Table des matières

Liste des figures

Liste des tableaux

1 Introduction

Quand, au XV^{e} siècle, Gutenberg inventa l'imprimerie typographique moderne, il commença une révolution qui changea complètement la façon dont nous accédons à l'information. L'écrit commença à devenir de plus en plus présent dans la vie courante des personnes, jusqu'au point de devenir naturel, faisant partie de notre environnement.

L'arrivée de l'informatique a donné un nouvel élan à cette révolution. Depuis les années 1940, l'informatique a accéléré de plus en plus la diffusion de l'information et des connaissances. L'invention du transistor, la démocratisation de l'ordinateur personnel, la révolution du Web ou l'accès aux réseaux sociaux depuis des terminaux mobiles, sont des étapes qui ont de plus en plus rapproché les technologies de l'information et de la communication de notre vie de tous les jours.

Quelle serait l'étape suivante ? Weiser [Wei91] proposa une réponse au début des années 1990 : les ordinateurs doivent s'intégrer dans notre environnement et devenir omniprésents. Weiser appela son idée l'*informatique ubiquitaire* (*Ubiquitous Computing*). Le dictionnaire définit le terme ubiquitaire comme quelque chose « *qui est ou semble être partout à la fois.* » Dans le cas de l'informatique, cet adjectif traduit que les dispositifs informatiques, capables de nous assister dans nos tâches quotidiennes, vont continuer à nous entourer progressivement. Nous allons passer d'une interaction de type « bureau », *un-vers-un*, vers une autre de type ubiquitaire, *plusieurs-vers-plusieurs*. Au lieu de nous asseoir face à un ordinateur pour lui demander des actions, une multitude d'ordinateurs, de capteurs et d'actionneurs vont interagir autour et avec nous dans nos bâtiments, dans nos moyens de transport, dans nos rues, etc. Ces dispositifs vont devenir de plus en plus petits et nombreux, ce qui va changer radicalement notre façon de les utiliser. Pour désigner ces espaces riches en dispositifs et en interactions on parle par abus de langage d'*environnements ubiquitaires*.

L'informatique ubiquitaire est aussi appelée *informatique diffuse*, *informatique pervasive*, *intelligence ambiante*, *informatique omniprésente*, ou encore *l'internet des objets*. Cette dernière appellation souligne le fait que les machines ne communiqueront pas seulement avec les utilisateurs humains, mais elles pourront aussi le faire directement entre elles afin de rendre un service encore plus satisfaisant. L'idée de Weiser a suscité de nombreuses recherches autour des réseaux mobiles, de la miniaturisation des dispositifs informatiques, des nouvelles interfaces homme-machine, des intergiciels spécifiques, etc.

L'un des aspects les plus novateurs de l'informatique ubiquitaire est qu'elle essaye d'interagir avec l'utilisateur de la façon la plus naturelle

possible. Dans la communication entre personnes, beaucoup de messages sont transmis de façon non verbale. Par exemple, dans une conversation, nous nous aidons implicitement des gestes ou des relations qu'ont les personnes avec d'autres personnes ou avec les objets. Dans l'informatique classique, les ordinateurs et les personnes utilisent peu d'informations non explicites dans leurs communications. L'un des buts de l'informatique ubiquitaire est de profiter de ce type d'information implicite, que l'on appelle *contexte*, afin que les machines puissent apprendre les souhaits et les intentions des utilisateurs [TE01]. Cela permettrait d'anticiper leurs besoins et d'améliorer leur interaction avec l'environnement.

Dans ce nouveau panorama introduit par l'informatique ubiquitaire, il est intéressant d'étudier comment ces systèmes peuvent assister la collaboration entre groupes d'utilisateurs humains. Les systèmes qui soutiennent la collaboration, bien étudiés dans le cas de l'informatique classique, peuvent être adaptés au paradigme ubiquitaire afin de tirer profit, notamment, des informations implicites contenues dans le contexte. Par exemple, dans un hôpital doté d'informatique ubiquitaire, l'intelligence ambiante proposerait aux spécialistes qui travaillent avec des malades qui ont des maladies similaires de collaborer lorsqu'ils se croisent dans un couloir. Une fois que les médecins décident de collaborer, le système leur permettrait de partager leurs fichiers cliniques ou de rentrer dans un salon de discussion virtuel avec d'autres médecins ayant travaillé sur le même sujet. Les possibilités de ce type de systèmes sont très grandes, mais elles n'ont été explorées que partiellement dans l'état actuel de l'avancement des recherches.

1.1 Contributions apportées dans ce mémoire

Nos travaux de recherche se situent dans le cadre des systèmes collaboratifs ubiquitaires (SCUs), c'est-à-dire, des systèmes qui cherchent à fournir des moyens de collaboration dans les environnements ubiquitaires. Le but est d'améliorer la collaboration humaine à travers ces nouvelles technologies qui sont déjà en train de modifier nos habitudes. Pour cela, nous considérons qu'il est nécessaire de fournir des modèles spécifiques pour soutenir les applications qui mettent en œuvre une collaboration sensible au contexte dans les environnements ubiquitaires. Nous avons mis l'accent sur l'aspect de l'*adaptation* aux changements de contexte. En effet, dans ce type d'environnements hautement dynamiques, le contexte varie fréquemment. Notre objectif est de pouvoir adapter l'architecture qui supporte l'activité collaborative en fonction des changements qui peuvent survenir dans le contexte. Nous considérons également que cette adaptation doit tenir compte tant des exigences de l'application et de ses utilisateurs que des contraintes imposées par les plate-formes matérielles qui supportent les activités collaboratives. Afin de faire interopérer ces éléments, parfois antagonistes, nous croyons que la prise en compte de la sémantique dans les modèles est cruciale.

Nos principales contributions sont les suivantes :

- La proposition d'un modèle générique de collaboration, appelé *Generic Collaboration Ontology* (GCO), qui est basé sur des techniques dites *sémantiques* ou de *représentation des connaissances*. Ces tech-

niques, développées dans le domaine de l'intelligence artificielle, permettent d'exprimer des connaissances d'une façon exploitable par les ordinateurs et d'effectuer des inférences ou des déductions. Les instances du modèle permettent d'exprimer des situations de collaboration. Le traitement de ces instances permet de déduire un schéma de déploiement abstrait à partir de la situation de collaboration de haut niveau.

- La proposition d'une approche de modélisation architecturale multi-niveaux pour les applications collaboratives ubiquitaires. Cette approche manipule des modèles de l'architecture d'un SCU à plusieurs niveaux. Les informations contextuelles sont prises en compte selon ces niveaux, ce qui facilite l'interopérabilité entre les exigences fonctionnelles de haut niveau et les contraintes de bas niveau imposées par l'intergiciel ou le matériel. Deux fonctions permettent de faire la transition entre les niveaux : le *raffinement* et la *sélection*. Le raffinement transmet les exigences des niveaux hauts vers les niveaux bas, tandis que la sélection impose, à chaque niveau, des contraintes spécifiques. Afin d'instancier cette approche pour les SCUs, trois niveaux ont été retenus : le niveau *application*, le niveau *collaboration* et le niveau *intergiciel*. Nous avons identifié les éléments à représenter dans chaque niveau.
- L'implantation d'un *framework*, appelé *Framework for Adaptive Collaborative Ubiquitous Systems* (Facus), qui se base sur l'approche de modélisation proposée et qui peut être utilisé pour la conception et l'exécution de SCUs. Facus est un prototype qui utilise des modèles sémantiques (basés sur GCO) dans les niveaux application et collaboration et des modèles basés sur les graphes dans le niveau intergiciel. Les transformations de modèles sont basées sur des règles de transformation. Facus fournit un service de déploiement qui met en œuvre les architectures obtenues par les modèles. Ce déploiement est sensible au contexte, car les changements de contexte sont détectés et pris en compte au niveau correspondant, ce qui déclenche les changements nécessaires dans l'architecture de collaboration.
- L'implantation d'un exemple d'application de type SCU qui permet de montrer la faisabilité de l'approche proposée. Cette application a été également utilisée pour évaluer les fonctionnalités et les performances de Facus.

1.2 Organisation du mémoire

Ce mémoire est organisé de la manière suivante :

Le Chapitre 2 présente l'état de l'art. Il définit les thématiques auxquelles se rapportent nos travaux : l'informatique ubiquitaire, la collaboration, la sensibilité au contexte, l'adaptation et la sémantique. Ensuite, il situe nos deux principales contributions, GCO et Facus, par rapport aux travaux récents qui existent dans la littérature et qui traitent des problèmes similaires.

Le Chapitre 3 présente notre modèle central de la collaboration, GCO. En premier lieu, nous décrivons les éléments principaux du langage *Web Ontology Language* (OWL), avec lequel nous avons exprimé GCO. Ce lan-

gage, issu des recherches dans le champ de la représentation des connaissances et du Web Sémantique, permet de faire des représentations logiques qui sont calculables automatiquement. En deuxième lieu nous détaillons GCO, ainsi que les règles de traitement associées. Finalement, nous étudions les propriétés de ce modèle et nous expliquons comment l'utiliser comme modèle de collaboration dans un système, non seulement pour sa conception, mais aussi lors de son exécution.

Dans le Chapitre 4, nous présentons notre approche de modélisation multi-niveaux pour les architectures logicielles, qui facilite l'adaptation sensible au contexte et qui permet une interopérabilité entre les exigences et les contraintes. Cette approche, générique, est ensuite instanciée pour le cas des SCUs, pour lesquels nous avons retenu trois niveaux. Nous détaillons les modèles associés aux trois niveaux, ainsi que les techniques de transformation qui permettent d'obtenir le modèle d'un niveau à partir de celui du niveau supérieur. Nous présentons également le *framework* Facus, qui est une implantation en Java de l'approche proposée. Nous présentons son architecture et nous expliquons la façon dont sont mis en œuvre le déploiement de composants collaboratifs et l'acquisition de données du contexte. Finalement, nous donnons un aperçu de son fonctionnement, et notamment du processus d'adaptation.

Le Chapitre 5 présente un exemple d'application qui se base sur Facus pour supporter la collaboration entre ses utilisateurs dans des environnements ubiquitaires. Cette application, développée dans le contexte du projet européen UseNet[1], que nous présentons aussi dans ce chapitre, nous sert de test pour montrer et pour valider les fonctionnalités de Facus. Nous évaluons également les performances de Facus, plus concrètement le temps nécessaire pour qu'il adapte l'architecture collaborative aux changements de contexte.

Nous concluons en établissant le bilan de nos contributions et en présentant les perspectives de recherche de notre travail.

[1] *Ubiquitous M2M Service Networks*. Projet européen (Itea2) de trois ans sur la thématique de la communication et la gestion dans les environnements ubiquitaires.

État de l'art et synthèse

2

Sommaire

Dans ce chapitre, nous introduisons les domaines de recherche dans lesquels s'inscrit notre travail. En premier lieu, nous présentons l'informatique ubiquitaire, en se focalisant sur l'évolution qu'elle représente dans la façon dont nous utilisons les ordinateurs. Ensuite nous étudions les travaux existants dans le domaine du travail collaboratif, et nous regardons particulièrement les possibilités que l'informatique ubiquitaire peut apporter à ce domaine. Après cela, nous étudions deux domaines, le contexte et l'adaptation, qui sont intimement liés à l'informatique ubiquitaire et dont les systèmes collaboratifs peuvent profiter afin d'offrir des caractéristiques plus intéressantes pour les utilisateurs. Finalement,

nous nous intéressons à l'intelligence artificielle, et plus concrètement au champ de la représentation des connaissances, qui fournit les techniques que nous utilisons pour mettre en œuvre une interopérabilité sémantique dans la collaboration.

Après avoir étudié ces domaines liés à notre contribution, nous comparons celle-ci à d'autres travaux récents qui visent des buts similaires ou connexes aux nôtres. Cela nous servira à présenter un panorama des travaux qui traitent la collaboration dans les environnements ubiquitaires et à positionner notre contribution par rapport à l'existant.

2.1 Informatique ubiquitaire

Le concept d'*informatique ubiquitaire* a été introduit par Weiser en 1991 dans son article *The computer for the 21st Century* [Wei91], concept qu'il a raffiné par la suite [Wei93, Wei94, Wei99]. Dans ces articles, Weiser propose une révolution étonnante dans les technologies de l'information afin qu'elles développent complètement leur potentiel : les ordinateurs doivent devenir *invisibles*. La façon de rendre les ordinateurs invisibles consiste à les intégrer avec les objets et les endroits que nous utilisons dans la vie quotidienne. En effet, Weiser affirme :

« *Les technologies les plus profondes sont celles qui disparaissent. Elles se mêlent dans le tissu de la vie de tous les jours jusqu'au point où elles en deviennent indiscernables.* »

Weiser donne l'exemple de l'écriture : nous n'avons pas des textes seulement dans les livres ; presque tous les objets et endroits que nous côtoyons contiennent des mots écrits. La puissance de la « technologie de l'écriture » vient du fait qu'elle est parfaitement intégrée à notre vie et qu'elle devient omniprésente, à tel point que nous ne la remarquons plus.

Dans le cas des ordinateurs, Weiser propose une démarche équivalente : ils doivent s'intégrer dans notre environnement de telle façon que nous ne soyons plus conscients ni de leur présence ni de leur fonctionnement. Les personnes pourront ainsi se concentrer sur leurs tâches tout en restant inconsciemment assistées par les machines. Les ordinateurs deviendront une aide, un moyen de réaliser des buts de haut niveau, sans devenir un but eux-mêmes. La mobilité et la miniaturisation des ordinateurs ne suffisent pas pour les rendre « invisibles », et l'évolution ne se réduit pas à un problème d'interface utilisateur ; elle implique des changements dans la conception même que nous avons des ordinateurs et dans la façon dont nous interagissons avec eux.

Dans ses travaux initiaux, Weiser a proposé quelques prototypes de *dispositifs ubiquitaires* appelés *tabs*, *pads* et *boards*. Les *tabs* sont l'équivalent électronique des petites notes papier que nous utilisons pour nous rappeler des choses. Les *pads* sont les équivalents des cahiers, des feuilles ou des magazines. Les *boards* sont des tableaux électroniques sur lesquels plusieurs personnes peuvent travailler ensemble. On peut considérer ces dispositifs comme les précurseurs des téléphones intelligents et des tablettes électroniques si populaires aujourd'hui.

