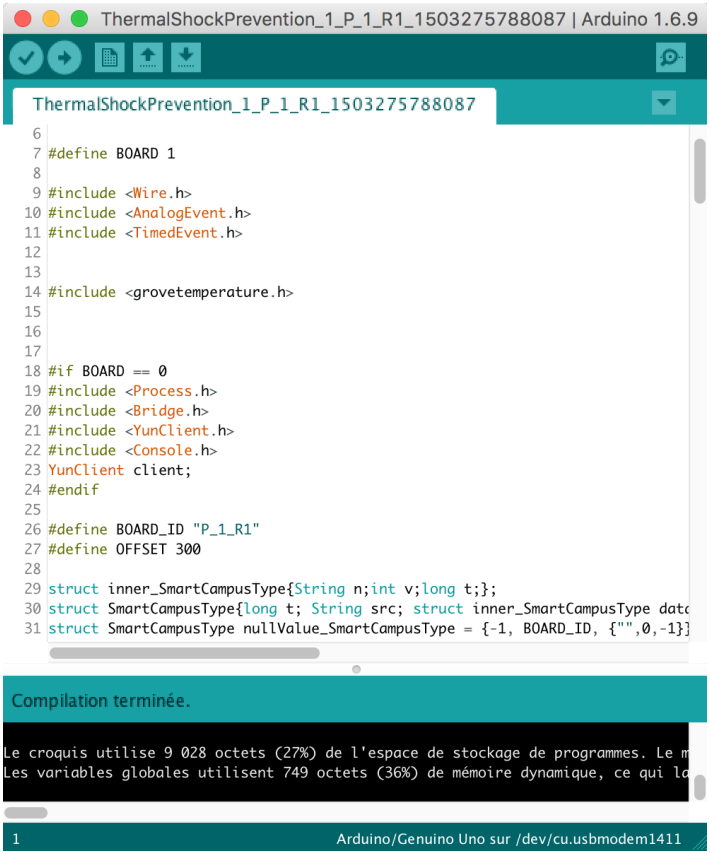


FIGURE 7.4 – Résultat de la compilation du code généré pour la plateforme `P_1_R1`

A petite échelle, nous avons déployé les fichiers sources générés sur de véritables plateformes afin de nous assurer de leur validité fonctionnelle. Comme montré dans le tableau TAB. 7.4, le déploiement d'une politique à l'échelle d'un bureau a produit des fichiers sources pour les 4 plateformes impliquées dans la réalisation de la politique de collecte de données (`P_OUTSIDE`, `P_1_R1`, `P_2_R1` et `BR`), satisfaisant le critère (C_4) dans le cadre de nos expérimentations.

Validation de (C_5)

La validation de ce critère se vérifie en obtenant un code compilable et prêt à être transféré sur une plateforme de l'infrastructure de capteurs. Nous présentons en FIG. 7.4 une capture d'écran illustrant le résultat de la compilation du code Wiring pour la plateforme `P_1_R1`. Le code obtenu étant prêt à être transféré, (C_5) est satisfait dans le cadre de nos expérimentations.

(C_4) et (C_5) étant satisfaits, nous considérons l'objectif (O_3) atteint.

Limites

(C_3) **:** La sélection des bonnes bibliothèques logicielles grâce aux modèles de variabilité permet de produire du code adapté à la plateforme, validant (C_3). Afin de couvrir l'ensemble des caractéristiques des plateformes, les bibliothèques doivent être exhaustives. Dans le cadre de cette expérimentation, nous avons uniquement développé des bibliothèques pour nos plateformes cibles. L'effort de développement pour ces bibliothèques logicielles est non-négligeable puisque nous devons fournir une mise en œuvre,

par générateur de code, de chaque fonctionnalité décrite dans les modèles de variabilité. Toutefois, cet effort sera amorti par la possibilité offerte à un ingénieur logiciel d'exprimer ses intentions métier indépendamment du matériel et la possibilité pour un expert réseau de réutiliser ces bibliothèques entre des infrastructures de capteurs différentes mais ayant des plateformes de capteurs partageant les mêmes fonctionnalités.

(C_4) **:** La projection des activités sur une plateforme est effectuée en appliquant l'opérateur *place* (spécifique à la stratégie de déploiement choisie) sur un ensemble de plateformes candidates. La validation de (C_4) a été effectuée en utilisant une stratégie de déploiement au plus près des capteurs (stratégie statique) et en vérifiant que les activités étaient correctement réparties sur les plateformes les plus près des capteurs. Cette stratégie est pertinente pour maximiser l'usage des plateformes de capteurs, mais ce type de déploiement peut présenter un risque pour le réseau de capteur si ces plateformes sont alimentées par batterie. Pour remédier à ce problème, une stratégie de déploiement statique prenant en compte le type d'alimentation (*par ex.*, privilégier des plateformes alimentées par courant secteur) ou dynamique (*par ex.*, déployer sur les plateformes ayant le plus haut niveau d'autonomie) serait plus adaptée. Dans l'éventualité où la stratégie ne peut trouver de plateforme candidate, la projection n'a pas lieu et un message d'erreur est renvoyé à l'ingénieur logiciel, ce qui nuance la validation de (C_4). Toutefois, nous faisons le choix de ne pas aller à l'encontre de la stratégie de déploiement, *par ex.*, en sélectionnant aléatoirement une plateforme parmi un ensemble de plateformes candidates, afin de ne pas nuire aux optimisations voulues par l'expert réseau.

(C_5) **:** La validation de (C_5) a été effectuée en transférant le code produit par le générateur `Wiring` sur la plateforme `P_1_R1`. Nous avons notamment vérifié que les opérations décrites à travers les activités étaient correctement traduites et que les valeurs de capteurs étaient bien transmises sur le réseau à travers les points d'extensions résultant de la décomposition de la politique globale en sous-politiques de collecte de données. Nous n'avons toutefois pas effectué de tests sur un éventuel désynchronisme des données de capteurs (*par ex.*, comme illustré sur le tableau Tab. 4.1 page 56) aux ports d'entrées de l'activité `AVG` (faisant la moyenne de la température entre `R1_T1` et `R1_T2`). En effet, nous considérons comme négligeable le risque de désynchonisme aux ports de ces activités, car les données de capteurs consommées sont issues de la même plateforme et donc, utilisent la même base de temps. De plus, nous considérons comme négligeable le risque de panne des capteurs.

7.2.4 Déploiement sur une infrastructure partagée

Nous avons présenté un proxy de composition (*cf.* Sec. 6.3.2) permettant de partager une infrastructure entre plusieurs plateformes. Nous plaçons ce proxy entre les ingénieurs logiciels et l'infrastructure de capteurs représentée en Fig. 7.1. Le proxy possède une représentation du modèle d'infrastructure décrit en annexe Ann.A. Afin de valider les critères (C_6) et (C_7), nous allons utiliser les politiques A, B et C pour vérifier les opérations de composition et de déploiement sur l'infrastructure de capteurs :

— **Politique A - Prévention de chocs thermiques *(fil rouge)***. Cette politique permet d'envoyer une alerte lorsque la différence entre la température moyenne d'un bureau climatisé et la température extérieure est supérieure à $8°C$. La présentation de cette politique est effectuée en Sec. 1.3

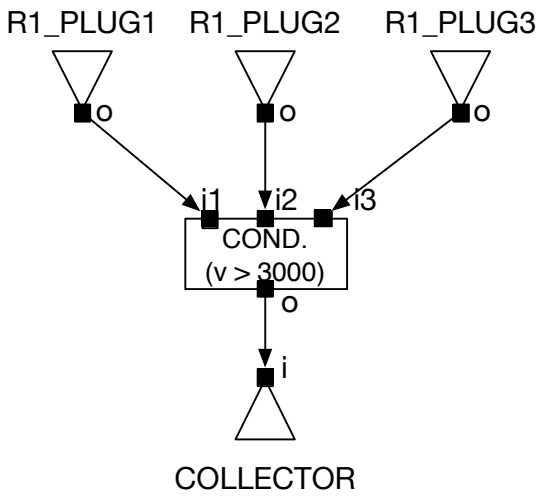


FIGURE 7.5 – Politique C - Surconsommation électrique (reproduction de l'illustration générée par DEPOSIT pour des raisons de mise en page)

TABLEAU 7.1 – Périodes d'échantillonnage pour les capteurs impliqués dans $\pi_{\text{thermalshock}}$ et $\pi_{\text{energylosses}}$ (en gras : périodes identiques)

	R1_T1	R1_T2	R1_AC	OUTSIDE_T
$\pi_{\text{thermalshock}}$	300	300	**60**	**3600**
$\pi_{\text{energylosses}}$	60	60	**60**	**3600**

— **Politique B - Déperdition énergétique**. Cette politique permet de renvoyer une alerte lorsque la température extérieure est inférieure à la température moyenne d'un bureau climatisé. La présentation de cette politique est effectuée en SEC. 6.2.4

— **Politique C - Surconsommation électrique** (FIG. 7.5). Un bureau est équipé de prises connectées sur lesquelles sont branchés des appareils à consommation électrique plus ou moins importante, *par ex.*, un écran d'ordinateur ou une cafetière. Dans le cadre d'une politique de réduction de la charge énergétique, les bureaux ne doivent pas avoir une consommation instantanée supérieure à 3000W.

Description des déploiements

▶ ***Déploiement de A***. Le déploiement de la politique A ne déclenche aucune composition, car l'infrastructure de capteurs est initialement vide. Le rôle du proxy étant d'assurer le déploiement automatique des politiques entrantes sur l'infrastructure, la méthode `registerPolicy` du proxy procède au déploiement de A en procédant à son adaptation aux caractéristiques des plateformes de l'infrastructure (*cf.* EQ. 5.5) et à la décomposition en sous-politiques de collecte de données grâce à l'opérateur *deploy* (*cf.* EQ. 5.6). Du déploiement résultent 4 sous-politiques de collecte de données à destination des plateformes `BR`, `P_2_R1`, `P_1_R1` et `P_OUTSIDE`. Les générateurs de codes (identifiés grâce à la modélisation de la variabilité des contrôleurs, *cf.* LST. A.1 – L.3) produisent ensuite du code prêt à être déployé sur l'infrastructure.

▶ ***Déploiement de B***. L'infrastructure exécute déjà la politique A. Le déploiement de B à travers la méthode `registerPolicy` du proxy entrainera sa composition

avec la politique courante (ici A, *cf.* FIG. 7.6). Les capteurs identiques (*cf.* DEF.6.2.1) ont été fusionnés. Comme expliqué à travers le déploiement de A, le proxy procède ensuite au déploiement de $A \oplus B$ en régénérant des sous-politiques spécifiques aux plateformes et en produisant du code prêt à être déployé sur l'infrastructure. Du déploiement de $A \oplus B$ résultent 4 sous-politiques de collecte de données à destination des plateformes `BR`, `P_2_R1`, `P_1_R1` et `P_OUTSIDE`.

▶ ***Déploiement de C***. L'infrastructure exécute déjà la politique $A \oplus B$ résultante de la composition de A avec B. Le déploiement de C à travers la méthode `registerPolicy` du proxy entrainera sa composition avec la politique courante (ici $A \oplus B$, *cf.* FIG. 7.6). Du déploiement de $A \oplus B \oplus C$ résultent 7 sous-politiques de collecte de données à destination des plateformes `BR`, `P_2_R1`, `P_1_R1`, `P_OUTSIDE`, `P_PLUG1`, `P_PLUG2`, `P_PLUG3`.

Afin de vérifier la bonne composition automatique des politiques de collecte de données, nous présentons en TAB. 7.2 la taille (nombre d'activités et de flux de données) des politiques A, B et C prises indépendamment les unes des autres et en TAB. 7.3, la taille des politiques globales et sous-politiques résultant des compositions automatiques. A et B présentent deux capteurs identiques qui seront fusionnés, d'où un nombre d'activités égal à $11 + 10 - 2 = 19$. Dans le cas de la composition $A \oplus B \oplus C$, le nombre d'activités est bien de, $19 + 6 = 25$ car aucun capteur n'est fusionné (C n'apporte pas de capteurs identiques).

La politique composée $A \oplus B \oplus C$ (FIG. 7.6) est ensuite déployée sur l'infrastructure de capteurs présentée en FIG. 7.1. Il en résulte 7 sous-politiques à destination des 7 plateformes composant le réseau de capteur (pour rappel, la plateforme `COLLECTOR` bien qu'appartenant à l'infrastructure de capteurs est hors du réseau de capteurs, car il s'agit du serveur collectant les données issues du réseau de capteur).

Validation de (C_6)

Grâce à l'opérateur de composition $\oplus$, la politique obtenue est une politique de collecte de données composée des intentions métier exprimés initialement dans des politiques différentes. Elle peut ainsi être déployée comme n'importe quelle autre politique et est indépendante du matériel. Le déploiement de la politique composée permet donc d'exécuter plusieurs politiques de collecte de données sur la même infrastructure de capteurs, satisfaisant (C_6) pour notre déploiement.

Validation de (C_7)

L'opérateur *deploy* décompose la politique de collecte de données en sous-politiques de collecte de données spécifiques aux plateformes de l'infrastructure de capteurs. La décomposition d'une politique issue de l'opérateur $\oplus$ engendre des sous-politiques spécifiques aux plateformes contenant des activités provenant initialement de politiques de collecte de données différentes. Ainsi, une même plateforme peut supporter simultanément plusieurs politiques, satisfaisant (C_7).

(C_6) et (C_7) étant satisfaits, nous considérons l'objectif (O_4) atteint.

