Infrastrucuture orientée service pour le développement d'applications ubiquitaires

Julien Lancia

N° d'ordre: 3745

THÈSE

Présentée devant

L'UNIVERSITÉ BORDEAUX 1

École Doctorale de Mathématiques et Informatique

par **Julien LANCIA**

pour obtenir le grade de :

DOCTEUR

Spécialité: INFORMATIQUE

Sujet :

Infrastructure orientée service pour le développement d'applications ubiquitaires

Après avis de :

MM. :	Franck	BARBIER,	Professeur, Université de Pau	Rapporteurs
	Christophe	DONY,	Professeur, Université de Montpellier	

Devant la commission d'examen présidée par Franck BARBIER et formée de :

MM. :	Franck	BARBIER,	Professeur, Université de Pau	Rapporteurs
	Christophe	DONY,	Professeur, Université de Montpellier	
MM. :	Stéphane	TREUCHOT,	Ingénieur, THALES	Examinateurs
	Charles	CONSEL,	Professeur, INRIA	
	Marc	PHALIPPOU,	Professeur, ENSEIRB	

Remerciements

Je remercie tout d'abord les membres de mon jury :
- Franck Barbier et Christophe Dony, qui ont accepté la charge de rapporteur. Je remercie tout particulièrement Franck Barbier qui m'a fait l'honneur de présider ce jury.
- Marc Phalippou d'avoir lu en détail ce document.
- Stéphane Treuchot de m'avoir aidé a concilier la recherche académique aux préoccupations industrielles.

Je tiens ensuite à remercier mon directeur de thèse, Charles Consel, pour m'avoir accueilli dans l'équipe Phoenix et m'avoir permis de mener à bien cette thèse grâce à ses conseils avisés.

Je veux remercier tous les membres de l'équipe Phoenix avec qui j'ai pu travailler au cours de ces années de thèse, et en particulier : Petit Laurent pour ses conseils sur la thèse, la vie et la musique ; Petit Julien pour ses commentaires éclairés et ses critiques enrichissantes ; Zoé pour la touche artistique qu'elle a su mettre dans ce monde de technique ; Julien Stagiaire pour les parties de baby-foot enflammées ; Wilfried pour son professionnalisme indéfectible et ses règles toujours surprenantes ; Benjamin pour sa bonne humeur, nos longues discussions guitaristiques et ses talents d'infographiste ; Damien pour m'avoir guidé patiemment dans les arcanes de Latex ; Henner pour les après-midi surf sur la côte Bordelaise et David pour son aide durant la gestation du projet PERSEWS.

Je remercie les collocs, Anne-Marie, Ludo, pour leur soutient tout au long de ma thèse, et plus particulièrement Bruno qui m'a supporté quotidiennement durant l'épreuve de la rédaction.

Ces trois ans de thèse ont aussi été riches musicalement. Je remercie JL des Oiseaux de Passage pour la Pension, la musique et les mots. Je remercie également la Grasse Bande et Losteoporos de m'avoir permis de m'évader lorsque la pression devenait trop forte.

J'embrasse tendrement Séverine dont l'amour m'a porté tout au long de cette thèse et qui reste à mes côtés pour continuer notre voyage.

Enfin, je remercie ma famille qui m'a soutenu et a cru en moi. Mes parents, Martine et Jean, grâce a qui je suis arrivé jusque là et qui ont fait de moi ce que je suis. Ma soeur Cécile et mon frère Nicolas qui ont été là pour moi chaque fois que j'en ai eu besoin.

Infrastructure orientée service pour le développement d'applications ubiquitaires

Résumé

Un grand nombre de périphériques actuels sont dotés de connexions réseaux qui permettent d'accéder à leurs fonctionnalités au travers d'un réseau informatique. Les applications ubiquitaires visent à structurer ces fonctionnalités pour les mettre au service des utilisateurs.

Les environnements ubiquitaires sont caractérisés par une disponibilité dynamique des fonctionnalités et une hétérogénéité matérielle et logicielle des périphériques. De plus, les applications ubiquitaires doivent s'adapter en fonction du contexte des utilisateurs.

Cette thèse propose une approche pour la programmation d'applications ubiquitaires. Notre approche est basée sur une infrastructure logicielle, appelée PERSEWS, qui permet la programmation d'applications sensibles au contexte à un haut niveau d'abstraction, exploitant des fonctionnalités offertes par des périphériques hétérogènes et mobiles.

Les contributions de cette thèse sont les suivantes :

- Nous proposons une approche langage pour la programmation d'applications ubiquitaires. Cette approche est basée sur un couplage entre un modèle de contexte et un langage de règles logiques.
- Nous avons développé une infrastructure logicielle qui constitue un environnement de développement et un environnement d'exécution pour les applications ubiquitaires. L'architecture orientée service de l'infrastructure PERSEWS permet d'intégrer des fonctionnalités offertes par des périphériques hétérogènes.

Mots clés

Informatique ubiquitaire, infrastructure logicielle, architecture orientée service, Services Web sémantiques, ontologie, langage de règles

Service-oriented infrastructure for the development of ubiquitous applications

Abstract

Increasingly current devices are network enabled allowing their functionalities to be accessed through a computer network. Ubiquitous applications aim at structuring these functionalities to make them available to the users.

Ubiquitous environments are characterized by the dynamic nature of functionalities and heterogeneity of hardware and software devices. In addition, ubiquitous applications must adapt depending on the user's context.

This thesis proposes an approach to ubiquitous application programming. Our approach is based on a software infrastructure called PERSEWS, which enables programming context-sensitive applications at a high level of abstraction, using functionalities provided by heterogeneous and mobile devices .

The contributions of this thesis are as follows :

– We propose a language approach to programming ubiquitous applications. This approach is based on a coupling between a model of context and a logic rule language.
– We have developed a software infrastructure that provides a development environment and a runtime environment for ubiquitous applications. The service oriented architecture of the PERSEWS infrastructure allows to integrate functionalities provided by heterogeneous devices.

Key words

Ubiquitous computing, software infrastructure, service oriented architecture, semantic Web Services, ontology, rule language

Table des matières

Table des figures

Liste des tableaux

Sois satisfait des fleurs, des fruits, même des feuilles, si c'est dans ton jardin à toi que tu les cueilles !

Edmond Rostand, *Cyrano de Bergerac*

1 Introduction

Avec la démocratisation des équipements informatiques et la miniaturisation des composants électroniques, notre environnement quotidien est aujourd'hui saturé en périphériques informatiques. Chacun de ces périphériques fournit un ensemble de fonctionnalités que les utilisateurs exploitent pour réaliser leurs activités. Toutefois les capacités disponibles dans l'environnement restent largement sous-exploitées, chaque utilisateur exploitant les fonctionnalités des périphériques individuellement. La combinaison de ces fonctionnalités dans le but d'accomplir des tâches communes permet d'assister les utilisateurs de manière plus efficace. L'environnement est alors caractérisé par une profusion de périphériques informatiques dont les fonctionnalités sont utilisées conjointement pour assister au mieux les utilisateurs. Les environnements ainsi décrits sont appelés environnement ubiquitaires.

Un grand nombre de périphériques actuels sont dotés de connexions réseaux qui permettent d'accéder à leurs fonctionnalités au travers d'un réseau informatique. Les applications ubiquitaires visent à structurer ces fonctionnalités par l'intermédiaire du réseau pour les mettre au service des utilisateurs.

Les périphériques dans les environnements ubiquitaires peuvent être fixes ou mobiles. Les périphériques fixes sont intégrés de manière permanente à l'environnement et leurs fonctionnalités sont donc disponibles sans interruption. A l'opposé, les périphériques mobiles sont transportés par leurs propriétaires et peuvent donc entrer et sortir de l'environnement, ce qui entraîne une disponibilité dynamique des fonctionnalités dans l'environnement ubiquitaire. De nouvelles fonctionnalités offertes par ces périphériques mobiles peuvent devenir disponibles lorsqu'ils entrent dans l'environnement, et des fonctionnalités peuvent devenir indisponibles lorsque ces périphériques quittent l'environnement.

La prolifération des périphériques informatiques entraîne la présence d'une grande variété de périphériques dans un même environnement. Les fonctionnalités disponibles dans les environnements ubiquitaires sont donc offertes par des périphériques de nature disparate. Ces périphé-

riques présentent une hétérogénéité tant matérielle que logicielle, ce qui constitue un obstacle à l'utilisation conjointe de leurs fonctionnalités.

De nombreux périphériques sont équipés de capteurs (capteurs optiques, microphones, accéléromètres, etc) leur permettant d'obtenir des informations sur l'environnement physique qui les entoure. Ces informations, appelées informations de contexte, peuvent être utilisées pour déterminer les circonstances qui entourent les activités des utilisateurs. Les applications qui exploitent les informations de contexte, dites sensibles au contexte, permettent de fournir des fonctionnalités aux utilisateurs en adéquation avec le contexte dans lequel se déroulent leurs activités.

1.1 Thèse

Ce document propose une approche pour la programmation d'applications ubiquitaires. Cette approche est basée sur une infrastructure logicielle, appelée PERSEWS, qui permet de développer des applications exploitant les fonctionnalités disponibles dans un environnement ubiquitaire.

PERSEWS est une infrastructure logicielle qui exploite les technologies du Web Sémantique pour fournir un cadre de programmation et un environnement d'exécution pour les applications ubiquitaires. L'infrastructure PERSEWS permet la programmation d'applications sensibles au contexte à un haut niveau d'abstraction, exploitant des fonctionnalités offertes par des périphériques hétérogènes et mobiles.

L'infrastructure PERSEWS est fondée sur un couplage entre un modèle de contexte et un langage de programmation par règles. Le modèle de contexte définit les concepts pertinents pour la programmation d'un environnement ubiquitaire donné. Les concepts du modèle constituent les briques de base du langage de programmation d'applications ubiquitaires. Ce langage est dynamiquement étendu avec des constructions permettant d'accéder aux fonctionnalités des périphériques qui apparaissent dans l'environnement.

1.2 Contributions

Les contributions apportées par cette thèse peuvent être regroupées en deux volets. Dans un premier temps, nous proposons une approche langage pour la programmation d'applications ubiquitaires. Dans un deuxième temps, nous mettons en œuvre cette approche dans une infrastructure logicielle basée sur les architecture orientées services.

Approche langage pour la programmation d'applications ubiquitaires. Notre approche langage est basée sur un couplage entre un modèle de contexte et un langage de règles logiques. Le modèle de contexte définit les constructions pouvant être utilisées dans le langage de règles logiques. Ce langage de règles est extensible pour permettre l'ajout de nouvelles constructions lors de l'apparition de nouvelles fonctionnalités dans l'environnement. Cette approche permet de programmer des applications ubiquitaires à un haut niveau d'abstraction, en manipulant les concepts définis dans le modèle de contexte. De plus, l'extensibilité du langage permet de prendre en compte le caractère dynamique des environnements ubiquitaires.

Infrastructure PERSEWS. L'infrastructure PERSEWS constitue un environnement de développement et un environnement d'exécution pour les applications ubiquitaires. Cette infrastructure exploite un modèle de contexte pour assister le développeur dans la programmation d'applications ubiquitaires. L'infrastructure étend dynamiquement le langage lors de l'apparition de nouvelles fonctionnalités dans l'environnement et génère le code permettant à ces extensions de s'exécuter. L'architecture orientée service de l'infrastructure PERSEWS permet d'intégrer des fonctionnalités hétérogènes en terme de langage de programmation et de plate-forme d'exécution.

1.3 Plan du mémoire

Cette thèse est composée de trois parties : la première partie présente le contexte et la problématique, la deuxième partie présente notre approche, la troisième partie présente le prototype implémenté et les évaluations effectuées sur ce prototype.

Contexte et problématique. La première partie de ce mémoire porte sur le contexte de notre étude, et définit la problématique que nous entendons résoudre. Le chapitre 2 présente le concept d'informatique ubiquitaire, les caractéristiques des environnements ubiquitaires, les défis que présentent la programmation d'applications ubiquitaires et les approches existantes dans ce domaine. Dans le chapitre 3, nous présentons notre démarche pour répondre aux défis posés par la programmation d'applications ubiquitaires.

Approche proposée. Dans la deuxième partie, nous détaillons les éléments de notre approche. Le chapitre 4 présente notre solution pour la modélisation du contexte. Le chapitre 5 présente notre approche langage pour la programmation d'applications ubiquitaires. Le chapitre 6 présente l'infrastructure PERSEWS qui constitue la mise en œuvre de notre approche. Finalement, le chapitre 7 présente deux cas d'application de l'infrastructure PERSEWS dans des environnements ubiquitaires.

Résultats expérimentaux. Dans la troisième partie, nous présentons le prototype de l'infrastructure PERSEWS. Le chapitre 8 détaille les choix technologiques qui sous-tendent l'implémentation du prototype et les évaluations de performances réalisées sur ce prototype.

Première partie

Contexte et problématique

Les choses changent. Mais si vite... Est-ce que les habitudes des hommes pourront suivre ?

Isaac Asimov

2 Informatique ubiquitaire

« Les technologies les plus profondes sont celles qui disparaissent : elles se mêlent à l'élaboration de notre vie quotidienne jusqu'à en devenir indiscernables ». C'est avec ces mots que débute l'article de Marc Weiser qui décrit en 1991 sa vision de l'ordinateur du XXI e siècle et qui pose les fondements de l'informatique ubiquitaire [92]. Il y oppose l'informatique traditionnelle où un périphérique unique, l'ordinateur personnel, assiste les utilisateurs dans leurs activités à celle d'une informatique ubiquitaire où la technologie se fond dans l'environnement jusqu'à en devenir indissociable, afin d'assister les utilisateurs dans leur vie quotidienne.

L'objectif de l'informatique ubiquitaire est de transformer l'espace physique en un environnement interactif capable de répondre aux besoins de ses occupants [23]. Cet objectif présuppose une prolifération dans l'espace physique de périphériques dotés de capacités de traitement et de connexions réseaux. Si ce pré-requis était futuriste lorsque Weiser a énoncé sa vision de l'informatique ubiquitaire, la miniaturisation des composants électroniques, le développement des réseaux filaires et sans fils et la démocratisation des périphériques informatiques ont permis une pénétration des technologies informatiques dans notre environnement quotidien. Cette omniprésence technologique constitue le fondement qui permet aujourd'hui la concrétisation de la vision d'informatique ubiquitaire.

La prolifération des équipements informatiques entraîne une évolution des interactions Homme-machine d'un mode où plusieurs utilisateurs interagissent avec un équipement unique vers un mode où un utilisateur interagit avec plusieurs équipements simultanément [93]. Ce mode d'interaction nécessite une évolution dans la façon dont les utilisateurs interagissent avec les équipements informatiques et dans les modalités d'interaction des équipements informatiques entre eux. Les équipements informatiques doivent assister les utilisateurs dans l'environnement ubiquitaire de manière non-intrusive et autonome. Cette considération amène à repenser la manière dont sont programmées les applications informatiques afin de les adapter au paradigme de l'informatique ubiquitaire [73].

Pour remplir cet objectif, nous définissons dans un premier temps le concept d'environnement ubiquitaire et ses caractéristiques.

2.1 Définition d'un environnement ubiquitaire

Un environnement ubiquitaire peut s'étendre sur une large zone géographique telle une ville ou un pays [15, 53] comme sur une zone localisée dédiée à une fonction particulière tel un musée ou une maison [79, 35]. On parle dans ce dernier cas d'espace intelligent (smart space). Ces environnements sont équipés de nombreux périphériques informatiques tels que des ordinateurs personnels, des consoles de jeux ou des décodeurs numériques. De plus, les occupants de ces environnements possèdent généralement des périphériques mobiles tels que des téléphones cellulaires, des assistants personnels ou des ordinateurs portables. Ces périphériques sont dotés de connexions réseau leur permettant de communiquer entre eux et d'interfaces Homme-machine leur permettant de communiquer avec les utilisateurs de l'environnement.

Un environnement ubiquitaire se définit comme un espace physique saturé de périphériques informatiques, dans lequel une infrastructure logicielle structure les fonctionnalités offertes par les périphériques pour assister les utilisateurs dans leurs activités. Cette définition met en jeu trois composants essentiels des environnements ubiquitaires :

Les périphériques qui fournissent les fonctionnalités élémentaires dans l'environnement ubiquitaire.

L'environnement physique qui englobe toutes les entités physiques présentes dans l'environnement ubiquitaire.

L'infrastructure logicielle qui gère les interconnexions entre les éléments de l'environnement et organise les fonctionnalités présentes dans l'environnement pour assister les utilisateurs dans leurs activités.

2.1.1 Périphériques

Les différents périphériques présents dans les environnements ubiquitaires peuvent être très disparates dans leur taille et leurs capacités. Cependant, ces différents périphériques peuvent être représentés de manière uniforme suivant le modèle en couche présenté dans la figure 2.1. Ce modèle en couche distingue cinq couches : la couche matériel, la couche réseau, le système d'exploitation, l'intergiciel et la couche applicative. Ces couches sont organisées par niveau d'abstraction : chaque couche au dessus de la couche matériel offre des primitives à la couche supérieure, permettant d'abstraire la couche inférieure.

2.1.1.1 Couche matériel

La couche matériel contient les composants électroniques qui constituent le périphérique [66]. Les composants essentiels d'un périphérique informatique sont décrits ci-dessous.

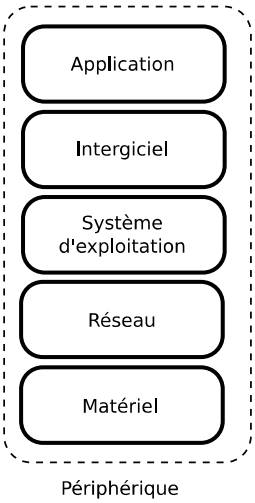

FIG. 2.1 – Modèle en couche des périphériques informatiques

Le microprocesseur est un circuit imprimé permettant d'interpréter et d'exécuter les instructions d'un programme. Les microprocesseurs peuvent être construits suivant diverses architectures qui déterminent le jeu d'instructions du processeur, la représentation binaire des données et le mode d'adressage du processeur.

La mémoire est un circuit à semi-conducteur permettant d'écrire, conserver et lire des données. Les mémoires sont caractérisées par leur capacité, leur temps d'accès et leur débit.

Les interfaces d'entrée/sortie permettent d'assurer la communication entre le microprocesseur et les organes externes d'entrée/sortie tels que les capteurs, les moniteurs, les imprimantes. Ces interfaces sont caractérisées par le type de connecteur, le type de liaison (parallèle, séquentiel) et le débit maximum supporté.

Les dispositifs de stockage , ou de mémoire de masse, permettent de stocker de grande quantités de données de manière permanente. Ces dispositifs utilisent principalement des supports magnétiques ou optiques. Ils sont caractérisés par leur capacité, leur temps d'accès et leur débit.

2.1.1.2 Couche réseau

La couche réseau désigne les connecteurs, les supports de transmission et les protocoles de communication permettant à plusieurs périphériques d'échanger des informations. Les réseaux, qui permettent d'interconnecter les périphériques, se distinguent par plusieurs caractéristiques [85].

L'étendue géographique. Les réseaux peuvent être regroupés suivant leur étendue géographique. En les classant de l'échelle la plus petite (périmètre immédiat d'une personne) à la plus grande (de l'ordre d'un pays ou d'un continent), on trouve les réseaux personnel (Personal Area Network, PAN) [9, 44], les réseaux locaux (Local Area Network, LAN) [21, 42],

les réseaux métropolitains (Metropolitan Area Network, MAN) [61, 71] et les réseaux étendus (Wide Area Network, WAN) [52].

La topologie. La topologie du réseau désigne la structure du réseau. Elle définit la manière dont les périphériques informatiques sont reliés entre eux (topologie physique) et la hiérarchie de ces périphériques (topologie logique). Les principales topologies physiques sont la topologie en bus, la topologie en étoile, la topologie en anneau, la topologie en arbre et la topologie maillée. Enfin, la topologie mixte consiste en une interconnection de différentes topologies

Le médium. Les connexions entre les différents périphériques peuvent se faire de manière filaire (câble coaxial, paire torsadée, fibre optique) ou par l'intermédiaire d'ondes électromagnétiques dans le cas des réseaux sans fils [42, 68]. On peut distinguer trois types de réseaux sans fil :

- Les réseaux avec infrastructure. Dans ce type de réseau, les périphériques se connectent à un point d'accès qui joue le rôle d'intermédiaire pour toutes les communications.
- Les réseaux sans infrastructure ou ad-hoc. Dans ce type de réseau, les périphériques se connectent directement les uns aux autres sans utiliser d'éléments d'infrastructures.
- Les réseaux ad-hoc mobiles. Dans ce type de réseaux, les connections s'effectuent directement entre périphériques sans infrastructure préexistante et les périphériques sont mobiles. Les réseaux sont donc formés dynamiquement, ce qui implique une grande variabilité dans la topologie du réseau.

Les protocoles. Les interactions entre périphériques s'effectuent en conformité avec des protocoles réseaux qui normalisent les échanges sur le réseau. Ces protocoles réseaux sont classifiés selon le modèle de référence OSI [95].

2.1.1.3 Système d'exploitation

Les périphériques informatiques forment des systèmes complexes composés de processeurs, de mémoires, de disques et d'interfaces de diverses natures. Piloter ces différents composants au niveau matériel est une tâche complexe. Le système d'exploitation est une couche logicielle qui fournit des abstractions de haut niveau pour accéder aux ressources matérielles des périphériques informatiques. Les couches logicielles supérieures accèdent aux ressources matérielles via les primitives systèmes offertes par le système d'exploitation. Les principaux rôles du système d'exploitation sont les suivants [86] :

Gestion des processus Le système d'exploitation fournit un environnement d'exécution pour les programmes. Les programmes en cours d'exécution sont généralement appelés processus. Le système d'exploitation a la tâche d'ordonnancer l'exécution des différents processus de manière optimale, et d'allouer les ressources nécessaires à l'exécution des processus ordonnancés.

Gestion des fichiers Les données stockées sont organisées en unités appelées fichiers. La partie du système d'exploitation qui gère la structure, le nommage, l'accès et la protection des fichiers est appelée système de fichiers.

Gestion de la mémoire Le gestionnaire de mémoire est la partie du système d'exploitation qui contrôle les zones de mémoire occupées et les zones de mémoire libres, qui alloue la mémoire libre aux processus qui en ont besoin et la libère ensuite.

Gestion des entrées/sorties Le système d'exploitation contrôle les interfaces d'entrée/sortie. Il émet des commandes vers les différentes interfaces d'entrée/sortie et fournit des primitives de haut niveau qui réduisent la complexité d'interaction avec les organes d'entrée/sortie.

On peut distinguer deux grandes familles de systèmes d'exploitation : les systèmes d'exploitation pour périphériques fixes et pour périphériques embarqués. En effet, les périphériques embarqués possèdent des ressources limitées en terme d'énergie, de mémoire, de stockage, d'affichage, et de puissance de calcul. De ce fait, les systèmes d'exploitation pour ce type de périphériques doivent prendre en compte leurs spécificités matérielles. Les systèmes d'exploitation les plus répandus pour périphériques fixes sont UNIX, Linux, Windows et MACOS. Dans le domaine des périphériques embarqués, les systèmes d'exploitation les plus courants sont Windows CE, PalmOS et Symbian OS.

2.1.1.4 Intergiciel

L'intergiciel est la couche logicielle qui se situe entre le système d'exploitation et les applications [7]. Le but de cette couche logicielle est de fournir des primitives de plus haut niveau que celles du système d'exploitation, afin de faciliter la programmation d'applications constituées de parties interconnectées en réseaux. De telles applications sont appelées applications réparties ou distribuées. Les fonctions des intergiciels sont les suivantes :

- Cacher la répartition, c'est-à-dire le fait qu'une application est constituée de parties interconnectées s'exécutant à des emplacements géographiques distincts.
- Cacher l'hétérogénéité des composants matériels, des systèmes d'exploitation et des protocoles de communication utilisés par les différentes parties des applications.
- Fournir des interfaces uniformes, normalisées et de haut niveau pour faciliter la construction, la réutilisation et le portage des applications.
- Fournir un ensemble de services communs réalisant des fonctions d'intêret général pour éviter la duplication des efforts et faciliter la coopération entre applications.

Emmerich [22] fournit une taxonomie des intergiciels basée sur le type d'abstractions fournies pour programmer les applications réparties. Selon cette taxonomie, on peut distinguer quatre catégories d'intergiciels :

Les intergiciels orientés transaction offrent des abstractions facilitant le développement de systèmes impliquant des transactions entre des hôtes répartis sur un réseau. Une transaction est un contrat qui garantit qu'une opération donnée maintiendra le système dans un état cohérent. Ce type d'intergiciel est souvent utilisé pour interconnecter des bases de données hétérogènes réparties sur un réseau.

Les intergiciels orientés procédure étendent le système d'appel de procédure classique pour permettre l'appel de procédure dont le corps est distant sur le réseau sans avoir à coder les détails de l'interaction. Ces intergiciels fournissent une interface pour la programmation distribuée avec laquelle les programmeurs sont familiers [81].

Les intergiciels orientés message permettent la communication par échange de fragments d'information appelés messages. Ils peuvent être classés en deux catégories suivant le mode d'échange de message utilisé. Ce mode d'échange peut être par passage de message ou par souscription/publication de message. Les intergiciels basés sur le mode passage de message fournissent une abstraction de queue de message accessible via le réseau. Les différentes parties constituant une application répartie peuvent communiquer entre elles par l'intermédiaire de ces queues de messages. Les intergiciels basés sur le mode publication/souscription de message fournissent des primitives permettant de souscrire à un bus d'information et de publier des messages sur ce bus d'information. Lorsqu'un message est publié sur le bus, tous les souscripteurs reçoivent ce message [62].

Les intergiciels orientés objet permettent d'accéder localement à des objets distants comme si ils étaient dans l'espace d'adresse local de l'appelant. Les intergiciels orientés objets permettent de bénéficier des techniques orientées objets (encapsulation, réutilisation, héritage, polymorphisme) pour la programmation d'applications réparties [31, 30].

Les intergiciels orientés service sont une classe d'intergiciels étant apparue récemment, aussi ne figure-t-elle pas dans la taxonomie d'Emmerich. Les intergiciels orientés service permettent la programmation d'applications réparties par assemblage de services dans une architecture orientée service. Les services sont des entités logicielles dont l'interface est bien définie, dont les modalités d'accès réseaux sont standardisées et qui peuvent être invoqués sans connaissance préalable des détails technologiques d'implémentation du service. La figure 2.2 présente les acteurs impliqués dans une architecture orientée service et leurs interactions. Les services sont hébergés par des fournisseurs de services et invoqués par des consommateurs de services. Un annuaire de services permet aux fournisseurs de services de publier leurs services, afin que ceux-ci puissent être découverts, et aux consommateurs de découvrir les services. Une fois qu'il a découvert un service, un consommateur peut invoquer ce service. Les intergiciels orientés service fournissent le support aux programmeurs permettant de déployer, de publier, de découvrir et d'invoquer les services dans une architecture orientée service [12].

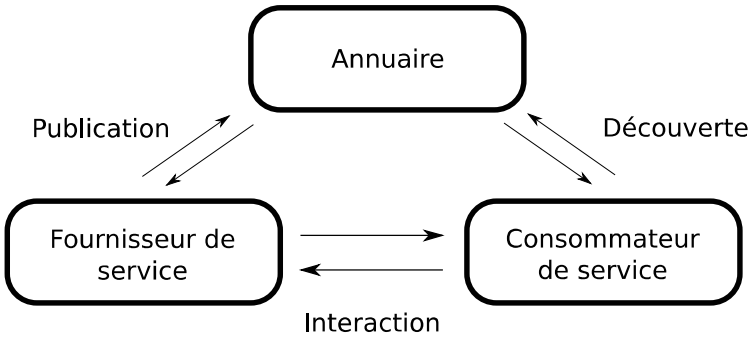

FIG. 2.2 – Acteurs et interactions dans une achitecture orientée service

Les architecture orientées service et le intergiciels orientés service constituent une avancée importante dans le domaine de la programmation d'applications réparties. En effet, le paradigme

orienté service offre une indépendance vis-à-vis des langages de programmation et des plates-formes d'exécution car l'invocation d'un service ne nécessite pas de connaissance sur les détails technologiques de son implémentation. Dans la suite de ce document, les entités logicielles constituant une application répartie seront désignées sous le terme de service.

2.1.1.5 Couche applicative

La couche applicative désigne l'ensemble des applications qui s'exécutent au-dessus du système d'exploitation dans le cas d'applications locales ou au-dessus de l'intergiciel dans le cas d'applications distribuées. Ces applications sont programmées à l'aide de langages de programmation grâce auxquels les programmeurs peuvent définir le comportement des applications [78]. Il existe de nombreux langages de programmations, qui peuvent être regroupés selon le paradigme de programmation utilisé [91] : impératif, fonctionnel, logique et objet.

2.1.1.6 Synthèse

Les environnements ubiquitaires sont saturés de périphériques informatiques qui fournissent des fonctionnalités au travers des services qu'ils exécutent. Ces périphériques, bien que possédant un structure commune (cf. figure 2.1), présentent une hétérogénéité tant au niveau matériel que logiciel.

2.1.2 Environnement physique

L'environnement physique désigne l'ensemble des utilisateurs et des périphériques présents dans un environnement ubiquitaire ainsi que les conditions physiques (température, luminosité, localisation, ...) de cet environnement. Ces informations peuvent être utilisées par les applications pour offrir des services pertinents aux utilisateurs. Par exemple, une information peut être présentée à l'utilisateur sous une forme textuelle si le niveau de bruit est élevé dans son environnement, et sous une forme sonore si le niveau de bruit est bas. La représentation des données provenant de l'environnement physique est appelée contexte, et son exploitation dans le cadre de l'exécution d'une application est appelée sensibilité au contexte.

2.1.2.1 Contexte

Dans le premier article évoquant le concept de contexte dans le domaine informatique, Schilit et Theimer [75] définissent le contexte comme "la localisation, l'ensemble des personnes et objets proches, ainsi que le changement de ces objets au cours du temps". Cette définition est raffinée par Schmidt et al. [76] qui proposent une organisation hiérarchique des informations contextuelles. Au plus haut niveau, on peut distinguer deux types d'informations contextuelles que sont les informations liées aux facteurs humains et les informations liées aux facteurs physiques.

Les facteurs humains peuvent être structurés en trois catégories :

- les informations sur les utilisateurs tels que leurs habitudes, leur état émotionnel, leur condition physique ;
- l'environnement social qui inclut la co-localisation avec d'autres utilisateurs et les interactions entre utilisateurs ;
- les tâches de l'utilisateur qui désignent les activités dans lesquelles il est impliqué et les buts qu'il cherche à réaliser.

De la même manière, les facteurs physiques peuvent être classifiés en trois catégories :

- les conditions physiques telles que la luminosité, la pression, l'accélération ;
- l'infrastructure qui désigne les ressources matérielles, les connexions réseaux disponibles, la bande passante, etc ;
- la localisation.

Si l'utilisation du contexte lors de la communication entre êtres humains se fait de manière efficace et implicite, l'exploitation de ces informations par des applications informatiques soulève un certain nombre de problèmes [74].

- Les informations de contexte doivent être abstraites afin d'être exploitées par une application. Par exemple, un système d'identification par marqueurs RFID fournit des identifications numériques alors qu'une application sera plus à même de traiter un nom d'utilisateur ou une localisation.
- Les informations de contexte sont obtenues de sources hétérogènes et distribuées. De plus, ces informations peuvent être erronées car elles sont obtenues par des capteurs qui présentent un degré d'incertitude. Par exemple, la localisation dans un bâtiment peut mettre en jeu des capteur RFID, des capteurs de mouvement et des systèmes de traitement d'images répartis dans le bâtiment. Le positionnement des utilisateurs peut nécessiter de combiner les résultats de ces différentes sources.
- Les informations de contexte sont dynamiques. Les utilisateurs évoluent dans l'environnement, ce qui modifie leur environnement social et leur localisation. Les périphériques que portent les utilisateurs sont également mobiles, ce qui modifie l'infrastructure. De plus, le but des utilisateurs peut évoluer au cours du temps en fonction des tâches qu'ils ont à accomplir. Les conditions physiques de l'environnement changent naturellement avec le temps, par exemple entre la nuit et le jour, ou au cours des déplacements des utilisateurs, par exemple entre des zones d'intérieur et d'extérieur. Les applications doivent détecter et s'adapter au changement constant de ces informations.

2.1.2.2 Sensibilité au contexte

La première définition des applications sensibles au contexte (context-aware) est donnée par Schilit et Theimer [75] qui les définissent comme des applications qui s'adaptent à leur lieu d'exécution, à l'ensemble des personnes et objets environnants, ainsi qu'au changement de ces objets au cours du temps. Une définition plus générale est donnée par Dey et al. [1] qui définissent un système sensible au contexte comme un système qui utilise le contexte pour fournir des informations et/ou des services pertinents à l'utilisateur, la pertinence dépendant de la tâche de l'utilisateur. Cette définition englobe les définitions de la sensibilité au contexte trouvées dans la littérature [2, 14, 75].

La majorité des systèmes ubiquitaires utilisent les informations de contexte provenant de l'environnement pour adapter leur comportement aux besoins de l'utilisateur. L'exploitation des informations de contexte peut prendre plusieurs formes qui peuvent être classifiées comme suit [75].

- La sélection par proximité consiste à sélectionner les objets à proximité de l'utilisateur afin de les rendre accessibles au travers d'une interface utilisateur.
- La reconfiguration contextuelle dynamique consiste à reconfigurer le système en ajoutant de nouveaux composants, en supprimant des composants existants ou en modifiant les connexions entre composants.
- Les requêtes d'informations et les commandes contextuelles qui produisent des résultats différents en fonction du contexte dans lequel elles sont émises.
- Les actions déclenchées par le contexte qui prennent la forme de règles conditionnelles définissant la manière dont l'adaptation doit être réalisée.

2.1.2.3 Modèle du contexte

Les informations contextuelles sont hétérogènes, distribuées et dynamiques. L'exploitation et le partage de ces informations par des applications dans un environnement ubiquitaire nécessite de définir un modèle commun de représentation de ces données. Le modèle du contexte permet de formaliser les différents concepts qui caractérisent un environnement ubiquitaire et facilite le partage d'informations hétérogènes entre les acteurs de l'environnement. Strang et Linnhoff-Popien [83] fournissent une classification des méthodes de modélisation les plus pertinentes basée sur les structures de données utilisées pour représenter les informations contextuelles.