La vision révolutionnaire de Weiser a inspiré de nombreux travaux dans des domaines divers tels que les communications sans fil, les systèmes distribués, les architectures matérielles et logicielles, la mobilité, la domotique, etc. Bien que beaucoup de progrès aient été accomplis, de nombreuses études et expérimentations restent encore à faire pour poursuivre l'exploration de l'informatique ubiquitaire.

Satyanarayanan a enrichi la vision de Weiser en faisant une étude des possibilités offertes par l'informatique ubiquitaire et des défis de recherche de ce domaine [Sato1]. Dans ce travail, Satyanarayanan explique que l'informatique ubiquitaire se base sur les systèmes distribués et sur l'informatique mobile, mais qu'elle y ajoute des caractéristiques comme les espaces intelligents, l'invisibilité, le passage à l'échelle localisé et la prise en compte des hétérogénéités.

Les *espaces intelligents* mélangent deux mondes différents : les environnements physiques et les infrastructures de traitement de l'information. Dans les espaces intelligents, ces deux mondes peuvent s'influencer entre eux : par exemple, l'éclairage d'une pièce peut dépendre du profil électronique d'un utilisateur, et inversement un logiciel embarqué dans un dispositif personnel peut se comporter différemment en fonction de la situation géographique de l'appareil.

Satyanarayanan propose le concept de *distraction minimale* comme approximation de l'*invisibilité* introduite par Weiser. L'idée est d'anticiper les besoins de l'utilisateur afin de lui demander explicitement des informations le moins de fois possible. Cela permet aux utilisateurs d'interagir avec les systèmes ubiquitaires au niveau du subconscient ; ils y prêteront attention seulement quand ce sera nécessaire.

Le *passage à l'échelle localisé* est nécessaire car le nombre d'interactions dans les espaces intelligents fait que l'utilisation des ressources (par exemple l'énergie, la bande passante ou l'attention de l'utilisateur) devient critique. Les études existantes sur le passage à l'échelle ne considèrent pas la localisation des entités ; par exemple dans un serveur web, on essaie de servir le plus de clients possible indépendamment de leur situation géographique. Dans le cas des systèmes ubiquitaires, la plupart des interactions ont lieu entre des entités proches, ce qui peut être exploité pour améliorer le passage à l'échelle. En effet, quand un utilisateur s'éloigne d'un certain environnement ubiquitaire, on pourra réduire le nombre d'interactions qu'il a avec les éléments de cet environnement, car elles seront moins pertinentes.

Lors des premières étapes du déploiement de l'informatique ubiquitaire, les divers espaces intelligents restent très différents et cloisonnés entre eux en termes des possibilités qu'ils offrent. Par exemple, les réseaux sans fil seront disponibles dans certains de ces espaces, mais pas dans tous. La *prise en compte des hétérogénéités* est donc nécessaire, selon Satyanarayanan, afin de limiter l'impact que ces différences ont sur l'expérience des utilisateurs. Leurs dispositifs personnels devront s'adapter à ces hétérogénéités et les masquer autant que possible.

Saha et Mukherjee ont également approfondi la description de l'informatique ubiquitaire [SM03]. Selon leur opinion, les capteurs fournissent des informations sur le contexte des systèmes ubiquitaires, ce qui rend ces derniers différents des systèmes classiques. Ces données facilitent une interaction plus naturelle avec les utilisateurs.

Pour ces auteurs, l'informatique ubiquitaire suppose une étape supplémentaire dans l'évolution initiée avec l'informatique distribuée et poursuivie avec l'informatique mobile. Ils proposent également un modèle des environnements ubiquitaires qui inclut quatre domaines : les dispositifs, les réseaux, les intergiciels et les applications. En plus des innovations dans la mobilité et dans la capacité des dispositifs et des réseaux, l'évolution des logiciels, tant au niveau intergiciel qu'au niveau application, est nécessaire. L'intergiciel doit être capable d'interagir avec les réseaux environnants sans l'intervention de l'utilisateur afin de faciliter l'interconnexion des applications. Il doit aussi être capable de masquer l'hétérogénéité des réseaux disponibles. Les applications ubiquitaires mettront l'accent sur

l'environnement et l'information contextuelle plutôt que sur la mobilité ou sur le Web.

Dans leur travail, Saha et Mukerjee identifient quelques points clés de l'informatique ubiquitaire : en plus des concepts déjà mentionnés comme le passage à l'échelle, l'invisibilité et l'hétérogénéité, ils rajoutent l'intégration, la perception et l'intelligence.

L'*intégration* est importante car, bien que beaucoup d'éléments de l'informatique ubiquitaire existent déjà et sont déployés dans des environnements réels, il ne suffit pas de les mettre ensemble pour créer un environnement ubiquitaire. Plus ces éléments sont nombreux, et plus l'intégration devient complexe. Il faut prendre en compte des aspects tels que la qualité de service, la sécurité, l'invisibilité ou la fiabilité. Saha et Mukherjee signalent que l'informatique autonome (*autonomic computic*), introduite par Kephart et Chess [KC03] pour la gestion de systèmes complexes, est nécessaire pour faciliter cette intégration.

La *perception* (ou sensibilité au contexte) est nécessaire afin de fournir un service satisfaisant et proactif aux utilisateurs. Cependant, elle introduit beaucoup de complexité. Par exemple, il faut pouvoir gérer l'incertitude, fusionner des données fournies par des capteurs différents (et qui parfois seront contradictoires), traiter ces informations en temps réel, etc.

L'*intelligence* des systèmes dépendra de leur façon d'exploiter les données de contexte acquises. Ces données doivent être interprétées pour exécuter les actions adéquates. Des techniques de contrôle sont nécessaires pour adapter les actions du système aux informations perçues de l'environnement et de l'utilisateur.

2.2 Collaboration

Les travaux de recherche dans le domaine du *travail collaboratif assisté par ordinateur* (*Computer Supported Collaborative Work* ou CSCW en anglais) [CS99] ont démarré dans les années 1990. Kraemer [KK88] et Ellis [EGR91], respectivement, ont proposé deux définitions générales du travail collaboratif :

« *Système à base d'ordinateurs qui assiste des groupes de personnes réalisant en commun une tâche ou un but, et qui fournit une interface pour accéder à un environnement commun.* »

« *Système informatique qui facilite la résolution de problèmes par un ensemble de décideurs travaillant en groupe.* »

Dans ces définitions, le terme *travail* au sens large fait référence à toute tâche commune entre plusieurs personnes, dans des domaines de type jeu, enseignement, co-conception, etc. Les techniques développées dans le domaine du travail collaboratif peuvent être appliquées à toute sorte de collaboration humaine assistée par ordinateur.

Le travail collaboratif puise ses références dans quatre domaines [Vil06] :

- les *sciences sociales* (plus concrètement, la sociologie et la théorie des organisations) pour étudier l'organisation des groupes de personnes, leurs rapports, l'efficacité d'un groupe, etc. ;
- les *sciences cognitives* et l'*intelligence artificielle distribuée* pour étudier

la sémantique des informations, la planification de tâches, l'aide à l'exécution de ces tâches, etc. ;
- les *interfaces homme-machine* pour concevoir des interfaces multi-utilisateurs adaptées au travail collectif ;
- l'*informatique distribuée* pour la conception de systèmes répartis qui permettent le stockage, l'échange et le traitement d'informations à distance.

Le nom de *collecticiel* (*groupware* en anglais) désigne l'ensemble de produits logiciels, services, plate-formes et outils qui soutiennent le travail collaboratif [Kar94].

Dans la suite de cette section, nous nous focalisons sur la notion de *session* de travail, autour de laquelle se structure la plupart des systèmes collaboratifs.

2.2.1 Sessions

Le concept de *session* est essentiel dans le travail collaboratif. Une session est composée d'un ensemble d'utilisateurs qui partagent des intérêts communs [DGLA99]. Les personnes qui participent à une session ne doivent pas forcément se trouver au même endroit ; l'utilisation de réseaux informatiques permet l'intervention de participants éloignés géographiquement.

Les sessions peuvent être *synchrones* ou *asynchrones*. Dans une session synchrone, tous les participants sont présents simultanément. Les échanges entre ces participants sont interactifs, et les données sont manipulées en temps réel. Un exemple de ce type de session est un ensemble de personnes qui assistent à une réunion par visioconférence.

Dans une session asynchrone, la co-présence des membres du groupe n'est pas nécessaire (mais elle reste possible). Les échanges ne se produisent pas en temps réel, car ils sont basés sur des médias asynchrones tels que le courrier électronique.

Cette séparation entre sessions synchrones et asynchrones a été utilisée historiquement du fait que les technologies réseau associées étaient très différentes. Actuellement on peut trouver des outils qui combinent les deux modes : par exemple, dans l'édition collaborative d'un document, il peut y avoir des phases où les auteurs travaillent séparément de façon asynchrone, avec certaines « réunions » ponctuelles en mode synchrone pour garantir la cohérence du document produit.

Une deuxième classification des sessions est celle qui les sépare en *explicites* et *implicites*. Les sessions sont appelées explicites quand les configurations possibles de la session sont définies hors-ligne, lors de la conception du système. Le concepteur définit explicitement les relations entre les membres du groupe, ainsi que leur évolution dans le temps. Ensuite, lors de son exécution, le système gère des instances des sessions définies par le concepteur. En général, un utilisateur privilégié initie la session, et les autres utilisateurs peuvent la rejoindre s'ils sont invités. La plupart des modèles proposés pour la formalisation des sessions synchrones sont basés sur des graphes [RPVD01]. Dans ces graphes, les nœuds représentent les utilisateurs, tandis que les arêtes représentent des flux de données que

les utilisateurs s'échangent. Les étiquettes des arêtes indiquent l'outil qui gère l'envoi et la réception des données.

Les sessions implicites émergent de l'observation des actions des utilisateurs et de leur contexte. Quand le système détecte des situations de collaboration potentielle, en fonction par exemple de la présence des utilisateurs et de leurs intérêts, il crée une session implicite et il invite les utilisateurs concernés à la rejoindre. Peu de travaux ont étudié ce type de sessions en profondeur ; cependant on peut citer les travaux d'Edwards [Edw94] et de Texier et Plouzeau [TP03], qui proposent des modèles basés sur la théorie des ensembles, ainsi que celui de Rusinkiewicz et al. [RKT+95], qui s'appuie sur la logique de premier ordre pour la description de sessions dont la structure n'est pas fixée à priori.

2.2.2 Gestion des sessions et déploiement

Dans les outils collaboratifs, les modèles de session sont utilisés par des gestionnaires de session qui contrôlent le cycle de vie des sessions. Ils servent à définir les sessions, à les activer, à contrôler l'accès des utilisateurs et leurs droits, à activer les outils nécessaires, etc.

Un aspect important est le déploiement des outils et des composants qui gèrent les flux de données envoyés entre les utilisateurs. En effet, il est nécessaire d'installer et de configurer ces éléments sur les machines des utilisateurs afin qu'ils puissent effectuer ces échanges. Par exemple, si le modèle de session indique qu'un flux audio existe entre l'utilisateur A et l'utilisateur B, alors il est nécessaire que A et B disposent sur leurs machines d'un outil (ou d'un composant faisant partie d'un outil) capable de gérer des flux audio.

Hammami [Ham07] a fait une étude très complète des types de déploiement et des systèmes de déploiement qui existent. Le déploiement peut être statique (où un administrateur désigne explicitement l'application à utiliser) ou dynamique (où le choix se fait d'une façon automatique pendant le processus de déploiement), centralisé (avec une entité principale qui dirige le processus) ou décentralisé (où les nœuds de déploiement interagissent entre eux). Il existe deux stratégies de déploiement : *push*, dans laquelle l'initiative du déploiement est donnée à l'administrateur ou au superviseur, et *pull*, dans laquelle les nœuds de déploiement initialisent le processus eux-mêmes. En général, les systèmes que l'on trouve dans la littérature implantent un déploiement de type statique, en mode *push*, et souvent centralisé. Les systèmes automatiques ont été peu explorés, et en général ils sont peu flexibles. Ils ne permettent pas, par exemple, le déploiement de composants *à chaud*, c'est-à-dire, sans demander un arrêt des logiciels qui s'en servent.

2.2.3 Collaboration et informatique ubiquitaire

Les activités collaboratives peuvent particulièrement bénéficier des possibilités offertes par l'informatique ubiquitaire. Notamment, le fait de disposer d'informations sur le contexte des utilisateurs est très intéressant pour les applications collaboratives [HL09]. De ce fait, la notion de session implicite prend tout son sens quand on la considère dans le cadre de

l'informatique ubiquitaire, car l'immersion des utilisateurs dans l'environnement et la sensibilité au contexte donnent beaucoup d'informations qui peuvent être exploitées afin de proposer des sessions adaptées aux besoins des utilisateurs. Comme Iqbal et al. [IIS+08] l'ont signalé, l'informatique ubiquitaire peut à la fois améliorer les relations des humains avec l'environnement et avec d'autres humains. Pour cela, il faut avoir une connaissance, représentée par des modèles, des tâches que les humains réalisent et de leurs intentions.

Ces dernières années, les techniques et les modèles *classiques* du travail collaboratif que nous avons cités, comme par exemple ceux basés sur les graphes ou sur la théorie des ensembles, ont été utilisés dans les collecticiels de type « bureau » d'une façon satisfaisante. Néanmoins, dans le cadre de l'informatique ubiquitaire, les besoins sont différents, et les approches existantes ne sont pas complètement adaptées. Il est nécessaire de proposer de nouveaux cadres conceptuels et des modèles qui prennent en considération les aspects spécifiques de la collaboration dans ce type d'environnements.

Le principal inconvénient des systèmes collaboratifs classiques pour leur utilisation dans les environnements ubiquitaires est que la conception des sessions est assez rigide [SBVT10]. La plupart des travaux considèrent des sessions explicites. Les travaux qui prennent en compte les sessions implicites proposent une implantation limitée ; elles sont vues comme un problème d'informatique distribuée (comment faire que plusieurs utilisateurs accèdent de façon concurrente aux mêmes ressources) et non comme un problème d'informatique ubiquitaire (*comment proposer spontanément des sessions en fonction des informations contextuelles*). Les sessions sont définies lors de la conception des logiciels et il n'est pas possible de les adapter en fonction du contexte de l'application et des utilisateurs.

