Langage dédié au traitement des événements complexes et modélisation des usages pour les réseaux de capteurs

Alexandre Garnier

UNIVERSITE BRETAGNE LOIRE

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Alexandre GARNIER

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
Label européen
sous le sceau de l'Université Bretagne Loire

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 15 décembre 2016
Thèse n° : 2016EMNA0287

Langage dédié au traitement des événements complexes et modélisation des usages pour les réseaux de capteurs

JURY

Présidente :	**M^me^ Sara BOUCHENAK**, Professeur, INSA Lyon
Rapporteurs :	**M. Olivier BARAIS**, Professeur, Université de Rennes 1
	M^me^ Sara BOUCHENAK, Professeur, INSA Lyon
Examinateurs :	**M^me^ Anne-Cécile ORGERIE**, Chargée de recherche, CNRS
	M^me^ Hala SKAF-MOLLI, Maître de conférence, Université de Nantes
Invité :	**M. Laurent TOUTAIN**, Maître de conférence HDR, Télécom Bretagne, Rennes
Directeur de thèse :	**M. Jean-Marc MENAUD**, Professeur, École des Mines de Nantes - IMT
Co-directeur de thèse :	**M. Nicolas MONTAVONT**, Professeur, Télécom Bretagne, Rennes

Fire,

XKCD #1794

Remerciements

Trois ans s'achèvent.

Durant ces trois dernières années, il m'a été donné la chance, lors de mon doctorat, de découvrir et contribuer à ce vivier qu'est la recherche scientifique. Je souhaite remercier ici les personnes qui m'ont donné cette chance, les personnes qui ont su m'accompagner et m'aider tout au long du chemin.

Je remercie avant tout l'École des Mines de Nantes pour m'avoir donné les moyens logistiques, matériels et financiers de mener à bien les travaux de ma thèse.

Merci à tous les membres de l'équipe ASCOLA, nos échanges ont grandement contribué au défrichage des domaines exposés dans ce document. Je souhaite remercier ici en particulier Simon Dupont pour avoir un jour évoqué le traitement des événements complexes au détour d'une conversation.

Merci aux membres de mon jury, Sara Bouchenak, Olivier Barais, Anne-Cécile Orgerie, Hala Skaf-Molli et Laurent Toutain, pour avoir lu ce document et avoir enrichi ma réflexion sur les sujets abordés de leurs retours.

Un grand merci à Jean-Marc Menaud, mon directeur de thèse, pour m'avoir accompagné depuis le master ALMA en 2010, pour m'avoir permis de participer à tes travaux, dans un premier temps en stage puis aux débuts de l'aventure EasyVirt. Merci de m'avoir proposé cette thèse en juin 2013 et d'avoir été présent, tout au long de celle-ci, quand j'en avais besoin malgré un emploi du temps surchargé.

Un grand merci également à Nicolas Montavont, mon co-directeur de thèse, qui a su encadrer et réorienter au besoin avec talent les travaux de ma thèse, malgré la collaboration à distance. Merci d'avoir toujours su enrichir mes réflexions et travaux d'un regard autre, m'amenant toujours un peu plus loin dans ma démarche.

Je remercie aussi les membres de l'équipe RSM de m'avoir accueilli lors de mes déplacement à Rennes pour des discussions toujours enrichissantes et conviviales. Je remercie plus particulièrement Renzo Navas et Tanguy Kerdoncuff, pour leur aide sur la prise en main et la compréhension du fonctionnement des capteurs Homadeus.

Merci à Rémy Pottier, Frédéric Dumont et Guillaume Le Louët, mes prédécesseurs et collègues dans cette aventure doctorale, pour nos discussions sur tout et rien, votre humour sans faille, et vos conseils.

Merci à Flavien Quesnel puis Jonathan Pastor de n'avoir jamais eu ni l'un ni l'autre la conviction suffisamment forte de refuser catégoriquement une pause café, pour nos échanges scientifiques, philosophiques, politiques et surtout toujours de bonne foi.

Merci à Ronan-Alexandre Cherrueau, Florent Marchand de Kerchove de Denterghem, Walid Benghabrit et Gustavo Soto Ridd pour ces soirées de détente salvatrice qui venaient achever nos journées de dur labeur sur nos thèses et stages respectifs.

Merci à Anicet Bart, collègue de bureau intermittent, d'avoir contribué toujours avec bonne humeur à nos échanges et à l'ambiance de ce lieu de travail.

Je remercie aussi Martin Dargent, Thierry Bernard et Ludovic Gaillard, pour m'avoir accueilli et accompagné un temps au sein d'EasyVirt, puis pour tous ces moments partagés autour d'un café ou d'un repas.

Un immense merci enfin à ma famille et mes amis pour avoir été mon point d'ancrage dans la vraie vie, celle en dehors de la thèse. Sans eux bien des moments de doute n'auraient su s'envoler.

Table des matières

Table des figures

Liste des tableaux

Listings

Contexte

1

Introduction

1.1 Problématique

La révolution numérique des trente dernières années a bouleversé nos rapports à l'outil informatique. La démocratisation de l'ordinateur personnel, puis le phénomène internet ont été des tournants majeurs de cette histoire en marche. Plus récemment, la miniaturisation constante des supports informatiques a permis, dès les années 2000 l'apparition de l'informatique embarquée. Ses domaines d'application se sont dès lors développés, que ce soit dans la domotique, la téléphonie mobile, les divers appareils connectés (appareils de géolocalisation, *activity trackers*, montres, etc), mais également la numérisation grandissante des appareillages électriques dans des industries telles que l'automobile ou l'électroménager. De même, l'émulation grandissante autour du matériel libre, du DIY (*Do It Yourself* — faites-le vous-même), des plateformes d'achat ou de financement participatif ont contribué à la démocratisation des microcontrôleurs, permettant l'outillage à coût réduit pour un nombre théoriquement infini d'applications. Le succès de cartes comme l'Arduino ou le Raspberry Pi témoignent de l'engouement autour de la mise en œuvre à domicile d'appareillages allant de la domotique au mini-serveur applicatif. En parallèle, des innovations telles que l'introduction de l'IPv6, la démocratisation de l'énergie solaire photovoltaïque et des accumulateurs lithium-ion, mais également la sensibilisation des acteurs publics et industriels aux problématiques énergétiques et environnementales, ont permis l'aboutissement de solutions clé en main et l'on commence à voir apparaître quelques standards. Ce contexte constitue l'essentiel du terreau de l'*internet des objets*.

En pratique, l'internet des objets consiste en la mise en réseau et l'accès *via* Internet à divers matériels permettant une interaction avec le monde extérieur. On distingue en général deux types de matériel, auxquels correspondent deux types d'interaction :

1. les capteurs, permettant la mesure de différents phénomènes physiques, électriques, etc ;
2. les actuateurs, qui permettent d'agir sur leur environnement, généralement par l'intermédiaire d'un appareillage électronique dédié.

Capteurs et actuateurs coopèrent *via* divers protocoles afin d'assurer l'accès à leurs mesures et fonctionnalités. Chacun des nœuds assure l'accès à ses voisins, qui de proche en proche sont à même de couvrir tous les appareils. L'organisation d'un tel réseau, qu'on appelle communément un *réseau de capteurs*, suit généralement une topologie maillée, *a fortiori* à grande échelle.

La standardisation des appareils et protocoles constitutifs de ces réseaux de capteurs, accompagnée d'une réduction des coûts pour les domaines où ils remplacent un matériel autrefois spécialisé et onéreux, laisse présager leur adoption exponentielle. Le cœur de la problématique tient au fait que les capteurs ne

remontent que des données brutes. Un utilisateur lambda, sans outil spécifique, n'a alors pas les armes pour les exploiter. En outre, en se confrontant à un nombre et une variété grandissants de capteurs, cette problématique gagne sans cesse en importance. Domotique, station météorologique, relevés topologiques, sismologiques, suivi de la consommation électrique, etc ; amateurs comme professionnels contribuent chaque jour à étendre les champs d'application de l'internet des objets. Gartner estime que d'ici à 2020, plus de 25 milliards d'objets connectés seront utilisés quotidiennement [Gar14]. Non seulement ces champs d'application se multiplient, mais ils en viennent également à se rencontrer, à se recouper. Entrent alors en conflit différents domaines d'expertise, chacun avec ses propres besoins vis-à-vis du réseau de capteurs, voire même une conception différente dans sa représentation et la manière de l'appréhender. Si un premier nœud du problème porte sur la variété des données à traiter et les différences fondamentales dans les besoins justifiant les dits traitements, la quantité de données elle-même est loin d'être à négliger. Que ce soit dans la panoplie de capteurs ou les applications qui en découlent, le spectre s'élargit tout autant qu'il se densifie et l'évolution de concert de ces deux points constitue une problématique de première importance.

La nécessité de fournir un outil unique en vue de la gestion des données du réseau de capteurs se confronte alors, d'une part aux incompatibilités conceptuelles entre les différentes attentes des utilisateurs, d'autre part au débit du flux de données à traiter. Il est pourtant crucial de résoudre ces deux points pour promouvoir les usages des réseaux de capteurs et les voir se simplifier malgré l'évolution vers plus de capteurs et d'actuateurs toujours plus variés.

1.2 Objectifs

Les travaux de cette thèse portent sur l'identification et le traitement des données issues de réseaux composés de milliers de capteurs et d'actuateurs aux fonctionnalités diverses et variées. Un tel réseau de capteurs nécessite une gestion pensée pour les multiples usages qui lui seront consacrés. L'intégration des fonctionnalités des actuateurs à cette gestion est également un point crucial à prendre en compte. En outre, une telle gestion ne doit en aucun cas être un frein au traitement du flux de données, nécessairement en temps réel. En effet, de la nature même du flux de données, théoriquement sans fin et au taux globalement régulier, tout retard dans le traitement est cumulatif et doit donc être exclus. L'objectif de cette thèse est de proposer un outil de modélisation et de traitement des données issues d'un réseau de capteurs, permettant tant la détection et le traitement d'événements plus ou moins complexes depuis le flux de données, que l'accès aux fonctionnalités permettant une interaction avec l'environnement, dite fonctionnalités d'*actuation*, du réseau. Dans un premier temps il s'agit de proposer une modélisation du réseau et de ses données à même d'adresser spécifiquement les connaissances et besoins d'utilisateurs multiples. Le second plan porte sur l'accès à cette modélisation *via* un langage de requêtes permettant l'expression de traitement des données et de prise de décisions en conséquence, pouvant accéder aux actuateurs du réseau. La modélisation proposée se doit d'être configurable afin de s'adapter à tout réseau de capteurs et permettre de se focaliser sur les domaines de compétences et attentes des utilisateurs. *Ipso facto*, le langage de requêtes doit également être suffisamment modulaire pour s'adapter à toute configuration permise par l'étape de modélisation, et le moteur sous-jacent doit assurer le traitement en temps réel des dites requêtes. Chacun des deux points exerçant des contraintes sur l'autre, il va de soit qu'il est également nécessaire de s'assurer que toute évolution d'un aspect ne remette pas en cause la réalisation de l'autre.

1.3 Contributions

Les contributions de cette thèse se répartissent en deux parties.

La première partie concerne la modélisation d'un réseau de capteurs. L'objectif est ici double, puisqu'il s'agit de proposer une représentation des capteurs et de leur environnement reflétant les différents usages qui en sont faits, tout en assurant un accès aux données issues de ces capteurs qui ne soit pas contraint par les choix de modélisation. La solution choisie utilise la notion de multiarbre. Un multiarbre est un graphe

orienté acyclique (ou DAG pour *Directed Acyclic Graph*) dont la structure se définit comme l'intrication de plusieurs arbres. Puisqu'il s'agit d'un seul graphe, les nœuds de chacun des arbres sont nécessairement tous connexes. L'usage qui est fait du multiarbre ne correspond toutefois en rien à la topologie du réseau de capteurs.

Considérant qu'à un profil d'utilisation des données correspond un domaine d'expertise, pour lequel les capteurs s'inscrivent dans un environnement propre à ce domaine, nous introduisons la notion de *contexte*. Un contexte se définit comme l'ensemble des éléments qui, concrets ou abstraits, ne sont ni des capteurs, ni des actuateurs, mais permettent l'identification *contextuelle* des capteurs et actuateurs qui s'y inscrivent. Une représentation simple de tels contextes peut prendre la forme d'arbres, où les capteurs et actuateurs forment les feuilles des arbres alors que les nœuds parents viennent définir une hiérarchie, qui est le contexte à proprement parler. Dès lors, il est assez naturel de considérer qu'à deux usages distincts d'un même type de capteur ou d'actuateur correspondent deux contextes, deux arbres distincts. C'est là que l'implémentation en multiarbre prend tout son sens, puisqu'il suffit alors de s'appuyer sur les nœuds communs entre ces arbres pour les intriquer au sein du graphe. C'est ainsi que sur la base des multiples usages prédéfinis du réseau de capteurs, il est possible de construire un graphe unique permettant la modélisation du réseau de capteurs et de ses usages. L'unicité du multiarbre garantit en outre un accès non restrictif aux données, puisque se focaliser sur un usage ne ferme absolument pas l'accès au reste du graphe.

La modélisation des usages d'un réseau de capteurs au sein d'un multiarbre a fait l'objet de deux publications, l'une francophone [1] et l'autre internationale [2].

La seconde partie porte sur le traitement des données issues du réseau de capteurs et le langage associé. L'objectif principal est cette fois d'assurer un traitement en temps réel des données. Pour cette raison, la voie choisie a été celle du traitement des événements complexes (ou CEP pour *Complex Event Processing*) plutôt que le traitement en base de données : si quelques uns des *Systèmes de Gestion de Bases de Données* (SGBD) adressent aujourd'hui les problématiques de volumes de données liées au *big data*, le modèle de stockage des données restent insatisfaisant en ce qui concerne le traitement de données en temps réel. Les requêtes exprimées sont, pour la majeure partie, exécutées lorsqu'elles sont entrées par l'utilisateur, ce sur l'ensemble des données mesurées depuis le début du stockage en base. On parle de requêtes *volatiles* sur des données *persistantes*. Le CEP permet quant à lui une analyse au fur et à mesure des données, au moment où elles sont mesurées, par des requêtes pré-établies dont le but est de filtrer et combiner les données en guise de traitement. Dans ce cas, on renverse le modèle puisque les requêtes sont alors *persistantes*, et analysent des données *volatiles*.

Afin d'offrir aux utilisateurs cet accès au traitement des données, il est nécessaire d'associer au CEP un langage dédié (ou DSL pour *Domain-Specific Language*). Le choix a été fait de s'appuyer sur la modélisation des données en multiarbre pour proposer un langage concis, dont les principes reprennent ceux de XPath [CD+99], puisqu'il permet également le parcours implicite entre des nœuds distants dans l'expression de sélection de nœuds du graphe.

Le langage dédié au traitement des événements complexes pour les réseaux de capteurs, et s'appuyant sur la modélisation en multiarbre vue précédemment a fait l'objet d'une publication internationale [3].

1.4 Diffusion scientifique

Les travaux présentés dans ce document ont fait l'objet de publications présentées dans des conférences francophone et internationales avec comité de lecture :

[1] Alexandre Garnier, Jean-Marc Menaud, and Rémy Pottier. Sensorscript : un langage de requête dédié, orienté métiers, pour les réseaux de capteurs. In *Conférence d'informatique en Parallélisme, Architecture et Système*, 2015.

[2] Alexandre Garnier, Jean-Marc Menaud, and Rémy Pottier. Sensorscript : a business-oriented domain-specific language for sensor networks. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, pages 44–49. IEEE, 2015.

[3] Alexandre Garnier, Jean-Marc Menaud, and Nicolas Montavont. Bringing complex event processing into multitree modelling of sensors. In *Distributed Applications and Interoperable Systems*, pages 196–210. Springer, 2016.

1.5 Organisation du document

Ce document se décompose en trois parties. La partie I introduit le contexte de la thèse. Le chapitre 2 définit les concepts constituant la base de ce document que sont l'internet des objets, les réseaux de capteurs, les langages dédiés et le CEP. Le chapitre 3 inscrit les travaux de cette thèse dans l'état de l'art de la recherche, au travers d'une revue de la littérature scientifique autour des problématiques présentées auparavant.

La partie II se concentre sur les contributions de la thèse. Le chapitre 4 présente plus en détail la modélisation en multiarbre d'un réseau de capteurs. Le chapitre 5 est consacré au traitement des événements complexes s'appuyant sur une telle modélisation.

La partie III aborde l'implémentation de ces contributions. Le chapitre 6 présente SensorScript, le fruit des travaux de la thèse. Le chapitre 7 propose plusieurs évaluations expérimentales de cette implémentation. Enfin, le chapitre 8 apporte une conclusion au document et ouvre la voie à d'autres perspectives.

2

Définitions et enjeux

Différents domaines de recherche ont été soumis à l'étude de cette thèse. Les concepts qu'ils apportent constituent pour partie la base des travaux présentés dans ce document. Une présentation approfondie de ces concepts est dès lors nécessaire.

En premier lieu, l'accent sera porté sur l'internet des objets, l'évolution technologique qui l'accompagne et les problématiques associées. Les réseaux de capteurs, en particulier les réseaux de capteurs sans fil, feront l'objet d'une attention plus poussée parmi tout le panel des objets connectés. En deuxième lieu, les langages dédiés seront abordés, avec une distinction nette entre ce qu'ils sont et ce qu'ils ne sont pas. Enfin, le traitement des événements complexes sera abordé en détail.

Sommaire

2.1 Internet des objets

Depuis l'invention du transistor au milieu du XX^{ème} siècle, son usage s'est démocratisé dans une multitude de secteurs, amenant au fil des décennies des circuits électroniques, plus ou moins complexes, presque omniprésents dans tout appareil électrique. Au tournant du XXI^{ème} siècle, la miniaturisation et la baisse des coûts dans l'informatique a mené à l'émergence de ce qu'on appelle l'informatique embarquée. Il s'agit d'une évolution presque naturelle de l'électronique déjà présente dans les matériels concernés, mais d'une évolution majeure, puisqu'elle a permis l'introduction de standards, pré-existants dans l'informatique, notamment en ce qui concerne les réseaux. Les domaines d'applications de ces innovations gravitent autour de la domotique pour le public amateur, d'outils de mesure à grande échelle dans des cadre plus professionnels, pour tout ce qui va concerner l'étude des reliefs ou de la sismologie, mais également des mesures énergétiques plus précises au sein des entreprises et centres de données. Cette mise en réseau, dans des domaines variés, de matériels non spécifiques à l'informatique constitue de fait l'Internet des objets [Ash09].

En parallèle, des évolutions technologiques récentes participent de l'expansion de l'Internet des objets. Les technologies de communication sans fil, des standards GSM et WiFi au 6LoWPAN [Mul07] ont permis une réduction de la complexité et du coût dans la mise en place des infrastructures sous-jacentes. L'émergence récente du matériel libre (*open-source hardware*) invite à une standardisation de la couche logicielle des objets connectés.

L'engouement autour de l'Internet des objets se confronte cependant à deux principaux problèmes. D'une part, l'absence de standard *a priori* a amené à de multiples solutions partiellement ou totalement incompatibles ; en outre, les solutions industrielles s'appuient encore trop souvent sur des *firmware* et protocoles fermés, du fait d'intérêts bien éloignés de telles considérations. D'autre part, les besoins autour de l'Internet des objets a un impact sur la taille des réseaux mis en jeu, et très vite apparaît le risque d'interférences dans les communications entre les objets, surtout quand les protocoles réseau entre en conflit avec les standards WiFi ou GSM. À ce sujet, Gartner [Gar14] estime une évolution au triple du nombre d'objets connectés d'aujourd'hui à 2020, comme le montre la figure 2.1.

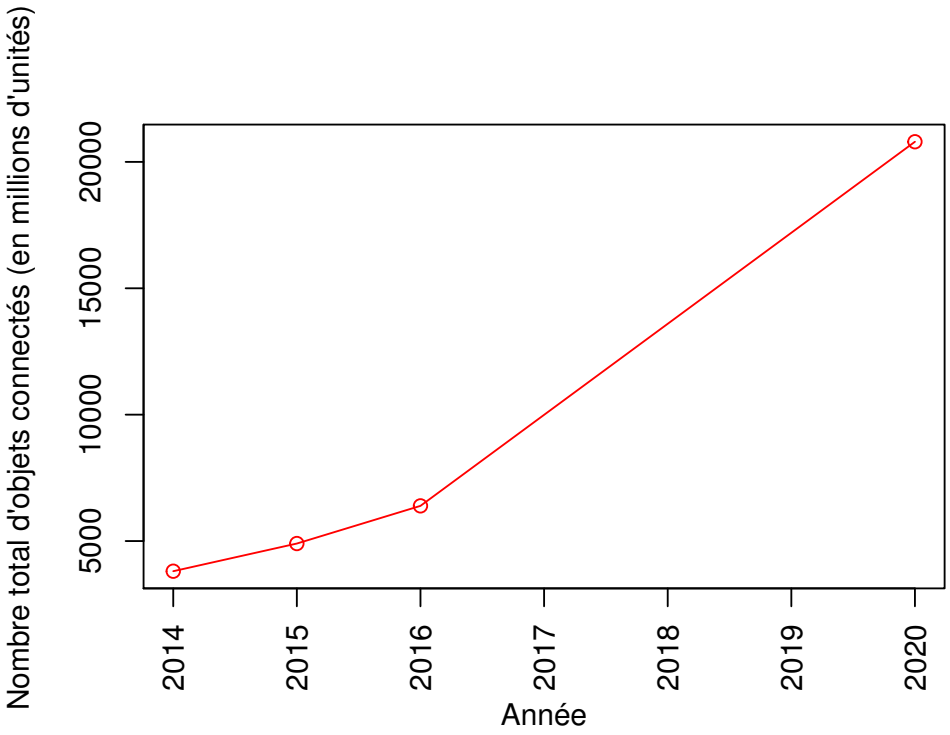


FIGURE 2.1 – Estimation de l'évolution du nombre d'objets connectés d'ici à 2020

2.2 Capteurs et actuateurs

En réponse à la problématique de l'absence de standards dans l'Internet des objets ont été introduits les réseaux de capteurs. Les réseaux de capteurs se proposent de ne considérer que les matériels constitutifs, d'un point de vue technique, de l'Internet des objets : les capteurs et actuateurs.

L'accroissement des capacités de calculs sur des circuits imprimés toujours plus petits a permis au fil du temps une portabilité de plus en plus d'applications de l'informatique. Avant même l'informatique personnelle ou la téléphonie, l'élaboration d'un outillage informatique dans nombre de domaines professionnels a très tôt été un des fers de lance de cette miniaturisation. Ces outils, majoritairement mono-tâche du fait de l'encombrement restreint auxquels ils étaient limités, s'inscrivant dans des pratiques pré-existantes, peuvent en effet être séparés en deux catégories :

- on a d'un côté les *capteurs*, dont la fonction première est la mesure de données simples ;
- de l'autre côté on trouve les *actuateurs*, permettant une action sur des éléments autres que le circuit informatique lui-même (ouverture et fermeture d'un portail coulissant par exemple).

Les domaines adoptant capteurs et actuateurs se sont diversifiés au fil des années, qu'il s'agisse de relevés météorologiques, sismographiques, topographiques, de sécurité incendie en intérieur ou extérieur, de domotique ou de suivi énergétique.

Cette évolution s'est de plus en plus confrontée à une problématique fondamentale. Le marché des capteurs et actuateurs a longtemps été le monopole d'entreprises spécialisées, avant tout pour des questions de coût. Par conséquent, les systèmes informatiques embarqués (ou *firmwares*) de ces capteurs et actuateurs étaient majoritairement fermés, et chaque entreprise était à même de proposer un ensemble d'outils fonctionnant très bien ensemble et en l'état, mais incompatibles avec la concurrence. Le besoin d'outrepasser cette limite s'est de plus en plus fait ressentir, la solution idéale étant la mise en place de standards permettant la communication de ces capteurs et actuateurs.

Une seconde problèmatique concerne le *medium* de communication lui-même. En effet, l'encombrement minimal d'un capteur ou d'un actuateur pâtit très vite du câblage d'un réseau de communication, plus encore dès qu'il s'agit de passer à une échelle plus vaste, avec potentiellement des centaines voire des milliers de capteurs.

2.2.1 Réseaux de capteurs

Les réseaux de capteurs [ASSC02a] s'inscrivent dans les solutions à cette problématique d'un standard de communication entre capteurs et actuateurs. En considérant l'Internet des objets comme un ensemble de capteurs et actuateurs équipant et interagissant avec les dits objets, l'introduction des réseaux de capteurs constitue une représentation unificatrice des objets connectés, tous ramenés à ces deux seules notions de capteurs et actuateurs. Ce socle constitue aujourd'hui la base conceptuelle des standards de communication entre capteurs, actuateurs et utilisateurs (ou applications clientes du réseau).

En pratique, différentes topologies des réseaux de capteurs existent. À ce sujet, Varshney et al. [RV06] introduisent la notion de nœud-puits comme point de collecte des données du réseau de capteurs, et opposent deux types de topologies différents : les réseaux à plat et les réseaux hiérarchiques.

Le réseau en étoile [AHK94] est la topologie à plat typique : absolument tous les capteurs et actuateurs sont en relation directe avec le nœuds-puits. Cette particularité pose évidemment problème vis-à-vis du passage à l'échelle, ce qui s'avère très vite limitant dans le déploiment de capteurs et d'actuateurs.

Les topologies hiérarchiques peuvent être distinguées en plusieurs catégories. Nous allons ici nous intéresser plus particulièrement à trois d'entre elles :

Réseaux en grappe

Dans un réseau en grappes, les capteurs sont regroupés en grappe, où tous les capteurs au sein d'une grappe communiquent avec un aggrégateur local ou *cluster head*, seul nœud de la grappe à communiquer avec le reste du réseau, réduisant ainsi le nombre de nœuds connectés au nœud-puits.

Heinzelman et al. [HCB02] considèrent le cas où, au sein d'une même région physique, des capteurs similaires mesurent des phénomènes similaires. Dans cette optique, le protocole LEACH, qui s'appuie sur les *cluster based networks*, propose une hiérarchisation en grappes permettant d'une part la sélection de *cluster heads* parmi les nœuds, d'autre part la formation de grappes visant à minimiser les coût réseaux entre les nœuds et le *cluster head* de chacune des grappes, ce de manière périodique. Dans un second temps, l'aggrégation des données dans chaque grappe est dévolue pour chaque période aux *cluster heads*.

Réseaux chaînés

Un réseau chaîné consiste en ce que chacun des nœuds ne communique qu'avec ses deux plus proches voisin, où chacun de ces liens représente le maillon d'une chaîne dont une des extrémités est le nœud-puits, réduisant d'autant l'énergie nécessaire à l'émission des paquets depuis les capteurs les plus excentrés.

Le protocole PEGASIS, par Lindsey et al. [LRS02], construit un réseau chaîné parmi les nœuds, soit de manière statique à l'initiative du nœud-puits, soit selon un algorithme dit «gourmand». Cet algorithme part du principe que chacun des nœuds connaît intégralement l'organisation du réseau. En partant du nœud le plus éloigné du nœud-puits, une chaîne est formée de proche en proche : à chaque étape, chaque nœud cherche son plus proche voisin encore libre (ne faisant pas déjà partie de la chaîne), ce jusqu'à atteindre le nœud-puits.

Réseaux en arbre

Un réseau en arbre propose une solution similaire basée sur un arbre d'aggrégation, permettant un traitement des données plus pertinent, puisqu'en se rapprochant du nœud-puits, les données de différents capteurs «fils» sont aggrégées par un même nœud «père».

Intanagonwiwat et al. [IGE00], s'appuient sur une hiérarchie en *tree based network* pour proposer un paradigme d'aggrégation des données qui apporte une notion d'intérêt pour telle ou telle information de la part du nœud-puits. À l'origine, le réseau est non-structuré, le nœud-puits «annonce» à ses nœuds voisins un intérêt pour un ensemble d'informations, qui relaient l'information à leurs propres voisins, permettant de proche en proche une propagation dans tout le réseau. Quand un nœud capable de délivrer au moins une de ces informations reçoit la déclaration d'intérêt, il renvoie en retour l'information. Une fois l'information collectée, un mécanisme de renforcement permet au nœud-puits de pondérer le chemin ayant permis la transmission de l'information : il indique au voisin qui a su lui ramener l'information qu'il est désormais dévolu à cette tâche, le renforcement étant propagé de manière similaire jusqu'au nœud ayant effectivement mesuré l'information, ce qui permet de construire l'arbre, le chemin d'une feuille au nœud-puits étant construit lors du dialogue intérêt-information.

2.2.2 Réseaux de capteurs sans fil

La solution à la seconde problématique d'encombrement matériel des systèmes de communication filaires consiste en l'introduction de systèmes de radiocommunication dans les réseaux de capteurs. On parle alors de réseaux de capteurs sans fil [ASSC02b].

S'appuyant sur les standards de communication issus de l'Internet des objets, ces réseaux deviennent chaque jour plus incontournables dès qu'il s'agit de faire interagir système informatique et monde réel. Ils sont toutefois sujet à de nouvelles problématiques. La largeur de bande, ou l'encombrement spectral. Des protocoles réseau, tels IEEE 802.15.4 [MBC+04], LoRa [All16] ou Sigfox [Dav15] ont été développés afin d'y répondre. En revanche, une seconde problématique, toujours prégnante celle-ci, concerne les

risques d'interférences lors du passage à l'échelle. Il s'agit là d'une problématique inhérente au support physique des communications sans fil : les ondes électromagnétiques. Avec un nombre grandissant d'ondes en circulation, le risque d'interaction entre deux ondes s'accroît. Or l'interaction entre deux ondes électromagnétiques chacune porteuse d'un signal peut-être source de perte de l'information des dits signaux.

Enfin, un enjeu dont l'importance ne cesse de grandir avec l'adoption des capteurs, encore accélérée par l'émergence des réseaux de capteurs sans fil, concerne la gestion du parc de capteurs et actuateurs, qu'il s'agisse du traitement des données mesurées par les capteurs ou l'intégration des actuateurs dans des processus d'automatisation. La standardisation des réseaux de capteurs a avant tout porté sur les couches basses des protocoles de communication. Reste à la charge de l'utilisateur de savoir *quoi* communiquer aux capteurs et actuateurs et d'en interpréter les données, avec potentiellement autant de langages de communication qu'il existe de matériels différents. S'attaquant à cette problématique, des protocoles plus haut niveau existent, sans pour autant qu'un réel standard ne se soit encore imposé. Parmi ces CoAP [SHB14], qui propose l'accès en UDP aux mesures d'un capteur dans une arborescence, ce *via* des URI.

2.3 Interfaces homme-machine

Le réseaux de capteurs font partie des systèmes informatique destinés à des utilisateurs. Une question primordiale concerne alors la façon de permettre leur utilisation. Le domaine consacré à cette considération concerne les *interfaces homme-machine* [LM90] (IHM). L'objectif premier d'une IHM est de répondre aux besoins des utilisateurs. Le travail effectué dans sa conception porte avant tout sur l'établissement d'une liste exhaustive de ces besoins et leur rationalisation en ce qu'on appelle des *cas d'utilisation*. Cette rationalisation porte sur le découpage en étapes des besoins utilisateurs, la factorisation de certaines de ces étapes. L'objectif est d'inscrire les besoins dans un nombre fini et raisonnable d'algorithmes.

La seconde étape consiste alors à trouver une solution pratique à même de répondre de façon satisfaisante à ces cas d'utilisation. Nous allons ici distinguer deux directions possibles sur un point de comparaison précis : la flexibilité de l'outil qu'est l'IHM. En étudiant les étapes en commun parmi les besoins identifiés, différentes manières d'aborder la conception de l'IHM sont envisageables. On peut considérer qu'à chaque besoin, à chaque séquence d'événements pré-identifiés correspondra un outil particulier. Dès lors l'interface, qu'elle soit graphique ou non, proposera pour chacun de ces besoins un scénario précis. L'utilisateur n'aura qu'à suivre les rails pré-établis pour arriver au résultat escompté. Cette solution est particulièrement adaptée aux applications dont les besoins sont restreints en nombre et clairement identifiés. La factorisation de ceux-ci ne sert alors qu'à l'optimisation du traitement des besoins, optimisation invisible à l'utilisateur, pour qui les cas d'utilisation restent bien distincts.

Dans le cas où les besoins sont autrement plus nombreux, voire s'entremêlent, une telle solution ne convient pas et il semble alors pertinent de proposer une solution à même de permettre directement à l'utilisateur de tirer partie de la factorisation des cas d'utilisation. Les langages dédiés font partie des implémentations d'une telle solution.

Langages dédiés

La communication avec les ordinateurs se fait, de manière ou non cachée, par le biais de langages. Que ce soit le langage machine tel que décrit dans la machine de Turing [Tur48] ou les langages de programmation, à différents niveaux d'abstraction vis-à-vis du fonctionnement logique de la machine, visant à offrir aux utilisateurs un accès plus intuitif aux capacités de calcul, tout passe par un langage informatique, réductible à une grammaire de Chomsky [Cho56]. L'intérêt premier d'un langage informatique, quel que soit son niveau d'abstraction, est de permettre, *via* l'accès à un nombre fini d'opérations simples, l'expression d'une infinité de calculs et d'algorithmes.

À considérer un cadre d'application plus restreint, tel que l'ensemble des cas d'utilisation évoqués plus haut, on constate que la mise à disposition d'un langage de haut niveau s'avère être un excellent choix en

terme d'IHM, dès lors qu'on retrouve des contraintes similaires : un grand nombre de besoins, potentiellement infinis, mais dans un cadre bien défini, constitués d'étapes, d'*opérations*, clairement identifiées et en nombre beaucoup plus raisonnable. C'est ce qu'on appelle les langages dédiés [Hud97].

Du fait d'un cadre applicatif plus restreint, circonscrit aux seuls besoins pré-établis, ils ne sont pas nécessairement Turing-complets (pas plus qu'ils ne sont nécessairement *non*-Turing-complets). L'objectif est avant tout, comme pour toute IHM, d'adresser tous les besoins utilisateurs pré-établis, idéalement de manière également efficace pour tous les besoins. En pratique, des contraintes diverses (incompatibilités entre les besoins, restrictions du langage de programmation sous-jacent, etc.) entrent en jeux et il s'agit surtout d'assurer au minimum pour chaque cas d'utilisation une accessibilité en proportion de la fréquence du besoin utilisateur lié. Outre la flexibilité, un autre intérêt des langages dédiés en tant qu'IHM porte sur l'extensibilité. Exprimés *via* une grammaire de Chomsky, l'ajout de fonctionnalités consiste assez naturellement en l'ajout de règles de développement afférentes. En outre, pour chacune de ces fonctionnalités, ce que l'étude des nouveaux cas d'utilisation associés aura révélé comme faisant déjà partie du langage pourra être réutilisé de manière transparente dès l'expression de la grammaire.

2.4 Traitement des événements complexes

Afin de simplifier l'expérience des utilisateurs d'un réseau de capteurs, il convient de faire émerger, depuis les données brutes, des informations compréhensibles par les utilisateurs. Le traitement des événements introduit une notion d'événement à même de s'inscrire dans les différents usages autour des capteurs et actuateurs.

La collecte des données issues d'un réseau de capteurs s'accompagne de nombreuses problématiques, toutes liées de près ou de loin à la taille et la diversité sans cesse croissantes de ces réseaux. Si le passage à l'échelle pose problème dans la quantitié d'informations à transiter sur des canaux aux capacités limitées, plus encore dans le cas des réseaux de capteurs sans fil pour lesquels s'ajoutent les risques d'interférence, c'est bien plus le traitement de ces données toujours plus nombreuses qui nous intéresse dans ce document.

Non sans rappeler les problématiques auxquelles se consacrent le *big data* [MCB$^+$11] depuis le début des années 2000, les spécificités des réseaux de capteurs, tant en ce qui concerne les contraintes matérielles que les besoins utilisateurs, ont amené peu à peu à considérer les données comme un flux continu à analyser en temps réel. Afin de se détacher des *tables* inhérentes aux SGBD traditionnels et bien plus adaptés à une logique d'analyse *a posteriori* des données, la notion d'événements est apparue. Dès lors il ne s'agit plus de traiter des données pré-inscrites dans des tables, mais de traiter des données constitutives d'événements, ce qu'on appelle le *traitement des événements complexes* [BK09].

Le terme *complexe* dans cette appellation fait référence à la façon dont se définissent les événements dans ce modèle. On vient de voir que les données constituent les événements. En effet, par définition tout flux de données constitue un fil d'évenements, dit atomique, pour lequel on considère que toute nouvelle valeur x du flux d'une donnée d constitue l'événement « x *est la nouvelle valeur de la donnée* d. » Un mécanisme fondamental dans le traitement des événements complexes concerne la *composition d'événements*. La composition des événements a pour objectif de permettre la construction d'événements dits *complexes*.

En comparaison des événements atomiques, qui ne sont que la transcription des flux de données, il s'agit d'événements plus concrets, plus parlants pour l'utilisateur. Les solutions de traitement des événements complexes mettent à la disposition des utilisateurs des opérateurs logiques permettant la composition des événements.

Considérons une cuisine équipées de deux capteurs, à savoir un détecteur de fumée et un capteur de température, et de deux actuateurs, permettant le déclenchement d'une alarme sonore et d'un jet d'eau. Tous ces appareils ont été installés afin d'assurer une sécurité incendie de la pièce. Le fait qu'il s'agisse d'une cuisine pose toutefois problème, la seule détection de la fumée ne permet pas d'assurer la présence d'un incendie, et deviendrait vite une nuisance pour le cuisinier, qui en utilisateur mécontent se hâterait de couper le système à la première fausse alerte.

C'est ici que la composition d'événements entre en jeu. Pour chacun des deux capteurs, le traitement des événements complexes considère un événement atomique, liées aux données de température t et au booléen f traduisant la présence de fumée. On souhaite ici construire l'événement « *il y a le feu à la cuisine*. » Cependant, l'activité de la pièce peut générer des fumées non-liées à un début d'incendie et ne permet donc pas de s'appuyer sur la seule détection de fumée. Le capteur de température permet toutefois de nous assurer de la nature de la fumée : si la température de la pièce dépasse un certain seuil s, on peut alors considérer qu'il y a effectivement un départ de feu dans la pièce. Notre événement complexe peut alors se traduire par : « f *est vrai* puis t *dépasse* s. » Le mot *puis* correspond ici à l'un des opérateurs logiques propres au traitement des événements complexes.