- Les modèles clé-valeur possédent la structure de données la plus simple pour représenter les informations contextuelles. Elle est constituée d'une paire de valeurs dont l'une est un identifiant et l'autre une valeur associée à cet identifiant. Les informations contextuelles représentées suivant le modèle clé-valeur sont faciles à gérer, cependant le manque de structure de ce modèle ne permet pas d'organiser les informations de manière efficace.
- Les modèles basés sur les schémas de balises possèdent une structure de données hiérarchique sous forme de balises possédant des attributs et des valeurs. Ces modèles sont souvent construits sur le langage XML [28].
- Les modèles graphiques permettent de modéliser les informations contextuelles de manière visuelle. Une approche répandue pour la modélisation graphique est le langage UML [11] qui possède une composante graphique au travers des diagrammes UML. La modélisation UML des informations contextuelles permet de décrire des structures de données sophistiquées tout en bénéficiant de l'accessibilité conférée par son aspect visuel.
- Les modèles orientés objets utilisent les techniques de la programmation orientée objet pour modéliser les informations de contexte. Ces modèles bénéficient des propriétés du paradigme orienté objet : encapsulation, réutilisation, héritage, polymorphisme. L'accès aux informations de contexte et le traitement de contexte se font au travers d'interfaces bien définies.
- Les modèles basés sur la logique possèdent un haut niveau de formalisme car ils sont basés sur la logique de premier ordre. Ces modèles sont définis à partir d'assertions qui

représentent les informations contextuelles et de règles logiques. Un système d'inférence permet d'établir de nouvelles assertions à partir des règles logiques, augmentant ainsi la base d'informations contextuelles.
– Les modèles basés sur les ontologies modélisent les informations contextuelles sous forme de concepts et de relations entre ces concepts. Les ontologies offrent un haut niveau d'expressivité et de formalisme. De plus, comme pour les modèles basés sur la logique, des techniques d'inférence peuvent être appliquées aux informations modélisées.

2.1.2.4 Synthèse

Les informations caractérisant l'environnement physique (informations de contexte), proviennent de sources hétérogènes et distribuées, et sont dynamiques. Les systèmes ubiquitaires doivent prendre en compte ces informations pour assister les utilisateurs dans leurs activités. L'exploitation et le partage d'informations contextuelles dans un environnement ubiquitaire nécessite de définir un modèle commun de représentation de ces informations, appelé modèle de contexte.

2.1.3 Infrastructure logicielle

Hong et al. [54] définissent une infrastructure logicielle comme "un ensemble bien établi de technologies omniprésentes, fiables et accessibles qui agissent comme une base pour d'autres systèmes". Dans un environnement ubiquitaire, les périphériques fournissent un ensemble de services élémentaires. L'infrastructure logicielle permet de programmer des applications ubiquitaires en structurant ces services afin d'assister les utilisateurs dans leurs activités. L'hétérogénéité et le caractère dynamique des environnements ubiquitaires posent un certain nombre de défis concernant la programmation d'applications ubiquitaires. Les propriétés intrinsèques de l'infrastructure ubiquitaire doivent permettre de répondre à ces défis.

2.1.3.1 Propriétés des infrastructures logicielles

Les services d'un environnement ubiquitaire s'exécutent sur des périphériques hétérogènes tant au niveau matériel que logiciel, aussi l'infrastructure logicielle doit gérer l'hétérogénéité. Ces périphériques sont mobiles, et peuvent donc apparaître et disparaître dans l'environnement. L'infrastructure doit donc être capable de découvrir les services qui apparaissent dans l'environnement, et de s'adapter pour gérer la disparition des services. L'infrastructure logicielle doit permettre de développer et déployer simplement des applications ubiquitaires. Finalement, l'infrastructure doit être sensible au contexte afin de fournir des services pertinents aux utilisateurs, en fonction de leur environnement physique.

Gestion de l'hétérogénéité Comme présenté dans la section 2.1.1, les services dans un environnement ubiquitaires sont écrits avec différents langages de programmation, s'exécutent sur différents systèmes d'exploitation et plates-formes matérielles. L'infrastructure logicielle doit prendre en charge l'assemblage de services hétérogènes afin qu'ils puissent coopérer et interagir pour accomplir un but commun.

Découverte de services La découverte de services est le mécanisme par lequel les fournisseurs de services peuvent annoncer les services qu'ils hébergent et les consommateurs de services peuvent rechercher et sélectionner des services. La découverte de services est basée sur des protocoles de découverte de services qui normalisent les échanges réseaux entre les fournisseurs et les consommateurs de services. Les protocoles de découverte de services peuvent être centralisés [34, 47, 65] ou distribués [57, 77]. Les protocoles de découverte de services centralisés nécessitent la présence d'un ou plusieurs serveurs centralisés appelés annuaires pour gérer les annonces des producteurs et les requêtes des consommateurs. Les protocoles de découverte de services distribués utilisent des mécanismes de diffusion ou de multidiffusion pour propager les annonces et requêtes de services. La découverte de services permet à l'infrastructure logicielle d'obtenir des informations sur les services disponibles dans l'environnement.

Capacité d'adaptation L'adaptation est nécessaire afin d'obvier au caractère dynamique des environnements ubiquitaires. La mobilité des utilisateurs et des périphériques qui pénètrent et quittent l'environnement se traduit par une disponibilité dynamique des services. L'infrastructure a pour rôle de prendre en charge l'adaptation en modifiant l'assemblage de services composant une application par ajout, suppression ou remplacement de services dans cet assemblage. Les intentions des utilisateurs sont également dynamiques. Lorsqu'un utilisateur veut réaliser une nouvelle activité, l'infrastructure doit prendre en charge l'assemblage de services à la volée pour fournir une application supportant l'utilisateur dans son activité.

Développement et déploiement Le nombre et la diversité des services dans les environnements ubiquitaires nécessitent des méthodes pour développer ces services et les déployer rapidement dans l'infrastructure logicielle. Le développement et le déploiement des applications ubiquitaires doit également se faire de manière simple. Les capacités intrinsèques de l'infrastructure doivent permettre de simplifier la programmation des applications ubiquitaires en évitant au programmeur de gérer des aspects comme la découverte de services ou l'adaptation. La rapidité et la simplicité de développement des applications reposent également sur le modèle de programmation des applications géré par l'infrastructure. En particulier, le modèle de programmation par tâche [3, 80] permet de spécifier la logique applicative des applications ubiquitaires sous forme de tâche, qui est l'expression haut niveau des intentions des utilisateurs. L'infrastructure logicielle gère alors l'assemblage des services permettant d'accomplir la tâche.

Sensibilité au contexte Comme présenté dans la section 2.1.2.2, l'infrastructure logicielle doit être en mesure de réagir aux informations provenant du contexte. Le traitement de ces information contextuelles repose sur le modèle de contexte qui permet à l'infrastructure de manipuler des informations homogènes et de haut niveau.

2.1.3.2 Acteurs dans la programmation des environnements ubiquitaires

La programmation des environnements ubiquitaires fait intervenir un certain nombre d'acteurs qui ont chacun un rôle propre. Les différentes phases intervenant dans ce type de pro-

grammation sont celles de la conception du modèle, de la conception de services et de la programmation de l'environnement. La figure 2.3 présente les différents acteurs intervenant dans la programmation des environnements ubiquitaires, ainsi que les éléments de l'environnement ubiquitaire sur lesquels ils interviennent.

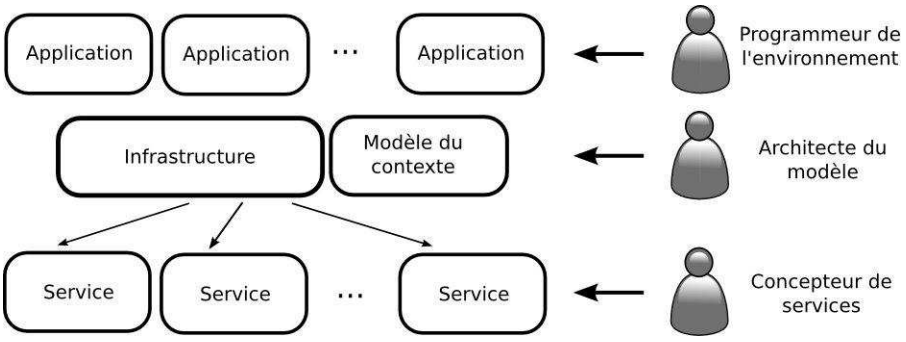

FIG. 2.3 – Acteurs intervenant dans la programmation des environnements ubiquitaires

L'architecte du modèle a pour rôle de décrire de manière formelle les concepts de l'environnement physique qui sont pertinents pour les utilisateurs de l'environnement. L'architecte du modèle doit avoir une bonne connaissance du domaine d'application pour lequel l'environnement ubiquitaire est conçu afin de cerner les concepts pertinents pour la spécification de tâches dans ce domaine.

Les concepteurs de services développent les services fournis par les périphériques de l'environnement. Ces services sont développés et déployés de manière indépendante sur chaque périphérique et peuvent être hétérogènes si l'infrastructure le supporte. Le concepteur de service peut développer les services spécifiquement pour l'infrastructure logicielle ou réutiliser des services existants. Dans ce dernier cas, il développe uniquement la couche logicielle qui permet d'intégrer le service existant dans l'infrastructure.

Le programmeur de l'environnement développe les applications qui permettent d'assister les utilisateurs dans leurs activités. Selon le niveau d'abstraction que présente le modèle de programmation, cette phase est réalisée en concertation avec les utilisateurs ou par les utilisateurs eux-mêmes. Dans les deux cas, les utilisateurs doivent être en mesure d'appréhender les concepts manipulés lors de la spécification des tâches.

2.1.3.3 Synthèse

Afin d'obvier à l'hétérogénéité et au caractère dynamique des environnements ubiquitaires, les infrastructures logicielles doivent prendre en charge l'hétérogénéité des services, permettre de découvrir les services dans l'environnement, s'adapter à la disparition de services et aux changements d'activité des utilisateurs et offrir une méthode de programmation et de déploiement simple pour les services et les application ubiquitaires. Ces différentes tâches font appel à différents acteurs dans l'environnement ubiquitaire que sont l'architecte du modèle, le concepteur de service et le programmeur de l'environnement.

2.2 Approches existantes

Il existe de nombreuses approches visant à fournir une infrastructure logicielle pour programmer les environnement ubiquitaires. Parmi ces approches, nous étudions en particulier quatre approches qui permettent la composition dynamique de services dans les environnements ubiquitaires : Gaia, Aura, WSAMI et PICO. Nous évaluons en particulier ces travaux à l'aune des propriétés énoncées dans la section 2.1.3.1.

2.2.1 Gaia

Le projet Gaia [70] est développé par le groupe de recherche dirigé par Roy H. Campbell de l'université d'Illinois à Urbana-Champaign. Gaia fournit une infrastructure logicielle pour coordonner des composants dans des espaces physiques appelés espaces actifs. Un espace actif est défini comme une région physique avec des frontières bien définies contenant des objets physiques, des périphériques et des utilisateurs effectuant des activités. Gaia introduit la notion de session dans les espaces actifs qui associe les données et les applications aux utilisateurs, ce qui permet de rendre ces données et applications disponibles aux utilisateurs en situation de mobilité.

2.2.1.1 Architecture de Gaia

Gaia se définit comme un méta système d'exploitation dans le sens où il fournit les fonctionnalités courantes des systèmes d'exploitation : gestion des événements, système de fichiers, processus, signaux, etc. Le noyau de Gaia est constitué d'un coeur de gestion de composants et d'un ensemble de services utilisés par les applications. Ces services sont le gestionnaire d'événements, le service de présence, le système de fichiers contextuel et l'annuaire de l'espace.

Les applications dans Gaia sont constituées d'un assemblage de composants basés sur le modèle de composants distribué CORBA [31]. Le coeur de gestion de composants fournit les service de gestion du cycle de vie des composants permettant de charger, transférer, créer et détruire les composants dans l'espace actif. Le service de gestion d'événements permet aux composants de communiquer par échange de messages asynchrones. Les composants utilisent le gestionnaire d'événements pour s'informer des changements dans l'espace actif et réagir en conséquence. Le service de contexte permet aux applications de rechercher et d'obtenir les informations de contexte dont elles ont besoin, ce qui leur permet de s'adapter à leur environnement. Le service de présence contrôle les apparitions et disparitions de composants dans l'espace. Il publie des événements pour signaler les nouveaux composants découverts et les composants qui ne sont plus disponibles. Le système de fichier incorpore les données contextuelles dans un système de fichier traditionnel pour fournir les fichiers aux applications en fonction de la présence des utilisateurs et sélectionner le format des données en fonction du contexte et des préférences de l'utilisateur. Finalement, l'annuaire de l'espace centralise les informations sur tous les composants telles que le type de composant, leur localisation, etc. Les applications dans Gaia sont écrites avec le langage de script luaOrb [16]. LuaOrb permet d'instancier des composants et de les assembler pour former des applications. Le système Gaia établit la relation entre ces applications

et les ressources disponibles dans l'espace.

2.2.1.2 Évaluation

Le projet Gaia supporte la découverte de services grâce au service de présence et à l'annuaire de l'espace qui permettent de connaître les composants disponibles dans l'environnement et d'obtenir des informations sur ces composants. L'adaptation est également gérée car Gaia établit dynamiquement la relation entre les applications et les ressources disponibles dans l'environnement en fonction de politiques utilisateurs spécifiées au niveau du système. Le service de contexte permet aux applications d'être sensibles au contexte et de réagir aux changements dans l'environnement. Le déploiement des composants dans le système Gaia est aisé grâce au coeur de gestion de composants.

Le langage à script LuaOrb, qui permet le développement des applications pour le système Gaia, demande au programmeur de gérer des aspects bas niveau tels que l'instanciation des composants et leur assemblage. De ce fait, le niveau d'abstraction du langage ne permet pas de spécifier directement les intentions des utilisateurs. Les composants dans le système Gaia sont basés sur le modèle de composant CORBA. De ce fait, les périphériques hébergeant ces composants doivent disposer d'un ORB, qui est l'environnement d'exécution CORBA. Gaia ne supporte donc pas l'hétérogénéité en termes d'environnement d'exécution.

2.2.2 Aura

Le projet Aura [80] est développé à l'université Carnegie Melon. Il introduit le concept d'Aura personnelle, un système qui assiste les utilisateurs dans leurs activités en situation de mobilité. Quand un utilisateur entre dans un nouvel environnement, son Aura organise les ressources disponibles pour l'assister dans ses activités. De plus, l'Aura prend en compte les contraintes que le contexte physique impose à l'activité de l'utilisateur. L'infrastructure d'Aura permet de spécifier les intentions des utilisateurs sous forme de tâches [90] ainsi que leurs préférences personnelles.

2.2.2.1 Architecture d'Aura

L'architecture d'Aura est constituée de trois composants. Le premier composant est le gestionnaire de tâches appelé Prism, qui réalise le concept d'Aura personnelle. Le second composant est l'observateur de contexte qui fournit des informations sur l'environnement physique et envoi des événements aux autres composants lors des changements dans le contexte physique. Enfin, le gestionnaire d'environnement constitue l'interface entre l'infrastructure d'Aura et les services présents dans l'environnement. L'architecture d'Aura considère que tous les environnements dans lesquels les utilisateurs peuvent se déplacer possèdent une instance de ces trois composants. Finalement, les fournisseurs apportent les services dont les tâches sont constituées.

Le gestionnaire de tâches Prism prend en compte les changements dans l'environnement et dans le contexte pour exécuter les tâches qui sont l'expression des intentions des utilisateurs. Une tâche est décrite comme un ensemble d'activités spécifiées de manière abstraite comme

« éditer du texte » ou « lire une vidéo ». Ainsi, les tâches peuvent être instanciées dans différents environnements en utilisant les ressources disponibles. Les changements pris en compte par le gestionnaire de tâches sont de différentes natures :

- Lors de la migration d'un utilisateur entre deux environnements, le gestionnaire de tâches prend en charge la migration des données nécessaires à la tâche et assemble les services disponibles dans l'environnement pour assurer l'exécution de la tâche.
- Lorsque Prism détecte que l'utilisateur interrompt sa tâche ou change de tâche, le gestionnaire de tâche sauvegarde l'état de la tâche interrompue et instancie la nouvelle tâche. Pour détecter le changement de tâche, Prism prend en compte les indications des utilisateurs et les événements provenant de l'observateur de contexte.
- Lors de changement de contexte, si les contraintes spécifiées sur le contexte dans la tâche ne sont pas respectées, le gestionnaire de tâches peut remplacer les services dont la tâche est composée ou interrompre l'exécution de la tâche.

L'observateur de contexte fournit des informations sur le contexte physique et envoie des événements au gestionnaire de tâches Prism et au gestionnaire d'environnement. Les informations concernant le contexte sont stockées dans une base de données sur laquelle les applications peuvent faire des requêtes. Le gestionnaire d'environnement centralise les informations sur les fournisseurs disponibles et les services qu'ils offrent. Il fournit également le mécanisme d'accès au système de fichiers distribué permettant la migration des données utilisateurs lors de la mobilité des utilisateurs. Le gestionnaire d'environnement fournit un annuaire auprès duquel les fournisseurs s'enregistrent, permettant de découvrir les services nécessaires à l'exécution d'une tâche. Finalement, les fournisseurs offrent les services dont les tâches sont composées. Les services sont des applications existantes, telles que des éditeurs de texte, encapsulées pour se conformer à l'interface de programmation (API, Application Programming Interface) d'Aura. Les services sont décrits à l'aide d'un langage de balisage, afin de permettre d'identifier la fonction du service sans se préoccuper des détails d'implémentation spécifiques à l'application. Aura présume que les fournisseurs partagent un vocabulaire commun pour la description de services, ce qui permet d'identifier les fournisseurs proposant des services identiques. Par exemple, Emacs, Microsoft Word et Notepad sont décrits comme fournisseurs de service d'édition de texte.

2.2.2.2 Évaluation

La découverte de services est gérée par Aura grâce au gestionnaire d'environnement qui centralise les informations sur les fournisseurs disponibles dans l'environnement. Aura assure également l'adaptation car les tâches sont instanciées en utilisant les services disponibles dans l'environnement, et ces services peuvent être remplacés lorsque l'utilisateur change d'environnement, que l'utilisateur interrompt sa tâche ou change de tâche, ou que les contraintes sur l'environnement ne sont plus respectées. Aura permet de développer les applications à un haut niveau d'abstraction grâce au paradigme de développement par tâche qui permet de spécifier les intentions des utilisateurs sous forme de tâches.

Les fournisseurs de services doivent décrire les fonctionnalités fournies par les services à l'aide d'un langage de balisage. Ils doivent donc utiliser un même vocabulaire pour décrire leurs services afin de permettre au service de découverte d'interpréter les descriptions de services. De

plus, les services doivent respecter l'API d'Aura pour s'intégrer dans l'infrastructure. L'hétérogénéité des services est donc limitée dans le sens où tous les services doivent se conformer à un vocabulaire de description et une API uniforme et propriétaire.

2.2.3 WSAMI

WSAMI [46] est une infrastructure basée sur les architectures orientées service qui permet d'accéder aux services indépendamment de leur localisation et des capacités du périphérique utilisé. Les applications sont développées à l'aide du langage WSAMI qui permet de spécifier un assemblage de services abstraits qui sont instanciés à l'exécution en fonction des services disponibles. L'infrastructure utilise la technologie des Services Web qui fournit un ensemble de protocoles pour la description d'interfaces, la publication, la découverte et l'invocation de services. Ces protocoles sont les suivants [64] :

- WSDL est un langage déclaratif pour spécifier les interfaces des Service Web
- SOAP définit une structure de message pour l'échange d'informations entre services
- UDDI joue le rôle d'annuaire de services permettant la publication et la découverte de services.

Ces protocoles sont basés sur le langage de balisage XML. L'infrastructure de WSAMI permet la sélection et l'assemblage dynamique de services. Le langage WSAMI permet de plus de spécifier des contraintes de qualité de service sur la description abstraite des services.

2.2.3.1 Architecture de WSAMI

L'architecture de WSAMI est constituée de deux composants principaux : l'intermédiaire central et le service de nommage et de découverte. L'intermédiaire central contient un parseur XML, un conteneur SOAP permettant de déployer les Services Web, l'outil WSDL2WSAMI qui permet de générer des descriptions WSAMI à partir des descriptions WSDL des services et l'outil InstallWSAMI permettant d'installer et de configurer les services sur l'intergiciel WSAMI. Le service de nommage et de découverte gère une base de descriptions abstraites de services avec les instances de services associées, et permet de découvrir les services qui sont accessibles sur le réseau. L'implémentation de WSAMI est conçue pour pouvoir s'exécuter sur des périphériques possédant des capacités limitées en termes de mémoire, de capacité de traitement et de stockage.

2.2.3.2 Évaluation

L'infrastructure WSAMI gère la découverte de services grâce au service de nommage et de découverte. L'adaptation est également gérée car les services composant une application WSAMI sont assemblés dynamiquement à l'exécution en fonction des services disponibles. L'hétérogénéité est prise en charge car l'infrastructure est basée sur les architectures orientées service, ce qui garantit une indépendance des services vis-à-vis des langages de programmation et des plates-formes d'exécution. La programmation des applications se fait en spécifiant les services abstraits qui composent l'application, permettant ainsi une programmation rapide et simple des applications ubiquitaires.

La sensibilité au contexte n'apparaît pas dans les objectifs du projet WSAMI. De fait, l'infrastructure WSAMI ne prend pas en compte les informations provenant du contexte.

2.2.4 PICO

L'infrastructure PICO [53] (Pervasive Information Community Organization) a pour but de réaliser des missions à temps contraint, par exemple dans le domaine militaire, médical ou la gestion de crises. L'infrastructure PICO permet la collaboration en temps réel entre les services dans un environnement dynamique et hétérogène. La collaboration entre les services est basée sur la création dynamique de communautés de services dévouée à la réalisation d'une mission.

2.2.4.1 Architecture de PICO

L'architecture de PICO est basée sur deux types d'entités : les entités logicielles appelées delegents (intelligent delegates) et les entités matérielles appelées camileuns (connected, adaptive, mobile, intelligent, learned, efficient, ubiquitous nodes). Un camileun est une représentation abstraite d'un périphérique qui capture les fonctionnalités fournies par le périphérique. Un camileun est décrit par un tuple <C,F,H> ou C est l'identifiant du camileun, F est l'ensemble des fonctionnalités fournies par ce périphérique et H l'ensemble des caractéristiques physiques du périphériques. Un delegent encapsule une fonctionnalité d'un camileun et offre cette fonctionnalité à une ou plusieurs communautés dans l'environnement. Les delegents réagissent aux événements provenant des capteurs et des communautés auxquelles il appartient. Le comportement d'un delegent est défini par des règles qui consistent en un ensemble de conditions et d'actions et qui déterminent la réaction des delegents aux événements. Les delegents s'organisent en communautés pour réaliser des missions. Une communauté est constituée d'un ensemble de delegents qui collaborent pour accomplir un but commun. Les delegent possèdent une interface de communication uniforme qui leur permet d'interagir entre eux au sein d'une communauté. Les communautés peuvent être formées statiquement lors du déploiement des delegents ou se former dynamiquement en réponse aux événements provenant de l'environnement. Lorsqu'une communauté est formée, ses delegents collaborent pour accomplir leur but commun.

2.2.4.2 Évaluation

La découverte de services est gérée par l'infrastructure de PICO qui permet de découvrir les services grâce aux tuples qui décrivent leurs fonctionnalités. Les delegents sont sensibles au contexte car leur comportement est décrit sous forme de règles qui définissent des réactions aux événements provenant de l'environnement. L'adaptation est également gérée car les communautés de delegents se forment dynamiquement en réponse aux événements provenant de l'environnement. La programmation de la logique applicative sous forme de règles permet de développer des applications de manière simple et orientée sur les missions à réaliser.

PICO prend en compte l'hétérogénéité matérielle des périphériques car la description des camileun intègre les capacités matérielles des périphériques. Cependant, l'hétérogénéité logicielle

n'est pas prise en compte car les delegents doivent être développés spécifiquement pour l'infrastructure de PICO et doivent respecter une interface de communication uniforme et propriétaire.

2.2.5 Synthèse

Le tableau 2.1 synthétise les capacités des différentes approches étudiées suivant les besoins exprimés dans la section 2.1.3.1.

	Gestion de l'hétérogénéité	Découverte de service	Capacité d'adaptation	Développement et déploiement	Sensibilité au contexte
Gaia		√	√		√
Aura		√	√	√	√
WSAMI	√	√	√	√	
PICO		√	√	√	√

TAB. 2.1 – Synthèse des capacités des approches présentées

Aucune des approches présentées ne répond aux besoins exprimés. Dans la suite de ce document, nous proposons une infrastructure qui répond à ces besoins et possède les propriétés nécessaires à la programmation des environnements ubiquitaires.

N'importe quel objet peut être un objet d'art pour peu qu'on l'entoure d'un cadre.

Boris Vian

3 Démarche

Dans le chapitre précédent, nous avons présenté les concepts d'informatique ubiquitaire et d'environnement ubiquitaire, et nous avons exposé les propriétés que doivent posséder les infrastructures logicielles qui permettent la programmation d'applications ubiquitaires. L'étude des approches existantes dans ce domaine a mis en lumière certains manques vis-à-vis des défis présentés.

Nous rappelons dans ce chapitre les problématiques posées par la programmation d'applications ubiquitaires. Nous présentons ensuite la démarche mise en œuvre afin de répondre à ces problématiques.

3.1 Problématique

Les environnements ubiquitaires sont des environnements saturés de périphériques hétérogènes au niveau matériel et logiciel. Ces périphériques peuvent, de par leur mobilité, apparaître et disparaître de l'environnement, conduisant à une disponibilité dynamique des services qu'ils exposent. Les occupants de l'environnement accomplissent diverses activités et leurs intentions peuvent évoluer avec le temps, par exemple lorsqu'ils entament ou terminent une activité. Afin d'être en mesure d'assister ses utilisateurs de manière pertinente dans leurs activités, l'infrastructure logicielle sur laquelle s'exécutent les applications ubiquitaires doit être capable d'exploiter les informations qui proviennent de l'environnement et qui caractérisent les circonstances entourant les activités des utilisateurs. Finalement, le paradigme de programmation des applications ubiquitaires doit offrir un haut niveau d'abstraction afin de libérer le programmeur des détails techniques de découverte et d'invocation de services, lui permettant ainsi de se focaliser sur les intentions des utilisateurs.

Une infrastructure logicielle pour la programmation d'applications ubiquitaires doit donc,

comme nous l'avons montré dans le chapitre précédent :

- gérer l'hétérogénéité des services de l'environnement,
- prendre en charge la découverte de services,
- être capable de s'adapter au caractère dynamique de l'environnement en terme de disponibilité des services et d'intentions des utilisateurs,
- être sensible au contexte,
- permettre la programmation des applications à un haut niveau d'abstraction.

Les approches existantes présentées dans la section 2.2 ne répondent pas à l'ensemble de ces besoins. Nous proposons donc une infrastructure logicielle qui réponde aux défis posés par la programmation d'applications ubiquitaires.

3.2 Démarche globale

La démarche que nous mettons en œuvre afin de répondre aux problématiques de la programmation d'applications ubiquitaires est fondée sur trois axes.

- Un modèle de contexte qui formalise les concepts de l'environnement ubiquitaire.
- Une approche langage basée sur un langage de règles extensible permettant une programmation à un haut niveau d'abstraction.
- Une infrastructure logicielle basée sur les architectures orientées service permettant d'intégrer des services hétérogènes.

La figure 3.1 illustre cette démarche. Le modèle de contexte est développé par l'architecte du modèle qui formalise les concepts pertinents pour la programmation d'applications dans un environnement ubiquitaire donné. Ces concepts définissent les abstractions offertes par le langage de règles logiques permettant la programmation d'applications pour cet environnement. Le langage est dynamiquement étendu lorsque des services apparaissent dans l'environnement pour offrir au programmeur des abstractions permettant d'interagir avec ces services. Cette approche est mise en œuvre dans l'infrastructure PERSEWS qui fournit un environnement de développement et un environnement d'exécution pour les règles logiques spécifiées par le programmeur de l'environnement. L'infrastructure PERSEWS est basée sur une architecture orientée service, ce qui permet de s'abstraire de l'hétérogénéité des services de l'environnement en termes de spécificités matérielles, de système d'exploitation, de langage de programmation et de plate-forme d'exécution. Dans le reste de ce document, nous détaillons les trois axes de notre démarche. Tout d'abord, nous décrivons la modélisation du contexte sous forme d'ontologie. Puis, nous présentons l'approche langage permettant la programmation des applications ubiquitaires à un haut niveau d'abstraction. Finalement, nous présentons l'infrastructure PERSEWS.

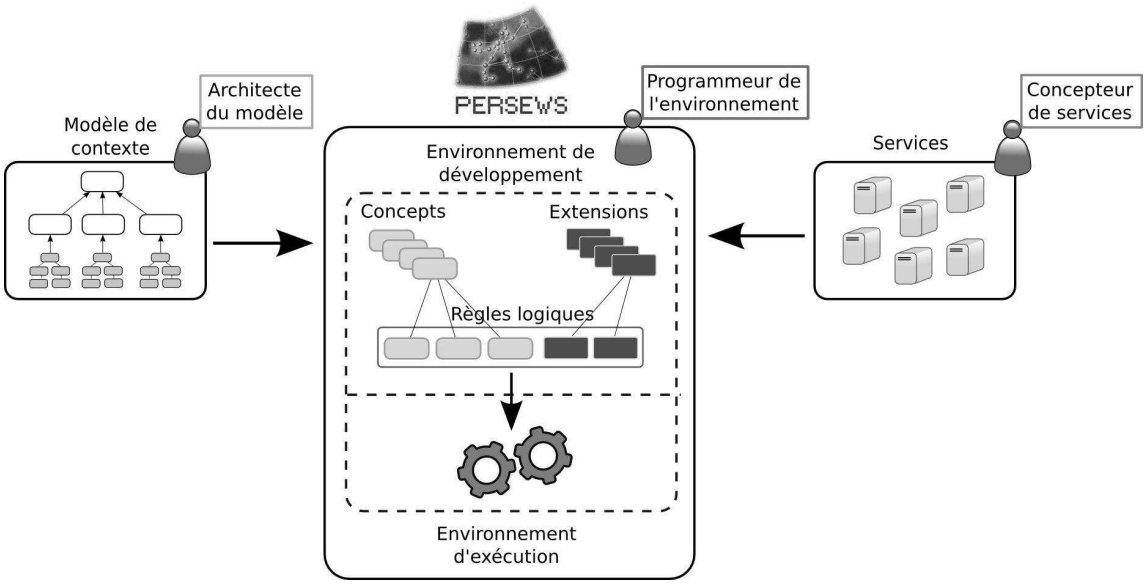

FIG. 3.1 – Démarche globale pour la programmation d'applications dans les environnements ubiquitaires.

Deuxième partie

Approche proposée

Ce qui distingue dès l'abord le pire architecte de l'abeille la plus experte, c'est qu'il a construit la cellule dans sa tête avant de la construire dans la ruche.

Karl Marx, *Le Capital*

4 Modélisation du contexte

Les environnements ubiquitaires sont saturés de périphériques informatiques hétérogènes au niveau matériel et logiciel. Ces périphériques et les services qu'ils hébergent sont développés et déployés de manière indépendante, et sont utilisés conjointement pour assister les utilisateurs dans leurs activités au sein de l'environnement. Les applications ubiquitaires qui assemblent ces services doivent être sensibles aux informations contextuelles afin de fournir des services pertinents aux utilisateurs. Toutefois, les informations contextuelles sont hétérogènes dans leur représentation, et parcellaires car obtenues de sources distribuées dans l'environnement. Ces informations doivent être partagées entre les services de l'environnement afin qu'ils disposent d'une vision complète de l'environnement. Les environnements ubiquitaires sont donc confrontés au phénomène de « tour de Babel » : les différents acteurs utilisent des langages et des structures de données hétérogènes pour représenter leurs connaissances. Le partage d'information dans un environnement ubiquitaire nécessite donc de définir un modèle uniforme de représentation des informations qui sera partagé par l'ensemble des services dans l'environnement. Nous proposons une approche basée sur les ontologies pour définir ce modèle.

4.0.1 Définition d'une ontologie

En philosophie, l'Ontologie est l'étude de l'être en tant qu'être, c'est-à-dire l'étude du type et de la structure des objets, de leurs propriétés et relations dans tous les domaines de la réalité. D'une manière générale, l'Ontologie a pour but la classification des entités. Bien que l'étude de l'Ontologie remonte à l'antiquité, le terme d'Ontologie n'a été introduit qu'au XVII^{e} siècle.

Dans le domaine informatique, le terme d'ontologie désigne « la spécification explicite d'une conceptualisation » [32], c'est-à-dire la représentation formelle d'un domaine. La conceptualisation désigne l'ensemble des concepts et objets qui existent dans un domaine, et leurs relations. La spécification de la conceptualisation désigne le formalisme déclaratif qui permet de représenter

ces concepts et objets. Une ontologie est l'ensemble structuré des termes et concepts fondant le sens d'un champ d'information. L'ontologie constitue un modèle de données représentatif d'un ensemble de concepts dans un domaine, ainsi que des relations entre ces concepts. Une ontologie est composée de classes, qui définissent les concepts du domaine (par exemple, dans une ontologie de périphériques, une classe peut être *Téléphone*), de propriétés qui définissent les relations entre ces concepts (par exemple, la classe *Téléphone* peut avoir la propriété *OS* qui définit son système d'exploitation), d'instances (ou individus) qui représentent les éléments concrets appartenant à une classe (par exemple *IPhone* est une instance de la classe *Téléphone*) et d'axiomes qui imposent des contraintes sur certains éléments de l'ontologie (par exemple un *Téléphone* doit avoir au moins un *OS*).

Les ontologies sont stockées dans des bases de connaissances, un type particulier de bases de données permettant la gestion des connaissances. Une base de connaissances permet de stocker et d'extraire des informations de manière consistante. De plus, des mécanismes de raisonnement peuvent être appliqués sur les informations stockées dans une base de connaissances.

4.0.2 Propriétés des ontologies

La représentation d'un domaine sous forme d'ontologie offre un certain nombre d'avantages. En particulier, l'utilisation d'ontologies facilite le partage de connaissances, la réutilisation des connaissances et l'inférence logique sur les connaissances [89].

Partage de connaissances. L'utilisation d'une ontologie commune par les différents services dans un environnement ubiquitaire permet à ces services de partager un ensemble de concepts communs. Ceci leur permet d'avoir une vision uniforme de l'environnement dans lequel ils évoluent et d'utiliser un vocabulaire commun lorsqu'ils interagissent.