Nous pouvons ajouter à cela le fait que les services de déploiement fournis dans les systèmes collaboratifs classiques ne sont pas assez flexibles. En effet, les outils et les composants sont souvent déployés au préalable et manuellement. Les systèmes de déploiement automatique se réduisent souvent à des systèmes de téléchargement et d'ajout de paquetages qui ne peuvent pas être installés ni démarrés à chaud. Ces méthodes sont suffisantes pour les systèmes de type « bureau », mais elles ne sont pas envisageables dans un environnement ubiquitaire dans lequel on ne veut pas focaliser l'attention de l'utilisateur sur l'installation de nouveaux logiciels à chaque fois qu'il veut rejoindre une activité collaborative.

2.3 La notion de contexte

Comme nous l'avons signalé dans les sections précédentes, le concept de contexte joue un rôle clé dans les systèmes ubiquitaires. Plusieurs auteurs ont essayé de cerner ce concept au moyen de définitions et de classifications, dont nous citons les plus importantes dans cette section. Nous nous intéressons également à la sensibilité au contexte, pour finalement présenter quelques solutions pour sa prise en compte dans les applications.

2.3.1 Définition

Le contexte étant une notion complexe et abstraite, il est difficile de trouver une définition détaillée. Les définitions générales sont les plus nombreuses ; nous listons ici les principales [Cha07].

Parmi les premiers à essayer de définir le contexte, se trouvent Schilit et Theimer, pour lesquels le contexte est constitué de la localisation de l'utilisateur, ainsi que des identités et des états des personnes et des objets qui l'entourent [ST94]. Brown et al. [BBC97] ajoutent à cette définition des données telles que l'identité de l'utilisateur, son orientation ou la température. Ryan et al. [RPM98] ajoutent la notion de temps.

Pascoe [Pas98] introduit un élément important : l'intérêt. En effet, il définit le contexte comme un sous-ensemble d'états physiques et conceptuels qui ont un certain intérêt pour une entité donnée. Cette notion d'intérêt ou de pertinence est reprise par Abowd, Dey et al. [ADB+99] dans leur définition, qui est communément acceptée :

« *Le contexte couvre toutes les informations qui peuvent être utilisées pour caractériser la situation d'une entité. Une entité est une personne, un endroit ou un objet que l'on considère pertinent par rapport à l'interaction entre un utilisateur et une application, y compris l'utilisateur et l'application eux-mêmes.* »

Winograd [Win01] reprend cette définition pour la détailler, car il considère que, malgré le fait qu'elle couvre tous les travaux existants, elle est trop générale : tout élément peut être considéré comme faisant partie du contexte. En premier lieu, il précise que le contexte est un ensemble d'informations. Cet ensemble est structuré et partagé, et il peut évoluer dans le temps. En deuxième lieu, selon lui, l'appartenance d'une information au contexte ne dépend pas de ses propriétés inhérentes, mais de la manière dont elle est utilisée. Une information fait partie du contexte seulement si le système dépend d'elle, d'une façon ou d'une autre.

2.3.2 Taxonomie des données contextuelles

Dans les travaux existants, nous avons trouvé plusieurs taxonomies du contexte, qui classifient les différents types de données contextuelles. Ces classifications permettent de lister les paramètres que l'on peut prendre en compte dans un système sensible au contexte.

La première catégorisation est celle de Schilit [SAW94], qui propose de répondre aux questions *où l'on se trouve*, *avec qui* et *quelles sont les ressources à proximité* afin de caractériser le contexte d'une entité. Ryan et al. [RPM98] proposent les catégories de *position*, *identité*, *environnement* et *temps*. La classification la plus acceptée est celle proposée par de Abowd, Dey et al. [ADB+99], qui est constituée de deux niveaux, avec des paramètres primaires et secondaires. Les paramètres primaires sont les mêmes que ceux de Ryan et al., sauf que la notion d'*activité* (c'est-à-dire, ce qu'on est en train de faire) remplace celle d'*environnement*, trop générale. Les paramètres secondaires sont des attributs des entités qui se trouvent dans le contexte primaire. Ces paramètres peuvent être retrouvés en utilisant les données primaires pour les indexer. Par exemple, le numéro de téléphone d'une personne peut être retrouvé en utilisant l'identité de la personne comme index dans une base de données qui contient des numéros de téléphone.

Chen et Kotz [CKoo] citent une taxonomie alternative qui divise le contexte en trois catégories principales : le *contexte informatique*, le *contexte utilisateur* et le *contexte physique*. Le contexte informatique contient toutes les informations relatives aux ressources matérielles qui permettent l'exécution de l'application. Le contexte utilisateur contient des informations relatives à l'identité de l'utilisateur, ses préférences, ses relations sociales, etc. Finalement, les données du contexte physique caractérisent l'environnement physique dans lequel se trouvent les personnes et les applications. Cette dernière catégorie regroupe également les données temporelles. Le Tableau 2.1 présente quelques exemples de paramètres de contexte de ces trois catégories.

Type d'information	**Catégorie**	**Exemples**
Identité	utilisateur	nom, prénom
Spatiale	physique	position, orientation, vitesse
Temporelle	physique	date, heure, saison
Environnementale	physique	température, luminosité, bruit
Sociale	utilisateur	personnes aux alentours, activités, calendrier
Ressources d'une machine	informatique	CPU, RAM, batterie
Ressources de communication	informatique	bande passante, nombre de connexions TCP
Physiologique	utilisateur	pression artérielle, pouls

TAB. 2.1 – *Exemples de paramètres de contexte*

Dans notre travail, nous avons utilisé une version simplifiée de cette dernière classification ; nous divisons le contexte en *contexte externe* et *contexte des ressources*. Le contexte externe regroupe le contexte utilisateur et le contexte physique de la classification de Chen et Kontz, tandis que le contexte des ressources est l'équivalent du contexte informatique. Nous avons fait cette division car nous considérons que l'influence et la façon de gérer les données pour ces deux catégories sont très différentes du point de vue des logiciels sensibles au contexte. En effet, le contexte des ressources détermine « ce qu'il est possible de faire » pour une application donnée. Par exemple, on ne peut pas déployer quatre composants qui ont besoin de 100 Mo de mémoire chacun sur une machine qui dispose seulement de 300 Mo de mémoire. De son côté, le contexte externe détermine « ce qu'il est souhaitable de faire » par rapport aux attentes de l'utilisateur et à son environnement. Par exemple, si un utilisateur entre dans une pièce la nuit, on peut considérer qu'il voudra que la lumière soit allumée, tandis que s'il fait jour, cela ne sera pas souhaitable. En définitive, le contexte des ressources représente des *contraintes* qui sont imposées au système, tandis que le contexte externe représente les *exigences* ou les attentes des utilisateurs. Le contexte externe à prendre en compte est fortement dépendant de l'application, tandis que le contexte des ressources est commun à toutes les applications.

2.3.3 Sensibilité au contexte

Comme nous l'avons vu, l'idée de sensibilité au contexte émerge naturellement du concept d'informatique ubiquitaire. En effet, le fait de disposer d'informations contextuelles sert à adapter l'application au contexte perçu afin de mieux répondre aux attentes des utilisateurs. L'idée principale de l'informatique sensible au contexte est de sortir le plus possible les humains de la boucle de contrôle des machines, en réduisant les interactions entre l'humain et la machine.

La notion de sensibilité au contexte (*context-awareness* en anglais) a donc été produite dans le cadre des recherches sur l'informatique ubiquitaire. Schilit et Theimer [ST94] furent les premiers à proposer une définition de la sensibilité au contexte : il s'agit de la capacité d'un système à découvrir et à réagir à des changements dans l'environnement où il se trouve. Ils signalent également l'importance de l'adaptation du système à ces changements.

Pour Abowd, Dey et al. [ADB+99], un système est sensible au contexte s'il utilise celui-ci pour fournir à l'utilisateur des informations et des services pertinents. Cette pertinence dépend de l'activité que l'utilisateur est en train de réaliser. Ces auteurs proposent également une classification des systèmes sensibles au contexte selon leur réponse aux changements de contexte. Cette réponse peut soit présenter des informations ou des services à l'utilisateur, soit exécuter automatiquement un service, soit stocker des données contextuelles.

2.3.4 Solutions pour la prise en compte du contexte

Comme l'a signalé Edwards [Edw05], la signification et l'utilisation du contexte changent très souvent ; c'est une notion fluide et peut-être même ambigüe. De ce fait, il est difficile de créer une infrastructure pour la prise en compte du contexte utile pour toutes les applications. Malgré cela, on peut trouver dans la littérature plusieurs travaux qui ont proposé des plate-formes pour la gestion du contexte. Bien que nous soyons encore loin d'une plate-forme unifiée et automatique pour la gestion du contexte, ces travaux ont fait des progrès remarquables. Nous citons ici les exemples les plus importants par rapport à notre étude.

Dey et al. [DAS01] ont proposé la boite à outils Java *Context Toolkit*, qui fournit notamment une architecture d'acquisition de données contextuelles basée sur des *widgets*. Les *widgets* sont des composants logiciels qui encapsulent les capteurs physiques ou logiciels qui produisent les données de contexte. *Context Toolkit* fournit des objets qui facilitent les communications réseau entre les *widgets* et les serveurs ou les applications qui veulent surveiller les données de contexte que les *widgets* produisent. Il fournit aussi des moyens pour stocker les données capturées. Dans *Context Toolkit*, les données de contexte se présentent sous la forme clé–valeur. Il est possible d'utiliser des unités pour les mesures, mais leur sémantique est à définir par l'application. L'avantage de cette boite à outils est sa simplicité, car tout en restant assez complète, la mise en œuvre de l'architecture d'acquisition de contexte dans une application requiert très peu d'effort.

Chen [Che04] propose l'architecture *Context Broker Architecture* (CoBrA). Cette architecture utilise des agents pour la prise en compte

du contexte dans des espaces intelligents. Dans chaque espace, on trouve un courtier central (*broker* en anglais), qui gère un modèle central du contexte qu'il construit avec les données acquises. Il partage ce modèle avec les dispositifs, les services et les agents qui sont dans le même espace. Le courtier peut aussi fournir des interprétations des données de contexte qu'il détient grâce à l'utilisation d'ontologies OWL [SWM04]. Le courtier fournit également des moyens pour garantir la confidentialité des données de contexte.

Le système *Service-Oriented Context-Aware Middleware* (SOCAM), proposé par Gu et al. [GPZ04], est un intergiciel distribué qui permet l'implantation rapide de services mobiles sensibles au contexte. Comme dans le cas de CoBrA, cet intergiciel utilise des ontologies OWL pour représenter et pour effectuer des raisonnements sur le contexte. Dans l'architecture de SOCAM, on trouve des composants tels que les fournisseurs de contexte, qui capturent les données de contexte et les représentent en OWL, les interpréteurs de contexte, qui fournissent des services de raisonnement sur le contexte, les bases de données de contexte, qui stockent les données de contexte, les services sensibles au contexte, qui adaptent leur fonctionnement au contexte courant, et les services de découverte de services, qui permettent l'annonce et la découverte des fournisseurs et des interpréteurs de contexte.

2.4 Adaptation

D'une façon générale, l'adaptabilité est la capacité d'un système à s'adapter à des changements. En informatique, les systèmes qui présentent cette propriété, dits adaptatifs (parfois auto-adaptatifs), ont connu un regain d'attention ces dernières années. Selon Laddaga [Lad01], la principale raison pour laquelle on a besoin de logiciels adaptatifs est le coût de la gestion de la complexité. En effet, les logiciels actuels sont de plus en plus complexes car leurs objectifs sont de plus en plus ambitieux. Il est nécessaire de gérer cette complexité, surtout quand les objectifs des logiciels évoluent dans le temps. Les logiciels adaptatifs sont une réponse à ce problème.

Satyanarayanan [Sat01] justifie le besoin d'adaptabilité comme réponse au compromis entre l'offre et la demande des ressources :

« *L'adaptabilité est nécessaire quand il y a des disparités significatives entre l'approvisionnement et la demande d'une ressource.* »

2.4.1 Définition

Plusieurs auteurs ont essayé de définir l'adaptation. Nous citons ici les définitions les plus répandues [ST09].

En 1997, Laddaga [Lad97] proposa une première définition :

« *Un logiciel auto-adaptatif évalue son propre comportement et change ce comportement quand l'évaluation indique qu'il n'est pas en train de réaliser ce qu'il est censé réaliser, ou que de meilleures fonctionnalités ou performances sont possibles.* »

Peu après, Oreizy et al. [OGT+99] ont donné une définition plus complète. Cette définition ajoute la notion d'environnement, que nous pouvons assimiler au contexte :

« *Un logiciel auto-adaptatif modifie son propre comportement en réponse à des changements dans son environnement de fonctionnement. Par environnement de fonctionnement, nous nous référons à tout ce qui est observable par le système logiciel, comme les données fournies par l'utilisateur, les matériels et les capteurs externes ou des mesures.* »

Ces premières définitions ne mentionnent pas explicitement l'informatique ubiquitaire. McKinley et al. [MSKC04] ont signalé que l'informatique ubiquitaire introduit un besoin d'adaptation en raison de la mobilité des utilisateurs et des interactions dynamiques avec l'environnement.

Raibulet [Rai08] identifie trois concepts importants des systèmes adaptatifs. En premier lieu, l'adaptabilité est demandée comme réponse à des changements qui arrivent soit à l'intérieur du système, soit dans son environnement. Dans ce deuxième cas, on dira que le système est *sensible au contexte*. En deuxième lieu, l'adaptabilité est atteinte avec des changements que le système effectue *sur lui-même*. Finalement, pour considérer qu'un système est pleinement adaptatif, ces changements doivent s'effectuer *lors de l'exécution* du système.

Récemment, Salehie et Tahvildari [ST09] ont proposé une vue d'ensemble très complète du domaine de l'adaptation, avec une énumération des défis à affronter dans ce domaine. Notamment, ils identifient la *conception pour l'adaptabilité* : il faut proposer des approches, des paradigmes, des architectures et des modèles appropriés pour la prise en compte de l'adaptabilité dans les systèmes depuis la phase de conception. Ces auteurs ont également signalé que le point clé dans les systèmes adaptatifs est que leur cycle de vie ne doit pas s'arrêter après leur développement et leur configuration initiale ; ce cycle doit continuer durant l'exécution du système, en répondant aux changements de l'environnement à tout moment.