Limites

▶ ***Limites de*** (C_6). Le déploiement de plusieurs politiques sur une même infrastructure à travers le proxy de composition nous a permis de valider (C_6). En revanche, nous n'avons pas mesuré l'impact du déploiement d'une politique « gourmande » en

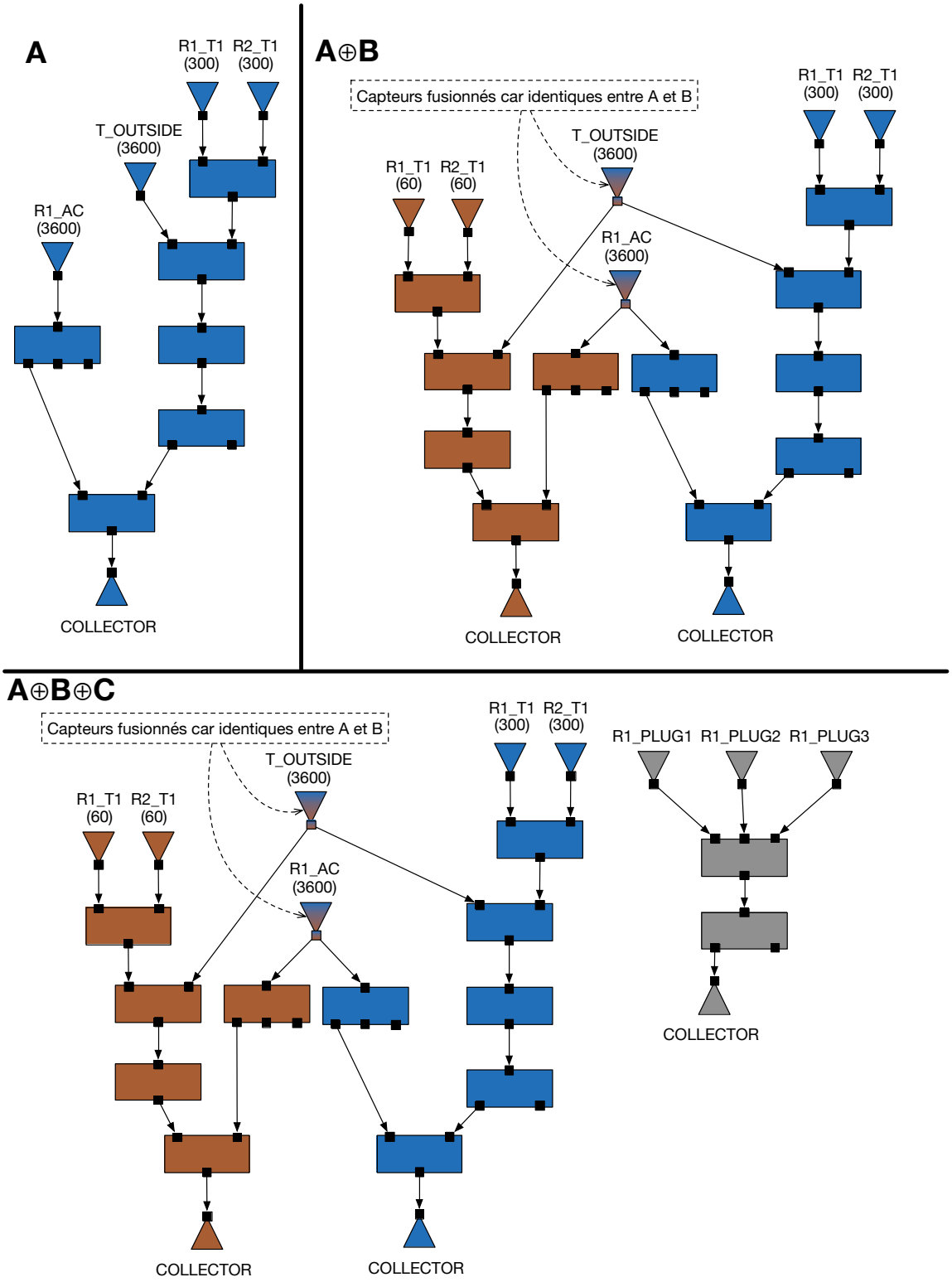


FIGURE 7.6 – Composition automatique des politiques consécutive au déploiement successif de A, B et C (reproduction de l'illustration générée par DEPOSIT)

TABLEAU 7.2 – Taille des politiques de collecte de données individuelles

	Politique A	**Politique B**	**Politique C**
# Activités	11	10	6
# Flux	10	9	5

TABLEAU 7.3 – Politique globale et sous-politiques de collecte de données produites par le proxy de composition

Déploiement de :	**A**	**B**	**C**
Politique globale			
# Activités	11	19	25
# Flux	10	19	24
Sous-politiques BR			
# Activités	5	9	15
# Flux	4	8	13
P_2_R1			
# Activités	6	11	11
# Flux	5	9	9
P_1_R1			
# Activités	4	8	8
# Flux	3	6	6
P_OUTSIDE			
# Activités	2	2	2
# Flux	1	1	1
P_PLUG1			
# Activités	-	-	2
# Flux	-	-	1
P_PLUG2			
# Activités	-	-	2
# Flux	-	-	1
P_PLUG3			
# Activités	-	-	2
# Flux	-	-	1

ressources sur l'infrastructure, *par ex.*, une politique nécessitant des valeurs de capteurs toutes les secondes. Suite à la composition de cette politique « gourmande » avec les autres politiques précédemment déployées, nous identifions un risque de monopolisation des ressources des plateformes à unique flux d'exécution, au détriment des autres actions. Nous pouvons toutefois mitiger ce risque en effectuant une simulation du comportement du code obtenu avant son déploiement sur la plateforme cible, *par ex.*, Cooja [ODE+06] et Autodesk Circuits [3] permettent respectivement de simuler une plateforme Contiki et Arduino.

► ***Limites de*** (C_7). Le déploiement d'activités provenant de politiques différentes sur une même plateforme nous a permis de valider (C_7). La consommation de la mémoire flash d'une plateforme est directement liée au nombre d'activités déployées et possède une limite propre au contrôleur, *par ex.*, 32 Ko de stockage flash pour un contrôleur ATMega328P. Nous identifions un risque que le déploiement de nombreuses politiques produise un code non transférable sur la plateforme car trop volumineux.

3. https://circuits.io/

Nous pouvons mitiger ce risque à l'aide d'une stratégie de déploiement dynamique connaissant, pour chaque plateforme, l'empreinte mémoire en stockage de la politique exécutée et la taille de stockage maximale. Ainsi, lors de la sélection de la plateforme cible, cette stratégie sélectionnera, parmi un ensemble de plateformes candidates, la plateforme ayant le plus grand espace mémoire disponible.

7.3 Évaluation à l'échelle de villes et larges bâtiments intelligents

Les villes et bâtiments intelligents représentent une classe d'applications reposant sur des infrastructures de capteurs déployées à grande échelle. Par exemple, la ville de Santander a déployé une infrastructure basée sur 12000 capteurs et la ville de Moscou a pour objectif de couvrir ses 70000 places de stationnement avec des capteurs (en 2017, il s'agissait du déploiement le plus important).

7.3.1 Objectifs de l'évaluation

Cette évaluation a pour objectif de mesurer l'applicabilité des politiques de collecte de données pour décrire des besoins métier utilisant de nombreux capteurs et traitements de données. En particulier, nous souhaitons identifier si le nombre de lignes de code mettant en œuvre une politique augmente en fonction de la taille des infrastructures. Ensuite, nous mesurons la capacité des politiques à être déployées sur des infrastructures de capteurs à l'échelle de villes et bâtiments intelligents. En particulier, nous mesurerons le temps de déploiement, *i.e.*, le temps nécessaire à l'identification des plateformes cibles et à la création de sous-politiques de collecte de données, de politiques sur des infrastructures non-partagées et partagées. À titre de référence, nous considérons 15 minutes comme étant le temps nécessaire à un expert réseau pour écrire et transférer sur les différentes plateformes le code mettant en œuvre la politique fil rouge à l'échelle d'un bureau sans utiliser les contributions présentées.

Générateurs d'infrastructures

Afin de raisonner sur des infrastructures à l'échelle de villes et bâtiments intelligents, nous avons construit au sein de DEPOSIT, deux générateurs d'infrastructures :

▶ ***Générateur 1***. Ce générateur d'infrastructures permet d'obtenir des modèles d'infrastructures reprenant la topologie réseau du réseau de capteur de la ville de Santander et permettant les applications de stationnement intelligent (*cf.* Fig. 7.7). Il prend en paramètre un triplet (x, y, z) où x correspond au nombre de plateformes de capteurs contenant chacune un capteur magnétique, où y correspond au nombre de plateformes relais envoyant les données de capteurs à un routeur frontal et où z correspond au nombre de plateformes de capteurs connectées par plateforme relais.

▶ ***Générateur 2***. Ce générateur d'infrastructures permet d'étendre la topologie réseau présentée en Fig. 7.1 à plusieurs bureaux. Il prend en paramètre un triplet (x, y, z) où x correspond au nombre de bureaux (contenant chacun 6 capteurs répartis sur 5 plateformes de capteurs), y le nombre de ponts par bureau (un pont est une plateforme faisant le relais entre un bureau et le reste de l'infrastructure) et z le nombre de routeurs frontaux. La structure de cette topologie est illustrée en Fig. 7.8.

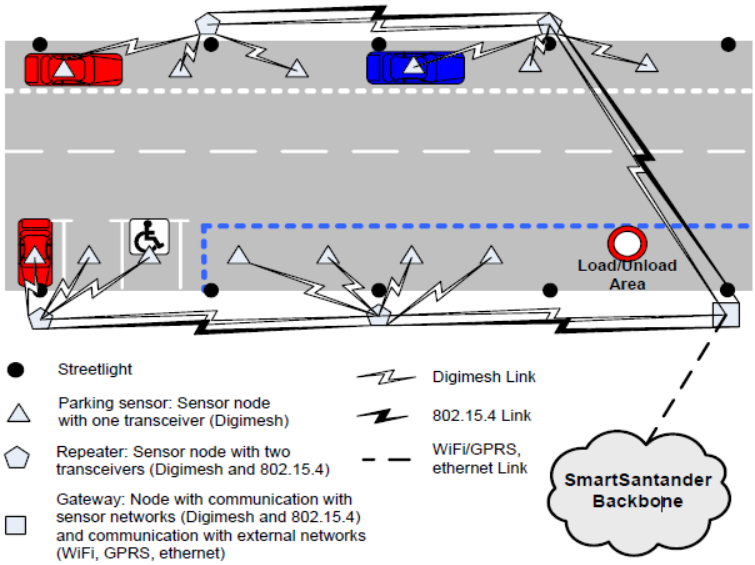


FIGURE 7.7 – Description de l'infrastructure de capteurs déployée pour le projet « parking intelligent » du projet SmartSantander (source : http://www.smartsantander.eu/index.php/testbeds/item/132-santander-summary)

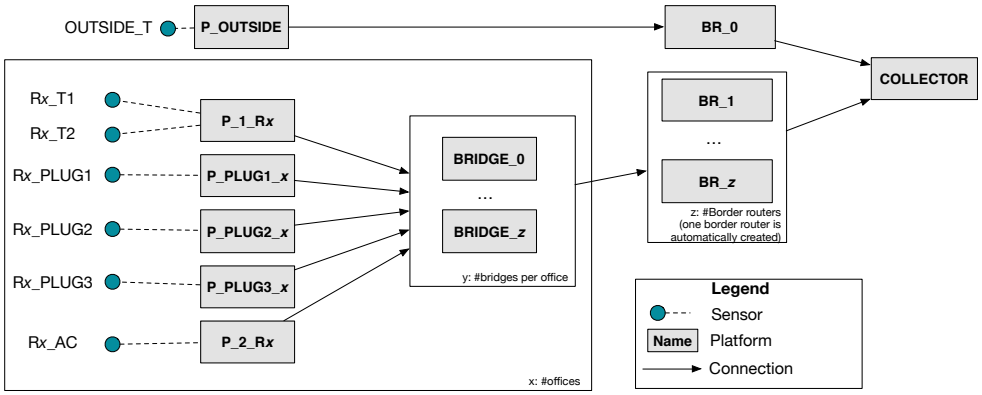


FIGURE 7.8 – Structure d'une topologie réseau générée de bâtiment intelligent

7.3.2 Définition de larges politiques de collecte de données

En utilisant une infrastructure à l'échelle de villes et bâtiments intelligents, un ingénieur logiciel pourra définir des politiques de collecte de données qui exploitent un grand nombre de capteurs et contiennent un nombre important d'activités. Pour mesurer la capacité des politiques de collecte de données à décrire des traitements à partir de milliers de capteurs, nous introduisons la métrique suivante :

Taille des politiques sources : Les infrastructures de capteurs à l'échelle de villes et bâtiments intelligents impliquent un grand nombre de capteurs. Nous devons nous assurer que la définition d'une politique de collecte à cette échelle reste acceptable en nombre de lignes sources dans le langage métier.

Protocole

Nous reprenons les besoins métier du fil rouge et nous les décrivons au sein *(i)* d'une unique politique de collecte de données couvrant l'ensemble des bureaux d'un bâtiment intelligent et *(ii)* d'une politique *template* (*cf.* Lst. 7.1) définie à l'échelle d'un bureau et dont l'ensemble de ses mises en œuvre pour chacun des bureaux a été composée. Nous pouvons ainsi comparer le gain en lignes de code apporté par l'utilisation de la politique *template* et de la composition par rapport à une unique politique.

Observations

Le tableau Tab. 7.4 présente les métriques relatives au déploiement de la politique de collecte de données fil rouge à l'échelle d'un bâtiment intelligent. À titre de référence, la mise en œuvre du scénario fil rouge nécessite l'utilisation de 11 activités et 25 lignes de code.

Nous avons observé la variation de ces métriques en augmentant le nombre de bureaux à 50 et en définissant deux politiques : *(i)* une politique contenant l'ensemble des activités et flux de données pour les 50 bureaux et *(ii)* une politique résultant de la composition de 50 mises en œuvre de la politique *template* (une mise en œuvre par bureau).

Dans le premier cas, la politique a nécessité l'écriture de 1054 lignes de codes afin d'exprimer 511 activités et flux de données associés. Dans le second cas, la politique a nécessité l'écriture d'une politique *template* (25 lignes de code, *cf.* Lst. 7.1) et une instruction de composition (1 ligne de code, *cf.* Lst. 7.3).