Réutilisation des connaissances. L'utilisation des ontologies pour la modélisation d'un domaine permet de construire des ontologies à partir d'ontologies existantes. Par exemple, une ontologie des périphériques peut utiliser une ontologie existante des télécommunications pour modéliser la partie du domaine concernant les téléphones. La réutilisation des connaissances permet de réduire considérablement le temps de développement des ontologies.

Inférence logique. L'utilisation de mécanismes de raisonnement logique sur les ontologies permet d'inférer des informations de haut niveau à partir des informations de bas niveau modélisées dans l'ontologie. On peut distinguer les mécanismes de raisonnement ontologiques, qui permettent de raisonner sur la validité d'une ontologie à partir des axiomes définis dans l'ontologie [37], et les mécanismes de raisonnement définis par l'utilisateur qui permettent aux utilisateurs de spécifier des règles logiques s'appliquant à l'ontologie.

4.1 Langages de representation des ontologies

La modélisation d'un domaine sous forme d'ontologie nécessite un langage pour spécifier les concepts du domaine. Il existe de nombreux langages de représentation d'ontologies (XOL [48], SHOE [58], OML [49], OIL [24], DAML+OIL [36]). Actuellement, les langages du Web

Sémantique [6] s'imposent comme la référence pour la modélisation de connaissance dans les environnements ubiquitaires [89, 67, 18, 84]. Le Web Sémantique est une initiative du World Wide Web Consotium (W3C) ayant pour but de permettre la description explicite de ressources sur l'Internet. Le Web Sémantique se compose d'un ensemble de standards pour faciliter la modélisation des connaissances dans un format exploitable par les applications informatiques. La figure 4.1 présente un modèle en couche inspiré du modèle introduit par Tim Berners-Lee [5] et montre le lien entre les différents standards du Web Sémantique. La norme UNICODE [20], les URI [43] et le langage XML permettent une représentation standard des informations facilitant le partage entre différentes plates-formes logicielles. RDF [56] permet une représentation des ressources et de leurs relations. RDF Schema [13] permet de définir des concepts et les relations entre ces concepts sous forme de classes et de propriétés. Finalement, OWL [4] constitue une extension sémantique à RDF Schema offrant un vocabulaire plus riche pour la description des classes et des propriétés. Nous nous intéressons particulièrement au langage OWL, qui est le langage de référence pour la définition d'ontologies pour le Web Sémantique.

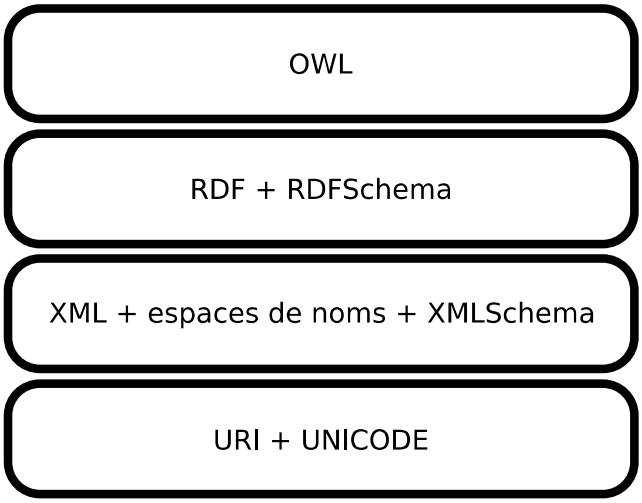

FIG. 4.1 – Achitecture du Web Sémantique

4.1.1 Unicode, URI et XML

Unicode est une norme informatique développée par le Consortium Unicode qui donne à tout caractère dans un système d'écriture de langue un nom et un identifiant numérique, ce qui permet d'obtenir une représentation uniforme des caractères quelle que soit la plate-forme logicielle. Une URI (Uniform Resource Identifier) est une chaîne de caractères qui identifie une ressource sur un réseau, que ce soit une ressource physique, dont la représentation peut être récupérée via l'Internet (page web, service) ou une ressource abstraite (livre, numéro de téléphone). Les exemples ci-dessous illustrent des usages courants d'URI :

- http://phoenix.labri.fr/ (une page web)
- mailto:johndoe@somewhere.edu (une adresse électronique)

– urn:isbn:0451450523 (un livre)
– urn:isan:0000-0000-9E59-0000-O-0000-0000-2 (un film)

XML (Extensible Markup Language) est un langage de balisage qui fournit une syntaxe pour la représentation de données hiérarchiques. XML a pour but de faciliter l'échange d'informations entre systèmes informatiques hétérogènes, et de permettre l'interopérabilité entre ces systèmes. XML est qualifié de langage extensible car il permet aux développeurs de définir leurs propres balises.

XMLSchema est un langage de description de format de document XML qui permet de définir la structure d'un document XML. Un schéma XML impose des contraintes sur la structure et le contenu d'un document XML auxquelles un document XML doit se conformer pour être considéré comme valide selon ce schéma. Les schémas XML permettent de définir des sous-ensemble du langage XML appelés dialectes XML.

Un espace de nom est un conteneur abstrait permettant de définir un ensemble d'identifiants uniques. Un espace de nom XML est identifié par une URI, et définit un vocabulaire qui peut être utilisé en tant que balises dans un document XML. L'utilisation des espaces de noms permet de préserver leur unicité des noms et empêche les conflits dans les documents XML.

4.1.2 RDF et RDF Schema

RDF (Resource Description Framework) est un modèle de graphe permettant de décrire de manière formelle des ressources et les relations entre ces ressources. Une description RDF est constituée d'un ensemble de triplets {sujet, prédicat, objet}. La figure 4.2 présente une vision graphique du triplet RDF. Le sujet représente la ressource à décrire. Le prédicat représente un type de propriété applicable à cette ressource. L'objet représente la valeur de cette propriété et peut être une donnée ou une autre ressource. RDF peut être sérialisé sous forme XML afin de permettre le traitement automatique des descriptions par des systèmes informatiques. Dans ce cas, les ressources et les prédicats sont identifiés par des URI. Par exemple, pour décrire le fait que l'article identifié par l'URI http://julien.lancia.googlepages.com/percom.pdf a pour titre « High-level Programming Support for Robust Pervasive Computing Applications », nous définissons un triplet RDF {http://julien.lancia.googlepages.com/percom.pdf, title, High-level Programming Support for Robust Pervasive Computing Applications}. La représentation XML de ce triplet est la suivante:

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dc="http://purl.org/dc/elements/1.1/">
<rdf:Description
 rdf:about="http://julien.lancia.googlepages.com/percom.pdf">
<dc:title>High-level Programming Support
 for Robust Pervasive Computing Applications</dc:title>
</rdf:Description>
</rdf:RDF>
```

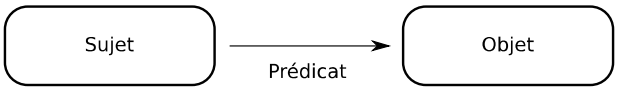

FIG. 4.2 – Triplet RDF

RDFS (RDF Schema) est un langage qui permet de décrire des ontologies avec le modèle RDF et constitue une extension sémantique à RDF. RDFS permet de décrire des groupes de ressources et les relations entre ces groupes de ressources. Deux mécanismes permettent de définir des ontologies à l'aide de RDFS: les classes et les sous-classes d'une part, les propriétés et les sous-propriétés d'autre part.

Classes et sous-classes. Les ressources peuvent être réparties en groupes appelés classes. Les classes sont représentées sous forme d'URI et décrites à l'aide de propriétés RDF. Les ressources appartenant à une classe sont désignées indifféremment sous le terme d'instances de la classe ou d'individus de la classe. Les classes peuvent être organisées de manière hiérarchique en classes et sous-classes. Toutes les instances appartenant à une sous-classe appartiennent également à ses classes mères. Cette relation entre les instances des classes et des sous-classes est appelée relation de subsomption. Comme présenté dans la figure 4.3, si une classe *Personne* a pour sous-classe une classe *Doctorant*, cela implique que les instances directes de la classe *Doctorant* (Pierre, Paul et Jacques) sont également des instances indirectes de la classe *Personne*. Dans ce cas, la classe *Personne* subsume la classe *Doctorant*, et la classe *Doctorant* est subsumée par la classe *Personne*.

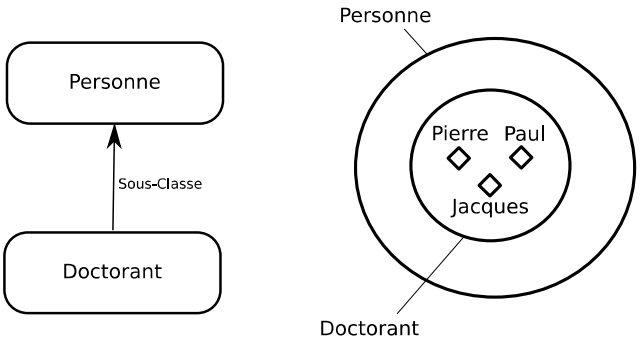

FIG. 4.3 – Relation de subsomption entre classes.

Propriétés et sous-propriétés. Les propriétés définissent les relations entre les classes. Par exemple, la propriété *Auteur* permet de lier la classe *Article* à la classe *Personne*, afin de spécifier des personnes comme auteurs d'un article. Deux mécanismes permettent d'appliquer des contraintes sur les propriétés: le domaine et la portée. Le domaine d'une propriété permet de restreindre le sujet d'une propriété à une classe ou un groupe de classes. De même, la portée d'une propriété permet de restreindre l'objet d'une propriété à une classe ou un groupe de classes. Par exemple, en spécifiant la classe *Article* comme domaine de la propriété *Auteur*, et la classe *Personne* comme portée de cette même propriété, on impose que la propriété *Auteur* ne puisse lier que des instances de *Personne* et d'*Article*. Lorsqu'on

déclare deux individus x et y comme étant liés par une propriété P, on dit que le couple (x,y) est une instance de la propriété P. Comme pour les classes, les propriétés peuvent être organisées hiérarchiquement en propriétés et sous-propriétés. La relation de subsomption s'applique également aux propriétés, ainsi les instances liées par une propriétés le sont également par ses propriétés mères. Le domaine et la portée d'une propriété sont hérités par ses sous-propriétés. Par exemple, si *PremierAuteur* est une sous-propriété d'*Auteur*, alors cette propriété ne peut lier que des instances de la classe *Article* et de la classe *Personne* entre elles.

4.1.3 Web Ontology Language

OWL (Web Ontology Language) est un langage de spécification d'ontologies pour le Web Sémantique développé par le World Wide Web Consortium (W3C) Web Ontology Working Group. Tout comme RDFS, OWL permet de décrire des classes et des propriétés sous forme d'URI et d'organiser ces classes de manière hiérarchique grâce à la relation de subsomption. Outre la relation de subsomption, OWL permet de définir des classes comme étant équivalentes, c'est-à-dire qu'elles contiennent le même ensemble d'individus, ou disjointes, c'est-à-dire qu'elles ne possèdent aucun individu en commun. Afin d'offrir une plus grande expressivité, OWL fournit des mécanismes additionnels pour décrire les classes et les propriétés.

Description de classes OWL offre trois mécanismes additionnels par rapport à RDFS pour définir les classes.

- Sous forme d'une énumération exhaustive d'individus. Par exemple, la classe *Continent* peut être définie comme l'énumération des individus *Eurasie, Afrique, AmériqueDuNord, AmériqueDuSud, Australie* et *Antarctique*.
- Sous forme d'une restriction de propriété. Les restrictions de propriétés définissent des classes constituées de tous les individus qui satisfont à la contrainte exprimée dans la restriction. OWL définit deux sortes de contrainte: les contraintes de valeur et les contraintes de cardinalité.

  Les contraintes de valeur s'appliquent sur la portée d'une propriété. Par exemple, comme présenté dans la figure 4.4, la classe *ProfesseurInformatique* peut être définie comme l'ensemble des individus de la classe *Professeur* dont la propriété *APourÉlève* a pour portée un individu de la classe *ÉtudiantInformatique*.

  La contrainte sur la cardinalité porte sur le nombre de valeurs que peut prendre une propriété. Par exemple, la classe *ÉquipeDeBasket* peut être définie par une restriction sur la classe *Équipe* imposant une cardinalité 5 à la propriété *Joueur*.
- Sous forme de combinaison logique (intersection, union ou complément) de classes existantes. Par exemple, on peut définir la classe *Homme* comme étant l'intersection de la classe *Personne* avec le complémentaire de la classe *Femme*. Un homme désigne alors l'ensemble des individus qui sont des personnes et ne sont pas des femmes.

Description de propriétés OWL distingue deux types de propriétés: les propriétés objets qui lient des individus entre eux, et les propriétés de types de données qui lient des individus à des données (entier, chaînes de caractère, etc). Outre les constructions offertes pas RDFS

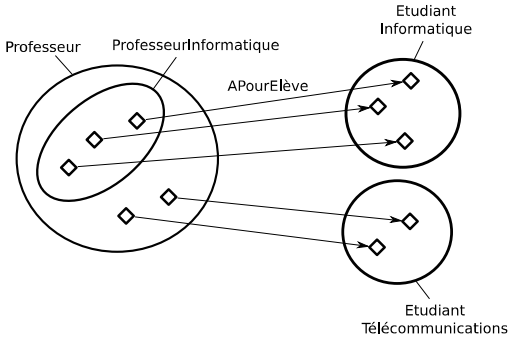

FIG. 4.4 – Définition d'une classe par contrainte de valeur.

(sous-propriété, domaine, portée), OWL offre trois mécanismes additionnels permettant de caractériser les propriétés:

– Les relations aux autres propriétés. Une propriété peut être définie comme l'inverse d'une propriété existante. Si une propriété P2 est définie comme l'inverse d'une propriété P1, alors pour toute instance (x,y) de la propriété P1, (y,x) est une instance de la propriété P2. La figure 4.5 présente un exemple de propriétés inverses. Étant donné un individu *Pierre* et une individu *IPhone*, une propriété *Possède* et sa propriété inverse *AppartientA*. Si on définit que Pierre possède l'Iphone, cela implique que l'IPhone appartient à Pierre de par la relation inverse des deux propriétés.

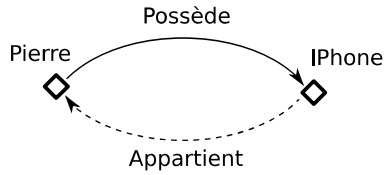

FIG. 4.5 – Exemple de propriétés inverses.

– Les contraintes sur la cardinalité globale. Une propriété peut être définie comme étant fonctionnelle, ce qui impose que cette propriété ne peut avoir qu'une seule valeur pour chaque instance de la classe domaine. Par exemple, la propriété *APourPère* ayant pour domaine la classe *Enfant* et pour portée la classe *Parent* peut être définie comme propriété fonctionnelle. Ainsi, on ne peut définir qu'un seul père pour chaque individu de la classe *Enfant*.
– Les caractéristiques logiques. Les propriétés peuvent être définies comme étant transitives et symétriques.
Lorsqu'une propriété P est définie comme étant transitive, cela signifie que si le couple (x,y) est une instance de P et que le couple (y,z) est une instance de P, alors le couple (x,z) sera également une instance de P. La figure 4.6 présente un exemple de propriété transitive. Étant donné trois individus Pierre, Paul et Jacques. Les instances {Pierre, Paul} et {Paul, Jacques} de la propriété *APourAncêtre* signifient que Pierre a pour ancêtre Paul, qui a pour ancêtre Jacques. Si la propriété *APourAncêtre* est transitive, cela implique

que {Pierre, Jacques} est également une instance de la propriété *APourAncêtre*, et que Pierre à donc pour ancêtre Jacques.

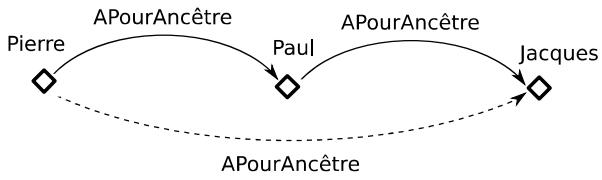

FIG. 4.6 – Exemple de propriété transitive.

Lorsqu'une propriété P est définie comme symétrique, si le couple (x,y) est une instance de P alors le couple (y,x) est également une instance de P. La figure 4.7 présente un exemple de propriété symétrique. Étant donné deux individus Pierre et Paul. {Pierre, Paul} est une instance de la propriété *EstFrère*, ce qui signifie que Pierre est le frère de Paul. Si la propriété *EstFrère* est symétrique, cela implique que {Paul, Pierre} est également une instance de la propriété *EstFrère*, ce qui signifie que Paul est le frère de Pierre.

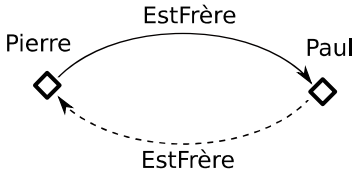

FIG. 4.7 – Exemple de propriété symétrique.

Le langage OWL est divisé en trois sous-langages offrant des puissances d'expressivité dégressives: OWL-Full, OWL DL et OWL Lite. Cette subdivision permet de choisir un langage possédant une expressivité adaptée au domaine à modéliser sans avoir à prendre en compte toutes les constructions du langage OWL. OWL-Lite est la version du langage OWL la moins expressive, la moins complexe et est adaptée à la description de thésaurus et de taxonomies. OWL DL (pour Description Logics, logique de description, une famille de langage de représentation de connaissances) est particulièrement adapté pour la description d'ontologies et le raisonnement sur ces ontologies. OWL Lite et OWL DL correspondent à des variantes de la logique de description, respectivement $\mathcal{SHIF}(\mathcal{D})$ et $\mathcal{SHOIN}(\mathcal{D})$ [40]. Ces deux version du langage OWL garantissent la complétude des raisonnements sur les ontologies (toutes les inférences sont calculables) et leur décidabilité (leur calcul se fait en une durée finie). Finalement, OWL Full est la variante la plus complexe du langage OWL et celle qui permet le plus haut niveau d'expressivité. OWL Full offre notamment la compatibilité avec RDFS. Cependant, cette expressivité se fait au détriment de la complétude et de la décidabilité.

4.2 Modèle de contexte

Afin de permettre l'échange d'informations de contexte hétérogènes entre les services dans un environnement ubiquitaire, nous modélisons le contexte sous forme d'ontologie. La spécification du modèle de contexte est basée sur le langage OWL DL. En effet, cette version du langage OWL est suffisamment expressive pour modéliser des ontologies tout en garantissant la complétude et la décidabilité des raisonnements sur ces ontologies. Comme présenté dans la figure 4.8, notre ontologie de contexte est divisée en deux parties distinctes: une ontologie supérieure et une ontologie du domaine. L'ontologie supérieure décrit les concepts récurrents dans l'ensemble des environnements ubiquitaires. Ces concepts sont les périphériques, les services, les lieux, les personnes, les conditions physiques et les états. L'ontologie du domaine permet de modéliser un domaine particulier en spécifiant des hiérarchies de classes héritant des classes génériques de l'ontologie supérieure.

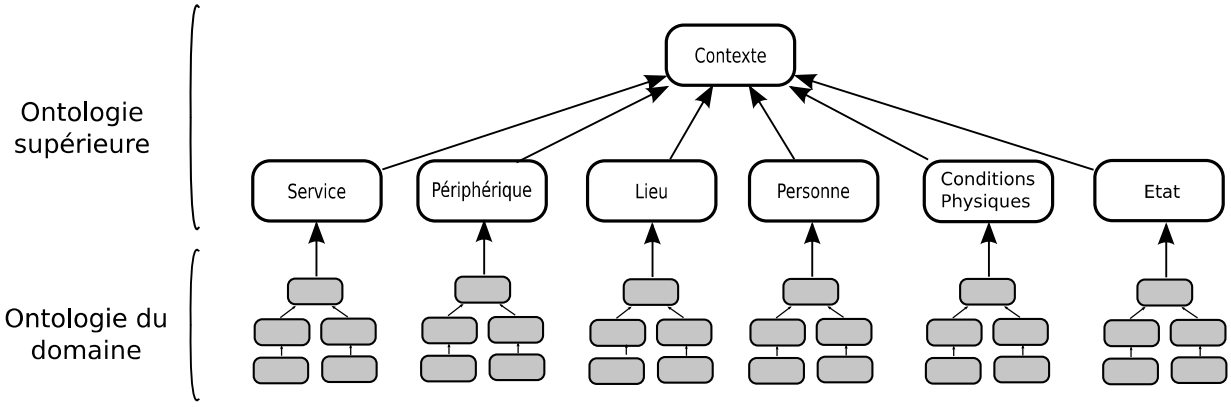

FIG. 4.8 – Ontologie supérieure et ontologie du domaine

4.2.1 Modélisation des périphériques et des services

La figure 4.9 présente un exemple d'ontologie modélisant un environnement ubiquitaire. La classe *Périphérique* permet de modéliser les types de périphériques informatiques pouvant apparaître dans l'environnement ubiquitaire. Un type de périphérique peut être un assistant personnel, un ordinateur, une console de jeux, un téléphone, etc. Les types de périphériques sont modélisés sous forme de sous-classes de la classe *Périphérique* dans l'ontologie du domaine. La description des périphériques dans l'ontologie du domaine permet de modéliser les capacités matérielles des différents périphériques dans l'environnement. Ces différentes capacités sont modélisées dans l'ontologie du domaine sous forme de propriétés des classes représentant les périphériques. Par exemple, dans l'ontologie présentée dans la figure 4.9, les propriétés *Résolution* et *Zoom* permettent de définir les capacités matérielles des instances de *Caméra*.

La classe *Service* permet de modéliser les différents types de services disponibles dans l'environnement. De manière similaire aux périphériques, les différents types de services sont modélisés sous forme de sous-classes de la classe *Service* dans l'ontologie du domaine. Les sous-

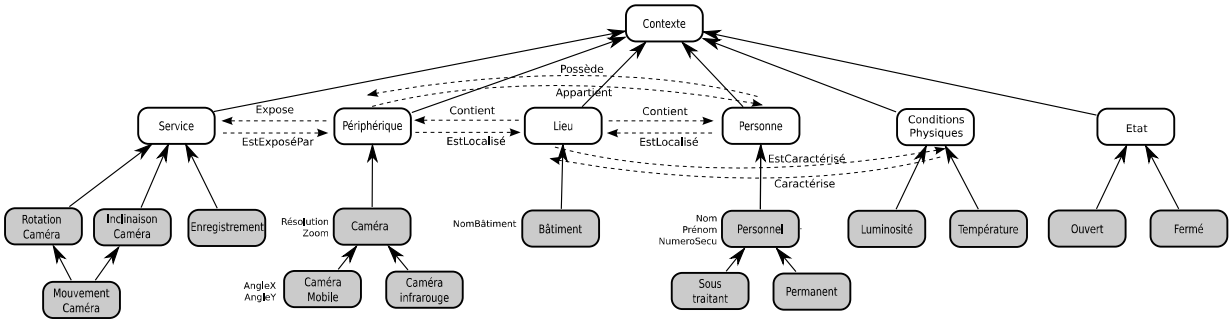

FIG. 4.9 – Exemple de modèle de contexte

classes de *Service* dans l'ontologie du domaine permettent de regrouper les services par type de fonctionnalité offerte. Par exemple, la classe *RotationCaméra* permet de regrouper l'ensemble des services fournissant une fonctionnalité de rotation de caméra et la classe *InclinaisonCaméra* l'ensemble des services permettant de contrôler l'inclinaison des caméras.

Les propriétés *Expose* et *EstExposéPar* permettent de lier les périphériques et les services. La propriété *Expose* permet de spécifier pour chaque individu de la classe *Périphérique* l'ensemble des services qu'il offre dans l'environnement ubiquitaire. La propriété *EstExposéPar*, inverse de la propriété *Expose*, permet de spécifier pour chaque individu de la classe *Service* le périphérique qui héberge ce service.

4.2.2 Modélisation des lieux, personnes et conditions physiques

Les lieux, les personnes et les conditions physiques sont modélisés comme des sous-classes des classes *Lieu*, *Personne* et *ConditionPhysique* de l'ontologie supérieure. Ces sous-classes appartiennent à l'ontologie du domaine et peuvent être organisées de manière hiérarchique. Les propriétés inverses *EstLocalisé* et *Contient* permettent d'établir la relation entre une personne ou un périphérique et un lieu. De même, les propriétés inverses *Possède* et *Appartient* permettent d'établir la relation entre une personne et un périphérique. Enfin, les propriétés inverses *Caractérise* et *EstCaractérisé* permettent d'établir la relation entre un lieu et les propriétés physiques qui le caractérisent. Des propriétés peuvent être définies pour les sous-classes des classes *Lieu*, *Personne* et *ConditionPhysique* dans l'ontologie du domaine afin de fournir des informations complémentaires sur les individus de ces classes. Par exemple, dans l'ontologie présentée dans la figure 4.9, les propriétés *Nom*, *Prénom* et *NuméroSécu* de la classe *Personnel* permettent de spécifier les nom, prénom et numéro de sécurité sociale pour les membres du personnel.

4.2.3 Modélisation des états

Les états sont modélisés comme des sous-classes de la classe *État* de l'ontologie supérieure. Ces classes permettent de caractériser l'état d'un individu à un moment donné, aussi les individus peuvent ils appartenir successivement à différentes classes d'état. Par exemple, dans l'ontologie présentée dans la figure 4.9, deux classes d'état *Ouvert* et *Fermé* sont définies dans l'ontologie

du domaine. Les individus qui appartiennent à la classe *Bâtiment* peuvent appartenir à l'une de ces classes pour caractériser le fait que le bâtiment est ouvert ou fermé.

4.3 Synthèse

La modélisation du contexte sous forme d'ontologie permet de manipuler une représentation formelle de l'environnement qui peut être traitée par des programmes informatiques. La modélisation sous forme d'ontologie facilite le partage et la réutilisation de connaissances et permet l'utilisation de mécanismes de raisonnement logique sur les connaissances modélisées. Nous définissons une ontologie de contexte pour les environnements ubiquitaires composée d'une ontologie supérieure qui modélise les concepts communs à tous les environnements et d'une ontologie du domaine dont les concepts sont propres à un environnement particulier. Nous utilisons le langage OWL DL, qui est un standard du W3C et qui garantit la complétude et la décidabilité des raisonnements sur l'ontologie, pour spécifier les ontologies.

Manier savamment une langue, c'est pratiquer une espèce de sorcellerie évocatoire.

Charles Baudelaire, *L'Art romantique*

5

Approche langage pour la programmation d'applications ubiquitaires

La programmation d'applications ubiquitaires a pour but de combiner et structurer les services disponibles dans un environnement ubiquitaire pour assister les utilisateurs dans leurs activités au sein de l'environnement. Afin de rendre la programmation des applications ubiquitaires simple et intuitive, le programmeur doit être déchargé des problèmes d'implémentation de bas niveau liés à la découverte de services, à la sélection de services et aux détails techniques d'interaction entre services. De plus, les applications ubiquitaires doivent réagir aux informations de contexte afin de fournir des services pertinents aux utilisateurs. Le paradigme de programmation doit permettre de spécifier simplement un comportement sensible au contexte pour les applications ubiquitaires.

Notre approche est basée sur le paradigme de tâche [60] qui permet de "faire le lien entre les tâches (ce que les utilisateurs veulent accomplir) et les services (les fonctionnalités disponibles)". Plus particulièrement, nous nous appuyons sur un langage de règles logiques pour spécifier le comportement des applications ubiquitaires à partir des abstractions décrites dans le modèle de contexte. Les concepts définis dans le modèle servent de briques de base pour la programmation des applications ubiquitaires à un haut niveau d'abstraction. Ainsi, le programmeur de l'environnement ne se focalise pas sur les détails techniques d'implémentation de l'application, comme la découverte des services disponibles dans l'environnement, où ces services sont exécutés, d'où proviennent les informations de contexte, mais uniquement sur la tâche que l'application doit réaliser. C'est l'infrastructure de l'environnement ubiquitaire qui a pour rôle d'établir la relation entre la spécification haut niveau de l'application sous forme de tâche et les ressources disponibles dans l'environnement.

Dans ce chapitre, nous étudions dans un premier temps le langage de règles logiques permettant la programmation d'applications ubiquitaires, puis nous présentons le mécanisme par lequel

ce langage permet de s'abstraire du caractère dynamique des environnements ubiquitaires. Finalement, nous présentons le paradigme de programmation permettant la spécification des tâches sous forme de règles logiques.

5.1 Langage de règles pour les environnements ubiquitaires

Comme présenté dans la section 4.1.3, le langage OWL permet de spécifier des ontologies sous forme de classes et de propriétés. Cependant, l'expressivité du langage OWL est limitée lorsqu'il s'agit d'exprimer des relations complexes à propos de ces classes et propriétés. En effet, le langage OWL ne permet pas d'exprimer de relations de composition entre propriétés. Par exemple, il est impossible de spécifier la propriété *Oncle* comme étant la composition de la relation *Parent* et *Frère*. Un autre exemple est l'incapacité de définir une classe *VolInternational* comme un sous-ensemble de la classe *Vol* impliquant des individus de la classe *Aéroport* appartenant à différents *Pays*. De plus, OWL ne permet pas de définir des classes basées sur la valeur des propriétés de types de données. Par exemple, il est impossible de définir une classe *Adulte* comme l'ensemble des individus de la classe *Personne* dont la propriété *Âge* possède une valeur supérieure à 18.

Afin de pallier ce problème d'expressivité, une solution consiste à étendre le langage de spécification d'ontologies avec un langage de règles permettant d'exprimer des relations concernant les classes, les propriétés et les individus d'une ontologie. Plusieurs travaux proposent des langages de règles permettant d'étendre l'expressivité du langage OWL [38, 39, 10].

De manière générale, un langage de règles est composé d'un *antécédent* et d'un *conséquent*, chacun composé de zéro ou plus *atomes*. Un atome est un prédicat possédant une arité de un ou plus. Les atomes peuvent faire référence aux éléments d'une base de connaissances afin d'exprimer des conditions sur les classes, les propriétés et les individus d'une ontologie. Le sens d'une règle est le suivant: lorsque la condition spécifiée dans l'antécédent est vraie, alors la conclusion spécifiée dans le conséquent est tenue pour vraie.

Nous nous intéressons particulièrement au langage SWRL [39] qui est soumis comme standard au W3C et permet d'étendre le langage OWL avec des règles logiques. SWRL s'impose aujourd'hui comme un standard de fait en tant que langage de règles pour le Web Sémantique, notamment grâce aux outils développés pour la spécification de règles dans l'environnement d'édition Protégé [27].

5.1.1 Raisonner sur les ontologies

Le raisonnement sur les ontologies est réalisé en appliquant un ensemble de règles aux faits contenus dans une base de connaissances. L'exécution des règles est effectuée par un moteur d'inférence. Le fonctionnement général d'un moteur d'inférence est présenté dans la figure 5.1. Le moteur d'inférence accepte en entrée un ensemble de règles et un ensemble de faits provenant d'une base de connaissances. Le mécanisme de raisonnement permet au moteur d'inférer de nouveaux faits à partir des faits qui sont stockés dans la base de connaissances. Les nouveaux

faits inférés sont à leur tour stockés dans la base de connaissances et sont utilisés pour exécuter de nouvelles règles.

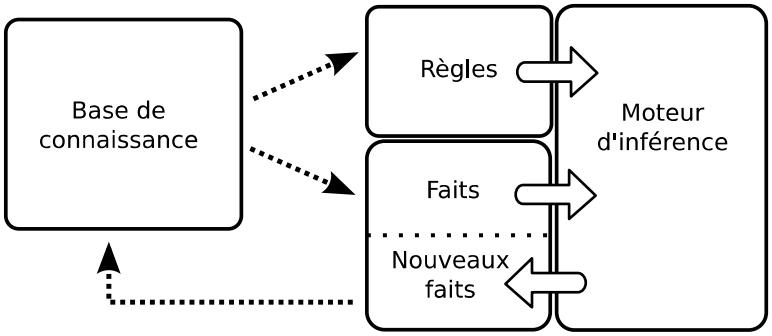

FIG. 5.1 – Fonctionnement général d'un moteur d'inférence

5.1.2 Semantic Web Rule Language

Le langage de règles pour le Web Sémantique SWRL est une combinaison des sous-langages OWL Lite et OWL DL du langage OWL avec la restriction unaire/binaire du langage RuleML (Rule Markup Language) [72]. Ce langage est un langage de règles logiques inspiré du langage Prolog [82]. SWRL étend le langage OWL en ajoutant des axiomes de règles aux axiomes OWL déjà existants. La syntaxe abstraite de OWL est étendue afin d'inclure la syntaxe de ces règles et la sémantique de OWL est étendue afin de fournir une définition formelle des ontologies comportant des règles écrites avec cette syntaxe.

5.1.2.1 Syntaxe de SWRL

Une règle SWRL est constituée d'un antécédent et d'un conséquent, composés chacun de zéro ou plusieurs atomes. Lorsque l'antécédent ou le conséquent est formé de plusieurs atomes, il est évalué comme la conjonction de ses atomes. Les atomes peuvent être de divers types:

C(x) où C représente une classe OWL et x est une variable, un individu ou une donnée.

P(x,y) où P représente une propriété objet ou une propriété de type de donnée, x est une variable ou un individu et y est une variable, un individu ou une donnée.

SameAs(x,y), DifferentFrom(x,y) où x et y sont des variables ou des individus.

BuiltIn(r,x$_0$,...,x$_n$) où r est une primitive et les x_i sont des variables, des individus ou des données.

Les atomes de la forme C(x) sont appelés atomes de classes et les atomes de la forme P(x,y) sont appelés atomes de propriétés. Les primitives (BuiltIns) sont des prédicats acceptant plusieurs arguments et permettent d'étendre l'expressivité du langage SWRL. La spécification du langage SWRL propose un ensemble de primitives pour la comparaison (supérieur, inférieur, etc), pour les opérations mathématiques (addition, soustraction,etc), pour la manipulation des chaînes de

caractères, des dates, des durées, des URIs et des listes. La liste de primitives proposée dans la spécification de SWRL n'est pas figée et est prévue pour être étendue au cours des versions du langage. Les règles écrites avec le langage SWRL doivent respecter la condition de sûreté (*safety condition*), c'est-à-dire que seules les variables qui apparaissent dans l'antécédent peuvent apparaître dans le conséquent.

SWRL possède une syntaxe XML pour spécifier des règles pouvant être traitées de manière logicielle, ainsi qu'une syntaxe à la Prolog permettant d'écrire les règles de manière plus lisible. Dans cette syntaxe simplifiée, une règle est de la forme *antécédent* $\rightarrow$ *conséquent*. L'antécédent et le conséquent sont une conjonction d'atomes notés $a_1 \wedge ... \wedge a_n$. Les variables sont préfixées avec un point d'interrogation (e.g. ?x). Par exemple, la règle spécifiant la propriété *Oncle* comme étant la composition de la relation *Parent* et *Frère* est notée comme suit.

Parent(?x,?y) $\wedge$ Frère(?y,?z) $\rightarrow$ Oncle(?x,?z)

La syntaxe simplifiée permet de noter les primitives sous une forme fonctionnelle. Les primitives SWRL sont préfixés par l'espace de nom *swrlb*. Par exemple, la primitive permettant de tester l'égalité est noté `swrlb:equal(?x,?y)` et est vraie si le premier argument et le deuxième argument sont égaux. Dans la suite du document, nous utiliserons cette syntaxe simplifiée pour l'écriture des règles SWRL.

5.1.2.2 Sémantique de SWRL

Une règle SWRL est interprétée comme une implication entre son antécédent et son conséquent. Ceci signifie que lorsque la condition spécifiée dans l'antécédent est vraie, alors la conclusion spécifiée dans le conséquent est tenue pour vraie. L'interprétation d'une règle SWRL consiste à établir une relation entre les atomes d'une règle et les éléments d'une ontologie OWL. L'antécédent et le conséquent d'une règle sont satisfaits si la conjonction de leurs atomes sont satisfaits. Un atome de classe C(x) est satisfait si x est un individu appartenant à la classe C. Un atome de propriété P(x,y) est satisfait si l'individu x est lié à l'individu y par la propriété P. Un atome SameAs(x,y) est satisfait si x est le même individu que y. Un atome DifferentFrom(x,y) est satisfait si x et y sont des individus différents. Enfin, un atome BuiltIn(r,x_1,...,x_n) est satisfait si la primitive r s'applique aux arguments x_i.

Pour reprendre l'exemple de la propriété composée *Oncle*, étant donné l'ontologie présentée dans la figure 5.2 (a) possédant trois individus John, Bob et Sam, {John, Bob} est une instance de la propriété *Parent* et {Bob, Sam} est une instance de la propriété *Frère*. Si on applique la règle Parent(?x,?y) $\wedge$ Frère(?y,?z) $\rightarrow$ Oncle(?x,?z) à cette ontologie, l'antécédent est satisfait si ?x prend la valeur John, ?y prend la valeur Bob et ?z prend la valeur Sam. Afin que le conséquent soit satisfait, une instance {John, Sam} de la propriété *Oncle* est créée dans l'ontologie, comme présenté dans la figure 5.2 (b).

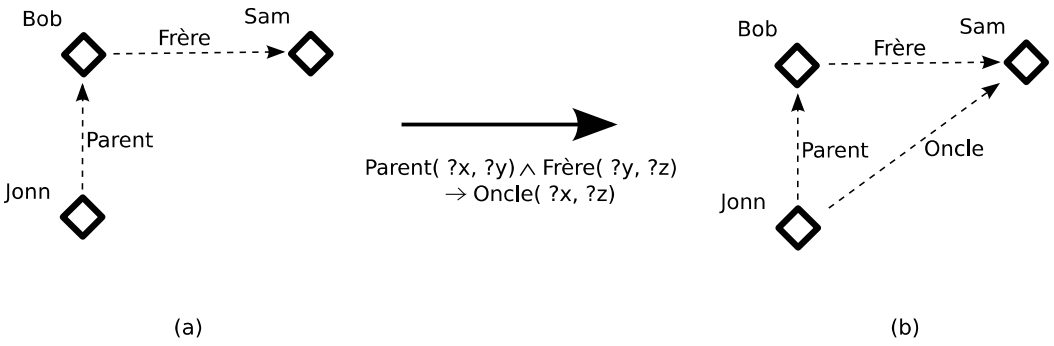

FIG. 5.2 – Interprétation d'une règle SWRL.

5.2 Adaptation aux évolutions de l'environnement

Les environnements ubiquitaires sont dynamiques par nature. Les périphériques entrent et sortent de l'environnement, ce qui entraîne une disponibilité dynamique des services. Afin de permettre au programmeur de l'environnement de développer des applications ubiquitaires à un haut niveau d'abstraction, le modèle de contexte doit refléter l'état de l'environnement, et en particulier la disponibilité des services. De plus, le langage de programmation doit offrir des abstractions de haut niveau au programmeur lui permettant d'intégrer les fonctionnalités offertes par les services de l'environnement dans la logique applicative.

5.2.1 Intégration des services

Lorsqu'un service apparaît dans l'environnement, il annonce sa présence à l'infrastructure par le biais d'un protocole de découverte de services. Au cours de cette phase de découverte, l'infrastructure intègre le nouveau service dans le modèle de contexte. Les services sont modélisés comme des individus appartenant à une sous-classe de la classe *Service* dans le modèle de contexte. Un service apparaissant dans l'environnement doit donc annoncer la classe de service à laquelle il appartient afin que l'infrastructure puisse l'intégrer dans le modèle de contexte. Si la classe de service à laquelle appartient le service existe dans le modèle, alors un individu représentant le service est ajouté à cette classe de service. Si au contraire cette classe n'existe pas, alors une nouvelle classe de service est créée dans le modèle et un individu est ajouté à cette classe. Lorsqu'un service annonce son départ, sa représentation dans le modèle de contexte est supprimée. Ainsi, la hiérarchie des sous-classes de la classe *Service* et les individus appartenant à ces classes dans le modèle de contexte constituent une représentation des services disponibles dans l'environnement. Les classes de services sont modélisées dans une ontologie, ce qui permet d'utiliser les classes de services et leurs individus dans les atomes d'un langage de règles. Par exemple, étant donnée l'ontologie présentée dans la figure 5.3, la conjonction suivante d'atomes permet de sélectionner l'ensemble des services pouvant enregistrer un flux multimédia fourni par une caméra.

Caméra(?caméra) ∧ Enregistrement(?serviceEnregistrement) ∧ Expose(?caméra,

?serviceEnregistrement)

Lorsque cette règle est exécutée, la variable ?serviceEnregistrement prend pour valeur l'ensemble des individus appartenant à la classe *Enregistrement* tel que ces individus soient liés à un individu de la classe *Caméra* par la propriété *Expose*. Les individus identifiés par la variable ?serviceDeRotation représentent les services permettant d'enregistrer un flux provenant d'une caméra.

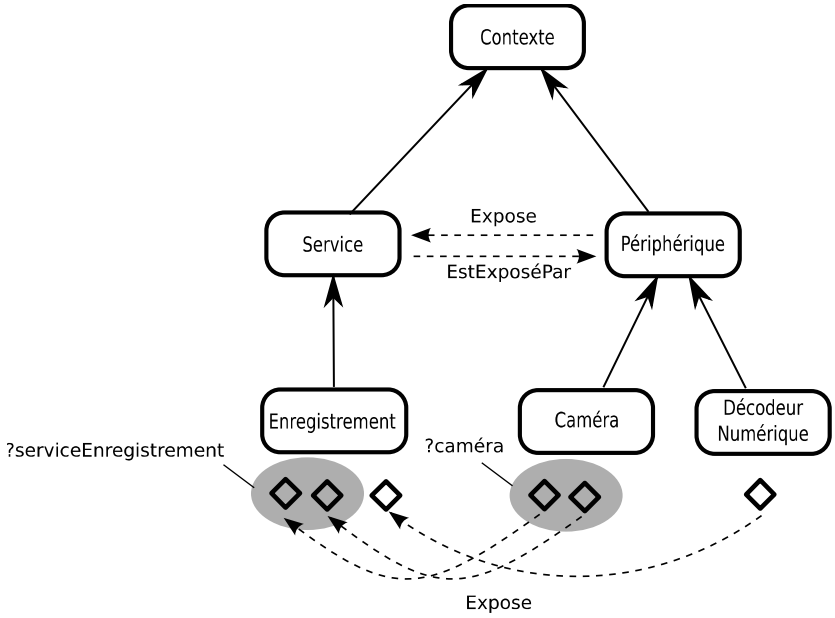

FIG. 5.3 – Sélection des services d'enregistrement hébergés pas des caméras.

5.2.2 Extensions au langage

L'intégration des services dans le modèle de contexte permet de manipuler une représentation des services disponibles dans l'environnement grâce à un langage de règles. Afin d'offrir des abstractions de haut niveau pour accéder aux fonctionnalités offertes par ces services, nous étendons dynamiquement la syntaxe du langage de règles utilisé pour la programmation des applications ubiquitaires. Lorsqu'un service est découvert dans l'environnement, le langage de règles est étendu avec de nouveaux atomes représentant les fonctionnalités offertes par ce service. Ces extensions prennent la forme de primitives pour le langage SWRL. Le programmeur de l'environnement peut alors utiliser ces primitives pour faire appel aux fonctionnalités des services de l'environnement dans la logique applicative des applications ubiquitaires. La syntaxe des primitives représentant les fonctionnalités des services est la suivante.

`ClasseService:op(?s, ?`r_0`, ..., ?`r_n`, ?`x_0`, ..., ?`x_n`)`

ClasseService est la classe du service fournissant la fonctionnalité, *op* est le nom de la fonctionnalité, ?s est un individu appartenant à la classe *ClasseService*, les x_i sont les paramètres d'entrée et les r_i sont les paramètres de retour. Par exemple, pour un service de positionnement de caméra appartenant à la classe *RotationCaméra*, fournissant une fonctionnalité de rotation *rotation* acceptant comme paramètre d'entrée un entier représentant l'angle de rotation, le langage de règles sera étendu avec une primitive `RotationCaméra:rotation-(?serviceDeRotation,?angle)` permettant au programmeur de contrôler la rotation des caméras dans la logique applicative des applications ubiquitaires. Les primitives générées sont sémantiquement typées afin d'assurer que le type des paramètres passés par le programmeur corresponde bien à celui qui est attendu. Ceci permet de vérifier que le paramètre ?s correspond bien à un individu de la classe *ClasseService*, et que les x_i et r_i correspondent bien aux types attendus par la primitive. Dans le cas contraire, l'atome est évalué comme faux.

5.2.3 Approche générative

Lorsque le langage de règles est étendu avec une primitive représentant une fonctionnalité, l'infrastructure génère le code permettant d'invoquer cette fonctionnalité et charge dynamiquement ce code dans le moteur d'inférence. Nous avons présenté le fonctionnement général d'un moteur d'inférence dans le section 5.1.1. De manière générale, lorsqu'une règle est exécutée, le moteur d'inférence est capable d'inférer de nouveaux faits. Le fonctionnement du moteur d'inférence étendu avec le code généré est sensiblement différent. Ce fonctionnement est présenté dans la figure 5.4. L'exécution d'une règle par le moteur d'inférence étendu peut produire deux effets.

– Le moteur d'inférence peut inférer de nouveaux faits.
– Le moteur d'inférence peut interagir avec les services de l'environnement.

Lorsque le moteur d'inférence évalue une primitive représentant une fonctionnalité, le code généré pour cette primitive est exécuté. L'exécution de la règle se traduit alors par une invocation sur le service passé en argument à la primitive. Si cet argument est une variable, le moteur d'inférence invoque successivement chacun des services désignés par la variable.

L'approche générative permet au moteur d'inférence d'interagir avec des services intégrés dynamiquement dans le modèle de contexte.

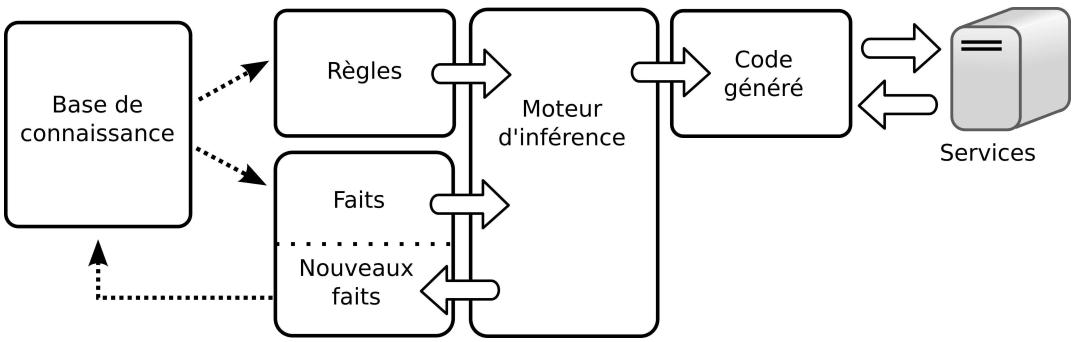

FIG. 5.4 – Fonctionnement du moteur d'inférence avec le code généré.

5.3 Spécification des tâches

Lors de la programmation d'une application ubiquitaire, le programmeur ne connaît pas les instances de services qui seront disponibles lors de l'exécution de l'application. Il spécifie le comportement de l'application à l'aide des concepts définis dans le modèle de contexte. Le modèle de contexte fournit des abstractions haut niveau qui permettent de programmer les applications ubiquitaires de manière simple et intuitive. La logique applicative est spécifiée sous la forme de tâches composées d'activités qui peuvent être de trois types.

Gestion de contexte. Une application ubiquitaire doit être sensible au contexte afin d'adapter son comportement en fonction des circonstances observées dans l'environnement. Les activités de gestion de contexte permettent d'effectuer des requêtes sur le modèle de contexte afin d'obtenir des informations sur l'état de l'environnement physique, et d'inférer des informations de plus haut niveau à partir des informations obtenues.

Requête de service. Une requête de service permet de sélectionner un ou plusieurs services disponibles dans l'environnement.

Interaction avec un service. Ce type d'activité permet d'invoquer une fonctionnalité disponible dans l'environnement. Cette interaction peut nécessiter l'envoi de données au service qui peut fournir des données en retour. Une interaction avec un service peut également se concrétiser par la réalisation d'une action dans l'environnement physique.

Une tâche est composée par l'assemblage de ces trois types d'activités. La programmation d'une application ubiquitaire consiste à spécifier un ensemble de tâches qui sont construites à partir des abstractions haut niveau décrites dans le modèle de contexte.

5.3.1 Gestion de contexte

Les activités de gestion de contexte sont composées d'atomes de classes et d'atomes de propriétés qui sont relatifs aux informations de lieux, de personnes et de paramètres physiques. Ces activités permettent d'extraire des informations spécifiques de la base de connaissances et d'inférer des informations de plus haut niveau. Par exemple, étant donnée l'ontologie présentée dans la figure 5.5, la conjonction suivante d'atome permet d'extraire de la base de connaissances l'ensemble des lieux d'intérieur où des personnes sont présentes.

$$\text{Intérieur}(?\text{lieu}) \wedge \text{Contient}(?\text{lieu}, ?\text{personne})$$

On peut définir la classe *Occupé* comme l'ensemble des lieux contenant des personnes grâce à la règle suivante.

$$\text{Intérieur}(?\text{lieu}) \wedge \text{Contient}(?\text{lieu}, ?\text{personne}) \rightarrow \text{Occupé}(?\text{lieu})$$

L'exécution de cette règle par le moteur d'inférence a pour conséquence de classifier l'ensemble des individus appartenant à la classe *Intérieur* et contenant au moins une personne comme appartenant à la classe *Occupé*.

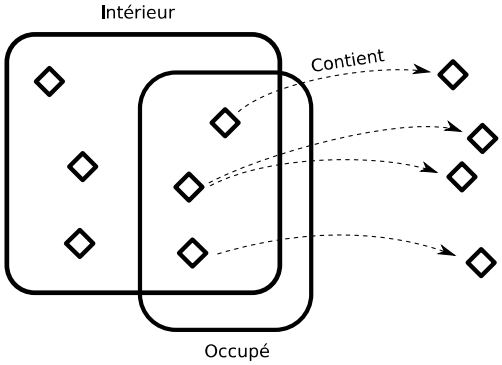

FIG. 5.5 – Exemple de gestion des informations de contexte.

5.3.2 Sélection des services

La sélection de service permet d'extraire un ensemble de services de la base de connaissances. La sélection de tous les services d'une classe *A* disponibles dans l'environnement se fait à l'aide d'un unique atome A(?x). Grâce à la relation de subsomption, les individus appartenant aux sous-classes de la classe *A* sont également sélectionnés. La figure 5.6 présente la hiérarchie de services extraite du modèle de contexte de la figure 4.9. Les fonctionnalités offertes par les services sont mentionnées à gauche de chaque classe de service. L'atome RotationCaméra(?rotcam) sélectionne l'ensemble des services permettant de contrôler la rotation des caméras mobiles. Les services de la classe *RotationCamera* qui disposent de la fonctionnalité *rotation* sont sélectionnés. Par application de la relation de subsomption, les services de la classe *MouvementCaméra* sont également sélectionnés. Ces services disposent également de la fonctionnalité *rotation* ainsi que des fonctionnalités additionnelles *inclinaison* et *mouvement*.

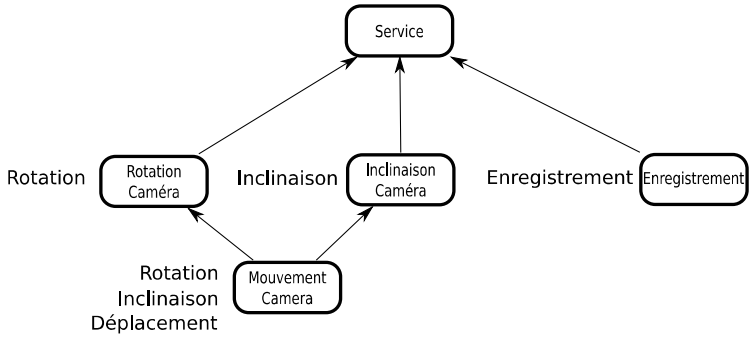

FIG. 5.6 – Hiérarchie de services d'un modèle de contexte

L'activité de sélection de services permet d'exprimer des contraintes sur les services afin de restreindre la sélection aux services respectant ces contraintes. Par exemple, la conjonction suivante d'atomes permet de sélectionner les services de rotation de caméra hébergés par une caméra possédant une résolution de 3 mégapixels.

RotationCaméra(?rotcam) ∧ EstExposéPar(?rotcam, ?périphérique) ∧ Caméra(?périphérique) ∧ Résolution(?périphérique,3000000)

Les fonctionnalités des services ainsi sélectionnés peuvent être invoquées grâce aux extensions syntaxiques du langage de règles.

5.3.3 Interaction avec les services

Une fois que des services ont été sélectionnés, le programmeur de l'environnement peut spécifier des interactions avec ces services grâce aux primitives générées pour ce type de service. Lors de la découverte du service de rotation de caméra présenté dans la figure 5.6, une extension syntaxique est générée pour la fonctionnalité *rotation* sous la forme d'une primitive `RotationCamera:rotation(?serviceRotation,?angle)`. Cette primitive permet d'invoquer la fonctionnalité *rotation* d'un service appartenant à la classe *RotationCamera* passé en premier argument. La caméra concernée effectuera une rotation d'un angle dont la valeur est passée en second argument.

Par exemple, pour effectuer une rotation à 360 degrés de toutes les caméras possédant un service de rotation, le programmeur sélectionne l'ensemble des services de rotation de caméra grâce à l'atome RotationCaméra(?rotcam), et passe le résultat de cette sélection par l'intermédiaire de la variable ?rotcam à la primitive représentant la fonctionnalité de rotation. La règle complète permettant de spécifier cette tâche est la suivante:

RotationCaméra(?rotcam) → RotationCamera:rotation(?rotcam,360)

Lorsque cette règle est exécutée, le moteur d'inférence extrait l'ensemble des individus appartenant à la classe *RotationCaméra*, et exécute le code associé à la primitive `RotationCamera:rotation` pour chacun de ces individus. Ceci se traduit par une invocation de la fonctionnalité de rotation sur chacun de ces services.

5.4 Synthèse

L'utilisation d'un langage de règles pour programmer les environnements ubiquitaires permet d'utiliser les concepts définis dans le modèle de contexte comme briques de base pour les applications ubiquitaires. Nous étendons le langage de règles avec des extensions syntaxiques permettant de spécifier les interactions avec les services de l'environnement dans la logique applicative des applications ubiquitaires. De plus, nous générons dynamiquement le code permettant au moteur d'inférence d'invoquer les fonctionnalités des services lors de l'évaluation des règles logiques. La programmation par règles logiques permet donc de manipuler les éléments de contexte et les fonctionnalités des services dans un paradigme unique. Le fait de pouvoir exprimer la logique applicative avec les concepts définis dans le modèle de contexte permet de programmer les environnements ubiquitaires à un haut niveau d'abstraction.

Créer, c'est vivre deux fois.

Albert Camus, *Le Mythe de Sisyphe*

6 L'infrastructure PERSEWS

Le modèle de contexte et l'approche langage présentés dans les chapitres précédents sont mis en œuvre dans l'architecture PERSEWS afin de permettre la programmation d'applications ubiquitaires. Cette infrastructure permet de développer des applications en utilisant un paradigme de tâche. La logique applicative est spécifiée sous forme de règles logiques dont les atomes sont les concepts du modèle de contexte, fournissant ainsi des abstractions de haut niveau au programmeur de l'environnement. L'approche générative mise en œuvre dans l'infrastructure PERSEWS permet d'intégrer dynamiquement les services qui apparaissent dans l'environnement: le langage de règles est étendu dynamiquement avec des abstractions permettant au programmeur de manipuler les fonctionnalités des services dans le paradigme de tâche et le support logiciel permettant l'invocation de ces services est généré automatiquement. Dans les sections suivantes, nous présentons les composants de l'architecture PERSEWS, le modèle architectural sur lequel cette infrastructure est construite et les éléments technologiques qui permettent sa réalisation.

6.1 Composants de l'infrastructure PERSEWS

L'infrastructure PERSEWS est constituée de cinq composants: l'annuaire, le générateur, le moteur d'inférence, le gestionnaire d'événements et l'éditeur de règles. La figure 6.1 présente ces différents composants ainsi que leurs interactions. Nous détaillons dans cette section chacun de ces composants et son rôle dans l'infrastructure PERSEWS.

6.1.1 Annuaire

L'annuaire est le composant central de la découverte de services. Les services qui pénètrent dans l'environnement ubiquitaire annoncent leur présence à l'infrastructure en s'enregistrant au-

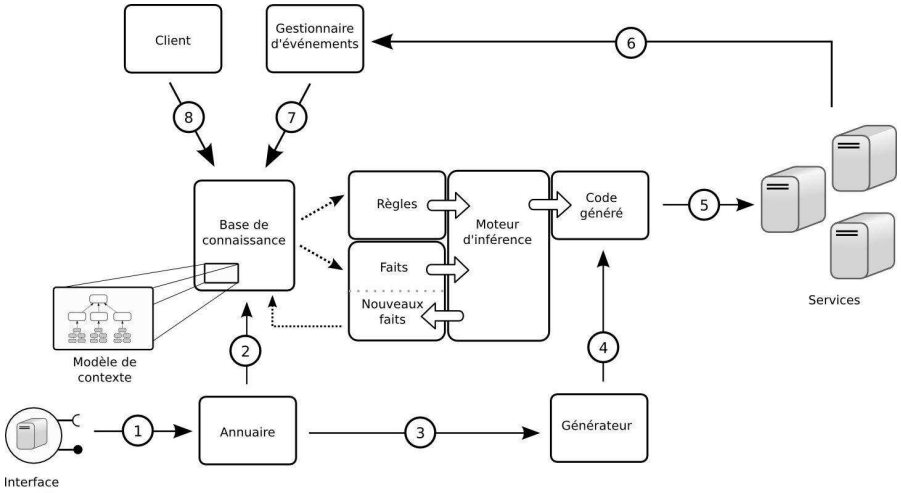

FIG. 6.1 – Composants de l'infrastructure PERSEWS

près de l'annuaire (phase 1 de la figure 6.1), et annoncent leur départ en résiliant leur inscription auprès de l'annuaire. L'annuaire accepte en entrée une description sémantique de l'interface d'un service. Cette description spécifie les fonctionnalités offertes par le service, les types d'entrée et de sortie de ses fonctionnalités, ainsi que la classe de service à laquelle le service appartient. L'annuaire exploite ces informations pour ajouter un individu de la classe de service adéquate dans la base de connaissances (phase 2 de la figure 6.1), qui représente le service souscripteur dans le modèle de contexte. Lors de la résiliation d'un service auprès de l'annuaire, l'annuaire supprime l'individu représentant ce service dans la base de connaissances.

L'annuaire produit également les extensions syntaxiques permettant d'invoquer les fonctionnalités du service à l'aide du langage de règles. Pour ce faire, l'annuaire ajoute à la base de connaissances une primitive pour chaque fonctionnalité offerte par le service. Finalement, l'annuaire transmet l'interface du service souscripteur au générateur afin de produire le code associé à chaque primitive (phase 3 de la figure 6.1).

6.1.2 Générateur

Le générateur a pour rôle de produire le code qui permet au moteur d'inférence d'invoquer les services de l'environnement. Le générateur exploite les descriptions de services fournies par l'annuaire pour produire le code permettant d'invoquer les fonctionnalités de ces services. Ce code est chargé dynamiquement dans le moteur d'inférence (phase 4 de la figure 6.1), ce qui lui permet d'invoquer les services de l'environnement. Le code généré est spécifique à une fonctionnalité, mais est utilisé par le moteur d'inférence pour invoquer les différents services possédant cette fonctionnalité. Par exemple, dans la hiérarchie de services présentée dans la figure 5.6, un service de type *RotationCamera* peut publier sa description de service. Le code permettant d'invoquer la fonctionnalité *rotation* est alors produit par le générateur. Ce même code est utilisé pour invoquer la fonctionnalité *rotation* de tous les services de la classe *RotationCamera*.

6.1.3 Moteur d'inférence

Comme introduit dans la section 5.1.1, le moteur d'inférence applique des règles logiques spécifiées par le programmeur de l'environnement aux faits contenus dans le base de connaissances. La base de connaissances contient le modèle de contexte défini par l'architecte de l'environnement, enrichi par les individus représentant les services de l'environnement. Lorsqu'une primitive représentant une fonctionnalité de service est évaluée par le moteur d'inférence, le code généré pour cette fonctionnalité est exécuté, ce qui se traduit par une invocation de service (phase 5 de la figure 6.1). Les primitives permettant de manipuler les fonctionnalités des services acceptent un argument qui spécifie le service sur lequel la fonctionnalité sera invoquée. Cet argument peut être une variable, ce qui permet d'invoquer une fonctionnalité sur un ensemble de services.

Le moteur d'inférence propose deux modes d'exécution des règles:

- Dans le mode d'exécution unique, les règles sont exécutées une seule fois avant d'être supprimées de la base de connaissances.
- Dans le mode d'exécution perpétuel, les règles sont exécutées continuellement.

Ces deux modes d'exécution permettent au programmeur d'interagir de manière ponctuelle avec les services de l'environnement, ou au contraire de spécifier des règles qui régissent l'environnement de manière continue.

6.1.4 Gestionnaire d'événements

Le gestionnaire d'événements permet à l'infrastructure d'interagir avec les services qui communiquent par échange de messages sur un mode publication/souscription. Ce mode de communication est exploité par les services capables d'acquérir des informations sur le contexte afin de propager ces informations aux autres services de l'environnement. Lorsqu'un service de publication s'enregistre auprès de l'annuaire, celui-ci transmet la description du service au gestionnaire d'événements qui souscrit au service afin de recevoir les messages publiés. Les messages publiés pas les services de l'environnement sont reçus par le gestionnaire d'événements (phase 6 de la figure 6.1) qui traite ces messages et stocke les informations sur le contexte dans la base de connaissances (phase 7 de la figure 6.1). Ainsi, la base de connaissances reflète en permanence l'état du contexte capté par l'ensemble des sources dans l'environnement. Ces informations peuvent être utilisées dans les activités de gestion de contexte lors de la spécification des tâches.

Par exemple, un service de détection de périphériques bluetooth peut émettre des événements de détection. La charge utile de ces événements précise le périphérique qui a été détecté. Lorsque le gestionnaire d'événements reçoit un évènement du détecteur bluetooth, il met à jour la propriété *EstLocalisé* du périphérique à la position du détecteur bluetooth. L'interprétation sémantique des événements reçus et le mécanisme de traduction entre le format du message et sa représentation sémantique sont spécifiés dans le contrat de service sémantique, qui est présenté en section 6.2.3.1.

6.1.5 Éditeur de règles

L'éditeur de règles est un environnement de développement permettant de programmer l'environnement en spécifiant des tâches sous forme de règles logiques. L'éditeur de règles exploite les informations contenues dans le modèle de l'environnement pour guider le programmeur. L'interface de programmation de l'éditeur fournit un ensemble de fonctions facilitant la programmation par règles logiques. Ces fonctions sont le menu contextuel, la complétion et la vérification.

Le menu contextuel propose au programmeur l'ensemble des classes, des propriétés, des individus et des primitives qui sont disponibles dans le modèle de contexte au moment où il écrit une règle. Grâce au menu contextuel, le programmeur dispose d'une vue d'ensemble des concepts qu'il peut utiliser pour programmer l'environnement.

La complétion assiste le programmeur en complétant automatiquement le nom des atomes en fonction des classes, propriétés, individus et primitives disponibles dans le modèle de contexte.

La vérification permet d'assurer la correction syntaxique des règles spécifiées. L'éditeur vérifie que les atomes respectent la syntaxe du langage SWRL. De plus, en ce qui concerne les extensions syntaxiques, l'éditeur de règles vérifie que le nombre et le type des arguments correspondent bien à ceux attendus par la primitive.

Une fois l'édition de règles achevée, l'éditeur de règles permet d'envoyer les règles dans la base de connaissances afin qu'elles soient exécutées par le moteur d'inférence (phase 8 de la figure 6.1).

6.2 Architecture orientée service

Le terme d'architecture orientée service désigne un modèle d'architecture logicielle pour l'exécution d'applications réparties. L'élément structurel de base d'une architecture orientée service est le service. Un service est une entité logicielle exposant un certain nombre de fonctionnalités, dont l'interface est bien définie par l'intermédiaire d'un contrat de service, dont les modalités d'accès réseaux sont standardisées et qui peut être invoqué sans connaissance préalable des détails techniques d'implémentation. L'infrastructure PERSEWS est construite sur le modèle d'architecture orientée service, chacun de ses composants étant développé sous forme de service.

Comme présenté dans la section 2.1.1.4, les services d'une architecture orientée service sont hébergés par des fournisseurs de services et invoqués par des consommateurs de services. Les fournisseurs annoncent leurs services en publiant les contrats de services dans un annuaire, et les consommateurs de services découvrent les services disponibles en émettant des requêtes auprès de l'annuaire. Dans la suite de cette section, nous détaillons la notion de service qui est inhérente aux architectures orientées service, puis nous présentons les Services Web qui constituent l'implémentation de référence des architectures orientées service. Finalement, nous présentons les Services Web Sémantiques qui fournissent un cadre pour l'intégration de descriptions sémantiques dans les Services Web.

6.2.1 Notion de service

La notion de service est centrale aux architectures orientées service. Un service est une entité logicielle qui donne accès à une ou plusieurs fonctionnalités accessibles au moyen de protocoles réseaux. Les communications entre services dans une architecture orientée service se font par échange de messages. Afin d'invoquer un service, le consommateur de service envoie un message au fournisseur de service. Celui-ci traite le message et renvoie un message de réponse au consommateur de service. Les fonctionnalités offertes par un service sont décrites dans un contrat de service qui est utilisé pour annoncer ces fonctionnalités aux consommateurs. Le contrat de service spécifie la structure des messages d'entrée et de sortie du service, les protocoles d'accès au service et peut aussi spécifier des critères de qualité du service.

Les architectures orientées service permettent de garantir un couplage lâche entre les services. Ceci signifie que les consommateurs peuvent invoquer les services sans connaissance préalable des détails technologique d'implémentation tels que le langage de programmation utilisé ou la plate-forme d'exécution. Un service apparaît aux consommateurs comme une boite noire, dont la seule partie visible est le contrat de service qui décrit les modalités d'accès au service.

6.2.2 Services Web

Les Services Web [94] sont une réification des architectures orientées service dans la mesure où l'architecture de Services Web définit un ensemble de standards permettant de décrire les interfaces des services, de publier ces interfaces et d'échanger des messages entre services au travers d'un réseau. Ces standards ont pour but de permettre l'interopérabilité entre des systèmes informatiques connectés en réseau. Les Services Web permettent une interaction machine à machine entre des applications écrites dans des langages différents et s'exécutant sur des plates-formes hétérogènes. L'interopérabilité des Services Web repose sur l'utilisation du langage XML qui constitue un modèle de donnée unique pour l'échange d'informations et pour la description des interfaces de services ainsi que sur l'utilisation de protocoles réseaux largement répandus sur l'Internet pour le transport des données entre services. Les composants de l'architecture PERSEWS sont implémentés sous forme de Services Web. De même, les services de l'environnement doivent être compatibles avec les protocoles des Services Web pour s'intégrer dans l'infrastructure PERSEWS. Les sections suivantes décrivent les protocoles utilisés pour la spécification des contrats de services, l'échange de message et la découverte de services dans l'architecture des Services Web.

6.2.2.1 Contrat de service

La spécification de contrat de service dans l'architecture des Services Web repose sur le langage WSDL [19] qui est un dialecte XML. Comme présenté dans la figure 6.2, la description WSDL du contrat de service contient cinq éléments principaux.

L'élément Description est l'élément racine du document WSDL. Il spécifie le nom du service et les espaces de noms utilisés dans le reste du document.

L'élément Types définit les types de données utilisés dans les messages échangés entre consommateurs et fournisseurs. Les types de données sont décrits à l'aide du langage XML Schema.

L'élément Interface définit les fonctionnalités supportées par le service sous forme d'opérations. Une opération spécifie les messages que le service émet et reçoit lors de l'invocation d'une fonctionnalité.

L'élément Binding décrit le format d'encodage des messages et les protocoles réseaux utilisés pour le transport des messages. Cet élément définit de manière concrète comment les messages sont reçus et émis par le service.

L'élément Service définit les adresses permettant d'invoquer les fonctionnalités du service. Ces adresses sont spécifiées sous forme d'URL.

A travers ces éléments, la description WSDL fournit tous les éléments permettant d'invoquer un Service Web. Le description WSDL est la seule partie du service visible aux consommateurs de services, et c'est ce document qui permet d'invoquer un Service Web comme une boite noire, indépendamment de l'implémentation du service.

6.2.2.2 Échange de message

L'échange de message entre Services Web se fait à l'aide du protocole SOAP [33]. Ce protocole léger, basé sur XML, permet d'échanger des informations typées et structurées au travers d'un réseau. La figure 6.3 présente un message de requête SOAP pour un service de rotation de caméra. L'élément racine du document SOAP est l'élément enveloppe, lui-même constitué d'un en-tête et d'un corps. L'en-tête est optionnel et permet de spécifier un ensemble d'informations sur le message afin de garantir certaines propriétés (sécurité, nœuds intermédiaires, etc) durant son acheminement. Le corps contient les données XML utiles. Les protocoles HTTP et SMTP sont des protocoles valides pour le transport des messages SOAP, le protocole HTTP étant actuellement le plus couramment utilisé.

6.2.2.3 Publication et découverte de services

La publication et la découverte de services dans l'architecture des Services Web se fait à l'aide du protocole UDDI (Universal Description Discovery and Integration) [65] qui définit un système d'annuaire distribué. Un annuaire UDDI supporte la publication et la recherche de descriptions de services et d'informations sur les fournisseurs de services. Les informations qui constituent un enregistrement UDDI sont composées de quatre types de structures de données:

businessEntity. Cette structure de donnée décrit les entreprises et fournisseurs de services. La structure businessEntity ne fournit pas de détails techniques mais des informations de haut niveau sur un fournisseur de service: nom, adresse postale, description textuelle, etc.

businessService. Cette structure de donnée permet de définir un ensemble logique de services. Aucune description technique de service n'est fournie, mais cette structure permet de regrouper des service sous une rubrique commune. Chaque structure businessService est liée à une seule structure businessEntity.