2.4.2 Types d'adaptation

Dans leur travail, Salehie et Tahvildari [ST09] proposent une taxonomie de l'adaptation qui essaie de classifier les propriétés de l'adaptation. Notamment, on y trouve des éléments qui répondent à la question de *comment* réaliser l'adaptation. Parmi ces éléments, les considérations suivantes nous semblent importantes :

- Prise de décisions *statique* versus *dynamique*. Dans le mode statique, les réponses aux changements sont codées en dur dans le système. Dans le mode dynamique, ces décisions sont implantées par des politiques ou des règles externes qui peuvent être changées lors de l'exécution du système.
- Adaptation *interne* versus *externe*. Dans l'adaptation interne, la logique d'adaptation est entremêlée avec la logique applicative. Dans l'adaptation externe, il existe un composant ou module, appelé *gestionnaire d'adaptation*, qui surveille le système et commande les changements.
- Adaptation *basée sur les modèles* versus adaptation *sans modèles*. Dans le second cas, le mécanisme d'adaptation ne dispose pas d'un mo-

dèle spécifique de l'application ni de son contexte ; le mécanisme contient toutes les informations nécessaires implicitement dans son code interne. Dans l'adaptation basée sur les modèles, le mécanisme d'adaptation s'appuie sur des modèles explicites de l'application et du contexte qui permettent de prendre en compte toutes les informations afin de réaliser l'adaptation.

– Adaptation *réactive* versus *proactive*. L'adaptation réactive agit une fois que les changements auxquels elle répond se sont produits. L'adaptation proactive intervient quand le système prédit que le changement va se produire.

Dans les systèmes qui présentent des contenus aux utilisateurs, deux types d'adaptation sont possibles : l'*adaptation du contenu* et l'*adaptation de la présentation*. Dans le premier cas, le système change les contenus à présenter en fonction de l'utilisateur. Par exemple, un système vidéo peut afficher des dessins animés si l'utilisateur est un enfant, tandis que, pour les adultes, il affichera des actualités économiques. L'adaptation de la présentation, en revanche, change la façon de présenter un même contenu. Par exemple, un système qui présente des informations en mode texte peut basculer vers l'audio lorsqu'une personne malvoyante l'utilise.

Selon la façon de mettre en œuvre l'adaptation d'un système, nous pouvons trouver dans la littérature deux types principaux d'adaptation : l'adaptation comportementale et l'adaptation architecturale.

L'*adaptation comportementale* consiste à changer le comportement d'un système lors son exécution [GLT08]. L'intérieur des composants est modifié, mais le nombre de composants ou la façon dont ils sont connectés restent les mêmes. Le composant adapte son fonctionnement interne en fonction des changements.

L'*adaptation architecturale* consiste à changer la structure d'un système lors de son exécution. Le nombre et la nature des composants présents, ainsi que la façon dont ils sont connectés, changent [GCH+04].

Dans la suite, nous nous focalisons sur l'adaptation architecturale, car elle nous semble la plus pertinente pour les systèmes collaboratifs ubiquitaires. En effet, les activités collaboratives sont intimement liées à l'architecture qui les soutient. Lorsque la structure d'un groupe change, la façon dont sont reliés les composants collaboratifs peut être amenée à changer.

Salehie et Tahvildari [ST09] ont signalé que l'adaptation architecturale agit souvent sur des systèmes structurés en couches. L'adaptation peut modifier des éléments dans une ou plusieurs couches de l'architecture. Dans ce sens, Satyanarayanan [Sat04] a ajouté qu'il est difficile de concilier les systèmes à couches avec l'adaptation. Pour lui, les architectures en couches sont nécessaires, spécialement dans le cas de l'informatique ubiquitaire, pour rendre la conception des systèmes propre et modulaire. En revanche, cette décomposition en couches peut complexifier l'adaptation architecturale, car les actions d'adaptation doivent gérer des objets qui sont dispersés sur plusieurs niveaux de l'architecture. Il considère que cette conciliation entre les logiciels en couches et l'adaptation architecturale est un des défis importants de l'informatique ubiquitaire.

Dans le cas des systèmes collaboratifs ubiquitaires, l'objet de l'adaptation architecturale est l'architecture de collaboration mise en place. Cette architecture est distribuée sur les différents dispositifs des utilisateurs et

de l'environnement qui interviennent dans la collaboration. Lors d'une adaptation, on modifiera les composants collaboratifs déployés sur les dispositifs, ainsi que les différents flux que ces composants s'envoient entre eux.

Par exemple, si un utilisateur A arrive dans un espace intelligent et rejoint une session dans laquelle il doit recevoir des données audio de l'utilisateur B, il faut que le système déploie un composant récepteur audio sur le dispositif personnel de A, puis un composant d'envoi audio sur le dispositif de B. Ensuite, il faut établir le flux audio qui va de B vers A.

2.5 Intelligence artificielle et sémantique

Une partie importante de nos travaux se base sur des techniques développées dans le domaine de l'*intelligence artificielle*, et notamment de la représentation des connaissances. Dans cette section nous présentons ce domaine, ainsi que le courant du Web Sémantique, qui a amélioré dans les dernières années la façon de prendre en compte la sémantique des données et des services dans le Web.

2.5.1 Petit historique de l'intelligence artificielle

Depuis l'antiquité, l'homme a rêvé de construire des machines artificielles capables de *penser* comme les hommes. On trouve de telles machines, par exemple, dans les mythes grecs ou dans ceux de l'ancienne Égypte. Plus récemment, des histoires de fiction comme *Frankenstein* de Mary Shelley ont utilisé l'idée de personnages artificiels dotés d'intelligence [Maz95]. Cette idée a été également abordée par les philosophes. En partant de l'hypothèse qu'elle peut être reproduite d'une façon mécanique, les philosophes ont essayé de reformuler la pensée humaine afin de la rendre formelle, comme les mathématiques. Aristote, Leibniz, Descartes ou Russell sont quelques uns parmi les grands philosophes qui ont travaillé sur ce sujet [RN03].

En informatique, dans la décennie de 1950, les chercheurs ont commencé à envisager la possibilité de créer la pensée artificielle à l'aide des ordinateurs. Le domaine de l'intelligence artificielle (IA) naquit officiellement en 1956 lors d'une conférence au Dartmouth College aux États Unis [RN03]. Parmi les participants à cette conférence se trouvaient ceux qu'on considère les pères de l'IA, comme McCarthy, Minsky, Newell ou Nilsson. Dans les années suivantes, ils travaillèrent dans le but de construire des machines et des algorithmes capables de fonctionner d'une façon autonome et de réfléchir. En plus de l'informatique, l'IA a reçu des contributions d'autres domaines très divers, tels que la psychologie cognitive, les mathématiques, la neurobiologie ou la philosophie.

Les premières années de recherche virent arriver beaucoup de succès, et les chercheurs étaient très optimistes. Par exemple, Simon déclara en 1965 : « *Dans une vingtaine d'années, les machines seront capables de réaliser n'importe quel travail que les humains peuvent réaliser.* » Cet optimisme s'effaça en grande partie dans les années 1970, quand les chercheurs commencèrent à se rendre compte de la difficulté intrinsèque de beaucoup des thèmes abordés [RN03]. Les ambitions du domaine ont été revues à

la baisse. Ainsi, l'idée de créer des machines intelligentes a été peu à peu abandonnée, pour se focaliser sur des machines dotées de programmes figés, dont l'exécution produit des résultats qui se rapprochent de ceux produits par des comportements intelligents.

Quelques années plus tard, dans les années 1980, l'IA regagna de l'attention avec notamment les progrès dans le champ des *systèmes experts* [Ign91] qui soulignèrent l'intérêt de l'étude de la *connaissance*.

Grâce à l'augmentation de la puissance de calcul, ce domaine a connu un nouvel essor dans les années 2000 avec l'émergence du Web, les moteurs de recherche, les agents, les technologies de reconnaissance de la voix, la robotique ou l'exploration de données (*data mining*), entre autres.

2.5.2 Champs d'étude de l'IA

L'IA se décompose en plusieurs sous-domaines qui ont donné naissance à des champs de recherche indépendants. Les principaux sous-domaines sont [RN03] :

- *Représentation des connaissances*. Ce champ se focalise sur la façon de représenter des connaissances dans les systèmes informatiques pour permettre notamment l'inférence, c'est-à-dire la déduction de nouvelles connaissances à partir de connaissances existantes.
- *Traitement du langage naturel*. Ce champ cherche à trouver des moyens automatiques pour que les machines puissent traiter le langage naturel humain (français, anglais, etc.), tant écrit que parlé. Les deux principaux problèmes dans ce champ sont la génération automatique de langage naturel et sa compréhension.
- *Perception*. Ce champ offre la capacité d'apprendre des faits du monde à partir de données acquises par des capteurs tels que des caméscopes, des microphones, etc. On y trouve notamment les études sur la vision artificielle.
- *Apprentissage*. Le but dans ce champ est de créer des techniques qui permettent à un système informatique d'*apprendre*. Cela signifie que le système est capable de généraliser des connaissances de façon inductive à partir d'exemples non structurés.
- *Planification*. Le but de ce champ est de créer des techniques permettant aux machines de résoudre des problèmes en générant des plans. Un plan est une stratégie pour la résolution du problème, et consiste en plusieurs actions à réaliser dans un ordre spécifique. Ces plans permettent de résoudre des problèmes complexes, comme par exemple la décision de la trajectoire optimale d'un robot pour se déplacer d'un point à un autre dans une pièce.

2.5.3 Représentation des connaissances

Historiquement, la représentation des connaissances [BL04] a été une des branches centrales de l'IA. Le but de ce champ est de représenter les connaissances de façon à faciliter leur traitement automatisé par une machine. En d'autres termes, les systèmes de représentation des connaissances formalisent la pensée et le savoir humains.

Les *connaissances* sont des faits ou des prédicats, qui peuvent être vrais

ou faux. Par exemple, dans la phrase « *Jean sait que la voiture de Claire est bleue* », on énonce que Jean possède une certaine connaissance, qui est contenue dans la proposition « *la voiture de Claire est bleue* ». Si, dans le monde réel, la voiture de Claire est vraiment bleue, cette proposition est vraie ; si elle est, par exemple, rouge, alors la proposition est fausse. La connaissance implique une certaine façon de concevoir le monde. Dans l'exemple précédent, Jean considère que la voiture de Claire est bleue et non pas d'une autre couleur.

Le terme *représentation* met en relation deux domaines, où l'un d'entre eux remplace, ou représente, l'autre [BL04]. On utilise un domaine à la place de l'autre car ce premier domaine est plus concret, plus immédiat ou plus accessible que le deuxième. Par exemple, quand on écrit le mot *chien*, on utilise le domaine des lettres, dont font partie la lettre *C*, la lettre *H*, etc., pour représenter le domaine des animaux. En conséquence, la représentation des connaissances étudie l'utilisation de symboles formels afin de représenter des propositions.

Un concept important lié à la représentation des connaissances est celui de *raisonnement*. Le raisonnement ou inférence est la capacité de traiter les connaissances que l'on possède afin de produire des nouvelles connaissances. Par exemple, si l'on sait que « *Claire est la femme de Jean* » et que « *la voiture de Claire est bleue* », alors on peut inférer que « *La voiture de la femme de Jean est bleue* ». Ce fait n'était pas parmi l'ensemble initial de faits connus, du moins explicitement. Le raisonnement a rendu explicite cette connaissance qui était implicite.

Ainsi que le signalent Brachman et Levesque [BL04], le raisonnement est intimement lié à la représentation des connaissances. Pour eux, le point le plus important est justement l'interaction entre la représentation des connaissances et le raisonnement. En effet, la capacité à effectuer des raisonnements dépend directement de la façon dont on représente les connaissances. Brachman et Levesque [LB87] ont souligné le compromis qu'il faut faire entre l'expressivité d'un langage de représentation et la complexité du raisonnement avec ce langage. Plus les constructions d'un langage sont complexes, et plus le raisonnement sera complexe en termes de calcul. Ceci a des fortes implications pour l'implantation d'algorithmes qui effectuent le raisonnement. Une autre propriété importante des langages est la décidabilité. Un langage est dit décidable s'il existe un algorithme qui peut décider en un nombre fini de pas si, en partant d'un ensemble de connaissances, un fait est vérifié ou non.

Ainsi, une grande partie des efforts fournis dans ce champ ont été orientés vers la recherche de systèmes qui représentent les connaissances d'une façon aussi compréhensible que possible et avec lesquels on puisse réaliser des raisonnements aussi efficacement que possible.

Parmi les premiers systèmes de représentation des connaissances, les plus importants furent les systèmes de cadres (*frame systems*) [Min75] et les réseaux sémantiques [Qui67]. Ces systèmes n'étaient pas basés sur la logique, mais sur des structures de données en forme de réseau qui imitent la structure de la mémoire humaine. Les systèmes basés sur la logique sont plus récents. On y trouve notamment la *logique de description* [BCM+], qui a été créée comme une extension des cadres et des réseaux sémantiques en ajoutant une base formelle construite sur la logique de premier ordre.

2.5.4 Le Web Sémantique

Les techniques de représentation des connaissances ont récemment subi un regain d'intérêt dans le cadre de l'initiative connue sous le nom de *Web Sémantique*. Cette initiative fut lancée par Tim Berners-Lee et al. en 2001 [BLHL01]. Berners-Lee partit d'un constat : l'énorme expansion du Web a produit des quantités immenses de données non structurées qui sont disponibles en ligne. Toute cette information n'est pas facilement exploitable. D'un côté, les humains sont capables de la comprendre, de la parcourir et de trouver des relations entre des données, mais l'échelle phénoménale que le Web a atteint[1] rend cette possibilité irréalisable dans la pratique. D'un autre côté, les machines, qui ont une grande capacité de traitement, ne peuvent pas *comprendre* l'information car elle n'est pas représentée d'une façon structurée qui facilite les traitements.

La solution proposée dans le cadre du Web Sémantique est d'ajouter des métadonnées aux données contenues dans le Web. Les métadonnées sont des données qui décrivent les données ordinaires du Web : elles peuvent décrire un document en soi (par exemple sa date de publication ou son auteur) ou des entités, des personnes, des organisations, etc. qui sont mentionnées dans le document. Afin de rendre le traitement le plus automatique possible, les métadonnées sont exprimées d'une façon structurée ou formelle. Le but final est de fournir des outils qui puissent par exemple trouver des informations pour les utilisateurs, même si ces informations sont dispersées sur plusieurs documents qui ne sont pas liés entre eux. Ce concept peut être étendu pour aussi prendre en compte des services (par exemple des Services Web).