S'il est possible de composer des événements atomiques en événements complexes, la composition ne s'arrête toutefois pas là. Dans le même exemple il est envisageable que la détection de feu dans au moins deux pièces d'un même bâtiment entraîne le déclenchement de l'alarme dans tout le bâtiment, voire permette de prévenir les secours. Plutôt que d'avoir à composer un événement à partir des données de chacun des capteurs équipant toutes le pièces du bâtiment, il suffit alors de s'appuyer sur les événements déjà mis en place pour détecter le feu dans chacune des pièces.

La difficulté principale dans le traitement des événements complexes est de parvenir à lier les données à un flux atomique identifiable par l'utilisateur. Cette problématique d'identification des données issues du réseau de capteurs est au cœur des réflexions à la base des travaux présentés dans ce document.

2.5 Résumé

Dans l'optique de permettre de meilleures expériences utilisateurs autour des réseaux de capteurs, il est nécessaire d'opérer une transcription des spécificités des capteurs et actuateurs vers les usages maîtrisés par les utilisateurs. Une telle transcription porte sur deux points.

Pour le premier point, il est nécessaire de permettre l'expression de ces multiples usages au sein d'une même interface, idéalement suffisamment flexible pour s'adapter à tout type d'usage d'une part et tout type de capteurs et actuateurs d'autre part. Dans ce contexte, les langages dédiés apparaissent comme une solution pleinement adéquate.

Le second point porte sur la nécessité d'abstraction des données issues du réseau de capteurs en vue de proposer des éléments s'inscrivant dans les usages des différents utilisateurs. En outre, cette abstraction doit supporter le flux continu de données depuis les capteurs. Le traitement des événements complexes, en proposant une composition d'événements en temps réel depuis les données brutes, répond exactement à ces exigences.

3

État de l'art

Les travaux présentés dans ce document s'inscrivent au point de convergence de différents domaines de la recherche. De fait ils s'inspirent de, se comparent à, voire s'appuient sur des travaux portant sur ces domaines, qui seront présentés ici.

Dans un premier temps l'état de l'art portera sur l'identification des données dans le domaine des réseaux de capteurs et du big data*, plus particulièrement sur les deux courants majeurs qu'on retrouve aujourd'hui, à savoir la contextualisation des données d'une part, l'attachement sémantique d'autre part. L'accent sera ensuite porté sur le traitement des événements complexes, avec une mise en lumière de ses langages dédiés.*

Sommaire

3.1 Analyse des données issues d'un réseau de capteurs

3.1.1 Exploration de données

L'analyse des données issues d'un réseau de capteurs s'inscrit dans le cadre plus large de l'exploration de données, ou *data mining*. Dans leur étude de 2005, Gaber *et al.* [GZK05] proposent un panel des différentes techniques de *data mining*, notamment le partitionnement de données (ou *clustering*), la classification et l'analyse de séries temporelles.

Le principe du *clustering* est de proposer un partitionnement de différents signaux (ou série de valeurs pour les signaux discrets) selon un critère de similarité. L'une des méthodes de *clustering* les plus connues est celles des k-moyennes et ses variantes, qui opère un regroupement de telle sorte que chaque série soit placée dans la partition dont elle est le plus proche de la valeur moyenne. Guha *et al.* [GMMO00] ont néanmoins démontré qu'il n'est pas possible de proposer un algorithme déterministe plus efficace que O(nk). Un second inconvénient de la méthode est le fait que l'utilisateur doit préciser à l'avance le nombre de partitions que celle-ci déterminera, ce qui implique un certain degré de connaissance *a priori* des signaux observés pour que les partitions aient un sens.

La classification se base sur la reconnaissance de forme : l'analyse d'un signal permet de décider s'il fait partie de telle ou telle catégorie préalablement connue. La reconnaissance de forme se base généralement sur un ensemble de métriques calculées à partir du signal, la classification sur un arbre de décision. AWSOM [PBF03], s'intéresse notamment à la problématique de l'identification, non pas *a priori*, mais en temps réel des catégories. Cet algorithme utilise la transformée en ondelettes afin de compresser le signal en entrée, et se base sur la détection de corrélation entre le signal et les catégories afin d'opérer la classification. Cette méthode permet tout autant la classification d'une nouvelle série que la mise à jour des catégories en une passe, en O(logN), N étant la longueur de la série étudiée.

L'analyse de séries temporelles découle de méthodes statistiques plus classiques. Par exemple StatStream [ZS02] propose une méthode basée sur les transformées de Fourier cherchant à identifier en temps réel des fréquences dans les signaux sur une fenêtre glissante de taille fixe, la précision des fréquences à chercher étant fonction de la taille de la fenêtre.

Une problématique inhérente à ces méthodes est qu'elles n'abordent pas la problématique de l'analyse des données de la même manière. Si la classification et l'analyse de séries temporelles sont en mesure de fonctionner en temps réel, le clustering ne fonctionne que sur des séries de taille fixe et, au vu de la complexité, *a posteriori*. HPStream [AHWY04] se propose précisément d'apporter une solution permettant un *clustering* incrémental, qui en outre supporte le passage à l'échelle. Enfin, toutes ces méthodes visent à traiter les données du flux dans leur globalité, ce qui nécessite le stockage de ces données. Dans un cadre où l'on s'intéresse à un traitement des toutes dernières données seulement, auquel cas le stockage n'est plus nécessaire, nous allons dorénavant nous intéresser à deux pistes : les bases de données de séries chronologiques, qui intègrent la notion de série temporelle dans les bases de données, et le traitement des événéments complexes.

3.1.2 Bases de données de séries chronologiques

La notion même de séries temporelles, ou séries chronologiques [Ham94], permet de considérer un flux de données, dans son ensemble, comme une suite de données listées selon un ordre temporel, soit qu'il s'agisse d'un horodatage précis des données, soit qu'un intervalle fixe soit défini entre chaque couple de données. Dans le domaine des bases de données, cette définition permet de rationaliser un flux de données en un ensemble de relations plus conforme aux SGBD classiques. On parle alors de base de données de séries chronologiques (ou *time series database*).

L'étude de Golab et Öszu [GÖ03] établit de nombreux domaines d'applications à la gestion des flux de données en général et aux bases de données de séries chronologiques en particulier. Ainsi, Dasgupta et Forrest [DF96] proposaient dès 1996 la détection de données inattendues, de *comportements* hors norme,

depuis des séries chronologiques. En s'inspirant de la discrimination dite *soi - non soi* des systèmes immunitaires, ils proposent, pour chaque série chronologique, une collection de séquence de données, qui correspondent au *soi* de la série. La détection du *non soi* est assurée par des *détecteurs*, conçus de telle sorte qu'ils ne reconnaissent aucune des séquence du soi. Ainsi, toute séquence détectée sera identifiée comme non soi. La forme des séquence ainsi que l'algorithme d'un détecteur peuvent prendre de multiples formes, depuis une simple comparaison de chaîne à l'utilisation de tranformées de Fourier, parmi les exemples évoqués. On retrouve donc ici les méthodes statistiques, rendues applicables à un flux de données.

Toutefois, considérer les flux de données issues d'un réseau de capteurs comme des séries chronologiques, qui pis est dans les SGBD, pose rapidement un problème de passage à l'échelle : leur taille grandissant sans cesse par définition, on finit par atteindre, pour tout SGBD, un seuil critique. Des solutions s'attellent à cette problématique. Lazaridis et Mehrotra [LM03] proposent ainsi une compression des séries temporelles, par les capteurs eux-mêmes, afin de réduire la taille des paquets transitant entre capteurs et nœud-puits. Dans un soucis de satisfaire au mieux au besoin de traitement en temps réel des applications clientes du réseau de capteurs, ils proposent également un mécanisme de prédiction des données, ce en fonction des données déjà connues. Ces prédictions, pour être valable, s'appuient sur un prédicat qu'on retrouvait dans les travaux de Dasgupta et Forrest, à savoir que le flux de données montre un comportement identifié ou identifiable. Notons également que ce mécanisme de prédiction peut prendre place tant au niveaux des capteurs que du nœud-puits. L'étude proposée permet de montrer que compression et prédiction peuvent fonctionner de concert tout en gardant des performances satisfaisantes pour l'une et l'autre.

Dans l'optique d'adresser les séries chronologiques au sein des requêtes d'un SGBD, COUGAR [BGS01] se propose d'intégrer l'accès, en *pull* plutôt qu'en *push*, aux données des capteurs *via* les *Abstract Data Types* [GNT91], en charge de la construction de séries chronologiques, considérées comme des relations virtuelles, depuis le flux de données. En pratique, la représentation des séries temporelles est basée sur le modèle de séquence pour les bases de données de Seshadri *et al.* [SLR95]. Une distinction conceptuelle est donc faite entre *relations*, format traditionnel des données stockées en base, et *séquences*, format utilisé pour représenter les séries chronologiques représentant les flux de données issues des capteurs. Afin d'adresser les deux au sein d'une même requête, trois opérateurs sont mis à disposition, permettant la projection d'une séquence en une relation, le produit en croix d'une relation et d'une séquence, et l'aggrégation d'une séquence. Néanmoins, aucun de ces trois opérateurs ne permet l'expression d'aggrégation *temporelle* d'une séquence, c'est à dire l'aggrégation *a posteriori* d'un nombre donné des dernières valeurs d'une série chronologique.

Dans un contexte plus précis, Hammad *et al.* [HAE03] envisagent la détection du mouvement d'objet selon la définition de jointures « fenêtrées. » Une fenêtre établit ici une limite haute à l'intervalle de temps séparant la détection d'un même objet par deux capteurs différents. La définition de ces intervalles se fait au moyen d'un opérateur dédié intégré au langage de requêtes du SGBD.

Le point commun à ces solutions est qu'elles visent à adapter le flux de données, *via* sa gestion sous forme de série chronologique à un SGBD. Si cet aspect peut permettre de récupérer les données des capteurs à la demande plutôt que de crouler sous le flux de données, il restreint également la modélisation du réseau de capteurs et d'actuateurs, ainsi que l'expression de requêtes permettant d'en traiter les données, au modèle objet-relationnel. Nous verrons dans la suite en quoi cette restriction à de telles requêtes pose problème vis-à-vis d'une expérience simplifiée des utilisateurs.

3.1.3 Traitement des événements complexes

L'accroissement en taille des réseaux de capteurs a fait du traitement des données qui en sont issues une problématique de plus en plus prépondérante. Gaber *et al.* proposent une réflexion sur cette problèmatique et les pistes de recherche qui s'y consacrent [GZK05]. Les solutions abordant le passage à l'échelle du traitement des données sont alors catégorisées en deux groupes, que sont les solutions basées sur les *données* — des solutions visant à réduire le nombre de données à traiter par transformation statistique ou partitionnement du jeu de données — et les solutions basées sur les *tâches*, sur le traitement lui-même — qui introduisent

notamment la notion de temps et de taux des flux de données, avec le traitement sur une fenêtre glissante des plus récentes données et l'adaptation des algorithmes à la quantité de données produites dans cette fenêtre.

Le traitement des événements complexes fait partie de cette seconde catégorie de solutions et se concentre sur le traitement en temps réel des données. Comme on l'a vu, l'inscription des données dans des événements, composables et composés, permet alors une gestion flexible du flux de données qui permet l'identification de comportements au fur et à mesure que se construisent les événements qui les constituent.

L'une des toutes premières solutions s'inscrivant dans le traitement des événements complexes est STREAM [ABB+04]. L'objectif de STREAM est d'étendre le modèle objet-relationnel afin de permettre l'expression de requêtes dites *continues*. Plutôt que d'être exécutée à un instant donné sur l'état d'un jeu de données à cet instant, le fonctionnement d'une requête continue consiste en le traitement systématique de toute nouvelle donnée correspondant à son expression. D'un point de vue conceptuel, l'évolution du modèle se fait par la catégorisation de la représentation des données sous forme de *relations* et de *streams* (flux). Une relation représente l'état d'un jeu de données à l'instant considéré et correspond aux tables des SGBD. De fait, les relations supportent toutes les opérations habituelles sur les tables, qu'il s'agisse d'inscription, de mise à jour ou de suppression des données. Quant au *stream*, il représente en quelque sorte l'*émission* de nouvelles données, de nouvelles entrées des tables. Un *stream* ne permet donc de gérer que la production de nouvelles données et consistue de fait les événements atomiques dans le traitement des événements complexes.

La composition des événements consiste alors en la génération de nouveau *streams* en fonction des *streams* atomiques et des relations. Dans cette optique, des transformations entre relations et *streams* sont mises à disposition. En effet, la base objet-relationnel de STREAM ne permet des opérations qu'entre relations. Cette restriction est contournée par la possibilité de spécifier des fenêtres — de temps ou à taille fixe — sur les *streams* pour en faire des relations volatiles, au sens où leur existence est limitée au traitement au sein duquel elles sont créées, permettant alors la composition de nouvelles relations. Il est alors possible, à l'inverse, de traduire ces relations composites en *streams* composites. Dans ce cas, chaque nouvelle donnée de la relation composite constitue de fait le *stream* correspondant.

La notion de fenêtres de temps se retrouvait dans TelegraphCQ [CCD+] dès 2003. TelegraphCQ se présente comme un moteur de traitement des données pour PostgreSQL permettant une gestion en temps réel et non-bloquante d'un flux de données, ce qui le distingue d'un moteur de SGBD classique. La problématique première à laquelle se consacre TelegraphCQ est d'adapter la gestion du flux de données à un traitement des données de type relationnel et c'est à cette fin qu'est introduite la notion de fenêtre de temps, puisqu'elles permettent la construction en temps réel de tables volatiles que PostgreSQL est alors en mesure de traiter.

Dans un contexte consacré aux réseaux de capteurs, Saleh [Sal13] propose une conception du traitement des événements complexes sous forme d'automates à états finis. L'idée directrice est de considérer la description d'un événement composite comme un automate dont chacun des états correspond à un événement atomique. Un événement complexe consiste alors en la séquence des événements qui le constituent. Cette conception du traitement des événements complexes fait l'objet d'une implémentation pour le système d'exploitation embarqué TinyOS [LMP+05] et est une première étape vers la distribution du traitement des événements complexes au sein des capteurs.

Au delà du traitement des données, la taille grandissante des réseaux de capteurs pose également problème concernant l'encombrement du réseau, notamment pour les capteurs sans fil avec des risques croissants d'interférence dans les communications entre les capteurs et le nœud-puits. Afin d'aborder ce sujet, une piste envisage donc de tirer parti des capacités de calculs de l'informatique embarquée des capteurs pour leur déléguer une partie du traitement des données. C'est notamment l'objet de CEPEMS [SS13], qui étend la modélisation des traitements sous forme d'automate sur deux points. D'une part, la liste des opérateurs s'étoffe et permet, outre la séquentialisation des événements, l'expression d'opérateurs logiques tels le « *ou* » logique et la négation sur les événements atomiques. D'autre part, l'implémentation s'appuie sur cette conception du traitement sous forme d'automate en le considérant alors comme une séquence de plusieurs sous-automates. Il est alors possible de séparer ces sous-automates et de les distribuer au sein

du réseau de capteurs, idéalement au plus proche du point de convergence des données nécessaire à la détection de l'événement constituant l'état. C'est alors à la charge des capteurs de communiquer entre eux l'avancement parmi les états de l'automate, ce qui permet à terme d'aboutir au traitement complexe dans son ensemble.

Dans la même optique de considérer le traitement des événements complexes sous forme d'automates, Schultz-Møller *et al.* [SMMP09] proposent d'autres *patrons d'événements*, décrivant les sous-automates évoqués précédemment. Ils catégorisent notamment deux types de séquences d'états. La première catégorie concerne le filtrage des événements, permettant la réduction du jeu de données concerné par l'événement par vérification d'une condition sur ces données. Les deux états considèrent ici un même événement dont le jeu de données a été réduit, dans le second état, à un sous-ensemble du jeu de données du premier état. La seconde catégorie porte sur la séquence de deux événements. Il s'agit alors de deux événements indépendants l'un de l'autre, sinon que le premier doit avoir lieu *avant*, d'un point de vue *temporel*, le second.

Ces deux types de traitement simple des événements dépassent le cadre des opérateurs booléens classiques que sont le « *et* », le « *ou* » et la négation logiques et constituent un des points fondamentaux de la composition des événements.

3.1.4 Langages dédiés au traitement des événements complexes

Afin d'adresser le traitement des événements complexes, il convient de proposer aux utilisateurs une interface permettant l'expression de la composition des événements. Si une telle interface peut se présenter sous différents formats, nous allons ici nous intéresser aux langages dédiés à cette tâche.

Nous avons vu précédemment que STREAM constitue l'une des premières solutions permettant le traitement des événements complexes et qu'il s'appuie sur le modèle objet-relationnel pour établir des règles de composition sur les jeux de données constituant les événements. De fait, le langage dédié à STREAM, CQL [ABW03], s'inspire des *langages de requête structurée* (ou SQL pour *Structured Query Languages*), eux-mêmes dédiés au traitement des données dans le modèle objet-relationnel. De la même manière que STREAM étend ce modèle, CQL étend les SQL et permet l'expression de requêtes faisant intervenir tant les relations que les *streams*. À cette fin, nous avons vu que STREAM introduit des transformations entre les deux modèles de représentation des données. En pratique, l'expression d'une fenêtre sur un *stream* en vue de générer une relation sur la base de ce *stream* se fait via la spécification de cette fenêtre entre crochets, à la suite de l'identifiant du STREAM. À l'inverse, trois opérateurs (*Istream*, *Dstream* et *Rstream*) permettent la génération d'un *stream* depuis une relation, volatile ou non. Reprenons ici un exemple permettant d'illustrer ces deux transformations :

```
SELECT Istream(Open.*) FROM Open[Now], User
        WHERE seller_id = id AND state = 'CA'
```

Sans trop entrer dans les détails, *Open* est ici un *stream*, *User* une relation. La spécification de la fenêtre *Now* permet alors de générer une relation volatile au sein de laquelle il existera au plus une entrée correspondant à la condition de la clause *where* pour chaque entrée de la relation *User*, c'est à dire d'assurer un mapping $0..1 \leftrightarrow 1$ entre le *stream Open* et la relation *User*. Par défaut, le résultat d'une requête est considéré comme une relation. Ici au contraire, dans la clause *select*, un nouveau *stream* est généré par l'usage de l'opérateur *Istream* après filtrage des entrées d'*Open* en fonction de la condition de la clause *where*.

Comme nous pouvons le voir, la syntaxe reste très proche de celle des SQL et les seuls apports concernent le besoin de contourner les restrictions propres à la distinction conceptuelle entre relations et *streams*. Comme on vient de le voir, la structure même d'un SQL charrie les notions sous-jacent propres au modèle objet-relationnel. Dans l'optique de fournir à un utilisateur *lambda* une interface au traitement des événements complexes, un tel choix de langage dédié est loin d'être le meilleur qui soit. En effet, il sera nécessaire à chaque utilisateur de bien maîtriser les notions de tables, de relations, comprendre comment l'un et l'autre s'articulent dans les différentes clauses des requêtes. À ces notions vient en outre s'ajouter celle du *stream*,

avec les opérations de transformations depuis et vers les relations qui s'invitent. Alors qu'un utilisateur arrive avec des attentes spécifiques autour des capteurs — connaître la consommation énergétique des serveurs d'un centre de données pour un électricien par exemple — il est très vite confronté à tout un ensemble de concepts qu'il ne maîtrise pas forcément, et qui en tout état de cause font barrage à son besoin. Illustrons l'exemple de la consommation énergétique du centre de données avec une requête CQL :

```
SELECT Servers.power FROM Servers[Now], Datacenter
        WHERE dc_id = id AND id = 'dc1'
```

On voit bien ici qu'il est alors nécessaire à l'électricien de savoir que :

- les données d'un serveur sont dans un *stream* ;
- récupérer les dernières données d'un *stream* se fait par la fenêtre *[Now]* ;
- l'information permettant de lier un serveur à un centre de données se retrouve dans la relation *Datacenter* ;
- cette information fait l'objet d'un clé étrangère dans le *stream Servers*.

Ces quatre points sont autant de barrière à un usage le plus simple possible pour l'électricien.

Pour TelegraphCQ [CCD+] au contraire la transcription des flux en relations, relations alors à même d'être gérées par PostgreSQL, prend place exclusivement dans la clause *from* des requêtes, par l'introduction d'une clause *for* permettant la construction d'une fenêtre de temps selon un instant initial et une durée paramétrables, selon la forme :

```
SELECT [...]
FROM [...]
for(t=initial_value; continue_condition(t); change(t)) {
        WindowIs(Stream, left_end(t), right_end(t));
}
```

Bien évidemment on retrouve ici les mêmes reproches que l'on pouvait faire à CQL dans la simplicité d'usage du langage.

Dans l'objectif de développer des interfaces plus spécialisées, Sadilek décrit un processus de prototypage d'un langage dédié aux réseaux de capteurs [Sad07]. Il propose de joindre la description objet des règles de grammaire d'un langage *via* un métamodèle à la spécification de la sémantique du langage au travers d'un *langage métaprogrammable*. Un langage métaprogrammable y est décrit comme un langage pouvant traiter son propre code comme structure de données. En pratique, le langage utilisé est le langage Scheme [SSJ98], qui permet donc de faire le lien entre le métamodèle et le DSL.

L'inconvénient d'une telle méthode est que le langage ainsi développé reste très fortement lié à la façon dont Scheme permet la description de nouvelles fonctionnalités en vue de construire le DSL. Dans les faits, on se retrouve alors avec un langage aux fonctionnalités certes restreintes à la seule gestion de la remontée des données depuis un réseau de capteurs, mais dont la syntaxe n'est pas adaptée à n'importe quel utilisateur. L'idée de s'appuyer sur la structure des données, telles que décrites par le métamodèle, constitue toutefois un des points forts de l'élaboration du langage, étant décrit comme plus rapide et moins fastidieux que de séquentialiser élaboration du langage *puis* implémentation de la sémantique dans tel ou tel langage.

LWiSSy [DRB+13] met précisément en œuvre un développement d'un langage dédié aux réseaux de capteurs sur la base seule d'un métamodèle. En se basant sur l'*Eclipse Modelling Framework* [SBMP08] (EMF), la solution met à disposition des utilisateurs une interface graphique permettant la définition des événements. Le modèle sous-jacent abstrait également les aspects matériels bas-niveau dans l'interface en vue de permettre aux utilisateurs de raisonner plus sur le comportement événementiel des matériels que sur les données elle-mêmes. Sans doute afin de rester accessible à tous, cette solution n'offre toutefois pas de fonctionnalité graphique dédiée à la réutilisation d'événements ainsi décrits. De fait, la composition d'événements y reste très limitée.

Intéressons-nous maintenant aux langage dédiés à la gestion des événements. Si Schultz-Møller *et al.* [SMMP09] proposent une spécification d'événements depuis le flux de données, le langage permettant

la gestion de ces événements reste toutefois un SQL. Il est cependant intéressant de noter ici que la gestion des données porte exclusivement sur des événements et non plus sur les entrées de tables ou de relations. Le langage s'abstrait alors complètement de ces notions et permet de raisonner sur un flux de données, et rien d'autre, dans chacune des clauses SQL. Il en résulte des requêtes plus concises, moins hybrides, que les langages s'appuyant sur la spécification de fenêtres de temps pour faire le lien entre flux de données et modèle relationnel.

Rupeas [WPT10] est un langage dédié à la gestion d'un réseau de capteurs, et s'approche du traitement des événements complexes en introduisant la notion d'événements comme abstraction des données issues du réseau de capteurs. La définition d'un événement reprend celle de STREAM, à savoir un ensemble de clés-valeurs, ou un *tuple* de données. Quatre opérateurs sont définis pour l'analyse des événements. Outre l'opérateur de filtrage des événements, l'opérateur d'agrégation (opération décrite comme *merge*) des événements permet la gestion simultanée de plusieurs événements en vue de la production d'un unique résultat. Toutefois, la distinction entre événements en entrée et résultats laisse à penser que la composition d'événements n'est pas permis par la solution. Le langage ici utilisé pour l'implémentation du DSL est Ruby [FM08], ce pour des raisons similaires aux choix de Scheme par Sadilek. Il en résulte également une syntaxe bien plus adaptée à des développeurs qu'à n'importe quel utilisateur.

UbiQuSE [SCR10], plutôt que de redéfinir de zéro un langage dédié, propose une modélisation des événements sous format XML plutôt que selon le formalisme objet-relationnel. Il est résulte qu'il est alors possible d'adresser les événements au travers des langages dédiés au traitement du formalisme XML. Le point intéressant concerne ici avant tout la modélisation des données, puisque l'arbre permis par la syntaxe de balisage XML permet plus de flexibilité dans le stockage des données qu'un ensemble de couples clé-valeurs. De la même manière, une unique notion d'arbre dans lequel sont stockées les données constitue intuitivement une charge moindre que les concepts de tables, relations et requêtes des SQL. Néanmoins, la description des événements au format XML réserve leur modélisation aux utilisateurs expérimentés.

Esper [Esp15] est un *framework* commercial implémenté en Java qui se propose de soulever les restrictions du modèle objet-relationnel en offrant tout un ensemble d'accès depuis Java aux requêtes et résultats exprimés dans le langage dédié EPL. EPL est lui aussi un SQL, il introduit néanmoins une notion de *pattern* pour la gestion des événements, permettant une description plus poussée de *streams* à partir de relations, comparé à CQL.

La principale problématique d'une telle solution est qu'elle introduit une syntaxe radicalement nouvelle pour l'expression des patterns à la syntaxe SQL, et réclame de fait de bien savoir jongler entre les deux et identifier quelle sous-partie du langage est appropriée à telle ou telle portion d'une requête. Une telle solution est donc clairement réservée à un public de développeurs. Cela est d'autant plus vrai que l'orchestration des requêtes et la gestion de leurs résultats sont pour partie déléguées à Java, ce qui introduit un nouveau langage dans le traitement des événements et réclame de l'utilisateur une maîtrise non négligeable de chacune des trois syntaxes. L'appel à une requête pourra alors ressembler à :

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
String expression =
        "SELECT a.id, count(*) FROM pattern [" +
        "        every a=Status -> (timer:interval(10 sec) and not Status(id=a.id)" +
        "] GROUP BY id";
EPStatement statement = epService.getEPAdministrator().createEPL(expression);
```

On voit bien ici la requête EPL exprimée dans la variable *expression* en Java, et la description d'un *pattern* introduite entre crochets au beau milieu de la syntaxe de type SQL. Qui plus est, cette partie concerne uniquement la création de la requête. Son traitement par le *framework* doit faire à son tour l'objet d'une implémentation Java spécifique, qui seule sera alors à même de permettre une réinjection du résultat de la requête dans le flux des événements.

COPAL [LSD10] se présente comme une surcouche à Esper pour proposer un unique langage dédié en vue de simplifier tout le processus qu'on vient de décrire. Dans sa définition des opérations sur les

événements, il regroupe une partie des fonctionnalités vues auparavant, mais permet également la génération de deux flux de données à partir d'un seul flux en entrée par un mécanisme dit de différentiation. Le langage en tant que tel est quant à lui déclaratif, avec une description séparée du format des événements, de leur détection et du traitement à effectuer. Ces déclarations permettent à des *éditeurs* de publier des contextes décrivant les événements, sur lesquels des *auditeurs* vont pouvoir exprimer des requêtes, se référant aux attributs décrits dans les contextes. En pratique, un éditeur devra être décrit *pour chaque* capteur fournissant des données. Enfin, des *processeurs* sont également décrits pour permettre la description d'*actions*, et non seulement d'attribut, dans les contextes événementiels. La description de ces actions permet, dans l'exemple donné, l'aggrégation d'un attribut du même contexte. Il résulte de cet ensemble de déclarations, même pour le cas simple qui illustre l'article, un ensemble de déclarations très verbeux. Dans l'exemple du suivi énergétique des serveurs d'un centre de données, voici une description possible des différents composants :

```
Publisher Server1 {
        contextType = Power
}
Publisher Server2 {
        contextType = Power
}
Publisher Server3 [...]
ContextType Power {
        attribute = value: Float !
        defaultAction = powerSum
}
Listener listener1 {
        query = q1
}
Query q1 {
        contextType = Power
        criteria = "sourceID='powerSum' and unit='Wh'"
}
```

3.2 Identification des données

Comme on a pu le voir, nombre des solutions de traitement des événements complexes étudiées proposent des langages dédiés complexes à appréhender, soit du fait de requêtes trop verbeuses, soit parce qu'ils mêlent de nouveaux concepts à des langages existants. Dans un contexte où des utilisateurs aux profils variés, n'ayant pas forcément l'expertise informatique nécessaire à la prise en main de ces langages et à la maîtrise des paradigmes qui les sous-tendent, ces mêmes langages, comme seules interfaces au traitement des événements complexes, posent problème. Une autre source de ce problème tient au fait que dans la majeure partie des cas, le modèle sous-jacent se concentre sur le traitement des événements complexes au détriment d'une modélisation compréhensible du réseau de capteurs. En effet, une telle modélisation constitue un premier pas dans une démarche d'adaptation du traitement des données aux besoins et usages des utilisateurs.

À ce sujet, Hadim *et al.* [HM06] proposent entre autres choses la gestion du *lieu* où sont situés les capteurs et actuateurs. Prenant pour exemple un capteur de température et partant du principe qu'un utilisateur sera plus volontiers amené à vouloir connaître la température d'un lieu plutôt qu'une valeur en degrés celsius remonté par un capteur, la modélisation du réseau de capteurs se doit de refléter cette notion afin de se rendre accessible à l'utilisateur. D'une manière plus générale, il s'agit ici de situer le capteur et les données qu'il remonte dans un existant, connu et maîtrisé par l'utilisateur concerné. La mise à disposition de cet existant dans l'interface du traitement des données permettra alors un usage plus adapté aux besoins de l'utilisateur. Cette considération est au cœur même des deux pistes abordées dans les sous-sections suivantes : la contextualisation des données, capteurs et actuateurs d'une part et l'enrichissement sémantique des données d'autre part.

3.2.1 Contextualisation

Concernant la contextualisation des actuateurs, des capteurs et de leurs données, Perera *et al.* [PZCG14] étudient plusieurs publications permettant la gestion des contextes de l'Internet des objets. Nous nous intéresserons ici aux solutions permettant à la fois la détection des événements et le traitement en temps réel des données, ce qui en fait les solutions les plus proches du traitement des événements complexes.

SCONSTREAM [KSKL10] introduit ainsi la notion de contexte *spatial*, considérant des capteurs mobiles mesurant et indiquant, entre autres, la position spatiale d'objets dans l'environnement. La nature de ces mesures prend place dans un *flux spatial des données*, définissant pour chaque capteur une donnée comme le couple d'une position et d'un instant t. Dès lors, il s'agit pour SCONSTREAM de regrouper l'ensemble de ces flux de données dans un *flux spatial des contextes*. Partant du même constat fait par Hadim *et al.* concernant la nécessité d'identifier des comportements de l'environnement plutôt que des mesures de capteurs, il s'agit ici, sur la base de prédicats dits spatiaux, de regrouper des objets équipés de capteurs au sein d'un même contexte environnemental. La définition des contextes à partir des flux de données est donc ici fixée par la transformation de ces flux en des *flux de contextes*. Il s'agit par contre bien d'un *flux* des contextes, puisque les objets vérifiant ou non le prédicat évolueront en même temps qu'ils se déplaceront dans l'environnement. En pratique, six opérateurs permettant l'expression des prédicats sont spécifiés. Ils concernent tous l'interaction des objets avec des *salles*. Il s'agira alors d'identifier la position — à l'intérieur, à l'extérieur ou à une distance maximale de la salle — ou le déplacement — entrée dans, sortie de ou station dans la salle — des objets concernés. L'expression des requêtes *via* Oracle Spatial [MAA+02] permet également la gestion d'un historique des données. En revanche, la gestion en temps réel des données n'est pas dynamique et se base en réalité sur une récupération périodique depuis la base de données sous-jacente, plutôt que de réagir directement à l'inscription de nouvelles données depuis les capteurs.

En s'intéressant à un système de navigation sur *smartphone* pour les piétons dans un environnement urbain, EventCJ [KAM11] propose une gestion de capteurs multiples pour la localisation d'une même personne en définissant deux contextes intérieur et extérieur. Les deux systèmes de localisation intervenant dans le système sont le GPS et la localisation par WiFi. Partant du principe que le GPS ne fonctionne qu'en extérieur, du fait des interférences provoquées par le bâti en intérieur, et que la localisation par WiFi ne fonctionne qu'en intérieur du fait d'une portée limitée, il convient alors d'utiliser l'un ou l'autre pour identifier un piéton selon qu'il soit en intérieur ou en extérieur. Le contexte impacte également la façon dont le système de navigation indique à un utilisateur sa position. Selon qu'il soit dehors ou dans un bâtiment, il s'agira de le situer dans un plan des rues ou dans un plan de masse du bâtiment. Cette aspect fait l'objet d'une notion *couche* correspondant au plan à afficher. En pratique EventCJ propose une séparation entre l'activation de ces couches et l'évolution des contextes selon les déplacements des piétons. D'une part un langage déclaratif permet de définir les processus d'activation de désactivation d'une couche. Ce sont ces processus qui permettront de passer d'un positionnement GPS à la localisation par WiFi et inversement. D'autre part, ce langage permet la déclaration d'événements traduisant le déplacement des piétons au sein des bâtiments. Des règles de transitions permettent alors de définir quelles couches activer et désactiver en fonction des événements identifiés depuis les déplacements des piétons. À l'instar de COPAL, l'aspect déclaratif du langage, permettant la déclaration des couches, événements et règles de transition, le rend toutefois assez verbeux, et impose de bien comprendre chaque élément à déclarer et la façon dont ces éléments interagissent dans chacun des trois concepts.

De la même manière, UbiQuSE [SCR10] s'intéresse à la possibilité de permettre aux systèmes et applications de situer les individus *via* une notion de contexte. La gestion des données et de leurs contextes s'appuient sur deux composants distincts que sont une base de données et une base des *métadonnées*, venant définir les contextes liées aux données. La définition d'un événement au format XML s'appuie alors sur ces deux bases afin d'établir les attributs constituant l'arborescence de l'événement.

Habitation [JRS+09] propose quant à lui un outil graphique destiné à un utilisateur expert en domotique, permettant la description des services offerts par un ensemble de capteurs et d'actuateurs au sein d'un

catalogue d'unités fonctionnelles. Tout l'intérêt de ce catalogue est alors de proposer une contextualisation des services en fonction du matériel qu'équipent les capteurs et actuateurs sous-jacents, voire de leur environnement immédiat. Ce catalogage des services est néanmoins présenté comme visant exclusivement des développeurs, qui auront alors la possiblité d'orchestrer les services offerts par les unités du catalogue dans un cadre applicatif concret.

Parmi les solutions basées sur Esper, COPAL [LSD10] considère des capteurs plus statiques au sein de leur environnement, tels que peut en proposer la domotique par exemple. Nous avons vu précédemment que le langage de COPAL s'appuie sur une déclaration distincte des événements, de leur détection et de leur gestion. En pratique, la détection s'adosse directement à une notion de contexte localisant les mesures dans les pièces d'une habitation, pour poursuivre l'exemple de la domotique. La modélisation des capteurs au sein des contextes permet alors l'expression d'événements dits *contextuels*, ce qui permet d'abstraire les capteurs pour décrire les événements au sein des pièces. Cette possibilité de raisonner sur les événements en terme de contextes au sein desquels s'inscrivent les données permet une appréhension très intuitive, car basée sur les pièces dans lesquelles on vit tous les jours, de ces événements.

Sans revoir la syntaxe d'Esper, WildCAT [DL05] propose un modèle des données regroupant des attributs, définis comme des couples clé-valeur dont la valeur évolue selon le flux de données, comme feuilles de structures hiérarchiques constituant les contextes. Il s'agira ici de lister les métriques de l'objet auquel sont attachés des capteurs et de situer cet objet dans un environnement pouvant être décrit selon une organisation hiérarchique. En pratique, la description de cette organisation passe par l'expression des attributs et objets, ou ressources, par des URI (*Unique Resource Identifiers* [MBLF05]). Un exemple donné concerne l'organisation d'un ordinateur, pour lequelle plusieurs attributs et ressources sont décrits :

```
sys://storage/memory                    // Une ressource
sys://storage/disks/hda#removable       // Un attribut
sys://devices/input/*                   // Ressources d'entree
sys://devices/input/mouse#*             // Attributs de la souris
```

Cette rationalisation des contextes sous forme hiérarchique est un point fort indéniable de cette solution. Cependant le principal inconvénient de WildCAT est qu'il ajoute une nouvelle syntaxe d'expression des URI à la complexité pré-existante de la gestion des événements d'Esper. La requête simple suivante, qui s'inspire d'un exemple du guide utilisateur de WildCAT [OW216], reprend l'exemple de l'électricien qui souhaite connaître la consommation électrique des serveur d'un centre de données :