```
<description xmlns="http://www.w3.org/ns/wsdl"
         xmlns:impl="http://persews/RotationCamera"
         targetNamespace="http://persews/RotationCamera"

 <types>
  <xsd:schema
        xmlns:intf="http://persews/RotationCamera"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        targetNamespace="http://persews/RotationCamera">
    <xsd:element name="angle" type="xsd:int">
  </xsd:schema>
 </types>

 <interface name="RotationCamera">
   <operation name="rotation" pattern="http://www.w3.org/ns/wsdl/in-only">
     <input element="impl:rotation"/>
   </operation>
 </interface>

 <binding xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
       name="RotationCameraSoapBinding"
       interface="impl:RotationCamera"
       type="http://www.w3.org/ns/wsdl/soap"
       wsoap:version="1.1"
       wsoap:protocol="http://www.w3.org/2006/01/soap11/bindings/HTTP/">
   <operation ref="impl:rotation"/>
 </binding>

<service name="RotationCameraService"  interface="impl:RotationCamera">
   <endpoint name="RotationCamera"
          binding="impl:RotationCameraSoapBinding"
          address="http://.../RotationCameraService"/>
 </service>

</description>
```

FIG. 6.2 – Contrat d'un service de rotation de caméra.

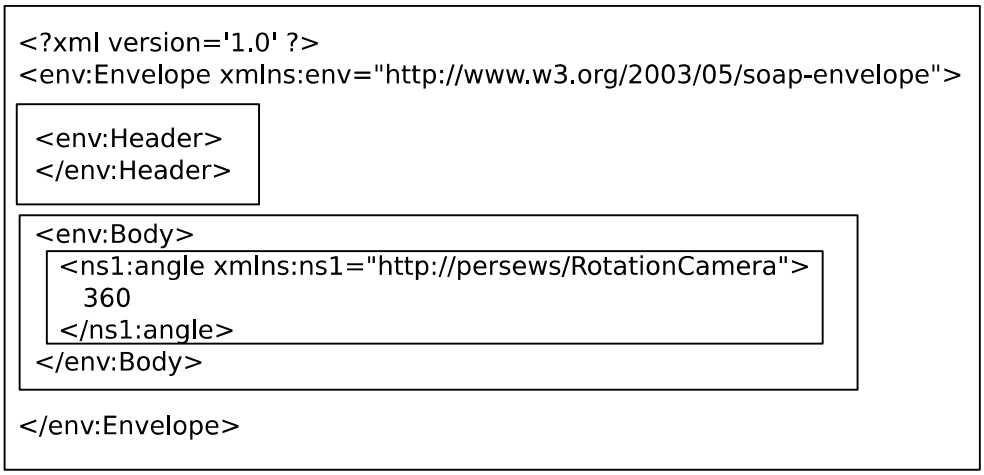

FIG. 6.3 – Requête SOAP au service de rotation de caméra.

bindingTemplate. Cette structure fournit les informations techniques requises pour invoquer un service. En particulier, l'adresse d'invocation du service est spécifiée dans cette structure. Une structure bindingTemplate faire référence aux structures tModel afin de décrire l'interface du service.

tModel. La structure tModel permet de décrire de manière unique un concept ou un point de conformité avec une spécification. Cette structure remplit deux rôles dans un enregistrement UDDI. Elle permet d'une part de décrire les interfaces des services publiés, et d'autre part de définir des taxonomies de services afin de faciliter la recherche dans l'annuaire UDDI.

categoryBag. Cette structure permet de définir des méta-données dans les structures tModel. Une structure categoryBag contient un ensemble de pointeurs vers des éléments de taxonomies définies par des structures tModel.

La figure 6.4 présente la relation entre les structures de données UDDI et les éléments du contrat de service WSDL 2.0 [17].

Un annuaire UDDI est lui-même un Service Web accessible via le protocole SOAP et dont l'interface est décrite dans un document WSDL. Afin de rendre un service disponible pour la découverte, un fournisseur peut publier son contrat de service auprès d'un annuaire UDDI. Lors de la publication du contrat WSDL, la relation entre les structures de données UDDI et les éléments du contrat de service WSDL est utilisée pour stocker les informations concernant le service.

La figure 6.5 présente un exemple d'enregistrement UDDI pour un service d'achat en ligne. La structure businessService (figure 6.5 (a)) spécifie l'adresse HTTP du service et fait référence aux structures tModel décrivant le binding et l'interface du service par leur identifiant unique. La structure tModel décrivant le binding (figure 6.5 (b)) indique l'adresse du fichier WSDL décrivant le service, ainsi que les protocoles pris en charge par le service pour l'échange de messages (SOAP sur HTTP dans ce cas). Finalement, la structure tModel décrivant l'interface (figure 6.5 (c)) indique l'adresse du fichier WSDL décrivant le service, ainsi que la catégorie à laquelle appartient l'interface.

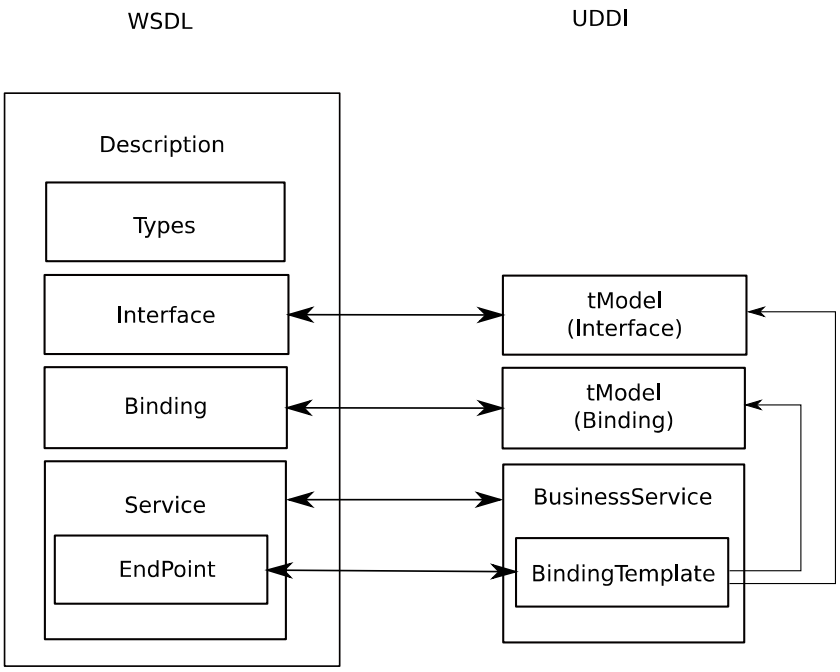

FIG. 6.4 – Relation entre les structures de données UDDI et les éléments du contrat de service WSDL.

6.2.2.4 Développement et déploiement

Un Service Web est un point d'accès atteignable au travers d'un réseau et dont l'interface est bien déterminée. Aucune contrainte n'est posée sur l'implémentation des fonctionnalités exposées par ce point d'accès. La figure 6.6 présente le mécanisme général de communication entre un consommateur et un fournisseur de service. De façon générale, un Service Web est un composant logiciel reposant sur un parseur SOAP. On peut distinguer le mécanisme d'envoi de message par le consommateur de service et le mécanisme de réception de message par le fournisseur de service.

Lors d'un envoi de message par le consommateur, l'application transmet les données utiles au talon client. Ce talon a pour rôle de sérialiser les données au format XML. Les données au format XML sont alors transmises au parseur SOAP qui a pour rôle de former un message SOAP valide à partir des données utiles. Ces données sont alors transmises au fournisseur de service via la couche de support protocolaire, qui forme un message valide conformément au protocole de transport choisi (par exemple HTTP), et le système d'exploitation.

Lors de la réception du message par le fournisseur, le message est transmis au serveur d'application via le système d'exploitation et la couche de support protocolaire. Le serveur d'applications a pour rôle d'accepter les requêtes provenant des clients et de les transmettre au parseur SOAP. Le parseur SOAP extrait les données utiles du message XML et les transmet au squelette serveur. Ce dernier désérialise les données au format XML pour les présenter à la couche applicative.

Le développement des Services Web est facilité par des outils qui permettent de générer

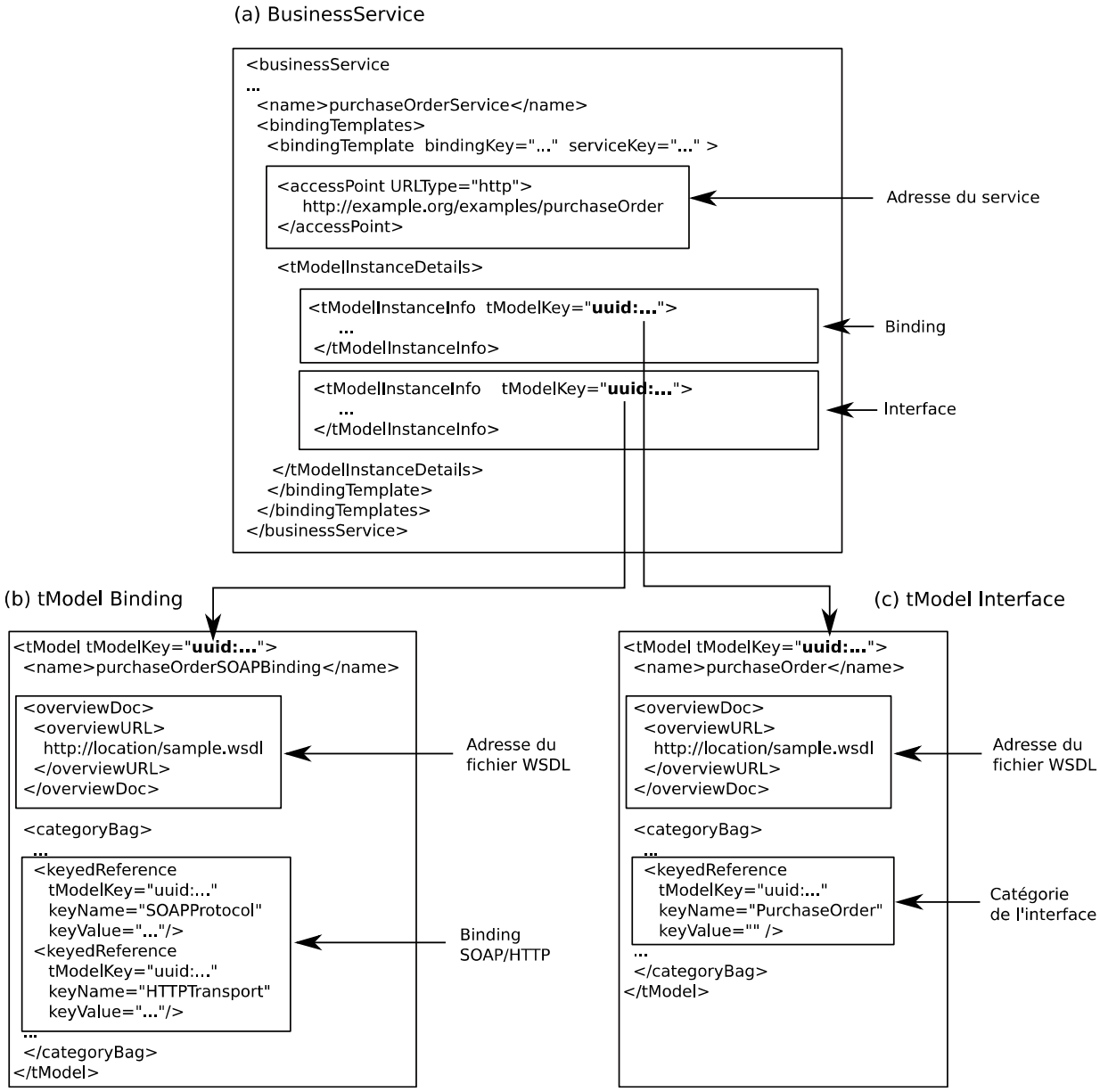

FIG. 6.5 – Structure businessService pour un service d'achat en ligne.

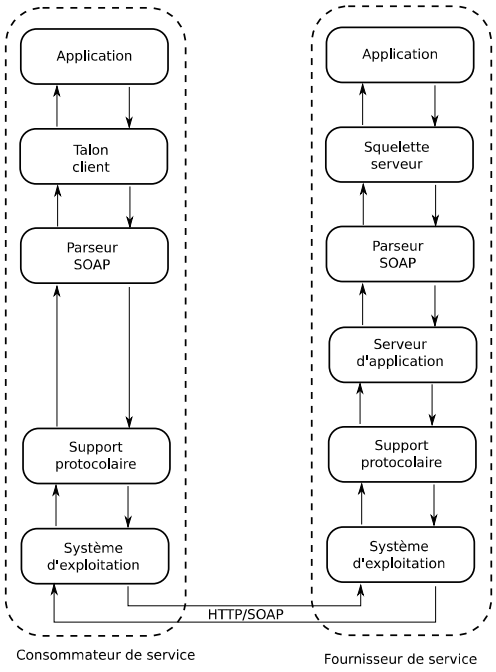

FIG. 6.6 – Mécanisme général de communication entre un consommateur et un fournisseur de service.

automatiquement le talon client et le squelette serveur à partir du contrat de service WSDL (cf. figure 6.7). Ainsi, la tâche de sérialisation et de désérialisation n'est pas à la charge du concepteur de service. Ces outils permettent d'intégrer simplement des applications existantes dans une architecture de Services Web, car le concepteur de service n'a qu'à développer la couche d'interface entre le squelette qui fournit des données désérialisées et l'application existante.

6.2.3 Services Web sémantiques

Les contrats de services spécifiés à l'aide du langage WSDL décrivent l'interface des services à un niveau syntaxique uniquement. Aucune information dans le document WSDL ne permet de connaître la signification (ou sémantique) des différents éléments qui le composent. Par exemple, différents services de rotation de caméra peuvent spécifier des noms de service différents dans leur description WSDL tels que RotationCamera, RotationDeCamera, CameraRotation, etc. Cet information syntaxique ne permet pas aux consommateurs de services de connaître la nature du service offert. De même, le nom des messages spécifiés dans la description WSDL et leur type (chaîne de caractères, entier, etc) ne permet pas aux consommateurs de connaître la nature des informations attendues pas le fournisseur de service. Par exemple, un message de type « chaîne de caractères » peut être un mot de passe, un nom d'utilisateur, ou tout autre information représentable sous forme de chaîne de caractères.

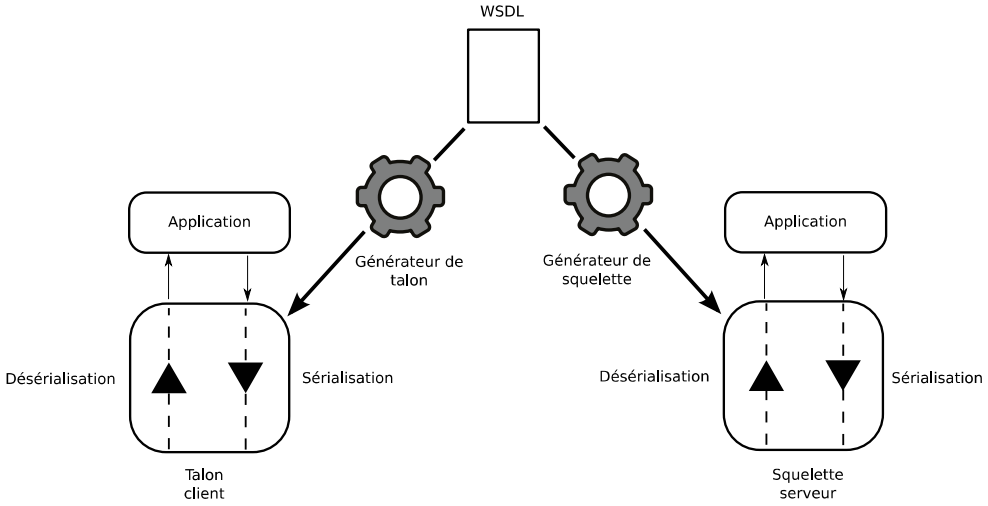

FIG. 6.7 – Génération des talons clients et des squelettes serveurs pour les Services Web.

6.2.3.1 Contrat de service sémantique

Afin de permettre aux consommateurs de services de disposer d'informations sémantiques à propos des fournisseurs de services, plusieurs spécifications proposent d'ajouter une description sémantique à la description syntaxique fournie par le document WSDL. Plusieurs spécifications [59, 69] proposent de modéliser les éléments du contrat de service sous forme de concepts dans une ontologie. Une partie du modèle permet de spécifier une relation entre le modèle sémantique et la description WSDL du service. Notre approche est basée sur la spécification SAWSDL [88] qui permet d'inclure des informations sémantiques sous forme d'annotations dans la description WSDL. Les annotation font référence à des concepts spécifiés dans une ontologie. Cette approche permet de conserver la compatibilité avec le standard WSDL, car la plupart des analyseurs de contrats WSDL ignorent les attributs qu'ils ne sont pas capable de traiter. De ce fait, ces analyseurs peuvent traiter des contrats SAWSDL comme des contrats WSDL standards.

Notre approche exploite la spécification SAWSDL pour établir la relation entre les contrats de services et les concepts du modèle de contexte. Le langage SAWSDL permet d'annoter les descriptions WSDL avec des références vers une ontologie OWL pour spécifier la relation entre les éléments WSDL ou XML Schema et les classes et individus du modèle de contexte. Les annotations peuvent porter sur les interfaces, les opérations et les types définis dans la description WSDL.

- Une annotation sur un élément interface permet de définir la classe de service à laquelle le service appartient.
- Une annotation sur un élément opération permet de fournir une description sémantique d'une fonctionnalité offerte par un service.
- Une annotation sur un élément de type permet de fournir une description sémantique des données échangées entre les services. Ceci permet de disposer d'une représentation des données partagée par l'ensemble des services dans l'environnement.

La figure 6.8 présente un extrait d'une interface sémantique pour un service d'agenda par-

tagé. L'annotation sur l'interface décrit ce service comme appartenant à la classe de service *Agenda*. L'annotation sur le type getAllEventResponse décrit ce type comme appartenant à la classe sémantique *Évènement*. Finalement, l'annotation sur l'opération getAllEvents décrit cette fonctionnalité comme appartenant à la classe sémantique *RécupèreToutÉvènement*.

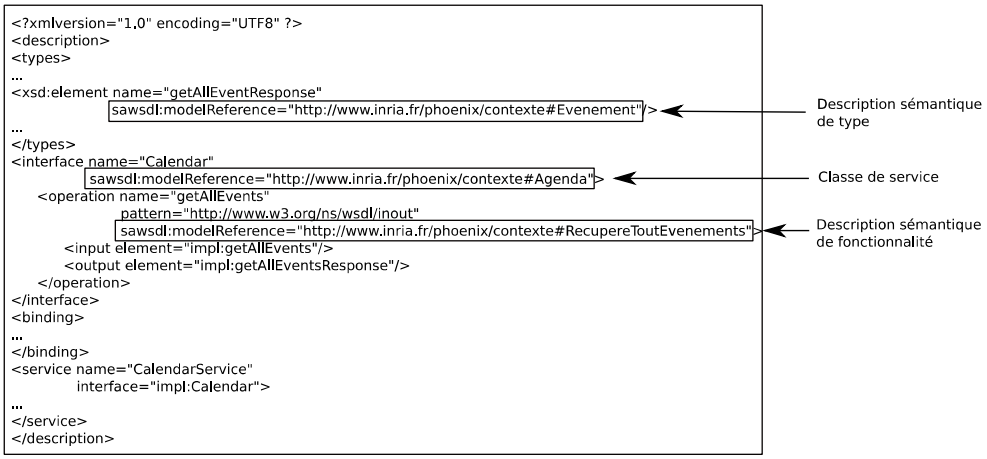

FIG. 6.8 – Exemple d'interface SAWSDL pour un service d'agenda partagé.

6.2.3.2 Schémas de traduction

Outre les annotations faisant référence aux concepts d'une ontologie, le langage SAWSDL permet d'annoter les types définis dans un document WSDL avec des schémas de traduction. Ces schémas de traduction peuvent être de type descendant ou montant. Un schéma de traduction montant permet de transformer des données XML depuis un message de Service Web vers un modèle sémantique (par exemple des données OWL dans une ontologie). Un schéma de traduction descendant permet la transformation inverse.

Comme présenté dans la figure 6.9, les schémas de traduction sont utiles pour communiquer avec un Service Web depuis un client sémantique. Par exemple, le client logiciel pourra utiliser le schéma descendant pour traduire certaines de ses informations et les envoyer au Service Web; il utilisera le schéma montant pour traduire les messages envoyés par le Service Web vers leur équivalent sémantique. Ces schémas de traduction sont exploités dans l'infrastructure PERSEWS par le moteur d'inférence lors des interactions avec les services pour transformer les données sémantiques spécifiées dans les règles vers le format XML attendu par le Service Web, et pour traduire le message de réponse du service vers son équivalent sémantique. Les schémas de traduction sont également exploités par le gestionnaire d'événements afin de transformer les événements reçus au format XML vers le format OWL.

L'utilisation du langage SAWSDL permet d'enrichir la description WSDL des Services Web, qui fournit les détails techniques d'invocation des services, avec des concepts sémantiques définis dans le modèle de contexte. Cette description à la fois syntaxique et sémantique des Services Web permet à l'infrastructure d'établir la relation entre les concepts du modèle de contexte et les services de l'environnement.

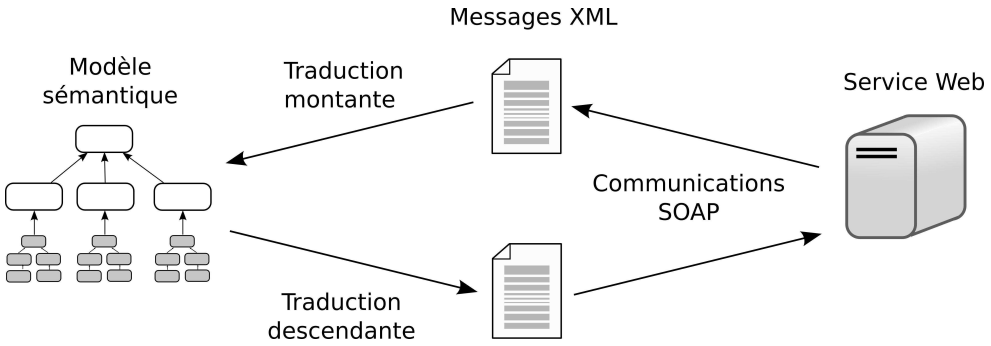

FIG. 6.9 – Schémas de traduction montants et descendants.

6.2.3.3 Pré-conditions

Afin de permettre aux concepteurs de services de spécifier des propriétés additionnelles pour les services, telles que le périphérique qui héberge le service et la localisation du périphérique, nous étendons le standard SAWSDL pour permettre d'annoter l'élément Service du contrat de service avec des pré-conditions spécifiées sous forme de règles logiques.

Les règles spécifiées comme pré-conditions sont exécutées lors de l'intégration du service dans l'infrastructure PERSEWS afin de modéliser les propriétés du nouveau service dans le modèle de contexte. Deux mécanismes permettent aux concepteurs de services de spécifier les propriétés d'un service:

- La primitive MakeOWLThing(?x) permet de créer un nouvel individu dans le modèle de contexte. Le nouvel individu est créé comme membre de la classe racine du modèle.
- L'individu *This* représente le service décrit par le contrat de service. Cet individu permet au concepteur de service de manipuler l'individu représentant le service dans le langage de règles.

Le tableau 6.1 présente une pré-condition spécifiant qu'un service est hébergé par une caméra qui possède un capteur de 3 Mégapixels. La règle logique qui constitue cette pré-condition est détaillée atome par atome.

Atome	Explication
MakeOWLThing(?periph)	Crée un individu de la classe racine.
→ Caméra(?periph)	Classifie l'individu comme membre de la classe caméra.
∧ Résolution(?periph,3000000)	La caméra possède une résolution de 3 Mégapixels.
∧ Expose(?periph, This)	La caméra expose le service décrit dans ce contrat

TAB. 6.1 – Règle de pré-condition pour un service hébergé par un périphérique de type caméra.

Lorsque le service s'enregistre auprès de l'annuaire, cette règle est envoyée au moteur d'inférence pour être exécutée. Ceci se traduit par la création d'un nouvel individu de la classe *Caméra* avec la propriété *Résolution* à 3 Mégapixels, et d'une nouvelle instance de la propriété *Expose* qui lie le service à l'individu caméra.

La figure 6.10 présente un extrait de contrat de service annoté avec la pré-condition détaillée dans le tableau 6.1.

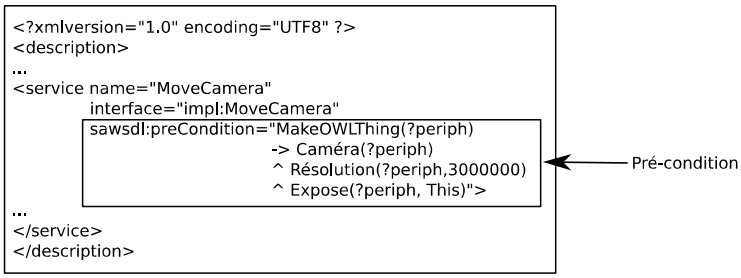

FIG. 6.10 – Exemple de pré-condition dans un contrat de service.

6.3 Synthèse

L'infrastructure PERSEWS permet de programmer des applications ubiquitaires à un haut niveau d'abstraction. L'approche langage mise en œuvre dans cette infrastructure permet au programmeur de spécifier la logique applicative en manipulant directement les concepts définis dans le modèle de contexte.

La caractère dynamique des environnements ubiquitaires est géré au niveau de l'infrastructure qui étend dynamiquement le langage de programmation pour fournir des abstractions de haut niveau permettant de manipuler les services lorsqu'ils apparaissent dans l'environnement. Le code permettant à l'infrastructure d'interagir avec ces services est également généré dynamiquement, aussi le programmeur n'a pas à se soucier des détails techniques d'invocation des services.

Le modèle d'architecture logicielle sur lequel est construite l'infrastructure PERSEWS permet de contrôler les services de l'environnement indépendamment des langages de programmation utilisés et des plates-formes d'exécution sur lesquels ils sont hébergés.

Si stupide que soit son existence, l'homme s'y rattache toujours.
Arthur Rimbaud, *Lettre du 10 juillet 1891*

7 Cas d'étude

Afin d'illustrer notre approche, nous présentons dans ce chapitre deux scénarios mettant en œuvre l'infrastructure PERSEWS dans des environnements ubiquitaires. Le premier scénario est un exemple d'application dans le domaine civil. L'infrastructure PERSEWS est déployée dans un musée pour assister les visiteurs et assurer la surveillance du bâtiment. Le deuxième scénario est un exemple d'application dans le domaine militaire. L'infrastructure PERSEWS est utilisée pour assister les forces en présence durant une mission de protection de zone.

7.1 Cas d'étude civil

Le cas d'étude présenté dans cette section se déroule dans un musée. Ce cas d'étude se divise en deux parties.

- Durant la journée, l'infrastructure a pour rôle d'assister les visiteurs du musée en leur proposant des vidéos explicatives concernant l'œuvre d'art à proximité de laquelle ils se situent.
- Durant la nuit, l'infrastructure gère la surveillance du bâtiment afin de prendre les mesures nécessaires en cas d'intrusion dans le bâtiment.

Un service d'agenda partagé envoie des événements pour signaler l'ouverture et la fermeture du musée et assurer la transition entre la phase d'assistance aux visiteurs et la phase de surveillance de bâtiment.

7.1.1 Assistance aux visiteurs

La figure 7.1 présente l'environnement ubiquitaire dans lequel se déroule la partie d'assistance aux visiteurs de ce cas d'étude. L'objectif est de proposer des vidéos explicatives aux vi-

siteurs concernant l'œuvre d'art à proximité de laquelle ils se situent. La détection des visiteurs est assurée par des lecteurs bluetooth qui détectent la présence des périphériques à proximité des oeuvres d'art. Pour afficher les vidéos, l'infrastructure exploite les périphériques de l'environnement qui fournissent une fonctionnalité de lecture de flux multimédia. Un grand nombre de périphériques grand public, tels que des téléphones mobiles ou des assistants personnels, sont compatibles avec le protocole bluetooth et sont équipés d'un lecteur multimédia. Ainsi, les visiteurs peuvent visualiser les vidéos explicatives directement sur leurs périphériques personnels.

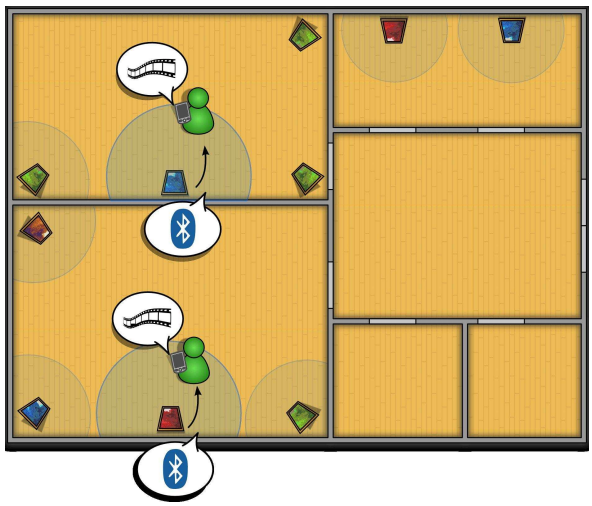

FIG. 7.1 – Exploitation de l'infrastructure PERSEWS pour l'assistance des visiteurs dans un musée.

7.1.1.1 Services de l'environnement

Les lecteurs bluetooth sont intégrés dans l'infrastructure comme des services fonctionnant sur le mode publication/souscription. Lorsqu'un périphérique est détecté par un lecteur bluetooth, le service publie un message de détection qui est traité par le gestionnaire d'événements. Ceci se traduit par l'ajout de l'individu représentant le périphérique détecté à la classe *Détecté*, et par la mise à jour de la propriété *EstLocalisé* qui lie le périphérique et l'oeuvre d'art à proximité de laquelle le périphérique a été détecté.

L'agenda partagé est intégré dans l'infrastructure comme un service fonctionnant sur le mode publication/souscription. Les horaires du musée sont enregistrés dans l'agenda qui publie des événements lors de l'ouverture et de la fermeture du musée. Le gestionnaire d'événements traite ces événements, et ajoute l'individu *Musée* à la classe *Ouvert* ou *Fermé* dans le modèle de contexte. Ces deux classes sont définies comme disjointes aussi l'individu *Musée* ne peut appartenir simultanément aux deux classes.

Lorsqu'un visiteur souhaite profiter des vidéos contextuelles proposées par le musée, il publie le contrat de son service de lecteur multimédia dans l'annuaire de l'infrastructure PERSEWS. Un individu de la classe *Lecteur* est alors ajouté dans la base de connaissances. Cette classe de service possède une fonctionnalité de lecture qui est accessible au programmeur par l'intermédiaire

de la primitive `Lecteur:lecture(?service, ?desc)`. Cette primitive démarre la lecture d'une description vidéo à partir du service de lecture.

7.1.1.2 Classes d'états

Lorsqu'un périphérique est en cours de lecture d'une vidéo explicative, l'individu représentant le périphérique dans le modèle de contexte est ajouté à la classe *EnLecture*. Cette classe est définie comme disjointe de la classe *Détecté*, aussi un même individu ne peut appartenir simultanément à ces deux classes. La figure 7.2 présente les classes successives auxquelles peut appartenir un périphérique lorsqu'il se déplace dans le musée. Lors de la première détection, le périphérique est ajouté à la classe *Détecté*. Ceci déclenche la lecture de la vidéo sur le périphérique qui passe alors dans la classe *EnLecture*. Une détection ultérieure le replace dans la classe *Détecté*.

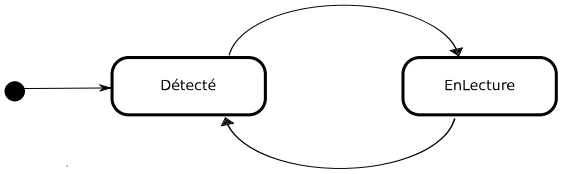

FIG. 7.2 – États d'un périphérique durant l'assistance aux visiteurs.

7.1.1.3 Tâche

La tâche de lecture des vidéos de description des oeuvres d'art consiste à démarrer la lecture de la vidéo de description lorsqu'un périphérique est détecté à proximité d'une oeuvre d'art. Cette tâche est implémentée par la règle logique suivante :

Ouvert(Musée) $\wedge$ Détecté(?periph) $\wedge$ EstLocalisé(?périph, ?art) $\wedge$ Description(?art, ?desc) $\wedge$ Expose(?périph, ?service) $\wedge$ Lecteur(?service) $\rightarrow$ Lecteur:lecture(?service, ?desc) $\wedge$ EnLecture(?periph)

Le tableau 7.1 présente les atomes qui composent cette tâche et explique le rôle de ces atomes.

Cette tâche permet d'assurer la lecture contextuelle des descriptions vidéos pour tous les visiteurs qui ont publié un contrat de service de lecteur multimédia dans l'annuaire de l'infrastructure PERSEWS. L'antécédent de la règle est évalué comme vrai si :

- l'individu *Musée* appartient à la classe *Ouvert*, ce qui signifie que le musée est ouvert;
- un individu représentant un périphérique appartient à la classe *détecté*, ce qui signifie qu'un périphérique a été détecté par un lecteur bluetooth ;
- ce périphérique est localisé à proximité d'une œuvre d'art. L'œuvre d'art concernée est accessible via la propriété *EstLocalisé* du périphérique détecté, qui est mise à jour par le gestionnaire d'événements lors de la détection.

Atome	Explication
Ouvert(Musée)	Le musée est ouvert.
∧ Détecté(?periph)	Un périphérique est détecté à proximité d'une oeuvre d'art.
∧ EstLocalisé(?périph, ?art)	
∧ Description(?art, ?desc)	L'oeuvre d'art possède une description vidéo.
∧ Expose(?périph, ?service)	Le périphérique expose un service de type lecteur multimédia.
∧ Lecteur(?service)	
→ Lecteur:lecture(?service, ?desc)	Lire la description vidéo sur le lecteur multimédia.
∧ EnLecture(?periph)	Le périphérique est en cours de lecture

TAB. 7.1 – Tâche d'assistance des visiteurs.

- L'individu qui représente l'œuvre d'art possède une propriété *Description* qui donne l'adresse permettant d'accéder au flux de la description vidéo ;
- l'individu qui représente le périphérique est lié à un individu de la classe *Lecteur* par la propriété *Expose*, ce qui signifie que le périphérique détecté expose un service de type lecteur multimédia.

Si l'antécédent est évalué comme vrai, alors le conséquent est tenu pour vrai. Dans ce cas :

- la primitive Lecteur:lecture(?service, ?desc) est exécutée par le moteur d'inférence. Ceci se traduit par une invocation du service de lecture multimédia du périphérique détecté afin de jouer la vidéo explicative sur le périphérique ;
- l'individu qui représente le périphérique est ajouté à la classe *EnLecture*, ce qui signifie que ce périphérique est en cours de lecture d'un flux multimédia.

7.1.2 Surveillance du bâtiment

La figure 7.3 présente l'environnement ubiquitaire dans lequel se déroule la partie de surveillance de bâtiment de ce cas d'étude. En cas de détection de mouvement dans une des pièces du musée, l'objectif est de déclencher les alarmes du musée, de capturer les images de l'effraction et d'envoyer ces images au gardien du musée qui prend alors les mesures nécessaires en fonction de la situation observée. Dans ce but sont installés dans chaque pièce des détecteurs de mouvements, des caméras vidéos contrôlables à distance et des alarmes. De plus, le gardien du musée est doté d'un assistant personnel disposant d'un lecteur multimédia.

7.1.2.1 Services de l'environnement

Les détecteurs de mouvements sont intégrés dans l'infrastructure comme des services fonctionnant sur le mode publication/souscription. Lorsqu'un mouvement est détecté, le service publie un message de détection qui est traité par le gestionnaire d'événements. Ceci se traduit par l'ajout de l'individu représentant la zone où le mouvement a été détecté à la classe *Mouvement*.

Les alarmes sont intégrées dans l'infrastructure comme des service de la classe *Alarme* possédant une fonctionnalité d'activation. Cette fonctionnalité est accessible au programmeur par l'intermédiaire de la primitive `Alarme:activer(?alarme)`.

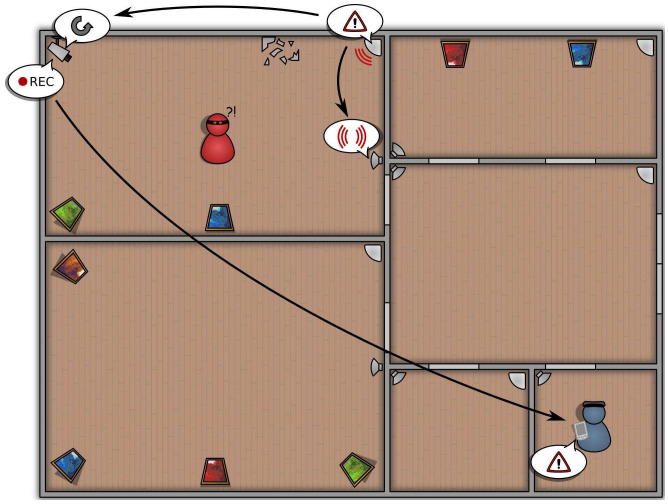

FIG. 7.3 – Exploitation de l'infrastructure PERSEWS pour la surveillance de bâtiment dans un musée.

Les caméras sont intégrées dans l'infrastructure comme des services de la classe *Mouvement-Caméra* possédant une fonctionnalité de contrôle à distance. Cette fonctionnalité est accessible aux programmeurs par l'intermédiaire de la primitive `MouvementCaméra:déplacement-(?mouvcam, ?zone)`. Cette primitive contrôle le déplacement de la caméra pour filmer une des zones déclarées dans le modèle de contexte.

Les zones accessibles par chaque caméra sont déclarées par la propriété *Portée* qui lie une caméra à une ou plusieurs zones. De plus chaque caméra possède une adresse permettant d'accéder à son flux vidéo, spécifiée dans le modèle de contexte par l'intermédiaire de la propriété *FluxMultimédia*.

Le lecteur multimédia installé sur l'assistant personnel du gardien est intégré dans l'infrastructure comme un service de la classe *Lecteur* possédant une fonctionnalité de lecture qui est accessible au programmeur par l'intermédiaire de la primitive `Lecteur:lecture-(?service, ?flux)`. Cette primitive démarre la lecture d'un flux multimédia à partir du service de lecture.

7.1.2.2 Classes d'états

Lorsqu'une zone est filmée par une caméra, l'individu représentant cette zone dans le modèle de contexte est ajouté à la classe *Surveillance*. Cette classe est définie comme disjointe de la classe *Mouvement*, aussi un même individu ne peut appartenir simultanément à ces deux classes. La figure 7.4 présente les classes successives auxquelles peut appartenir une zone lorsque le musée est victime d'une intrusion. Lorsqu'un mouvement est détecté dans une zone, la zone est ajoutée à la classe *Mouvement*. Ceci provoque l'activation des alarmes, la réorientation des caméras et l'envoi du flux sur le lecteur du gardien. La zone passe alors dans la classe *Surveillance*.

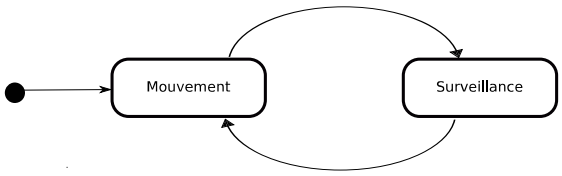

FIG. 7.4 – États d'une zone durant la surveillance de bâtiment.

7.1.2.3 Tâche

La tâche de surveillance du bâtiment consiste à déclencher les alarmes lorsqu'un mouvement est détecté, à réorienter les caméras pour saisir la scène et à envoyer le flux des caméras sur le lecteur du gardien du musée. Cette tâche est implémentée par la règle logique suivante :

Fermé(Musée) ∧ Mouvement(?zone) ∧ Caméra(?c) ∧ Portée(?c, ?zone) ∧ Expose(?c, ?mvmt) ∧ MouvementCaméra(?mvmt) ∧ FluxMultimédia(?c, ?flux) ∧ Alarme(?alarme) → Alarme: activer(?alarme) ∧ MouvementCaméra:déplacement(?mvmt, ?zone) ∧ Lecteur:lecture (LecteurSurveillant, ?flux) ∧ Surveillance(?zone)

Le tableau 7.2 présente les atomes qui composent cette tâche et explique le rôle de ces atomes.

Atome	Explication
Fermé(Musée)	Le musée est fermé.
∧ Mouvement(?zone)	Un mouvement est détecté sur une zone.
∧ Caméra(?c)	Une caméra peut accéder à cette zone.
∧ Portée(?c, ?zone)	
∧ Expose(?c, ?mvmt)	La caméra expose un service de type mouvement de caméra.
∧ MouvementCaméra(?mvmt)	
∧ FluxMultimédia(?c, ?flux)	La caméra fournit un flux multimédia.
∧ Alarme(?alarme)	Sélection des service de déclenchement d'alarme du musée.
→ Alarme:activer(?alarme)	Déclencher toutes les alarmes du musée.
∧ MouvementCaméra:déplacement(?mvmt, ?zone)	Réorienter la caméra vers la zone de détection.
∧ Lecteur:lecture(LecteurSurveillant, ?flux)	Envoyer le flux multimédia sur le lecteur multimédia du gardien du musée.
∧ Surveillance(?zone)	La zone est sous surveillance.

TAB. 7.2 – Tâche de surveillance du bâtiment.

Cette tâche permet d'assurer la surveillance du bâtiment lorsque le musée est fermé. L'antécédent de la règle est évalué comme vrai si :

- l'individu *Musée* appartient à la classe *Fermé*, ce qui signifie que le musée est fermé ;
- un individu représentant une zone appartient à la classe *Mouvement*, ce qui signifie qu'un détecteur de mouvement a détecté du mouvement dans cette zone ;
- un individu qui représente un périphérique de type caméra est lié à l'individu qui représente

la zone par la propriété *Portée*, ce qui signifie qu'une caméra peut filmer cette zone ;
- l'individu qui représente la caméra est lié à un individu de la classe *MouvementCaméra* par la propriété *Expose*, ce qui signifie que la caméra expose un service de type mouvement de caméra qui permet de contrôler la position de la caméra à distance ;
- l'individu qui représente la caméra possède une propriété *FluxMultimédia* qui donne l'adresse permettant d'accéder au flux filmé par la caméra ;
- au moins un individu appartient à la classe *Alarme*, ce qui signifie qu'au moins un service de type alarme est disponible dans l'environnement.

Si l'antécédent est évalué comme vrai, alors le conséquent est tenu pour vrai. Dans ce cas :
- la primitive `Alarme:activer(?alarme)` est exécutée par le moteur d'inférence. Ceci se traduit par une invocation de tous les services de type alarme de l'environnement afin de déclencher l'activation des alarmes.
- la primitive `MouvementCaméra:déplacement(?mvmt,?zone)` est exécutée par le moteur d'inférence. Ceci se traduit par une invocation de tous les services de type mouvement de caméra afin de régler l'orientation des caméras qui peuvent filmer la zone.
- la primitive `Lecteur:lecture(LecteurSurveillant,?flux)` est exécutée par le moteur d'inférence. Ceci se traduit par une invocation du service de lecture multimédia de l'assistant personnel du gardien du musée afin d'afficher le flux des caméras qui filment la zone.
- l'individu qui représente la zone est ajouté à la classe *Surveillance*, ce qui signifie que cette zone est sous surveillance.

7.1.3 Évaluation

Le cas d'étude présenté dans cette section illustre la mise en œuvre de l'infrastructure PERSEWS dans un cadre civil pour fournir une assistance aux visiteurs dans un musée et assurer la surveillance du musée. Dans ce domaine, la programmation par règles logiques permet de spécifier simplement et avec concision la logique de l'application ubiquitaire.

7.2 Cas d'étude militaire

Nous présentons dans cette section un scénario militaire de protection de zone mis en oeuvre grâce à l'infrastructure PERSEWS. La figure 7.5 illustre le terrain des opérations. La mission globale est la suivante:
- Interdiction permanente de franchissement de frontière à des véhicules autres que les résidents civils;
- Surveillance permanente de pénétration de la zone A le long de la frontière;
- Déclenchement d'un tir si les véhicules entrent dans la zone B après la frontière.

Cette mission peut être décomposée en cinq phases:

Phase de surveillance. Deux drones de type Moyenne Altitude Longue Endurance (MALE) équipés de capteurs de mouvements surveillent en permanence la zone A afin de détecter toute activité anormale.

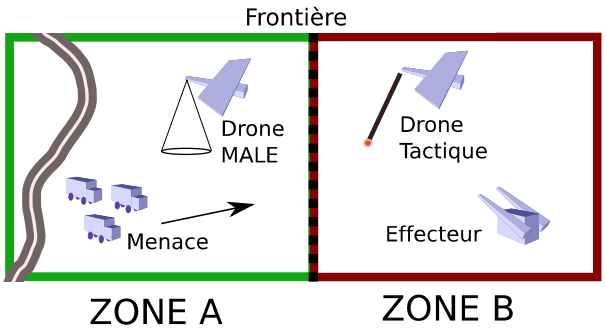

FIG. 7.5 – Terrain des opérations pour une mission de protection de zone.

Phase d'identification. Lorsqu'une activité est détectée, les images issues du capteur optique du drone de type MALE qui a effectué la détection sont transmises au centre de commandement pour identification de la menace.

Phase de poursuite. Si la menace est effectivement confirmée, le drone de type MALE passe en mode poursuite de cible pour suivre la cible de manière précise. A l'aide des informations fournies par le drone de type MALE, un drone tactique équipé d'un dispositif de visée laser assure l'illumination laser de la cible pour permettre sa destruction.

Phase de suppression. Si le drone de type MALE entre dans la zone B, un missile à guidage laser (effecteur) est alors tiré pour détruire la menace.

Phase de désengagement. Le centre de commandement qui dispose des images de la zone peut émettre une alerte de désengagement qui annule la mission en cas de présence de civils sur la zone concernée.

Phase d'évaluation. Finalement, le centre de commandement valide la destruction de la cible à partir des images transmises par le drone de type MALE.

7.2.1 Services de l'environnement

Les forces présentes sur le terrain des opérations (drones de type MALE, drones tactiques, effecteurs et centre de commandement) sont intégrées dans l'infrastructure sous la forme de services qui exposent des fonctionnalités. Les drones de type MALE, les drones tactiques et les effecteurs possèdent chacun une fonctionnalité d'annulation qui permet de les désengager d'une mission. Cette fonctionnalité est accessible au programmeur par l'intermédiaire des primitives:

- `MALE:annulation(?drone)`
- `DroneTactique:annulation(?dronetact)`
- `Effecteur:annulation(?effecteur)`

7.2.1.1 Drones de type MALE