Listing 7.3 – Mise en œuvre de la politique *template* à l'échelle de 50 bureaux

```
val policy = (1 to 50).foldLeft(new Policy())
    {(acc,id) => acc + ThermalShockPreventionTemplate(id)}
```

Nous en concluons que l'utilisation d'une politique *template* et de l'opérateur de composition durant la phase l'expression permet de réduire le nombre de lignes de code source, et donc cet aspect de la complexité du développement. Nous avons vérifié cette conclusion en ciblant ensuite 100 et 500 bureaux et en observant que le nombre de lignes de code source restait constant.

Limites

Le faible nombre de lignes de code a été rendu possible par la réutilisation d'une politique *template* à l'échelle de plusieurs bureaux, car les traitements sur les données de capteurs sont strictement identiques entre les différents bureaux. Dans un scénario où un ingénieur logiciel aurait besoin d'adapter ce traitement pour chacun des bureaux, il devrait écrire une politique de données entière. Cette validation a également été effectuée sur un faible nombre de scénarios que nous trouvons toutefois représentatif des problématiques énergétiques au sein des bâtiments intelligents. Dans nos perspectives, nous prévoyons de définir des politiques de collecte de données à l'échelle d'industries (*Industrie 4.0*) ou de grilles énergétiques (*Smart grids*).

TABLEAU 7.4 – Évaluation du déploiement de la politique fil rouge

	DEPOSIT source (en LoC)	**# Fichiers générés**	**# Activités**	**Durée du déploiement (en s)**
Modèle	25	N/A	N/A	N/A
Un bureau	25	4	11	0,2
50 bureaux (sans composition)	1054	104	511	0,8
50 bureaux (avec composition)	25 + 1	104	511	0,8
100 bureaux (avec composition)	25 + 1	204	1011	2,5
500 bureaux (avec composition)	25 + 1	1004	5011	50

7.3.3 Déploiement sur l'infrastructure de capteurs

Passage à l'échelle du modèle d'infrastructure

Lors de l'utilisation d'infrastructures à l'échelle de villes et bâtiments intelligents, le nombre de capteurs et de plateformes disponibles font du développement une activité complexe. En prenant en considération le nombre de capteurs pouvant être impliqués dans la mise en œuvre de politiques de collecte de données, le modèle d'infrastructure doit passer à l'échelle. C'est dans cet objectif que nous évaluons la métrique suivante :

Temps de chargement en mémoire du modèle d'infrastructure : Le modèle d'infrastructure est la base du déploiement des politiques de collecte de données. Nous devons nous assurer que le chargement en mémoire de modèles à l'échelle de villes et bâtiments intelligents s'effectue en temps linéaire et reste acceptable pour un utilisateur.

▶ ***Protocole***. Grâce au générateur 1, nous avons fait varier le nombre de plateformes de capteurs x (par tranche de 1000) et avons défini la relation $y = x * 10\%$ afin d'obtenir une plateforme relais par tranche de 100 plateformes de capteurs. Chaque plateforme relais est ainsi connectée par 100 plateformes de capteurs. Pour chacune des infrastructures générées, nous avons créé un modèle d'infrastructure et mesuré son temps de chargement.

▶ ***Observation***. La figure FIG. 7.9 présente le temps de chargement du modèle d'infrastructure en fonction du nombre de capteurs. Nous pouvons observer que le temps de chargement du modèle d'infrastructure est une fonction affine par rapport au nombre de capteurs contenus dans ce modèle. Nous pouvons aussi observer que le temps de chargement d'un modèle décrivant une ville comme Santander s'effectue en moins de 4 secondes sur un ordinateur équipé de 16 Go de mémoire vive. Le modèle décrivant le stationnement intelligent de Moscou sera quant à lui chargé en approximativement 25 secondes. Cette première expérimentation nous a permis de montrer que de très grandes infrastructures peuvent être manipulées par les opérateurs nécessitant une représentation du modèle d'infrastructure.

▶ ***Limites***. Cette expérimentation a permis de vérifier le temps de chargement en mémoire du modèle d'infrastructure. Nous n'avons pas mesuré l'impact du nombre de capteurs sur les opérateurs reposant sur ce modèle. Nous effectuerons cette évaluation au sein des expérimentations de déploiement sur une infrastructure partagée et non partagée, *cf.* ci-après.

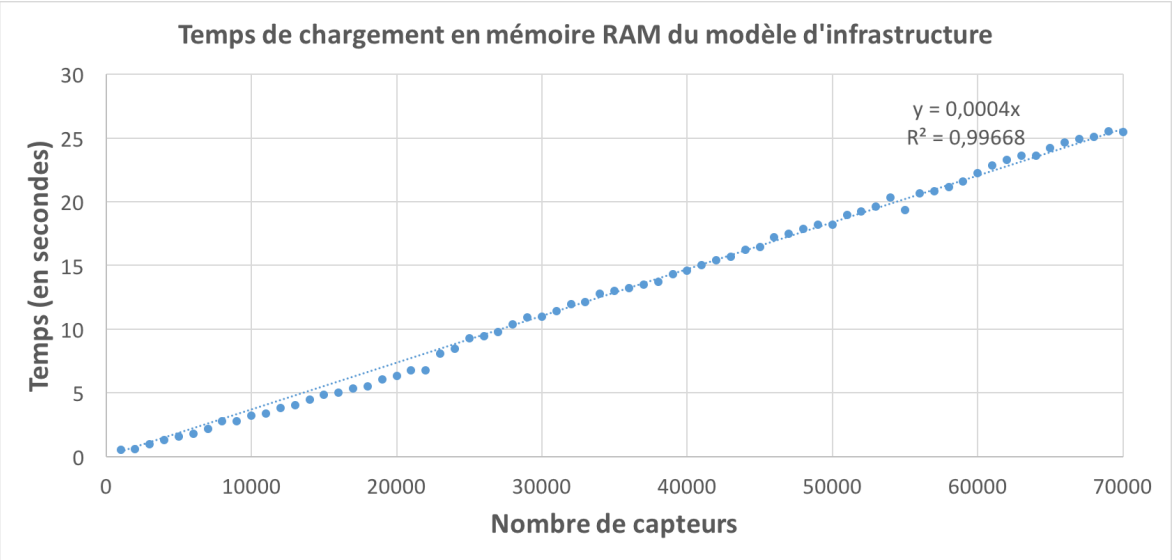


FIGURE 7.9 – Temps de chargement en mémoire RAM d'un modèle d'infrastructure en fonction du nombre de capteurs

Déploiement sur une infrastructure non partagée

Cette évaluation a pour objectif de vérifier que l'utilisation, soit d'une politique *template* et du mécanisme de composition, soit d'une politique entière, n'entraînent pas d'effets de bord sur le temps de déploiement. Nous introduisons la métrique suivante :

Temps de déploiement : Le temps nécessaire à l'identification des plateformes cibles et à la création des sous politiques de collecte de données.

Ensuite, nous utilisons le générateur 2 afin d'augmenter le nombre de capteurs et le nombre de plateformes de capteurs et observer la variation du temps de déploiement en fonction du nombre de plateformes. En particulier, nous allons observer le temps nécessaire à l'opérateur *place* pour identifier les plateformes cibles.

► ***Protocole***. En SEC. 7.3.2, nous avons défini des politiques de collecte de données à l'échelle de 50 bureaux de deux manières différentes, par écriture de l'ensemble de la politique par utilisation et composition d'une politique modèle. Dans un premier temps, nous déployons ces deux politiques afin d'observer si la manière dont elles sont exprimées a un effet sur le temps de déploiement. Dans un second temps, nous avons déployé les politiques A, B et C (*cf.* SEC. 7.2.4) sur différentes infrastructures en faisant varier le nombre de capteurs (et donc de plateformes de capteurs) afin d'observer l'impact sur l'opérateur *place* de la stratégie de placement « au plus près des capteurs ».

► ***Observations***. Le tableau TAB. 7.4 (page 126) présente les temps de déploiement pour les deux manières d'exprimer les politiques à l'échelle de 50 bureaux. Nous observons que leur temps respectif de déploiement est identique (0.8 seconde). Nous en concluons que la manière d'exprimer avec ou sans composition n'impacte pas le temps de déploiement. Nous avons également mesuré le temps de déploiement des politiques à l'échelle de 100 et 500 bureaux, et observé que ceux-ci s'effectuaient en moins d'une minute. La figure FIG. 7.10 présente le temps de déploiement pour les politiques A, B et C en fonction du nombre de capteurs présents dans l'infrastructure. Nous remarquons que le temps de déploiement de A et B augmentent plus rapidement que celui requis par la politique C. Nous pouvons lier cela au fait que A et B ont besoin d'agréger des données en provenance d'un nombre supérieur de capteurs, et par conséquent, l'iden-

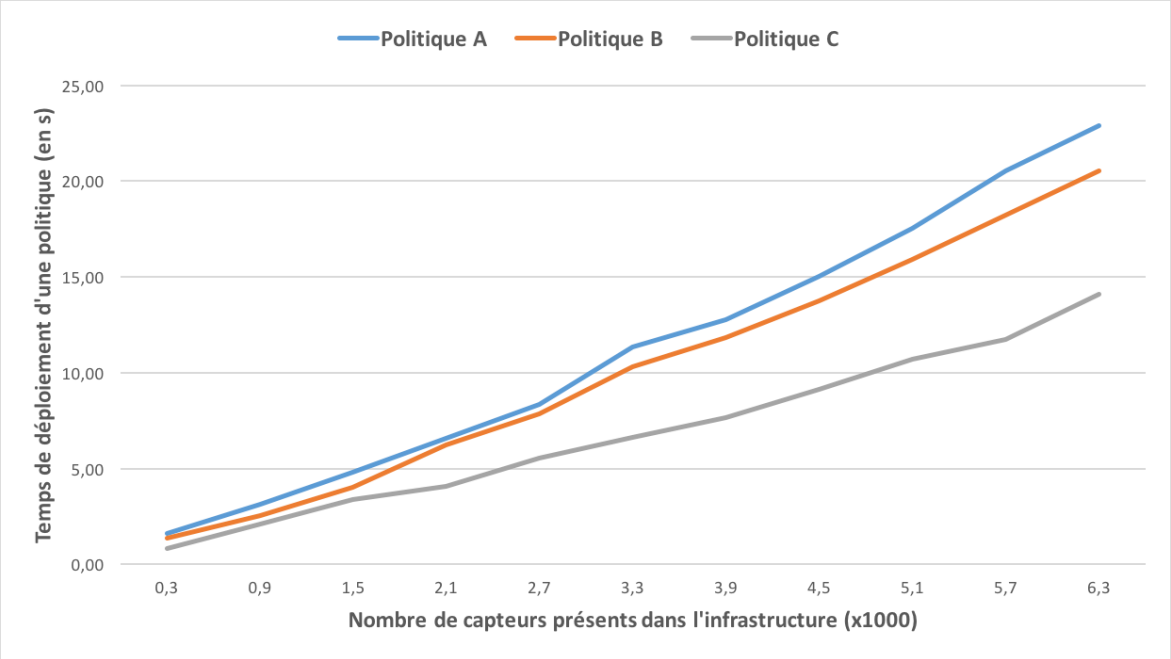


FIGURE 7.10 – Temps de déploiement de trois politiques indépendantes en fonction du nombre de capteurs présents dans l'infrastructure

tification des plateformes sur lesquelles sont accessibles l'ensemble de ces capteurs est plus complexe. Toutefois, à l'échelle de 6300 capteurs, le temps de déploiement reste inférieur à 30 secondes, ce qui est un gain par rapport au temps de déploiement manuel.

▶ ***Limites***. A travers le tableau TAB. 7.4, nous pouvons observer que le déploiement des politiques à l'échelle d'un bâtiment intelligent génère un grand nombre de fichiers sources qui devront être transférés manuellement par un expert réseau sur l'infrastructure de capteurs, *par ex.*, le déploiement de la politique fil rouge à l'échelle de 50 bureaux nécessite de déployer manuellement 104 fichiers. En revanche, par rapport à un développement traditionnel, l'expert réseau n'a pas eu besoin d'exprimer les politiques mettant en œuvre les intentions métier d'un ingénieur logiciel pour chaque plateforme de l'infrastructure.

Déploiement sur une infrastructure partagée par un proxy de composition

Le proxy de composition a pour responsabilité de partager une infrastructure de capteurs. Ce proxy collecte et compose plusieurs politiques de collecte de données afin de re-générer automatiquement des sous-politiques spécifiques aux plateformes.

Dans le cadre d'infrastructures partagées par un grand nombre d'utilisateurs à l'échelle de villes et bâtiments intelligents, de nombreuses politiques viendront à être enregistrées à travers le proxy.

L'objectif de cette validation est de vérifier que le temps de déploiement de politiques de collecte de données au sein du proxy de composition reste acceptable pour un ingénieur logiciel. Nous réutilisons la métrique *Temps de déploiement* introduite en SEC. 7.3.3.

Temps de déploiement : Le temps nécessaire à l'identification des plateformes cibles et à la création des sous politiques de collecte de données.

► ***Protocole***. Nous utilisons un proxy de composition reposant sur un modèle d'infrastructure de bâtiment intelligent défini à l'échelle de 50 bureaux. Les politiques *modèles* définies permettent de mettre en œuvre une politique de collecte de données à l'échelle d'un bureau. Afin de cibler l'ensemble des bureaux avec les politiques A, B et C, nous avons donc enregistré, à travers le proxy de composition, 50 mises en œuvre de la politique *modèle* A (une par bureau), puis 50 mises en œuvre de la politique *modèle* B (une par bureau) et enfin 50 mises en œuvre de la politique *modèle* C (une par bureau). Un total de 150 politiques est alors déployé sur l'infrastructure de capteurs.