Le mot *sémantique* vient du fait que les métadonnées décrivent en quelque sorte le *sens* des données de telle manière que les machines peuvent l'interpréter. En définitive, cette approche transforme le Web, en le faisant passer d'un ensemble chaotique de données éparses vers une vraie bibliothèque globale pleine de connaissances.

Les métadonnées sont en fait une façon de représenter les connaissances contenues dans un document. Le Web Sémantique s'est naturellement tourné vers les diverses techniques développées dans le champ de la représentation de connaissances afin d'exprimer les métadonnées. Dans ce cadre, plusieurs langages ont été développés. Notamment, on trouve *Ressource Description Framework* (RDF), *RDF Schema* (RDFS) et *Web Ontology Language* (OWL[2]) [SWM04]. Chacun de ces langages se base sur le précédent, et chacun possède une syntaxe en XML qui est utilisée dans les documents Web. OWL permet l'expression d'ontologies, une ontologie étant, selon la définition de Gruber [Gru93], « *une spécification formelle d'une conceptualisation d'un domaine de connaissance* ». Une ontologie est donc une représentation des connaissances d'un certain domaine, qui sert comme modèle partagé de ce domaine et qui peut être traitée automatiquement

[1] En 2008, Google estimait que le nombre de pages uniques disponibles dans le Web était de plus de 1000 milliards : http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

[2] Bien que l'acronyme devrait être *WOL*, *OWL* a été retenu pour des raisons esthétiques : http://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0169.html

par une machine. Une ontologie OWL est constituée principalement de *concepts* et de *relations* entre ces concepts.

Il existe trois variantes du langage OWL qui se différencient selon leur expressivité : OWL-Lite, OWL-DL (pour *Description Logics*) et OWL-Full. Chacune de ces variantes est contenue dans la suivante, qui ajoute des éléments qui augmentent l'expressivité. OWL-Lite est la variante la plus simple, servant principalement à la création de hiérarchies de classification. OWL-DL a été conçu pour fournir une expressivité maximale tout en restant complet et décidable. Le nom d'OWL-DL provient du fait qu'il correspond à une logique de description $\mathcal{SHOIN}(\mathcal{D})$. L'interprétation de cette notation est la suivante : $\mathcal{S}$, raccourci pour $\mathcal{ALC}$, indique qu'il s'agit d'une logique de description attributive avec négation de concepts. OWL-DL permet aussi l'expression de hiérarchies de rôles ($\mathcal{H}$), de concepts énumérés ($\mathcal{O}$), de rôles inverses ($\mathcal{I}$), de restrictions dans la cardinalité des rôles ($\mathcal{N}$) et de types de données ($(\mathcal{D})$) [BCM+]. La dernière variante, OWL-Full, est celle qui fournit la plus grande expressivité des trois (par exemple elle permet d'enrichir le vocabulaire RDF/OWL), mais la propriété de décidabilité du raisonnement est perdue avec cette variante.

En général, OWL-DL offre un bon compromis entre l'expressivité fournie et la capacité d'effectuer des raisonnements. Cette variante est donc la plus utilisée en pratique. Le fait qu'elle soit basée sur la logique de description implique que les bases théoriques du raisonnement avec OWL-DL sont formelles, et donc solides et bien définies. Concrètement, l'inférence est possible en utilisant les algorithmes développés dans le cadre de la logique de description.

Il existe deux versions du langage : OWL 1 (d'habitude appelée OWL simplement) et OWL 2. La version la plus utilisée jusqu'à récemment a été OWL 1, mais OWL 2 commence à être de plus en plus populaire. La version OWL 2, apparue en octobre 2009, reste compatible avec OWL 1. Elle ajoute à OWL 1 des caractéristiques avancées, comme des raccourcis syntaxiques, des extensions des types de données disponibles et quelques nouvelles constructions pour les propriétés. OWL 2 contient également la notion de *profils*, qui sont des variantes qui incluent certaines propriétés d'implantation ou de complexité de l'inférence.

Les ontologies OWL sont devenues le standard *de facto* pour la représentation des connaissances dans le Web Sémantique. Il existe de nombreux outils libres pour la gestion des ontologies OWL et pour effectuer du raisonnement avec elles. Un autre standard important est le *Semantic Web Rule Language* (SWRL) [HPSB+04], une extension d'OWL pour l'expression de règles qui augmentent l'expressivité disponible.

Les langages de description de métadonnées du Web Sémantique sont essentiels pour atteindre une *interopérabilité sémantique*, tant des données que des services. L'interopérabilité sémantique vise à faire travailler ensemble des systèmes ou des données qui n'ont pas un même langage, ou qui n'ont pas été développés pour travailler ensemble à l'origine. La sémantique peut servir comme « pont » entre ces systèmes et donc lisser leurs hétérogénéités [SBT05][PSW+10]. Lassila [Las05] a signalé que le Web Sémantique peut être très utile pour l'informatique ubiquitaire, notamment pour résoudre les problèmes d'interopérabilité entre des sys-

tèmes qui doivent travailler ensemble sans attirer l'attention de l'utilisateur.

Tout comme la représentation des connaissances ou l'IA en général, le Web Sémantique doit poursuivre sa phase de maturité. Les défis à relever restent nombreux et complexes : par exemple, comment gérer des données inconsistantes ou contradictoires, comment travailler avec des degrés d'incertitude ou comment passer à l'échelle pour traiter les immenses quantités de données que le Web héberge sont des problèmes auxquels on ne sait pas encore répondre actuellement. Cependant, les technologies et les langages mentionnés commencent à se déployer sur le Web et ils sont déjà utilisés avec succès dans certains domaines de l'industrie [BOPV10], comme par exemple les Services Web, le commerce électronique ou l'intégration d'applications d'entreprise.

2.6 Synthèse des travaux existants et positionnement

Dans cette section, nous faisons une synthèse des travaux existants dans le domaine des systèmes collaboratifs ubiquitaires et nous positionnons notre contribution par rapport à ces travaux. En premier lieu, nous comparons notre modèle de la collaboration, l'ontologie appelée *Generic Collaboration Ontology* ou GCO, que nous présenterons dans le Chapitre 3, à d'autres travaux qui utilisent les ontologies pour gérer la collaboration. En deuxième lieu, nous positionnons notre plate-forme pour la modélisation et l'exécution de systèmes collaboratifs ubiquitaires, appelée *Framework for Adaptive Collaborative Ubiquitous Systems* ou Facus, présentée dans le Chapitre 4, par rapport à d'autres travaux qui se situent dans le même domaine.

2.6.1 Gestion de la collaboration avec des ontologies

Dans la littérature il existe plusieurs travaux qui ont proposé des ontologies pour modéliser la collaboration (ou des aspects spécifiques de la collaboration). Nous synthétisons ici ceux que nous considérons les plus pertinents.

Anzures-García et al. [AGSGHPR07] proposent une ontologie pour la gestion des sessions et des politiques dans les collecticiels. Cette ontologie met l'accent sur les *politiques* qui décrivent comment la session peut évoluer lors des changements dans le groupe et dans la façon de travailler. La collaboration que considèrent les auteurs est la collaboration entre les différentes personnes travaillant au sein d'une entreprise ou d'une organisation. En conséquence, les structures organisationnelles considérées sont hiérarchiques. L'ontologie proposée est conçue pour une utilisation en ligne, c'est-à-dire pour gérer les sessions en temps réel. Cependant, les auteurs ne proposent pas d'architecture ou de service de gestion de session qui utilise l'ontologie définie.

Rajsiri et al. [RLBP08] présentent une ontologie, appelée *Collaborative Network Ontology*, qui modélise les processus collaboratifs entre des organisations. Le but principal de cette ontologie est de déduire, à partir des exigences de collaboration d'un réseau d'entreprises, un processus pour

gérer les relations économiques entre ces entreprises. Pour cela, les auteurs proposent des règles qui permettent de faire la déduction à partir des connaissances contenues dans l'ontologie. Par exemple, à partir d'une instance de l'ontologie qui contient trois entreprises qui jouent le rôle de fournisseur, de distributeur et d'acheteur, l'exécution des règles crée un processus qui détaille toutes les étapes à franchir pour implanter la collaboration. Par exemple, en premier lieu l'acheteur effectue une commande, puis le fournisseur livre la marchandise au distributeur, et ainsi de suite. En conséquence, la collaboration prise en compte dans ce travail n'a pas lieu entre des humains directement mais plutôt entre des entreprises.

Gu et al. [GXW+05] proposent une architecture pour la résolution de conflits sémantiques lors de l'édition collaborative de documents en mode synchrone. Cette architecture, qui suit le paradigme client/serveur, repose sur des ontologies pour mettre en œuvre l'interopérabilité sémantique entre les termes utilisés par les utilisateurs qui travaillent ensemble sur un même document. Ce travail considère aussi l'interopérabilité entre les intentions des utilisateurs lors de l'édition du document. Les auteurs de ce travail proposent un langage d'ontologies appelé *Four-Layered ontology Description Language* (FLoDL) afin de représenter les ontologies dans leur architecture. L'objet des ontologies utilisées n'est pas la collaboration en elle-même mais plutôt les termes et les intentions utilisés lors de la conception collaborative du document.

L'ontologie que nous proposons dans ce mémoire, *Generic Collaboration Ontology* (GCO) [SVT10], modélise les sessions, synchrones ou asynchrones, entre plusieurs utilisateurs qui ont un but commun. La collaboration considérée dans GCO est générale ; elle est applicable au cas des entreprises, mais d'autres sont aussi possibles. La modélisation d'éléments qui ont une relation avec le déploiement, comme les dispositifs des utilisateurs, les flux de données ou les composants qui gèrent ces flux rend possible l'utilisation de cette ontologie dans des environnements ubiquitaires, dans lesquels le déploiement doit être adaptatif. Les règles associées à l'ontologie permettent de déduire une configuration de déploiement de haut niveau à partir de la structure du groupe de collaboration. Le contenu de la collaboration n'est pas modélisé dans cette ontologie ; ce sont les participants, leurs rôles et la structure du groupe qui sont considérés.

Comparatif Le Tableau 2.2 présente une vue synthétique des approches présentées. Les critères que nous avons pris en compte sont les suivants :

- le langage utilisé pour la description des ontologies ;
- le but de l'approche considérée ;
- les principaux concepts qui sont modélisés dans les ontologies de l'approche ;
- le type de collaboration considérée, selon qu'il s'agisse de sessions de collaboration synchrones ou asynchrones ;
- l'utilisation ou non de règles de déduction en complément des ontologies ;
- la possibilité d'utiliser l'approche considérée dans des environnements ubiquitaires ou non.

Approche	Language	But	Concepts principaux	Collaboration synchrone/ asynchrone	Utilisation de règles	Informatique ubiquitaire
Approche d'Anzures-Garcia et al. [AGSGHPR07]	OWL	gestion des politiques de session	groupe, politique, rôle, droit, tâche, utilisateur	synchrone et asynchrone	non	non
Collaborative Network Ontology [RLBP08]	OWL	automatisation de la spécification des processus collaboratifs inter-organisations	réseau collaboratif, relation, rôle, participant, but commun, topologie, services	asynchrone	oui	non
Approche de Gu et al. [GXW+05]	FLoDL	édition collaborative de documents	non spécifié	synchrone	non	non
Generic Collaboration Ontology [SVT10]	OWL	gestion de session et déploiement de composants dans les environnements ubiquitaires	session, rôle, nœud, flux, outil, composant, type de données, dispositif	synchrone et asynchrone	oui	oui

TAB. 2.2 – *Comparaison des approches basées sur les ontologies pour la gestion de la collaboration*

Le langage OWL est utilisé pour la description des ontologies dans tous les travaux, sauf dans celui de Gu et al. En effet, ces derniers ont proposé leur propre langage, FLoDL.

Concernant le but des différentes approches, on remarque que, bien que le thème central soit la collaboration, les aspects considérés dans chacun des travaux sont légèrement différents. Le travail de Rajsiri et al. se focalise sur la collaboration entre organisations et non pas entre humains comme dans le reste des travaux. Dans le cas de Gu et al., la collaboration est appliquée au cas particulier de l'édition de documents. Dans les cas de Anzures-García et al. et de GCO, on considère la collaboration en général (le domaine n'est pas spécifié). Cependant, le premier se focalise plus sur les organisations fortement hiérarchisées telles que les entreprises, tandis que GCO est plus générique.

Concernant les concepts modélisés, certains se retrouvent dans plusieurs ontologies, notamment les concepts *rôle* ou *groupe*. Dans le cas de Rajsiri et al., la notion de groupe est implicite, car chaque *participant* est un groupe (une entreprise). Ensuite, chaque ontologie a des concepts plus spécifiques selon le but poursuivi.

Le type de collaboration pris en compte majoritairement est la collaboration synchrone. Anzures-García et al. ne précisent pas le type de collaboration. Leurs exemples traitent cependant d'avantage le cas de collaboration synchrone, mais ils n'excluent pas des cas d'utilisation asynchrones. Le seul travail qui traite explicitement le cas de la collaboration asynchrone est celui de Rajsiri et al., motivé par la nature des participants dans la collaboration : des organisations. Dans GCO, le type de collaboration n'est pas figé ; il traite aussi bien le cas synchrone que le cas asynchrone.

Les deux approches qui se servent des règles pour implanter des inférences sont celle de Rajsiri et al. et GCO. Dans le premier cas, il s'agit de déduire un processus de collaboration à partir des rôles des participants, tandis que dans le deuxième il s'agit de trouver un ensemble de flux et de composants qui soutiennent la collaboration des participants.

Finalement, le seul travail qui considère la collaboration dans les environnements ubiquitaires est GCO. En fait, les autres travaux ne précisent pas quel type d'environnement est envisagé ; ils font cependant l'hypothèse implicite que les utilisateurs travaillent avec des systèmes de type bureau, excluant de fait les environnements ubiquitaires. Les éléments comme les dispositifs ou les composants collaboratifs ne sont pas modélisés dans ces travaux. Ils considèrent que chacun des participants a installé et a configuré les logiciels nécessaires au préalable, avant le début de la collaboration. La grande dynamique des environnements ubiquitaires et l'idée de ne pas distraire l'utilisateur requièrent des solutions de déploiement adaptatif. Dans le cas de GCO, nous voulons justement faciliter le déploiement adaptatif de composants sur les machines des utilisateurs.