```
QueryAttribute qa = new QueryAttribute("select cast(value.load,int)
        from WAttributeEvent(source='self://dc1/servers#power').win:time(5 sec)");
```

Si l'URI permet de s'abstraire de la nécessité d'une clause *where*, on constate en revanche, sur deux lignes seulement, le mélange des syntaxes de Java, de la base SQL d'EPL, de l'expression d'une fenêtre d'EPL, et d'un URI introduit par WildCAT.

3.2.2 Enrichissement sémantique

Une autre piste permettant l'identification des données issues d'un réseau de capteurs concerne l'enrichissement sémantique des données. Ainsi que le décrivent Huang *et al.* [HJ08], cet enrichissement sémantique repose sur la notion d'*ontologie*. Le W3C définit les ontologies dans le contexte d'un langage qui leur est dédié : OWL [W3C09] (sigle réarrangé pour *Web Ontology Language*). Appliquée aux réseaux de capteurs, une ontologie se définit comme une spécification formelle et explicite de la conceptualisation des capteurs. Il s'agit ici d'établir une modélisation formelle stricte dans laquelle les capteurs et leurs données s'inscrivent. Cette inscription permet alors à OWL de s'appuyer sur cette stricte modélisation pour interagir avec le réseau de capteurs et en traiter les données. L'enrichissement sémantique proposé par une ontologie n'est cependant contraint que dans la modélisation sous-jacente qui se doit de respecter un formalisme objet. La façon d'enrichir les données, qu'il s'agisse de contexte, de hiérarchie de types entre les capteurs, de relations spatiales ou temporelle, *etc.*, est laissée libre lors de la conception de l'ontologie selon le formalisme OWL.

En pratique, l'attachement d'une ontologie à un réseau de capteurs s'appuie sur la notion de *base de connaissances*, ou *knowledge base* (KB), introduite par Teymourian *et al.* [TP10]. La base de connaissance est alors en charge de fournir les connaissances de fond permettant la détection des événements selon la façon dont l'enrichissement sémantique les a décrits, qu'il s'agisse, pour ce qui nous intéresse, de hiérarchie de types ou de relations avec d'autres objets du domaine d'application concerné. L'avantage de l'utilisation d'une base de connaissances dans le traitement des événements complexes est décrit comme permettant une plus grande expressivité, ce qui rejoint la motivation de l'identification des données, mais également une plus grande flexibilité. La flexibilité porte alors sur la possibilité d'intégrer plus aisément des changements dans le fonctionnement métier du système en se basant sur l'abstraction proposé par les ontologies plutôt que de revoir dans le détail les règles sous-jacentes de traitement des événements.

L'adressage de la base de connaissances se fait via le langage SPARQL [W3C13], également décrit par le W3C dans le cadre des ontologies. Cependant, la gestion des événements fait quant à elle intervenir une collection d'opérateurs de composition algébriques, ce qui mène, dans l'expression des requêtes, à une sémantique hybride, mêlant la composition des événements selon ces opérateurs à la syntaxe SPARQL permettant l'enrichissement sémantique depuis la base de connaissances.

Une réflexion intéressante est menée sur la manière de collecter les métadonnées sémantiques depuis la base de connaissances. Le principe d'une récupération périodique des métadonnées est alors remis en cause dans le cadre du traitement en temps réel des données, notamment pour les applications pour lesquels une réaction immédiate au flux de données est jugé nécessaire, comme par exemple la détection de fraude.

Taylor *et al.* [TL11] proposent quant à eux une surcouche d'Esper permettant l'enrichissement sémantique dans le traitement des événements complexes par la définition d'une ontologie des événements. L'accent est ici porté sur la nécessité d'intégrer la spécification des événements dans le domaine d'expertise de l'utilisateur final. Les ontologies sont alors utilisées afin de permettre la spécification et la détection d'événements portant sur la corrélation de données issues de différents capteurs. Ces événements sont alors appelés des événements *sémantiques*. La définition de ces événements se fait *via* une interface graphique pour OWL, *Protégé* [KFNM04]. En se basant sur la formalisation de l'ontologie, une étape de transformation des événements sémantiques en requêtes EPL est alors effectuée.

Dans le registre de l'enrichissement sémantique d'un flux de données, Textor *et al.* [TMT+12] définissent un ensemble d'opérateurs en vue de l'analyse des données. Ces opérateurs, en raisonnant sur la sémantique attachée aux données, permettent de concentrer cette analyse sur ce que l'ontologie a établi autour des données. En détails, l'ontologie construit à partir du flux de données un *réseau* d'exploration du flux (*data stream mining network*) qui, comme son nom l'indique, propose une organisation en réseau des données. En plus de quelques opérateurs d'analyse statistique et numérique des données, des opérateurs dits *structurels* sont introduits et s'appuient sur ce réseau pour proposer des filtrages et jointures de données en fonction de leur emplacement dans le réseau. Le modèle de ce réseau est décrit par un DSL, nommé mDSL, sur la base d'un métamodèle des réseaux d'exploration de flux de données.

3.3 Synthèse

Comme nous venons de le voir, plusieurs travaux ont été mené en rapport avec les problématiques auxquelles nous nous attelons dans ce document. Que ce soit pour le traitement des événements complexes, pour la gestion des réseaux de capteurs ou les deux, la nécessité de proposer une interface plus adaptée aux utilisateurs finaux a été au cœur des réflexions de nombreux de ces travaux. Le tableau 3.1 tire les grande lignes de cette étude pour proposer un récapitulatif des fonctionnalités adressées par ces travaux.

Nous pouvons rapidement voir que dans l'objectif de proposer un unique langage dédié au traitement des événements complexes dans les réseaux de capteurs, l'enrichissement sémantique par ontologies ne répond pas aux exigences. Le fait qu'il s'appuie sur la définition stricte des ontologies proposée par le W3C implique que la majorité de ces solutions utilisent les langages dédiées à ces ontologies et aux bases

TABLE 3.1 – Comparaison entre les différentes solutions

Travaux	**Réseaux de capteurs**	**Traitement des événements complexes**	**Langage dédié**	**Contextualisation**	**Enrichissement sémantique**
CQL [ABW03]		✓	✓		
CEPEMS [SS13]	✓	✓			
EQL [SMMP09]		✓	✓		
Sadilek [Sad07]	✓		✓		
LWiSSy [DRB+13]	✓		∼ interface graphique		
Rupeas [WPT10]	✓	∼ pas de composition	✓		
UbiQuSE [SCR10]	✓	✓	✓	✓	
Esper [Esp15]		✓	✓		
COPAL [LSD10]	✓	Esper	✓	✓	
SCONSTREAM [KSKL10]	✓	∼ pas en temps réel		✓	
EventCJ [KAM11]	✓	✓	✓	✓	
WildCAT [DL05]		Esper	✓	✓	
Teymourian *et al.* [TP10]		✓	OWL & SPARQL		✓
Taylor *et al.* [TL11]	✓	✓	OWL & SPARQL		✓
Textor *et al.* [TMT+12]		∼ pas de composition	∼ métamodèle		✓

de connaissances associées, en l'occurrence OWL et SPARQL, qui ne permettent pas le traitement des événements complexes.

Dès lors, la contextualisation nous semble une piste plus à même de répondre à nos attentes. Plus précisément, on retrouve trois solutions permettant la contextualisation des réseaux de capteurs dans le cadre du traitement des événements complexes, à savoir UbiQuSE, EventCJ et COPAL. Comme on a pu le voir, ce dernier délègue à Esper la gestion du traitement des événements. Or EPL, le langage dédié d'Esper, s'avère beaucoup trop complexe pour des utilisateurs finaux. En ce qui concerne UbiQuSE, plutôt que de proposer un langage dédié, il s'appuie sur sa gestion des événements au format XML pour permettre le requêtage *via* XQuery. Si ce choix permet la modélisation des événements dans des structures d'arbre pouvant s'inscrire aisément dans un contexte, la contrainte d'une seule arborescence par événement risque de s'avérer trop limitante dans l'objectif de proposer une interface multi-utilisateurs. Enfin, EventCJ établit une syntaxe très déclarative, tant pour la définition des événements que leur orchestration, ce qui rend l'ensemble très verbeux. En outre le domaine d'application de cette solution, dédié aux capteurs mobiles, qui peuvent équiper un *smartphone* par exemple, constitue un cas limite des réseaux de capteurs, et s'éloigne de fait des problématiques adressées dans ce document.

La nécessité d'apporter à chacun des utilisateurs une expérience la plus simple possible du réseau de capteurs, au travers du traitement des événements complexes, nous a mené à l'idée selon laquelle il est alors nécessaire d'inscrire les capteurs dans autant de contextes qu'il y a d'utilisateurs, ou à tout le moins de profils d'utilisateurs, de rôles, différents. Dès lors qu'aucune des solutions étudiées ici ne permet de base une telle solution, et que les langages dédiés ne semblent pas adaptés à n'importe quel utilisateur, nous avons décider de proposer un nouvel outil, SensorScript, joignant une modélisation des contextes utilisateurs à un langage dédié que nous souhaitons simple à appréhender. La conception de cet outil s'appuie toutefois sur des concepts mis en lumière dans cette étude, qu'il s'agisse de la modélisation des contextes sous forme d'arbres, de structures hiérarchiques, telles que proposées par UbiQuSE ou WildCAT ou des différents opérateurs de composition d'événements évoqués dans l'étude du traitement des événements complexes et des langages dédiés.

Nous avons également pu voir que des travaux ont été menés dans la distribution du traitement des événements complexes au sein des capteurs même. Toutefois, aucune de ces solutions n'adresse la problématique de l'identification des données. En se plaçant dans un cas où le traitement des événements est éparpillé parmi les capteurs, aucun des nœuds du réseau n'a une connaissance globale du réseau et des données associées. Dès lors, une modélisation de ceux-ci, que ce soit dans un contexte ou dans le cadre d'une ontologie, devient une problématique bien plus complexe encore. C'est la raison pour laquelle nous nous concentrons dans un premier temps sur l'identification des capteurs et des données dans un contexte de récupération centralisée des données.

Modélisation et conception

Introduction

Cette partie porte sur la modélisation et la conception de SensorScript. L'introduction qui suit vient rappeler les enjeux et problématiques qui sous-tendent les travaux ayant mené à cette solution.

Le chapitre suivant se propose de présenter plus en détails la contextualisation des données issues d'un réseau de capteurs à la base de la modélisation de SensorScript. Enfin, l'on s'attardera sur la manière dont le traitement des événements complexes s'inscrit dans cette modélisation et comment il se retranscrit dans le langage qui lui est dédié.

Rappel des enjeux

SensorScript est un *framework* de traitement des événements complexes dédiés aux réseaux de capteurs sans fil. Dans la perspective de le voir adaptable à tout réseau de capteurs sans fil, l'objectif premier était, comme pour tout outil, de favoriser la facilité d'utilisation. Dans ce contexte, la problématique majeure dans la collecte et le traitement des données issues d'un réseau de capteurs concerne l'identification des données. Ce qu'on entend par identification des données, c'est apporter à l'utilisateur, pour chacune des données, la réponse à trois questions :

1. qu'est-ce qui est mesuré ? quelle que soit le type de la valeur remontée (nombre, booléen, chaîne de caractères, etc.), la signification reste vide de sens si on est incapable de connaître la nature de ce qui est mesuré (température ou consommation en watts ? détection de fumée ou ouverture d'un système électrique ? état d'une machine virtuelle ou nom d'une personne ? etc.) ;

2. où est-ce mesuré ? au sein d'un même réseau de capteurs, il est quasi-systématique de retrouver un même capteur en plusieurs exemplaires, dès lors il est crucial de savoir localiser une donnée, c'est à dire la lier à un capteur physique ;

3. quand est-ce mesuré ? connaître l'instant où une mesure a été effectuée est crucial dans l'analyse de données, la concordance ou non de certaines mesures correspondant à des comportements attendus ou non des phénomènes mesurés.

Le traitement des événements complexes permet, dans une certaine mesure, de se soustraire à la troisième question. À moins d'être confronté à une grande latence entre les capteurs et le point de collecte des données, aussi appelé *nœud-puits*, ou d'avoir besoin d'une précision sous la seconde, la gestion du flux de remontée des données en temps réel permet de considérer qu'une donnée a été mesurée à l'instant où elle est connue. Pour les cas extrêmes suscités, la mesure de la latence, pour chaque capteur, entre celui-ci et le nœud-puits est alors nécessaire. Cette problématique n'est toutefois pas l'objet de ce document.

Solution proposée

Restent donc les deux premières questions. Comme on l'a vu en section 3.2 de l'état de l'art, deux pistes ont été suivies pour répondre à chacune de ces deux questions. La piste mise en œuvre dans SensorScript est

celle de la contextualisation des données. De fait, l'accent est mis sur la localisation de ce qui est mesuré, puisque la contextualisation permet l'inscription de la donnée dans un existant, que celui-ci corresponde à une réalité physique ou une représentation plus abstraite, comme par exemple l'organisation en services d'un organisme. La modélisation proposée se consacre à répondre au cas par cas aux besoins de chacun des utilisateurs du réseau de capteurs. Elle fait l'objet du chapitre 4.

Le chapitre 5 présente quant à lui la manière dont le traitement des événements complexes s'inscrit dans le modèle de SensorScript. Il est alors mis en exergue la façon dont il tire partie de la modélisation en multiarbre du réseau de capteurs pour simplifier l'expérience utilisateur. Cette expérience se fait alors au travers d'un langage dédié présenté en détails.

Existant

Modélisation

La modélisation utilisée dans SensorScript repose historiquement sur btrScript [PM12], un *framework* générique d'introspection et d'intercession sur les centres de données. Il propose de centraliser la gestion d'un centre de données mettant en œuvre différents outils logiciels de supervision de centres de données virtualisés (appelés *hyperviseurs*) en y ajoutant une dimension énergétique avec la possibilité d'intégrer la communication avec des *cartes de management* (capteurs énergétiques équipant les serveurs). Pour ce faire, il propose une modélisation du centre de données en deux arbres statiques, telle que l'illustre la figure 3.1, l'un représentant l'organisation physique du centre, avec les serveurs comme feuilles, l'autre l'organisation logique des machines virtuelles (VM pour Virtual Machines) hébergées par ces serveurs, les VM étant alors les feuilles.

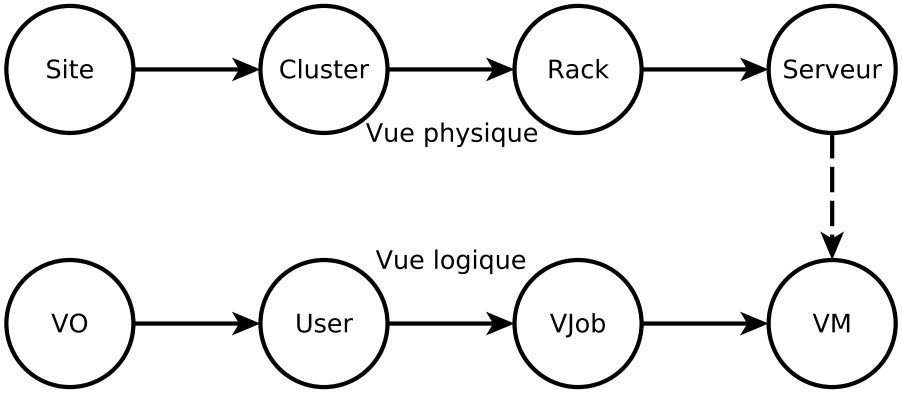


FIGURE 3.1 – Modèle de btrScript

Le point crucial de cette modélisation réside en ce que le lien entre machines virtuelles et serveurs existe bien dans la modélisation, ce qui implique que les deux arbres sont liés l'un à l'autre au niveau de leur feuilles. Qui plus est, du fait que l'un des objectifs de btrScript est de permettre et de suivre les migrations de VM d'un serveur à un autre, ces liens sont nécessairement dynamiques. Comme nous le verrons par la suite, la modélisation en multiarbre trouve son inspiration dans la modélisation de btrScript.

Langage

Outre la modélisation en deux arbres statiques, btrScript permet également le suivi du parc (principalement des métriques systèmes sur les serveurs et les VM) et la gestion des VM (création, démarrage, extinction, migration, etc.) au travers d'un langage dédié. Du fait de la structure en deux arbres distincts du modèle de centre de données sur lequel s'appuie btrScript, ce langage s'inspire d'XPath [CD+99] et permet la sélection de nœuds de l'arbre selon des règles de filtrage s'appuyant sur un parcours implicite d'un niveau

à l'autre au sein d'un même arbre. Qui plus est, ce parcours peut se faire également entre les deux arbres, en considérant alors les VM comme des nœuds fils des serveurs (du fait qu'un même serveur peut héberger plusieurs VM, et qu'une VM en fonctionnement n'est hébergée que par un serveur).

Prenons un exemple : considérons que l'arbre physique se présente de telle sorte que les serveurs sont regroupés au sein de grappes de serveurs (ou *clusters*). Par conséquent le type *cluster* se trouve être le père du type *serveur* dans l'arbre physique. Supposons que l'on souhaite accéder aux VM du cluster CL1, voici la requête correspondante :

```
/CL1/vm
```

On constate ici qu'il est inutile de faire la moindre mention des serveurs qui hébergent les dites VM, pourtant bien présents au sein de CL1. Le langage s'appuie en effet sur la structure d'arbre pour restreindre les serveurs à parcourir comme étant ceux de CL1, et ce faisant ne lister les VM qui ne sont hébergées que par ces serveurs, et aucun autre. De fait, cette requête met également en avant le lien existant entre les deux arbres qui constituent le modèle.

Si les contraintes exercées sur l'interface de SensorScript, tant liées au modèle qu'au traitement des événements complexes, diffèrent de celles de btrScript, tirer partie du modèle pour réduire la taille des expressions du langage en permettant un parcours implicite en tâche de fond constitue également un point fondamental du langage dédié développé pour SensorScript.

4

Contextualisation multi-utilisateurs d'un réseau de capteurs

Sommaire

4.1 Problématique

L'objectif premier de la modélisation du réseau de capteurs est de permettre l'identification sans équivoque, par les utilisateurs, des capteurs et actuateurs, de leurs données et fonctionnalités. Comme on l'a vu, le nombre et la variété des capteurs entraîne des usages tout aussi variés, et de fait implique des utilisateurs aux profils, aux domaines de compétences différents, en un mot aux *connaissances* différentes. De fait, l'identification d'un même capteur ne signifie pas la même chose pour deux acteurs différents.

Pour un responsable de centre de données, un wattmètre permet la mesure énergétique d'un switch, d'un serveur, peut-être d'une armoire de serveurs ou du système de climatisation, et lui donnera une idée précise de la consommation électrique de son parc. Pour un électricien, ce même wattmètre sera branché à une prise, elle-même reliée à un compteur et un disjoncteur précis, et lui permettra d'estimer la santé du système électrique du lieu. Même sur cet exemple simple, avec un type de capteur pour deux acteurs, on constate de suite qu'un même capteur peut s'inscrire de manière complètement différente dans des modèles différents, selon qu'on privilégie un domaine de compétences ou un autre.

Multiplions les capteurs, ajoutons des actuateurs pour arriver à un cas plus proche de ce qu'on peut trouver aujourd'hui dans le monde de l'entreprise par exemple : quelques centaines d'appareils contrôlant la consommation électrique, la température, permettant une veille incendie, la gestion de la climatisation, des badgeuses pour les accès restreints. Multiplions les profils d'utilisateurs en conséquence et de fait le nombre d'acteurs en jeu. Il devient évident que trouver une modélisation qui réponde à chacun de ces profils n'est pas chose aisée, et que cela se fait au risque d'imposer à chaque acteur de maîtriser des connaissances qu'il n'a pas, dès lors que les domaines de compétences se retrouvent au sein d'un même modèle.

C'est pourtant la problématique qu'adresse la modélisation de SensorScript. Pour ce faire nous établissons d'emblée un concept fondamental à notre solution à cette problématique.

4.1.1 Contextes utilisateurs

Une partie du problème est donc d'assurer à chacun des utilisateurs un usage reflétant ses connaissances vis-à-vis des capteurs et actuateurs. Ce qui sert à l'utilisateur, parmi ces connaissances, à identifier un capteur, un actuateur, ce sont des choses qui vont lui permettre de situer l'appareil dans un ou plusieurs *contextes* s'inscrivant dans son domaine de compétence. Il s'agira ici de savoir pour le service informatique à quel serveur ou switch est lié tel wattmètre, pour le service électrique de savoir quel compteur, quel disjoncteur se cache derrière la prise où est branché ce même wattmètre.

Naturellement, il convient de garder à l'esprit qu'une distinction existe entre ces contextes et les domaines de compétence de chacun des utilisateurs. Un profil d'utilisateur pourra maîtriser plusieurs contextes, de même qu'un même contexte pourra être maîtrisé, en tout ou partie, par différents profils utilisateurs.

Nous tenons avec cette notion de contexte un élément à même de permettre d'inscrire les capteurs et actuateurs dans une modélisation, non pour chacun des utilisateurs, mais pour chaque contexte, leur donnant un outil qui s'inscrit dans un *usage* correspondant à leur domaine de compétence. L'objectif est alors de permettre tous ces usages au sein d'un même modèle, c'est à dire de regrouper tous les contextes sous-jacents dans une même modélisation. Dans cette optique, une contrainte forte que nous posons d'emblée concernant la modélisation des contextes est que chacun d'eux sera représenté par un arbre. Nous mettrons en avant dans la section suivante quelles en sont les raisons.

4.2 Le multiarbre

Nous venons de voir que la rationalisation des besoins utilisateurs en contextes d'utilisation permet, pour chaque contexte, l'inscription des capteurs dans un environnement relatif à l'un des domaines de compétence du profil d'utilisateurs concerné. Afin d'assurer une modélisation à même de répondre aux besoins de chacun des utilisateurs, nous considérons que celle-ci doit intégrer chacun des contextes relatifs à ces besoins. En considérant que chacun des cas d'utilisation, correspondant à un usage d'un des acteurs, n'adresse

qu'une sous-partie de l'ensemble des contextes, il est en outre primordial d'assurer que ces seuls contextes, et aucun autre, entrent en jeu dans l'expérience du dit cas d'utilisation. La solution que nous avons mise en place pour répondre à ces contraintes est le multiarbre.

4.2.1 Définitions

Le multiarbre [FZ94] est un cas particulier des graphes orientés acycliques.

Graphe orienté acyclique

Un graphe orienté acyclique est un graphe au sein duquel tous les liens entre deux nœuds sont *orientés*, et dont les orientations assurent que le parcours du graphe ne boucle jamais, d'où le terme *acyclique*. Il est important de noter que la seule orientation des liens suffit à assurer sa non-cyclicité. De fait, si l'on fait abstraction de l'orientation des liens, il est tout à fait possible de voir des boucles dans le graphe. C'est cependant l'orientation des liens qui permet d'éliminer toute possibilité de cycle dans un parcours au sein des nœuds. Pour lever toute ambiguïté, on parlera dans la suite de *structure fermée* plutôt que de boucle. Ainsi, le graphe 4.1 est bel et bien un graphe orienté acyclique, puisque, quel que soit le nœud duquel on parte, il est impossible, en suivant l'orientation indiquée par les flèches sur les liens, de revenir sur ce même nœud.

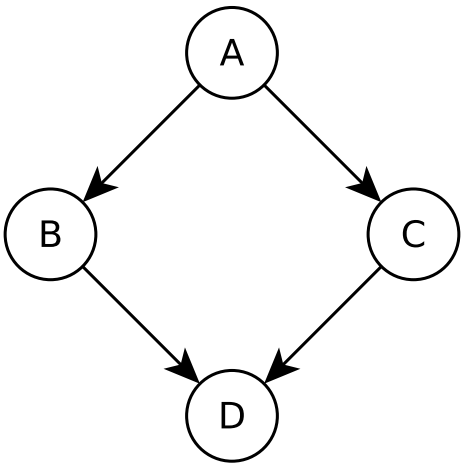


FIGURE 4.1 – Graphe orienté acyclique

Multiarbre

Le multiarbre est donc un graphe orienté acyclique dont la particularité consiste en ce que, pour chacune de ses structure fermées, il est possible de définir une hiérarchie pour chaque couple de nœud de la structure.

Par exemple, le graphe 4.1 n'est pas un multiarbre ; l'orientation permet trivialement de trouver les relations hiérarchique suivantes :

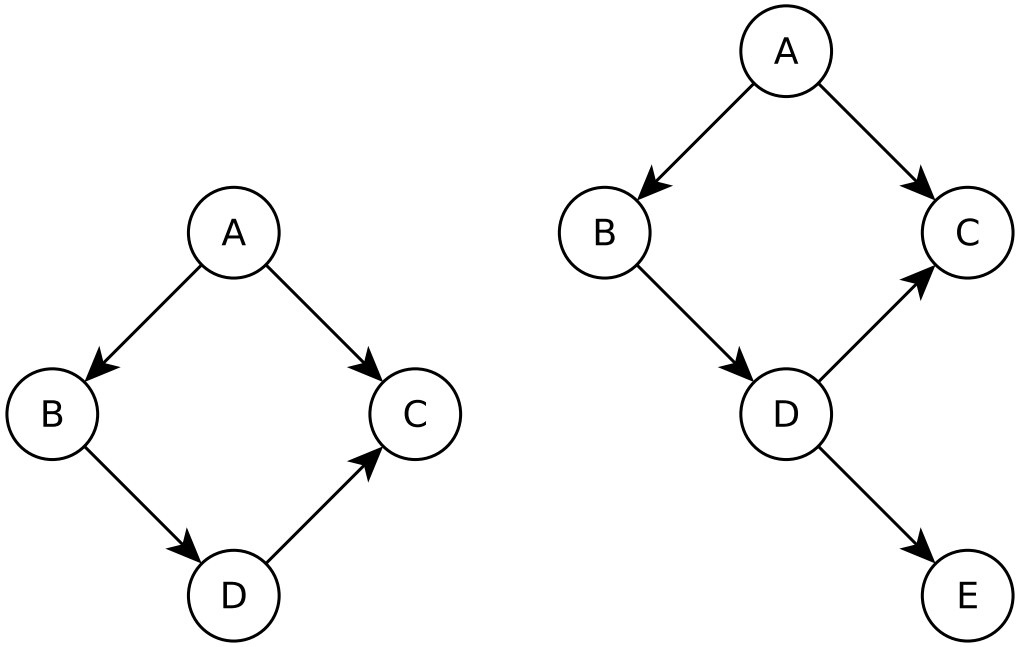
(1) $A > B$
(2) $B > D$
(3) $A > C$
(4) $C > D$

Par transitivité — sur les relations *(1)* et *(2)* ou sur les relations *(3)* et *(4)* — on trouve également la relation $A > D$, mais il est impossible de définir une quelconque hiérarchie entre les nœuds B et C.

Au contraire, le graphe 4.2(a) est bel et bien un multiarbre. Les relations triviales sont cette fois :

(1) $A > B$
(2) $B > D$
(3) $D > C$
(4) $A > C$

Par transitivité sur *(1)* et *(2)* on trouve $A > D$, et par transitivité sur *(2)* et *(3)* on trouve $B > C$; on a bien une relation hiérarchique pour chacun des six couples possibles parmi les nœuds du graphe.



(a) (b)

FIGURE 4.2 – Multiarbres

Une autre manière d'observer un tel graphe, qui donne tout son sens au nom de multiarbre, est de considérer ce modèle comme la conjonction de, par exemple, deux modèles d'arbre $A \to B \to D \to C$ et $A \to C$. Cette obligation de hiérarchie entre deux nœuds assure l'existence d'un unique plus court chemin entre eux.

Notons enfin que celle-ci n'existe qu'au sein des structures fermées. Ainsi, le graphe 4.2(b) est également un multiarbre. Bien qu'on qu'on ne puisse définir de relation entre les nœuds C et E, cela n'a pas d'importance puisque ces nœuds ne font pas partie d'une même structure fermée.

4.2.2 Contextes et multiarbre

La façon dont nous mettons à profit le multiarbre dans la modélisation des contextes d'utilisation d'un réseau de capteurs revient à tirer parti de la conception d'un multiarbre comme la conjonction de plusieurs modèles d'arbre. En pratique, à chacun des contextes à modéliser correspond un des arbres composant la modélisation en multiarbre.

Exemple

Nous nous proposons ici d'étoffer l'exemple d'un centre de données virtualisé pour lequel différents acteurs sont confrontés à un même type de capteurs. Considérons alors quatre acteurs :

- l'*électricien*, qui possède la vision la plus externe au centre de données, responsable de l'infrastructure énergétique du centre de données ;
- le *responsable réseau* en charge de la mise en place et de la maintenance des communications au sein et en dehors du centre de données ;
- le *responsable SI* (Service Informatique) en charge de la couche applicative des services fournis par le centre de données ;
- le *responsable développement durable*, qui a comme mission de réduire les pertes énergétiques du centre de données.

Chacun de ces acteurs sera amené à interagir avec divers capteurs et actuateurs. Les capteurs permettent la mesure de l'énergie consommée par les appareils équipant le centre de données, de la température, de l'hygrométrie et de la luminosité. Les actuateurs, quant à eux, prennent en charge l'extinction et l'allumage des appareils, la gestion de la climatisation, de l'éclairage et des volets électriques.

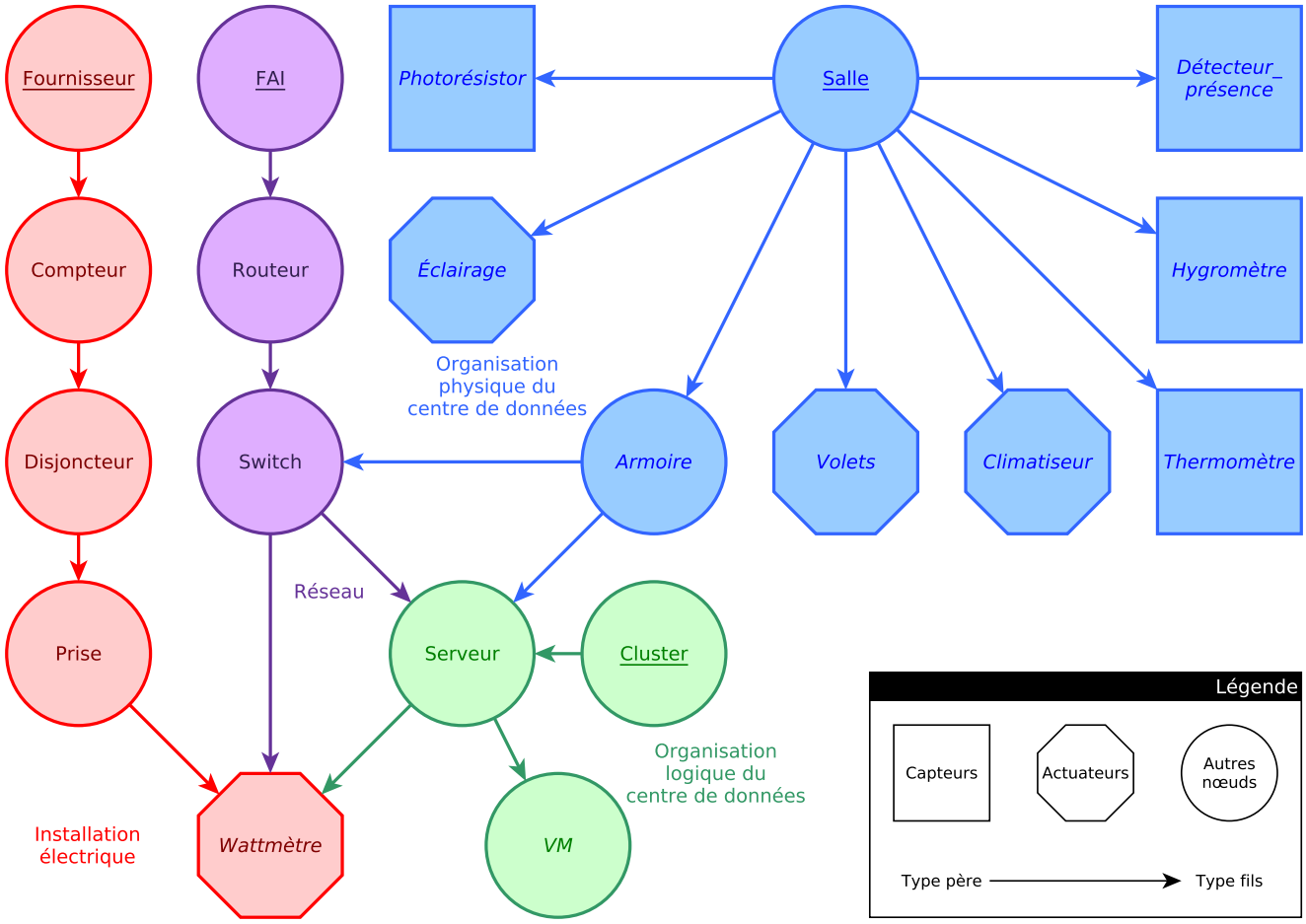


FIGURE 4.3 – Exemple de multiarbre

Le graphe 4.3 propose une modélisation en multiarbre de tels capteurs et actuateurs. Chaque groupe de nœuds du multiarbre constitue à lui seul un modèle d'arbre, un contexte nommé, qui s'inscrit dans le domaine de compétence d'un ou de plusieurs des acteurs précédemment considérés.

Avant tout, il est important de noter qu'on a ici affaire au modèle permettant la contextualisation des capteurs au sein d'un centre de données. Une instance de ce modèle correspondra alors au matériel réellement installé dans le centre. Cela explique pourquoi chacun des nœuds représente un type de matériel plutôt que chacun des matériels clairement identifiés. En résumé, le *modèle* de multiarbre décrit des *types* pour chacun des nœuds, tandis que chaque *instance* de ces nœuds possède un *identifiant* unique.

Profils utilisateurs

Comme on vient de le voir, plusieurs contextes entrent en jeu dans la modélisation. Il est possible de cataloguer une partie de ces contextes comme correspondant à un service ou un autre. Ainsi l'installation électrique ne concerne que l'électricien tandis que l'organisation physique du centre de données, qui ne regroupe que des capteurs et actuateurs non essentiels aux *services* fournis par le centre de données, concerne seulement le responsable développement durable.

Au contraire, l'organisation logique du centre de données concernera tous les acteurs excepté l'électricien. S'il est évident que le responsable SI a besoin de connaître la répartition des machines virtuelles au sein des serveurs et le regroupement des serveurs en *clusters*, il est également nécessaire au responsable réseau de maîtriser l'infrastructure réseau jusqu'aux feuilles de cette infrastructure, c'est à dire jusqu'aux serveurs, voire jusqu'aux machines virtuelles. En ce qui concerne le responsable développement durable, connaître la consommation énergétique des serveurs peut lui être utile en vue de l'amélioration de la consommation énergétique du centre de données. De la même manière, le réseau concernera non seulement le responsable réseau, mais également le responsable développement durable, qui s'intéressera à la consommation des *switches*.

Cet exemple montre bien la catégorisation en différents contextes des besoins de chacun des utilisateurs. Tout en même temps que cela offre aux utilisateurs plusieurs moyens d'identifier un même appareil, le fait que certains de ces contextes répondent à différents profils permet la mise en évidence des usages communs et une factorisation de ces usages *de facto* dans la modélisation.

Usages *cross-context*

L'ensemble des usages utilisateurs, bien que découpés en plusieurs contextes, s'inscrit bel et bien dans un même graphe. Tous les contextes sont connexes au sein du multiarbre. Par conséquent, il est tout à fait possible que des usages adressent plusieurs contextes simultanément. Le responsable développement durable aura ainsi la possibilité de s'intéresser à la consommation électrique de tout une armoire, voire de tout une salle, bien que cela mette en jeu la gestion du réseau, l'organisation physique *et* l'organisation logique du centre de données. Qui plus est, ces usages multi-contextuels ne se limitent pas à l'expérience d'un utilisateur. Il est ainsi tout à fait possible de prendre en considération des usages inter-services, et ce indépendamment de la distance des contextes dans le modèle. Ainsi, pour des questions de redondance par exemple, rien n'empêche de lister les disjoncteurs d'une armoire afin d'assurer au mieux l'alimentation des serveurs et *switches* présents dans l'armoire. Cela fait pourtant intervenir deux contextes — l'organisation physique et l'installation électrique — catalogués comme concernant chacun des services différents, qui pis est les deux contextes ne sont pas directement liés l'un à l'autre dans le modèle.

Cette flexibilité des usages dits *cross-context* est garante d'une grande évolubilité dans l'expérience des utilisateurs du réseau de capteurs. Puisque la modélisation ne limite en rien de nouveaux cas d'utilisation mettant en jeu n'importe quels contextes du multiarbre, indépendamment de la distance séparant ces contextes, une part de l'évolution des usages pourra prendre place sans nécessiter de modification du modèle, tout au moins tant que ces nouveaux usages n'impliquent pas des éléments de type non encore présents dans le multiarbre. À cette évolubilité correspond une flexibilité inhérente au modèle, dont la variété des usages n'est limitée que par l'ensemble des combinaisons entre contextes. C'est ici qu'on peut retrouver une part de la factorisation des cas d'utilisation, dès lors que certains de ces cas d'utilisation mettent en jeu plusieurs contextes.

4.2.3 Données et fonctionnalités du réseau de capteurs

Nous venons de voir comment l'environnement du réseau de capteurs vient s'articuler autour des capteurs et actuateurs sous la forme de contextes dans le multiarbre. L'objectif de SensorScript restant de permettre le traitement des données issues de ce réseau de capteurs, il convient d'inscrire ces données au sein du multiarbre.

De fait, à un type de capteurs peut être associé un ensemble de données remontées par ce capteur. Prenons l'exemple du wattmètre. Se présentant sous la forme d'un boîtier dont le matériel inclue un voltmètre et un ampèremètre, il permet de fait la mesure de la tension en volts et de l'intensité en ampères du courant. En outre, ces deux données lui permettent de fournir la consommation de puissance en watts sur la base de ces deux mesures. Trois données lui sont donc associées : la *tension*, l'*intensité* et la *puissance*. Dans le multiarbre, tous les nœuds de type *Wattmètre* vont alors proposer un *attribut* pour chacune de ces trois données. Liés au réseau de capteurs, la valeur de ces trois attributs changera pour chaque nouvelle valeur remontée depuis les capteurs afférents.

Considérons dorénavant que ces prises proposent des fonctionnalités d'actuation, par exemple l'ouverture et la fermeture du circuit et, par conséquent, l'extinction et l'allumage du matériel branché sur la prise. Cette fonctionnalité va alors correspondre à une *méthode*, comparable aux méthodes de la programmation par objet, permettant l'appel à l'ouverture ou la fermeture du circuit, en fonction de l'état dans lequel se trouve le wattmètre au moment de l'appel de la méthode. Il est également envisageable qu'un attribut puisse être alors spécifié pour connaître l'état, allumé ou éteint, du wattmètre.

Attributs et méthodes feront l'objet d'une présentation plus poussée dans la sous-section 6.1.1, montrant l'implémentation du multiarbre et des nœuds le composant.

5

Traitement des événements complexes dans un réseau de capteurs

Le chapitre précédent a établi la modélisation en multiarbre d'un réseau de capteurs et de ses environnements comme base de SensorScript.

Ce chapitre introduit le traitement des événements complexes tel qu'il s'inscrit dans SensorScript. Dans un premier temps, chacune de ses fonctionnalités est présentée en détail. Dans un second temps, l'objectif initial de proposer une interface multi-utilisateurs est adressé avec la présentation du langage dédié qui constitue cette interface.

À partir de ce point et jusqu'à la fin du document, les termes traitement des événements complexes, complex event processing *et* CEP *désigneront la même chose. L'expression* complex event processor *désignera quant à elle le* framework *permettant le traitement des événements complexes.*

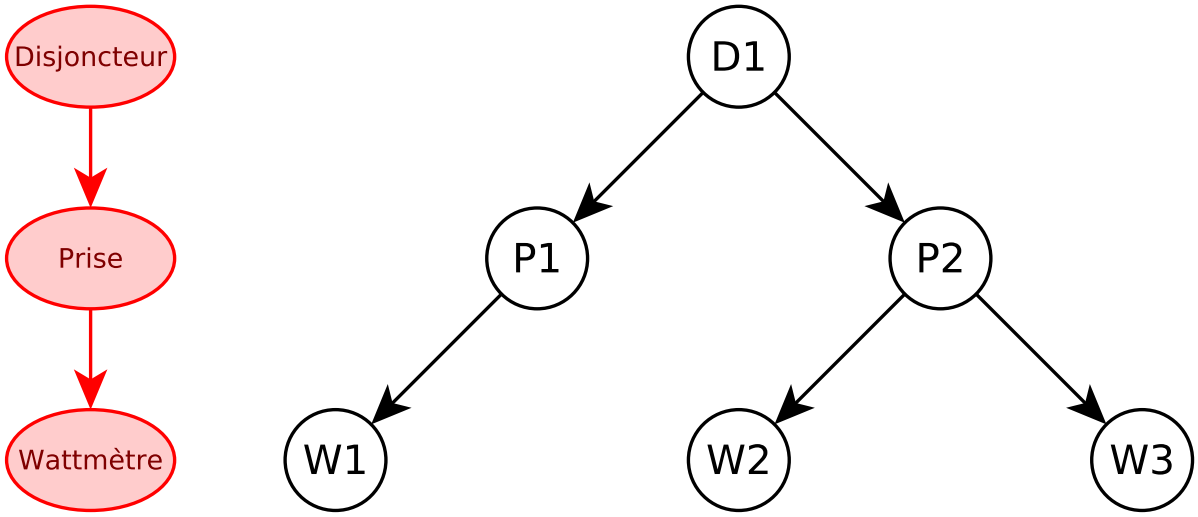
Sommaire

5.1 Traitement des événements complexes et multiarbre

La modélisation d'un réseau de capteurs et de son environnement sous forme de multiarbre fournit une structure visuellement compréhensible. Les règles qui régissent une telle structure permettent en outre un parcours simplifié au sein de cette structure. Le fait que le multiarbre consiste en un unique graphe orienté acyclique permet d'assurer l'existence systématique d'au moins un chemin entre deux types de nœuds du modèle.

Cette propriété permet d'utiliser le multiarbre lui-même comme support de filtrage intuitif sur la sélection des nœuds du graphe, ce qui sera illustré dans la sous-section 5.1.1. Dès lors, la prise en compte d'une telle fonctionnalité dans le traitement des événements complexes permet de résoudre implicitement une bonne partie des conditions exprimables dans la composition des événements. En s'appuyant sur le multiarbre, un *complex event processor* est alors à même de réduire la sélection d'un ensemble de nœuds d'un même type selon un autre nœud auquel ils sont connexes, en suivant les plus courts chemins identifiés dans le *modèle* du multiarbre.

Poursuivons sur notre exemple et concentrons-nous sur une sous-partie de l'installation électrique, selon la description proposée en figure 5.1. Il devient donc possible de réduire l'ensemble des nœuds de type *Wattmètre* liés, *au plus proche*, au nœud $P2$ à l'ensemble $\{W2, W3\}$, qu'intuitivement on identifie comme les wattmètres branchés à la prise $P2$.



(a) Modèle (b) Instance

FIGURE 5.1 – Exemple de multiarbre

Dès lors, il nous est apparu nécessaire de concevoir un *complex event processor* s'appuyant sur la structure du multiarbre pour tirer partie de cette particularité. La sous-section 5.1.1 montre comment ce mécanisme entre en jeu dans la sélection de nœuds du multiarbre. Les conditions autres que le filtrage entre nœuds sont abordées dans la sous-section 5.1.2. Enfin, l'accès aux données fait l'objet de la sous-section 5.1.3.

5.1.1 Sélection de nœuds

Dans le cadre de SensorScript, la sélection des nœuds au sein du multiarbre permet un filtrage implicite sur les types et identifiants des nœuds. L'objectif est de pouvoir lister les capteurs, actuateurs ou tout autre élément leur servant de contexte, selon le besoin exprimé, pour s'intéresser aux mesures associées ou interagir avec l'environnement lié aux dits éléments. Il est possible de distinguer deux premiers types de sélection :

- l'*identifiant* unique d'un nœud permet la sélection immédiate de ce seul nœud ; en reprenant l'exemple du graphe 5.1 il est possible de sélectionner directement la prise $P1$;

- un *type* de nœud permet la sélection de tous les nœuds du type concerné ; la sélection sur le type *Wattmètre* sélectionnera les nœuds d'identifiant $W1$, $W2$ et $W3$;

Ces deux sélections sont considérées comme *atomiques*, du fait qu'elles ne fassent intervenir qu'un élément d'identification (identifiant ou type) du ou des nœuds à sélectionner.

La notion de filtrage implicite entre les nœuds introduite précédemment constitue dans les faits une composition de ces deux sélections atomiques, on parle alors de sélection *composite*. Cela correspond à l'exemple précédent dans lequel on sélectionne les wattmètres de la prise $P2$, afin de construire l'ensemble $\{W2, W3\}$.

Plus précisément, une sélection composite fait intervenir deux sélections dans la construction de l'ensemble de nœuds S à sélectionner. L'une de ces sélections, nécessairement une sélection par type, permet la spécification du type de nœuds présents dans S. La seconde sélection spécifie quant à elle un ensemble de nœuds F servant de filtre à la sélection S. En effet, tout nœud de S qui ne sera pas directement lié à un nœud de F, en suivant le plus court chemin entre les *types* de S et F dans le *modèle* du multiarbre sera retiré de S. La sélection F peut être une sélection par type, par identifiant — auquel cas F est nécessairement un singleton — ou même une autre sélection composite. Il n'y a de fait pas de limite dans la profondeur de la composition des sélections. Dans notre exemple, la sélection S correspond aux nœuds de type *Wattmètre*, la sélection F correspond à la prise $P2$.

Pour résumer, une sélection est soit une sélection *atomique*, sur l'identifiant ou le type des nœuds, soit une sélection *composite*, mettant en œuvre la composition d'une sélection spécifiant le type à sélectionner, que nous appellerons dorénavant la sélection *finale*, et d'une sélection *filtre*, atomique ou composite, venant filtrer la sélection finale. La figure 5.2 montre un diagramme de classe illustrant ces trois sélections et les relations entre elles.

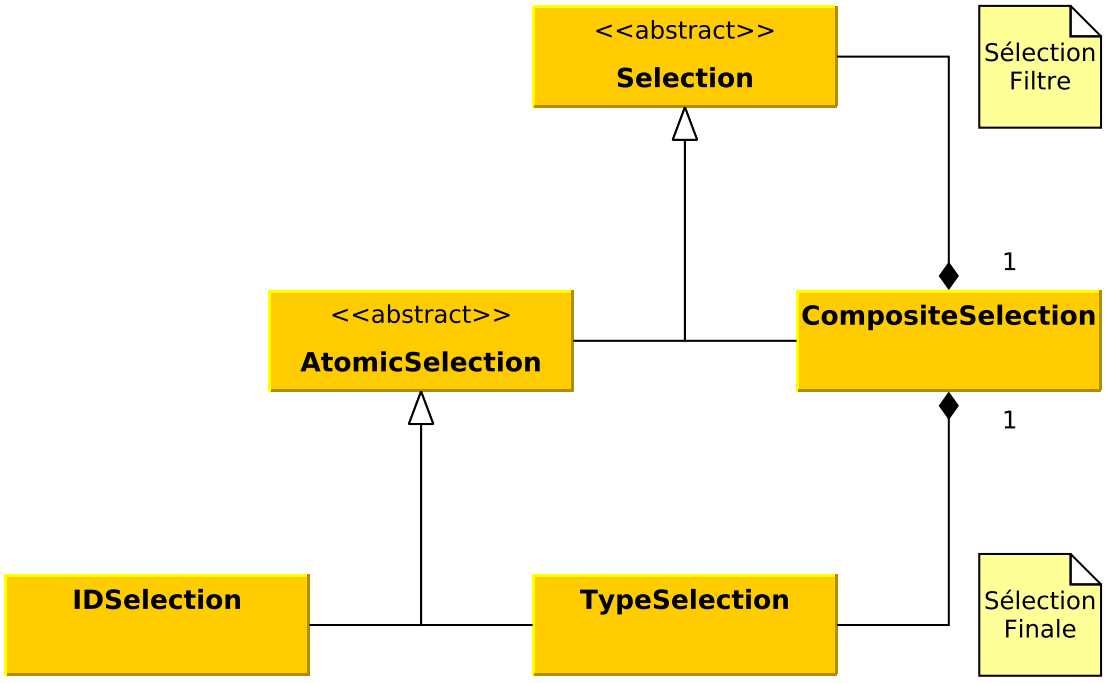


FIGURE 5.2 – Diagramme de classe des sélections

5.1.2 Expressions conditionnelles

Le traitement des événements complexes permet la composition d'événements depuis le flux de données. Cette composition met en œuvre différentes manipulations sur les données brutes (considérées alors comme événements atomiques) permises par un ensemble d'opérateurs. Le cœur de ces opérateurs porte sur des conditions à vérifier sur les événements analysés. Les conditions entrant en jeu dans le CEP permettent le suivi de l'évolution des données mesurées, et la détection de comportements constitutifs de ce qu'on appelle les *événements*.

Puisque le CEP analyse en temps réel le flux incessant des données, les opérateurs nécessaires à l'expression conditionnelle des événements doivent permettre le suivi temporel de l'évolution de ces données. Outre les opérateurs booléens et comparatifs classiques, pouvant intervenir tant sur les données elles-mêmes que sur leur horodatage, on a vu que la littérature introduit un opérateur, que nous appellerons l'opérateur *puis*, permettant de composer un événement comme la séquence d'événements atomiques. À considérer la mesure de consommation énergétique d'un centre de données, cet opérateur permet par exemple, dans le cadre de la gestion de l'organisation logique du centre de données, la détection de l'extinction d'un serveur :

Le serveur $s1$ *est allumé — il consomme plus que 0 W —* puis *le serveur* $s1$ *est éteint — il consomme 0 W.*

Puisque les wattmètres remontent des mesures énergétiques indépendamment de l'état des serveurs, la seule mesure à 0 W de la consommation d'un serveur ne suffit pas à constituer l'évènement *« le serveur s'éteint. »* En réalité c'est bien ici à l'instant où l'on va constater que le serveur commence à ne rien consommer qu'on va considérer qu'il *s'éteint*, ce qui est un *événement* (et non seulement qu'il *est* éteint, ce qui est un *état*).

Dans le multiarbre, les conditions s'appliquent sur les attributs et retours de méthodes des nœuds. Elles peuvent être exprimées tant sur les sélections atomiques que sur n'importe quelle sous-sélection d'une sélection composite, et proposent un filtre supplémentaire sur ces sélections. Chacune des conditions est alors confrontée à chaque nœud de l'ensemble des nœuds issus de la sélection concernée et seuls sont conservés les nœuds remplissant la condition. Dans un centre de données, il est ainsi possible de ne lister que les serveurs allumés par exemple.

Attributs et événements complexes

Dans le modèle, l'évolution des attributs spécifiés pour les capteurs et actuateurs, liés aux données remontées depuis le réseau de capteurs, correspond aux événements *atomiques* du CEP. L'un des points fondamentaux de la composition des événements dans SensorScript concerne la possibilité de spécifier des attributs non seulement en fonction des données issues des capteurs, mais également *via* des opérations conditionnelles, d'agrégation ou non, depuis les attributs des capteurs.

L'expression de ces attributs *composites* permet en outre d'en recalculer la valeur pour chaque nouvelle donnée pour un attribut entrant dans le calcul. Ce recalcul perpétuel constitue de fait un *événement complexe*, sous la forme d'un attribut non-atomique. Il est naturellement possible de faire appel à ces attributs composites dans la définition d'autres attributs composites.

5.1.3 Accès aux attributs et méthodes

On a vu que les conditions permettent des expressions booléennes sur les attributs et retours de méthodes des nœuds d'une sélection. Cette comparaison s'appuie sur un accès à ces attributs et méthodes, qui de la même manière sont effectués sur l'ensemble des nœuds de la sélection considérée.

L'accès à ces attributs et méthodes peut également se faire en dehors des conditions, sur les sélections atomiques et finales. Si dans le cas des attributs il s'agira de proposer une impression écran des nouveaux attributs, l'intérêt concerne ici avant tout les méthodes, puisqu'il est alors possible d'appeler une méthode sur un ensemble de nœuds ayant été filtré au travers des sélections et conditions.

Dit autrement, l'ensemble des nœuds pour lesquels a été détecté un événement complexe précis, correspondant aux sélections et conditions ayant permis la construction et le filtrage de cet ensemble de nœuds, pourra faire l'objet d'une méthode d'actuation particulière. Il s'agit là du mécanisme permettant l'automatisation de l'utilisation des fonctionnalités des actuateurs dans SensorScript.

5.2 Langage dédié

Afin de permettre l'accès au CEP à chacun des utilisateurs visés, SensorScript propose un langage dédié comme interface. Le listing 5.1 propose une grammaire simplifiée de ce langage. Puisqu'il s'agit là de l'interface permettant l'expression des interactions entre multiarbre et CEP, les trois éléments constitutifs du CEP — à savoir la sélection de nœuds du multiarbre, l'expression de conditions et l'accès à des attributs et méthodes sur ces nœuds — sont au cœur du langage.

Listing 5.1 – Grammaire simplifiée de SensorScript