Chaque drone de type MALE présent sur le terrain des opérations est intégré dans l'infrastructure comme deux services distincts:

- Un services fonctionnant sur le mode publication/souscription. Lorsqu'un mouvement est détecté, le service publie un message de détection. Lors de la réception d'un message de ce type, le gestionnaire d'événements ajoute le drone comme individu de la classe *Détection* dans la base de connaissance.
- Un service fournissant une fonctionnalité d'identification de menace et une fonctionnalité de poursuite de cible. La fonctionnalité d'identification de menace est accessible au programmeur par l'intermédiaire de la primitive `MALE:identification(?drone)`. Cette primitive active le capteur optique du drone dont les images sont transmises au centre de commandement. La fonctionnalité de poursuite de cible est accessible au programmeur par l'intermédiaire de la primitive `MALE:poursuite(?drone)`. Cette primitive active le mode poursuite de cible du drone qui suit alors une cible de manière précise.

7.2.1.2 Drones tactiques

Les drones tactiques sont intégrés dans l'infrastructure comme des service de la classe *DroneTactique* possédant une fonctionnalité de visée laser. Cette fonctionnalité est accessible aux programmeurs par l'intermédiaire de la primitive `DroneTactique:viséeLaser(?dronetact,?drone)`.

Cette primitive active la visée laser du drone tactique qui illumine la position identifiée par un drone de type MALE en mode poursuite. Le drone de type MALE sur lequel le drone tactique synchronise sa visée laser est identifié par le second argument ?drone.

7.2.1.3 Effecteurs

Les missiles à guidage laser sont intégrés dans l'infrastructure comme des services de la classe *Effecteur* possédant une fonctionnalité de tir. Cette fonctionnalité est accessible au programmeur par l'intermédiaire de la primitive `Effecteur:tir(?effecteur, ?dronetact)`.

Cette primitive commande l'initialisation d'une séquence de tir sur une cible visée par un drone tactique. Le drone tactique qui illumine la cible est identifié par le second argument ?dronetact.

7.2.1.4 Centre de commandement

Le centre de commandement est intégré dans l'infrastructure comme un service fonctionnant sur le mode publication/souscription. Ce service émet quatre types de messages:

Message de confirmation Lorsque le centre de commandement confirme la menace sur les images envoyées par un drone de type MALE en phase d'identification, le service émet un message de confirmation indiquant le drone concerné. Ce message est traité par le gestionnaire d'événement qui ajoute l'individu représentant le drone comme membre de la classe *MenaceConfirmée*.

Message de franchissement de zone Lorsqu'un drone franchit les limites de la zone B, le service émet un message de franchissement de zone indiquant le drone concerné. Ce message est traité par le gestionnaire d'événement qui met à jour la propriété *EstLocalisé* du drone.

Message de désengagement Le centre de commandement peut également émettre un message de désengagement en cas de présence de civils sur une zone. Ce message contient l'identifiant du drone qui fournit les images de la zone. Lors de la réception d'un tel message, le gestionnaire d'événements ajoute l'individu représentant le drone comme membre de la classe *Annulation*.

Message de destruction En fin de mission, le centre de commandement émet un message de destruction qui confirme ou infirme la destruction des cibles à partir des images transmises par le drone de type MALE.

7.2.2 Classes d'états

Outre les différents services présentés, des classes sont définies dans le modèle de contexte pour caractériser l'état des forces au cours des différentes phases de la mission. Ces classes sont définies comme disjointes, de ce fait un même individu ne peut appartenir simultanément à ces différentes classes. La figure 7.6 présente les classes qui caractérisent l'état des forces durant les différentes phases de la mission.

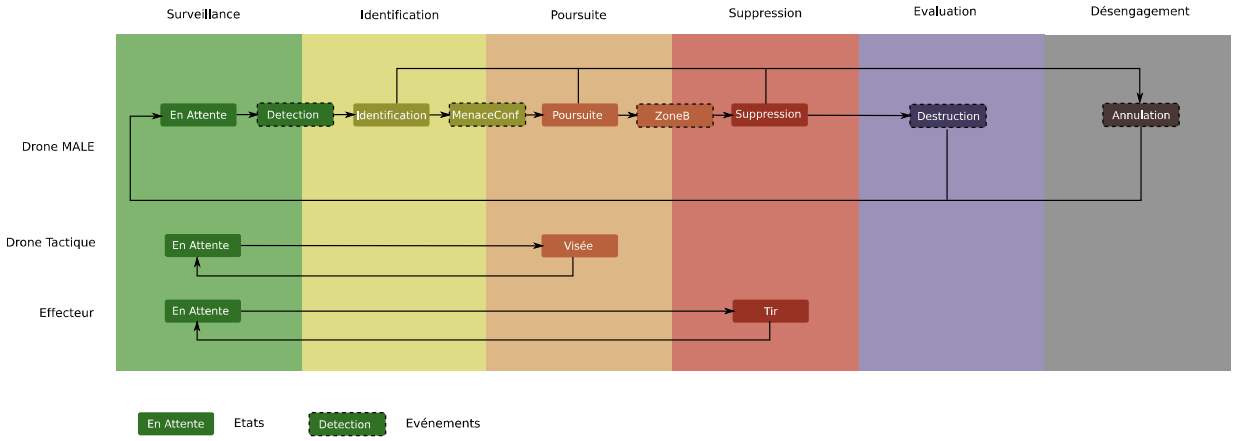

FIG. 7.6 – État des forces au cours d'une mission de protection de zone.

Phase de surveillance. Durant la phase de surveillance, toutes les forces sont dans l'état *EnAttente*. Un événement *Détection* déclenche la phase d'identification.

Phase d'identification. Durant la phase d'identification, le drone MALE passe dans l'état *Identification*. Un événement *MenaceConfirmée* déclenche la phase de poursuite.

Phase de poursuite. Durant la phase de poursuite, le drone MALE passe dans l'état *Poursuite* et le drone tactique passe dans l'état *Visée*. Un événement de localisation en *ZoneB* déclenche la phase de suppression.

Phase de suppression. Durant la phase de suppression, le drone MALE passe dans l'état *Suppression* et l'effecteur passe dans l'état *Tir*.

Phase d'évaluation Un événement *Destruction* entraîne un retour à la phase de surveillance.

Désengagement Durant la mission, un événement *Annulation* entraîne un retour à la phase de surveillance.

La propriété *Synchronisé* permet de lier dans la base de connaissances les couples d'individus dont les fonctionnalités sont utilisées de manière conjointe. Cette propriété lie un drone de type MALE et un drone tactique dans la phase de poursuite, puis un drone tactique et un effecteur durant la phase de suppression.

7.2.3 Tâches

La mission de protection de zone est programmée à l'aide de cinq tâches distinctes dans l'infrastructure PERSEWS. Ces tâches correspondent aux phases de la mission.

La tâche d'identification a pour but de fournir une identification visuelle des cibles lors d'une détection de mouvement dans la zone protégée.

La tâche de poursuite a pour but de synchroniser un dispositif de visée laser sur les cibles si la menace a été confirmée par le centre de commandement.

La tâche de suppression a pour but de déclencher une séquence de tir sur la cible visée.

La tâche d'évaluation a pour but de permettre au centre de commandement d'évaluer le résultat de la mission à partir des images fournies par le drone de type MALE.

Les tâches de désengagement ont pour but de permettre au centre de commandement d'annuler la mission en cas de présence de civils.

7.2.3.1 Tâche d'identification

Le tableau 7.3 présente la tâche d'identification de la mission de protection de zone. Lorsqu'un mouvement est détecté par un drone de type MALE, ce drone active ses capteurs optiques pour fournir des images au centre de commandement. Le drone passe en phase d'identification.

Atome	Explication
Détection(?drone)	Un mouvement est détecté par un drone de type MALE.
$\rightarrow$ MALE:identification(?drone)	Le drone MALE identifie la menace à l'aide de ses capteurs optiques.
$\wedge$ Identification(?drone)	Le drone MALE passe en phase d'identification.

TAB. 7.3 – Tâche d'identification dans une mission de protection de zone.

7.2.3.2 Tâche de poursuite

Le tableau 7.4 présente la tâche de poursuite de la mission de protection de zone. Si une phase d'identification est engagée et que le centre de commandement a confirmé la menace sur les images envoyées par le drone MALE, alors le drone MALE passe en mode de poursuite de

cible afin de suivre la cible de manière précise. Le drone tactique synchronise sa visée laser ce drone. Le drone de type MALE passe en phase de poursuite, le drone tactique passe en phase de visée et une synchronisation impliquant le drone de type MALE et le drone tactique est engagée.

Atome	Explication
MenaceConfirmée(?drone)	Le centre de commandement a confirmé la menace.
$\wedge$ DroneTactique(?dronetact)	Un drone tactique est en attente de mission.
$\wedge$ EnAttente(?dronetact)	
$\rightarrow$ MALE:poursuite(?drone)	Le drone MALE passe en mode poursuite de cible.
$\wedge$ DroneTactique:viséeLaser(?dronetact, ?drone)	Le drone tactique synchronise sa visée laser sur la cible poursuivie.
$\wedge$ Poursuite(?drone)	Le drone MALE passe en phase de poursuite.
$\wedge$ Visée(?dronetactique)	Le drone tactique passe en phase de visée.
$\wedge$ Synchronisé(?drone, ?dronetact)	La synchronisation entre le drone MALE et le drone tactique est engagée.

TAB. 7.4 – Tâche de poursuite dans une mission de protection de zone.

7.2.3.3 Tâche de suppression

Le tableau 7.5 présente la tâche de suppression de la mission de protection de zone. Si une phase de poursuite est engagée et que le drone passe dans le zone B, alors un effecteur en attente de mission est sélectionné. La séquence de tir de l'effecteur est initialisée sur la cible visée par le drone tactique. L'effecteur passe en phase de tir et une synchronisation impliquant le drone tactique et l'effecteur est engagée.

7.2.3.4 Tâche de désengagement

Le tableau 7.6 présente la tâche de désengagement de drone en phase d'identification. Si une phase d'identification est engagée et que le centre de commandement annule la mission à partir des images envoyées par le drone MALE (présence de civils), alors le drone de type MALE repasse en attente de mission.

Le tableau 7.7 présente la tâche de désengagement en phase de suppression de la mission de protection de zone. Si une phase de suppression est engagée et que le centre de commandement émet un message de désengagement, alors la séquence de tir de l'effecteur impliqué dans la phase de suppression est annulée. Le drone de type MALE et le drone tactique repassent en attente de mission.

Atome	Explication
Poursuite(?drone)	Un drone MALE est en phase de poursuite.
∧ EstLocalisé(?drone, ZONEB)	Le drone MALE est situé dans la zone B.
∧ Synchronisé(?dronetact, ?drone)	Une synchronisation entre le drone MALE et le drone tactique est engagée.
∧ Effecteur(?effecteur)	Un effecteur est en attente de mission.
∧ Attente(?effecteur)	
→ Effecteur:tir(?effecteur, ?dronetact)	Initialisation de la séquence de tir de l'effecteur sur la cible visée par le drone tactique
∧ Tir(?effecteur)	L'effecteur passe en phase de tir.
∧ Synchronisé(?dronetact, ?effecteur)	La synchronisation entre le drone tactique et l'effecteur est engagée.
∧ Suppression(?drone)	Le drone MALE passe en phase de suppression.

TAB. 7.5 – Tâche de suppression dans une mission de protection de zone.

Atome	Explication
Annulation(?drone)	Le centre de commandement émet une alerte de désengagement à partir des images d'un drone MALE.
→ MALE:annulation(?drone)	Le drone MALE passe en attente de mission.
∧ EnAttente(?drone)	

TAB. 7.6 – Tâche de désengagement en phase d'identification dans une mission de protection de zone.

Atome	Explication
Annulation(?drone)	Le centre de commandement émet une alerte de désengagement à partir des images d'un drone MALE.
∧ DroneTactique(?dronetact)	Le drone MALE est synchronisé avec un drone tactique.
∧ Synchronisé(?drone, ?dronetact)	
∧ Effecteur(?effecteur)	Le drone tactique est synchronisé avec un effecteur.
∧ Synchronisé(?dronetact, ?effecteur)	
→ Effecteur:annulation(?effecteur)	L'effecteur passe en attente de mission.
∧ EnAttente(?effecteur)	
∧ DroneTactique:annulation(?dronetact)	Le drone tactique passe en attente de mission.
∧ EnAttente(?dronetact)	

TAB. 7.7 – Tâche de désengagement en phase de suppression dans une mission de protection de zone.

7.2.3.5 Tâche d'évaluation

Le tableau 7.8 présente la tâche d'évaluation de la mission de protection de zone. Si le centre de commandement émet un message de destruction à partir des images envoyées par le drone de

type MALE, alors les forces impliquées dans la mission repassent en attente de mission.

Atome	Explication
Destruction(?drone)	Le centre de commandement émet un message de destruction à partir des images d'un drone MALE.
∧ Synchronisé(?drone, ?dronetact)	Le drone MALE est synchronisé avec un drone tactique.
∧ Synchronisé(?dronetact, ?effecteur)	Le drone tactique est synchronisé avec un effecteur.
→ MALE:annulation(?drone)	Le drone MALE passe en attente de mission.
∧ EnAttente(?drone)	
∧ Effecteur:annulation(?effecteur)	L'effecteur passe en attente de mission.
∧ EnAttente(?effecteur)	
∧ DroneTactique:annulation(?dronetact)	Le drone tactique passe en attente de mission.
∧ EnAttente(?dronetact)	

TAB. 7.8 – Tâche d'évaluation dans une mission de protection de zone.

7.2.4 Évaluation

Le cas d'étude présenté dans cette section illustre la mise en œuvre de l'infrastructure PERSEWS dans un cadre militaire pour assurer une mission de protection de zone. Ce cas d'étude montre les limites de notre approche. En effet, la programmation par règles logiques ne permet pas de spécifier explicitement un flot de contrôle. Le programmeur doit donc définir un grand nombre de classes pour modéliser l'état des forces au cours de la mission, ce qui réduit la simplicité et la concision des règles.

7.3 Synthèse

L'infrastructure PERSEWS permet de programmer des applications ubiquitaires dans différents domaines. Le premier cas d'étude présenté dans ce chapitre illustre la mise en œuvre de l'infrastructure PERSEWS dans un cadre civil pour fournir une assistance aux visiteurs dans un musée et assurer la surveillance du musée. Dans ce domaine, la programmation par règles logiques permet de spécifier simplement et avec concision la logique de l'application ubiquitaire.

Le deuxième cas d'étude illustre la mise en œuvre de l'infrastructure PERSEWS dans un cadre militaire pour assurer une mission de protection de zone. Ce cas d'étude met en lumière les limites de l'approche. En effet, la programmation par règles logiques ne permet pas de spécifier explicitement un flot de contrôle. De ce fait, la réalisation de ce cas d'étude nécessite de créer un nombre important de classes pour modéliser l'état des forces au cours des différentes phases de la mission. La réalisation de scénarios complexes avec l'infrastructure PERSEWS nécessite donc une expertise avancée en programmation par règles logiques.

Troisième partie

Résultats expérimentaux

Sans technique, le talent n'est rien qu'une sale manie.

Georges Brassens, *Le mauvais sujet repenti*

8 Prototype de l'infrastructure PERSEWS

Afin de valider notre approche et d'évaluer les performances de notre infrastructure, nous avons développé un prototype de l'infrastructure PERSEWS. Ce prototype constitue une preuve des concepts présentés dans ce document et sert de base de test pour les mesures de performances. Dans la suite de ce chapitre, nous présentons les choix technologiques qui sous-tendent l'implémentation du prototype, puis nous présentons une évaluation des performances de l'infrastructure PERSEWS basée sur notre prototype.

8.1 Implémentation du prototype

Les technologies utilisées pour l'implémentation du prototype de l'infrastructure PERSEWS sont issues du domaine du logiciel libre. Ces technologies sont largement diffusées dans le monde universitaire et industriel et fournissent donc une base solide et éprouvée pour notre prototype. L'intégralité du développement a été effectuée en langage JAVA, ce qui permet de porter notre prototype sur les systèmes d'exploitation les plus répandus. Le développement de l'annuaire de services repose sur l'implémentation JAVA jUDDI [1] de la spécification UDDI. Le moteur d'inférence est basé sur le moteur Jess [26]. Les Services Web qui composent l'infrastructure PERSEWS et les différents services de l'environnement ont été développés à l'aide de la pile SOAP Axis2 [2] et ont été déployés sur le serveur d'application Tomcat [3] de la fondation Apache. Les ontologies ont été développées grâce à l'éditeur d'ontologies Protégé [51]. Finalement, l'éditeur de règles est basé sur le module complémentaire SWRLJessTab [4] pour Protégé.

[1] Apache jUDDI, http://ws.apache.org/juddi/
[2] Apache Axis2, http://ws.apache.org/axis2/
[3] Apache Tomcat, http://tomcat.apache.org/
[4] SWRLJessTab, http://protege.cim3.net/cgi-bin/wiki.pl?SWRLJessTab

8.1.1 Annuaire

Comme présenté dans la section 6.1.1, l'annuaire est le composant central de la découverte de services dans l'infrastructure PERSEWS. Les services de l'environnement peuvent publier leurs contrats de service annotés suivant la spécification SAWSDL.

A partir des informations sémantiques contenues dans le document SAWSDL, l'annuaire enrichit la base de connaissances avec une instance représentant le service. Deux cas de figure peuvent se présenter.

- La classe sémantique de l'interface du service existe dans le modèle de contexte. Dans ce cas, l'annuaire ajoute une instance qui représente le nouveau service à cette classe de service.
- La classe sémantique de l'interface du service n'existe pas dans le modèle de contexte. Dans ce cas, l'annuaire examine les opérations définies dans l'interface. Si l'interface constitue une spécialisation d'une interface existante (la nouvelle interface possède au moins les même opérations qu'une interface existante), alors ce service appartient à une sous-classe d'une classe de service existante (cf. figure 8.1 partie (b)). Si l'interface ne constitue pas une spécialisation d'une interface existante, alors ce service appartient à une sous-classe directe de la classe *Service* de l'ontologie supérieure (cf. figure 8.1 partie (c)). Finalement, l'annuaire crée la classe dans l'ontologie et ajoute une instance qui représente le nouveau service.

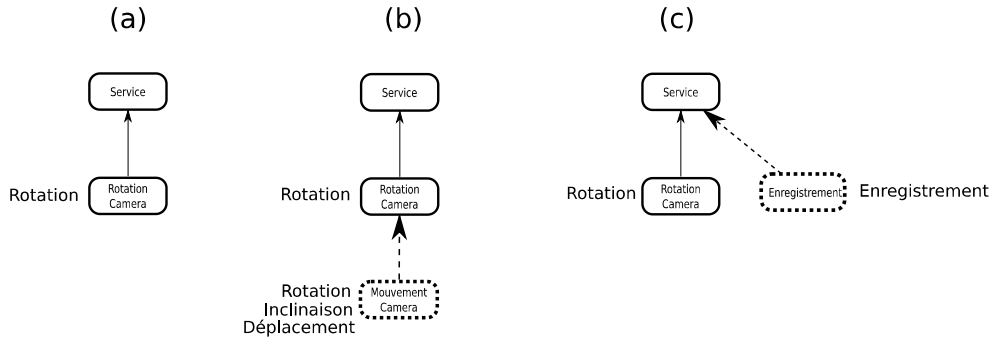

FIG. 8.1 – Ajout d'une classe de service lors de la publication d'un contrat de service.

8.1.2 Générateur

Le générateur produit le code qui permet au moteur d'inférence d'invoquer les service de l'environnement. Comme présenté dans le figure 8.2, le code généré constitue une couche d'interface entre le moteur d'inférence et le moteur SOAP. La couche d'interface intercepte l'exécution des primitives représentant les fonctionnalités des services par le moteur d'inférence. Les arguments de la primitive sont transformés du format OWL au format XML grâce au schéma de traduction descendant fourni dans le document SAWSDL. La couche d'interface extrait l'adresse des services passés en arguments afin de permettre la création des points d'accès aux services. Les arguments sont alors envoyés aux services via le moteur SOAP. Si les services émettent un

message en retour, la couche d'interface transforme les données du message du format XML au format OWL grâce au modèle de traduction montant et enregistre ces informations dans la base de connaissances du moteur d'inférence.

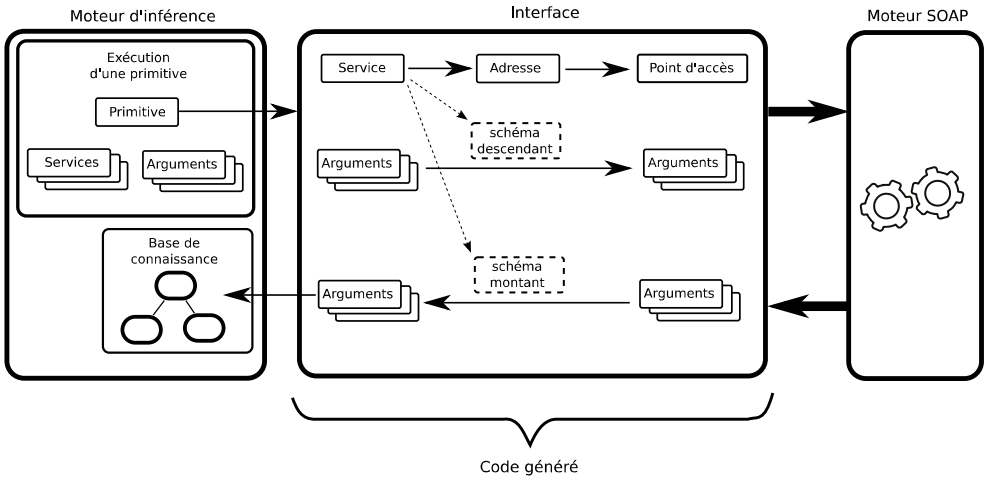

FIG. 8.2 – Code généré pour une classe de services.

8.1.3 Moteur d'inférence

Le modèle de contexte est stocké dans une base de connaissances Protégé-OWL et le processus d'inférence est assuré par le moteur Jess. Le moteur d'inférence Jess est constitué d'une base de règles, d'une base de faits et d'un moteur d'exécution. Le moteur d'exécution applique les règles contenues dans la base de règles aux faits contenus dans la base de faits. L'API Protégé fournit un pont (pont SWRL Jess) qui permet de connecter une base de connaissances Protégé-OWL contenant un modèle OWL avec un moteur d'inférence Jess. La figure 8.3 présente le fonctionnement du pont SWRL Jess. Ce pont fournit les mécanisme permettant (1) d'importer des règles SWRL dans une base de règles Jess; (2) d'importer des classes, individus, propriétés et restrictions OWL d'un modèle OWL dans une base de faits Jess; (3) d'écrire les faits inférés par le moteur d'inférence dans le modèle OWL.

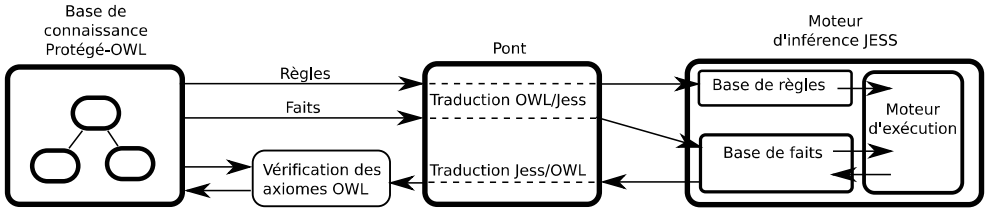

FIG. 8.3 – Fonctionnement du pont SWRL Jess.

Une limitation du pont vers le moteur Jess est qu'il ne prend pas en compte tous les axiomes spécifiés dans la base de connaissances OWL. En particulier, le caractère disjoint des classes

OWL et les arités des propriétés OWL ne sont pas pris en compte. Ceci peut induire l'ajout de connaissances conflictuelles dans la base de connaissances OWL (par exemple un individu appartenant à deux classes disjointes ou un individu possédant plusieurs instances d'une propriété d'arité 1). Afin de résoudre ce problème, nous avons développé une extension au pont de l'API Protégé qui vérifie la consistance des classes et propriétés OWL après avoir ajouté les faits inférés par le moteur Jess. Dans le cas où des inconsistances sont observées, les modifications suivantes sont apportées à la base de connaissances.

Classes disjointes. Soient *A* et *B* deux classes définies dans l'ontologie OWL comme disjointes et I_a un individu appartenant à la classe A. Si le moteur d'inférence classifie l'individu I_a comme appartenant à la classe B, alors l'individu I_a est retiré de la classe A.

Propriétés d'arité 1. Soit P une propriété d'arité 1 et $\{I_1, I_2\}$ une instance de cette propriété. Si le moteur d'inférence classifie le couple $\{I_1, I_3\}$ comme instance de la propriété P, alors l'instance $\{I_1, I_2\}$ est supprimée.

La gestion de ces deux types d'axiomes permet (1) de reclassifier les individus appartenant à des classes disjointes lors de l'exécution des règles; (2) de modifier la valeur des propriétés d'arité 1 lors de l'exécution des règles.

8.1.4 Gestionnaire d'événements

La gestion des événements dans le prototype de l'infrastructure PERSEWS s'appuie sur l'implémentation Apache Muse [5] de la spécification WS-BaseNotification [29] qui standardise la manière dont les Services Web interagissent suivant le mode d'interaction publication/souscription. Cette spécification fournit des contrats de service standards pour les services fournisseurs et consommateurs d'événements. Le contrat de service pour les fournisseurs décrit les échanges de messages permettant aux consommateurs de souscrire aux événements. Le contrat de service pour les consommateurs décrit les échanges de messages permettant aux consommateurs de recevoir les messages auxquels ils ont souscrit.

Le gestionnaire d'événements souscrit auprès de tous les fournisseurs d'événements qui s'enregistrent dans l'annuaire afin de recevoir tous les messages publiés par les services de l'environnement. Lors de la réception d'un événement, le gestionnaire d'événement exploite le modèle de traduction montant fourni dans le contrat du service fournisseur pour transformer les données utiles du message du format XML au format OWL. Les informations contenues dans le message sont alors enregistrées dans la base de connaissances afin d'être exploitées lors de l'exécution des règles. Le gestionnaire d'événements assure également la vérification des axiomes OWL présentée dans la section 8.1.3 afin de maintenir la consistance de la base de connaissances.

8.1.5 Éditeur de règles

L'éditeur de règles est basé sur l'extension SWRLJessTab pour l'éditeur d'ontologies Protégé. Nous avons développé une version autonome de cette extension, dotée d'un talon client

[5] Apache Muse - A Java-based implementation of WSRF 1.2, WSN 1.3, and WSDM 1.1, http://ws.apache.org/muse/

lui permettant de communiquer avec le moteur d'inférence afin (1) d'initialiser le modèle de contexte avec le modèle contenu dans la base de connaissance du moteur d'inférence; (2) d'envoyer les règles à exécuter au moteur d'inférence. La figure 8.4 présente l'interface graphique de l'éditeur de règles. Cette interface dispose de trois fonctions permettant de mettre à jour le modèle de contexte à partir d'une base de connaissances, d'envoyer une règle au moteur d'inférence en mode d'exécution unique ou en mode d'exécution perpétuel.

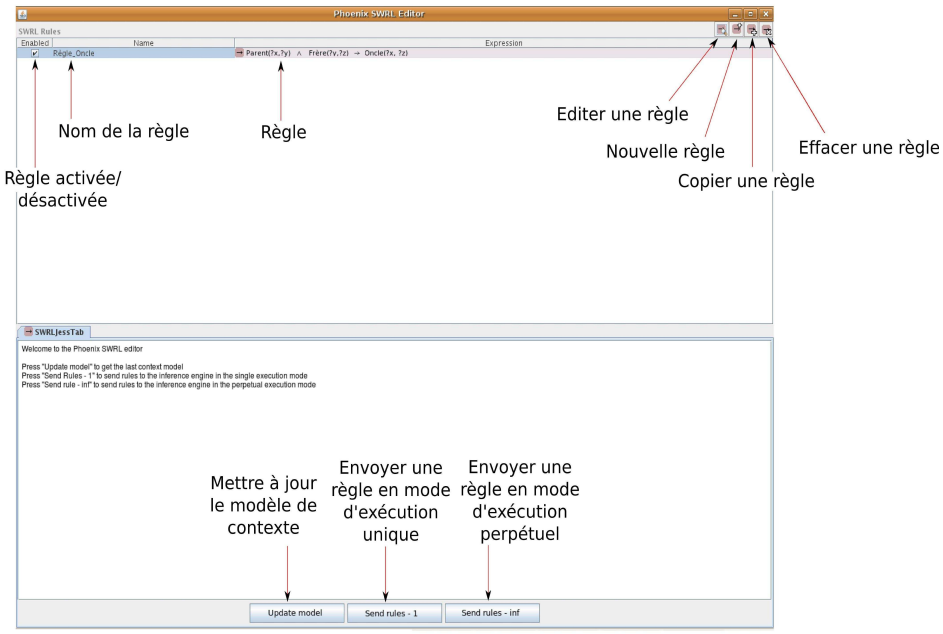

FIG. 8.4 – Interface graphique de l'éditeur de règles.

Lors de la création d'une nouvelle règle, la fenêtre d'édition présentée dans la figure 8.5 permet d'éditer une règle SWRL. Cette fenêtre offre un menu contextuel qui permet de consulter les classes, propriétés, individus et primitives disponibles dans le modèle. Les atomes spécifiés dans la zone d'édition de règles sont automatiquement complétés en fonction des classes, propriétés, individus et primitives du modèle à l'aide de la touche *tabulation*. Lorsqu'une règle est syntaxiquement incorrecte, la zone d'édition est encadrée de rouge et un message d'erreur fournit des informations sur la nature de l'erreur. La complétion, le menu contextuel et la vérification syntaxique guident le programmeur de l'environnement lors du développement d'applications ubiquitaires.

8.2 Évaluation du prototype

L'évaluation du prototype de l'infrastructure PERSEWS est constituée de deux volets. Dans un premier temps, nous évaluons le générateur selon les critères du nombre de lignes générées et du temps nécessaire à la génération pour différents types de services. Dans un deuxième temps, nous évaluons les performances de l'infrastructure en ce qui concerne la découverte et

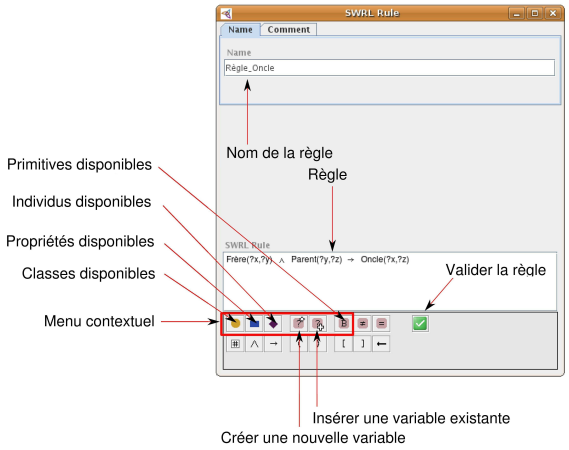

FIG. 8.5 – Fenêtre d'édition des règles.

l'invocation de services. Finalement, nous étudions l'impact de la génération de code sur les performances de l'infrastructure.

8.2.1 Évaluation du générateur

Le générateur de l'infrastructure PERSEWS est le composant qui produit le code nécessaire à l'invocation des services par le moteur d'inférence. La génération de code a lieu durant l'exécution, c'est à dire que le code est généré dynamiquement lorsque de nouveaux services apparaissent dans l'environnement. Nous mesurons dans un premier temps la quantité de code générée pour différents services afin de mettre en évidence l'intérêt de l'approche générative.

8.2.1.1 Génération de code

Afin de tester le générateur de l'infrastructure PERSEWS, nous avons développé un ensemble de Services Web pour un environnement ubiquitaire et l'ontologie qui modélise cet environnement. L'environnement ubiquitaire contient deux caméras Axis [6] contrôlables à distance, deux lecteurs multimédias basés sur le lecteur vlc [7] et un agenda partagé basé sur le service Google Agenda [8]. Nous avons déployé ces éléments sous forme de Services Web dans l'environnement :

- un service MouvementCaméra qui permet de contrôler la caméra à distance (rotation, inclinaison, zoom, déplacement) ;
- un service LecteurMultimédia qui permet de définir le flux à afficher ;
- un service DétectionDeMouvement qui permet d'activer et désactiver la détection de mouvement sur les caméras et configurer le type d'action à effectuer en cas de détection de mouvement (envoi de mail ou déclenchement d'alerte) ;

[6] Axis communications, http://www.axis.com/
[7] VideoLAN - VLC media player, http://www.videolan.org/
[8] Google agenda, http://www.google.com/calendar/

– un service Agenda permettant d'ajouter des activités dans l'agenda et d'accéder aux activités enregistrées.

Les contrats WSDL de ces services sont présentés en annexe A et l'ontologie correspondante est présentée en annexe B. Le tableau 8.1 détaille le nombre d'opérations et d'annotations sémantiques spécifiées dans les contrats de ces services.

Service	Nombre d'opérations	Nombre d'annotations sémantiques
MouvementCaméra	4	5
DétectionDeMouvement	6	7
LecteurMultimédia	1	2
Agenda	2	4

TAB. 8.1 – Nombre d'opérations et d'annotations sémantiques pour différents services d'un environnement ubiquitaire.

On peut constater que l'intégration des services dans l'infrastructure PERSEWS nécessite un faible nombre d'annotations sémantiques sur les contrats de services.

Nous alimentons le générateur avec ces différents contrats de service afin de produire le code permettant au moteur d'inférence d'invoquer ces services. La figure 8.6 présente le nombre de lignes de code générées pour chacun des contrats de service.

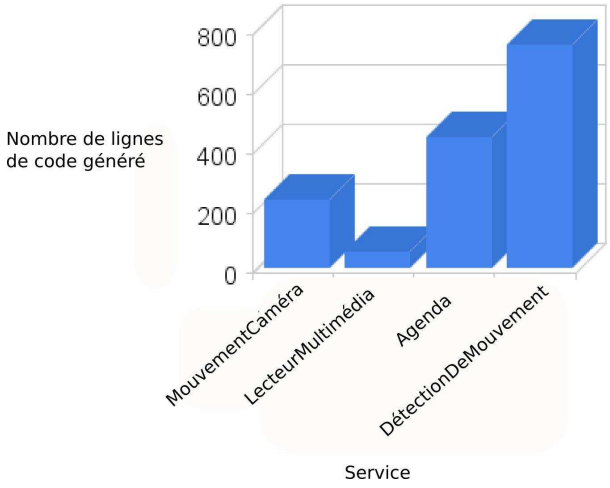

FIG. 8.6 – Nombre de lignes de code générées par contrat de service.

Le nombre de lignes générées va de 53 lignes pour le service LecteurMultimédia à 751 lignes pour le service de détection de mouvements. Le nombre de lignes générées varie en fonction du nombre d'opérations, de la complexité des types d'entrée et de sortie des opérations et du nombre d'annotations sémantiques. Ce code est généré automatiquement lors de la découverte de services, et devrait en l'absence du générateur être développé manuellement. A partir d'un faible nombre d'annotations sur les contrats de service (cf. tableau 8.1), le générateur produit un nombre important de lignes de codes (cf. figure 8.6) qui permettent la découverte et l'invocation de ce service à l'aide du langage de règles dans l'infrastructure PERSEWS.

8.2.1.2 Performances du générateur

La génération automatique du code par le générateur possède un coût en temps. Cette durée correspond à l'analyse de l'arbre XML représentant le contrat de service, à la génération de code pour le moteur d'inférence et à l'ajout de l'individu représentant le service dans la base de connaissances. La figure 8.7 présente le temps nécessaire à la réalisation de ces tâches pour les services décrits dans la section 8.2.1.1. Les mesures ont été effectuées sur un ordinateur doté d'un processeur Intel T1300 cadencé à 1.6 GHz et possédant 1 Go de mémoire RAM.

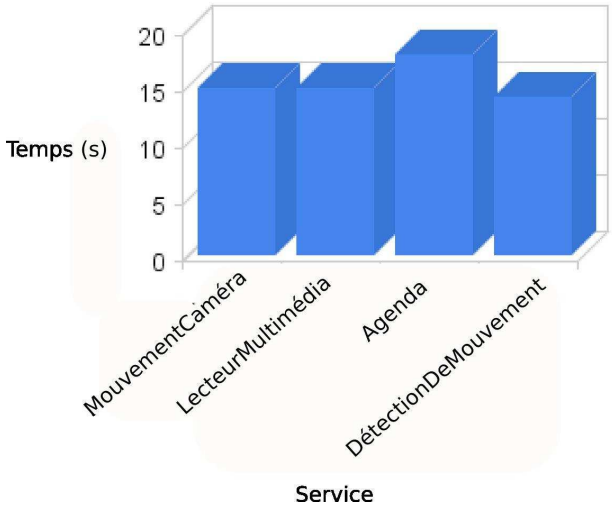

FIG. 8.7 – Temps nécessaire à la génération de code par contrat de service.

Le temps moyen de génération de code pour un contrat de service est de 15 secondes. La génération de code pour une classe de service a lieu lors de la découverte de la première instance de cette classe, et le code généré est générique à l'ensemble des individus de cette classe. Le coût de la génération devient donc négligeable lorsque l'environnement est saturé de services, ce qui est un cas courant dans les environnements ubiquitaires. Dans la section suivante, nous contrastons ce temps par rapport au temps de découverte et d'invocation des services.

8.2.2 Évaluation de la découverte et de l'invocation

Dans cette section, nous étudions les performances de l'infrastructure PERSEWS en ce qui concerne la découverte et l'invocation de services. Les environnements ubiquitaires sont des environnements saturés de périphériques, aussi la mesure de performance doit-elle porter sur un nombre conséquent d'instances de services afin de valider le passage à l'échelle de l'infrastructure.

8.2.2.1 Performances de la découverte et de l'invocation

Nous mesurons le temps de découverte de services et d'invocation de service pour un nombre de services variant de 1000 à 9000 instances. Pour ce test, nous avons utilisé le service Mouve-

mentCaméra présenté dans la section 8.2.1.1. Le test consiste à découvrir l'ensemble des services MouvementCaméra et à effectuer une rotation de 10 degrés sur l'ensemble de ces services.

Nous comparons les résultat obtenus en utilisant l'infrastructure PERSEWS avec les résultats obtenus en utilisant une infrastructure standard. Cette dernière est basée sur un annuaire jUDDI accédé au moyen de l'API UDDI4J [9] et sur un moteur SOAP axis2.

La découverte de services en utilisant l'infrastructure PERSEWS consiste à exécuter un unique atome MouvementCaméra(?s). La découverte et l'invocation consistent à exécuter la règle suivante qui sélectionne l'ensemble des services de la classe *MouvementCaméra* et invoque l'opération de rotation sur chacun de ces services.

MouvementCaméra(?s) → MouvementCaméra:rotation(?s, 10)

La découverte de services en utilisant l'infrastructure standard consiste à créer une structure tModel représentant l'interface du service désiré, à émettre une requête auprès de l'annuaire UDDI pour obtenir les services correspondants à cette structure tModel et à extraire les adresses de services de la réponse à la requête. Ces adresses sont utilisées pour spécifier les points d'accès du talon client lors de l'invocation.

La figure 8.8 présente le temps nécessaire pour la découverte seule et pour la découverte suivie de l'invocation de toutes les instances de service avec l'infrastructure PERSEWS d'une part et avec l'infrastructure standard d'autre part.

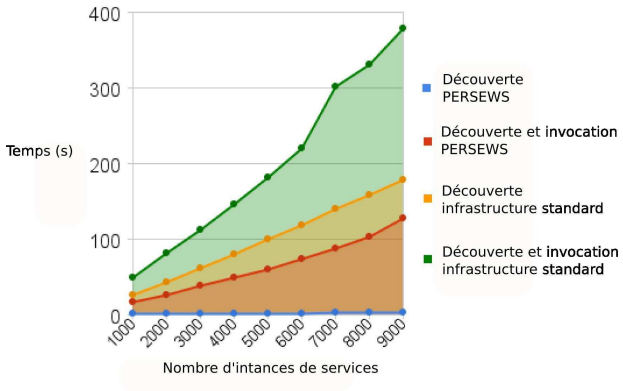

FIG. 8.8 – Comparaison des performances de découverte et d'invocation avec l'infrastructure PERSEWS et avec l'infrastructure standard.

Les résultats obtenus montrent que la découverte de services est en moyenne 45 fois plus rapide avec l'infrastructure PERSEWS qu'avec l'infrastructure standard. Ceci s'explique par le fait que la découverte de services dans l'infrastructure PERSEWS consiste en une simple requête sur la base de connaissances afin d'extraire les services disponibles dans l'environnement. En revanche, la découverte avec l'API UDDI4J requiert la manipulation de classes JAVA complexes.

Les résultats concernant la découverte suivie de l'invocation montrent des performances en moyenne 3 fois supérieures avec l'infrastructure PERSEWS. Ceci s'explique d'une part par les

[9] UDDI4J, http://uddi4j.sourceforge.net/