2.6.2 Systèmes collaboratifs pour les environnements ubiquitaires

Dans cette sous-section, nous analysons des systèmes conçus pour les environnements ubiquitaires qui comportent des aspects collaboratifs. Nous nous intéressons particulièrement à la façon dont l'adaptation est prise en compte dans ces systèmes ainsi qu'au type de contexte considéré.

Edwards [Edw05] propose un système appelé *Intermezzo* qui fournit un service de stockage de données contextuelles et un service de notification lié à ces données. Intermezzo peut être utilisé comme une bibliothèque pour faciliter la construction d'applications collaboratives et sensibles au contexte dans des environnements ubiquitaires. Les aspects collaboratifs s'articulent autour des *activités*. Une activité représente l'utilisation que fait un utilisateur d'un ensemble d'objets du système (par exemple des documents). Le modèle de contexte derrière Intermezzo est très riche. En particulier, il propose des solutions pour des problèmes tels que l'ambiguïté, l'évolution des données contextuelles ou la détermination de leur équivalence. Cependant, seul le contexte externe est pris en compte. L'adaptation est en dehors de l'étendue de ce travail : c'est aux applications de réagir aux changements dans le contexte que le service de notification signale.

Locatelli et Vizzari [LV07] présentent une approche pour la modélisation et la conception d'environnements collaboratifs ubiquitaires basée sur le modèle *Multilayered Multi-Agent Situated System* (MMASS). Ce modèle facilite la représentation et la gestion d'information contextuelle sur plusieurs couches, tant physiques, par exemple la position géographique, que logiques, par exemple le rôle d'un utilisateur dans un groupe. Chaque couche correspond à un niveau d'abstraction du contexte, qui est représenté par un graphe. Ces graphes contiennent des agents logiciels qui communiquent directement entre eux ou indirectement à travers des signaux qu'ils émettent et reçoivent. Les interactions entre les agents modélisent les relations entre les éléments du contexte ; par exemple la relation de *proximité* de deux membres d'un groupe représente la probabilité qu'ils ont de collaborer ensemble. Cette approche théorique permet de faire des modélisations complexes. En revanche, elle reste assez abstraite ; les auteurs ne donnent pas de détails sur les couches spécifiques à prendre en compte ou sur comment implanter des signaux pertinents pour les échanges entre les agents du modèle. La collaboration humaine est prise en compte dans ce travail comme une activité de haut niveau qui est soutenue par les instances mises en place à partir des modèles gérés. Quant à l'adaptation, elle n'est pas prise en compte explicitement dans ce travail ; la réponse aux changements dans le modèle sera implantée dans les applications qui utilisent l'approche proposée.

Padovitz et al. [PLZ08] proposent un modèle de contexte et une approche de raisonnement pour supporter la sensibilité au contexte dans les environnements ubiquitaires. Le modèle proposé, appelé *Context Spaces*, décrit le contexte et les situations comme des structures géométriques dans un espace multidimensionnel. Ce modèle est utilisé par des agents qui ont des vues partielles du contexte, et qui se coordonnent ensemble pour construire une vue globale. Le raisonnement est basé sur une algèbre qui utilise ces structures, et qui permet de fusionner les différentes vues que les agents ont du contexte. Cette fusion de vues partielles met en œuvre un raisonnement global sur le contexte. Dans cette approche, la collaboration est utilisée entre les agents afin d'échanger leurs vues du contexte et d'effectuer le raisonnement global. La collaboration de haut niveau entre des utilisateurs humains n'est pas prise en compte. L'adaptation prise en compte dans ce travail est comportementale : le comportement des agents s'adapte aux changements observés dans le contexte.

Ejigu et al. [ESB07] présentent une plate-forme de services sensibles au contexte appelée *Collaborative Context-Aware service platform* (CoCA), qui peut être utilisée pour le développement d'applications sensibles au contexte dans des environnements ubiquitaires. Les données de contexte sont représentées dans un modèle appelé *Generic Context Management Model* (GCoM). GCoM est un modèle du contexte, tant externe que des ressources, basé sur des ontologies OWL. CoCA utilise ce modèle afin d'acquérir, de représenter et de stocker les données de contexte. Ensuite, il peut agréger et interpréter, avec des techniques de raisonnement basées sur OWL, les informations contextuelles, et il transmet des notifications à l'application afin qu'elle puisse réagir aux changements de contexte. CoCA utilise la collaboration comme un moyen pour mettre en œuvre un partage de ressources entre les dispositifs : les dispositifs les plus puissants peuvent *aider* les moins capables afin de réaliser des tâches lourdes comme le raisonnement ou l'acquisition du contexte. Cette collaboration est implantée en Java en mode pair-à-pair avec le protocole JXTA[3]. La collaboration humaine n'est pas considérée dans cette plate-forme, mais les applications qui l'utilisent, comme l'application illustrative de campus ubiquitaire présentée par les auteurs, peuvent le faire.

Fawaz et al. [FNBS07] proposent l'intergiciel appelé *Collaboration-based Content Adaptation Middleware* (ConAMi). Cet intergiciel vise l'adaptation du contenu dans les réseaux ad-hoc mobiles (*Mobile Ad-hoc Networks* ou MANET en anglais), qui sont un cas particulier des environnements ubiquitaires où des dispositifs mobiles forment des réseaux ad-hoc dynamiques. ConAMI permet aux dispositifs de collaborer afin d'adapter au contexte le contenu offert à l'utilisateur. Par exemple, si un utilisateur préfère avoir des informations en audio et en français, et que son dispositif ne supporte que le format *wav*, alors on lui proposera des informations audio en langue française dans ce format. L'adaptation prend donc en compte tant le contexte externe, par exemple les préférences de l'utilisateur, que celui des ressources, par exemple les formats reconnus par son dispositif. L'adaptation est mise en œuvre en composant les différents services proposés par les dispositifs à proximité. La collaboration prise en compte dans ce travail est celle qui est utilisée entre les dispositifs pour composer dynamiquement leurs services. Cette collaboration est implantée dans un algorithme de construction d'arbres qui permet d'effectuer efficacement l'adaptation du contenu.

Le *framework* que nous proposons dans ce mémoire, Facus [SBT+09], permet la conception et le développement d'applications collaboratives dans des environnements ubiquitaires. Dans Facus, l'accent est mis sur l'adaptation de l'architecture collaborative aux changements du contexte externe et du contexte des ressources. En particulier, l'objet de l'adaptation est le déploiement de composants utilisés pour envoyer des données entre les participants dans la collaboration. La collaboration prise en compte concerne les activités de haut niveau menées par des humains afin d'atteindre un but commun.

[3] https://jxta.dev.java.net/

Comparatif Le Tableau 2.3 présente une vue synthétique des approches présentées. Les critères que nous avons pris en compte sont les suivants :

- le rôle de la collaboration dans l'approche : la collaboration est un but en elle-même ou bien elle est utilisée comme un moyen ou outil ;
- le type de contexte (externe et/ou des ressources) pris en compte ;
- la prise en compte ou non de l'adaptation, et le type d'adaptation visé ;
- l'existence d'une implantation de l'approche proposée.

On voit dans le tableau que trois travaux étudiés parmi six prennent en compte la collaboration entre des humains comme une activité de haut niveau pour laquelle il faut fournir des moyens qui la soutiennent : Intermezzo, MMASS et Facus. Les autres travaux utilisent la collaboration (entre les différentes parties du système, souvent des agents) comme un outil pour mettre en œuvre divers objectifs : partage de ressources, raisonnement distribué, etc.

Chaque approche propose des modèles plus ou moins riches pour la représentation et le traitement du contexte. Le contexte externe est pris en compte dans toutes les approches, tandis que CoCA, ConAMi et Facus modélisent aussi le contexte des ressources. Cette prise en compte du contexte des ressources dans ces systèmes leur permet d'optimiser l'utilisation des moyens de calcul et de communication, ce qui est nécessaire dans les environnements ubiquitaires, où les dispositifs ont souvent des ressources contraintes.

L'adaptation n'est pas considérée explicitement dans toutes les approches. Dans les approches Intermezzo, MMASS et CoCA, l'adaptation est prise en compte implicitement ; on informe l'application des changements du contexte, mais la responsabilité de s'adapter à ces changements revient à l'application. Chacune des trois approches restantes prend en compte une adaptation différente. Dans l'approche Context Spaces, l'adaptation est comportementale, car les agents du système changent leur comportement en fonction du contexte perçu. Dans ConAMi, l'adaptation transforme les contenus présentés à l'utilisateur, qui sont adaptés à ses préférences et aux capacités de son dispositif. Finalement, dans Facus l'adaptation est architecturale, car on adapte le déploiement de composants afin de mieux soutenir la collaboration humaine.

Concernant les implantations, la plupart des approches sont implantées par des prototypes, majoritairement en Java, qui servent à montrer la faisabilité des approches.

Conclusion du chapitre

Notre travail se trouve au carrefour de deux grands domaines : l'informatique ubiquitaire et la collaboration. De plus, il met à contribution des techniques développées dans d'autres domaines plus ou moins connexes comme la modélisation du contexte, l'adaptation pendant l'exécution et la prise en compte de la sémantique. Dans ce chapitre nous avons parcouru les bases de ces domaines, dans lesquels s'inscrit notre travail.

Ensuite, nous avons étudié les travaux récents qui s'attaquent à des problématiques similaires à celle de notre contribution. Nous avons comparé les objectifs de chaque travail et la façon de les mettre en œuvre.

Travail	Rôle de la collaboration	Type de contexte considéré	Prise en compte de l'adaptation	Implantation
Intermezzo [Edw05]	but (gestion de session et contrôle d'accès)	externe	non considérée	plate-forme en C++ et Python
Approche MMASS [LV07]	but (diffusion d'informations)	externe	non considérée	plusieurs prototypes
Approche Context Spaces [PLZ08]	outil (traitement distribué du contexte avec des agents)	externe	comportementale (agents)	bibliothèque en Java
CoCA [ESB07]	outil (partage de ressources)	externe et des ressources	non considérée	prototype en Java
ConAMi [FNBS07]	outil (collaboration entre dispositifs pour composer des services)	externe et des ressources	adaptation du contenu	prototype
Facus [SBT+09]	but (gestion de session et déploiement)	externe et des ressources	architecturale (déploiement de composants)	prototype en Java

Tab. 2.3 – *Comparaison des approches collaboratives pour les environnements ubiquitaires*

Concernant la partie modèle, il existe peu d'ontologies qui modélisent la collaboration humaine. Notre ontologie de la collaboration, GCO, contient quelques éléments similaires à ceux des autres ontologies de la collaboration. Cependant, GCO est la seule ontologie qui vise la déduction d'un schéma de déploiement à partir des éléments collaboratifs de haut niveau, atteignant ainsi une interopérabilité sémantique entre la collaboration de haut niveau et le déploiement de bas niveau. GCO est expliqué en détail dans le Chapitre 3.

Concernant les systèmes collaboratifs ubiquitaires, peu d'entre eux fournissent des moyens pour soutenir les activités collaboratives de haut niveau entre les humains. Finalement, seul notre *framework* FACUS soutient les activités collaboratives dans des environnements ubiquitaires avec un déploiement adaptatif de composants qui prend en compte le contexte externe et celui des ressources. Les caractéristiques détaillées et la conception de FACUS seront développées dans le Chapitre 4.

Ontologie Générique de la Collaboration (GCO)

3

Sommaire

Ce chapitre présente notre modèle central de la collaboration : l'Ontologie Générique de la Collaboration (*Generic Collaboration Ontology* ou GCO en anglais) [STV08][STV10][SVT10]. Le but principal de ce modèle est de servir de point de référence pour l'expression de situations de collaboration entre des utilisateurs organisés dans des groupes. Ce modèle est indépendant du domaine considéré, mais il peut être spécifié pour la prise en compte de domaines concrets. Le deuxième but de GCO est de servir

de base pour la déduction et l'expression d'un schéma de déploiement à partir des besoins de collaboration d'un groupe.

Tout d'abord, nous introduisons les langages et les technologies utilisés pour l'expression et le traitement de GCO : les ontologies OWL, les règles SWRL et le raisonnement avec OWL. Ces outils, issus du Web Sémantique, permettent la représentation et la gestion de connaissances, et sont dotés d'une base formelle. Par la suite, nous présentons les outils disponibles pour l'édition et la manipulation des ontologies.

Dans la partie suivante, nous expliquons le contenu de l'ontologie, les règles associées et la façon dont ces règles sont traitées dans le processus d'inférence. Par la suite, nous détaillons les principes suivis dans la conception de GCO et les propriétés qui en résultent.

Finalement, nous expliquons comment GCO peut être utilisé en ligne comme modèle de la collaboration.

3.1 Introduction aux ontologies OWL

Dans cette section, nous décrivons les principes de base des ontologies OWL [SWM04], du raisonnement avec OWL et des règles SWRL [HPSB+04], ainsi que les possibilités qu'ils offrent. Cette description permettra de comprendre les concepts et les choix détaillés dans le reste de ce chapitre. Nous présentons également quelques uns des outils disponibles actuellement pour la manipulation des ontologies OWL et des règles SWRL et pour le raisonnement dans le cadre d'OWL.