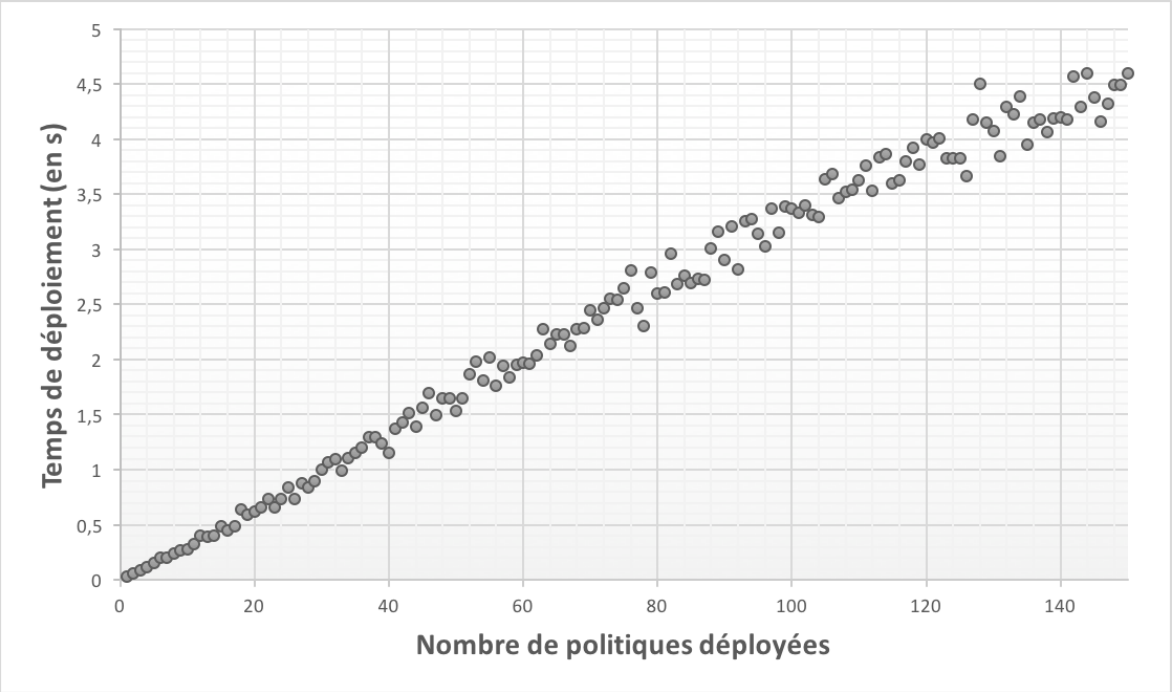


FIGURE 7.11 – Déploiement de 150 politiques de collecte de données à travers un proxy de composition

► ***Observations***. La figure FIG. 7.11 présente le temps de déploiement (composition et re-génération) de chaque politique au fur et à mesure de l'enregistrement de nouvelles politiques. Nous remarquons que le proxy de composition n'introduit pas de complexité puisque le temps de déploiement reste une fonction affine par rapport aux nombres de politiques enregistrées. Nous observons ainsi que le proxy de composition peut être utilisé à l'échelle de larges bâtiments intelligents.

► ***Limites***. L'expérimentation conduite a correctement partagé l'infrastructure de capteurs lors du déploiement des politiques A, B, et C (*cf.* SEC. 7.2.4) à l'échelle de 50 bureaux. Nous identifions toutefois un risque si une politique « agressive », *i.e.*, mobilisant les capteurs en continu (*par ex.*, période inférieure à la seconde pour les capteurs périodiques) est déployée. En effet, le déploiement d'une activité `PeriodicSensor` récupérant des valeurs à une faible période sur une plateforme alimentée par batterie risque de vider la batterie et de rendre inactive cette plateforme au détriment des autres activités, pouvant provenir de politiques différentes, déployées.

7.4 Limites à la validité

▶ ***Mise en oeuvre logicielle***. La mise en oeuvre des contributions a été effectuée au sein d'un prototype et vérifié par des tests unitaires et d'intégration. En revanche, si certains résultats ont pu être perturbés par des bogues, nous pensons qu'ils sont assez proches de ce qui était attendu pour considérer les éventuels bogues restants comme non préjudiciables aux résultats.

▶ ***Infrastructure de tests***. Dans le cadre des expérimentations à l'échelle de villes et bâtiments intelligents, nous avons utilisé des générateurs d'infrastructure afin de faire varier librement des paramètres comme le nombre de capteurs ou de plateformes disponibles. Ces générateurs sont basés sur des infrastructures réelles déployées dans le cadre de projets de ville intelligente (*par ex.*, SmartSantander [SMG+14]) ou campus intelligent (*par ex.*, SmartCampus [CJMR14]). Nous sommes ainsi confiants que cette infrastructure simulée reproduit au mieux les infrastructures existantes en nombre de plateformes et en type de topologie.

▶ ***Déploiement des politiques***. Les contributions proposées permettent de composer des politiques de collecte de données et de générer leur code associé. Les ingénieurs logiciels et experts réseaux obtiennent ainsi un ensemble de fichiers source prêts à être transférés sur l'infrastructure de capteurs. En revanche, nous n'effectuons pas de déploiement dynamique du fait que certaines plateformes ont besoin d'être reprogrammées avec un nouveau micro-logiciel. Nous laissons donc à la responsabilité de l'expert réseau de transférer ces fichiers sources sur l'infrastructure de capteurs. Toutefois, le principe de séparation des préoccupations reste appliqué puisque l'expert réseau a pour responsabilité de maintenir l'infrastructure de capteurs, et donc de reprogrammer les plateformes lorsque cela est nécessaire.

7.5 Conclusion

Ce chapitre nous a permis de valider, à partir de critères d'acceptation, les objectifs identifiés en Chap. 3. Nous avons notamment vu que les contributions présentées dans ce document permettent, pour un ingénieur logiciel, la définition de politiques de collecte de données indépendantes du matériel et à leur déploiement sur des infrastructures de capteurs hétérogènes et partagées. Puis, nous avons procédé à une évaluation de nos contributions à l'échelle de ville et bâtiments intelligents afin de vérifier leur applicabilité sur des infrastructures à plusieurs milliers de capteurs.

7.5.1 Retour sur les objectifs

Nous avons utilisé le scénario fil rouge afin de montrer comment les objectifs soulevés dans le chapitre Chap. 3 sont atteints à travers nos différentes contributions. Nous généralisons ici la réponse aux objectifs grâce à nos contributions :

▶ ***Un ingénieur logiciel ne manipule que des concepts métiers***. Le langage spécifique au domaine permet d'exprimer des politiques de collecte de données, indépendantes du matériel, et à haut niveau d'abstraction. L'ingénieur logiciel décrit, à l'aide d'activités et de flux de données, les traitements à effectuer sur les données de capteurs. Le déploiement de la politique ne nécessite pas de connaissances sous-jacentes sur les infrastructures de capteurs : dans le cas d'une infrastructure non-partagée, l'ingénieur logiciel utilise un modèle d'infrastructure (*cf.* Def.5.3.3 et encapsulant les préoccupations liées réseaux de capteurs) fourni par un expert réseau, ou dans le cas

d'une infrastructure partagée, le proxy de composition (*cf.* Sec. 6.3.2) a pour responsabilité de déployer la politique sur l'infrastructure et de la composer avec la politique couramment exécutée.

▶ ***Une politique est adaptée aux capacités matérielles***. Lors de son déploiement, la politique de collecte de données est décomposée en sous-politiques de collecte de données spécifiques aux infrastructures de capteurs grâce à l'opérateur *deploy*, *cf.* Eq. 5.6. La génération de code mettant en œuvre ces sous-politiques utilise des modèles de variabilité (*cf.* Sec. 5.2) représentant les caractéristiques physiques et logicielles des plateformes afin de produire du code adapté.

▶ ***Une politique est projetée sur l'infrastructure***. La procédure de prédéploiement (*cf.* Eq. 5.5) associe chaque activité de la politique de collecte de données à un ensemble de plateformes candidates. L'opérateur *deploy* (*cf.* Eq. 5.6) procède ensuite, grâce à la stratégie de déploiement, à la sélection de la plateforme cible pour chaque activité puis à la construction des sous-politiques. La génération de code produit *in fine* du code directement transférable sur les différentes plateformes de l'infrastructure de capteurs.

▶ ***Une infrastructure doit être partagée entre différentes plateformes***. L'opérateur de composition $\oplus$ (Eq. 6.1) permet la création d'une nouvelle politique résultant de la composition de plusieurs autres politiques. Cette nouvelle politique *composée* est construite par union de politiques et fusion des capteurs identiques. Étant donné que cet opérateur construit une nouvelle politique, cette dernière est déployable grâce à l'opérateur *deploy* (*cf.* Eq. 5.6).

7.5.2 Retour sur l'évaluation

Les infrastructures liées aux villes et bâtiments intelligents contiennent plusieurs milliers de capteurs, *par ex.*, 12000 capteurs pour la ville de Santander et 70000 pour le stationnement à Moscou. Nous avons effectué une évaluation de nos contributions sur des infrastructures à l'échelle de villes et bâtiments intelligents. Nous avons observé que les différentes contributions sont applicables à de telles infrastructures.

▶ ***Définition de politiques de collecte de données***. La manipulation des activités et flux de données peut constituer une limite lors de l'expression de besoins métiers complexes. Nous avons vu que la définition d'une politique *template* et l'utilisation de l'opérateur de composition $\oplus$ permettaient de réduire le nombre de lignes de code. Nous avons notamment exprimé des politiques de collecte de données à l'échelle de plusieurs centaines de bureaux en quelques dizaines de lignes de code.

▶ ***Modèle d'infrastructure***. Le modèle d'infrastructure permet l'identification des plateformes cibles pour le déploiement d'une politique de collecte de données et l'adaptation aux ressources matérielles des différentes plateformes. Nous avons observé qu'un modèle d'infrastructure ayant 70000 capteurs était chargé en mémoire en moins de 30 secondes ce qui illustre son applicabilité sur de tels scénarios.

▶ ***Déploiement de politiques de collecte de données***. Nous avons pu observer que le temps nécessaire au déploiement de politiques de collecte de données était principalement dépendant du nombre de capteurs présents dans l'infrastructure, augmentant la complexité d'identification des plateformes cibles grâce à l'opérateur *place* de la stratégie de déploiement, et au nombre de politiques déployées sur l'infrastructure dans le cadre d'infrastructures partagées. Cependant, ces temps de déploiement restent négligeables par rapport au temps nécessaire à un expert réseau d'exprimer et déployer

manuellement des politiques de collecte de données sur les différentes plateformes d'une infrastructure.

Chapitre 8

Conclusion & Perspectives

Sommaire

8.1 Conclusion

Nous avons soulevé au début de cette thèse trois problématiques visant à permettre l'exploitation par un ingénieur logiciel d'une infrastructure de capteurs hétérogène et mutualisée :

— *(P_1, partage)* « comment une infrastructure peut-elle être partagée entre plusieurs politiques de collecte de données afin d'éviter des déploiements redondants ? »

— *(P_2, adaptation)* « comment une politique de collecte de données peut-elle être développée sur une infrastructure hétérogène ? »

— *(P_3, séparation)* « comment permettre à un ingénieur logiciel et un expert réseau de travailler ensemble à l'expression de collecte de données tout en restant concentré sur leurs compétences respectives ? »

Au sein de cette section, nous synthétisons les réponses à ces problématiques grâce à nos contributions et nous présentons leurs limites. Ensuite, au sein des tableaux Tab. 8.1, Tab. 8.2 et Tab. 8.3, nous positionnons nos contributions par rapport aux critères identifiés durant l'analyse de l'état de l'art (*cf.* Chap. 2). Nous rappelons également le fil rouge qui a été déroulé au cours des différents chapitres :

Fil rouge : Description

Au sein d'un bâtiment intelligent, on considère que chaque bureau est équipé de 2 capteurs de température ambiante (`T1_x` et `T2_x` où x est le numéro du bureau) ainsi que d'un capteur de température dans le bloc climatiseur réversible (`TAC_x`). Un capteur situé à l'extérieur du bâtiment (`T_OUTSIDE`) permet d'obtenir la température extérieure.

Une application doit être capable d'identifier des situations où, lorsque la climatisation est en fonctionnement, la différence entre la température moyenne du bureau et la température extérieure est supérieure au seuil de 8°C. Pour cela, cette application se connectera à un collecteur (`COLLECTOR`) où les données issues du réseau de capteur seront accessibles.

8.1.1 Modélisation et mises en œuvre de politiques de collecte de données

La manipulation de langages bas-niveau pour le développement de politiques de collecte de données demande à un ingénieur logiciel des connaissances propres aux infrastructures de capteurs, qui sont hors de son domaine d'expertise. Au sein de l'état de l'art, nous avons vu que les approches proposées pour augmenter le niveau d'abstraction restent spécifiques à une architecture donnée et n'offrent pas de mécanismes de réutilisation logicielle pour de futurs développements.

Afin de proposer une approche d'expression de besoins métiers indépendamment du matériel, nous nous sommes appuyés sur le paradigme de l'ingénierie dirigée par les modèles afin de proposer un méta-modèle d'activités (*cf.* Sec. 4.2.3). Les activités proposées décrivent un traitement à appliquer sur des données de capteurs et non une mise en œuvre spécifique à une architecture.

Ensuite afin de la rendre réutilisable, nous avons introduit un mécanisme de réutilisation hiérarchique (activité `PROCESS`, *cf.* Sec. 4.3.2) permettant d'encapsuler une politique à l'intérieur d'une autre politique de collecte de données et des opérateurs de

sélection (opérateur σ, *cf.* Eq. 4.7) et de tissage (opérateur ω, *cf.* Eq. 4.8) permettant de réutiliser une politique partiellement.

Ces deux approches sont complémentées par un langage spécifique au domaine afin d'augmenter le niveau d'abstraction pour l'ingénieur logiciel (*cf.* Sec. 4.2.4).