```
1:            Query  →  Selection(.Access)? |
                        Selection:Selection.AggregationMethod
2:        Selection  →  (Selection/)?AtomicSelection({Condition})?
3:           Access  →  SimpleAccess | AggregationMethod
4:        Condition  →  SimpleAccess Comparator SimpleAccess |
                        Condition BooleanOperator Condition |
                        (Condition, Duration)
5:     SimpleAccess  →  Attribute | SimpleMethod | Constant
6:       Comparator  →  = | != | < | > | <= | >=
7:  BooleanOperator  →  & | | | ;
8:  AtomicSelection  →  NodeName | NodeType
```

Symboles non-terminaux
Opérateurs de description de grammaire
Symboles terminaux

Les sous-sections suivantes vont s'intéresser une à une aux règles de la grammaire et faire les liens entre celle-ci, le CEP et le multiarbre.

5.2.1 Requêtes

Du fait que le *complex event processor* de SensorScript vise non seulement à permettre la composition d'événements, mais également d'en tirer parti dans l'automatisation de décisions en fonction de ces événements, le langage ne se contente pas de permettre l'élaboration d'événements toujours plus complexes. La notion de requête recouvre donc ici non seulement la composition des événements *et* les actions à entreprendre lors de la détection d'un événement ainsi décrit. Chacune des règles de la grammaire va nous permettre de comprendre comment une telle requête se construit.

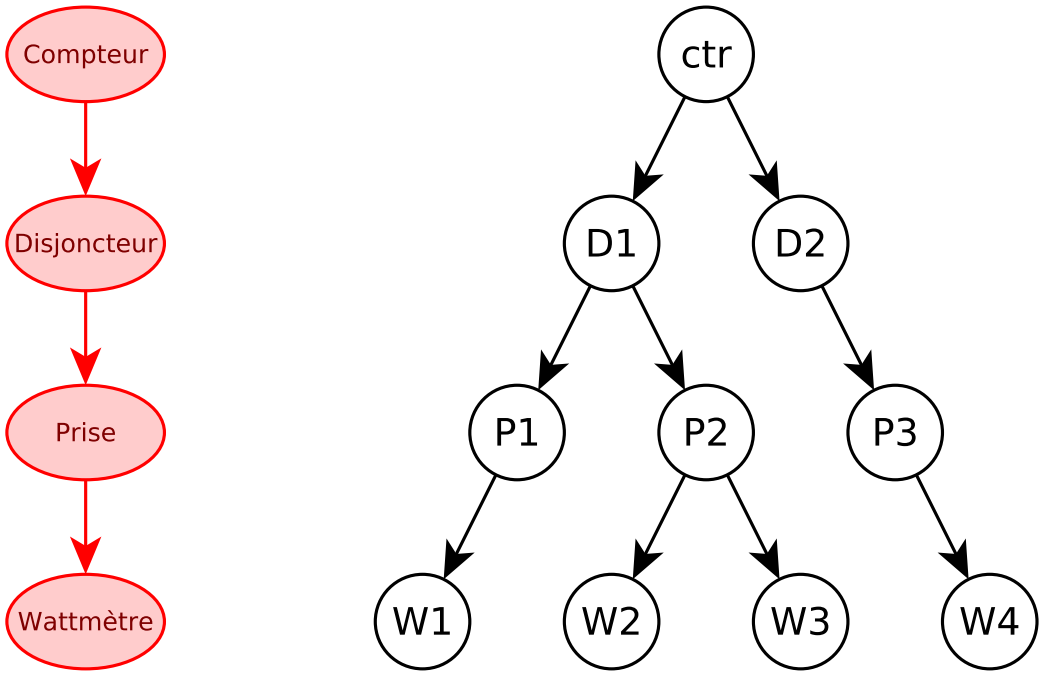
La règle 1 nous montre que toute requête est au moins une sélection de nœuds. Un accès aux attributs et méthodes des nœuds sélectionnés peut alors être exprimé. Un cas spécifique concerne les méthodes dites d'agrégation et sera détaillé dans la sous-section 5.2.4.

5.2.2 Sélections

Le développement de la règle 2 permet un nombre potentiellement illimité de sous-sélections composant la sélection filtre. La sélection finale sera quant à elle systématiquement une sélection atomique. Il est ainsi possible de décrire l'ensemble d'une sélection, non seulement comme une sélection finale atomique et une

sélection filtre composite, mais également comme plusieurs *sous-sélections* atomiques, dont celle la plus à droite sera la sélection finale.

En pratique, le filtrage implicite entre sélection filtre et sélection finale se décompose en autant de filtres qu'il y a de sous-sélections composant la sélection filtre. Pour illustrer ceci, nous étoffons en figure 5.3 la sous-partie de l'instance du multiarbre vue précédemment en y réintégrant les compteurs et en ajoutant le disjoncteur $D2$ et sa sous-hiérarchie dans l'instance.



(a) Modèle (b) Instance

FIGURE 5.3 – Exemple de multiarbre

Ainsi, pour la sélection `D1/Prise/Wattmetre`, les prises sélectionnées $\{P1..P3\}$ seront filtrées par le disjoncteur $D1$ et réduites à l'ensemble $\{P1, P2\}$, *puis* les wattmètres $\{W1..W4\}$ seront filtrés en fonction des prises sélectionnées et pré-filtrées en fonction de $D1$, ce avant même que ne soit effectivement opérée la sélection des wattmètres, pour enfin produire l'ensemble $\{W1..W3\}$. Comme vu précédemment, la structure même du multiarbre permet le filtrage entre les nœuds de l'arbre indépendamment de leur distance, telle que définie par le modèle. Ainsi, la sélection précédente peut-être simplifié en `D1/Wattmetre` : puisque le seul chemin entre wattmètres et disjoncteurs passe par le type *Prise*, il n'est pas nécessaire de le spécifier dans la requête.

Sur chacune des sous-sélections il est possible d'exprimer une condition sur les nœuds composant la sélection. L'ensemble de ces conditions, associées aux sous-sélections, viennent définir les filtres permettant l'identification des nœuds concernés par la détection de l'événement complexe associé.

5.2.3 Conditions

En ce qui concerne les conditions, la règle 4 montre une construction de conditions plus ou moins complexes sous forme d'arbres binaires [Pre99]. Une distinction est à faire entre les deux premiers développements de la règle. Le premier permet l'expression d'une *comparaison* entre deux accès sur les nœuds de la sélection finale et correspond aux feuilles de l'arbre binaire. Le second développement permet quant à lui l'expression des nœuds de l'arbre comme opérateurs booléens entre conditions. À considérer quatre comparaisons $C\{1..4\}$ comme feuilles, la figure 5.4 illustre un tel arbre binaire.

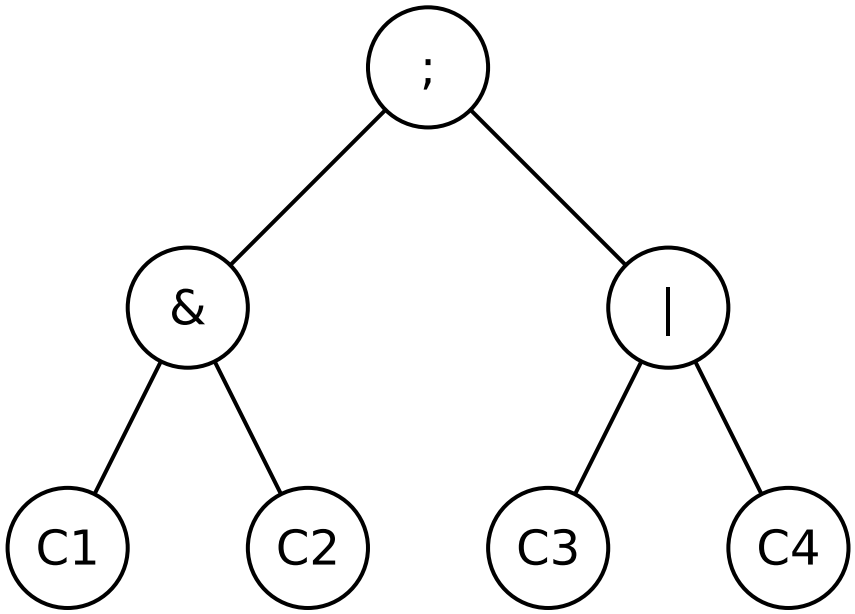


FIGURE 5.4 – Exemple d'une condition exprimée sous forme d'arbre binaire

Outre les opérateurs de comparaison et opérateurs booléens $\&$ et $|$ habituels, la règle 7 introduit le symbole « ; », correspondant à l'opérateur de séquence, ou opérateur *puis*. L'expression booléenne correspondant à notre exemple, `(C1 & C2) ; (C3 | C4)`, s'interprète alors comme :
C1 *et* C2 sont remplies, *puis* C3 *ou* C4 est remplie.

Par définition, l'opérateur « ; » n'est pas symétrique, $A; B$ n'est donc pas équivalent à $B; A$, contrairement aux opérateurs booléens classiques. Il convient alors de faire la distinction entre partie gauche et partie droite de l'expression, ou les sous-arbres gauche et droit de l'arbre binaire associé.

Ainsi, à considérer que les conditions $C\{1..4\}$ s'appliquent sur le type *Prise*, il sera alors possible, pour la sélection introduite en sous-section 5.2.2, de filtrer les prises selon ces quatre conditions des deux manières suivantes :

```
D1/Prise{(C1 & C2) ; (C3 | C4)}/Wattmetre
D1/Prise{(C2 & C1) ; (C4 | C3)}/Wattmetre
```

Enfin cette dernière sélection est bel et bien différente, puisque l'opérateur *puis* n'est pas symétrique :

```
D1/Prise{(C3 | C4) ; (C1 & C2)}/Wattmetre
```

Conditions temporelles

Le troisième développement de la règle 4 introduit l'établissement d'un lien entre une condition et une durée. L'idée est ici de permettre la vérification d'une condition, quelle que soit la complexité de son expression, sur toute la durée indiquée.

Prenons par exemple la condition temporelle $Ct \equiv (C, 10min)$, où $10min$ désigne une durée de dix minutes et où C est la sous-condition à vérifier sur cette durée. En pratique, le fait que C soit vérifiée ne permet pas à Ct de l'être immédiatement.

Supposons un moment t_0 où C est fausse. Si, à t_n, C devient vraie, Ct sera considérée vraie *si et seulement si* C ne cesse d'être vraie jusqu'à $t_n+10min$. Au contraire, si à tout instant entre t_n et $t_n+10min$, C est invalidée, alors Ct pourra être immédiatement considérée comme fausse.

Ainsi, dans notre exemple, si l'on souhaite que $C1$ et $C2$ soient vérifiées pendant dix minutes avant de vérifier $C3$ ou $C4$, l'on écrira alors :

```
D1/Prise{(C1 & C2, 10min) ; (C3 | C4)}/Wattmetre
```

5.2.4 Accès

Cette sous-section porte sur les accès aux attributs et méthodes sur les nœuds d'une sélection. La grammaire 5.1 nous montre qu'un accès peut être effectué dans deux contextes :

- soit il spécifie le résultat d'une requête, auquel cas il apparaît en suffixe d'une sélection comme le montre la règle 1 ;
- soit il intervient dans les deux membres des comparaisons.

On distingue trois types d'accès, et un cas particulier concernant l'accès aux méthodes dites d'agrégation.

Attributs

L'accès aux attributs consiste simplement a trouver la valeur de l'attribut spécifié pour chacun des nœuds de la sélection concernée. Pour les accès comme résultats de requêtes, les nœuds considérés seront ceux de la sélection finale. Pour les accès intervenant dans une comparaison, les nœuds considérés seront ceux de la sélection sur laquelle s'applique la condition. Dans tous les cas, l'accès est effectué après pré-fitrage de la sélection par ses super-sélections. La spécification d'un attribut est rendue possible par l'allocation d'un identifiant unique pour chaque attribut.

Par exemple, pour accéder à la consommation de puissance électrique des wattmètres précédemment sélectionnés, on écrira alors :

```
D1/Prise{(C1 & C2, 10min) ; (C3 | C4)}/Wattmetre.puissance
```

Constantes

Afin de permettre la comparaison d'un attribut non seulement avec un autre attribut, mais également avec des valeurs fixes, la grammaire introduit la notion d'accès à une constante. Naturellement dans le cadre du résultat d'une requête, un tel accès, bien que possible, ne présente que peu d'intérêt : cela permettra de lister la même valeur constante autant de fois qu'il y a de nœuds dans la sélection finale. L'intérêt de l'accès à une constante porte donc avant tout sur les comparaisons.

Pour illustrer cela, on va dorénavant proposer une expression concrète pour chacune des quatre conditions introduite précédemment. Supposons que l'on souhaite détecter les veilles des appareils branchés que mesurent les wattmètres durant au moins dix minutes, au moment où la dite veille prend fin. On définit la veille d'un appareil comme un profil énergétique où il présente une consommation non-nulle mais négligeable (inférieure à un seuil donné, qu'on considérera ici de 30 W) et la fin de cette veille comme soit une reprise d'activité, donc un dépassement du seuil, soit l'extinction de l'appareil. La requête devient :

```
D1/Prise{(puissance > 0 & puissance <= 30, 10min) ;
      (puissance = 0 | puissance > 30)}/Wattmetre.puissance
```

Elle permet alors, lorsque ces conditions sont remplies, de connaître la puissance alors mesurée, et donc d'identifier le type de sortie de veille : extinction ou reprise d'activité.

Méthodes

Outre les attributs, il est également possible d'accéder à des méthodes sur les nœuds. La spécification des méthodes se fait également par le truchement d'un identifiant unique. On distingue deux types de méthodes :

- les méthodes dites *simples*, qui renvoient une valeur associée à chacun des nœuds de la sélection considérée ;

- les méthodes d'*agrégation*, qui renvoient une seule valeur pour l'ensemble des nœuds de la sélection ; il s'agit ici de permettre la somme des valeurs d'un même attribut pour tous les nœuds de la sélection par exemple.

Notons qu'il est rendu possible aux développeurs de spécifier de nouvelles méthodes pour un type de nœud par héritage objet. Le principal intérêt de ces méthodes est alors de permettre l'interaction avec l'environnement par l'intermédiaire des actuateurs, voire de permettre d'agir directement sur l'organisation de l'instance du multiarbre. Considérant que les wattmètres permettent la coupure du courant pour les appareils dont ils mesurent la consommation électrique, il est alors possible de modifier notre requête pour couper l'alimentation des appareils lors de leur extinction :

```
D1/Prise{(puissance > 0 & puissance < 30=, 10min) ;
      (puissance = 0)/Wattmetre.eteindre()
```

Concernant l'évolution de l'instance même du multiarbre, considérons l'organisation logique du centre de données, notamment sa partie virtualisée. Lors d'une migration d'une machine virtuelle $VM1$ d'un serveur $S1$ à un serveur $S2$, pouvoir détacher le nœud d'identifiant $S1$ du nœud $VM1$ puis rattacher ce dernier au nœud $S2$ prend alors tout son sens.

Il est dès lors plus pertinent d'envisager ce type d'accès sur la sélection finale d'une requête que dans les conditions. C'est la raison pour laquelle la spécification de ces nouvelles méthodes peut se faire sans type de retour.

Foreach

Le troisième développement de la règle 1 nous montre un cas particulier concernant l'appel aux méthodes d'agrégation. Cet appel introduit l'opérateur « : », ou opérateur *foreach* (*pour chaque*). L'opérateur *foreach* vise à permettre le partitionnement d'un ensemble de nœuds N en plusieurs sous-ensembles $\{N_1, N_2, \ldots, N_n\}$ avant l'appel à la méthode d'agrégation. La méthode est alors appelé séparément pour chacun des sous-ensembles $N_{\{1..n\}}$, de telle sorte que l'ensemble des résultats consiste en une valeur pour chaque sous-ensemble.

Ainsi que le montre la règle 1, le partitionnement de la sélection finale se fait en fonction de la sélection filtre. Considérons la requête `S1:S2.mtd()` où S1 et S2 sont des sous-sélections, mtd une méthode d'agrégation. Le partitionnement de $S2$ se fera alors de telle sorte que chaque partition sera liée à un nœud spécifique de $S1$. Il y aura donc autant de partitions de $S2$ qu'il y a de nœuds dans $S1$.

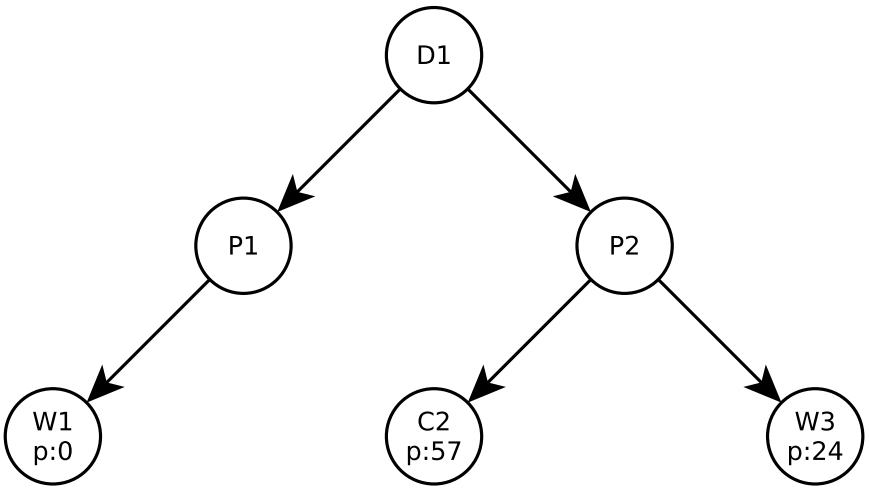


FIGURE 5.5 – *Foreach* : exemple de multiarbre

Intéressons-nous aux attributs de puissance tels qu'illustrés en figure 5.5 et, dans notre requête, faisons abstraction de la condition :

```
D1/Prise:Wattmetre.sum(puissance)
```

La requête peut-être lue ainsi : *« Pour chaque prise de $D1$, lister les wattmètres et calculer la somme de leurs puissances. »* Puisqu'il y a deux prises liées à $D1$ ($P1$ et $P2$), deux partitions seront construites. Il s'agira en l'occurrence du singleton $\{W1\}$ pour la prise $P1$, de l'ensemble $\{W2, W3\}$ pour la prise $P2$. Enfin, pour chacune des deux partitions, il s'agira de sommer l'ensemble des valeurs de l'attribut *puissance* des nœuds de la partition. Le résultat peut alors être représenté ainsi :

$P1$	$\{W1\}$	0
$P2$	$\{W2, W3\}$	81

Notons enfin que cette notion de *foreach* est à rapprocher de la clause *group by* de SQL. Le changement de nom cherche toutefois à distinguer une différence importante : l'ordre dans lequel on spécifie ce qui est sélectionné et ce qui définit le partitionnement est inversé. L'équivalent de notre exemple en SQL pourrait alors ressembler à :