3.1.1 Définition des éléments d'une ontologie OWL

Dans cette sous-section, nous fournissons des définitions nécessaires pour comprendre le principe des ontologies OWL. Nous commençons par une définition générale :

Déf. 3.1 (Domaine du discours) *Le* domaine du discours *est le domaine que l'on représente dans une ontologie, c'est-à-dire, la partie du monde qui est l'objet de la modélisation.*

Ensuite, nous définissons les éléments principaux que l'on trouve dans une ontologie OWL :

Déf. 3.2 (Individu/instance) *Les* individus *ou* instances *sont les objets du domaine de discours que l'on représente dans une ontologie.*

Déf. 3.3 (Concept/classe, sous-classe, super-classe, taxonomie) *Un* concept *ou* classe *regroupe un ensemble d'individus qui ont des caractéristiques communes. Une classe peut être* sous-classe *d'une autre, appelée* super-classe *: dans ce cas, tout individu appartenant à la sous-classe appartient aussi à la super-classe. Une* taxonomie *est une hiérarchie de classes qui ont des relations sous-classe/super-classe entre elles.*

Déf. 3.4 (Relation/propriété, domaine, portée, sous-relation, super-relation) *Une* relation *ou* propriété *modélise le rapport qui existe entre deux classes (relation* objet*) ou entre une classe et un type de données (relation* type de données*). Le* domaine *d'une relation est l'ensemble de classes qui peuvent être à l'origine de la relation. La* portée *de la relation est l'ensemble de classes ou de types de données qui peuvent être la destination de la relation. Une relation peut être une* sous-relation *d'une autre, appelée* super-relation*. Dans ce cas, le domaine et la portée de la sous-relation sont contenus respectivement dans le domaine et dans la portée de la super-relation.*

Déf. 3.5 (Instance d'une relation) *Une* instance d'une relation *relie un individu qui appartient au domaine de la relation à un individu ou à un type de données qui appartient à la portée de la relation.*

Par abus de langage, souvent on appelle les instances de relations tout simplement *relations*. Le contexte du terme permet de distinguer si l'on parle de relations proprement dites ou de leurs instances.

Les définitions suivantes détaillent les attributs des propriétés.

Déf. 3.6 (Propriété inverse) *On peut définir une propriété comme l'*inverse *d'une propriété donnée. Cela veut dire que, si une instance de cette dernière propriété relie l'indi-*

vidu a *à l'individu* b, *alors on peut déduire qu'une instance de la propriété inverse relie* b *à* a.

Déf. 3.7 (Propriété fonctionnelle, inverse fonctionnelle) *Une propriété est* fonctionnelle *quand elle ne peut avoir qu'une seule instance pour un individu donné. L'inverse d'une propriété fonctionnelle est son* inverse fonctionnelle.

Déf. 3.8 (Propriété transitive) *Une propriété est* transitive *lorsque, si l'individu* a *est relié à* b *par une instance de cette propriété et que* b *est relié à* c *par une autre instance de la propriété, alors on peut déduire que* a *est relié à* c *par une instance de la propriété.*

Déf. 3.9 (Propriété symétrique) *Une propriété est* symétrique *quand pour tout* a *relié à* b *par une instance de cette propriété, on peut déduire que* b *est relié à* a *par une autre instance de la propriété.*

Pour définir une classe dans OWL, on fournit un ensemble de conditions logiques. Ces conditions peuvent être nécessaires ou nécessaires et suffisantes. Elles sont construites à partir d'autres classes, par union, par intersection ou par héritage. On peut également imposer des restrictions aux propriétés de la classe.

Déf. 3.10 (Restriction, restriction existentielle, universelle, de cardinalité, de valeur) *Une* restriction *consiste à limiter le nombre ou la nature des valeurs que peuvent avoir les propriétés des individus d'une classe. Une restriction peut être* existentielle *(si elle oblige à avoir au moins une valeur de la propriété dans un ensemble donné),* universelle *(si elle oblige à avoir toutes les valeurs d'une propriété dans un ensemble donné), de* cardinalité *(si elle oblige à avoir un nombre de valeurs minimal, maximal ou exact pour une propriété) ou* de valeur *(si elle oblige à avoir une valeur donnée pour la propriété).*

Déf. 3.11 (Classes disjointes) *Deux classes sont dites* disjointes *quand il ne peut exister des individus qui appartiennent à la fois aux deux classes.*

Dans OWL, les classes ne sont pas disjointes par défaut ; il faut le déclarer explicitement.

Les ontologies OWL sont extensibles grâce au mécanisme d'importation :

Déf. 3.12 (Importation d'ontologie) *Une ontologie peut* importer *une autre ontologie pour avoir une visibilité sur ses éléments. L'ontologie qui importe a un accès en lecture seule à tous les éléments contenus dans l'ontologie importée. Elle peut ensuite ajouter de nouveaux éléments, qui ne seront visibles que dans l'ontologie qui importe. L'importation est transitive : une ontologie qui importe une deuxième importe indirectement toutes les ontologies importées dans cette deuxième ontologie.*

En général, l'importation se fait en indiquant, dans l'ontologie qui importe, l'URL où se trouve l'ontologie importée. Ce mécanisme permet une grande flexibilité, car on peut réutiliser des ontologies existantes juste en les référençant dans un nouveau fichier OWL et en ajoutant de nouvelles classes, propriétés, individus, règles, etc. Souvent, on trouve des ontologies qui suivent ce schéma : une première ontologie de *haut niveau* qui contient des éléments génériques est importée par une deuxième ontologie *de domaine* qui les spécialise pour un domaine concret. Dans ce cas, on dit que la deuxième ontologie *étend* la première.

Exemple La Figure 3.1 montre une ontologie simple pour illustrer les éléments expliqués. Dans cette ontologie, il y a 8 classes : `Personne`, `TravailleurManuel`, `Plombier`, `Politicien`, `Métier`, `MétierManuel`, `Plomberie` et `Politique`. Les flèches `is-a` représentent les relations de super-classe/sous-classe : par exemple, `Plombier` est une sous-classe de `Personne`, car tous les plombiers sont des personnes. La relation `aMétier` relie `Personne` à `Métier`, ce qui veut dire que les personnes peuvent avoir un métier. Les relations `père`, `oncle` et `frère` relient la classe `Personne` à elle même, car le père, l'oncle ou le frère d'une personne sont aussi des personnes. Les classes `Personne` et `Métier` sont disjointes.

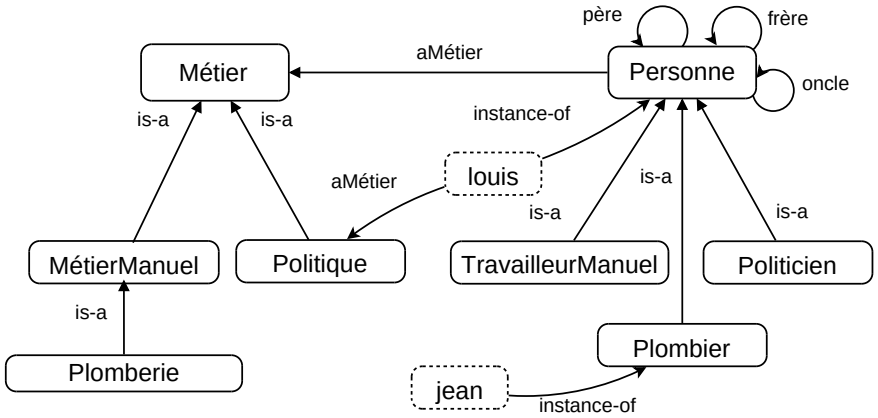


FIG. 3.1 – *Exemple d'ontologie OWL*

La classe `TravailleurManuel` est définie comme

$$TravailleurManuel \equiv Personne \sqcap \exists aMétier.MétierManuel$$

Cela veut dire que la condition nécessaire et suffisante ($\equiv$) pour qu'un individu soit considéré comme travailleur est qu'il doit être une personne (*Personne*) et ($\sqcap$) avoir une instance de la relation `aMétier` qui le relie à un métier manuel ($\exists aMétier.MétierManuel$). De la même façon :

$$Politicien \equiv Personne \sqcap \exists aMétier.Politique$$

$$Plombier \equiv Personne \sqcap \exists aMétier.Plomberie$$

Finalement, il y a deux individus, `jean` et `louis`. On sait que le premier est un plombier et que le deuxième est une personne et que son métier est la politique.

3.1.2 Raisonnement avec OWL

Comme nous l'avons mentionné, le fait qu'OWL possède une base théorique formelle (la logique de description) permet l'implantation de logiciels appelés *moteurs d'inférence* ou *raisonneurs*, qui sont capables de traiter une ontologie OWL pour déduire des faits qui ne sont pas explicitement déclarés [BCM+]. C'est-à-dire, ils peuvent trouver des informations qui sont *implicitement* contenues dans l'ontologie pour les rendre explicites. Ce processus s'appelle *inférence* ou *raisonnement*.

La principale tâche de raisonnement qu'un moteur d'inférence peut effectuer est connue comme *subsomption*. La subsomption permet de savoir si une classe est une sous-classe d'une autre ou non. En effectuant cette tache sur toutes les classes d'une ontologie OWL, le moteur d'inférence peut construire une *hiérarchie de classes inférée* (en opposition à la hiérarchie de classes déclarée) dans laquelle toutes les relations super-classe/sous-classe sont explicitées.

Dans notre exemple, la hiérarchie de classes inférée contient le fait que `Plombier` est une sous-classe de `TravailleurManuel`, car tous les plombiers ont comme métier la `Plomberie`, qui est un `MétierManuel`, donc ils remplissent toutes les conditions nécessaires et suffisantes pour être dans la classe `TravailleurManuel`. Également, `Plomberie` est une sous-classe de `Métier`, car elle l'est de `MétierManuel`, et tous les métiers manuels sont des métiers.

Une autre tâche de raisonnement standard est la vérification de la cohérence de l'ontologie, qui permet de détecter s'il y a des classes qui ne sont pas cohérentes, c'est-à-dire, pour lesquelles il n'est pas possible de déclarer un individu sans avoir une contradiction logique. Cette tâche est très utile lors du processus de création d'une ontologie pour le débogage de celle-ci. Par exemple, si on veut déclarer une nouvelle classe qui est un sous-concept de `Métier` et de `Personne` en même temps, un raisonneur déduira que cette classe est incohérente, car, `Personne` et `Métier` étant disjointes, on ne peut avoir aucun individu qui appartient aux deux classes.

D'autres tâches de raisonnement permettent de déduire des faits tels que l'appartenance d'un individu à une classe, l'existence de propriétés liant deux individus à partir de la transitivité, de la symétrie et des propriétés inverses, etc.

Les implantations de ces tâches utilisent des algorithmes bien connus comme les algorithmes basées sur la *méthode des tableaux* [SSS91] qui travaillent avec les définitions logiques des classes et des propriétés afin de faire les déductions.

Les moteurs d'inférence peuvent utiliser ces tâches de raisonnement de base afin de fournir des services de raisonnement plus complexes, notamment :

1. *recherche* : permet de trouver tous les individus qui sont des instances (directes ou indirectes) d'un concept donné ;
2. *réalisation* : permet de trouver le concept le plus spécifique auquel appartient (directement ou indirectement) un individu donné.

Par exemple, dans notre ontologie, ces services permettent de trouver que l'individu `jean` appartient aux classes `Plombier`, `TravailleurManuel` et `Personne` et que `louis` appartient aux classes `Personne` et `Politicien`. Ces tâches sont d'une grande utilité car elles permettent l'utilisation transparente des connaissances inférées dans les applications. Par exemple, si l'application demande au moteur d'inférence tous les individus appartenant à un concept donné, le moteur inclura dans la réponse tant les individus qui ont été explicitement déclarés comme instances du concept que ceux qui le sont implicitement.

Le raisonnement dans OWL applique le principe connu comme *hypothèse du monde ouvert*. Ce principe stipule que l'on ne peut pas considérer qu'un fait n'existe pas, à moins d'avoir déclaré explicitement sa non-existence. Autrement dit, on ne considérera pas comme fausse une proposition simplement parce qu'on ne l'a pas déclarée comme étant vraie ; la proposition sera considérée comme « inconnue ». Cette hypothèse a une forte influence sur la façon de définir les éléments d'une ontologie. Par exemple, si on déclare deux classes (non disjointes) A et B et un individu a qui appartient à une classe A, on ne peut pas considérer que a n'appartient pas à la classe B ; il faudra le déclarer explicitement afin que le raisonnement (et le traitement des règles) s'appuie sur le fait que a n'est pas une instance de B. De la même façon, il faudra déclarer explicitement que deux classes sont disjointes pour qu'on puisse déduire qu'un individu appartenant à la première n'appartient pas à la deuxième, etc. En définitive, avec l'hypothèse du monde ouvert, on considère que les connaissances dont on dispose ne sont pas forcément complètes, et on ne peut donc rien supposer sur ce qui n'a pas été déclaré. Dans notre exemple, on ne pourra pas supposer que `jean` n'est pas politicien, car les classes `Plombier` et `Politicien` ne sont pas disjointes.

Un autre concept, intimement lié à celui du monde ouvert, est l'hypothèse du nom unique (UNA ou *Unique Name Assumption* en anglais), qui n'est pas pris en compte dans OWL. Cela veut dire que le fait d'avoir deux individus ayant deux noms différents n'implique pas que ces individus ne sont pas le même. Si on souhaite que deux individus donnés soient considérés différents, il faut le déclarer explicitement. Dans notre exemple, on ne peut pas savoir si `jean` et `louis` sont le même individu ou pas, car on n'a pas assez d'information.

Une dernière caractéristique importante du raisonnement avec OWL est la monotonicité. Le fait qu'OWL soit monotone implique que tout fait présent dans une ontologie ne peut pas être *retiré* [SWM04] par des nouvelles informations apprises. Notamment, les propositions inférées par un moteur d'inférence ne peuvent qu'ajouter de l'information, mais jamais en effacer. Même si l'information ajoutée contredit celle qui existait précédemment, cette dernière ne sera pas effacée. Par exemple, si dans notre ontologie on dit que `jean` est un politicien, cela n'efface pas le fait qu'il est un plombier. Il sera un politicien et un plombier *en même temps* dans cette ontologie, car le nouveau fait ne remplace pas le précédent. Si on avait déclaré les classes `Politicien` et `Plombier` comme disjointes, alors un moteur d'inférence pourra détecter que le nouveau fait est incohérent.

3.1.3 Règles SWRL

Le Semantic Web Rule Langage (SWRL) [HPSB+04] est un langage de règles proposé par le W3C qui combine OWL-DL avec le Rule Markup Langage (RuleML). SWRL étend OWL-DL en ajoutant des clauses de Horn [Hor51]. Cet ajout augmente l'expressivité d'OWL, mais, en général, cette expressivité implique la perte de la décidabilité [PSG+05]. Toutefois, pour la plupart des applications pratiques, il est possible de se limiter à un sous-ensemble de règles appelé *DL-sûres* (*DL-safe* en anglais), qui est décidable.