Ces contributions ont été présentées au sein du chapitre Chap. 4 et ont permis la validation de *(P_3, séparation)* du point de vue génie logiciel.

▶ ***Validation***. Nous avons exprimé le scénario fil-rouge en utilisant uniquement des activités proposées par le méta-modèle ce qui nous a permis de la valider l'indépendance vis-à-vis du matériel et le haut niveau d'abstraction pour l'ingénieur logiciel. Ensuite, l'utilisation hiérarchique d'une politique de collecte de données (permettant d'obtenir l'état de fonctionnement du bloc climatiseur) nous a également permis de valider le mécanisme de réutilisation totale de politiques. Enfin, nous avons également procédé à une extension de la politique fil-rouge à partir de fragments d'autres politiques de collecte de données afin de valider la réutilisation partielle.

▶ ***Limites***. Nous avons proposé une approche indépendante du matériel permettant d'exprimer des besoins métiers sur une infrastructure de capteurs à haut niveau d'abstraction. Afin d'offrir un ensemble d'activités cohérent à un ingénieur logiciel, nous avons étudié des travaux offrant des langages de manipulations de données et avons intégré ces actions au sein d'un méta-modèle. L'expressivité des politiques est ainsi limitée à cet ensemble d'action. Plus particulièrement, nous n'avons pas intégré d'activités permettant une interaction avec l'environnement grâce à des actionneurs. Si de telles activités permettaient à des politiques d'interagir avec l'environnement (*par ex.*, des boucles de rétroaction avec un équipement local), la gestion des conflits pouvant survenir lors de l'exploitation de mêmes actionneurs par plusieurs utilisateurs soulève de nouvelles perspectives de recherche (*par ex.*, un utilisateur A nécessite d'ouvrir une électro-vanne pour sa politique avec qu'un utilisateur B doit la fermer).

8.1.2 Adaptation automatique des politiques à la variabilité du matériel

Le déploiement d'une politique de collecte de données demande l'identification les plateformes cibles et une gestion des spécificités d'architecture matérielle pour chacune de ces plateformes. Ces tâches étant hors du domaine d'expertise de l'ingénieur logiciel, le déploiement est généralement laissé à la charge d'un expert réseau. De plus, au sein de l'art, nous avons vu que les approches proposées pour permettre un déploiement automatisé d'applications sur une infrastructure de capteurs n'offraient pas de mécanismes d'optimisations basés sur des critères non fonctionnels.

Chaque activité d'une politique de collecte de données est amenée à être exécutée sur une plateforme cible. Nous avons introduit une procédure de pré-déploiement qui, pour une activité, identifie automatiquement (à partir d'un modèle d'infrastructure), un ensemble de plateformes candidates (*cf.* Eq. 5.5). Ensuite, afin d'orienter le choix d'une plateforme cible par rapport à des critères non fonctionnels, nous avons introduit des stratégies de déploiement *cf.* Sec. 5.3). Ces stratégies définissent un opérateur *place* (*cf.* Eq. 5.1) permettant de retourner la plateforme cible satisfaisant le mieux les critères non fonctionnels parmi un ensemble de plateformes candidates. L'ensemble des activités destinées à être exécutées sur une plateforme cible est regroupé sous la forme d'une sous-politique de collecte de données grâce à l'opérateur *deploy* (*cf.* Eq. 5.6). Les données de capteurs circulent sur l'infrastructure d'une sous-politique à une autre grâce à des points d'extensions (*cf.* Sec. 4.3.3).

Dans une dernière étape, ces sous-politiques sont traduites sous forme de code grâce à un ensemble de générateurs de code qui, en se basant sur un modèle des fonctionnalités des plateformes, produit du code adapté aux capacités matérielles de chacune des plateformes cibles et prêt à être transféré.

Ces contributions ont été présentées au sein du chapitre CHAP. 5 et ont permis la validation de *(P_3, séparation)* du point de vue d'un expert réseau et *(P_2, adaptation)*.

▶ ***Validation***. Le déploiement automatique de la politique fil rouge au sein d'un bâtiment intelligent à partir d'un modèle d'infrastructure fourni par un expert réseau nous a permis de valider le fait que les préoccupations entre ingénieur logiciel et expert réseau restent dès lors séparées : l'ingénieur logiciel ne manipule pas de code spécifique à l'infrastructure ciblée et l'expert réseau ne manipule pas la politique de collecte de données. Le code ensuite généré, et mettant en œuvre les sous-politiques, utilise des bibliothèques spécifiques aux caractéristiques techniques de la plateforme cible, visant la capacité à adapter une politique au matériel et de projeter cette politique sur l'infrastructure. L'expérimentation du passage à l'échelle du déploiement automatisé sur de très grandes infrastructures de capteurs simulées nous a montré que le temps de chargement en mémoire du modèle d'infrastructure était linéaire par rapport au nombre de capteurs. De plus, l'expérimentation du déploiement automatisé de plusieurs politiques de collecte de données sur des infrastructures simulées contenant plusieurs milliers de capteurs nous a montré des temps de déploiements inférieurs à la minute. À titre de comparaison, nous considérons 15 minutes comme étant le temps nécessaire à un expert réseau pour écrire et transférer sur les différentes plateformes le code mettant en œuvre la politique fil rouge à l'échelle d'un bureau sans utiliser les contributions présentées.

▶ ***Limites***. Les utilisations d'un modèle du domaine et de modèles de variabilité sont une bonne solution pour représenter la diversité des plateformes et technologies de capteurs. Les modèles du domaine permettent de représenter les éléments d'un système et les modèles de variabilité permettent de représenter, à partir d'un ensemble fini d'éléments, les différentes configurations possibles pour chacun de ces éléments. Les technologies liées aux infrastructures de capteurs évoluant, les modèles de variabilité exprimés à un instant donné risquent toutefois de ne pas être applicables aux prochaines évolutions matérielles et nécessiteront donc des mises à jour permanentes. Ces mises à jour auront un coût qui, bien qu'inférieur par rapport à celui d'écrire un code mettant en œuvre la politique sur le nouveau matériel, mérite d'être mentionné. Il est dans nos perspectives de mettre à jour automatiquement les modèles de variabilité à partir des nouvelles plateformes déployées dans une infrastructure de capteurs.

8.1.3 Composition de politiques

Le principal problème des infrastructures de capteurs actuelles est lié au fait qu'elles soient spécifiques à un domaine et configurées pour une application avec peu ou aucune possibilité de réutilisation lorsque de nouveaux besoins sont identifiés [KBG+16]. De plus, lors du déploiement d'une nouvelle politique, un ingénieur logiciel ne connait pas *a priori* les politiques déjà exécutées sur l'infrastructure. L'état de l'art présente des solutions visant à partager une infrastructure de capteurs entre plusieurs applications, mais ces travaux ne permettent pas le partage de la plateforme (indispensable pour le passage à l'échelle de l'infrastructure) tout en étant indépendants du type de plateforme utilisé.

Afin de proposer une approche de partage indépendante des fonctionnalités du matériel, nous proposons un opérateur de composition $\oplus$ (*cf.* Eq. 6.1) agissant directement au niveau de la politique de collecte de données. Cet opérateur assimile les politiques de collecte de données à des graphes orientés et effectue une composition mise en parallèle pour obtenir une nouvelle politique de collecte de données.

Nous avons également introduit un service de composition, situé entre l'ingénieur logiciel et l'infrastructure de capteurs, ayant pour responsabilité de collecter les nouvelles politiques de collecte de données, de les composer avec la politique couramment exécutée sur l'infrastructure et de déployer le résultat de cette composition.

Ces contributions ont été présentées dans le chapitre Chap. 6 et ont permis de répondre à la problématique *(P_1, partage)*.

▶ ***Validation***. Lors d'une expérimentation, le déploiement de plusieurs politiques de collecte de données sur une même infrastructure a automatiquement composé les nouvelles politiques avec les politiques précédemment déployées, validant la capacité à partager une infrastructure entre différentes politiques. Une seconde expérimentation visant à vérifier le passage à l'échelle de l'approche avec le déploiement de 150 politiques sur la même infrastructure nous a présenté des temps de déploiement (avec composition) de l'ordre de quelques secondes. Nous avons pu également vérifier que le proxy de composition n'engendrait pas de complexité supplémentaire lors du déploiement.

▶ ***Limites***. Le proxy de composition permet de partager automatiquement l'infrastructure de capteurs lors du déploiement de nouvelles politiques de collecte de données. Ce proxy a pour responsabilité de composer une nouvelle politique avec la politique couramment exécutée et d'automatiser son déploiement. Nous identifions toutefois un risque d'explosion combinatoire lors de la recherche et fusion des capteurs identiques si une politique contient un grand nombre de flux de données. En effet, pour chaque capteur fusionné, l'opérateur $\oplus$ va identifier et réécrire chaque flux de données ayant comme source le capteur fusionné.

8.1.4 Intégration des contributions au sein de l'état de l'art

Lors du chapitre consacré à l'état de l'art (Chap. 2), nous avions identifié 13 critères permettant la comparaison des travaux existants. Nous positionnons ci-dessous nos travaux par rapport à ces critères.

Expression des besoins métiers

Nous avions identifié que les travaux de l'état de l'art relatif à l'expression de besoins métier n'offraient pas de contribution permettant l'expression de besoins métiers à un haut niveau d'abstraction (**critère 1**) pour des infrastructures de capteurs hétérogènes (**critère 3**) tout en offrant des mécanismes de réutilisation logicielle (**critère 5**). Le tableau Tab. 8.1 permet de positionner nos contributions par rapport aux critères de l'état de l'art.

Tableau 8.1 – Résumé des contributions par rapport aux critères de l'état de l'art de l'expression des besoins métiers

1. Niveau d'abstraction	Un langage spécifique au domaine (Sec. 4.2.4) permet à l'ingénieur logiciel de définir à haut-niveau d'abstraction une politique de collecte de données. Le langage s'appuie sur un méta-modèle de définition de politiques de collecte de données.
2. Cible des abstractions	La politique de collecte de données définit des intentions métier au niveau de l'infrastructure. Un opérateur de déploiement *deploy* (Sec. 5.4) assure la création automatique de sous-politiques spécifiques à chacune des plateformes.
3. Hétérogénéité du matériel	Les actions proposées au sein du méta-modèle de définition de politiques de collecte de données sont génériques : elles décrivent l'action à effectuer et non la manière de la mettre en oeuvre (Sec. 4.2.3). Les générateurs de codes exploitent des modèles de variabilité décrits au sein du modèle des fonctionnalités des plateformes afin de produire du code adapté aux caractéristiques de la plateforme cible (Sec. 5.5).
4. Transparence des communications	L'ensemble des problématiques réseaux est masqué à l'ingénieur logiciel. Lors du découpage de la politique de collecte de données générale en sous politiques spécifiques au matériel, les points d'extensions assurent automatiquement l'envoi et la réception de messages en provenance et à destination d'autres plateformes (Sec. 5.4.3).
5. Réutilisabilité d'artefacts logiciels	Nous permettons la réutilisation hiérarchique (sous forme d'encapsulation, *cf.* Sec. 4.3.2) d'une politique au sein d'une nouvelle politique de collecte de données. Les opérateurs de sélection (σ, *cf.* Eq. 4.7) et de tissage (ω, *cf.* Eq. 4.8), permettent de réutiliser un fragment de politique de collecte de données au sein d'une nouvelle politique de collecte de données. L'ensemble de ces mécanismes a été mis en œuvre au sein du langage spécifique au domaine afin d'être directement manipulable par un ingénieur logiciel.

Déploiement d'une application sur une infrastructure de capteurs

Nous avions identifié que les travaux de l'état de l'art relatifs au déploiement d'application sur une infrastructure de capteurs ne permettaient pas d'optimiser le déploiement par rapport à des critères non fonctionnels (**critère 7**). Le tableau Tab. 8.2 permet de positionner nos contributions par rapport aux critères de l'état de l'art.

Tableau 8.2 – Résumé des contributions par rapport aux critères de l'état de l'art du déploiement des applications

6. Identification des plateformes cibles	Le prédéploiement (*cf.* Eq. 5.5) d'une politique de collecte de données associe un ensemble de plateformes candidates au déploiement de chaque activité. L'opérateur *place* (*cf.* Eq. 5.1) sélectionne ensuite la meilleure plateforme pour le déploiement de l'activité par rapport à une fonction d'optimisation du déploiement définie par l'expert réseau.
7. Utilisation d'une fonction d'optimisation de déploiement	La stratégie de déploiement permet d'orienter le déploiement des activités sur l'infrastructure en fonction de critères décidés par l'expert réseau. La stratégie peut être statique (*i.e.*, à partir d'une représentation statique de l'infrastructure) ou dynamique (*i.e.*, prend en compte l'état courant de l'infrastructure).
8. Suivi des applications déployées et/ou de l'infrastructure	Les stratégies de déploiement dynamiques permettent d'orienter le déploiement en fonction de l'état courant de l'infrastructure de capteurs, *par ex.*, prise en compte du nombre d'activités déployées sur une plateforme.

Partage de l'infrastructure de capteurs

Nous avons identifié que les travaux de l'état de l'art relatif au partage des infrastructures de capteurs ne permettaient pas le partage d'une plateforme de capteurs (**critère 9**) au sein d'une infrastructure de capteurs hétérogène (**critères 11 et 13**). Le tableau Tab. 8.3 permet de positionner nos contributions par rapport aux critères de l'état de l'art.

Tableau 8.3 – Résumé des contributions par rapport aux critères de l'état de l'art du partage de l'infrastructure de capteurs

9. Partage de la plateforme	L'application de l'opérateur de déploiement *deploy* au résultat de la composition de plusieurs politiques (obtenu grâce à l'opérateur $\oplus$, *cf.* Eq. 6.1) permet d'associer des activités provenant de politiques différentes à une même plateforme.
10. Partage du réseau (isolation)	L'application de l'opérateur de déploiement *deploy* (*cf.* Eq. 5.6) au résultat de la composition de plusieurs politiques (obtenu grâce à l'opérateur $\oplus$) crée des points d'extensions permettant l'envoi et la réception sur le réseau de messages en provenance d'applications différentes. Chaque point d'extension sortant est lié à un point d'extension entrant permettant à chaque point d'extension entrant de ne recevoir que les messages qui lui sont destinés. Le principe d'isolation est alors respecté.
11. Approche indépendante du type de plateforme	L'opérateur de composition $\oplus$ (*cf.* Eq. 6.1) retourne une nouvelle politique de collecte de données contenant des activités indépendantes du matériel.