performances de la découverte de services exposées précédemment, d'autre part par le fait que la découverte de services à l'aide de l'API jUDDI retourne des structure de données complexes qui doivent être explorées afin d'extraire les informations nécessaires à l'invocation. Dans le cas de l'infrastructure PERSEWS, ces données sont stockées dans la base de connaissances lors de la publication du contrat de service dans l'annuaire.

La génération du code permettant au moteur d'inférence d'invoquer les services et l'ajout des individus représentant les services lors de la publication des contrats de services dans l'annuaire a également un impact sur les performances générales de l'infrastructure. Dans la section suivante, nous mesurons cet impact sur la découverte et l'invocation de services.

8.2.2.2 Impact de la génération sur les performances

Lors de la publication d'un contrat de service dans l'annuaire de l'infrastructure PERSEWS, le code permettant au moteur d'inférence d'invoquer ce service est généré et un individu représentant ce service est ajouté à la base de connaissances. Le code généré pour une classe de service est générique à l'ensemble des individus de cette classe. De ce fait, le coût de la génération est constant quel que soit le nombre d'instances de la classe de service présentes dans l'environnement. Ce coût a été évalué dans la section 8.2.1.2 et représente en moyenne 15 secondes. La figure 8.9 présente les performances comparées de la découverte et de l'invocation pour un nombre d'instances de services variant de 1000 à 9000 instances avec l'infrastructure standard et avec l'infrastructure PERSEWS en tenant compte du temps de génération.

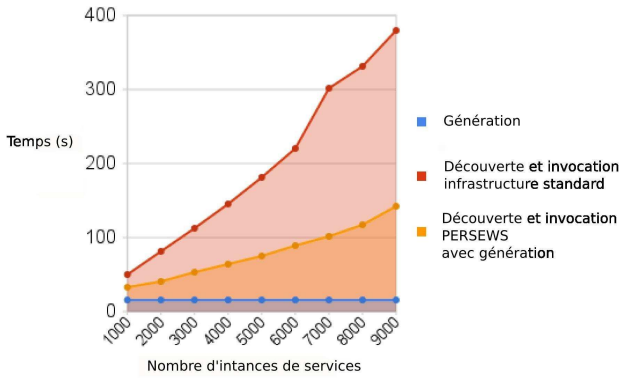

FIG. 8.9 – Comparaison des performances de découverte et d'invocation avec l'infrastructure PERSEWS et avec l'infrastructure standard en prenant en compte le temps de génération.

Les résultats montrent que les performances de découverte et d'invocation de l'infrastructure PERSEWS restent supérieure à celles de l'infrastructure standard lorsque l'on prend en compte le coût de la génération.

8.3 Synthèse

Nous avons réalisé un prototype de l'infrastructure PERSEWS afin de valider notre approche. Ce prototype nous a permis d'évaluer les performances de l'infrastructure PERSEWS en terme de nombre de lignes de code générées et de temps de découverte et d'invocation d'un ensemble de services. Le nombre de lignes de code générées montre que l'infrastructure facilite l'intégration des services en automatisant une partie du développement qui serait fastidieuse à réaliser à la main. Les résultats de l'évaluation du temps de découverte et d'invocation d'un ensemble de services montrent que l'infrastructure PERSEWS est nettement plus performante que l'infrastructure standard à laquelle elle est comparée. Ce résultat s'explique principalement par le fait que les services sont abstraits dans le modèle de contexte lors de la publication du contrat de service, aussi la découverte de services dans l'infrastructure PERSEWS consiste en une simple requête sur la base de connaissance.

Une sortie, c'est une entrée que l'on prend dans l'autre sens.

Boris Vian

9 Conclusion

La programmation d'applications ubiquitaires consiste à structurer un ensemble de services disponibles dans l'environnement pour assister les utilisateurs dans leurs activités au sein de l'environnement. Ces services sont hébergés sur des périphériques fixes ou mobiles. Les périphériques mobiles peuvent entrer et sortir de l'environnement, ce qui influence la disponibilité des fonctionnalités offertes par cet environnement. Qu'ils soient fixes ou mobiles, les périphériques qui hébergent les services sont de nature disparate et possèdent des caractéristiques matérielles hétérogènes et des systèmes d'exploitation différents. Les services qu'ils hébergent sont développés à l'aide de langages de programmation différents et s'exécutent sur diverses plates-formes d'exécution. Cette hétérogénéité matérielle et logicielle constitue un obstacle à l'utilisation des fonctionnalités offertes par les services pour accomplir des tâches communes. De plus, afin d'offrir des services en adéquation avec les activités des utilisateurs, il est nécessaire d'exploiter le contexte dans lequel se déroulent ces activités. La programmation d'applications ubiquitaires nécessite donc un paradigme de programmation à un haut niveau d'abstraction qui libère le programmeur des tâches de bas niveau telles que la découverte de services ou l'adaptation à la disparition de services. Il peut ainsi se concentrer sur la réalisation des intentions des utilisateurs.

Nous avons présenté dans ce document une approche pour le développement d'applications ubiquitaires. Cette approche repose sur un modèle de contexte spécifié sous forme d'ontologie, un langage de règles extensible et une infrastructure logicielle basée sur une architecture orientée service. Les concepts définis dans le modèle de contexte forment les constructions du langage de règles. Ce langage est dynamiquement étendu pour offrir des abstractions permettant d'accéder aux services de l'environnement. L'infrastructure PERSEWS constitue un environnement de développement pour le langage de règles logiques et un environnement d'exécution pour les règles logiques. Cette infrastructure est basée sur une architecture orientée service, ce qui permet l'intégration de services hétérogènes en termes de langage de programmation et de plate-forme

d'exécution.

Ce chapitre présente une évaluation de notre approche, dresse le bilan des contributions apportées par cette thèse, puis examine des directions de recherches futures.

9.1 Évaluation

Afin de valider notre approche, nous l'évaluons selon les critères définis dans la section 2.1.3.1. Ces critères sont les suivants: gestion de l'hétérogénéité, découverte de services, capacité d'adaptation, développement et déploiement, sensibilité au contexte. Nous étudions les capacités de l'infrastructure PERSEWS à l'aune de ces critères.

Gestion de l'hétérogénéité. Les Services Web sont accessibles sous forme de boites noires dont la seule partie visible est le contrat de service qui fournit les détails d'invocation. De ce fait, l'invocation d'un Service Web peut se faire sans connaissance préalable des détails techniques d'implémentation du service. L'infrastructure PERSEWS est construite sur une architecture orientée service, et ses composants sont développés sous forme de Services Web. Cette infrastructure permet d'intégrer les services de l'environnement qui respectent l'architecture des Services Web, indépendamment des langages de programmation avec lesquels les services ont été développés ou des plates-formes sur lesquels ils s'exécutent.

Découverte de services. L'annuaire de l'infrastructure PERSEWS permet aux fournisseurs de services de publier leur contrat de service et aux consommateurs de découvrir ces services. Les services dont le contrat est publié dans l'annuaire sont abstraits sous forme d'individus dans le modèle de contexte. De ce fait, la découverte de services consiste pour le programmeur de l'environnement à spécifier dans une règle logique la classe de service recherchée. Lors de l'exécution de la règle logique, le moteur d'inférence de l'infrastructure PERSEWS extrait les individus appartenant à cette classe dans le modèle de contexte.

Capacité d'adaptation. Le modèle de contexte de l'infrastructure PERSEWS reflète en permanence l'état de l'environnement ubiquitaire. En particulier, les services disponibles dans l'environnement sont abstraits sous forme d'individus appartenant à des classes de services dans le modèle de contexte. Les règles logiques spécifiées pas le programmeur font référence à ces classes de services et aux fonctionnalités qu'elles fournissent. Lors de l'exécution des règles logiques, l'infrastructure établit dynamiquement la relation entre les fonctionnalités spécifiées dans les règles logiques et les services disponibles dans l'environnement.

Développement et déploiement. L'infrastructure PERSEWS permet de développer des applications ubiquitaires à un haut niveau d'abstraction en manipulant directement les concepts spécifiés dans le modèle de contexte et les extensions syntaxiques représentant les fonctionnalités des services. En termes de déploiement, on peut distinguer le déploiement des services sur l'infrastructure PERSEWS et le déploiement des applications ubiquitaires. Le déploiement des services sur l'infrastructure PERSEWS est facilité par l'approche générative mise en œuvre. En effet, les services qui publient leurs contrats dans l'annuaire de l'infrastructure PERSEWS sont automatiquement intégrés à l'infrastructure. Un individu

représentant le nouveau service est ajouté au modèle de contexte et le code d'invocation du service est généré. Pour le déploiement des applications ubiquitaires, les règles spécifiées dans l'éditeur de règles peuvent être déployées sur l'infrastructure PERSEWS à distance. C'est l'éditeur de règles qui gère la transmission des règles et leur sauvegarde dans la base de connaissances.

Sensibilité au contexte. Le gestionnaire d'événements gère la réception et le traitement des informations contextuelles envoyées par les services de l'environnement. Ces informations contextuelles sont centralisées dans la base de connaissances et peuvent être exploitées dans le langage de règles logique. Ainsi, l'infrastructure PERSEWS permet de manipuler les informations de contexte à un haut niveau d'abstraction dans les applications ubiquitaires.

9.2 Contributions

Les contributions apportées dans cette thèse peuvent être regroupées en deux volets. Dans un premier temps, nous proposons une approche langage pour la programmation des environnements ubiquitaires. Dans un deuxième temps, nous mettons en œuvre cette approche dans une infrastructure logicielle basée sur les architectures orientées service.

9.2.1 Approche langage pour la programmation des environnements ubiquitaires

Nous proposons une approche langage basée sur un couplage entre un modèle de contexte spécifié sous forme d'ontologie et un langage de règles logiques extensible. Le modèle de contexte définit les concepts pertinents pour la programmation d'un environnement ubiquitaire. Ce modèle constitue une représentation de l'environnement qui est maintenue par l'infrastructure. L'infrastructure met à jour dynamiquement le modèle de contexte avec les services disponibles dans l'environnement et les informations de contexte fournies par ces services.

Le développement des applications repose sur un langage de règles logiques extensible. Les constructions de ce langage sont définies par les concepts du modèle de contexte. De ce fait, le programmeur de l'environnement peut manipuler directement des concepts de haut niveau dans la logique des applications. Le langage de règles logiques est étendu dynamiquement avec des constructions permettant de contrôler les services qui apparaissent dans l'environnement. Le langage de règles logiques extensible, couplé au modèle de contexte permet de programmer des applications ubiquitaires à un haut niveau d'abstraction.

9.2.2 Infrastructure PERSEWS

Notre approche langage est mise en œuvre dans l'infrastructure PERSEWS. Cette infrastructure logicielle constitue un environnement de développement pour le langage de règles et un environnement d'exécution pour les règles spécifiées par le programmeur de l'environnement.

Durant le développement, l'éditeur de règles de l'infrastructure exploite le modèle de contexte pour guider le programmeur dans le développement des applications. L'éditeur de règles prend également en charge le déploiement des applications sur l'infrastructure. Durant l'exécution, l'infrastructure PERSEWS gère la découverte de services, l'intégration des services dans l'infrastructure, l'adaptation des applications en fonction de la disponibilité des services et la mise à jour du modèle de contexte. De plus, l'architecture orientée service sur laquelle est construite l'infrastructure PERSEWS permet d'exploiter les fonctionnalités offertes par des services hétérogènes.

9.3 Perspectives

Les applications ubiquitaires ont pour but d'assister les utilisateurs dans leur environnement quotidien et de « se mêler à l'élaboration de leur vie quotidienne jusqu'à en devenir indiscernables ». Dans cette optique, ces applications doivent posséder trois propriétés que sont la sûreté de fonctionnement, la sécurité et l'utilisabilité.

9.3.1 Sûreté de fonctionnement

La sûreté de fonctionnement consiste à assurer que l'exécution d'une application produira l'effet attendu sans générer d'erreur. Une perspective pour l'infrastructure PERSEWS est de garantir que les différentes règles qui régissent le comportement d'une application n'entrent pas en conflit. Une approche intéressante est proposée dans les travaux de Moyaux et al. [63] qui modélisent sous forme d'ontologie les interdépendances entre activités (e.g. exclusion mutuelle entre activités ou synchronisation entre activités). Dans cette approche, le langage de règles SWRL est utilisé pour raisonner sur l'ontologie et déterminer les activités qui peuvent survenir. Cette approche pourrait s'intégrer à l'approche mise en œuvre dans infrastructure PERSEWS, ces deux approches reposant sur un modèle ontologique et un mécanisme de raisonnement basé sur les langages de règles.

D'autre part, certains domaines peuvent nécessiter de définir des propriétés qui s'appliquent à l'ensemble du domaine et doivent être respectées par l'ensemble des applications ubiquitaires du domaine. Dans le cadre de l'architecture PERSEWS, de telles règles doivent être spécifiées par l'architecte du modèle qui possède une bonne connaissance du domaine d'application pour lequel l'environnement ubiquitaire est conçu. Ces propriétés du domaine peuvent être spécifiées sous forme de règles logiques et stockées dans la base de connaissance du moteur d'inférence. Elles peuvent alors être exécutées avec les règles du programmeur de l'environnement pour assurer que les règles qui constituent une application ubiquitaire n'enfreignent pas les propriétés du domaine.

9.3.2 Sécurité

La recherche dans le domaine de l'informatique ubiquitaire est centrée sur le développement d'infrastructures permettant de connecter des périphériques et de programmer des applications

pour fournir des fonctionnalités de plus en plus avancées aux utilisateurs. La sécurité et la confidentialité, qui consistent à assurer que les données échangées ne sont pas accessibles par des tiers non autorisés et ne sont pas altérées durant les échanges, sont deux enjeux qui doivent être également pris en compte. Langheinrich [55] nous met en garde contre la possibilité d'un cauchemar orwellien si la recherche dans le domaine de l'informatique ubiquitaire néglige les aspects sécuritaires.

Un effort conséquent concernant la sécurité des architectures orientées service est la spécification Web Service Security [25] qui propose un ensemble d'extensions SOAP standards permettant de garantir l'intégrité et la confidentialité des messages échangés entre Services Web. Cette spécification est implémentée dans le module Rampart [1] pour le moteur SOAP Axis2. Une perspective pour l'infrastructure PERSEWS est d'inclure les aspects sécuritaires dans le modèle de contexte à l'aide de l'ontologie pour la sécurité définie par Kim et al. [50] afin de spécifier le degré de sécurité requis par chaque service de l'environnement. Le mécanisme de sécurité requis serait alors mis en œuvre grâce au module Rampart.

9.3.3 Utilisabilité

L'utilisabilité est définie par la norme ISO 9241-11 [45, 8] comme « le degré selon lequel un produit peut être utilisé, par des utilisateurs identifiés, pour atteindre des buts définis avec efficacité, efficience et satisfaction, dans un contexte d'utilisation spécifié ». Si les applications ubiquitaires doivent régir le quotidien des utilisateurs, il est souhaitable que les utilisateurs puissent spécifier eux-mêmes la logique de ces applications. L'approche langage proposée dans ce document offre un paradigme de programmation à un haut niveau d'abstraction, ce qui permet aux utilisateurs d'appréhender les concepts manipulés dans les applications ubiquitaires. Toutefois, une approche graphique à la programmation par règles permettrait à des utilisateurs non programmeurs de comprendre et spécifier le comportement des applications ubiquitaires. Dans ce domaine, plusieurs approches existantes proposent des interfaces graphiques basées sur des métaphores et pourraient fournir une couche d'abstraction additionnelle pour la programmation d'applications dans l'infrastructure PERSEWS. Par exemple, Truong et al. [87] proposent une interface graphique basée sur une métaphore de mots magnétiques. Les mots magnétiques consistent en un ensemble de pièces aimantées sur lesquelles sont imprimés des mots que les utilisateurs combinent pour former des phrases. L'interface CAMP reproduit ce mécanisme au travers d'une interface graphique afin de permettre aux utilisateurs de programmer des applications ubiquitaires à partir d'un vocabulaire contraint. Humble et al. [41] proposent une interface graphique basée sur une métaphore de puzzle où chaque service est représenté par une icône en forme de pièce de puzzle. Les utilisateurs assemblent ces pièces pour former une application ubiquitaire. Les interfaces graphiques proposées par ces deux approches pourraient améliorer l'utilisabilité de l'infrastructure PERSEWS.

[1] Securing SOAP Messages with Rampart, http://ws.apache.org/axis2/modules/rampart/

Bibliographie

[1] Gregory D. ABOWD, Anind K. DEY, Peter J. BROWN, Nigel DAVIES, Mark SMITH et Pete STEGGLES : Towards a better understanding of context and context-awareness. *In HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[2] Gregory D. ABOWD, Anind K. DEY, Robert ORR et Jason BROTHERTON : Context-awareness in wearable and ubiquitous computing. *In ISWC '97: Proceedings of the 1st IEEE International Symposium on Wearable Computers*, page 179, Washington, DC, USA, 1997. IEEE Computer Society.

[3] Guruduth BANAVAR, James BECK, Eugene GLUZBERG, Jonathan MUNSON, Jeremy SUSSMAN et Deborra ZUKOWSKI : Challenges: an application model for pervasive computing. *In MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 266–274, New York, NY, USA, 2000. ACM.

[4] Sean BECHHOFER, Frank van HARMELEN, Jim HENDLER, Ian HORROCKS, Deborah L. MCGUINNESS, Peter F. PATEL-SCHNEIDER et Lynn Andrea STEIN : OWL Web Ontology Language Reference. Rapport technique REC-owl-ref-20040210, The Worldwide Web Consortium, February 2004.

[5] Tim BERNERS-LEE : Semantic Web presentation at XML2000, 2000.

[6] Tim BERNERS-LEE, James HENDLER et Ora LASSILA : The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[7] Philip A. BERNSTEIN : Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[8] Patricia A. BILLINGSLEY : Interface: Ergonomic standards go beyond hardware. *j-IEEE-SOFTWARE*, 11(2):82–84, mars 1994. Discusses ISO 9241, "Ergonomic Requirements for Office Work with Visual Display Terminals".

[9] BLUETOOTH : Specification of the bluetooth system. Bluetooth specification, Bluetooth, décembre 1999.

[10] Harold BOLEY, Said TABET et Gerd WAGNER : Design rationale for ruleml: A markup language for semantic web rules. *In SWWS*, pages 381–401, 2001.

[11] Grady BOOCH, James RUMBAUGH et Ivar JACOBSON : *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[12] David BOOTH, Hugo HAAS, Francis MCCABE, Eric NEWCOMER, Michael CHAMPION, Chris FERRIS et David ORCHARD : *Web Services Architecture*, February 2004. http://www.w3.org/TR/ws-arch/.

[13] Dan BRICKLEY et Ramanathan V.GUHA : Rdf vocabulary description language 1.0: Rdf schema. W3C Recommendation (http://www.w3.org/TR/rdf-schema/), 2004.

[14] Peter J. BROWN, John D. BOVEY et Xian CHEN : Context-aware applications: from the laboratory to the marketplace. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 4(5):58–64, Oct 1997.

[15] T. CATARCI, M. de LEONI, A. MARRELLA, M. MECELLA, B. SALVATORE, G. VETERE, S. DUSTDAR, L. JUSZCZYK, A. MANZOOR et Hong-Linn TRUONG : Pervasive software environments for supporting disaster responses. *Internet Computing, IEEE*, 12(1):26–37, Jan.-Feb. 2008.

[16] Renato CERQUEIRA, Carlos CASSINO et Roberto IERUSALIMSCHY : Dynamic component gluing across different componentware systems. *In DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications*, pages 362–371, Washington, DC, USA, 1999. IEEE Computer Society.

[17] Pierre CHÂTEL : "wsdl 2.0 to uddi mapping, sawsdl to uddi mapping". Technical note, Thales, 29 May 2006.

[18] Harry CHEN, Filip PERICH, Tim FININ et Anupam JOSHI : Soupa: Standard ontology for ubiquitous and pervasive applications. *In International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pages 258–267, Aug. 2004.

[19] Roberto CHINNICI, Jean-Jacques MOREAU, Arthur RYMAN et Sanjiva WEERAWARANA : "web services description language (wsdl) version 2.0 part 1: Core language". W3c recommendation, The WorldWide Web Consortium, June 2007. http://www.w3.org/TR/wsdl20/.

[20] The Unicode CONSORTIUM : The unicode standard, version 5.0, 2006.

[21] Digital Equipment CORPORATION, Intel CORPORATION et Xerox CORPORATION : The ethernet: a local area network: data link layer and physical layer specifications. *SIGCOMM Comput. Commun. Rev.*, 11(3):20–66, 1981.

[22] Wolfgang EMMERICH : Software engineering and middleware: a roadmap. *In ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 117–129, New York, NY, USA, 2000. ACM.

[23] Deborah ESTRIN, David CULLER, Kris PISTER et Gaurav SUKHATME : Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 01(1):59–69, 2002.

[24] Dieter FENSEL, Frank van HARMELEN, Ian HORROCKS, Deborah L. MCGUINNESS et Peter F. PATEL-SCHNEIDER : Oil: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[25] Organization for the Advancement of STRUCTURED INFORMATION STANDARDS : Web services security: Soap message security 1.1 (ws-security 2004), 2006.

[26] Ernest FRIEDMAN-HILL : *Jess in Action*. Manning, 2003.

[27] Christine GOLBREICH : Combining rule and ontology reasoners for the semantic web. *In Proc. of RuleML 2004, number 3323 in LNCS*, pages 6–22. Springer, 2004.

[28] Charles F. GOLDFARB et Paul PRESCOD : *The XML handbook*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[29] Steve GRAHAM, David HULL et Bryan MURRAY : Oasis web services notification (wsn) tc, 2007.

[30] William GROSSO : *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, 2002.

[31] Object Management GROUP : The common object request broker: Architecture and specification. OMG Document 91.12.1 Revision 1.1, 1992.

[32] Thomas R. GRUBER : A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[33] Martin GUDGIN, Marc HADLEY, Jean-Jacques MOREAU, Henrik Frystyk NIELSEN, Anish KARMARKAR et Yves LAFON : Soap 1.2 part 1: Messaging framework, 27 April 2007. http://www.w3.org/TR/soap12-part1/.

[34] Erik GUTTMAN, Charles PERKINS, John VEIZADES et Michael DAY : RFC 2608: Service Location Protocol, Version 2, juin 1999.

[35] Sumi HELAL, William MANN, Hicham EL-ZABADANI, Jeffrey KING, Youssef KADDOURA et Erwin JANSEN : The gator tech smart house: A programmable pervasive space. *Computer*, 38(3):50–60, 2005.

[36] Ian HORROCKS : DAML+OIL: a description logic for the semantic web. *IEEE Data Engineering Bulletin*, 25(1):4–9, 2002.

[37] Ian HORROCKS : Reasoning with expressive description logics: Theory and practice. *In CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 1–15, London, UK, 2002. Springer-Verlag.

[38] Ian HORROCKS et Peter F. PATEL-SCHNEIDER : A proposal for an owl rules language. *In WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 723–731, New York, NY, USA, 2004. ACM.

[39] Ian HORROCKS, Peter F. PATEL-SCHNEIDER, Harold BOLEY, Said TABET, Benjamin GROSOF et Mike DEAN : Swrl: A semantic web rule language combining owl and ruleml. W3C Member Submission 21 May 2004, 2004.

[40] Ian HORROCKS, Peter F. PATEL-SCHNEIDER et Frank van HARMELEN : From shiq and rdf to owl: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[41] Jan HUMBLE, Andy CRABTREE, Terry HEMMINGS, Karl-Petter ÅKESSON, Boriana KOLEVA, Tom RODDEN et Pär HANSSON : "playing with the bits" user-configuration of ubiquitous domestic environments. *In Ubicomp*, pages 256–263, 2003.

[42] IEEE 802.11 Wireless LANs Working Group. *TGn Sync Proposal Technical Specification*, May 18 2005, http://www.ieee802.org/16/tge/contrib/C80216e-04_529r2.pdf.

[43] Internet Engineering Task Force (IETF) : Rfc 3986: Uniform resource identifier (uri): Generic syntax, 2005. http://tools.ietf.org/html/rfc3986.

[44] The Infrared Data Association (IrDA) : Infrared communication. http://www.irda.org/, 2008.

[45] ISO : En iso 9241-11. ergonomic requirements for office work with visual display terminals (vdt's). part 11, 1997.

[46] Valérie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Françoise Sailhan, Rafik Chibout, Nicole Levy et Angel Talamona : Developing ambient intelligence systems: A solution based on web services. *Journal of Automated Software Engineering*, 12(1):101–137, 2005.

[47] Jini Architectural Overview. White Paper, 1999.

[48] Peter D. Karp, Vinay K. Chaudhri et Jerome Thomere : Xol: An xml-based ontology exchange language. Rapport technique, SRI International, 1999.

[49] Robert E. Kent : Conceptual knowledge markup language: An introduction. *Netnomics*, 2(2):139–169, 2000.

[50] Anya Kim, Jim Luo et Myong H. Kang : Security ontology for annotating resources. *In OTM Conferences (2)*, pages 1483–1499, 2005.

[51] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy et Mark A. Musen : The protégé owl plugin: An open development environment for semantic web applications. *In 3rd International Semantic Web Conference*, pages 229–243. Springer, 2004.

[52] Ed Krol et Ellen Hoffman : FYI on "What is the Internet?". RFC 1462 (Informational), mai 1993.

[53] Mohan Kumar, Behrooz A. Shirazi, Sajal K. Das, Byung Y. Sung, David Levine et Mukesh Singhal : Pico: A middleware framework for pervasive computing. *IEEE Pervasive Computing*, 02(3):72–79, 2003.

[54] Jason I. Hong; James A. Landay : An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2–4):287–303, Dec. 2001.

[55] Marc Langheinrich : Privacy by design - principles of privacy-aware ubiquitous systems. *In UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing*, pages 273–291, London, UK, 2001. Springer-Verlag.

[56] Ora Lassila et Ralph R. Swick : Resource description framework (rdf) model and syntax specification.

[57] Scott Lawrence : UPnP basic device definition version 1.0. UPnP Forum, décembre 2002.

[58] Sean Luke et Jeff Heflin : Shoe 1.01. proposed specification, shoe project technical report, 2000.

[59] David Martin, Mark Burstein, Drew Mcdermott, Sheila Mcilraith, Massimo Paolucci, Katia Sycara, Deborah L. Mcguinness, Evren Sirin et Naveen Srinivasan : Bringing semantics to web services with owl-s. *World Wide Web*, 10(3):243–277, 2007.

[60] Ryusuke MASUOKA, Bijan PARSIA et Yannis LABROU : Task computing - the semantic web meets pervasive computing. *In Proceedings of 2nd International Semantic Web Conference (ISWC2003), Sanibel Island*, pages 866–881, 2003.

[61] David E. MCDYSAN et Darren L. SPOHN : *ATM: theory and application*. McGraw-Hill, Inc., New York, NY, USA, 1994.

[62] Richard MONSON-HAEFEL et Dave CHAPPELL : *Java Message Service*. O'Reilly & Associates, Sebastopol, CA, USA, December 2000.

[63] Thierry MOYAUX, Ben Lithgow SMITH et Shamimabi PAUROBALLY : Towards service-oriented ontology-based coordination. *In ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 265–274, Washington, DC, USA, 2006. IEEE Computer Society.

[64] Eric NEWCOMER : *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[65] OASIS : UDDI Version 3.0.2. Technical Standard, Feb 2005. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[66] David A. PATTERSON et John L. HENNESSY : *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[67] Anand RANGANATHAN, Robert E. MCGRATH, Roy H. CAMPBELL et M. Dennis MICKUNAS : Use of ontologies in a pervasive computing environment. *Knowl. Eng. Rev.*, 18(3):209–220, 2003.

[68] Siegmund REDL, Malcolm W. OLIPHANT, Mathias K. WEBER et Matthias K. WEBER : *An Introduction to GSM*. Artech House, Inc., Norwood, MA, USA, 1995.

[69] Dumitru ROMAN, Uwe KELLER, Holger LAUSEN, Rubén Lara Jos de BRUIJN, Michael STOLLBERG, Axel POLLERES, Cristina FEIER, Christoph BUSSLER et Dieter FENSEL : Web service modeling ontology. *Applied Ontology*, 2005.

[70] Manuel ROMÁN, Christopher HESS, Renato CERQUEIRA, Anand RANGANATHAN, Roy H. CAMPBELL et Klara NAHRSTEDT : A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[71] Floyd E. ROSS : An overview of fddi: the fiber distributed data interface. *Selected Areas in Communications, IEEE Journal on*, 7(7):1043–1051, Sep 1989.

[72] The Rule Markup Initiative. *RuleML*. http://www.ruleml.org/.

[73] Debashis SAHA et Amitava MUKHERJEE : Pervasive computing: A paradigm for the 21st century. *Computer*, 36(3):25–31, 2003.

[74] Daniel SALBER, Anind K. DEY et Gregory D. ABOWD : The context toolkit: aiding the development of context-enabled applications. *In CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM.

[75] Bill N. SCHILIT et Marvin M. THEIMER : Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, Sep/Oct 1994.

[76] Albrecht SCHMIDT, Michael BEIGL et Hans w. GELLERSEN : There is more to context than location. *Computers and Graphics*, 23:893–901, 1999.

[77] *Bluetooth Specification v1.1, Part E: Service Discovery Protocol (SDP).*

[78] Robert W. SEBESTA : *Concepts of Programming Languages*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[79] Yuanchun SHI, Weikai XIE, Guangyou XU, Runting SHI, Enyi CHEN, Yanhua MAO et Fang LIU : The smart classroom: merging technologies for seamless tele-education. *Pervasive Computing, IEEE*, 2(2):47–55, April-June 2003.

[80] Joáo Pedro SOUSA et David GARLAN : Aura: an architectural framework for user mobility in ubiquitous computing environments. *In WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[81] Raj SRINIVASAN : RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831 (Proposed Standard), août 1995.

[82] Leon STERLING et Ehud Y. SHAPIRO : *The Art of Prolog*. Logic Programming. MIT Press, Cambridge, Mass, 1994.

[83] Thomas STRANG et Claudia LINNHOFF-POPIEN : A context modeling survey. *In Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, September 2004.

[84] Thomas STRANG, Claudia LINNHOFF-POPIEN et Korbinian FRANK : CoOL: A Context Ontology Language to enable Contextual Interoperability. *In* Jean-Bernard STEFANI, Isabelle DAMEURE et Daniel HAGIMONT, éditeurs : *LNCS 2893: Proceedings of 4th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS2003)*, volume 2893 de *Lecture Notes in Computer Science (LNCS)*, pages 236–247, Paris/France, November 2003. Springer Verlag.

[85] Andrew S. TANENBAUM : *Computer networks (3rd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[86] Andrew S. TANENBAUM : *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[87] Khai N. TRUONG, Elaine M. HUANG et Gregory D. ABOWD : Camp: A magnetic poetry interface for end-user programming of capture applications for the home. *In Ubicomp*, pages 143–160, 2004.

[88] Kunal VERMA et Amit SHETH : Semantically annotating a web service. *IEEE Internet Computing*, 11(2):83–85, 2007.

[89] Xiao Hang WANG, Da Qing ZHANG, Tao GU et Hung Keng PUNG : Ontology based context modeling and reasoning using owl. *In PERCOMW '04: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, page 18, Washington, DC, USA, 2004. IEEE Computer Society.

[90] Zhenyu WANG et David GARLAN : Task-driven computing. Rapport technique CMU-CS-00-154, Carnegie Mellon University School of Computer Science, May 2000.

[91] David A. WATT : *Programming language concepts and paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[92] Mark WEISER : The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999.

[93] Mark WEISER : Some computer science issues in ubiquitous computing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):12, 1999.

[94] World Wide Web Consortium. *Web Services Architecture*, February 2004.

[95] Hubert ZIMMERMANN : Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on [legacy, pre - 1988]*, 28(4):425–432, Apr 1980.

Quatrième partie

Annexes

A

Contrats de services WSDL

A.1 Service de déplacement de caméra