```
select sum(Wattmetre.puissance)
  from Wattmetre, Prise
  where Prise.disjoncteur = 'D1'
  group by Prise.id
```

III

Implémentation et résultats

Introduction

Nous avons introduit dans les chapitres précédents la modélisation des capteurs en multiarbre, le traitement des événements complexes s'y référant et le langage dédié permettant d'orchestrer le tout. Si la façon dont ils s'articulent a été abordée, il convient de regarder dans les détails la manière dont le traitement des événements complexes est permis par SensorScript — de par une étude de la manière dont le flux de données s'inscrit dans les différentes couches applicatives — et dont il est rendu accessible aux utilisateurs au travers du langage dédié.

La figure 5.6 propose une illustration à gros grains de comment s'articulent ces différentes couches applicatives, et comment elles interagissent avec le réseau de capteurs d'une part, les utilisateurs d'autre part.

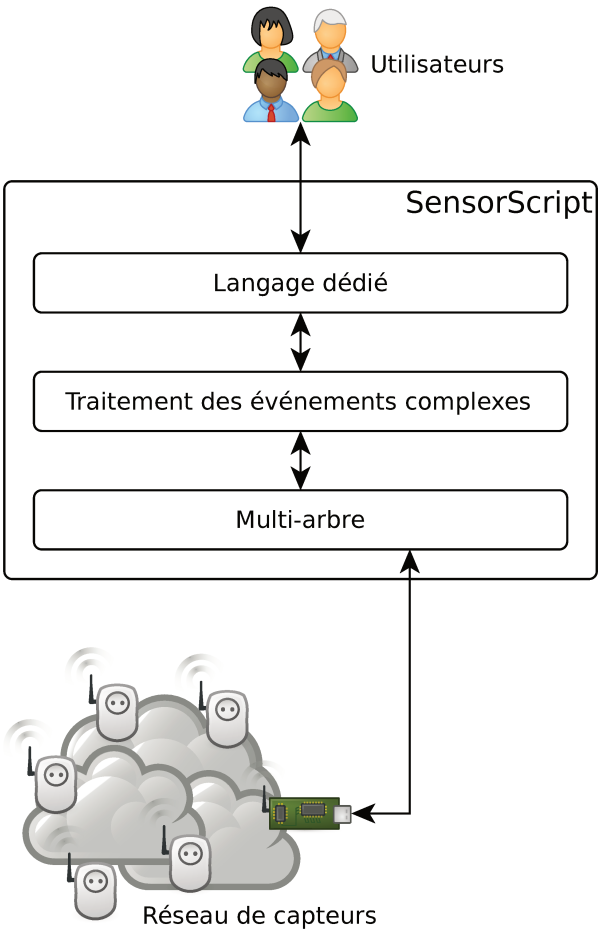


FIGURE 5.6 – Architecture de SensorScript

L'approfondissement de ce modèle fera l'objet du chapitre 6. Dans un second temps, une évaluation de la modélisation et du langage sera proposée dans le chapitre 7. Enfin, les pistes laissées ouvertes par cette thèse seront abordées en conclusion.

6

SensorScript

Ce chapitre met en lumière l'implémentation de SensorScript en s'intéressant aux mécaniques permettant le traitement des données.

L'inscription d'une requête dans le modèle par l'analyse de son expression permettra de voir ce en quoi le langage sert d'interface au traitement des événements complexes. Nous porterons également une attention toute particulière à la gestion dynamique et en temps réel du flux de données, depuis son inscription dans le multiarbre jusque dans le traitement de celles-ci par les requêtes en place.

***Nota :** l'implémentation en Java des classes les plus fondamentales présentées dans ce chapitre, ainsi que l'implémentation de la grammaire du langage en JavaCC, sont à retrouver en annexe du document.*

Sommaire

6.1 Cycle de vie d'une requête

La section 5.2 a montré comment le langage reflète les concepts entrant en jeu dans le *complex event processor* de SensorScript. Nous allons dorénavant voir comment l'expression de ces requêtes se traduit dans le modèle objet constituant l'implémentation de SensorScript.

Dans un premier temps, nous allons montrer comment l'implémentation elle-même reflète les concepts adressés précédemment et conserve de fait la flexibilité dans leurs combinaisons. L'accent sera ensuite porté sur la gestion des conditions temporelles et le mécanisme permettant le suivi de leurs sous-conditions durant toute la durée indiquée.

6.1.1 Destructuration d'une requête

Le chapitre 5 introduit la spécification du CEP appliqué au multiarbre associé en sélections, conditions et accès. Outre une logique de gestion du multiarbre permettant la sélection d'ensemble de ses nœuds, le filtrage conditionnel de ces ensembles et l'accès aux attributs et méthodes des nœuds les constituant, ces éléments sont également constitutifs du langage dédié à SensorScript.

La flexibilité dans l'expression du langage, au travers des différentes combinaisons mises en œuvre entre ces trois éléments, reflète la flexibilité du CEP sous-jacent. En pratique, le modèle de SensorScript

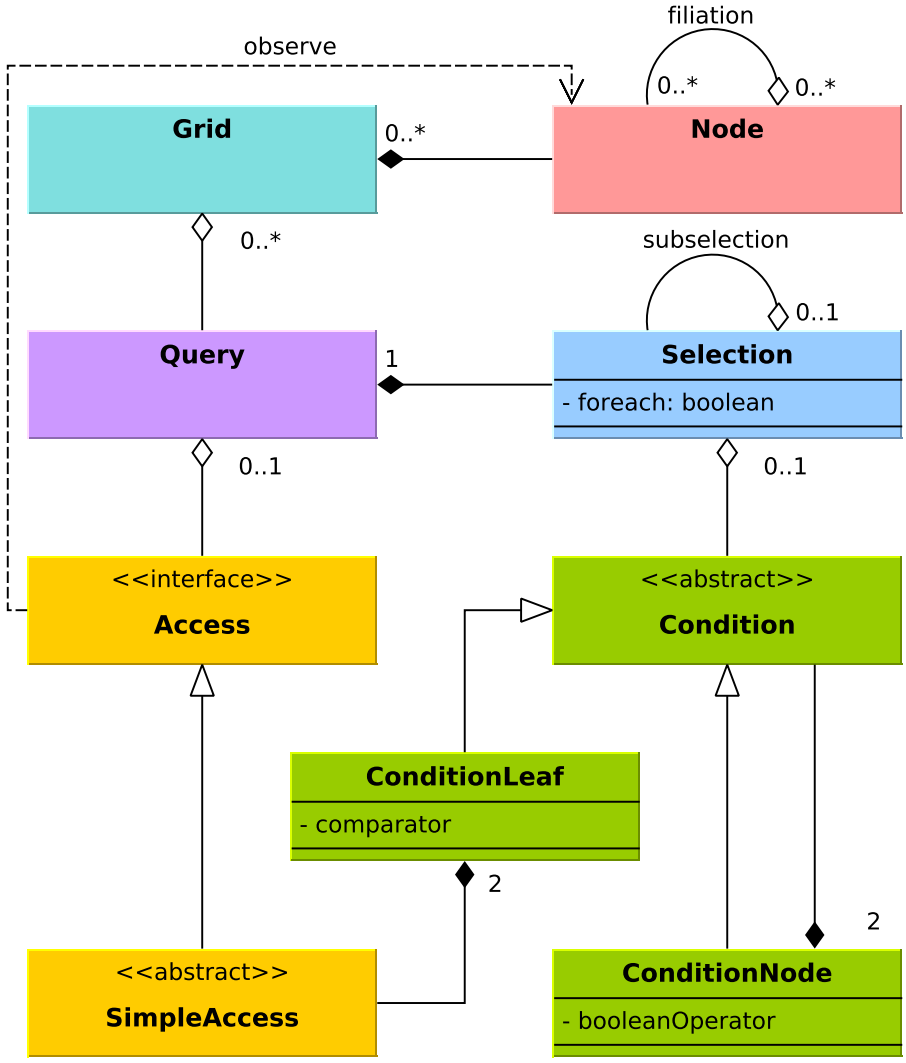


FIGURE 6.1 – SensorScript : diagramme de classe simplifié

traduit cette flexibilité par l'implémentation distincte de ces trois éléments et de leurs relations traduisant les combinaisons permises par le langage. La figure 6.1 illustre cette implémentation, et comment ces trois éléments s'articulent avec le multiarbre.

Multiarbre et requêtes

Les classes *Grid* et *Node* vienne définir l'implémentation du multiarbre. La *grille* du multiarbre est composée de nœuds, la filiation entre les nœuds étant définis par agrégation de la classe *Node* sur elle-même. Cette relation de filiation permet à un nœud d'avoir plusieurs nœuds fils, mais également plusieurs nœuds pères. En effet, un nœud pouvant appartenir à plusieurs contextes du multiarbre, il peut alors avoir un père dans chacun des sous-arbres correspondant à ces contextes. L'unique restriction concerne ici le fait que chacun des pères d'un nœud sera de *type* différent, chacune des relations prenant alors place dans un sous-arbre différent. Notons enfin qu'un nœud peut n'avoir aucun fils ou n'avoir aucun père, selon qu'il s'agisse d'une feuille ou d'une racine du multiarbre.

La classe *Grid* agrège également des requêtes, correspondant à la classe *Query*. Il s'agit simplement ici de collecter et conserver l'ensemble des requêtes à confronter aux changements de données dans le multiarbre. Pour rappel, une requête est définie comme étant soit une sélection, soit un accès sur une sélection. Cela se traduit par le fait que la classe *Query* est composée d'une et une seule instance de la classe *Selection*, et peut agréger ou non une instance de la classe *Access*.

Sélections

On a vu que, outre la dichotomie des sélections composites en sélection atomique finale d'une part et sélection — composite ou atomique — filtre d'autre part, il est également possible de considérer une sélection composite comme une chaîne de sélections atomiques. Dans une expression du type $A/B/C$, où A, B et C sont trois sélections atomiques, C sera alors la sélection finale.

Dans l'implémentation, cela se traduit par une agrégation de la classe *Selection* sur elle-même. Chaque sélection peut alors posséder ou non une *sous-sélection*. La chaîne de sous-sélections correspond alors à la sélection composite totale et la toute dernière sous-sélection, ou sélection *feuille*, est la sélection finale.

Une instance de la classe sous-sélection n'est pas composite par essence. Cela signifie que, si l'on fait abstraction de son éventuelle sous-sélection, tout sélection peut-être considérée comme sélection atomique. En pratique, une sélection sera considérée atomique *à défaut* dès lors qu'elle n'agrège pas de sous-sélection.

Conditions

Dans cette sous-section, les notions de nœud et de feuille ne concerneront que les arbres binaires et en aucun cas le multiarbre.

Sur chacune des sous-sélections composant une sélection peut être exprimée, ou non, une condition. La relation d'agrégation entre les classes *Selection* et *Condition* vient assurer cela. La représentation des conditions sous forme d'arbre binaire a été abordée en sous-section 5.2.3. Cette représentation définit l'implémentation des conditions. Ainsi, la classe *Condition* est une classe abstraite, étendue par deux classes instantiables, *ConditionNode* et *ConditionLeaf*, qui correspondent respectivement aux nœuds —racine ou non — et aux feuilles de l'arbre binaire.

Cette séparation distincte entre nœuds et feuilles est nécessaire puisque les comparaisons, constituant le cœur des conditions, ne prennent place que dans les feuilles. Ces comparaisons se font en pratique toujours entre deux accès, et de fait la classe *ConditionLeaf* est composée de deux instances de la classe abstraite *SimpleAccess*. Un attribut *comparator*, correspondant aux comparateurs présentés dans la règle 6 de la grammaire 5.1 vient définir la comparaison à effectuer entre accès *droit* et *gauche* de l'expression.

Compte tenu que les nœuds s'inscrivent dans un arbre binaire, ceux-ci sont également composée de deux sous-conditions *gauche* et *droite*. C'est la non-symétrie de l'opérateur *puis*, introduite en sous-section 5.2.3,

qui force la distinction entre sous-conditions gauche et droite. Chacune de ses deux sous-conditions pourra être à son tour un nœud ou une feuille de l'arbre binaire. De fait, c'est bien la classe abstraite *Condition* qui vient composer la classe *ConditionNode*. L'attribut *booleanOperator* est quant à lui l'un des trois opérateurs logiques « & » (*et*), « | » (*ou*) et « ; » (*puis*).

Accès

On a vu précédemment que les accès peuvent intervenir en deux points d'une requête :

- en appel à effectuer sur les nœuds constituant la sélection de la requête ;
- pour permettre la vérification des conditions, plus précisément par comparaison entre accès au sein des conditions feuilles.

Ainsi qu'on l'a déjà vu, la figure 6.1 reflète bien ces deux occurrences en liant l'interface *Access* aux classes *Query* et *ConditionLeaf*. En outre, les accès sont multiples. On a distingué en sous-section 5.2.4 trois types d'accès, que sont les accès aux constantes, attributs et méthodes, avec un cas particulier concernant les méthodes d'agrégation.

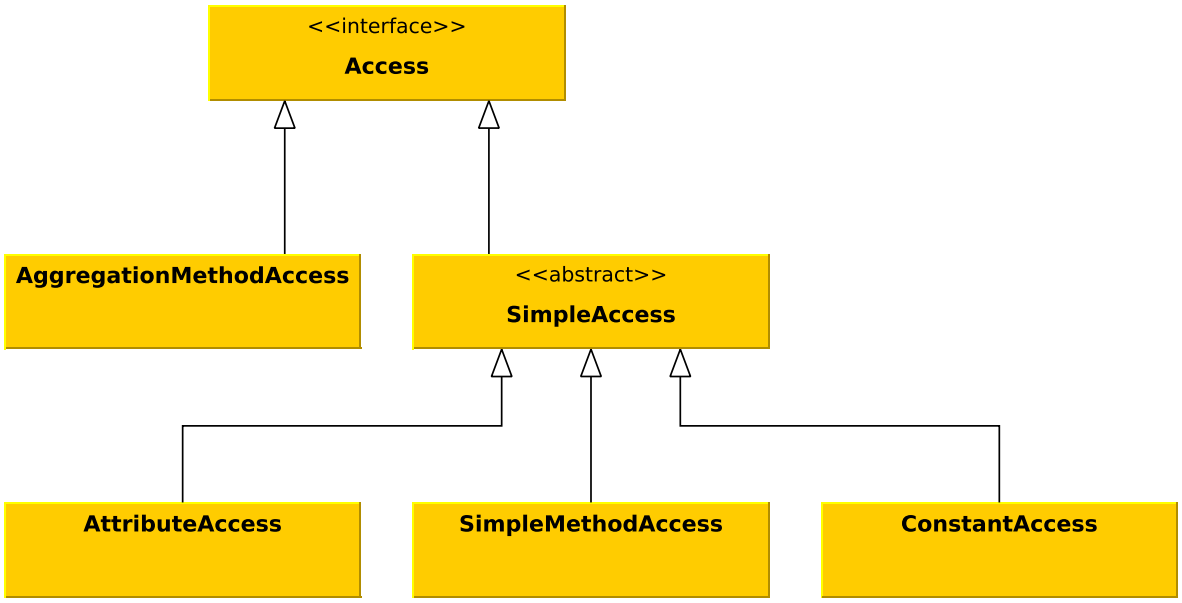


FIGURE 6.2 – Diagramme de classe des accès

La figure 6.2 montre comment ces différents accès se traduisent par héritage sur l'interface *Access*. La première distinction se fait sur les deux classes héritant directement de l'interface. On a d'un côté la classe *AggregationMethodAccess*, correspondant aux méthodes d'agrégation. De l'autre côté on trouve une classe abstraite *SimpleAccess*, classe mère des trois autres classes *AttributeAccess*, *SimpleMethodAccess* et *ConstantAccess*.

Cette distinction entre accès aux méthodes d'agrégation et accès dits simples tient au fait que, appliqués à une sélection, une méthode d'agrégation *agrège* une unique valeur pour l'ensemble des nœuds de la sélection, alors qu'un accès simple retourne une valeur pour chacun des nœuds.

Foreach

Le cas particulier concernant le partitionnement d'une sélection en vue d'une aggrégation pour chaque partition se traduit par l'attribut booléen *foreach* dans la classe sélection. Ainsi, dans une sélection composite faisant intervenir un foreach, la sous-sélection spécifiant le partitionnement aura cet attribut vrai.

Par défaut, toute sélection aura *faux* comme valeur attribuée à cet attribut. Par exemple, dans la sélection `D1/Prise:Wattmetre.sum(puissance)`, l'attribut de la sous-sélection $Prise$ aura pour valeur *vrai*, ce même attribut pour les sous-sélections $D1$ et $Wattmetre$ aura pour valeur *faux*.

6.1.2 Conditions temporelles

Les spécificités des conditions temporelles amènent à la nécessité d'un traitement particulier, en parallèle de la structure des requêtes interrogeant le multiarbre. En effet, comme nous l'avons introduit en 5.2.3, une condition temporelle consiste en un couple liant une durée à une condition « classique » non-temporelle, que par commodité nous allons dorénavant appeler *sous-condition* de la condition temporelle.

Toute la particularité d'une condition temporelle réside dans la *durée*, durant laquelle la sous-condition doit impérativement rester vraie pour que la condition temporelle soit remplie. Considérons une condition temporelle C_T de la forme (C_{SC}, d). Les contraintes sont alors multiples :

1. lorsque la sous-condition C_{SC} est remplie, il est nécessaire de suivre, durant tout la durée d, les changements du multiarbre pouvant affecter C_{SC} ;
2. si un seul de ces changements invalide C_{SC}, on peut déclarer immédiatement C_T fausse et cesser de suivre les changements du multiarbre *pour cette sélection temporelle* ;
3. lorsque la durée d s'est écoulée depuis que C_{SC} a été remplie pour la première fois, on doit pouvoir déclarer C_T vraie sans avoir à acquitter de changements dans le multiarbre ; dit autrement, c'est ici à la condition temporelle de décider dynamiquement de sa complétion et non de réagir aux changements du modèle ;
4. plusieurs conditions temporelles doivent pouvoir exister en parallèle, ne pas s'interbloquer ni bloquer le suivi dynamique des changements du multiarbre.

SensorScript étant implémenté en Java, et afin de satisfaire aux contraintes 3 et 4 sans pour autant que l'efficacité pâtisse du passage à l'échelle, nous faisons appel aux classes *Timer* (chronomètre) et *TimerTask* (tâches temporelles) du package *java.util*.

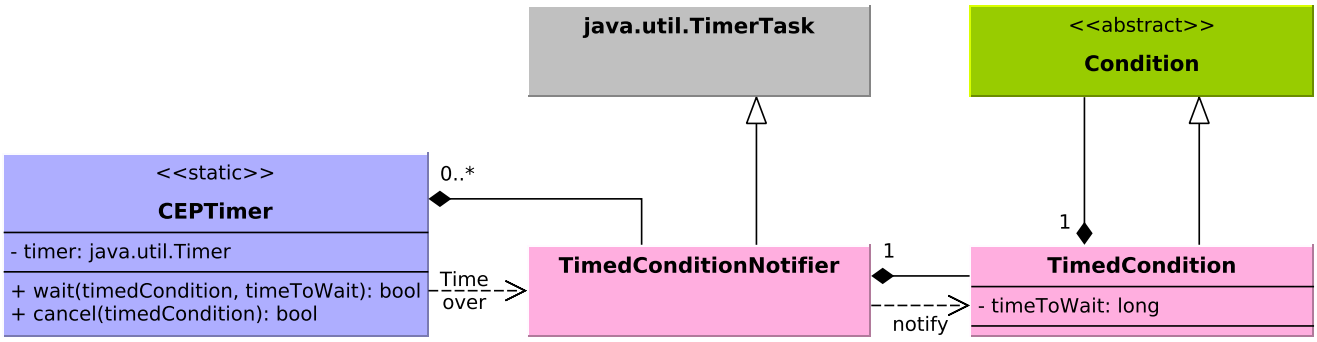


FIGURE 6.3 – Diagramme des conditions temporelles

La figure 6.3 illustre l'intégration de ces classes dans la gestion des conditions temporelles et leur relation avec les conditions « classiques », permettant la gestion des sous-conditions. La classe *TimedCondition* correspond aux conditions temporelles, et de fait elle étend la classe *Condition*. On voit également qu'une instance de cette même classe la compose, il s'agit là de la sous-condition C_{SC}. La durée d fait quant à elle l'objet de l'attribut *timeToWait*.

La gestion de d en tant que durée à attendre se fait au travers de la création d'une *TimerTask* spécifique implémentée par la classe *TimedConditionNotifier*. Définissons un instant initial t_0 où C_{SC} est fausse et un instant t_n où C_{SC} *devient* vraie. À t_n, l'instance de la classe *TimedCondition* correspondant à C_T se charge

de créer une instance de la classe *TimedConditionNotifier*. Cette instance est alors inscrite dans la classe statique *CEPTimer*.

La classe *CEPTimer* possède un attribut *timer* en charge de la gestion des *TimerTask*. C'est cet attribut qui va permettre à notre classe statique, au terme de la durée d, c'est à dire à t_{n+d}, d'exécuter la tâche de la classe *TimedConditionNotifier*. Ce mécanisme est ici représenté par le lien « *Time over* ». En pratique, une condition temporelle, lorsqu'elle crée une instance de *TimedConditionNotifier*, vient inscrire cet instance dans cette classe statique. La tâche à proprement parler de la classe *TimedConditionNotifier* consiste simplement en la notification à la condition temporelle que le temps s'est écoulé. Si, au moment où arrive cette notification, C_{SC} n'a cessé d'être vraie, alors C_T est considérée vraie, les contrainte 1 et 3 sont remplies, à savoir la gestion de la durée et le remplissage de la condition temporelle dès la fin de cette durée.

Concernant la seconde contrainte, portant sur l'arrêt du suivi de la condition temporelle dès que C_{SC} devient fausse, deux principaux mécanismes entrent ici en jeu :

1. premièrement, la classe *TimedCondition* reste au fait, durant tout la durée d, des changements de l'instance de *Condition* constituant sa sous-condition ;

2. sur un second plan, entre les instants t_n et t_{n+d}, si et lorsque la sous-condition C_{SC} devient fausse, la condition temporelle C_T indique à la classe statique *CEPTimer*, *via* sa méthode *cancel()*, qu'elle peut annuler la gestion de l'instance de *TimedConditionNotifier* correspondant à l'attente de d.

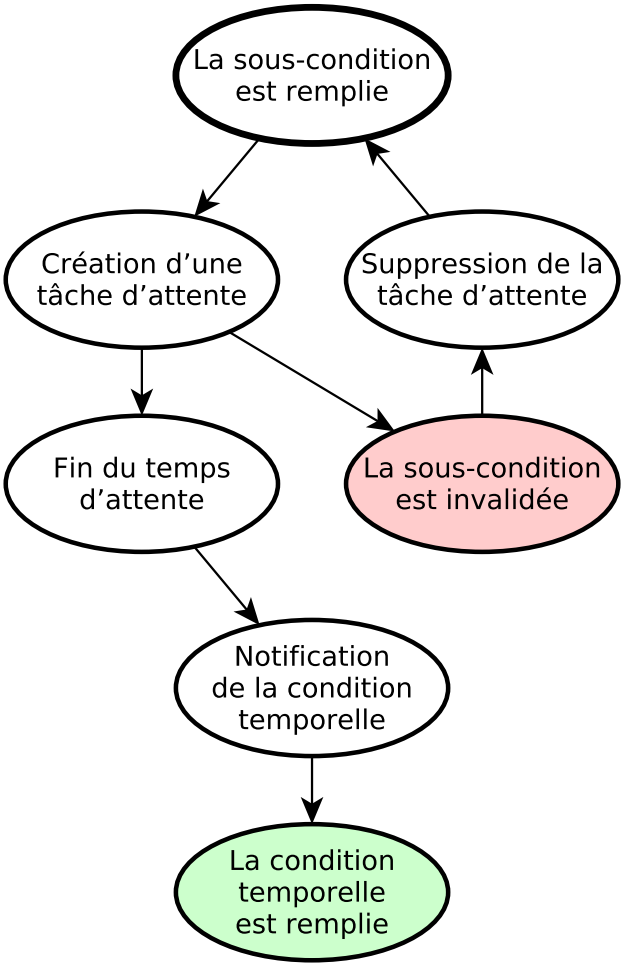


FIGURE 6.4 – Cycle de vérification d'une condition temporelle

le diagramme 6.4 présente le *workflow* correspondant à toutes ces étapes dans la gestion des conditions temporelles.

6.2 Traitement du flux de données

Le suivi dynamique des changements du multiarbre est un point fondamental dans la gestion en temps réel des requêtes. En outre, il est indispensable de prendre garde au passage à l'échelle dans son implémentation. Comme nous l'avons vu en sous-section 6.1.2, la gestion des conditions temporelles gère cette problématique en déléguant à la classe *java.util.Timer* la gestion des tâches à gérer en parallèle, plutôt que de s'appuyer sur des *Threads*, dont le nombre exploserait avec le nombre de conditions temporelles à suivre, ce qui deviendrait très vite ingérable pour la JVM (*Java Virtual Machine*).

De la même manière, toute la gestion des requêtes et du multiarbre, bien que dynamique, doit autant que faire se peut se passer de la création de sous-processus qui risqueraient de mettre à mal l'efficacité de la solution. La dynamicité s'appuie alors bien plus sur le paradigme de la programmation orientée objet, notamment le patron de conception *observateur*[1], que sur une parallélisation du traitement des données.

6.2.1 Gestion dynamique des données

Le suivi dynamique du multiarbre par les requêtes porte sur deux points :

- le premier point concerne les nouvelles valeurs du flux de données, et impacte en premier lieu les attributs des nœuds du modèle ;
- le second point porte sur les changements dans l'organisation de l'instance du multiarbre, tels que des ajouts, suppressions et déplacements de nœuds dans le modèle.

Attributs et méthodes

Les événements atomiques issus du flux de données constituent les attributs des nœuds au sein du multiarbre. De fait, le lien entre flux de données et requêtes, par le biais du modèle, passe exclusivement par la gestion de ces attributs, à savoir par les accès.

Le suivi des attributs des nœuds du multiarbre par les accès se fait grâce au patron de conception observateur, ainsi que l'illustre le diagramme 6.5. Selon ce patron, l'interface *NodeObserver* — étendue par

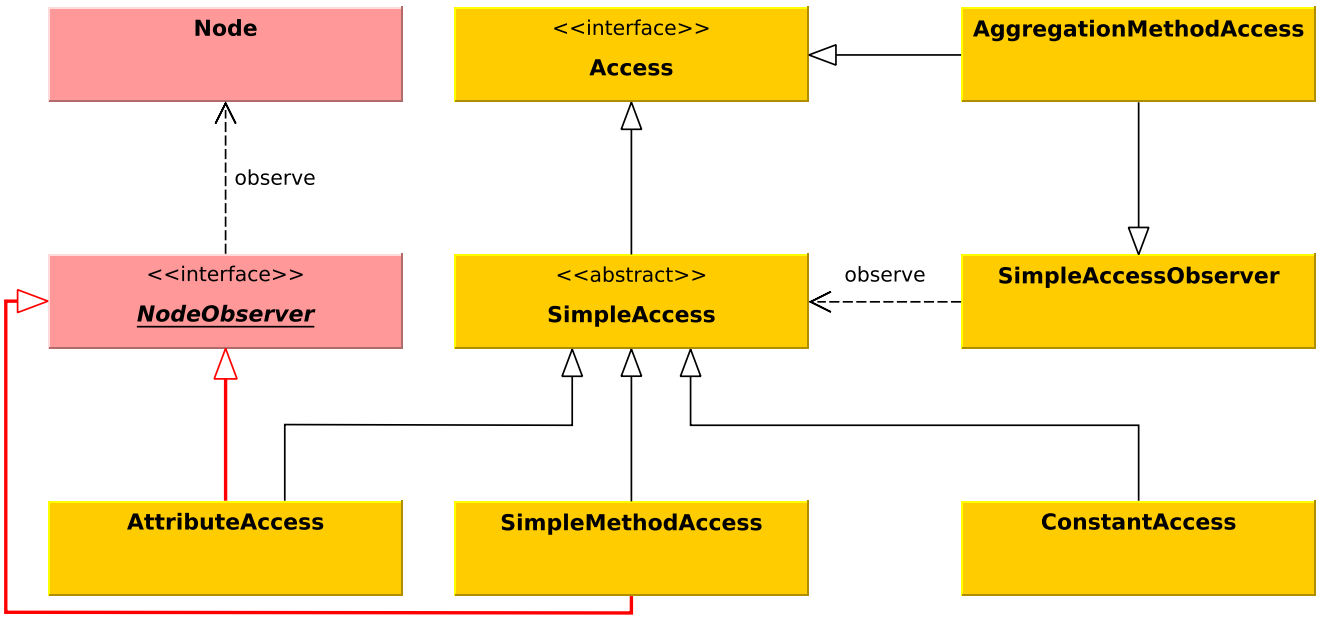


FIGURE 6.5 – Diagramme de classe d'observation par les accès des nœuds du multiarbre

[1] https://fr.wikipedia.org/wiki/Observateur_(patron_de_conception)

les classes *AttributeAccess* et *SimpleMethodAccess* — permet le suivi des attributs de la classe *Node*. Le lien entre multiarbre et accès est ici explicité par les deux relations d'héritage en rouge. Ainsi, lors de la modification d'un attribut d'un nœud en fonction du flux de données, les instances des accès observant ce nœud en seront informées.

Le fait que les méthodes simples puissent également observer le nœud s'explique par le fait que les instructions de ces méthodes peuvent faire intervenir un ou plusieurs attributs du nœud. Afin d'améliorer l'efficacité, chaque accès, en plus d'observer un nœud précis, indique à ce nœud quel sont les attributs qui l'intéressent. Ainsi, le nœud, lors de l'arrivée dans le flux de données d'une nouvelle valeur pour l'un de ces attributs, n'avertira, parmi tous ses observateurs, que les accès qui s'intéressent à cet attribut particulier.

Méthodes d'agrégation

Un cas particulier porte sur les méthodes d'agrégation. Nous considérons qu'une agrégation sur un ensemble de nœuds consiste en le calcul d'un unique résultat en fonction d'un accès simple sur chacun de ces nœuds. Cela correspondra par exemple à la somme ou la moyenne des valeurs d'un même attribut pour chacun des nœuds.

De fait, les méthodes d'agrégation observent elles-mêmes les accès simples sur lesquels elles s'appuient. L'évolution du retour d'une telle méthode après un changement dans le multiarbre sera alors fonction d'un premier acquittement du changement incriminé par au moins un des accès *agrégés* (c'est à dire les accès qui composent, et sont observés par, la méthode d'agrégation) en vue du calcul du résultat.

Organisation du multiarbre

Le dynamisme au sein de l'instance d'un multiarbre porte non seulement sur la mise à jour des attributs et retours de méthodes des nœuds en fonction du flux de données, mais également sur l'organisation même de l'instance. Ainsi, durant le cycle de vie de SensorScript, il est possible d'ajouter, de supprimer voire de déplacer des nœuds au sein du graphe. La seule restriction tient au fait qu'un nœud doit respecter les hiérarchies établies par le modèle : si dans le modèle le type A est défini comme père du type B, on ne pourra déplacer le nœud B_1 fils du nœud A_1 que comme fils d'un autre nœud de type A.

Toutefois, de telles modifications, de telles *mutations* dans l'organisation du multiarbre impliquent que les ensembles de nœuds correspondant aux sélections des requêtes changent de fait lors d'ajouts, de suppression et de déplacement de nœuds. Ainsi la sélection A_1/B devra voir B_1 disparaître lors de son déplacement dans le modèle, ou de sa suppression du modèle.

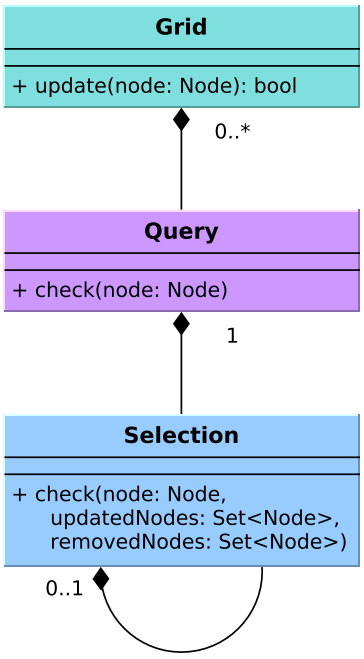


FIGURE 6.6 – Suivi des mutations du multiarbre

Afin d'assurer un tel fonctionnement, la classe *Grid*, listant à la fois l'ensemble des requêtes et l'ensemble des nœuds du multiarbre, est non seulement en charge des mutations de l'instance du multiarbre, mais se charge, *via* sa méthode *update()*, de prévenir chacune des requêtes de vérifier si l'une de ses sous-sélections est affectée par l'ajout, la suppression ou le déplacement d'un nœud. Les méthodes *check()* des classes *Query* et *Selection* sont celles qui se chargent effectivement de prendre en compte la mutation et d'affecter les ensembles de nœuds correspondants. Les entrées et sorties de ces trois méthodes sont illustrées en figure 6.6.

Partant du principe que les mutations du modèle demeurent des événements plus ponctuels que la mise à jour des attributs des nœuds depuis le flux de données, le fait de forcer chaque requête à acquitter la mutation, que la requête soit effectivement impactée ou non, ne pose pas un problème majeur. En outre, afin d'assurer dans ce processus un minimum d'efficacité, le parcours des sous-sélections d'une sélection prend

fin dès qu'une sous-sélection est impactée par la mutation. La mutation est alors prise en compte par la dite sous-sélection et le traitement de la mutation par la requête est terminé.

Traitement en temps réel des nouvelles données

Dans le traitement en temps réel, par les requêtes, de l'inscription du flux de données dans le multiarbre, l'observation des nœuds n'est que la première étape. Afin de permettre aux requêtes la prise en compte des données mises à jours et le suivi de l'évolution des sélections et des conditions, tout un processus se met en branle lorsqu'un accès observe un changement sur les attributs d'un nœud.

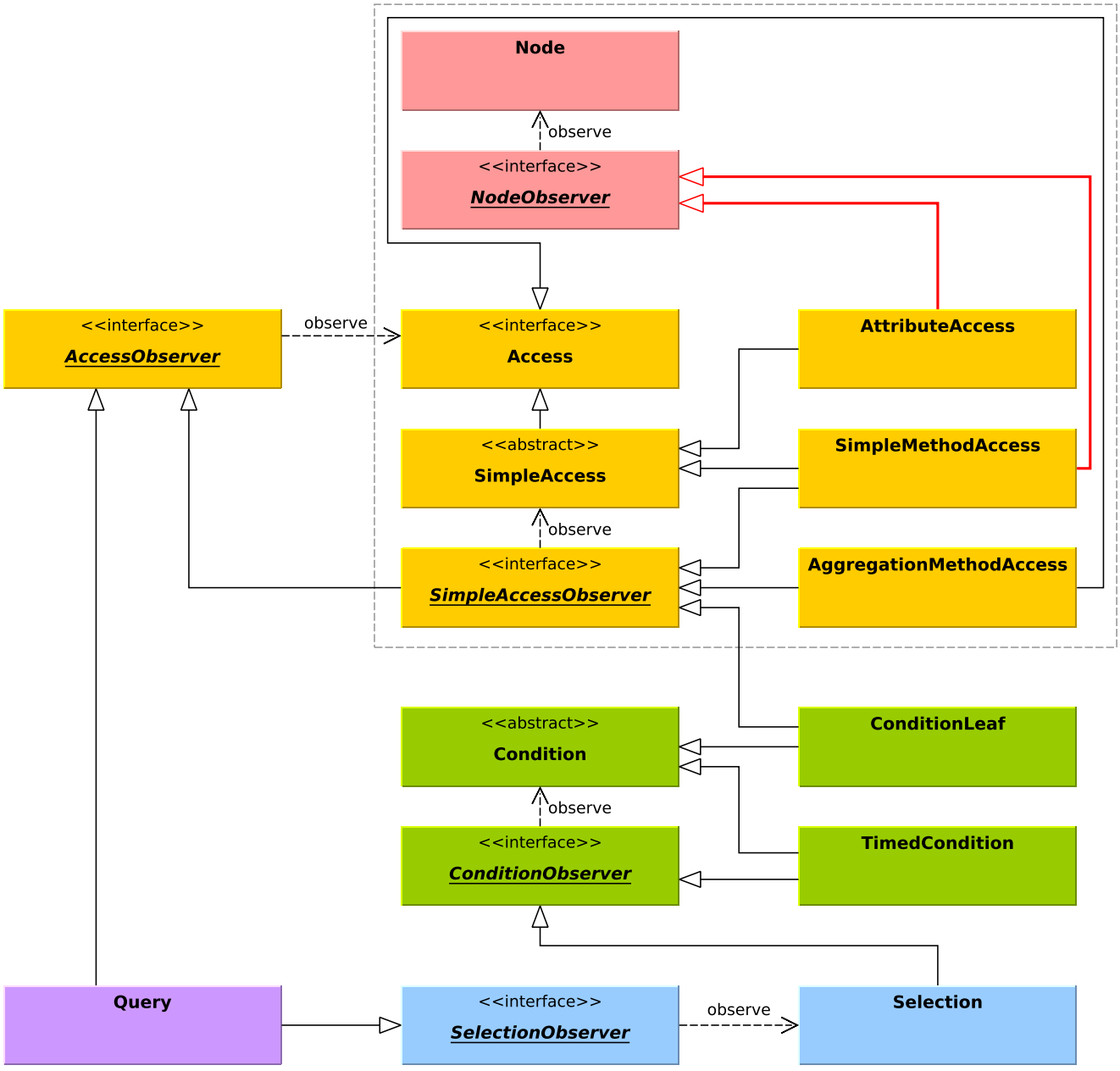


FIGURE 6.7 – Diagramme de classe de la chaîne d'observation des nœuds aux requêtes

Dans cette optique, c'est en pratique plusieurs itérations du patron de conception observateur qui, couplées aux relations d'héritage entre les classes constitutives d'une requête, permettent la propagation des changements du multiarbre parmi toutes ces classes et, ce faisant, le traitement dynamique de ces changements par les requêtes. Le diagramme de la figure 6.7 illustre cette *chaîne d'observation* depuis le multiarbre

jusqu'aux requêtes. On retrouve ici le diagramme 6.5 dans le cadre en pointillés gris en haut à droite. Seuls les accès constants ne sont plus montrés, du fait que par définition ils n'ont aucune influence sur la dynamicité des requêtes.

En réalité la notification d'un accès par le nœud qu'il observe peut se répercuter de deux manières jusque dans la requête. On a de fait affaire à deux sous-chaînes d'observation plutôt qu'une. Pour illustrer cela, nous proposons de suivre ici, pour une même requête, deux événements l'impactant chacun selon l'une de ces séquences d'observation.

Considérons donc la requête $R \equiv VM_1\{cpu > 90\}.ram$. On cherche ici à connaître la consommation mémoire de la machine virtuelle VM_1 quand sa consommation processeur dépasse les 90%. La requête R se décompose donc en :

- une sélection $S \equiv VM_1$;
- une condition $C \equiv \{cpu > 90\}$, composée d'une comparaison unique entre un accès à un attribut (cpu) et un accès à une constante (90) ;
- un accès final $A \equiv .ram$.

Premier événement : mise à jour de cpu

Comme on l'a vu, la mise à jour d'un attribut d'un nœud force celui-ci à notifier chacun des accès observant ce nœud et cet attribut. L'accès simple à cpu entre ici en jeu dans la condition de R. Le second accès de la condition est ici un accès à une constante (ayant pour valeur 90), et n'entre donc pas en jeu dans la chaîne d'observation.

Comme le montre la figure 6.7, la gestion par la condition C des accès qui la constituent se traduit par l'observation de ces accès par la condition C. Ainsi, lors d'un changement de la consommation processeur de la machine virtuelle VM_1, l'accès à cette attribut présent dans la condition C prend en compte immédiatement ce changement et, en conséquence, notifie à son tour la condition C.

Plus précisément, les accès d'une condition ne peuvent notifier qu'une instance de la classe *ConditionLeaf*, dans notre cas l'arbre binaire de la condition C est un simple nœud, et de fait la condition C sera instanciée par la classe *ConditionLeaf*. Dans le cadre d'une condition C plus complexe, dont l'arbre binaire contiendrait plus d'un nœud, la classe *ConditionLeaf* servirait alors d'intermédiaire, dans la chaîne de notification, entre l'accès et la condition C, qui serait alors instanciée par la classe *ConditionNode*.

De la même manière, la sélection S — en l'occurrence VM_1 — sur laquelle s'applique la condition C observe directement C et est observée par l'instance de la requête correspondant à R. De fait, lors de l'acquittement par la condition du changement de valeur de cpu pour le nœud VM_1, celle-ci notifie la sélection S, qui à son tour notifie la requête R. C'est alors que, en bout de course et à supposer que la condition soit vraie — que la nouvelle valeur de cpu dépasse 90 — la requête accède à la dernière valeur de ram. Dans ce cas, la gestion du résultat de la requête est laissé au comportement par défaut, qui peut être défini comme un simple affichage de la valeur à l'écran.

Notons enfin que la classe *Selection* n'est pas la seule à étendre l'interface *ConditionObserver*, et donc à observer les conditions. Comme on l'a vu en 6.1.2, une condition temporelle est systématiquement composée d'une sous-condition. La prise en compte dynamique de la vérification ou non de la sous-condition par la condition temporelle avait alors été abordée. En pratique, cette gestion en temps réel passe également par l'observation, par les conditions temporelles, de leur sous-condition. C'est pourquoi on constate que la classe *TimedCondition* hérite également de l'interface *ConditionObserver*.

Second événement : mise à jour de ram

Dans la requête R, on constate que l'accès à ram intervient cette fois non pas dans la condition, mais comme accès final de la requête. En pratique, une requête observe non seulement sa condition, mais également son accès, s'il se trouve qu'elle en agrège un. De fait, lors de la mise à jour de ram et *seulement si* la

dernière valeur de la condition C est vraie — c'est à dire si et seulement si la dernière valeur remontée de cpu dépasse 90 — l'affichage à l'écran de la valeur de ram a alors à nouveau lieu.

Un point important à noter ici concerne l'introduction de l'interface *AccessObserver*, dont l'observation n'est pas limitée aux seuls accès simples — et dont l'interface *SimpleAccessObserver* hérite. En effet, l'accès final d'une requête n'est, contrairement aux accès dans les conditions, en aucun cas limité aux seuls accès simples. Ainsi, l'accès final à une méthode d'agrégation sur la sélection d'un ensemble de nœud peut donc ainsi être également gérée dynamiquement par les requêtes.

6.2.2 SensorScript en fonctionnement

Cette section propose une analyse plus fonctionnelle de la gestion du multiarbre et des requêtes par SensorScript. En prenant un peu de recul vis-à-vis de l'implémentation, nous proposons ici une rationalisation des mécanismes tels qu'ils peuvent être vus avec une vision à plus gros grain de la solution.

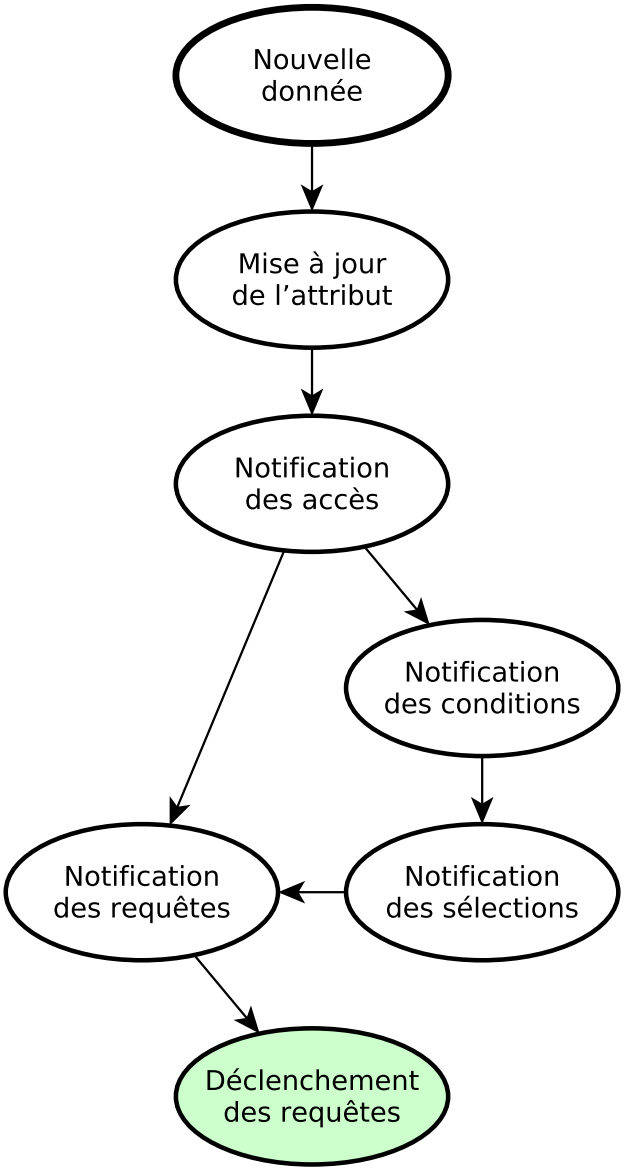


FIGURE 6.8 – Chaîne de notification lors de l'arrivée d'une nouvelle donnée

Dans un premier temps, la figure 6.8 montre le *workflow* correspondant à la chaîne d'observation décrite *via* la figure 6.7. L'arrivée d'une nouvelle donnée provoque ainsi la mise à jour de l'attribut du nœud

correspondant dans le multiarbre. Ainsi, la mise à jour de l'attribut a d'un nœud N_1 s'accompagne de la notification des accès observant précisément l'attribut a du nœud N_1.

Ensuite, selon que ces accès prennent place comme accès final d'une requête ou s'inscrivent dans une condition, la notification des requêtes sera soit directe, soit passera par les notifications successives des conditions puis des sélections constitutives des dites requêtes.

Enfin, les requêtes, une fois notifiées, prennent en compte les changements opérées sur chacune de leurs parties impactées par la nouvelle donnée — et donc notifiées dans la chaîne — en vue de leur déclenchement.

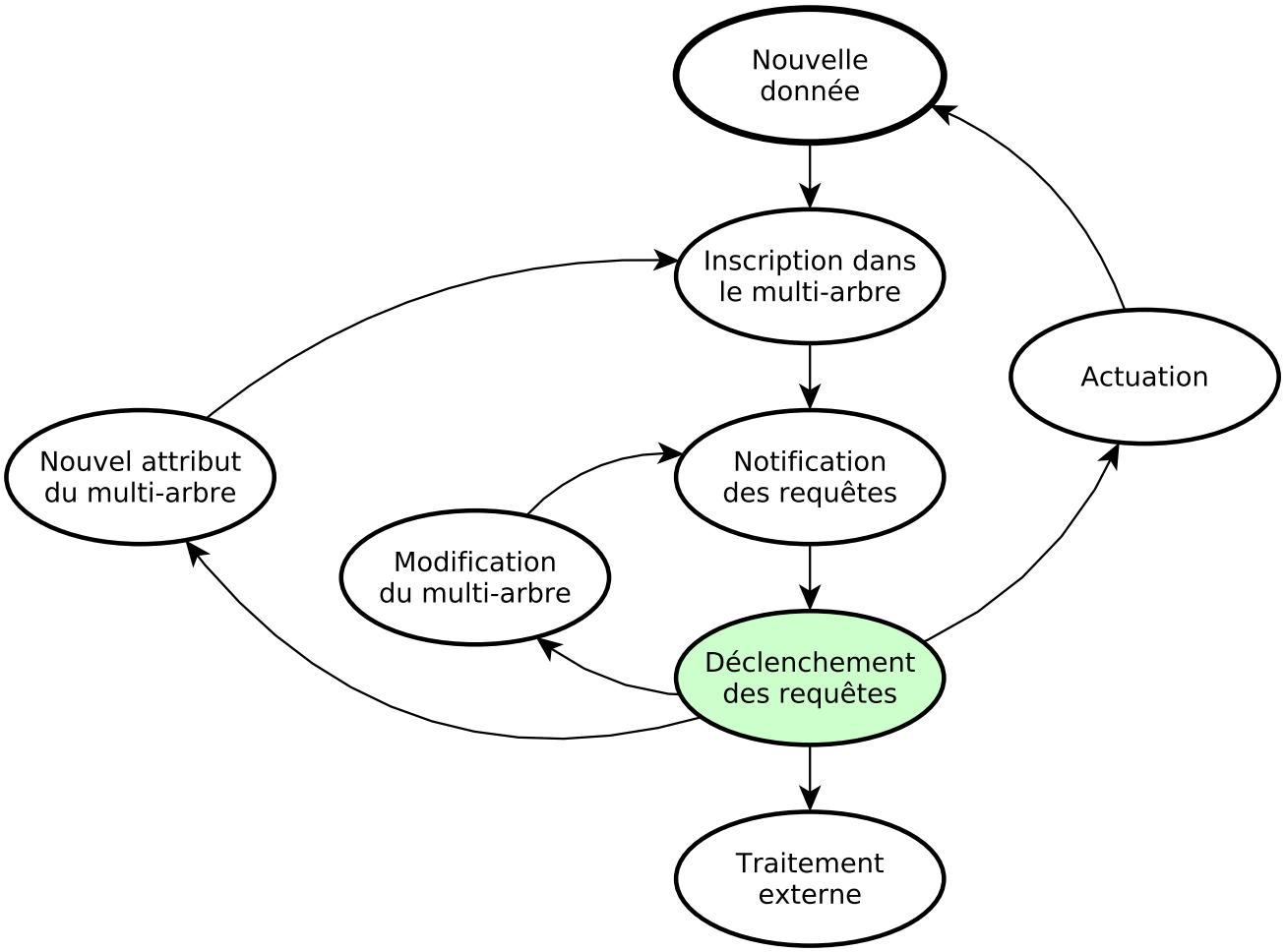


FIGURE 6.9 – Traitement d'une nouvelle donnée

La figure 6.9 illustre quant à elle le fonctionnement global de SensorScript. On retrouve ainsi — en simplifié — la notification des requêtes lors de l'arrivée d'une nouvelle donnée. Les mutations du multiarbre sont également abordées comme parmi les possibilités offertes par le déclenchement des requêtes.

L'actuation du réseau de capteur fait également partie des fonctionnalités offertes par les requêtes. Nous considérons que l'actuation d'un capteur par une requête peut alors mener à une nouvelle donnée. C'est alors un processus continu qui se met en place.

Outre l'actuation, deux autres fonctionnalités offertes par les requêtes peuvent en retour influer sur leur processus de déclenchement. Il s'agit d'une part de l'inscription d'un nouvel attribut, par essence composite puisque calculé par une requête, dans le multiarbre, d'autre part des modifications, ou mutations, du modèle. C'est ici qu'entre en jeu la composition des événements en continu et en temps réel. Si les requêtes dont la composition dépend d'autres requêtes *primitives* ne seront pas directement influencées par les changements du multiarbre, c'est le déclenchement des requêtes primitives qui viendra permettre l'analyse des requêtes

plus complexes, ce *via* les impacts qu'auront les requêtes primitives sur les attributs et l'organisation du multiarbre.

Le traitement externe est enfin présent pour toutes les fonctionnalités n'ayant d'influence ni sur le réseau de capteurs, ni sur le multiarbre. Il s'agira par exemple d'une impression écran ou d'un stockage en base de données.

7

Évaluations

Ce chapitre se consacre à l'évaluation de SensorScript. Cette évaluation porte sur deux points.

Du fait que la majeure partie des solutions de traitement des événements complexes s'appuie sur le paradigme objet-relationnel, nous proposons d'abord une étude de la modélisation du multiarbre et des contextes sous-jacents dans un tel modèle de traitement des données.

Sur base de ce modèle, nous établissons une comparaison entre le langage dédié à SensorScript et un SQL, où l'on s'intéresse plus particulièrement à l'expressivité des langages, à leur concision syntaxique, aux concepts qu'ils mettent en jeu et aux fonctionnalités qu'ils permettent.

Sommaire

7.1 Protocole d'évaluation

Les deux contributions majeures de cette thèse, implémentées dans SensorScript, concernent pour l'une l'inscription d'un réseau de capteurs dans une modélisation orientée utilisateurs, pour l'autre le langage permettant d'interagir, *via* cette modélisation, avec les capteurs et actuateurs du réseau de capteurs.

Si SensorScript se veut une preuve de concept d'une solution mettant en œuvre ces deux contributions, la solution a été implémentée dans un contexte clairement défini — le traitement des événements complexes sur la base des données issues d'un réseau de capteurs — et met en œuvre un langage dédié, non Turing-complet et par essence limité aux seuls cas prédéfinis. De fait, l'évaluation proposée ici s'inscrit également dans ce contexte.

Afin d'estimer la valeur des deux contributions, nous nous proposons d'établir une comparaison, sur la base d'un scénario test, entre SensorScript et les solutions permettant le traitement des événements complexes dans des cadres plus génériques. L'objectif est donc de montrer l'avantage d'apporter une solution spécifique aux réseaux de capteurs, tant pour la modélisation que pour le langage de requêtes, vis-à-vis d'une solution générique. Rappelons que la plupart des solutions génériques étudiées dans la littérature s'inspirent toutes des SQL, et s'appuient de fait sur le modèle objet-relationnel.

Après l'introduction du scénario d'évaluation en section 7.2, cette comparaison portera sur deux points. La section 7.3 s'attardera sur la représentation du réseau de capteurs et des contextes utilisateurs dans le modèle objet-relationnel et ce qu'une telle modélisation permet et ne permet pas, comparée à SensorScript. La section 7.4 s'intéressera quant à elle à l'expérience d'un utilisateur tant avec le langage de SensorScript qu'avec CQL [ABW03], sur la base du modèle spécifié en 7.3.

7.2 Scénario de comparaison

Le scénario d'évaluation repose sur l'exemple introduit dès le chapitre 4, dans lequel quatre acteurs que sont l'électricien et les responsable SI, réseau et développement durable d'un centre de données, sont amenés à interagir avec différents capteurs et actuateurs. Le graphe 7.1 est la modélisation du multiarbre que nous proposions alors dans SensorScript.

Détaillons ici les capteurs et actuateurs présents dans le réseau sous-jacent :

- *Wattmètre* : comme vu précédemment, il s'agit à la fois d'un capteur et d'un actuateur, en mesure de remonter les consommations en intensité, tension et puissances du matériel qu'il alimente et d'ouvrir et fermer le circuit permettant effectivement l'alimentation du dit matériel ; il fait le lien entre le matériel — *switches* ou serveurs — du centre de données et les prises électriques ;

- toute une série d'actuateurs et de capteurs directement reliés à chaque salle du centre de données, et principalement mis à disposition du responsable développement durable :

  - *Photorésistor* : il s'agit d'un capteur de lumière placé à l'*extérieur* du bâtiment, à côté des fenêtres de la salle concernée et permettant la mesure de l'ensoleillement permis par les dites fenêtres ;
  - *Éclairage* : un actuateur permet ici le contrôle de l'éclairage de la salle ;
  - *Volets* : cet actuateur contrôle l'ouverture et la fermeture des volets électriques aux fenêtres ;
  - *Climatiseur* : il s'agit ici du contrôle de la climatisation ;
  - *Thermomètre* : un simple capteur de température ;
  - *Hygromètre* : un simple capteur d'humidité de l'air ;
  - *Détecteur_présence* : un capteur permettant d'identifier la présence d'un des acteurs dans la salle.

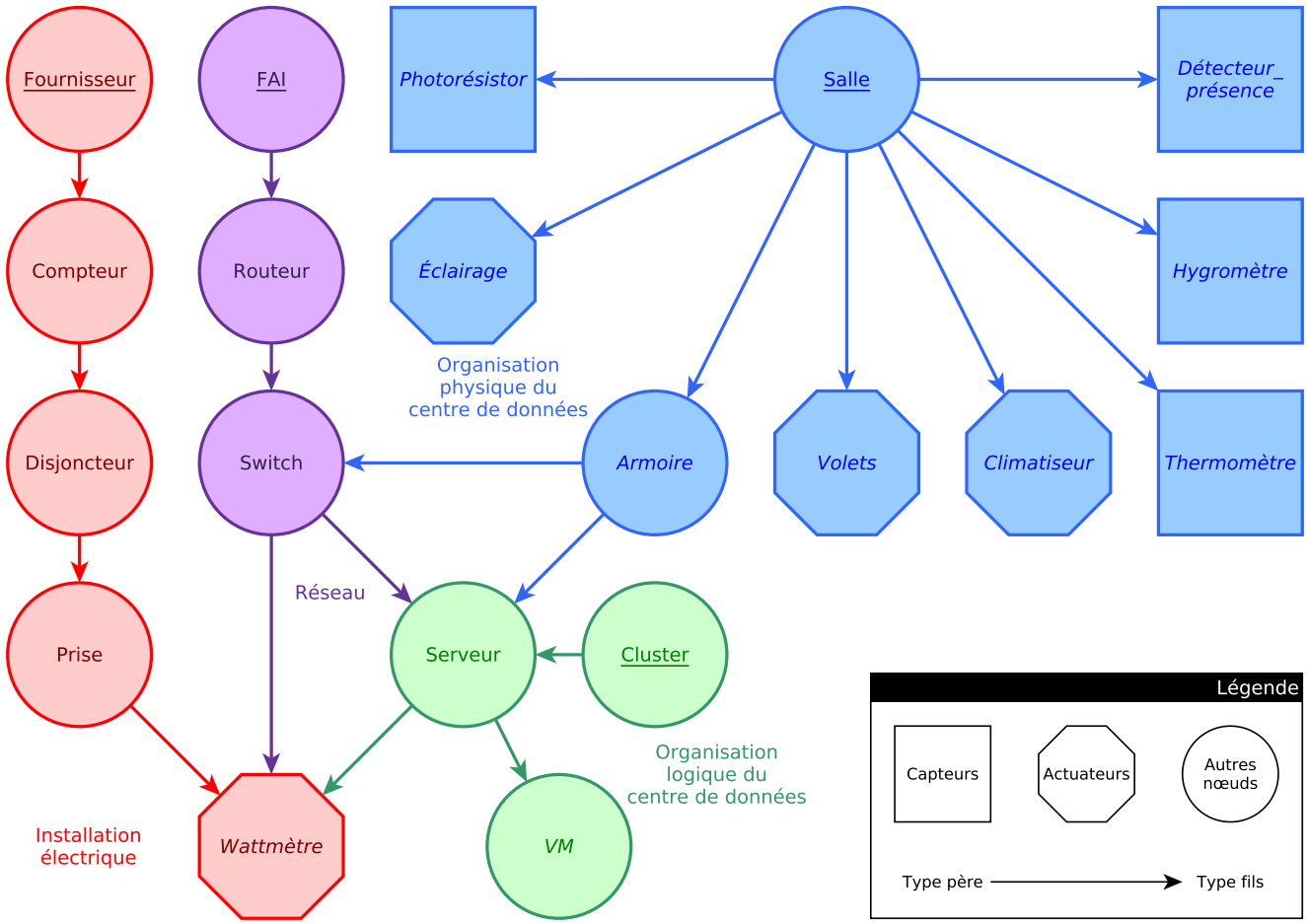


FIGURE 7.1 – Modèle de multiarbre pour le scénario de comparaison

Sur la base de ce modèle sera établi en section 7.4 le scénario d'utilisation du réseau de capteurs *via* le multiarbre. Ce scénario fera intervenir tour à tour des besoins de chacun des différents acteurs.

7.3 Modélisation des contextes utilisateurs d'un réseau de capteurs

Le modèle objet-relationnel s'appuie sur la représentation d'*objets* — au sens compris dans la programmation orientée objet – sous la forme des entrées des tables du paradigme relationnel, issu des bases de données. Les relations entre les *tables* viennent alors définir les relations entre les *classes* définissant ces objets. Du fait que, dans SensorScript, la modélisation du multiarbre s'appuie sur ce même paradigme objet afin d'établir les liens entre les nœuds du multiarbre, une solution intuitive consiste en la définition, pour chaque type de nœud du multiarbre, d'une table correspondant dans le modèle objet relationnel, avec l'établissement de relations $0..1 \rightarrow 0..*$ entre un type père et un type fils.

La figure 7.2 illustre un tel modèle. Les relations père-fils sont ici assurées par l'attribution, dans la table fille, d'une clé étrangère identifiant une entrée de la table mère. Naturellement, pour les types ayant plusieurs types pères, il est alors nécessaire d'avoir autant de clés étrangères qu'il y a de types pères. Ainsi, par exemple, en plus d'un champ identifiant chaque entrée de la table *Serveur*, cette dernière possèdera également deux attributs, chacun clé étrangère pointant respectivement vers les tables *Armoire* et *Switch*.

Si le modèle objet-relationnel ne se cantonne pas à la modèlisation de telles structures arborescentes, le revers de cette médaille est qu'il n'est pas en mesure de permettre la liaison implicite entre deux types d'un même arbre distants dans le multiarbre. De fait, sur la base de cette seule modélisation, accéder dans une même requête à deux types distants du modèle force à identifier chacune des tables séparant les deux

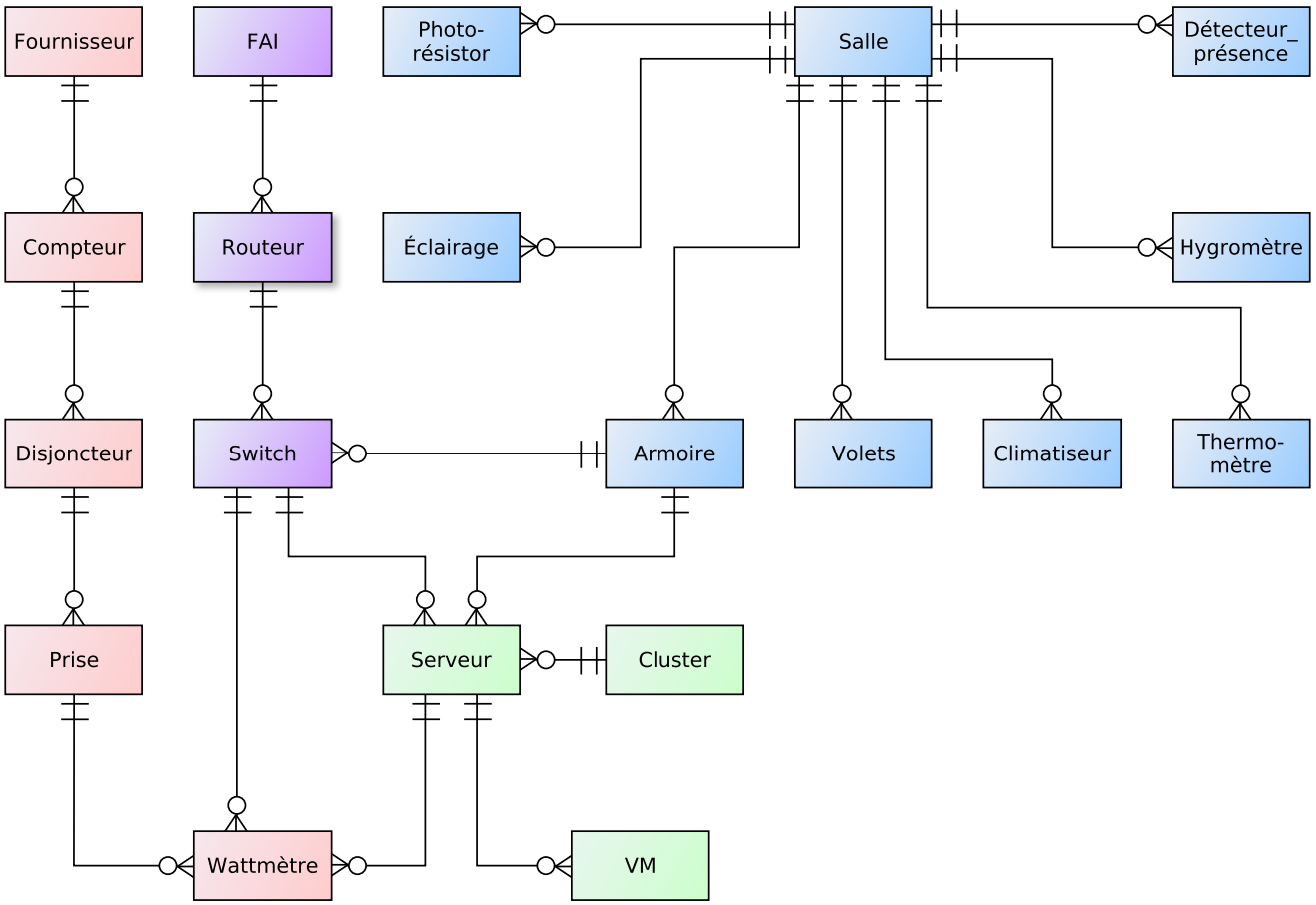


FIGURE 7.2 – Diagramme entité-association de la modélisation objet-relationnel du multiarbre

types. Par exemple, si l'on souhaite lister les disjoncteurs d'un cluster, il sera nécessaire de faire appel dans la requête aux tables *Serveur*, *Wattmètre* et *Prise*, par jointure sur chacune des clés étrangères successives.

Pour pallier à ce problème, il est possible d'établir des *vues* pour chacun des contextes constitutifs du multiarbre, qui consistent en des tables virtuelles listant les résultats d'une requête de manière dynamique. Il sera alors nécessaire d'exprimer explicitement chacune de ces vues. Par exemple, pour le contexte portant sur l'installation électrique :