12. Applicabilité aux plateformes à ressources contraintes	Le partage d'une plateforme de l'infrastructure de capteurs ne nécessite pas de déployer au préalable un système d'exploitation ou une machine virtuelle. La sous-politique spécifique à la plateforme est directement traduite en code prêt à être transféré. Ainsi, l'approche nous semble applicable sur des plateformes à ressources contraintes. En revanche, nous n'avons pas effectué de mesures relatives à la consommation mémoire d'une politique.
13. Support de l'hétérogénéité de l'infrastructure	L'application de l'opérateur *deploy* (*cf.* Eq. 5.6) sur le résultat de la composition crée des sous-politiques spécifiques aux plateformes. Les générateurs de codes exploitent des modèles de variabilité décrits au sein du modèle des fonctionnalités des plateformes afin de produire du code adapté aux caractéristiques de la plateforme cible (Sec. 5.5).

8.2 Perspectives

Nous présentons dans cette partie les perspectives ouvertes par cette thèse.

8.2.1 Transfert industriel

Les infrastructures de capteurs vont être déployées dans de plus en plus de contextes au cours des prochaines années. Au sein des industries, l'émergence de ces infrastructures dans les chaînes de production permet la création de systèmes cyber-physiques et participe au paradigme de l'Industrie 4.0 [LBK15].

Nous prévoyons de transférer cette approche au sein d'une entreprise luxembourgeoise concevant des solutions d'apprentissage automatique pour des industries spécialisées dans la sidérurgie et les grilles énergétiques (*smart grids*). En particulier, afin d'étendre leurs solutions dans un contexte Industry 4.0, ils ont la nécessité de considérer des infrastructures de capteurs. Nos contributions permettraient alors à cette entreprise de définir des collectes indépendamment des préoccupations matérielles liées aux plateformes de l'infrastructure.

Au cours de ce transfert, nous prévoyons également d'effectuer une évaluation empirique du langage spécifique au domaine avec un protocole visant à vérifier si les activités proposées au sein du méta-modèle permettent de traduire les besoins métier spécifiques au domaine de l'industrie 4.0.

8.2.2 Gestion automatique de nouveaux types de plateformes de capteurs

Les modèles de variabilité logicielle permettent de représenter les différences ou les similitudes entre des logiciels appartenant à une même ligne de produits. Ces modèles résultent de la capture de la variabilité et de sa transcription sous forme de caractéristiques (*features*). Au sein du chapitre Chap. 5, nous avons étendu cette notion à la variabilité matérielle permettant de représenter les différences ou similitudes entre plateformes de capteurs.

Les modèles de variabilité représentent une vision de la variabilité connue au moment où ils ont été construits. Au fur et à mesure de l'évolution technologique, de nouvelles caractéristiques peuvent apparaitre et rendre les modèles de variabilité utilisés obsolètes. Au sein de notre approche, les modèles de variabilité sont utilisés pour identifier les contraintes de placement des activités (*cf.* SEC. 5.4.2) et pour produire du code adapté aux plateformes cibles du déploiement (*cf.* SEC. 5.5). Dans le futur, ces modèles de variabilité devront être mis à jour afin de permettre à de nouvelles plateformes d'être intégrées. Cette tâche de mise à jour est non-trivial pour un expert réseau. En effet, il doit comprendre et représenter sous forme de *feature* chaque nouvelle caractéristique matérielle de chacun des composants de la plateforme.

Au niveau logiciel, des techniques permettent de rétro-ingéniérer des modèles de variabilité à partir d'un ensemble de produits. Dans Shatnawi et al. [SSS16], les auteurs constatent que le développement complet d'une ligne de produits logiciel est une activité à haut risque puisqu'elle demande de développer les caractéristiques logicielles de chaque *feature* (ce que nous avons fait à travers les bibliothèques appelées lors de la génération de code).

Les auteurs proposent alors une approche visant à récupérer la variabilité architecturale d'un ensemble de variantes de produits en analysant statiquement leur code source orienté objet. Leur approche repose sur 4 étapes : *(i)* l'extraction d'une architecture orientée composants à partir du code source de chaque logiciel (définition d'une configuration), *(ii)* l'analyse de la similarité cosinus à partir d'une description textuelle de chaque composant de logiciel, *(iii)* identification des variantes de configuration (définition d'une famille de produits), et *(iv)* identification des dépendances entre les composants (définition de la ligne de produits).

Il serait intéressant d'étudier l'applicabilité de cette approche afin de récupérer automatiquement la variabilité architecturale d'un ensemble de plateformes de capteurs. À partir d'un modèle décrivant une architecture de plateforme considérée (*par ex.*, le modèle du domaine présenté en FIG. 5.2), les travaux de Allier et al. [ASSF11] permettraient de transformer ce modèle sous forme de composants. Les composants seraient ensuite comparés, à l'aide d'une similarité cosinus, aux *features* déjà existantes au sein des modèles de variabilité. Le résultat de cette comparaison permettrait la définition d'une famille de produits à jour et l'identification des dépendances produirait une nouvelle ligne de produits intégrant la nouvelle plateforme.

Une fois les nouvelles *features* intégrées au sein des modèles de variabilité, de nouvelles bibliothèques logicielles devraient alors être mises à la disposition des générateurs de code afin de produire du code adapté aux nouvelles plateformes. Une piste consiste à extraire des répertoires de code source afin de récupérer les bibliothèques logicielles correspondantes aux nouvelles fonctionnalités [DNRN15, BOL14, AS13].

8.2.3 Gestion des préoccupations énergétiques

L'économie d'énergie dans une infrastructure de capteurs est un problème important puisque des plateformes peuvent fonctionner sur batteries et être déployées dans des zones reculées. La consommation énergétique d'une plateforme de capteurs est directement liée au programme exécuté et à ses transmissions radio. Actuellement, l'économie d'énergie est effectuée au niveau du code et demande une expertise hors du domaine métier d'un ingénieur logiciel. De plus, au sein d'une infrastructure partagée, les économies d'énergie faites par une application peuvent être inefficaces face au déploiement d'une application consommatrice de ressources. Nous souhaitons donc explorer com-

ment les préoccupations énergétiques peuvent être résolues au moment du déploiement de la politique de collecte de données. Ainsi, ces préoccupations resteraient masquées pour l'ingénieur logiciel.

Une première piste qu'il nous semble pertinent d'explorer est de simuler avant de déployer. L'état de l'art propose à ce jour des approches comme Cooja [ODE+06] (spécifique au système d'exploitation Contiki [DGV04]) permettant de simuler l'exécution d'un programme sur une infrastructure de capteurs et de surveiller des paramètres tel que la quantité de données reçues/envoyées, la charge CPU et le niveau de batterie. À partir du résultat de la simulation, nous pourrions orienter le déploiement vers des plateformes moins sensibles énergétiquement [SHC+04]. Toutefois, les principaux points compliqués de cette piste sont d'obtenir un modèle de simulation de l'ensemble des plateformes de capteur déployées dans l'infrastructure et de permettre une simulation entre plusieurs plateformes de capteurs hétérogènes.

Une seconde piste potentielle est l'étude de l'augmentation de la période des capteurs périodiques pour réduire la consommation énergétique de la plateforme. Des travaux ont montré que l'utilisation de périodes fixes pour des phénomènes physiques variables n'est pas toujours efficace et peut impacter la consommation d'énergie [HDB04]. Toutefois, une augmentation de la période de collecte peut également dégrader la qualité des données envoyées à la politique de collecte de données et ainsi ne plus respecter la valeur *offset* définie par l'ingénieur logiciel. Des approches à base d'apprentissage automatique peuvent permettre d'identifier de telles situations lorsque des phénomènes physiques périodiques sont considérés [GDSW07]. Nous pourrions dès lors introduire un nouveau type de capteur *dynamique* comme étant un capteur à période adaptative et intégrant un modèle de prédiction des futures valeurs. Ainsi, plutôt que d'effectuer une mesure, le capteur pourrait extrapoler la valeur courante à partir de son modèle de prédiction. Ce capteur viendrait alors se positionner comme un proxy aux différents capteurs périodiques souhaités par l'ingénieur logiciel.

8.3 Liste des publications

Les contributions présentées dans cette thèse ont été publiées dans des conférences internationales à comité de lecture :

- Cecchinel, C., Mosser, S., & Collet, P. (2015, January). Software development support for shared sensing infrastructures : A generative and dynamic approach. In International Conference on Software Reuse (pp. 221-236). Springer, Cham.
- Cecchinel, C., Mosser, S., & Collet, P. (2016, April). Towards a (de)composable workflow architecture to define data collection policies. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (pp. 1344-1346). ACM.
- Cecchinel, C., Mosser, S., & Collet, P. (2016, December). Automated Deployment of Data Collection Policies over Heterogeneous Shared Sensing Infrastructures. In Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific (pp. 329-336). IEEE.

L'infrastructure utilisée pour la validation des contributions a été présentée dans « Cecchinel, C., Jimenez, M., Mosser, S., & Riveill, M. (2014, June). An architecture to support the collection of big data in the internet of things. In Services (SERVICES), 2014 IEEE World Congress on (pp. 442-449). IEEE. »

Bibliographie

[ABF+15] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, et al. Fit iot-lab : A large scale open experimental iot testbed. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pages 459–464. IEEE, 2015.

[ABJ+04] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler : an extensible system for design and execution of scientific workflows. In *16th ICSSDM*. IEEE, 2004.

[ACDFP09] Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella. Energy conservation in wireless sensor networks : A survey. 7(3) :537–568, 2009.

[ACN10] Sarfraz Alam, Mohammad MR Chowdhury, and Josef Noll. Senaas : An event-driven sensor virtualization approach for internet of things cloud. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–6. IEEE, 2010.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. 53(4) :50–58, 2010.

[AS13] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[ASSF11] Simon Allier, Salah Sadou, Houari Sahraoui, and Regis Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 214–223. IEEE, 2011.

[AVSPQ03] João Paulo Almeida, Marten Van Sinderen, Luís Ferreira Pires, and Dick Quartel. A systematic approach to platform-independent design based on the service concept. In *Enterprise Distributed Object Computing Conference, 2003. Proceedings. Seventh IEEE International*, pages 112–123. IEEE, 2003.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run. time. *Computer*, 42(10), 2009.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software : Practice and Experience*, 36(11-12) :1257–1284, 2006.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[BDRCD+11] Clément Burin Des Rosiers, Guillaume Chelius, Tony Ducrocq, Eric Fleury, Antoine Fraboulet, Antoine Gallais, Nathalie Mitton, Thomas Noël, and Julien Vandaële. SensLAB : a Very Large Scale Open Wireless Sensor Network Testbed. In *Networking 2011*, pages 241–253, Valencia, Spain, 2011.

[BHG+13] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. Riot os : Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[BL12] Niels Brouwers and Koen Langendoen. Pogo, a middleware for mobile phone sensing. In *Proceedings of the 13th International Middleware Conference*, pages 21–40. Springer-Verlag New York, Inc., 2012.

[BOL14] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer : An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79 :241–259, 2014.

[Bor14] Eleonora Borgia. The internet of things vision : Key features, applications and open issues. *Computer Communications*, 54 :1–31, 2014.

[BPRL05] Amol Bakshi, Viktor K Prasanna, Jim Reich, and Daniel Larner. The abstract task graph : a methodology for architecture-independent programming of networked sensor systems. In *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, pages 19–24. USENIX Association, 2005.

[CB11] Lianping Chen and Muhammad Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. 53(4) :344–362, 2011.

[CDCE+13] Michel Catan, Roberto Di Cosmo, Antoine Eiche, Tudor A Lascu, Michel Lienhardt, Jacopo Mauro, Ralf Treinen, Stefano Zacchiroli, Gianluigi Zavattaro, and Jakub Zwolakowski. Aeolus : mastering the complexity of cloud application deployment. In *European Conference on Service-Oriented and Cloud Computing*, pages 1–3. Springer, 2013.

[CFRS17] Maxime Colmant, Pascal Felber, Romain Rouvoy, and Lionel Seinturier. Wattskit : Software-defined power monitoring of distributed systems. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 514–523. IEEE Press, 2017.

[CGDLR13] Sylvain Cherrier, Yacine M Ghamri-Doudane, Stephane Lohier, and Gilles Roussel. Salt : a simple application logic description using transducers for internet of things. In *Communications (ICC), 2013 IEEE International Conference on*, pages 3006–3011. IEEE, 2013.

[CGLSM03] Luc Courtrai, Frédéric Guidec, Nicolas Le Sommer, and Yves Mahéo. Resource management for parallel adaptive components. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 7–pp. IEEE, 2003.

[CJMR14] Cyril Cecchinel, Matthieu Jimenez, Sébastien Mosser, and Michel Riveill. An architecture to support the collection of big data in the internet of things. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 442–449. IEEE, 2014.

[CKLZY12] Georgios Chatzimilioudis, Andreas Konstantinidis, Christos Laoudias, and Demetrios Zeinalipour-Yazti. Crowdsourcing with smartphones. *IEEE Internet Computing*, 16(5) :36–44, 2012.

[CRS07] Denis Conan, Romain Rouvoy, and Lionel Seinturier. Scalable processing of context information with cosmos. In *DAIS*, volume 7, pages 210–224. Springer, 2007.

[CSSW13] Peter Chapin, Christian Skalka, Scott Smith, and Michael Watson. Scalaness/nest : type specialized staged programming for sensor networks. In *ACM SIGPLAN Notices*, volume 49, pages 135–144. ACM, 2013.