```
<?xml version="1.0" encoding="UTF-8"?>
<description xmlns="http://www.w3.org/ns/wsdl"
             xmlns:impl="http://www.inria.fr/phoenix/calendar"
             targetNamespace="http://www.inria.fr/phoenix/calendar"
             xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
             xmlns:Ontology0="http://www.inria.fr/phoenix/contexte#">
    <types>
      <xsd:schema
          xmlns:apachesoap="http://xml.apache.org/xml-soap"
          xmlns:intf="http://www.inria.fr/phoenix/calendar"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="http://www.inria.fr/phoenix/calendar">
          <xsd:element name="getAllEvents">
             <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="user" type="xsd:string" />
                  <xsd:element name="password" type="xsd:string" />
                </xsd:sequence>
             </xsd:complexType>
          </xsd:element>
          <xsd:element name="getAllEventsResponse"
                       sawsdl:liftingSchemaMapping="http://melas.labri.fr:8080/XSLT/EventLifting
                           .xslt" />>
             <xsd:complexType>
                <xsd:sequence>
                  <xsd:element maxOccurs="unbounded"
                               name="getAllEventsReturn"
                               type="impl:Event"/>
                </xsd:sequence>
             </xsd:complexType>
```

```
        </xsd:element>
        <xsd:complexType name="Event">
           <xsd:sequence>
             <xsd:element name="location" nillable="true"
                          type="xsd:string" />
             <xsd:element name="startTime" nillable="true"
                          type="xsd:string" />
             <xsd:element name="stopTime" nillable="true"
                          type="xsd:string"/>
             <xsd:element name="title" nillable="true"
                          type="xsd:string" />
             <xsd:element name="user" nillable="true"
                          type="xsd:string" />
           </xsd:sequence>
        </xsd:complexType>
        <xsd:element name="addEvent">
           <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="user" type="xsd:string" />
                <xsd:element name="password" type="xsd:string" />
                <xsd:element name="minTime" type="xsd:string" />
                <xsd:element name="maxTime" type="xsd:string" />
                <xsd:element name="title" type="xsd:string"/>
                <xsd:element name="content" type="xsd:string"/>
                <xsd:element name="location" type="xsd:string" />
              </xsd:sequence>
           </xsd:complexType>
        </xsd:element>
     </xsd:schema>
  </types>
  <interface name="Calendar" sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
      Agenda">
     <operation name="getAllEvents"
                pattern="http://www.w3.org/ns/wsdl/in-out"
                sawsdl:modelReference="http://www.inria.fr/phoenix/context#
                    RecupereTousEvenements">
       <input element="impl:getAllEvents"/>
       <output element="impl:getAllEventsResponse"/>
     </operation>
     <operation name="addEvent"
                pattern="http://www.w3.org/ns/wsdl/in-only"
                sawsdl:modelReference="http://www.inria.fr/phoenix/context#AjouteEvenement">
       <input element="impl:addEvent"/>
     </operation>
  </interface>
  <binding xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
           name="CalendarSoapBinding" interface="impl:Calendar"
           type="http://www.w3.org/ns/wsdl/soap" wsoap:version="1.1"
           wsoap:protocol="http://www.w3.org/2006/01/soap11/bindings/HTTP/">
    <operation ref="impl:getAllEvents"/>
    <operation ref="impl:addEvent"/>
  </binding>
  <service name="CalendarService" interface="impl:Calendar" >
    <endpoint name="Calendar" binding="impl:CalendarSoapBinding"
              address="http://melas.labri.fr:8080/axis2/services/CalendarService"/>
  </service>
</description>
```

A.2 Service de lecteur multimédia

```
<?xml version="1.0" encoding="UTF-8"?>
<description xmlns="http://www.w3.org/ns/wsdl"
             xmlns:impl="http://www.inria.fr/phoenix/display"
             xmlns:tns="http://www.inria.fr/phoenix/display"
             xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
             xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
             xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions"
             xmlns:whttp="http://www.w3.org/ns/wsdl/http"
             targetNamespace="http://www.inria.fr/phoenix/display"
             xmlns:Ontology0="http://www.inria.fr/phoenix/contexte#"
             xmlns:sawsdl="http://www.w3.org/ns/sawsdl">
  <types>
       <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:intf="http://www.inria.fr/phoenix/display"
                   xmlns:apachesoap="http://xml.apache.org/xml-soap"
                   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
                   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                   attributeFormDefault="unqualified"
                   elementFormDefault="qualified"
                   targetNamespace="http://www.inria.fr/phoenix/display"
                   >
         <xsd:element name="display">
           <xsd:complexType>
               <xsd:sequence>
                 <xsd:element name="StreamURI" type="xsd:string" />
                 <xsd:element name="resolution" type="xsd:string" />
                 <xsd:element name="compression" type="xsd:int" />
                 <xsd:element name="fps" type="xsd:int" />
               </xsd:sequence>
           </xsd:complexType>
         </xsd:element>
       </xsd:schema>
  </types>
       <interface name="Display"
                  sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#Lecture">
               <operation name="display"
                          pattern="http://www.w3.org/ns/wsdl/in-only">
                 <input element="impl:display" />
               </operation>
       </interface>
       <binding name="DisplaySoapBinding"
                interface="tns:Display"
                type="http://www.w3.org/ns/wsdl/soap"
                wsoap:version="1.1"
                wsoap:protocol="http://www.w3.org/2006/01/soap11/bindings/HTTP/"
                whttp:queryParameterSeparatorDefault="&amp;">
         <operation ref="tns:display"/>
       </binding>
       <service name="DisplayService" interface="tns:Display">
         <endpoint name="Display" binding="tns:DisplaySoapBinding"
                   address="http://http://melas.labri.fr:8080/axis2/services/DisplayService"
                   />
       </service>
</description>
```

A.3 Service de détection de mouvement

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl2:description xmlns:wsdl2="http://www.w3.org/ns/wsdl"
```

```
                xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
                xmlns:ns0="http://www.inria.fr/phoenix/cameramotiondetectionservice"
                xmlns:ns1="http://org.apache.axis2/xsd"
                xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:whttp="http://www.w3.org/ns/wsdl/http"
                xmlns="http://www.w3.org/ns/wsdl"
                targetNamespace="http://www.inria.fr/phoenix/cameramotiondetectionservice"
                xmlns:Ontology0="http://www.inria.fr/phoenix/contexte#"
                xmlns:sawsdl="http://www.w3.org/ns/sawsdl">
    <wsdl2:types>
        <xs:schema xmlns:ns0="http://cameramotiondetection"
                xmlns:ns="http://www.inria.fr/phoenix/cameramotiondetectionservice"
                attributeFormDefault="qualified"
                elementFormDefault="qualified"
                targetNamespace="http://www.inria.fr/phoenix/cameramotiondetectionservice">
          <xs:element minOccurs="0" name="addEmailAddress"
                      type="xs:string" />
          <xs:element minOccurs="0"
                      name="removeEmailAddress" type="xs:string" />
        </xs:schema>
    </wsdl2:types>
    <wsdl2:interface name="MotionSensor"
                     sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
                         DetectionDeMouvement">
        <wsdl2:operation name="stopDetection"
                         pattern="http://www.w3.org/ns/wsdl/in-only"
                         sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
                             ArretDetection">
          <wsdl2:input element="#none"
                       wsaw:Action="urn:stopDetection"/>
        </wsdl2:operation>
        <wsdl2:operation name="startDetection"
                         pattern="http://www.w3.org/ns/wsdl/in-only"
                         sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
                             DebutDetection">
          <wsdl2:input element="#none"
                       wsaw:Action="urn:startDetection"/>
        </wsdl2:operation>
        <wsdl2:operation name="setMailDetection"
                         pattern="http://www.w3.org/ns/wsdl/in-only"
                         sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
                             DefinirDetectionEMail">
          <wsdl2:input element="#none"
                       wsaw:Action="urn:setMailDetection"/>
        </wsdl2:operation>
        <wsdl2:operation name="setNotificationDetection"
                         pattern="http://www.w3.org/ns/wsdl/in-only"
                         sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
                             DefinirDetectionNotification">
          <wsdl2:input element="#none"
                       wsaw:Action="urn:setNotificationDetection"/>
        </wsdl2:operation>
        <wsdl2:operation name="removeEmailAddress"
                         pattern="http://www.w3.org/ns/wsdl/in-only"
                         sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
                             SupprimerAdresseEmail">
          <wsdl2:input element="ns0:removeEmailAddress"
                       wsaw:Action="urn:removeEmailAddress"/>
        </wsdl2:operation>
        <wsdl2:operation name="addEmailAddress"
                         pattern="http://www.w3.org/ns/wsdl/in-only"
                          sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
                              AjouterAdresseEmail">
```

```
          <wsdl2:input element="ns0:addEmailAddress"
                       wsaw:Action="urn:addEmailAddress"/>
        </wsdl2:operation>
    </wsdl2:interface>
    <wsdl2:binding name="CameraMotionDetectionServiceSOAP12Binding"
                   interface="ns0:MotionSensor"
                   type="http://www.w3.org/ns/wsdl/soap"
                   wsoap:version="1.2">
      <wsdl2:operation ref="ns0:stopDetection"
                       wsoap:action="urn:stopDetection"/>
      <wsdl2:operation ref="ns0:startDetection"
                       wsoap:action="urn:startDetection"/>
      <wsdl2:operation ref="ns0:setMailDetection"
                       wsoap:action="urn:stopDetection"/>
      <wsdl2:operation ref="ns0:setNotificationDetection"
                       wsoap:action="urn:startDetection"/>
      <wsdl2:operation ref="ns0:removeEmailAddress"
                       wsoap:action="urn:removeEmailAddress"/>
      <wsdl2:operation ref="ns0:addEmailAddress"
                       wsoap:action="urn:addEmailAddress"/>
    </wsdl2:binding>
    <wsdl2:service name="CameraMotionDetectionService" interface="ns0:MotionSensor">
        <wsdl2:endpoint name="SOAP12Endpoint"
                        binding="ns0:CameraMotionDetectionServiceSOAP12Binding"
                        address="http://melas.labri.fr:8080/axis2/services/
                            CameraMotionDetectionService">
        </wsdl2:endpoint>
    </wsdl2:service>
</wsdl2:description>
```

A.4 Service d'agenda

```
<?xml version="1.0" encoding="UTF-8"?>
<description xmlns="http://www.w3.org/ns/wsdl"
             xmlns:impl="http://www.inria.fr/phoenix/calendar"
             targetNamespace="http://www.inria.fr/phoenix/calendar"
             xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
             xmlns:Ontology0="http://www.inria.fr/phoenix/contexte#">
    <types>
      <xsd:schema
          xmlns:apachesoap="http://xml.apache.org/xml-soap"
          xmlns:intf="http://www.inria.fr/phoenix/calendar"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="http://www.inria.fr/phoenix/calendar">
          <xsd:element name="getAllEvents">
             <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="user" type="xsd:string" />
                  <xsd:element name="password" type="xsd:string" />
                </xsd:sequence>
             </xsd:complexType>
          </xsd:element>
          <xsd:element name="getAllEventsResponse"
                       sawsdl:liftingSchemaMapping="http://melas.labri.fr:8080/XSLT/EventLifting
                           .xslt" />>
             <xsd:complexType>
                <xsd:sequence>
```

```
                <xsd:element maxOccurs="unbounded"
                             name="getAllEventsReturn"
                             type="impl:Event"/>
             </xsd:sequence>
          </xsd:complexType>
       </xsd:element>
       <xsd:complexType name="Event">
          <xsd:sequence>
            <xsd:element name="location" nillable="true"
                         type="xsd:string" />
            <xsd:element name="startTime" nillable="true"
                         type="xsd:string" />
            <xsd:element name="stopTime" nillable="true"
                         type="xsd:string"/>
            <xsd:element name="title" nillable="true"
                         type="xsd:string" />
            <xsd:element name="user" nillable="true"
                         type="xsd:string" />
          </xsd:sequence>
       </xsd:complexType>
       <xsd:element name="addEvent">
          <xsd:complexType>
             <xsd:sequence>
               <xsd:element name="user" type="xsd:string" />
               <xsd:element name="password" type="xsd:string" />
               <xsd:element name="minTime" type="xsd:string" />
               <xsd:element name="maxTime" type="xsd:string" />
               <xsd:element name="title" type="xsd:string"/>
               <xsd:element name="content" type="xsd:string"/>
               <xsd:element name="location" type="xsd:string" />
             </xsd:sequence>
          </xsd:complexType>
       </xsd:element>
    </xsd:schema>
  </types>
  <interface name="Calendar" sawsdl:modelReference="http://www.inria.fr/phoenix/contexte#
      Agenda">
    <operation name="getAllEvents"
               pattern="http://www.w3.org/ns/wsdl/in-out"
               sawsdl:modelReference="http://www.inria.fr/phoenix/context#
                   RecupereTousEvenements">
      <input element="impl:getAllEvents"/>
      <output element="impl:getAllEventsResponse"/>
    </operation>
    <operation name="addEvent"
               pattern="http://www.w3.org/ns/wsdl/in-only"
               sawsdl:modelReference="http://www.inria.fr/phoenix/context#AjouteEvenement">
      <input element="impl:addEvent"/>
    </operation>
  </interface>
  <binding xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
           name="CalendarSoapBinding" interface="impl:Calendar"
           type="http://www.w3.org/ns/wsdl/soap" wsoap:version="1.1"
           wsoap:protocol="http://www.w3.org/2006/01/soap11/bindings/HTTP/">
    <operation ref="impl:getAllEvents"/>
    <operation ref="impl:addEvent"/>
  </binding>
  <service name="CalendarService" interface="impl:Calendar" >
    <endpoint name="Calendar" binding="impl:CalendarSoapBinding"
              address="http://melas.labri.fr:8080/axis2/services/CalendarService"/>
  </service>
</description>
```

A.5 Schéma de traduction pour le service d'agenda

```
<?xml version="1.0"?>
<xsl:transform version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <xsl:template match="getAllEventsResponse">
    <rdf:RDF >
      <xsl:for-each select="getAllEventsReturn">
        <Evenement rdf:ID="{title}">
          <AUtilisateur rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            <xsl:value-of select="user" />
          </AUtilisateur>
          <ADebut rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            <xsl:value-of select="startTime" />
          </ADebut>
          <AFin rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            <xsl:value-of select="stopTime" />
          </AFin>
          <ALieu rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            <xsl:value-of select="location" />
          </ALieu>
        </Evenement>
      </xsl:for-each>
    </rdf:RDF>
  </xsl:template>
</xsl:transform>
```

B

Ontologie OWL

B.1 Ontologie des services

```
<?xml version="1.0"?>


<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
    <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY protege "http://protege.stanford.edu/plugins/owl/protege#" >
    <!ENTITY xsp "http://www.owl-ontologies.com/2005/08/07/xsp.owl#" >
]>


<rdf:RDF xmlns="http://www.inria.fr/phoenix/context#"
     xml:base="http://www.inria.fr/phoenix/contexte"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:swrl="http://www.w3.org/2003/11/swrl#"
     xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
     xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
     xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <owl:Ontology rdf:about=""/>
    <owl:FunctionalProperty rdf:about="http://www.inria.fr/phoenix/context#AdresseService">
        <rdf:type rdf:resource="&owl;DatatypeProperty"/>
        <rdfs:domain rdf:resource="http://www.inria.fr/phoenix/context#Service"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:FunctionalProperty>
    <owl:DatatypeProperty rdf:about="http://www.inria.fr/phoenix/context#AdresseTransformation"
        >
```

```
        <rdfs:domain rdf:resource="http://www.inria.fr/phoenix/context#Service"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#Agenda">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Service"/>
    </owl:Class>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#AjouteEvenement"/>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#AjouterAdresseEMail"/>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#ArretDetection"/>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#ConditionPhysique">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Contexte"/>
    </owl:Class>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#Contexte"/>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#DebutDetection"/>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#DefinirDetectionEMail"/>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#DefinirDetectionNotification"/>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#Deplacement"/>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#DetectionDeMouvement">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Service"/>
    </owl:Class>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#Inclinaison"/>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#LecteurMultimedia">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Service"/>
    </owl:Class>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#Lecture"/>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#Lieu">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Contexte"/>
    </owl:Class>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#MouvementCamera">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Service"/>
    </owl:Class>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#Operation"/>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#Personne">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Contexte"/>
    </owl:Class>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#P&#233;riph&#233;rique">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Contexte"/>
    </owl:Class>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#RecupereTousEvenements"/>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#Rotation"/>
    <owl:Class rdf:about="http://www.inria.fr/phoenix/context#Service">
        <rdfs:subClassOf rdf:resource="http://www.inria.fr/phoenix/context#Contexte"/>
    </owl:Class>
    <owl:ObjectProperty rdf:about="http://www.inria.fr/phoenix/context#SupporteOperation">
        <rdfs:domain rdf:resource="http://www.inria.fr/phoenix/context#DetectionDeMouvement"/>
    </owl:ObjectProperty>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#SupprimerAdresseEMail"/>
    <Operation rdf:about="http://www.inria.fr/phoenix/context#Zoom"/>
</rdf:RDF>
```