```
CREATE VIEW electrical_view AS
        SELECT *
        FROM LEFT JOIN(Fournisseur f, Compteur c,
                Disjoncteur d, Prise p, Wattmetre w)
        ON (f.id = c.f_id AND c.id = d.c_id AND
                d.id = p.d_id AND p.id = w.p_id);
```

Il est alors nécessaire d'établir autant de vues qu'il existe de contextes, avec une syntaxe soit très verbeuse, soit truffée d'alias — ce qui est le cas ici — l'un comme l'autre étant source d'erreurs potentielles. De plus, de telles vues ne permettent de simplifier drastiquement que les usages mono-contexte. Une solution peut alors constituer, une fois que chacune de ces vues a été établie, en l'expression d'une vue globable du multiarbre. Reste toutefois toujours à identifier les attributs permettant les jointures entre les vues. On devine en outre que toute évolution durant la phase de conception du modèle réclame systématiquement un travail minutieux pour faire évoluer en conséquence chacune de ces vues.

Au contraire, la seule étape d'importance pour la modélisation dans SensorScript concerne la conception des relations entre les nœuds. En restreignant *forcément* la structure sous la forme d'un multiarbre, la gestion du graphe est à même de trouver implicitement un plus court chemin entre tout couple de types de nœud, comme nous l'avons vu dans le chapitre 4. Il découle de cela, ainsi que du fait que le découpage d'un

multiarbre en sous-arbres peut se faire de bien des manières possibles, que la notion de contexte n'est au final qu'une notion abstraite nous permettant — et permettant aux utilisateurs — de raisonner sur l'instance d'un multiarbre. Pour preuve, le contexte sur l'organisation physique du centre de données, du fait du nombre d'actuateurs et de capteurs reliés directement à une salle, est en réalité constitué des modèles d'au moins dix arbres, dont un pour chacun des huit capteurs et actuateurs. Cette abstraction de la notion de contexte est à nos yeux une force : si l'établissement des contextes est nécessaire à la modélisation du multiarbre, le fait qu'ils ne soient pas présents « en dur » dans SensorScript permet une certaine souplesse dans leur définition — ou comment le contexte d'organisation physique n'est pas le modèle d'un arbre unique. De la même manière, les usages *cross-context* sont ici promus puisque la modélisation ne cloisonne en rien les usages d'un utilisateur dans ses contextes d'utilisation.

Cette modélisation ne concerne toutefois que la structure du multiarbre.

7.4 Expressivité du langage

L'évaluation du langage va prendre place tout au long d'un scénario d'utilisation du réseau de capteurs tel que modélisé par le graphe 7.1. Dans un premier temps, ce scénario d'utilisation sera sujet à l'établissement de requêtes dans le langage de SensorScript et en CQL dans le but de comparer l'expressivité et la concision des requêtes dans ces deux langages. La seconde partie de l'évaluation fera évoluer le scénario vers l'automatisation de la prise de décision *via* la spécification en Java et l'appel, dans les requêtes, de méthodes d'actuation. Les solutions approchant seront alors évoquées et leurs différences avec SensorScript mises en lumière.

7.4.1 Comparaison avec CQL

Dans CQL, la notion de table est remplacée par les *streams* (flux) et les relations. Pour faire un parallèle avec le modèle objet-relationnel sous-jacent, une relation, au sens de CQL, correspond à une table et un *stream* correspond à une entrée d'une table, *à l'instant* où elle est inscrite dans la table. L'un comme l'autre peuvent donc se représenter sous la forme d'une entité, c'est à dire un ensemble prédéfini de couples clé-valeurs. Pour n'importe quel instant T, un *stream* produit au maximum une entrée, là où une relation met à disposition toutes les entrées stockées avant T.

Pour ce langage, nous considérerons dans la suite avoir à disposition des relations correspondant aux tables définies par le graphe 7.2 et aux vues spécifiées en section 7.3. En ce qui concerne la gestion du flux de données, nous considérons avoir à disposition un *stream* pour chacun des capteurs, définis selon le graphe 7.3. On établit donc que seuls les *streams* et aucune des relations contient les données mesurées par les capteurs.

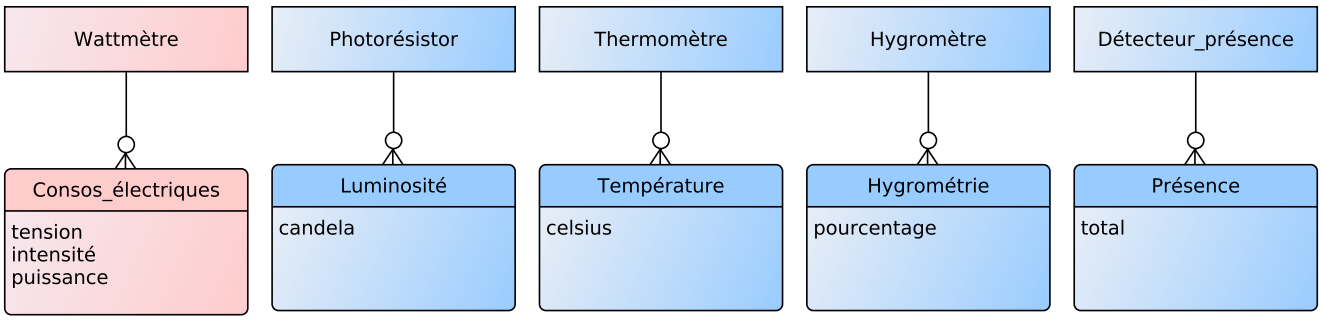


FIGURE 7.3 – Gestion des données dans les *streams* CQL

Considérons donc le scénario suivant, dans lequel interviennent chacun des quatre acteurs dont les contextes d'utilisation ont servi de base à la modélisation du multiarbre :

1. *Le responsable SI souhaite lister les serveurs :*

SensorScript	**CQL**
`Serveur`	`SELECT * FROM Serveur;`

Il s'agit ici simplement de lister les instances du type *Serveur*, la sélection de SensorScript est donc des plus simples. En CQL, cela se traduit par la liste des entrées de la relation *Serveur* et, outre la syntaxe différente, la requête est tout aussi triviale. Notons tout de même que les SQL étant centrés sur la sélection des champs avant le filtrage des entrées, il est alors nécessaire, contrairement à SensorScript, de spécifier tous les champs de la table *Serveur* dans la clause *select*.

2. *Le responsable SI souhaite lister les wattmètres du serveur* s1 *:*

SensorScript	**CQL**
`s1/Wattmetre`	`SELECT * FROM Wattmetre` `WHERE serveur_id = 's1';`

Il s'agit ici de réduire l'ensemble des wattmètres à ceux sur lesquels est branché le serveur *s1*. On considère bel et bien que plusieurs wattmètres peuvent alimenter un même serveur, précisément dans le cas de serveurs à double alimentation. Encore une fois, les deux requêtes s'écrivent très naturellement, avec toujours la nécessité de spécifier les champs à sélectionner pour CQL.

3. *Le responsable SI souhaite connaître la consommation en puissance du serveur* s1 *:*

SensorScript	**CQL**

`s1/Wattmetre.sum(puissance)`	`SELECT SUM(c.puissance)`
	`FROM Wattmetre w,`
	`Consos_electriques[Now] c`
	`WHERE serveur_id = 's1'AND`
	`w.id = c.w_id;`

Concernant SensorScript, la construction de la requête suit ici une logique itérative : avec la précédente requête nous listions les wattmètres du serveur *s1*, il s'agit simplement ici de récupérer la somme de leurs consommations.

Pour ce qui est de CQL, il est ici nécessaire de faire une jointure entre la relation *Wattmètre* et le *stream Consos_électriques*. Puisque rien n'assure que les consommations de tous les wattmètres de *s1* soient remontées au même instant T, il est nécessaire, pour CQL, d'utiliser un opérateur de conversion d'un *stream* vers une relation. Ici, la notation entre crochets permet la spécification d'une fenêtre sur laquelle construire une relation, limitée au traitement de la requête, permettant de retrouver plus d'entrée du *stream Consos_électriques* que celle émise à l'instant présent. En l'occurrence, la spécification *Now* entre crochets permet de trouver la dernière correspondance (ou *mapping*[1]) des entrées du *stream* pour chacun des wattmètres, *mapping* spécifié dans la seconde condition de la clause *where*. Ainsi on s'assure que, si chacun des wattmètres a émis au moins une entrée dans le *stream Consos_électriques*, la toute dernière entrée pour chaque wattmètre est bien liée à son wattmètre dans la relation *Wattmètre*.

La requête CQL gagne ici beaucoup plus en complexité que son équivalent SensorScript. En effet, le *mapping* à effectuer entre *stream* et relation prend ici la forme d'une jointure explicite entre les deux entités, inhérente à la gestion de l'objet-relationnel par les SQL, là où SensorScript repose sur le parcours implicite du multiarbre.

4. *Le responsable SI souhaite connaître la consommation des serveurs :*

[1] https://fr.wikipedia.org/wiki/Data_mapping

On souhaite ici généraliser le suivi de la consommation pour tous les serveurs.

SensorScript

```
Serveur:Wattmetre.sum(puissance)
```

CQL

```
SELECT SUM(c.puissance)
FROM wattmetre w,
Consos_electriques[Now] c
WHERE w.id = c.w_id
GROUP BY l.serveur_id;
```

Pour SensorScript, il s'agira de changer la sélection filtre pour l'élargir à tous les serveurs. En outre, il convient de veiller à bien partitionner les wattmètres par serveur, afin de ne pas sommer la consommation des wattmètres sur lesquels sont branchés des serveurs. Cela passe donc par l'opérateur *foreach* (correspondant au caractère *« : »*) sur le type *Serveur*.

Naturellement, l'équivalent en CQL passe par la clause *group by*. Nous pouvons constater cette fois que l'évolution des deux requêtes connaît dans les deux cas un impact comparable en terme de verbosité et de complexité. Cela s'explique facilement par le fait que l'opérateur *foreach* a été conçu en partie comme un équivalent à la clause *group by*. La différence la plus notoire porte sur le fait que cette clause porte sur un attribut dans les SQL, là où l'opérateur *foreach* s'applique directement sur des nœuds du multiarbre.

5. *Le responsable SI souhaite lister les wattmètres des serveurs hébergeant les VM :*

Le responsable SI entre ici dans une démarche d'identification des wattmètres servant à l'alimentation des services (à considérer un service par VM). Cette requête est une première étape dans cette démarche.

SensorScript

```
VM/Wattmetre
```

CQL

```
SELECT w.*
FROM Wattmetre w, logical_view l
WHERE w.id = l.wattmetre_id;
```

Les requête CQL est ici triviale, puisqu'il s'agit simplement d'opérer une jointure entre les relations *Wattmètre* et *logical_view*.

Pour SensorScript, nous pouvons voir dans le multiarbre du graphe 7.1 qu'il existe deux chemins entre les types *VM* et *Wattmètre*, qui sont :

- $VM \rightarrow Serveur \rightarrow Wattmetre$
- $VM \rightarrow Serveur \rightarrow Switch \rightarrow Wattmetre$

Cependant, du fait que le parcours entre les nœuds du multiarbre se fait toujours selon le plus court chemin, tel qu'identifié dans le modèle du multiarbre, il n'est ici pas nécessaire de spécifier le chemin permettant de ne récupérer que les wattmètres des serveurs hébergeant des VM. En contrepartie, une telle requête ne peut de fait en aucun cas lister les wattmètres des *switches* liés aux VM.

6. *Identification des* switches *utilisés par les machines virtuelles :*

Toujours dans cette démarche d'identification des wattmètres, et pour pallier à la restriction que l'on vient d'évoquer, le responsable SI souhaite dorénavant identifier les wattmètres alimentant les *switches* liés à des serveurs hébergant des machines virtuelles. Cette démarche constitue une première étape en vue de réduire les coûts énergétiques du centre de données. Elle implique de fait également les responsables réseau et développement durable. On est en effet ici dans le cas d'un usage *cross-context* puisqu'il implique à la fois l'organisation logique du centre de données et la gestion du réseau.

SensorScript

```
VM/Switch/Wattmetre
```

CQL

```
SELECT w.* FROM Wattmetre w,
network_view n, logical_view l
WHERE w.id = n.wattmetre_id AND
n.serveur_id = l.serveur_id AND
l.vm_id <> NULL;
```

Cette fois, il s'agit pour SensorScript d'identifier spécifiquement le chemin à suivre entre les VM et les wattmètres pour bien s'assurer de récupérer non pas les wattmètres des serveurs hébergeant les VM, mais les wattmètres des *switches* connectés à ces serveurs. On retrouve ici cette facilité incrémentale du langage puisqu'il s'agit simplement d'ajouter une sous-sélection pour forcer le chemin entre VM et wattmètres à passer par le type *Switch*.

En ce qui concerne CQL, on est en revanche confrontée ici aux limites de la modélisation en objet-relationnel du multiarbre, puisqu'il est alors nécessaire d'opérer de multiple jointure entre les vues correspondant aux contextes suscités et la relation *Wattmètre*. Enfin, il est également nécessaire de s'assurer que chaque entrée de la vue représentant l'organisatio logique du centre concerne bien une machine virtuelle, ce qui est fait par la vérification qu'une valeur est bien attribué au champ identifiant une machine virtuelle.

7. *L'électricien souhaite connaître la consommation énergétique totale sur chacun des disjoncteurs :*

Il s'agit ici de partitionner l'ensemble des wattmètres selon chacun des disjoncteurs en vue de sommer leurs puissances, cela passe donc pour chacun des langages par l'opérateur *foreach* et par la clause *group by* :

SensorScript

```
Disjoncteur:Wattmetre.sum(puissance)
```

CQL

```
SELECT SUM(c.puissance)
FROM electrical_view e,
Consos_electriques[Now] c
WHERE e.wattmetre_id = c.w_id
GROUP BY e.disjoncteur_id;
```

Les deux requêtes sont ici comparables à la requête 4. On va s'attarder ici à une analyse sémantique, plus que syntaxique, des différences entre les deux langages et comment il répondent aux connaissances de l'acteur concerné, ici l'électricien.

Considérons tout d'abord la requête SensorScript. Puisqu'il s'agit de partitionner selon les disjoncteurs, il convient encore de définir *quoi* partitionner. En l'occurrence, il s'agit des wattmètres. Dans la définition des contextes, on a clairement établi le wattmètre comme faisant partie du domaine de compétence de l'électricien, il est donc à même de savoir qu'il lui faut calculer la somme des puissances sur les wattmètres, ce pour chacun des disjoncteurs. Si on passe outre la *syntaxe* de SensorScript, cette requête ne fait donc intervenir aucun *concept* que l'électricien ne maîtrise pas.

La requête CQL, quant à elle, fait abstraction du type *Disjoncteur* au profit de la relation *electrical_view*, qui correspond au même contexte maîtrisé par l'électricien, ce afin de réduire le nombre de jointures. Si l'électricien reste en la matière dans un cadre qu'il maîtrise, l'information centrale du cas d'utilisation, à savoir la consommation des *disjoncteurs* ne se retrouve plus que dans la clause *group by*. Qui pis est, il est alors nécessaire à l'électricien de connaître la structure interne de la relation *electrical_view* pour savoir que le nom, l'identifiant d'un disjoncteur y constitue l'attribut $disjoncteur\_id$. Le point le plus évident concerne enfin la longueur des deux requêtes. Même en ayant cherché à réduire la verbosité de SQL sur la base d'un modèle de contextes plus adapté à l'objet-relationnel, la requête CQL souffre de cette verbosité dans la comparaison avec SensorScript.

7.4.2 Méthodes d'actuation

La suite du scénario concerne l'adressage des actuateurs par SensorScript. Cet adressage passe par la spécification en Java de méthodes d'actuations. Cela dépasse de fait le cadre du simple traitement des données, et la comparaison avec CQL n'est dès lors plus possible. Nous nous attelons de fait ici à illustrer la spécification de telles méthodes et la façon dont elles peuvent être appelés dans le langage.

8. *Le responsable développement durable souhaite gérer l'éclairage dynamique des salles :*

   Afin d'optimiser la consommation énergétique, il convient ici, lorsque quelqu'un entre dans une salle jusqu'alors vide, d'assurer une luminosité correcte dans la salle. Il conviendra alors d'ouvrir les volets si ceux-ci sont fermés ou d'allumer les éclairages de la pièce selon qu'il fasse clair ou sombre à l'extérieur. Dans ce cas de figure, en plus de suivre la luminosité à l'extérieur de la salle, il devient donc nécessaire à l'acteur de pouvoir appeler les fonctionnalités d'actuation des volets et des éclairages des salles. Cela se fait par la spécification de méthodes d'actuation appropriées.