[DCLT+14] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 211–222. ACM, 2014.

[dFPD+13] Claudio M de Farias, Luci Pirmez, Flávia C Delicato, Wei Li, Albert Y Zomaya, and José N de Souza. A scheduling algorithm for shared sensor and actuator networks. In *Information Networking (ICOIN), 2013 International Conference on*, pages 648–653. IEEE, 2013.

[DFR+13] Antônio Dâmaso, Davi Freitas, Nelson Rosa, Bruno Silva, and Paulo Maciel. Evaluating the power consumption of wireless sensor network applications using models. *Sensors*, 13(3) :3473–3500, 2013.

[DGM09] David Déharbe, Stephenson Galvão, and Anamaria Martins Moreira. Formalizing freertos : First steps. *SBMF*, 9 :101–117, 2009.

[DGMS07] Yanlei Diao, Deepak Ganesan, Gaurav Mathur, and Prashant J Shenoy. Rethinking data management for storage-centric sensor networks. In *CIDR*, volume 7, pages 22–31, 2007.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[DNRN15] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa : Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1) :7, 2015.

[DRI14] DRIEE. Les smart buildings : définition, May 2014.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads : simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. Acm, 2006.

[Epp92] David Eppstein. Parallel recognition of series-parallel graphs. *Information and Computation*, 98(1) :41–55, 1992.

[FMS11] Franck Fleurey, Brice Morin, and Arnor Solberg. A model-driven approach to develop adaptive firmwares. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 168–177. ACM, 2011.

[FNM+14] Francois Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. Kevoree modeling framework (kmf) : Efficient modeling techniques for runtime use. 2014.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[FRL07] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Towards a flexible global sensing infrastructure. 4(3) :1–6, 2007.

[Gar17] Gartner. Gartner says 8.4 billion connected "things" will be in use in 2017, Feb 2017.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot) : A vision, architectural elements, and future directions. 29(7) :1645–1660, 2013.

[GDSW07] Orazio Giustolisi, Angelo Doglioni, DA Savic, and BW Webb. A multi-model approach to analysis of environmental phenomena. *Environmental Modelling & Software*, 22(5) :674–682, 2007.

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. In *DCoSS*, volume 5, pages 126–140. Springer, 2005.

[GLVB+03] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language : A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[GMB+09] Tristan Glatard, Johan Montagnat, Raphaël Bolze, Frédéric Desprez, and E Modalis. On scientific workfkow representation languages. Technical report, Tech. rep., Laboratoire d'Informatique Signaux et Systmes de Sophia-Antipolis, 2009.

[GMLP08] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with moteur. 22(3) :347–360, 2008.

[HDB04] Barbara Hohlt, Lance Doherty, and Eric Brewer. Flexible power scheduling for sensor networks. In *Proceedings of the 3rd international symposium on Information processing in sensor networks*, pages 205–214. ACM, 2004.

[Hoo08] Andy Hooper. Green computing. *Communication of the ACM*, 51(10) :11–13, 2008.

[HRS13] Nicolas Haderer, Romain Rouvoy, and Lionel Seinturier. Dynamic deployment of sensing experiments in the wild using smartphones. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 43–56. Springer, Berlin, Heidelberg, 2013.

[IHLH12] Md Motaharul Islam, Mohammad Mehedi Hassan, Ga-Won Lee, and Eui-Nam Huh. A survey on virtualization of wireless sensor networks. 12(2) :2175–2207, 2012.

[ISH10] Muhammad Imran, Abas Md Said, and Halabi Hasbullah. A survey of simulators, emulators and testbeds for wireless sensor networks. In *Information Technology (ITSim), 2010 International Symposium in*, volume 2, pages 897–902. IEEE, 2010.

[JAS12] Cullen Jennings, Jari Arkko, and Zach Shelby. Media types for sensor markup language (senml). 2012.

[KBG+15] Imran Khan, Fatna Belqasmi, Roch Glitho, Noel Crespi, Monique Morrow, and Paul Polakos. Wireless sensor network virtualization : early architecture and research perspectives. 29(3) :104–112, 2015.

[KBG+16] Imran Khan, Fatna Belqasmi, Roch Glitho, Noel Crespi, Monique Morrow, and Paul Polakos. Wireless sensor network virtualization : A survey. 18(1) :553–576, 2016.

[KBGC13] Imran Khan, Fatna Belqasmi, Roch Glitho, and Noel Crespi. A multilayer architecture for wireless sensor network virtualization. In *Wireless and Mobile Networking Conference (WMNC), 2013 6th Joint IFIP*, pages 1–4. IEEE, 2013.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[KEW+10] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics : online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th international conference on Autonomic computing*, pages 141–150. ACM, 2010.

[KGMG07] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *ACM SIGPLAN Notices*, volume 42, pages 200–210. ACM, 2007.

[KLD02] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. 19(4) :58–65, 2002.

[Kru92] Charles W Krueger. Software reuse. 24(2) :131–183, 1992.

[Kru04] Philippe Kruchten. *The rational unified process : an introduction*. Addison-Wesley Professional, 2004.

[KW07] Holger Karl and Andreas Willig. *Protocols and architectures for wireless sensor networks*. John Wiley & Sons, 2007.

[LBK15] Jay Lee, Behrad Bagheri, and Hung-An Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3 :18–23, 2015.

[LC02] Philip Levis and David Culler. Maté : A tiny virtual machine for sensor networks. 37(10) :85–95, 2002.

[LEMC12] Ilias Leontiadis, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft. Senshare : transforming sensor networks into multi-application sensing infrastructures. In *European Conference on Wireless Sensor Networks*, pages 65–81. Springer, 2012.

[LOQS07] Matthieu Leclercq, Ali Erdem Ozcan, Vivien Quema, and Jean-Bernard Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *Proceedings of the 29th international conference on Software Engineering*, pages 209–219. IEEE Computer Society, 2007.

[LVCD13] Fei Li, Michael Vogler, Markus Claeßens, and Schahram Dustdar. Towards automated iot application deployment by a cloud-based approach. In *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on*, pages 61–68. IEEE, 2013.

[MEP17] Adrian Mizzi, Joshua Ellul, and Gordon Pace. An embedded dsl framework for distributed embedded systems : Doctoral symposium. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 374–377. ACM, 2017.

[MFHH05] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb : an acquisitional query processing system for sensor networks. 30(1) :122–173, 2005.

[MFT17] Saleh Mohamed, Matthew Forshaw, and Nigel Thomas. Automatic generation of distributed run-time infrastructure for internet of things. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pages 100–107. IEEE, 2017.

[MKD14] S. Madria, V. Kumar, and R. Dalvi. Sensor cloud : A cloud of virtual sensors. 31(2) :70–77, Mar 2014.

[MMW08] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask : Staged functional programming for sensor networks. In *ACM Sigplan Notices*, volume 43, pages 335–346. ACM, 2008.

[MP11] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks : Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3) :19, 2011.

[MV05] Matthew J Miller and Nitin H Vaidya. A mac protocol to reduce sensor network energy consumption using a wakeup radio. *IEEE Transactions on mobile Computing*, 4(3) :228–242, 2005.

[MZ10] Amjed Majeed and Tanveer A Zia. Multi-set architecture for multi-applications running on wireless sensor networks. In *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*, pages 299–304. IEEE, 2010.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. 27(2) :262–264, 1995.

[NW04] Ryan Newton and Matt Welsh. Region streams : Functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks : in conjunction with VLDB 2004*, pages 78–87. ACM, 2004.

[OAL14] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys (CSUR)*, 46(4) :47, 2014.

[ODE+06] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *Local computer networks, proceedings 2006 31st IEEE conference on*, pages 641–648. IEEE, 2006.

[PA11] Virginia Pilloni and Luigi Atzori. Deployment of distributed applications in wireless sensor networks. *Sensors*, 11(8) :7395–7419, 2011.

[PC15] Pankesh Patel and Damien Cassou. Enabling high-level application development for the internet of things. 103 :62–84, 2015.

[PFL16] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[Pic10] Gian Pietro Picco. Software engineering and wireless sensor networks : Happy marriage or consensual divorce ? In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 283–286. ACM, 2010.

[RCS08] Romain Rouvoy, Denis Conan, and Lionel Seinturier. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online*, 9(6), 2008.

[RM09] Romain Rouvoy and Philippe Merle. Leveraging component-based software engineering with fraclet. *annals of telecommunications-annales des télécommunications*, 64(1-2) :65–79, 2009.

[ROGLA06] Venkatesh Rajendran, Katia Obraczka, and Jose Joaquin Garcia-Luna-Aceves. Energy-efficient, collision-free medium access control for wireless sensor networks. *Wireless networks*, 12(1) :63–78, 2006.

[Sad07] Daniel A Sadilek. Prototyping domain-specific languages for wireless sensor networks. In *Workshop on Software Language Engineering, Nashville, Tennessee, USA*, 2007.

[SBW99] Clemens Szyperski, Jan Bosch, and Wolfgang Weck. Component-oriented programming. In *European Conference on Object-Oriented Programming*, pages 184–192. Springer, 1999.

[Sch06] Douglas C Schmidt. Model-driven engineering. 39(2) :25, 2006.

[SCZ+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing : Vision and challenges. *IEEE Internet of Things Journal*, 3(5) :637–646, 2016.

[SHC+04] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 188–200. ACM, 2004.

[SMBJ09] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model pruning. In *International Conference on Model Driven Engineering Languages and Systems*, pages 32–46. Springer, 2009.

[SMG+14] Luis Sanchez, Luis Muñoz, Jose Antonio Galache, Pablo Sotres, Juan R Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, Evangelos Theodoridis, et al. Smartsantander : Iot experimentation over a smart city testbed. 61 :217–238, 2014.

[SSS16] Anas Shatnawi, Abdelhak-Djamel Seriai, and Houari Sahraoui. Recovering software product line architecture of a family of object-oriented product variants. *Journal of Systems and Software*, 2016.

[SU05] ITU Strategy and Policy Unit. Itu internet reports 2005 : The internet of things. 2005.

[SW08] Stefan Schmid and Roger Wattenhofer. Modeling sensor networks. *Algorithms and Protocols for Wireless Sensor Networks*, 62 :77, 2008.

[TD11] Nicolas Tsiftes and Adam Dunkels. A database in every sensor. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 316–332. ACM, 2011.

[TDGS14] Ian J Taylor, Ewa Deelman, Dennis B Gannon, and Matthew Shields. *Workflows for e-Science : scientific workflows for grids*. Springer Publishing Company, Incorporated, 2014.

[TEL95] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading : Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995.

[TLA+10] Amirhosein Taherkordi, Frédéric Loiret, Azadeh Abdolrazaghi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. Programming sensor networks using remora component model. In *International Conference on Distributed Computing in Sensor Systems*, pages 45–62. Springer, 2010.

[UBFC14a] Simon Urli, Mireille Blay-Fornarino, and Philippe Collet. Handling complex configurations in software product lines : a tooled approach. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 112–121. ACM, 2014.

[UBFC14b] Simon Urli, Mireille Blay-Fornarino, and Philippe Collet. Spinefm & tocsin : un moteur de raisonnement et son interface de configuration, 2014.

[VDATH05] Wil MP Van Der Aalst and Arthur HM Ter Hofstede. Yawl : yet another workflow language. 30(4) :245–275, 2005.

[vdAtHW03] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, and Mathias Weske. Business process management : A survey. In Wil M.P. van der Aalst and Mathias Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2003.

[VDAVH04] Wil Van Der Aalst and Kees Max Van Hee. *Workflow management : models, methods, and systems*. MIT press, 2004.

[VDL03] Tijs Van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 171–180. ACM, 2003.

[VMD+05] Péter Völgyesi, Miklós Maróti, Sebestyen Dóra, Esteban Osses, and Akos Lédeczi. Software composition and verification for sensor networks. 56(1-2) :191–210, 2005.

[VSID15] Michael Vogler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. Diane-dynamic iot application deployment. In *Mobile Services (MS), 2015 IEEE International Conference on*, pages 298–305. IEEE, 2015.

[VVLM08] Jussi Vanhatalo, Hagen Völzer, Frank Leymann, and Simon Moser. Automatic workflow graph refactoring and completion. *Service-Oriented Computing–ICSOC 2008*, pages 100–115, 2008.

[YG02] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. 31(3) :9–18, 2002.

[YHE02] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM 2002. Twenty-First*

Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 3, pages 1567–1576. IEEE, 2002.

[YJS12] Won-Jae Yi, Weidi Jia, and Jafar Saniie. Mobile sensor data collector using android smartphone. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 956–959. IEEE, 2012.

[YRBL06] Yang Yu, Loren J Rittle, Vartika Bhandari, and Jason B LeBrun. Supporting concurrent applications in wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 139–152. ACM, 2006.

Annexe A

Modèle d'infrastructure utilisé pour la validation des objectifs

Nous présentons en Lst. A.1 un extrait d'une mise en œuvre (au format XML) du modèle d'infrastructure (Def.5.3.3). Nous rappelons que ce modèle, exprimé par un expert réseau, décrit :

— La stratégie de déploiement utilisée (*cf.* Sec. 5.3).

— La topologie réseau, *i.e.*, les connexions entre les plateformes ainsi que leur pondération (*cf.* Sec. 5.2.3).

— Les éléments variables de chaque plateforme contenue dans l'infrastructure (*cf.* Sec. 5.2).

Dans l'exemple ci-dessous, l'expert réseau utilise la stratégie de déploiement `ClosestToTheSensors` – Au plus près des capteurs (L.1, *cf.* Sec. 5.3). Ensuite, pour chaque plateforme, les éléments variables sont décrits en termes de `FEATURES` [1] :

— **Variabilité des contrôleurs** : chaque langage de programmation supporté par la plateforme est décrit dans le bloc `languages`, *cf.* L.29. Ici, la plateforme est programmée au moyen du langage *WIRING* (L.31).