```
public class Volets extends Nodes {
        private final InetAddress ip;
        private final int port;

        public Volets(String name, InetAddress ip, int port) {
                super("Volets", name);
                this.ip = ip;
                this.port = port;
        } // Volets(String, InetAddress, int)

        /**
         * Ouvre les volets
         */
        public void ouvrir() throws IOException {
                connectAndSend("open");
                super.attributes.set("etat", "ouvert");
        } // void ouvrir()


        /**
         * Ferme les volets
         */
        public void fermer() throws IOException {
                connectAndSend("close");
                super.attributes.set("etat", "ferme");
        } // void fermer()

        /**
         * Connecte une socket a port:ip et envoie un
         * ordre d'ouverture ou de fermeture aux volets
         *
         * @throws IOException si la socket n'a pu etre ouverte
         */
        private void connectAndSend(String order) throws IOException {
                Socket socket = new Socket(ip, port);
                new PrintStream(socket.getOutputStream()).println(order);
                socket.close();
        } // void connectAndSend(String)

} // class Volets
```

Listing 7.1 – Spécification de la classe *Volets*

Le listing 7.1 illustre la spécification de telles méthodes d'actuation pour l'ouverture et la fermeture des volets. Comme on le voit, il s'agit ici de concevoir une classe correspondant au type de nœud pour lequel on souhaite fournir une méthode d'actuation. Cette classe doit alors étendre la classe *Node*, dont les instances constituent les nœuds du multiarbre. Toute méthode publique et non statique définie dans cette classe devient alors disponible, pour ce type, dans le langage.

En pratique, lors de l'exécution, il suffit que cette classe soit présente dans le *classpath*[2] de la JVM (*Java Virtual Machine*) pour que la solution soit à même d'instancier les nœuds de type *Volets* depuis cette classe plutôt que depuis la classe *Node*, ce qui est le comportement par défaut.

De la même manière, le listing 7.2 illustre la spécification des méthodes d'allumage et d'extinction des éclairages.

```
public class Eclairage extends Nodes {

[...]

        /**
         * Allume l'eclairage
         */
        public void allumer() {
                connectAndSend("turnOn");
                super.attributes.set("etat", "allume");
        } // void allumer()


        /**
         * Eteint l'eclairage
         */
        public void eteindre() {
                connectAndSend("turnOff");
                super.attributes.set("etat", "eteint");
        } // void eteindre()

[...]

} // class Eclairage
```

Listing 7.2 – Extrait de la spécification de la classe *Eclairage*

Naturellement, la spécification de telles méthodes d'actuation est réservée aux développeurs. Ces spécifications restent toutefois ponctuelles et auront plutôt tendance à prendre place lors du processus de modélisation du multiarbre et du déploiement de l'application.

Considérons alors les deux requêtes suivantes :

```
PhotoResistor{candela > 60}/Detecteur_presence{total > 0}
        /Volets{etat = ferme}.ouvrir()

PhotoResistor{candela <= 60}/Detecteur_presence{total > 0}
        /Eclairage{etat = eteint}.allumer()
```

Elles permettent, comme attendu, lorsqu'une présence est détectée dans la salle (lorsque le total de présences détectées dépasse zéro) de décider d'ouvrir les volets ou d'allumer la lumière, selon que la luminosité extérieure suffise ou non à éclairer la salle.

Comme on le voit ici, bien que l'on raisonne sur une salle, le type *Salle* est complètement absent des deux requêtes. Bien qu'une salle assure bel et bien le parcours au plus proche entre chacun des éléments

[2] https://fr.wikipedia.org/wiki/Classpath_(java)

(photorésistor, détecteur de présence, volets et éclairage) présents dans cette salle, l'absence dans les requêtes de ce qui forme le cœur même du besoin utilisateur peut être considéré contre-productif, dans l'optique de proposer une interface orientée utilisateurs.

Une solution à cela consiste en l'appel à une méthode bien particulière des nœuds, la méthode *set*. Elle consiste en la possiblité de spécifier, pour une sélection, un nouvel attribut en fonction des résultats d'une *sous-requête*. Cette sous-requête est obligatoirement constituée d'une sous-sélection et d'un accès. En pratique, les requêtes suivantes permettent de remonter les attributs des capteurs d'une salle parmi les attributs de la salle elle-même :

```
Salle.set(total, /Detecteur_presence.sum(total))

Salle.set(candela, /Photoresistor.avg(candela))
```

Dès lors, les requêtes précédentes deviennent :

```
Salle{candela > 60 & total > 0}/Volets{etat = ferme}.ouvrir()

Salle{candela <= 60 & total > 0}/Eclairage{etat = eteint}.allumer()
```

Dans le cas où plusieurs photorésistors et détecteurs de présence équipent une même salle, de telles spécifications permettent même d'éviter la redondance de traitement que provoquent les requêtes précédentes, dans lesquels chacun de ces capteurs vient éventuellement impacter la gestion des volets et de l'éclairage.

C'est ici que prend place la composition de flux, où les flux de données des capteurs de présence d'une même salle, par exemple, viennent construire un unique flux composite portant sur le nombre de personnes présentes au total dans la salle. Enfin, si l'appel à des méthodes d'agrégation dans les conditions n'est pas permis par le langage, la possibilité de spécifier des attributs composites à partir de l'appel à ces méthodes d'agrégation permet de contourner cette restriction, puisqu'il est alors possible aux conditions d'accéder à ces attributs composites agrégés.

Dès lors, on a la possibilité d'utiliser ces nouveaux attributs dans de nouvelles requêtes :

9. *Le responsable développement durable souhaite identifier les salles vides où les lumières sont restées allumées*

   Afin de ne pas crouler sous une masse de faux positifs (une personne quitte son bureau pour quelques secondes et ne prend pas la peine d'éteindre la lumière), il s'avère qu'il est préférable de s'assurer que la salle est vide depuis un certain temps avant de considérer l'état de l'éclairage :

SensorScript

```
Salle{(total = 0, 15 min) ;
eclairage = allume}
```

CQL

```
SELECT s.* FROM Salle s,
Detecteur_presence[Range 15 min] d,
Eclairage[Now] e
WHERE s.id = d.salle_id AND
s.id = e.salle_id AND  e.etat = 'allume'
GROUP BY s.id HAVING SUM(d.total) = 0
```

En raison de l'absence d'opérateur *puis* dans CQL, on constate ici la nécessité de faire appel à deux fenêtres de temps différentes et à les conjuguer au sein d'une même requête.

7.4.3 Limites et contraintes de SensorScript

SensorScript permet, pour les cas d'utilisation étudiés dans ce scénario, de proposer des requêtes plus concises et qui, outre la syntaxe opérationnelle du langage, ne réclament pas aux acteurs de connaître plus

que ce qui a été défini comme leurs contextes d'utilisation lors de la modélisation du multiarbre. Si le langage permet cela en s'appuyant sur la gestion du multiarbre par SensorScript, notamment en ce qui concerne le parcours implicite entre les nœuds à partir de l'expression d'une requête, il n'en demeure pas moins que le langage est également contraint pas ces choix de conception.

De fait, un point non abordé reste la gestion d'un historique des données. Si les conditions temporelles permettent le suivi, pendant une période de temps, des attributs issus du flux de données, ces données sont oubliées dès que possible par le *framework*. De fait, en dehors du cadre d'une condition temporelle, rien dans le langage ne permet de raisonner, pour chaque attribut, sur autre chose que la dernière valeur remontée. Ainsi récupérer une valeur vieille d'une heure d'un attribut *a posteriori* n'est pas possible dans le langage, contrairement à ce que proposent par définition les langages basés sur le paradigme objet-relationnel, tel que CQL.

Concernant la composition d'événements, notamment la construction de flux composites, l'on vient de voir qu'elle ne peut prendre place que dans la spécification de nouveaux attributs des nœuds. Si c'est bel et bien une contrainte de SensorScript, nous considérons néanmoins ce point comme un point fort de la solution, puisqu'il s'inscrit naturellement dans la modélisation du réseau de capteurs en multiarbre et cadre de fait la composition des événements dans les contextes d'utilisation pré-établis.

8

Conclusions et travaux futurs

Sommaire

8.1 Synthèse et bilan

Le succès grandissant des réseaux de capteurs conduit à un accroissement des domaines d'application des capteurs et une multiplication des usages qui en sont faits. Dans ce contexte, la taille des réseaux de capteurs devient de plus en plus problématique quant à l'identification, au niveau des points de collecte, de la localisation et des mesures effectuées par tel ou tel capteur. Cette problématique devient d'autant plus prégnante que les profils d'utilisateur tendent à se diversifier avec les domaines d'expertise qu'adressent ces capteurs. Le besoin devient alors de permettre à tout un chacun d'identifier aisément les données qui l'intéressent sans que ces données ne lui paraissent noyées dans le flux de données incessant issu du réseau de capteurs.

Dans cette optique, nous avons établi par une étude de la littérature les différentes pistes permettant l'adaptation des structures de données aux besoins des utilisateurs. Au terme de cette étude il nous est apparu que l'inscription des capteurs et données au sein de contextes, définis comme la localisation des capteurs dans des environnements maîtrisés par l'utilisateur, offrent une solution à même de répondre au besoin exprimé. Deux autres points restent cependant en suspens.

D'une part, il est nécessaire non seulement de proposer une contextualisation des données en fonction des besoins utilisateur, mais de le faire pour *chaque* utilisateur final de la solution. En effet, les besoins exprimés par chacun de ces utilisateurs seront différents d'un profil d'utilisateur à un autre. Naturellement, certains de ces contextes peuvent être communs à plusieurs utilisateurs, de la même manière qu'un seul utilisateur peut être amené à maîtriser plusieurs contextes.

D'autre part, il est crucial de ne pas perdre de vue le fait que certains de ces utilisateurs, si ce n'est la majorité, ne sont pas des utilisateurs intensifs. Il est alors indispensable de proposer à cette gestion multi-contextes et multi-utilisateurs du réseau de capteurs une interface permettant un usage brassant le moins possible de concepts externes aux contextes concernant un utilisateur, ce pour chaque utilisateur. Dans cette optique, nous avons fait le choix de la conception d'un langage dédié.

La gestion multi-contextes du réseau de capteurs fait l'objet d'une modélisation prenant la forme d'un multiarbre. Chaque arbre constitutif du multiarbre correspond alors à un contexte d'utilisation, tels que définis par une étude préalable des besoins de chaque utilisateur vis-à-vis du réseau de capteurs. Le choix d'une telle modélisation s'appuie sur les facilités de parcours qu'elle permet : en s'appuyant sur les propriétés propres au multiarbre, on assure pour la suite un parcours implicite à même de simplifier l'interface et ce faisant l'expérience utilisateur.

À cette modélisation du réseau de capteurs s'adosse un *complex event processor*. La tâche du traitement des événements complexes est alors d'intégrer la notion de contextes dans la définition des événements, qu'ils soient atomiques ou composites. Cela passe par la définition d'un traitement selon trois points :

- la sélection de nœuds du multiarbre, sélection qui peut être filtrée en fonction d'autre nœuds ;
- l'expression de conditions sur ces sélections, permettant d'étendre le fitrage des nœuds aux données ;
- l'accès aux attributs et méthodes des nœuds, correspondant pour partie aux données brutes et aux événements composites.

Ces trois éléments constitutifs du traitement des événements complexes par SensorScript se reflètent dans les requêtes du langage dédié. C'est à ce stade que la sélection tire parti du parcours implicite dans le multiarbre puisque le filtrage des nœuds par sous-sélection s'exprime indépendamment de la distance, dans le modèle, entre les nœuds des deux sous-sélections. L'expression des conditions introduit également la notion de gestion du temps, avec la possibilité de suivre la vérification d'un prédicat durant un temps minimal.

Les réseaux de capteurs sont constitués non seulement de capteurs, mais également d'actuateurs, permettant une interaction avec l'environnement. Une contribution de cette thèse porte sur la possibilité d'adresser, directement depuis le langage dédié, ces fonctionnalités d'actuation. Cette possibilité passe par l'implé-

mentation, par des utilisateur intensifs, de méthodes dites d'actuation pour les types associés aux actuateurs dans le modèle du multiarbre.

Au travers du scénario développé en guise d'évaluation, nous avons cherché à mettre en lumière la simplicité d'utilisation du langage. Nous avons pu montrer que, comparé à un SQL, solution la plus répandue aujourd'hui pour le traitement des événements complexes, les requêtes ont tendance à être plus concises avec SensorScript. Cela devient d'autant plus vrai que les besoins auxquels répondent les requêtes s'avèrent plus complexes. De la même manière, les concepts qu'il est nécessaire d'avoir en tête lors de la rédaction d'une requête, outre l'inévitable collection d'opérateurs inhérents à la syntaxe du langage, sont en moindre nombre avec SensorScript. Pour CQL, le modèle objet-relationnel, qui sépare multiarbre et données, oblige l'utilisateur à comprendre un minimum le fonctionnement des tables et relations, en plus du multiarbre. Au contraire, et c'est tout l'intérêt d'un langage dédié, SensorScript est à même de ne faire intervenir que la notion de multiarbre et les contextes de l'utilisateur concerné. Le scénario a mis également en lumière la possibilité d'établir une requête pas à pas, en ajoutant de manière itérative les accès sur des requêtes pré-établies.

Pour toutes ces raisons, nous estimons avoir montré avec SensorScript l'intérêt d'une solution de traitement des événements complexes pour les réseaux de capteurs qui soit véritablement dédiée, dans sa conception, aux réseaux de capteurs, ce depuis la modélisation des capteurs et des données qui en sont issues jusque dans l'adressage, par les utilisateurs, du traitement des événements complexes.

8.2 Perspectives de recherche

Plusieurs points adressés par les solutions de traitement des événements complexes étudiés dans l'état de l'art n'ont pu être à ce jour implémentée dans SensorScript. Nous allons à présent nous intéresser à deux de ces points, qui pourraient faire l'objet de travaux futurs.

8.2.1 Historisation des données

La gestion d'un historique des données remontées depuis les capteurs n'est aujourd'hui pas prise en charge par SensorScript. Si la possibilité est offerte par les requêtes de déléguer le traitement de leurs résultats à des outils externes et donc de permettre le stockage en base de ces résultats par un SGBD, rien ne permet d'accéder par la suite à ces données, et donc de les conserver en vue de la composition des événements. La gestion des conditions temporelles pallie seulement en partie à ce manque et quand bien même on conserve une historisation de l'état d'un prédicat, rien ne permet en dehors de la condition temporelle d'accéder aux données spécifiant ce prédicat.

Une véritable intégration d'une historisation des données pourrait alors permettre l'application d'analyses plus étendues des données, avec par exemple l'identification par analyse statistique et numérique de comportements à moyen et long terme des mesures.

8.2.2 Traitement distribué

Pour des raisons de faisabilité, la solution actuelle ne permet la modélisation des capteurs que de manière centralisée. Nous avons toutefois commencé à envisager sur les derniers mois de la thèse une première étape vers la distribution de SensorScript, sans néanmoins avoir le temps de le mettre en place.

Cette distribution consisterait à considérer un nœud-puits principal et plusieurs nœuds-puits secondaire. Chacun des nœuds-puits secondaires serait alors une simple instance de SensorScript, dont tout ou partie des résultats des requêtes serait envoyé au nœud-puits principal. Le nœud-puits principal gérerait quant à lui des requêtes sur les données de son propre réseau de capteurs et les résultats envoyés par les nœuds-puits secondaires. Dans le cadre de la future école IMT Atlantique, la figure 8.1 illustre un tel déploiement de trois réseaux de capteurs sur chacun des trois sites de l'école : Brest, Nantes et Rennes.

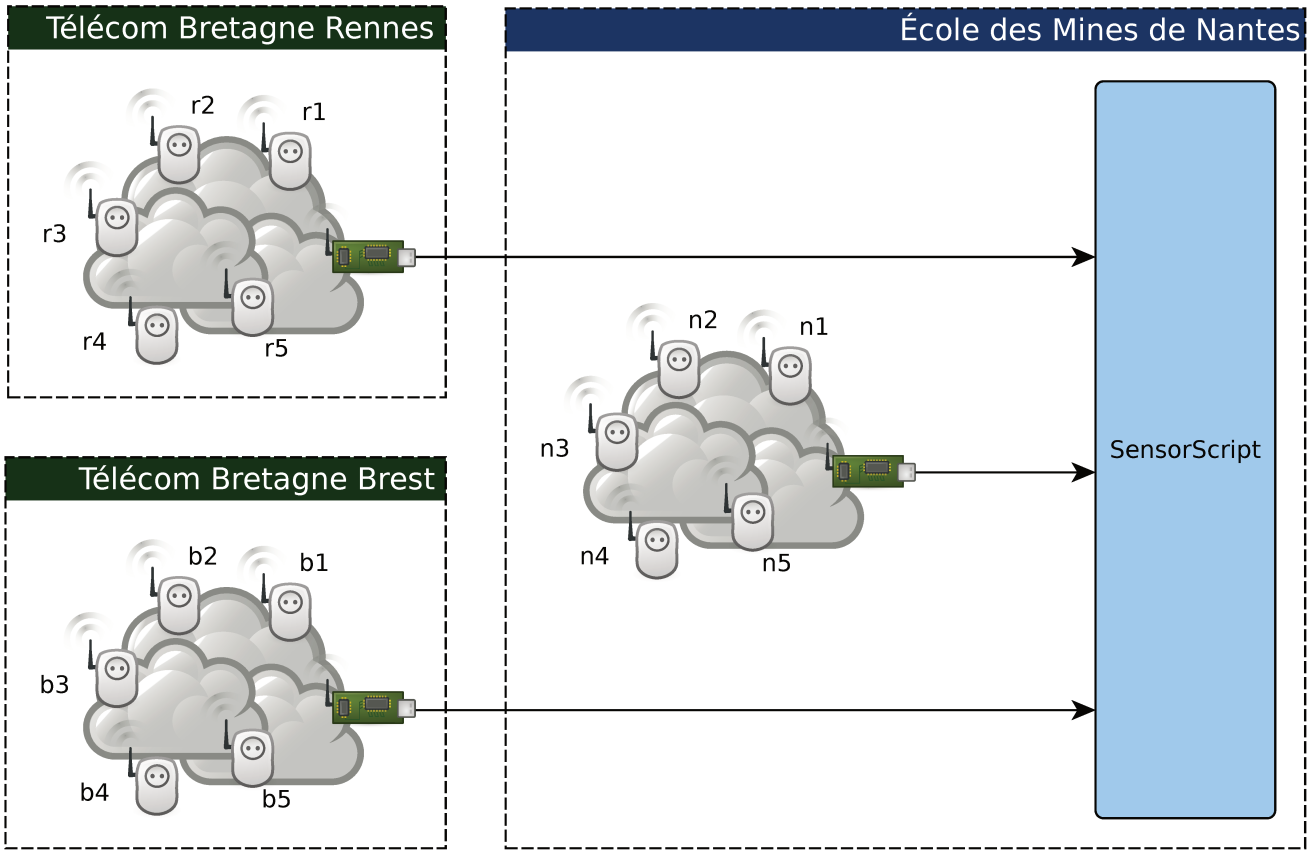


FIGURE 8.1 – Instances distribuées de SensorScript

Au-delà de ce premier pas vers la distribution de la solution, être en mesure de déléguer une partie de la gestion du traitement des événements complexes, selon un partitionnement similaire du réseau en plusieurs sous-réseaux de capteurs, devient alors envisageable, et permettrait de soulager tant la charge du nœud-puits que l'encombrement du réseau.

Annexes

Listing 9.1 – Grammaire du langage de SensorScript en JavaCC

```
options {
  STATIC = false;
  OUTPUT_DIRECTORY = "../../../java/fr/emn/sensorscript/grammar";
  LOOKAHEAD = 3;
  FORCE_LA_CHECK = true;
}

PARSER_BEGIN(SensorScriptLanguage)
package fr.emn.sensorscript.grammar;

import fr.emn.sensorscript.multitree.Node;
import fr.emn.sensorscript.Grid;
import fr.emn.sensorscript.language.Query;
import fr.emn.sensorscript.language.selection.*;
import fr.emn.sensorscript.language.condition.*;
import fr.emn.sensorscript.language.access.*;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class SensorScriptLanguage {

  public static void main(String args[])
    throws IllegalAccessException, ParseException {
    SensorScriptLanguage parser = new SensorScriptLanguage(System.in);
    parser.command();
  }

private static Map<Set<Node>, Number> numberMap(Map<Set<Node>, ? extends Object> map){
  Map<Set<Node>, Number> numberMap = new HashMap();
  for (Set<Node> nodes:map.keySet()) {
    numberMap.put(nodes, (Number) map.get(nodes));
  } // for
  return numberMap;
}

private static Map<Set<Node>, Boolean> boolMap(Map<Set<Node>, ? extends Object> map){
  Map<Set<Node>, Boolean> boolMap = new HashMap();
  for (Set<Node> nodes:map.keySet()) {
    boolMap.put(nodes, (Boolean) map.get(nodes));
  } // for
```

```
  return boolMap;
}

  private static void print(Object result) {
    StringBuilder sb = new StringBuilder();
    if (result instanceof Set) {
      Set<Node> set = (Set) result;
      sb.append("Set<Node>:");
      for (Node node : set) {
        sb.append(' ').append(node.getType()).append(':').append(node.getName()).
            append(';');
      } // for
    } else if (result instanceof Map) {
      Map<Set<Node>, ? extends Object> map = (Map) result;
      sb.append("Map<Set<Node>, ? extends Object>:");
      for (Set<Node> nodes : map.keySet()) {
        sb.append("\n\t[");
        boolean first = true;
        for (Node node : nodes) {
          if (!first || (first = false)) {
            sb.append("; ");
          } // if
          sb.append(node.getType()).append(':').append(node.getName());
        } // for
        sb.append("]: ").append(map.get(nodes));
      } // for
    } else {
      throw new Error("Unrecognized command return.");
    } // if
    System.out.println(sb.toString());
  }

private static enum NumericAggregationMethod {
  MIN, MAX, AVG, SUM
}

private static enum BooleanAggregationMethod {
  ALL, NONE
}

}

PARSER_END(SensorScriptLanguage)

/* Skip the following characters */
SKIP: {
  " " | "\t" | "\r"
}

TOKEN: {<
  EOL: "\n"
>}

TOKEN: {<
  DOT: "."
>}

TOKEN: {<
  COMMA: ","
>}
```

```
TOKEN: {<
  SLASH: "/"
>}

TOKEN: {<
  OPENING_BRACE: "{"
>}

TOKEN: {<
  CLOSING_BRACE: "}"
>}

TOKEN: {<
  NOT: "!"
>}

TOKEN: {<
  BOOLEAN_OPERATOR: "|" | "&" | "->"
>}

TOKEN: {<
  COMPARATOR: "=" | "!=" | "<" | ">" | "!<" | "!>" | "<=" | ">="
>}

TOKEN: {<
  IS_EMPTY: "isEmpty"
>}

TOKEN: {<
  CONTAINS: "contains"
>}

TOKEN: {<
  CONTAINS_ONE: "containsOne"
>}

TOKEN: {<
  SIZE: "size"
>}

TOKEN: {<
  NUMBER: ( "-" )? ( ["0"-"9"] )+ ( "." ( ["0"-"9"] )+ )?
>}

TOKEN: {<
  BOOLEAN: "true" | "false" | "TRUE" | "FALSE" | "0" | "1"
>}

TOKEN: {<
  SINGLE_QUOTE: "'"
>}

TOKEN: {<
  DOUBLE_QUOTE: "\""
>}

TOKEN: {<
  FOREACH: "foreach"
>}
```

```
TOKEN: {<
  COLON: ":"
>}

TOKEN: {<
  NAV: "nav("
>}

TOKEN: {<
  GET: "get("
>}

TOKEN: {<
  SET: "set("
>}

TOKEN: {<
  WHERE: "where("
>}

TOKEN: {<
  HAS_ATTR: "hasAttribute("
>}

TOKEN: {<
  HAS_METH: "hasMethod("
>}

TOKEN: {<
  OPENING_PARENTHESIS: "("
>}

TOKEN: {<
  CLOSING_PARENTHESIS: ")"
>}

TOKEN: {<
  MIN: "min("
>}

TOKEN: {<
  MAX: "max("
>}

TOKEN: {<
  AVG: "avg("
>}

TOKEN: {<
  SUM: "sum("
>}

TOKEN: {<
  ALL: "all("
>}

TOKEN: {<
  NONE: "none("
>}
```

```
TOKEN: {<
  TIME_UNIT: "ms" | "s" | "m" | "h" | "d" | "w"
>}

TOKEN: {<
  IDENTIFIER: ( ["a"-"z"] | ["A"-"Z"] ) (  ["a"-"z"] | ["A"-"Z"] | <NUMBER> | "-" ) *
>}

TOKEN: {<
  METHOD_IDENTIFIER: <IDENTIFIER> <OPENING_PARENTHESIS>
>}

TOKEN: {<
  STRING: <IDENTIFIER>
>}

Query command() throws IllegalAccessException :
{
  Selection s;
  // Set<Node> s;
  Access a = null;
  //Map<Set<Node>, ? extends Object> a = null;
}
{
  (<EOL> | <EOF>) {return null;} |
  (
    s = selection() ( <DOT> a = access() )? (<EOL> | <EOF>) |
    s = foreachSelection() <DOT> a = aggregationMethodAccess() (<EOL> | <EOF>)
  )
  {
  return new Query(s, a, null);
  }
}

Selection foreachSelection() :
{
  String selectionString;
  Condition condition = null;
  Selection foreachSelection = null, subSelection = null;
}
{
  (
    selectionString = identifier() ( <OPENING_BRACE> condition = condition() <CLOSING_BRACE>
    selectionString = identifier() ( <OPENING_BRACE> condition = condition() <CLOSING_BRACE>
  )
  {
    if (foreachSelection == null) {
      return new Selection(selectionString, condition, subSelection, true);
    } // if
    return new Selection(selectionString, condition, subSelection);
  }
}

Selection selection() :
{
  String selectionString;
  Condition condition = null;
  Selection subSelection = null;
}
```

```
{
  selectionString = identifier() ( <OPENING_BRACE> condition = condition()
      <CLOSING_BRACE> )? ( <SLASH> subSelection = selection() )?
  {
    return new Selection(selectionString, condition, subSelection);
  }
}

Condition condition() :
{
  Condition condition;
  String timeToWait = null;
  String timeUnit = null;
}
{
  (
    condition = simpleCondition() |
    <OPENING_PARENTHESIS> condition = simpleCondition() <COMMA> timeToWait =
        <NUMBER>.image timeUnit = <TIME_UNIT>.image <CLOSING_PARENTHESIS>
  )
  {
    if (timeToWait == null) {
      return condition;
    } // if
    long lttw = Long.parseLong(timeToWait);
    switch(timeUnit) {
      case "w":
        lttw *= 7L;
      case "d":
        lttw *= 24L;
      case "h":
        lttw *= 60L;
      case "m":
        lttw *= 60L;
      case "s":
        lttw *= 1000L;
      case "ms":
        break;
      default:
        throw new ParseException();
    }
    return new TimedCondition(condition, lttw);
  }
}

Condition simpleCondition() :
{
  boolean not = false;
  ConditionLeaf conditionLeaf;
  Condition condition = null;
  String bool_op = null;
}
{
  (not() {not = true;})? conditionLeaf = conditionLeaf() ( bool_op =
      <BOOLEAN_OPERATOR>.image condition = condition())?
  {
    if (condition == null) {
      return conditionLeaf;
    } // if
    return new ConditionNode(conditionLeaf,
```

```
        ConditionNode.Operator.fromSign(bool_op), condition);
  }
}

void not() :
{}
{
  <NOT>
}

ConditionLeaf conditionLeaf() :
{
  SimpleAccess left, right;
  String comp;
}
{
  left = simpleAccess() comp = <COMPARATOR>.image right = simpleAccess()
  {
    return new ConditionLeaf(left, ConditionLeaf.Comparator.fromSign(comp), right);
  }
}

Comparable value() :
{
  String n = null;
  Boolean b = null;
  String s = null;
}
{
  (
    n = <NUMBER>.image |
    b = bool() |
    <SINGLE_QUOTE> ( s = <STRING>.image )? <SINGLE_QUOTE> |
    <DOUBLE_QUOTE> ( s = <STRING>.image )? <DOUBLE_QUOTE>
  )
  {
    if (n != null) {
      if (n.contains(".")) {
        return Double.parseDouble(n);
      } // if
    return Long.parseLong(n);
    } // if
    if (b != null) {
      return b;
    } // if
    if (s == null) {
      return "";
    } // if
    return s;
  }
}

Access access() :
{
  Access access;
}
{
  (
    access = simpleAccess() |
    access = aggregationMethodAccess()
```

```
  )
  {
    return access;
  }
}

SimpleAccess simpleAccess() :
{
  SimpleAccess simpleAccess;
}
{
  (
    simpleAccess = attributeAccess() |
    simpleAccess = constantAccess() |
    simpleAccess = simpleMethodAccess() |
    simpleAccess = userDefinedSimpleMethodAccess()
  )
  {
    return simpleAccess;
  }
}

AttributeAccess attributeAccess() :
{
  String attribute;
}
{
  attribute = identifier()
  {
    return new AttributeAccess(attribute);
  }
}

ConstantAccess constantAccess() :
{
  Comparable value;
}
{
  value = value()
  {
    return new ConstantAccess(value);
  }
}

SimpleAccess simpleMethodAccess() :
{
  String method;
  String entry;
}
{
  (
    method = <NAV>.image |
    method = <GET>.image |
    method = <HAS_ATTR>.image |
    method = <HAS_METH>.image
  )
  entry = identifier() <CLOSING_PARENTHESIS>
  {
    if (method.equals("get(")) {
      return new AttributeAccess(entry);
```

```
    } // if
    return new SimpleMethodAccess(method.replaceAll("\\(", ""),
        Arrays.asList(new ConstantAccess(entry)));
  }
}

SimpleMethodAccess userDefinedSimpleMethodAccess() :
{
  String method;
  List<SimpleAccess> entries = new ArrayList();
  SimpleAccess entryBuffer;
}
{
  method = <METHOD_IDENTIFIER>.image ( entryBuffer =
      simpleAccess() {entries.add(entryBuffer);} )* <CLOSING_PARENTHESIS>
  {
    return new SimpleMethodAccess(method.replaceAll("\\(", ""), entries);
  }
}

AggregationMethodAccess aggregationMethodAccess() :
{
  AggregationMethodAccess.AggregationMethod aggregationMethod;
  SimpleAccess simpleAccess;
}
{
  (
    <MIN> {aggregationMethod = AggregationMethodAccess.AggregationMethod.MIN;} |
    <MAX> {aggregationMethod = AggregationMethodAccess.AggregationMethod.MAX;} |
    <SUM> {aggregationMethod = AggregationMethodAccess.AggregationMethod.SUM;} |
    <AVG> {aggregationMethod = AggregationMethodAccess.AggregationMethod.AVG;} |
    <ALL> {aggregationMethod = AggregationMethodAccess.AggregationMethod.ALL;} |
    <NONE> {aggregationMethod = AggregationMethodAccess.AggregationMethod.NONE;}
  ) simpleAccess = simpleAccess() <CLOSING_PARENTHESIS>
  {
    return new AggregationMethodAccess(aggregationMethod, simpleAccess);
  }
}

Boolean bool() :
{
  String b = null;
}
{
  b = <BOOLEAN>.image
  {
    return (b == null) ? null : Boolean.parseBoolean(b);
  }
}

String identifier() :
{
  String identifier;
}
{
  identifier = <IDENTIFIER>.image
  {
    return identifier;
  }
}
```

Listing 9.2 – Classe Grid

```
package fr.emn.sensorscript;

import com.google.common.collect.BiMap;
import com.google.common.collect.HashBiMap;

import fr.emn.sensorscript.language.Query;
import fr.emn.sensorscript.multitree.Node;

import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 *
 * @author Alexandre Garnier <alexandre.garnier at mines-nantes.fr>
 */
public class Grid {

  private final static BiMap<Node, String> nodeNames = HashBiMap.create();
  private final static Map<Node, String> nodeTypes = new HashMap();
  private final static Set<Query> queries = new HashSet();
  private final static Map<Class<? extends Node>, Map<String, List<Class>>>
      nodeSubClasses = new HashMap();

  public final static Set<Node> nav(String selection) {
    if (nodeNames.containsValue(selection)) {
      return new HashSet(Arrays.asList(
          nodeNames.inverse().get(selection)));
    } // if
    Set<Node> nodes = new HashSet<>();
    for (Node node : nodeTypes.keySet()) {
      if (nodeTypes.get(node).equals(selection)) {
        nodes.add(node);
      } // if
    } // for
    return nodes;
  }

  public final static boolean add(Node node) throws IllegalAccessException {
    if (!nodeNames.containsKey(node) && !nodeTypes.containsKey(node)) {
      nodeNames.put(node, node.getName());
      nodeTypes.put(node, node.getType());
      return update(node);
    } // if
    return false;
  }

  public final static boolean update(Node node)
      throws IllegalAccessException {
    if (nodeNames.containsKey(node) && nodeTypes.containsKey(node)) {
      for (Query query : queries) {
        query.check(node);
      } // for
```

```
      return true;
    } // if
    return false;
  }

  public final static boolean register(Query query) {
    return query != null && queries.add(query);
  }

  public final static boolean register(Class<? extends Node> nodeSubClass) {
    Map<String, List<Class>> methods = new HashMap();
    for (Method method : nodeSubClass.getDeclaredMethods()) {
      int mod = method.getModifiers();
      if (method.getAnnotation(Node.SensorScriptMethod.class) != null
          && Modifier.isPublic(mod) && Modifier.isFinal(mod)) {
        List<Class> retParamList
            = Arrays.asList(method.getReturnType());
        retParamList.addAll(Arrays.asList(method.getParameterTypes()));
        methods.put(method.getName(), retParamList);
      } // if
    }
    return !methods.isEmpty()
        && nodeSubClasses.put(nodeSubClass, methods) == null;
  }
}
```

Listing 9.3 – Classe Node

```
package fr.emn.sensorscript.multitree;

import com.google.common.collect.BiMap;
import com.google.common.collect.HashBiMap;

import fr.emn.sensorscript.Grid;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.Set;
import java.util.SortedMap;
import java.util.Stack;
import java.util.TreeMap;

/**
 *
 * @author Alexandre Garnier <alexandre.garnier at mines-nantes.fr>
 */
public class Node {

  protected final String type, name;
  protected final Map<String, Object> attributes = new HashMap();
  protected final Map<NodeObserver, String> attributeObservers
      = new HashMap();
  protected final BiMap<String, Set<Node>> children = HashBiMap.create();
  protected final BiMap<String, Node> fathers = HashBiMap.create();

  public Node(String type, String name)
      throws InstantiationException, IllegalAccessException {
    this.type = type;
    this.name = name;
    register();
  }

  private void register()
      throws InstantiationException, IllegalAccessException {
    if (!Grid.add(this)) {
      throw new InstantiationException();
    }
  }

  public String getType() {
    return type;
  }

  public String getName() {
    return name;
```

```
  }

  public final boolean addChild(Node node) throws IllegalAccessException {
    children.putIfAbsent(node.type, new HashSet<>());
    if (children.get(node.type).add(node)) {
      node.fathers.put(type, this);
      return Grid.update(node);
    } // if
    return false;
  }

  public final boolean removeChild(Node node) throws IllegalAccessException {
    if (children.get(node.type).remove(node)) {
      node.fathers.remove(type);
      return Grid.update(node);
    } // if
    return false;
  }

  private <T extends Object> Set<T> flaten(Set<Set<T>> metaset) {
    Set<T> set = new HashSet<>();
    for (Set<T> subSet : metaset) {
      set.addAll(subSet);
    } // for
    return set;
  }

  public final Object get(String attribute) {
    switch (attribute) {
      case "name":
        return name;
      case "type":
        return type;
      default:
        return attributes.get(attribute);
    }
  }

  public final Object set(String attribute, Object value)
      throws IllegalAccessException {
    long timestamp = System.currentTimeMillis();
    Object previous = attributes.put(attribute, value);
    for (NodeObserver observer : attributeObservers.keySet()) {
      String observedAttribute = attributeObservers.get(observer);
      if (observedAttribute == null
          || observedAttribute.equals(attribute)) {
        observer.update(this, value, timestamp);
      } // if
    } // for
    return previous;
  }

  public final boolean hasAttribute(String attribute) {
    switch (attribute) {
      case "name":
      case "type":
        return true;
      default:
        return attributes.containsKey(attribute);
    }
```

```
  }

  public final boolean hasMethod(String methodName) {
    for (Method method : getClass().getMethods()) {
      if (method.getName().equals(methodName)) {
        return true;
      } // if
    } // for
    return false;
  }

  public final String observe(NodeObserver observer,
      String attribute) {
    return attributeObservers.put(observer, attribute);
  }

  public final Object invoke(String method, List parameters)
      throws NoSuchMethodException, IllegalAccessException,
      IllegalArgumentException, InvocationTargetException {
    Class[] parameterTypes = new Class[parameters.size()];
    for (int i = 0; i < parameters.size(); ++i) {
      parameterTypes[i] = parameters.get(i).getClass();
    } // for
    return getClass().getMethod(method, parameterTypes).
        invoke(this, parameters.toArray());
  }

  public Set<Node> nav(String selectionString) {
    for (Set<Node> nodeSet : doDoNav(selectionString, new Stack<>()).values()) {
      return nodeSet;
    } // for
    return Collections.EMPTY_SET;
  }

  @SuppressWarnings("UnnecessaryLabelOnBreakStatement")
  private SortedMap<Integer, Set<Node>> doDoNav(String selectionString,
      Stack<String> callerTypes) {

    SortedMap<Integer, Set<Node>> result = new TreeMap();
    int depth = callerTypes.size();

    childrenNav:
    for (String childType : children.keySet()) {
      if (!callerTypes.contains(childType)) {
        if (selectionString.equals(childType)) {
          if (!result.containsKey(depth)) {
            result.put(depth, children.get(childType));
          } else {
            result.get(depth).addAll(children.get(childType));
          } // if
//          break childrenNav;
        } // if
        for (Node child : children.get(childType)) {
          if (selectionString.equals(child.name)) {
            if (!result.containsKey(depth)) {
              result.put(depth, new HashSet());
            } // if
            result.get(depth).add(child);
            break childrenNav;
          } // if
```

```
          callerTypes.push(type);
          SortedMap<Integer, Set<Node>> nav
              = child.doDoNav(selectionString, callerTypes);
          callerTypes.pop();
          for (int i : nav.keySet()) {
            if (!result.containsKey(i)) {
              result.put(i, nav.get(i));
            } else {
              result.get(i).addAll(nav.get(i));
            } // if
//            break childrenNav;
          } // for
        } // for
      } // if
    } // for

    fathersNav:
    for (Node father : fathers.values()) {
      if (!callerTypes.contains(father.type)) {
        if (selectionString.equals(father.type)
            || selectionString.equals(father.name)) {
          if (!result.containsKey(depth)) {
            result.put(depth, new HashSet());
          }
          result.get(depth).add(father);
//          break fathersNav;
        } // if
        callerTypes.push(type);
        SortedMap<Integer, Set<Node>> nav
            = father.doDoNav(selectionString, callerTypes);
        callerTypes.pop();
        for (int i : nav.keySet()) {
          if (!result.containsKey(i)) {
            result.put(i, nav.get(i));
          } else {
            result.get(i).addAll(nav.get(i));
          } // if
//          break fathersNav;
        } // for
      } // if
    } // for

    return result;
  }

  // TODO: distance notion
  private Set<Node> doNav(String selectionString, Stack<String> callerTypes) {
    for (String childType : children.keySet()) {
      if (!callerTypes.contains(childType)) {
        if (selectionString.equals(childType)) {
          return new HashSet(children.get(childType));
        } // if
        for (Node child : children.get(childType)) {
          if (selectionString.equals(child.name)) {
            return new HashSet(Arrays.asList(child));
          } // if
          callerTypes.push(type);
          Set<Node> nodeSet
              = child.doNav(selectionString, callerTypes);
          callerTypes.pop();
```

```
          if (!nodeSet.isEmpty()) {
            return nodeSet;
          } // if
        } // for
      } // if
    } // for
    for (Node father : fathers.values()) {
      if (!callerTypes.contains(father.type)) {
        if (selectionString.equals(father.type)
            || selectionString.equals(father.name)) {
          return new HashSet(Arrays.asList(father));
        } // if
        callerTypes.push(type);
        Set<Node> nodeSet = father.doNav(selectionString, callerTypes);
        callerTypes.pop();
        if (!nodeSet.isEmpty()) {
          return nodeSet;
        } // if
      } // if
    } // for
    return Collections.EMPTY_SET;
  }

  @Override
  public int hashCode() {
    int hash = 3;
    hash = 79 * hash + Objects.hashCode(this.type);
    hash = 79 * hash + Objects.hashCode(this.name);
    return hash;
  }

  @Override
  public boolean equals(Object obj) {
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final Node other = (Node) obj;
    if (!Objects.equals(this.type, other.type)) {
      return false;
    }
    return Objects.equals(this.name, other.name);
  }

  @Override
  public String toString() {
    return "Node{" + "type=" + type + ",␣name=" + name + '}';
  }

  @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.METHOD)
  public static @interface SensorScriptMethod {
  }
}
```

Listing 9.4 – Classe Query

```
package fr.emn.sensorscript.language;

import com.google.common.collect.BiMap;
import com.google.common.collect.HashBiMap;

import fr.emn.sensorscript.language.action.Action;

import fr.emn.sensorscript.language.access.Access;
import fr.emn.sensorscript.language.access.AccessObserver;
import fr.emn.sensorscript.language.access.AggregationMethodAccess;
import fr.emn.sensorscript.language.action.DefaultAction;
import fr.emn.sensorscript.language.selection.FlatSelectionSet;
import fr.emn.sensorscript.language.selection.GroupedSelectionSet;

import fr.emn.sensorscript.language.selection.Selection;
import fr.emn.sensorscript.language.selection.SelectionObserver;
import fr.emn.sensorscript.language.selection.SelectionSet;

import fr.emn.sensorscript.multitree.Node;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 *
 *
 * @author Alexandre Garnier <alexandre.garnier at mines-nantes.fr>
 */
public final class Query implements AccessObserver, SelectionObserver {

  public static long total = 0;
  public final long id = total++;
  private final static Logger logger = LoggerFactory.getLogger(Query.class);
  private final Selection selection;
  private final Access access;
  private final Set<Node> nodes = new HashSet();
  private final BiMap<Set<Node>, Access> accessesInstances
      = HashBiMap.create();
  private final BiMap<Node, AggregationMethodAccess> groupedAccessesInstances
      = HashBiMap.create();
  private final Action action;

  public Query(Selection selection, Access access, Action action)
      throws IllegalAccessException {
    this.selection = selection;
    this.access = access;
    this.action = action == null ? new DefaultAction() : action;
    observeSelection();
    SelectionSet matchingNodes = selection.getMatchingNodes();
    if (access instanceof AggregationMethodAccess) {
      if (matchingNodes instanceof FlatSelectionSet) {
        Access accessInstance
            = access.instantiate((FlatSelectionSet) matchingNodes);
```

```
        accessesInstances.put((FlatSelectionSet) matchingNodes,
            accessInstance);
        observe(accessInstance);
      } else {
        for (Node node
            : ((GroupedSelectionSet) matchingNodes).keySet()) {
          Set<Node> subNodes = ((GroupedSelectionSet) matchingNodes).
              get(node);
          Access accessInstance = access.instantiate(subNodes);
          accessesInstances.put(subNodes, accessInstance);
          groupedAccessesInstances.put(node,
              (AggregationMethodAccess) accessInstance);
          observe(accessInstance);
        } // for
      } // if
    } else if (access != null) {
      for (Node node : (FlatSelectionSet) matchingNodes) {
        Access accessInstance = access.instantiate(node);
        accessesInstances.put(new HashSet<>(Arrays.asList(node)),
            accessInstance);
        observe(accessInstance);
      } // for
    } else {
      long now = System.currentTimeMillis();
      nodes.addAll((FlatSelectionSet) matchingNodes);
//      trigger(nodes, now);
    } // if
  }

  private void trigger(Set<Node> nodes, long timestamp) {
    action.trigger(this, Collections.unmodifiableSet(nodes), null,
        timestamp);
  }

  /**
   * Access observation
   *
   * @param access
   * @param value
   * @param timestamp
   * @throws IllegalAccessException
   */
  @Override
  public void update(Access access, Object value, long timestamp)
      throws IllegalAccessException {
    action.trigger(this, Collections.unmodifiableSet(
        access.getNodes()), value, timestamp);
  }

  /**
   * Selection observation
   *
   * @param updatedNodes
   * @param removedNodes
   * @param timestamp
   * @throws java.lang.IllegalAccessException
   */
  @Override
  public void update(SelectionSet updatedNodes, SelectionSet removedNodes,
      long timestamp) throws IllegalAccessException {
```

```
    if (updatedNodes instanceof FlatSelectionSet) {
      flatUpdate((FlatSelectionSet) updatedNodes,
          (FlatSelectionSet) removedNodes, timestamp);
    } else {
      groupedUpdate((GroupedSelectionSet) updatedNodes,
          (GroupedSelectionSet) removedNodes, timestamp);
    } // if
  }

  private void flatUpdate(FlatSelectionSet updatedNodes,
      FlatSelectionSet removedNodes, long timestamp)
      throws IllegalAccessException {
    if (access instanceof AggregationMethodAccess) {
      Iterator<Access> it = accessesInstances.values().iterator();
      if (it.hasNext()) {
        AggregationMethodAccess accessInstance
            = (AggregationMethodAccess) it.next();
        if (!removedNodes.isEmpty() || !accessInstance.getNodes().
            containsAll(updatedNodes)) {
          Set<Node> newNodes
              = new HashSet<>(accessInstance.getNodes());
          accessInstance.forget(this);
          it.remove();
          newNodes.addAll(updatedNodes);
          newNodes.removeAll(removedNodes);
          accessInstance = (AggregationMethodAccess) access.
              instantiate(newNodes);
          accessesInstances.put(newNodes,
              accessInstance);
          observe(accessInstance);
        } // if
        action.trigger(this, Collections.unmodifiableSet(
            accessInstance.getNodes()), accessInstance.access(),
            timestamp);
      } // if
    } else if (access != null) {
      for (Node node : updatedNodes) {
        Access accessInstance = access.instantiate(node);
        action.trigger(this, Collections.unmodifiableSet(
            accessInstance.getNodes()), accessInstance.access(),
            timestamp);
        accessesInstances.put(new HashSet(Arrays.asList(node)),
            accessInstance);
        observe(accessInstance);
      } // for
      for (Node node : removedNodes) {
        accessesInstances.remove(new HashSet(Arrays.asList(node))).
            forget(this);
      } // for
    } else {
      // TODO: define how to manage this
      nodes.addAll(updatedNodes);
      nodes.removeAll(removedNodes);
      action.trigger(this, Collections.unmodifiableSet(nodes), null,
          timestamp);
    } // if
  }

  private void groupedUpdate(GroupedSelectionSet updatedNodes,
      GroupedSelectionSet removedNodes, long timestamp)
```

```
    throws IllegalAccessException {
  Iterator<Node> it = groupedAccessesInstances.keySet().iterator();
  if (it.hasNext()) {
    Node node = it.next();
    AggregationMethodAccess accessInstance
        = groupedAccessesInstances.get(node);
    Set<Node> subNodes = accessInstance.getNodes();
    if (removedNodes.containsKey(node)
        || (updatedNodes.containsKey(node) && !subNodes.containsAll(
            updatedNodes.get(node)))) {
      Set<Node> newNodes
          = new HashSet<>(subNodes);
      accessInstance.forget(this);
      it.remove();
      accessesInstances.inverse().remove(accessInstance);
      newNodes.addAll(updatedNodes.get(node));
      newNodes.removeAll(removedNodes.get(node));
      accessInstance = (AggregationMethodAccess) access.
          instantiate(newNodes);
      accessesInstances.put(newNodes,
          accessInstance);
      groupedAccessesInstances.put(node, accessInstance);
      observe(accessInstance);
    } // if
    action.trigger(this, Collections.unmodifiableSet(
        accessInstance.getNodes()), accessInstance.access(),
        timestamp);
  } // if
  for (Node node : updatedNodes.keySet()) {
    if (!groupedAccessesInstances.containsKey(node)) {
      AggregationMethodAccess accessInstance
          = (AggregationMethodAccess) access.instantiate(
              updatedNodes.get(node));
      action.trigger(this, Collections.unmodifiableSet(
          accessInstance.getNodes()), accessInstance.access(),
          timestamp);
      accessesInstances.put(updatedNodes.get(node), accessInstance);
      groupedAccessesInstances.put(node, accessInstance);
      observe(accessInstance);
    } // if
  }
}

private boolean observe(Access access) {
  return access.observe(this);
}

private boolean observeSelection() throws IllegalAccessException {
  return selection.observe(this);
}

public void check(Node node) throws IllegalAccessException {
  if (!selection.isGrouped()) {
    flatCheck(node);
  } else {
    groupedCheck(node);
  } // if
}

private void flatCheck(Node node) throws IllegalAccessException {
```

```
    FlatSelectionSet updatedNodes = new FlatSelectionSet(),
        removedNodes = new FlatSelectionSet();
    selection.flatCheck(node, updatedNodes, removedNodes);
    update(updatedNodes, removedNodes, System.currentTimeMillis());
  }

  private void groupedCheck(Node node) throws IllegalAccessException {
    GroupedSelectionSet updatedNodes = new GroupedSelectionSet(),
        removedNodes = new GroupedSelectionSet();
    selection.groupedCheck(node, updatedNodes, removedNodes);
    update(updatedNodes, removedNodes, System.currentTimeMillis());
  }

  @Override
  public String toString() {
    return "Query#" + id;
  }
}
```

Listing 9.5 – Classe Selection

```
package fr.emn.sensorscript.language.selection;

import com.google.common.collect.BiMap;
import com.google.common.collect.HashBiMap;

import fr.emn.sensorscript.Grid;

import fr.emn.sensorscript.language.condition.Condition;
import fr.emn.sensorscript.language.condition.ConditionObserver;

import fr.emn.sensorscript.multitree.Node;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

/**
 *
 * @author Alexandre Garnier <alexandre.garnier at mines-nantes.fr>
 */
public final class Selection implements ConditionObserver {

  private final String selectionString;
  private final Set<Node> matchingNodes = new HashSet();
  private final Condition condition;
  private final BiMap<Node, Condition> conditionInstances
      = HashBiMap.create();
  private final Selection subSelection;
  private final boolean foreach;
  private final Map<Node, Node> foreachMap = new HashMap();
  private Selection superSelection = null;
  private SelectionObserver observer;

  public Selection(String selectionString, Condition condition,
      Selection subSelection, boolean foreach) {
    this.selectionString = selectionString;
    this.condition = condition;
    this.subSelection = subSelection;
    this.foreach = foreach;
    if (this.subSelection != null) {
      luke();
    } // if
  }

  public Selection(String selectionString, Condition condition,
      Selection subSelection) {
    this(selectionString, condition, subSelection, false);
  }

  public final boolean observe(SelectionObserver observer)
      throws IllegalAccessException {
    if (superSelection != null) {
      throw new IllegalAccessException("One␣shall␣not␣observe␣a␣non-root␣"
          + "selection.");
    } // if
    if (this.observer != null) {
```

```
    return false;
  } // if
  this.observer = observer;
  if (!foreach) {
    this.observer.update(new FlatSelectionSet(addAndCheck(Grid.nav(
        selectionString))), new FlatSelectionSet(),
        System.currentTimeMillis());
  } else {
    GroupedSelectionSet newNodes = new GroupedSelectionSet();
    for (Node node : Grid.nav(selectionString)) {
      Set<Node> checkedNodes = addAndCheck(node);
      if (!checkedNodes.isEmpty()) {
        newNodes.put(node, checkedNodes);
      } // if
    } // for
    if (!newNodes.isEmpty()) {
      this.observer.update(newNodes, new GroupedSelectionSet(),
          System.currentTimeMillis());
    } // if
  } // if
  return true;
}

public final boolean add(Node node) throws IllegalAccessException {
  if (conditionInstances.containsKey(node)) {
    return false;
  } // if
  Set<Node> updatedNodes = addAndCheck(node);
  if (!updatedNodes.isEmpty()) {
    if (!foreach) {
      observer.update(new FlatSelectionSet(updatedNodes),
          new FlatSelectionSet(), System.currentTimeMillis());
    } else {
      GroupedSelectionSet groupedSelectionSet
          = new GroupedSelectionSet();
      groupedSelectionSet.put(node, updatedNodes);
      observer.update(groupedSelectionSet, new GroupedSelectionSet(),
          System.currentTimeMillis());
    } // if
  } // if
  return true;
}

public final boolean remove(Node node) throws IllegalAccessException {
  if (conditionInstances.containsKey(node)) {
    Set<Node> removedNodes = removeAndCheck(node);
    if (!removedNodes.isEmpty()) {
      if (!foreach) {
        observer.update(new FlatSelectionSet(),
            new FlatSelectionSet(removedNodes),
            System.currentTimeMillis());
      } else {
        GroupedSelectionSet groupedSelectionSet
            = new GroupedSelectionSet();
        groupedSelectionSet.put(node, removedNodes);
        observer.update(new GroupedSelectionSet(),
            groupedSelectionSet, System.currentTimeMillis());
      } // if
    } // if
    return true;
```

```
    } // if
    return false;
  }

  public final SelectionSet getMatchingNodes() {
    return subSelection == null ? new FlatSelectionSet(matchingNodes)
        : parse(subSelection.getMatchingNodes());
  }

  private SelectionSet parse(SelectionSet subSelectionSet) {
    if (!foreach) {
      return subSelectionSet;
    } // if
    if (subSelectionSet instanceof GroupedSelectionSet) {
      throw new IllegalArgumentException("One␣shall␣not␣use␣more␣than␣"
          + "one␣foreach␣in␣a␣single␣query.");
    } // if
    GroupedSelectionSet groupedSelectionSet = new GroupedSelectionSet();
    for (Node subNode : (FlatSelectionSet) subSelectionSet) {
      Node node = foreachMap.get(subNode);
      groupedSelectionSet.putIfAbsent(node, new HashSet());
      groupedSelectionSet.get(node).add(subNode);
    } // for
    return groupedSelectionSet;
  }

  @Override
  public final void update(Condition condition, boolean match, long timestamp)
      throws IllegalAccessException {
    if (!foreach) {
      flatUpdate(condition, match, timestamp);
    } else {
      groupedUpdate(condition, match, timestamp);
    } // if
  }

  private void flatUpdate(Condition condition, boolean match, long timestamp)
      throws IllegalAccessException {
    Node node = condition.getNode();
    FlatSelectionSet updatedNodes = new FlatSelectionSet(),
        removedNodes = new FlatSelectionSet();
    if (match) {
      if (matchingNodes.add(node)) {
        if (subSelection != null) {
          updatedNodes.addAll(subSelection.addAndCheck(node.nav(
              subSelection.selectionString)));
        } else {
          updatedNodes.add(node);
        } // if
      } // if
    } else if (matchingNodes.remove(node)) {
      if (subSelection != null) {
        removedNodes.addAll(subSelection.removeAndCheck(
            node.nav(subSelection.selectionString)));
      } else {
        removedNodes.add(node);
      } // if
    } // if
    if (!updatedNodes.isEmpty() || !removedNodes.isEmpty()) {
      observer.update(updatedNodes, removedNodes, timestamp);
```

```
    } // if
  }

  private void groupedUpdate(Condition condition, boolean match,
      long timestamp) throws IllegalAccessException {
    Node node = condition.getNode();
    GroupedSelectionSet updatedNodes = new GroupedSelectionSet(),
        removedNodes = new GroupedSelectionSet();
    if (match) {
      if (matchingNodes.add(node)) {
        updatedNodes.put(node, subSelection.addAndCheck(node.nav(
            subSelection.selectionString)));
      } // if
    } else if (matchingNodes.remove(node)) {
      removedNodes.put(node, subSelection.addAndCheck(node.nav(
          subSelection.selectionString)));
    } // if
    if (!updatedNodes.isEmpty() || !removedNodes.isEmpty()) {
      observer.update(updatedNodes, removedNodes, timestamp);
    } // if
  }

  private void luke() {
    subSelection.superSelection = this; // I'm your father
  }

  private Set<Node> addAndCheck(Node... nodes) throws IllegalAccessException {
    boolean hasChanged = false;
    Set<Node> subNodes = new HashSet();
    Map<Node, Node> foreachBuffer = new HashMap();
    for (Node node : nodes) {
      if (condition != null) {
        Condition conditionInstance = condition.instantiate(node);
        conditionInstances.put(node, conditionInstance);
        if (conditionInstance.matches()) {
          hasChanged = true;
          matchingNodes.add(node);
          // if selection not final
          if (subSelection != null) {
            Set<Node> navNodes = node.nav(
                subSelection.selectionString);
            if (foreach) {
              for (Node subNode : navNodes) {
                foreachBuffer.put(subNode, node);
              } // for
            } // if
            subNodes.addAll(navNodes);
          } // if
        } // if
        conditionInstance.observe(this);
      } else {
        hasChanged = true;
        matchingNodes.add(node);
        // if selection not final
        if (subSelection != null) {
          Set<Node> navNodes = node.nav(
              subSelection.selectionString);
          if (foreach) {
            for (Node subNode : navNodes) {
              foreachBuffer.put(subNode, node);
```

```
            } // for
          } // if
          subNodes.addAll(navNodes);
        } // if
      } // if
    } // for
    // if selection has changed
    if (hasChanged) {
      // if selection is final, return new node set, else call addAndCheck
      // on impacted new subSelection nodes
      if (subSelection == null) {
        return matchingNodes;
      } // if
      Set<Node> checkedNodes
          = new HashSet(subSelection.addAndCheck(subNodes));
      if (foreach) {
        checkedNodes.retainAll(foreachBuffer.keySet());
        foreachBuffer.keySet().retainAll(checkedNodes);
        foreachMap.putAll(foreachBuffer);
      } // if
      return checkedNodes;
    } // if
    // if no change, empty set
    return Collections.EMPTY_SET;
  }

  private Set<Node> addAndCheck(Set<Node> nodes)
      throws IllegalAccessException {
    return addAndCheck(nodes.toArray(new Node[nodes.size()]));
  }

  private Set<Node> removeAndCheck(Node... nodes) {
    boolean hasChanged = false;
    Set<Node> removedNodes = new HashSet<>(Arrays.asList(nodes));
    removedNodes.retainAll(matchingNodes);
    Set<Node> subNodes = new HashSet();
    for (Node node : nodes) {
      hasChanged |= matchingNodes.remove(node);
      Condition conditionInstance = conditionInstances.remove(node);
      if (conditionInstance != null) {
        hasChanged = true;
        conditionInstance.forget(this);
        // if selection not final
        if (subSelection != null) {
          subNodes.addAll(node.nav(subSelection.selectionString));
        } // if
      } // if
    } // for
    // if selection has changed
    if (hasChanged) {
      // if selection is final, return remaining nodes, else call
      // removeAndCheck on impacted subSelection nodes
      return subSelection == null ? removedNodes
          : subSelection.removeAndCheck(subNodes);
    } // if
    // if no change, empty set
    return Collections.EMPTY_SET;
  }

  private Set<Node> removeAndCheck(Set<Node> nodes) {
```

```
  return removeAndCheck(nodes.toArray(new Node[nodes.size()]));
}

public void flatCheck(Node node, FlatSelectionSet updatedNodes,
    FlatSelectionSet removedNodes) throws IllegalAccessException {
  if (selectionString.equals(node.getType())
      || selectionString.equals(node.getName())) {
    if (superSelection == null) {
      if (matchingNodes.contains(node)) {
        removedNodes.addAll(removeAndCheck(node));
      } else {
        updatedNodes.addAll(addAndCheck(node));
      } // if
    } else {
      for (Node superNode : superSelection.matchingNodes) {
        if (!superNode.nav(node.getName()).isEmpty()) {
          updatedNodes.addAll(addAndCheck(node));
          return;
        } // if
      } // for
      if (matchingNodes.contains(node)) {
        removedNodes.addAll(removeAndCheck(node));
      } // if
    } // if
  } else if (subSelection != null) {
    subSelection.flatCheck(node, updatedNodes, removedNodes);
  } // if
}

public void groupedCheck(Node node, GroupedSelectionSet updatedNodes,
    GroupedSelectionSet removedNodes) throws IllegalAccessException {
  if (selectionString.equals(node.getType())
      || selectionString.equals(node.getName())) {
    if (foreach) {
      if (superSelection == null) {
        if (matchingNodes.contains(node)) {
          Set<Node> checkedNodes = removeAndCheck(node);
          if (!checkedNodes.isEmpty()) {
            removedNodes.put(node, checkedNodes);
          } // if
        } else {
          Set<Node> checkedNodes = addAndCheck(node);
          if (!checkedNodes.isEmpty()) {
            updatedNodes.put(node, checkedNodes);
          } // if
        } // if
      } else {
        for (Node superNode : superSelection.matchingNodes) {
          if (!superNode.nav(node.getName()).isEmpty()) {
            Set<Node> checkedNodes = addAndCheck(node);
            if (!checkedNodes.isEmpty()) {
              updatedNodes.put(node, checkedNodes);
            } // if
            return;
          } // if
        } // for
        if (matchingNodes.contains(node)) {
          Set<Node> checkedNodes = removeAndCheck(node);
          if (!checkedNodes.isEmpty()) {
            removedNodes.put(node, checkedNodes);
```

```
            } // if
          } // if
        } // if
      } else {
        doGroupedCheck(node, updatedNodes, removedNodes);
      } // if
    } else if (foreach) {
      FlatSelectionSet flatUpdatedNodes = new FlatSelectionSet(),
          flatRemovedNodes = new FlatSelectionSet();
      subSelection.flatCheck(node, flatUpdatedNodes, flatRemovedNodes);
      if (!flatUpdatedNodes.isEmpty()) {
        updatedNodes.put(node, flatUpdatedNodes);
      } // if
      if (!flatRemovedNodes.isEmpty()) {
        removedNodes.put(node, flatRemovedNodes);
      } // if
    } else {
      subSelection.groupedCheck(node, updatedNodes, removedNodes);
    } // if
  }

  public boolean isGrouped() {
    return foreach || (subSelection != null && subSelection.isGrouped());
  }
}
```

Listing 9.6 – Classe Condition

```
package fr.emn.sensorscript.language.condition;

import fr.emn.sensorscript.multitree.Node;

import java.util.HashSet;
import java.util.Set;

/**
 *
 * @author Alexandre Garnier <alexandre.garnier at mines-nantes.fr>
 */
public abstract class Condition {

  private final Set<ConditionObserver> observers = new HashSet<>();
  private long timestamp = -1L;

  protected final void notify(boolean match, long timestamp)
      throws IllegalAccessException {
    this.timestamp = timestamp;
    for (ConditionObserver observer : observers) {
      observer.update(this, match, timestamp);
    } // for
  }

  public final boolean observe(ConditionObserver observer) {
    return observers.add(observer);
  }

  public final boolean forget(ConditionObserver observer) {
    return observers.remove(this);
  }

  protected final long getTimestamp() {
    return timestamp;
  }

  public abstract Condition instantiate(Node node);

  public abstract boolean matches() throws IllegalAccessException;

  public abstract Node getNode();
}
```

Listing 9.7 – Classe Access

```
package fr.emn.sensorscript.language.access;

import fr.emn.sensorscript.multitree.Node;

import java.util.Set;

/**
 *
 * @author Alexandre Garnier <alexandre.garnier at mines-nantes.fr>
 */
public interface Access {

  Object access() throws IllegalAccessException;

  Access instantiate(Node... nodes);

  Set<Node> getNodes();

  boolean observe(AccessObserver observer);

  boolean forget(AccessObserver observer);

  default Access instantiate(Set<Node> nodes) {
    return instantiate(nodes.toArray(new Node[nodes.size()]));
  }
}
```

Bibliographie

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream : The stanford data stream management system. *Book chapter*, 2004.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql : A language for continuous queries over streams and relations. In *Database Programming Languages*, pages 1–19. Springer, 2003.

[AHK94] Sheldon B Akers, Dov Harel, and Balakrishnan Krishnamurthy. The star graph : An attractive alternative to the n-cube. In *Interconnection networks for high-performance parallel computers*, pages 145–152. IEEE Computer Society Press, 1994.

[AHWY04] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. A framework for projected clustering of high dimensional data streams. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 852–863. VLDB Endowment, 2004.

[All16] LoRa Alliance. Lora technology. *Online verfügbar unter https ://www. loraalliance. org/What-Is-LoRa/Technology, zuletzt geprüft am*, 16 :2016, 2016.

[Ash09] Kevin Ashton. That 'internet of things' thing. *RFiD Journal*, 22(7) :97–114, 2009.

[ASSC02a] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE communications magazine*, 40(8) :102–114, 2002.

[ASSC02b] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks : a survey. *Computer networks*, 38(4) :393–422, 2002.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *International Conference on Mobile Data Management*, pages 3–14. Springer, 2001.

[BK09] Alejandro Buchmann and Boris Koldehofe. Complex event processing. *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 51(5) :241–242, 2009.

[CCD+] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, et al. Telegraphcq : Continuous dataflow processing for an uncertain world.

[CD+99] James Clark, Steve DeRose, et al. Xml path language (xpath) version 1.0, 1999. 19

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3) :113–124, 1956.

[Dav15] A Davies. On lpwans : Why sigfox and lora are rather different, and the importance of the business model, 2015.

[DF96] Dipankar Dasgupta and Stephanie Forrest. Novelty detection in time series data using ideas from immunology. In *Proceedings of the international conference on intelligent systems*, pages 82–87, 1996.

[DL05] Pierre-Charles David and Thomas Ledoux. Wildcat : a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7. ACM, 2005.

[DRB+13] Priscilla Dantas, Taniro Rodrigues, Thais Batista, Flávia Coimbra Delicato, Paulo F Pires, Wei Li, and Albert Y Zomaya. Lwissy : A domain specific language to model wireless sensor and actuators network systems. In *Software Engineering for Sensor Network Applications (SESENA), 2013 4th International Workshop on*, pages 7–12. IEEE, 2013.

[Esp15] EsperTech. Esper. http://www.espertech.com/esper, 2015.

[FM08] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. " O'Reilly Media, Inc.", 2008.

[FZ94] George W Furnas and Jeff Zacks. Multitrees : enriching and reusing hierarchical structure. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 330–336. ACM, 1994.

[Gar14] Gartner. Gartner says 4.9 billion connected "things" will be in use in 2015. http://www.gartner.com/newsroom/id/2905717, 2014. 18

[GMMO00] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *Foundations of computer science, 2000. proceedings. 41st annual symposium on*, pages 359–366. IEEE, 2000.

[GNT91] M Gargano, Enrico Nardelli, and Maurizio Talamo. Abstract data types for the logical modeling of complex data. *Information Systems*, 16(6) :565–583, 1991.

[GÖ03] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2) :5–14, 2003.

[GZK05] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams : a review. *ACM Sigmod Record*, 34(2) :18–26, 2005.

[HAE03] Moustafa A Hammad, Walid G Aref, and Ahmed K Elmagarmid. Stream window join : Tracking moving objects in sensor-network databases. In *Scientific and Statistical Database Management, 2003. 15th International Conference on*, pages 75–84. IEEE, 2003.

[Ham94] James Douglas Hamilton. *Time series analysis*, volume 2. Princeton university press Princeton, 1994.

[HCB02] Wendi B Heinzelman, Anantha P Chandrakasan, and Hari Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on wireless communications*, 1(4) :660–670, 2002.

[HJ08] Vincent Huang and Muhammad Kashif Javed. Semantic sensor information description and processing. In *Sensor Technologies and Applications, 2008. SENSORCOMM'08. Second International Conference on*, pages 456–461. IEEE, 2008.

[HM06] Salem Hadim and Nader Mohamed. Middleware : Middleware challenges and approaches for wireless sensor networks. *IEEE distributed systems online*, 7(3) :1, 2006.

[Hud97] Paul Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3(39-60) :21, 1997.

[IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion : a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67. ACM, 2000.

[JRS+09] Manuel Jimenez, Francisca Rosique, Pedro Sanchez, Barbara Alvarez, and Andres Iborra. Habitation : a domain-specific language for home automation. *IEEE software*, 26(4) :30–38, 2009.

[KAM11] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Eventcj : a context-oriented programming language with declarative event-based context transition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 253–264. ACM, 2011.

[KFNM04] Holger Knublauch, Ray W Fergerson, Natalya F Noy, and Mark A Musen. The protégé owl plugin : An open development environment for semantic web applications. In *International Semantic Web Conference*, pages 229–243. Springer, 2004.

[KSKL10] Oje Kwon, Yong-Soo Song, Jae-Hun Kim, and Ki-Joune Li. Sconstream : A spatial context stream processing system. In *Computational Science and Its Applications (ICCSA), 2010 International Conference on*, pages 165–170. IEEE, 2010.

[LM90] Brenda Laurel and S Joy Mountford. *The art of human-computer interface design*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[LM03] Iosif Lazaridis and Sharad Mehrotra. Capturing sensor-generated time series with quality guarantees. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 429–440. IEEE, 2003.

[LMP+05] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos : An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[LRS02] Stephanie Lindsey, Cauligi Raghavendra, and Krishna M. Sivalingam. Data gathering algorithms in sensor networks using energy metrics. *IEEE Transactions on parallel and distributed systems*, 13(9) :924–935, 2002.

[LSD10] Fei Li, Sanjin Sehic, and Schahram Dustdar. Copal : An adaptive approach to context provisioning. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2010 IEEE 6th International Conference on*, pages 286–293. IEEE, 2010.

[MAA+02] Chuck Murray, Dan Abugov, Nicole Alexander, et al. Oracle spatial user's guide and reference. *System Documentation Release*, 2(9.2), 2002.

[MBC+04] Andreas F Molisch, Kannan Balakrishnan, Chia-Chin Chong, Shahriar Emami, Andrew Fort, Johan Karedal, Juergen Kunisch, Hans Schantz, Ulrich Schuster, and Kai Siwiak. Ieee 802.15. 4a channel model-final report. *IEEE P802*, 15(04) :0662, 2004.

[MBLF05] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. Uniform resource identifier (uri) : Generic syntax. 2005.

[MCB+11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data : The next frontier for innovation, competition, and productivity. 2011.

[Mul07] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82. ACM, 2007.

[OW216] OW2. OW2 WildCAT User Guide, version 2.3.0. http://wildcat.ow2.org/userguide.html, 2016.

[PBF03] Spiros Papadimitriou, Anthony Brockwell, and Christos Faloutsos. Adaptive, hands-off stream mining. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 560–571. VLDB Endowment, 2003.

[PM12] Rémy Pottier and Jean-Marc Menaud. Btrscript : a safe management system for virtualized data center. In *ICAS 2012, The Eighth International Conference on Autonomic and Autonomous Systems*, pages 49–56, 2012.

[Pre99] Bruno R Preiss. *Data structures and algorithms*. John Wiley & Sons, Inc., 1999.

[PZCG14] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things : A survey. *Communications Surveys & Tutorials, IEEE*, 16(1) :414–454, 2014.

[RV06] Ramesh Rajagopalan and Pramod K Varshney. Data aggregation techniques in sensor networks : A survey. 2006.

[Sad07] Daniel A Sadilek. Prototyping domain-specific languages for wireless sensor networks. In *Proc. of the 4th Int. Workshop on Software Language Engineering*, pages 76–91, 2007.

[Sal13] Omran Saleh. Complex event processing in wireless sensor networks. In *Grundlagen von Datenbanken*, pages 69–74, 2013.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF : eclipse modeling framework*. Pearson Education, 2008.

[SCR10] Adel Shaeib, Paolo Cappellari, and Mark Roantree. A framework for real-time context provision in ubiquitous sensing environments. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 1083–1085. IEEE, 2010.

[SHB14] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap). Technical report, 2014.

[SLR95] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Seq : A model for sequence databases. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 232–239. IEEE, 1995.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 4. ACM, 2009.

[SS13] Omran Saleh and Kai-Uwe Sattler. Distributed complex event processing in sensor networks. In *2013 IEEE 14th International Conference on Mobile Data Management*, volume 2, pages 23–26. IEEE, 2013.

[SSJ98] Gerald Jay Sussman and Guy L Steele Jr. Scheme : A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4) :405–439, 1998.

[TL11] Kerry Taylor and Lucas Leidinger. Ontology-driven complex event processing in heterogeneous sensor networks. In *The Semanic Web : Research and Applications*, pages 285–299. Springer, 2011.

[TMT+12] Andreas Textor, Fabian Meyer, Marcus Thoss, Jan Schaefer, Reinhold Kroeger, and Michael Frey. An architecture for semantically enriched data stream mining. In *Proceedings of the First International Conference on Data Analytics, S. Bhulai, J. Zernik, and P. Dini, Eds., Barcelona, Spain*, 2012.

[TP10] Kia Teymourian and Adrian Paschke. Enabling knowledge-based complex event processing. In *Proceedings of the 2010 EDBT/ICDT Workshops*, page 37. ACM, 2010.

[Tur48] Alan M Turing. Intelligent machinery, a heretical theory. *The Turing Test : Verbal Behavior as the Hallmark of Intelligence*, 105, 1948.

[W3C09] W3C. OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref, 2009.

[W3C13] W3C. SPARQL 1.1 Overview. http://www.w3.org/TR/sparql11-overview/, 2013.

[WPT10] Matthias Woehrle, Christian Plessl, and Lothar Thiele. Rupeas : Ruby powered event analysis dsl. In *Networked Sensing Systems (INSS), 2010 Seventh International Conference on*, pages 245–248. IEEE, 2010.

[ZS02] Yunyue Zhu and Dennis Shasha. Statstream : Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.

UNIVERSITE BRETAGNE LOIRE

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Alexandre GARNIER

Langage dédié au traitement des événements complexes et modélisation des usages pour les réseaux de capteurs

Complex event processing domain-specific language and modelling of usages for sensors networks

Résumé
On assiste ces dernières années à une explosion des usages dans l'Internet des objets. La démocratisation de ce monde de capteurs est le fruit, d'une part de la baisse drastique des coûts dans l'informatique embarquée, d'autre part d'un support logiciel toujours plus mature. Que ce soit au niveau des protocoles et des réseaux (CoAP, IPv6, etc) ou de la standardisation des supports de développement, notamment sur microprocesseurs ATMEL, les outils à disposition permettent chaque jour une plus grande homogénéisation dans la communication entre des capteurs toujours plus variés. Cette diversification rassemble chaque jour des utilisateurs aux attentes et aux domaines de compétence différents, avec chacun leur propre compréhension des objets connectés. La complexification des réseaux de capteurs, confrontée à cette nécessité d'adresser des usages fondamentalement différents, pose problème. Sur la base d'un même réseau de capteurs hétéroclite, il est crucial de pouvoir répondre aux besoins de chacun des utilisateurs, sans réclamer d'eux une maîtrise du réseau de capteurs dépassant exagérément leur domaine de compétence. L'outil décrit dans ce document se propose d'adresser cette problématique au travers d'un moteur de requête dédié au traitement des données issus des capteurs. Pour ce faire, il repose sur une modélisation des capteurs au sein de différents contextes, chacun à même de répondre à un besoin utilisateur précis. Sur la base de ce modèle est mis à disposition un langage dédié pour le traitement des événements complexes issus des données mesurées par les capteurs. L'implémentation de cet outil permet en outre d'interagir avec d'éventuelles fonctionnalités d'actuation du réseau de capteurs.

Mots clés
Internet des objets, Réseaux de capteurs, Traitement des événements complexes, Langages dédiés

Abstract
Usages of the internet of things experience an exponential growth these last few years. As a matter of fact, this is the result of, on one hand the significantly lower costs in embedded computing systems, on the other hand the maturing of the software layers. From protocols and networks (CoAP, IPv6, etc) to standardization of ATMEL microcontrollers, tools at hand allow a better communication between more and more various sensors. This diversification gather every day users with different needs, expectations and fields of expertise, each one of them having his own approch, his own understanding of the connected things. The main issue concerns the complexity of the sensor networks, with regard to this necessity to address deeply different usages. Based on a single heterogeneous sensor network, it is critical to be able to meet the needs of each user, without having them to master the network beyond their own field of expertise. The tool described in this document aims at addressing this issue via a query engine dedicated to the processing of data collected from the sensors. Towards this end, it relies on a modelling of the sensors within several contexts, each of them reflecting a specific usage. On this basis a domain-specific language is provided, allowing complex event processing over the data monitored by the sensors. Furthermore, the implementation of this tool allows to interact with optional actuation functionalities of the sensor network.

Key Words
Internet of things, Sensor networks, Complex event processing, Domain-specific languages