— **Variabilité de la mémoire** : Le type de mémoire est représenté au sein de l'attribut `computation`, *cf.* L.3. Ici, la plateforme peu de mémoire (il s'agit d'une plateforme Arduino Uno).

— **Variabilité des capteurs** : chaque capteur est décrit dans le bloc `sensors`, *cf.* L.4. Un capteur a un identifiant, un éventuel port de connexion sur la plateforme (*pin*) et un ensemble de *features*. Ici, le capteur utilisé est un capteur de température (L.8) de constructeur Electronic Brick (L.7).

— **Variabilité des communications** : chaque moyen de communication est décrit au sein du bloc `communications`, *cf.* L.12. Ici, la plateforme a un moyen de communication de type XBEE (L.18), sans-fil (L.19) et de sens sortant (L.17).

— **Variabilité de l'énergie** : le type d'alimentation énergétique est décrit au sein du bloc `power`, *cf.* L.23. Ici, la plateforme est alimentée sur batterie (L.25).

Afin d'éviter à décrire manuellement l'ensemble des *features* au sein du fichier XML, nous utilisons le logiciel TOCSIN [UBFC14b] afin de bénéficier d'une interface de configuration graphique. TOCSIN permet la construction graphique d'une configuration

1. Pour des raisons de place et de lisibilité, nous ne présentons la description que pour la plateforme `P_OUTSIDE`. Le lecteur intéressé par l'ensemble du code pourra consulter la page : https://gist.github.com/ulrich06/9d8d48501927fb58005d14be490401fb

complexe par sélection de fonctionnalités décrivant les éléments du modèle du domaine. Deux captures d'écran, Fig. A.1 et Fig. A.2, illustrent respectivement l'ensemble des configurations possibles pour un capteur et le choix de configuration par sélection de fonctionnalités : le capteur configuré est un capteur de *température* et de type *Electronic Brick*.

Dans une dernière étape, l'ingénieur décrit les connexions réseaux pondérées entre les plateformes, L.38 – L.46.

Listing A.1 – Modèle d'infrastructure décrivant l'infrastructure de capteurs pour le bureau R1

```
<sensornetwork id="InfrastructureModel_R1" strategy="ClosestToTheSensors">
   <entities>
      <entity computation="Low" name="P_OUTSIDE" type="Uno">
         <sensors>
            <sensor id="OUTSIDE_T" pin="1">
               <features>
                  <feature>EBTemperature</feature>
                  <feature>Temperature</feature>
               </features>
            </sensor>
         </sensors>
         <communications>
            <communication>
               <features>
                  <feature>Way</feature>
                  <feature>Media</feature>
                  <feature>Out</feature>
                  <feature>XBEE</feature>
                  <feature>Wireless</feature>
               </features>
            </communication>
         </communications>
        <power>
            <features>
                <feature>Battery</feature>
            </features>
        </power>
         <languages>
            <language>
               <features>
                  <feature>Wiring</feature>
               </features>
            </language>
         </languages>
      </entity>
      <!-- suite sur le lien fourni ...-->
   </entities>
   <connections>
      <connection from="P_PLUG1" to="BR" weight=2/>
      <connection from="P_PLUG2" to="BR" /weight=2>
      <connection from="P_PLUG3" to="BR" /weight=2>
      <connection from="P_1_R1" to="P_2_R1" />
      <connection from="P_OUTSIDE" to="P_2_R1" />
      <connection from="P_2_R1" to="BR" weight=2/>
      <connection from="BR" to="COLLECTOR" weight=3/>
   </connections>
</sensornetwork>
```

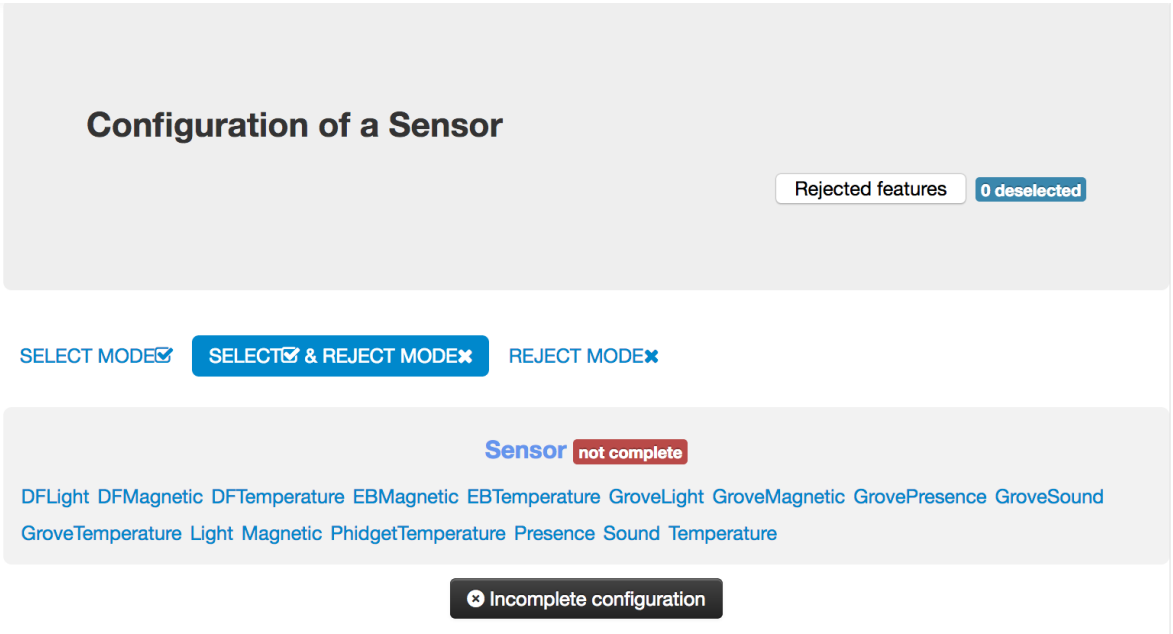


FIGURE A.1 – Illustration des choix de configuration possibles pour un capteur à travers TOCSIN

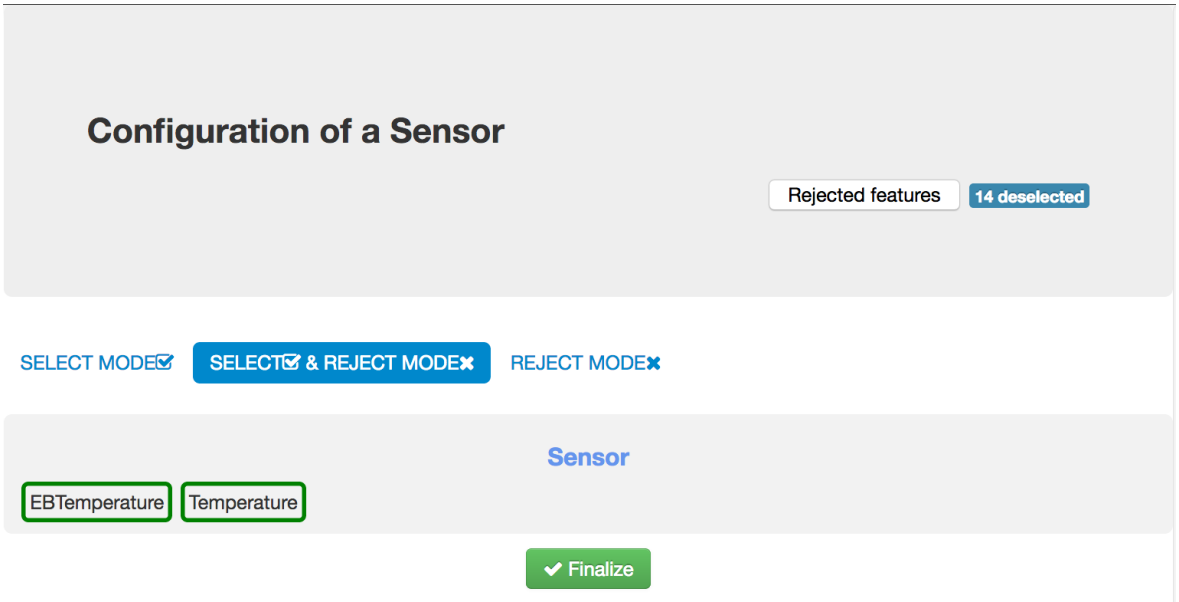


FIGURE A.2 – Illustration du résultat d'une configuration pour un capteur à travers TOCSIN

Annexe B

Notations & Opérateurs

Nous regroupons au sein de cet annexe, les notations et opérateurs introduits dans les différents chapitres.

B.1 Politiques de collecte de données

B.1.1 Ensembles

— Π : ensemble des politiques de collecte de données

— A : ensemble des activités d'une politique de collecte de données

— F : ensemble des flux de données d'une politique de collecte de données

— E : ensemble des points d'extension d'une politique de collecte de données

— L : ensemble des associations de points d'extension d'une politique de collecte de données

B.1.2 Opérateurs

— $\sigma : A \times \Pi$: opérateur de sélection (*cf.* Eq. 4.7)

— $\omega : \Pi \times \Pi \times L$: opérateur de tissage (*cf.* Eq. 4.8)

B.2 Modèle d'infrastructure

B.2.1 Ensembles

— I : ensemble des modèles d'infrastructure

— P : ensemble des plateformes d'un modèle d'infrastructure

— S : ensemble des capteurs d'un modèle d'infrastructure

— T : ensemble des topologies réseau

— C : ensemble des connexions topologiques

— Σ : ensemble des stratégies de déploiement

B.2.2 Opérateurs

— *accessible* : $P \rightarrow S$: capteur(s) accessible(s) au niveau d'une plateforme de la topologie réseau (*cf.* Eq. 5.2)

— *req* : $A \rightarrow S$: capteur(s) requis pour la mise en œuvre d'une activité (*cf.* Eq. 5.3)

— *isDeployable* : $A \times P \rightarrow \{\text{vrai ;faux}\}$: compatibilité du déploiement d'une activité sur une plateforme cible (*cf.* Eq. 5.4)

— *place* : $A \times P^{+} \times T \rightarrow P$: plateforme parmi un ensemble de plateforme maximisant les critères de la stratégie de déploiement (*cf.* Eq. 5.1)

— *deploy* : $\Pi \times I \rightarrow \Pi^{+}$: création des sous-politiques de collecte de données à partir d'une politique *pré-déployée* (*cf.* Eq. 5.6)

B.3 Composition de politiques

$\oplus$: $\Pi \times \Pi$: politique issue de la composition de deux politique de collecte de données

Résumé

Les réseaux de capteurs sont classiquement utilisés dans l'Internet des Objets pour collecter des données alimentant principalement des scénarios de villes et maisons intelligentes. Cependant, une connaissance approfondie des réseaux de capteurs est requise pour interagir correctement avec les systèmes déployés. Du point de vue d'un ingénieur logiciel, cibler de tels systèmes est un problème difficile. Premièrement, les spécifications des plateformes composant l'infrastructure de capteurs les obligent à travailler à un faible niveau d'abstraction et à utiliser des plateformes hétérogènes. Cela rend cette activité fastidieuse et peut conduire à un code exploitant de manière non optimisée l'infrastructure. Or, en étant spécifiques à une infrastructure, ces applications ne peuvent pas être réutilisées de manière transparente entre différents systèmes. De plus, le déploiement de ces applications est hors du champ de compétences d'un ingénieur logiciel puisqu'il doit identifier, au sein du réseau, la ou les plateforme(s) requise(s) pour supporter l'application. Enfin, l'architecture peut ne pas être conçue pour supporter l'exécution simultanée d'application, engendrant des déploiements redondants lorsqu'une nouvelle application est identifiée. Dans cette thèse, nous présentons une approche qui supporte (i) la définition de politiques de collecte de données à haut niveau d'abstraction avec une focalisation sur leur réutilisation, (ii) leur déploiement sur une infrastructure de capteurs hétérogène dirigée par des modèles apportés par des experts réseau et (iii) la composition automatique de politiques sur des infrastructures de capteurs hétérogènes. À partir de ces contributions, un ingénieur logiciel peut dès lors manipuler un réseau de capteurs sans en connaitre les détails, en réutilisant des abstractions architecturales directement disponibles à partir de l'expression des politiques, des politiques qui pourront également coexister au sein d'un même réseau. L'approche présentée est outillée et a été validée sur une infrastructure de campus intelligent. L'évaluation du passage à l'échelle de l'approche, effectuée sur une simulation de ville et bâtiments intelligents comportant plusieurs milliers de capteurs, nous a donné des résultats satisfaisants quant au temps requis pour le déploiement des politiques.

Abstract

Sensing infrastructures are classically used in the Internet of Things to collect data, typically supporting Smart Cities or Smart Homes use cases. However, a deep knowledge of sensing infrastructures is needed to properly interact with the deployed systems. From the point of view of a software engineer, targeting these systems is a challenging problem. First, the specifies of the platforms composing the sensing infrastructure compel them to work at a low level of abstraction and heterogeneous devices. This makes this activity tedious and can lead to code that badly exploit the network infrastructure. Moreover, by being infrastructure specific, these applications cannot be reused in a transparent way across different systems. Secondly, the deployment of an application is outside the domain expertise of a software engineer as she needs to identify, within the network, the required platform(s) to support her application. Lastly, the sensing infrastructure might not be designed to support the concurrent execution of various applications leading to redundant deployments when a new application is contemplated. In this thesis we present an approach that supports (i) the definition of data collection policies at high level of abstraction with a focus on their reuse, (ii) their deployment over a heterogeneous infrastructure driven by models designed by a network export and (iii) the automatic composition of the policy on top of the heterogeneous sensing infrastructures. Based on these contributions, a software engineer can exploit sensor networks without knowing the associated details, while reusing architectural abstractions available off-the-shelf in their policy. The network will also be shared automatically between the policies. The approach is tooled and has been validated on the top of a smart campus infrastructure. The scaling of the approach towards simulated smart cities and smart buildings with thousands of sensors has given us good results concerning the deployment time of policies.