Une règle SWRL présente la forme suivante :

$$b_1 \wedge \cdots \wedge b_n \rightarrow a_1 \wedge \cdots \wedge a_n$$

où $b_1 \wedge \cdots \wedge b_n$ est le corps ou antécédent de la règle et $a_1 \wedge \cdots \wedge a_n$ est l'entête ou conséquent. Les termes $a_1, \ldots, a_n, b_1, \ldots, b_n$ sont des *atomes* SWRL. Un atome peut représenter une relation (prédicat binaire), un concept (prédicat unaire) ou un *built-in* (prédicats n-aires). L'interprétation de la règle est la suivante : si les conditions spécifiées dans l'antécédent sont vérifiées, alors on peut déduire que les propositions spécifiées dans le conséquent sont vérifiées aussi. Il existe d'autres syntaxes, notamment une syntaxe XML basée sur celle d'OWL, ce qui permet d'inclure les règles SWRL associées à une ontologie dans le même fichier XML que l'ontologie.

L'utilité de ces règles est d'exprimer des relations qui seraient trop compliquées, voire impossibles, à exprimer avec OWL-DL seulement. Par exemple, dans notre ontologie on peut représenter la relation entre un oncle et son neveu à partir des relations père–fils et frère–frère. La règle SWRL qui exprime cette relation est représentée dans l'exemple de la Figure 3.2.

```
Personne(?x) ∧ Personne(?y) ∧ Personne(?z)
∧ pere(?x,?y) ∧ frere(?x,?z) → oncle(?z,?y)
```

FIG. 3.2 – *Exemple de règle SWRL*

Les éléments `x`, `y` et `z`, qui sont précédés d'un point d'interrogation, sont des variables qui représentent des individus. Cette règle veut dire que, si l'on a trois individus appartenant à la classe `Personne` appelés `x`, `y` et `z`, que `x` est le père de `y` et que `x` a un frère `z`, alors `z` est l'oncle de `y`.

Dans les règles SWRL, on peut utiliser deux relations spéciales `sameAs(p,q)` et `differentFrom(p,q)` qui servent respectivement à déclarer que deux individus sont le même ou sont différents. Elles sont nécessaires du fait qu'OWL n'utilise pas l'hypothèse du nom unique, comme nous l'avons dit précédemment.

Les *built-ins* sont des prédicats d'arité libre, qui servent à implanter des fonctions pratiques utilisables dans les règles SWRL. Il existe un ensemble de *built-ins* prédéfinis qui servent par exemple à faire des comparaisons, à faire des opérations mathématiques, à concaténer des chaînes de caractères, etc. Par exemple, *le built-in* `swrlb:greaterThan(?age,17)` permet de comparer deux nombres (ici, celui qui est dans la variable `age` et l'entier 17). Si le premier est plus grand, alors le `built-in` sera évalué comme vrai ; autrement il sera faux.

3.1.4 Outils pour le traitement des ontologies OWL et des règles SWRL

Une des raisons du succès d'OWL et des technologies du Web Sémantique est l'existence de nombreux outils pour la gestion des ontologies. En effet, on dispose de bibliothèques, d'interfaces de programmation (API en anglais), d'éditeurs, de moteurs d'inférence et de règles, etc. qui facilitent la création et l'édition d'ontologies et de règles, leur accès depuis un programme, l'exécution du raisonnement et le traitement de règles. De

plus, une grande partie de ces logiciels ont des licences libres, ce qui permet de les obtenir, de les étudier, de les modifier et de les partager plus facilement.

Édition d'ontologies

L'éditeur Protégé[1] [KFNM04], développé par l'Université de Stanford en collaboration avec l'Université de Manchester, est le standard de facto pour la création et l'édition d'ontologies OWL. Son code source, libre, est écrit en Java, et il admet des extensions sous forme de *plugins*. Il existe de nombreux *plugins*, par exemple pour la visualisation des ontologies et pour l'édition des règles SWRL associées. La dernière version stable disponible actuellement est la 3.4.4, mais la version *bêta* 4.1 est disponible également. La version 4 constitue une refonte totale de l'éditeur. Notamment, elle est conforme au standard OWL 2.

Parmi les autres éditeurs d'ontologies, moins répandus, nous pouvons citer KAON2[2], Swoop[3] et Ontolingua[4].

APIs pour le traitement des ontologies OWL

Ces bibliothèques logicielles permettent un accès par programme aux ontologies OWL ; elles fournissent des fonctions qui permettent de créer une ontologie, de la lire, de créer le modèle en mémoire correspondant, de le modifier, de l'enregistrer, etc. La plupart de ces bibliothèques sont implantées avec le langage Java.

Les deux bibliothèques principales sont OWL API[5] et Protégé-OWL API. OWL API est l'implantation de référence pour la création, la manipulation et la sérialisation d'ontologies OWL. OWL-API est un projet libre mené par l'université de Manchester qui supporte pleinement OWL 2. Il fournit également plusieurs interfaces pour l'accès à des moteurs d'inférence d'une façon transparente. Protégé-OWL API est un *plugin* de Protégé qui est utilisé pour l'accès aux ontologies OWL dans les versions 3.4 de l'éditeur (à partir de la version 4.0, OWL API est utilisé à la place). Il peut être utilisé par des programmes externes comme une API Java, indépendamment de Protégé. Tout comme OWL-API, il permet l'accès à des moteurs d'inférence externes.

Moteurs d'inférence

Ces logiciels sont capables, par déduction, d'extraire des connaissances implicites contenues dans une ontologie. Ils peuvent s'exécuter comme une application indépendante ou être appelés depuis un autre programme, notamment les APIs mentionnées ci-dessus. Il existe des implantations propriétaires, telles que Bossam[6] ou RacerPro[7], ainsi que d'autres

[1] Disponible sur http://protege.stanford.edu/
[2] Disponible sur http://kaon2.semanticweb.org/
[3] Disponible sur http://www.mindswap.org/2004/SWOOP/
[4] Disponible sur http://ksl.stanford.edu/software/ontolingua/
[5] Disponible sur http://owlapi.sourceforge.net/
[6] Disponible sur http://bossam.wordpress.com/
[7] Disponible sur http://www.racer-systems.com/

libres comme Pellet[8], Fact++[9], KAON2 ou HermiT[10]. Parmi ces outils, Pellet est celui qui rencontre le plus de succès actuellement, grâce à ses performances, à ses fonctionnalités et à sa conception claire et simple.

Outils pour le traitement des règles SWRL

La plupart des moteurs d'inférence sont capables de traiter les règles SWRL ajoutées à une ontologie. Par exemple, Pellet implante de façon native un algorithme spécifique pour des règles DL-sûres dans OWL.

Afin de traiter les règles, il est également possible d'utiliser des moteurs spécifiquement dédiés aux règles, tel que le moteur Jess[11]. Ce moteur de règles possède un langage propre pour l'expression des connaissances sous forme de règles. Il peut être utilisé depuis Protégé (ou Protégé-OWL API) grâce à l'existence d'un pont qui permet de traduire un modèle d'ontologie dans le langage de Jess, d'exécuter les règles dans Jess et finalement de récupérer le résultat dans Protégé.

Choix d'outils

Ici, nous expliquons les choix que nous avons faits pour la création et le traitement d'ontologies et de règles OWL. Ces choix ont été utilisés pour la création de GCO ainsi que pour son utilisation (instanciation d'individus, raisonnement, traitement des règles, lecture d'individus) dans le cadre de Facus (voir chapitre suivant). Le Tableau 3.1 résume nos choix.

Outil/langage	**Choix**
Création/édition d'ontologies	Protégé 3.4.1
Création/édition de règles SWRL	*Plugin* SWRLTab pour Protégé
Accès par programme aux ontologies	Protégé-OWL API 3.4.1
Moteur d'inférence	Pellet 1.5.2
Moteur de règles	Jess 7.0
Version d'OWL	OWL 1

Tab. 3.1 – *Choix d'outils et de langages pour la création et le traitement de GCO*

Le choix retenu est le suivant : Protégé 3.4.1 pour la création et l'édition d'ontologies, avec le *plugin* SWRLTab pour l'édition des règles SWRL, Protégé-OWL API pour l'accès par programme aux ontologies, Pellet comme moteur d'inférence et Jess pour l'exécution des règles. Ce choix a été conditionné par le besoin d'utiliser certains *built-ins* SWRL qui sont disponibles seulement dans la version 3.4 de Protégé. Notamment, nous avions besoin du *built-in* expérimental `swrlx:createOWLThing`, qui permet de créer des nouveaux individus dans une règle SWRL (son utilisation est expliquée dans la section 3.3). Ce genre de *built-in* expérimental n'est pas disponible dans Pellet, ce qui a motivé l'utilisation de Jess pour le traitement des règles. Nous avons toutefois retenu Pellet pour les autres tâches de raisonnement telles que la vérification de la cohérence

[8] Disponible sur http://clarkparsia.com/pellet/
[9] Disponible sur http://owl.man.ac.uk/factplusplus/
[10] Disponible sur http://hermit-reasoner.com/
[11] Disponible sur http://www.jessrules.com/

de l'ontologie ou sa classification car il est plus complet et plus puissant que Jess pour effectuer ces tâches. L'accès à Pellet et à Jess depuis Protégé-OWL API est très simple grâce aux ponts spécifiques disponibles. Le choix de Protégé 3.4 et de Jess empêche l'utilisation de la version 2 d'OWL pour la description des ontologies et d'OWL-API pour leur traitement.

3.2 Description de GCO

Dans cette section nous présentons notre modèle de collaboration. Nous commençons par donner une vue d'ensemble des éléments du modèle, pour ensuite détailler la description du modèle sous forme d'ontologie OWL.

3.2.1 Vue d'ensemble du modèle

La notion fondamentale est celle de *session*. Une session représente la collaboration entre un ensemble d'entités qui veulent atteindre ensemble un but commun. Ces entités collaborent en s'envoyant des données à travers des *flux*, qui peuvent être de type *audio*, *vidéo* ou *texte*[12]. Chaque flux est envoyé par une entité vers une ou plusieurs entités. La session est composée de l'ensemble des flux envoyés. Un flux peut être implanté de façon synchrone ou asynchrone ; GCO n'impose pas un type de communication spécifique.

Une *entité* est soit un logiciel contrôlé directement par un humain, soit un logiciel autonome, comme par exemple un agent ou un composant. Dans les deux cas, l'entité a un *dispositif* associé dans lequel elle s'exécute. Les entités appartiennent à des *groupes*. Chaque entité a un *rôle* qui indique sa position logique dans le groupe ; en fonction de son rôle, une entité sera amenée à communiquer avec certaines entités et non pas avec d'autres. Le type de données échangées (audio, vidéo ou texte) dépendra également des rôles.

Afin de mettre en œuvre les flux de données, on a besoin d'outils. Un *outil* est donc un logiciel qui permet l'envoi et la réception de flux de données. Les outils sont composés d'un ou plusieurs *composants*, souvent un *composant d'envoi* et un *composant de réception* de données. Ce sont ces composants qui gèrent l'envoi et la réception des flux. Afin de gérer des flux d'une entité, le composant doit être déployé dans le même dispositif que l'entité.

Le modèle ne considère pas le contenu des flux ; c'est à chaque entité de décider ce qu'elle va envoyer. L'établissement des flux assure juste qu'une entité peut communiquer avec toutes les autres entités dont elle a besoin, en fonction de leurs rôles.

[12] Le type de données *texte* prend en compte le cas de communications entre composants logiciels qui suivent un protocole textuel, comme par exemple les Services Web.

3.2.2 Description détaillée

Les principaux éléments de GCO[13] sont présentés dans la Figure 3.3. Les concepts sont représentés avec des rectangles, tandis que les relations apparaissent comme des flèches allant d'un concept (le domaine de la relation) vers un autre (la portée de la relation). Les individus sont représentés avec des rectangles en ligne pointillée, et les types de données primitifs avec des rectangles en ligne hachurée. Les paragraphes suivants décrivent tous les éléments présents dans la figure.

Le premier concept de l'ontologie est celui de *nœud* (`Node`). Un nœud représente une *entité* qui participe à une activité collaborative. Cela peut être un humain (ou une entité logicielle contrôlée par un humain), mais aussi des entités logicielles autonomes telles que des agents, des composants, etc.

Les nœuds jouent un ou plusieurs rôles dans l'activité collaborative dans laquelle ils interviennent. Ces rôles déterminent la *position logique* de l'entité dans le groupe, c'est-à-dire, sa fonction et l'ensemble de nœuds avec lesquels elle peut communiquer afin de mener l'activité. Cela est modélisé par le concept `Role`. Le concept `Node` est relié au concept `Role` par une propriété `hasRole` qui exprime leur relation. Cette propriété n'est pas fonctionnelle, car comme nous avons dit, un nœud peut avoir plus d'un rôle.

Les rôles appartiennent à des groupes, qui sont représentés par le concept `Group`. L'appartenance des rôles aux groupes est exprimée par la relation `hasMember` (qui va de `Group` vers `Role`) et son inverse `belongsToGroup` (qui est fonctionnelle, car un rôle ne peut appartenir qu'à un groupe). La propriété `belongsToSameGroup` relie le concept `Role` avec lui-même. Si deux individus de `Role` sont reliés par cette propriété, cela veut dire qu'ils appartiennent au même groupe. Cette propriété est symétrique et transitive.

Les nœuds doivent être hébergés dans une plate-forme matérielle qui leur fournit des ressources d'exécution et de communication. Ces plate-formes sont modélisées par le concept `Device` (*dispositif* en français). La propriété `hasHostingDevice` de `Node` pointe donc vers `Device`. Elle est fonctionnelle, car un nœud donné ne peut être hébergé que par un seul dispositif à un moment donné. La propriété inverse s'appelle `hasHostedNode`. Les dispositifs sont identifiés par une chaîne de caractères unique, qui est représentée par la propriété `hasId` de `Device`, qui pointe vers le type primitif `string`. Cet identificateur peut représenter par exemple le nom du dispositif dans le réseau ou son adresse IP.

Les entités mettent en œuvre la collaboration en s'envoyant des données entre elles afin de se coordonner. Le concept `Flow` représente un *flux de données* unidirectionnel entre deux entités. En conséquence, ce concept est relié à celui de *nœud* par deux relations `hasSource` et `hasDestination`, indiquant, respectivement, les nœuds qui sont la source et la destination du flux. Puisque les flux sont définis comme étant unidirectionnels, pour implanter une communication bidirectionnelle deux flux de sens inverse sont nécessaires. La possibilité d'avoir des flux

[13] Le fichier OWL de GCO est disponible sur http://homepages.laas.fr/gsancho/ontologies/sessions.owl.

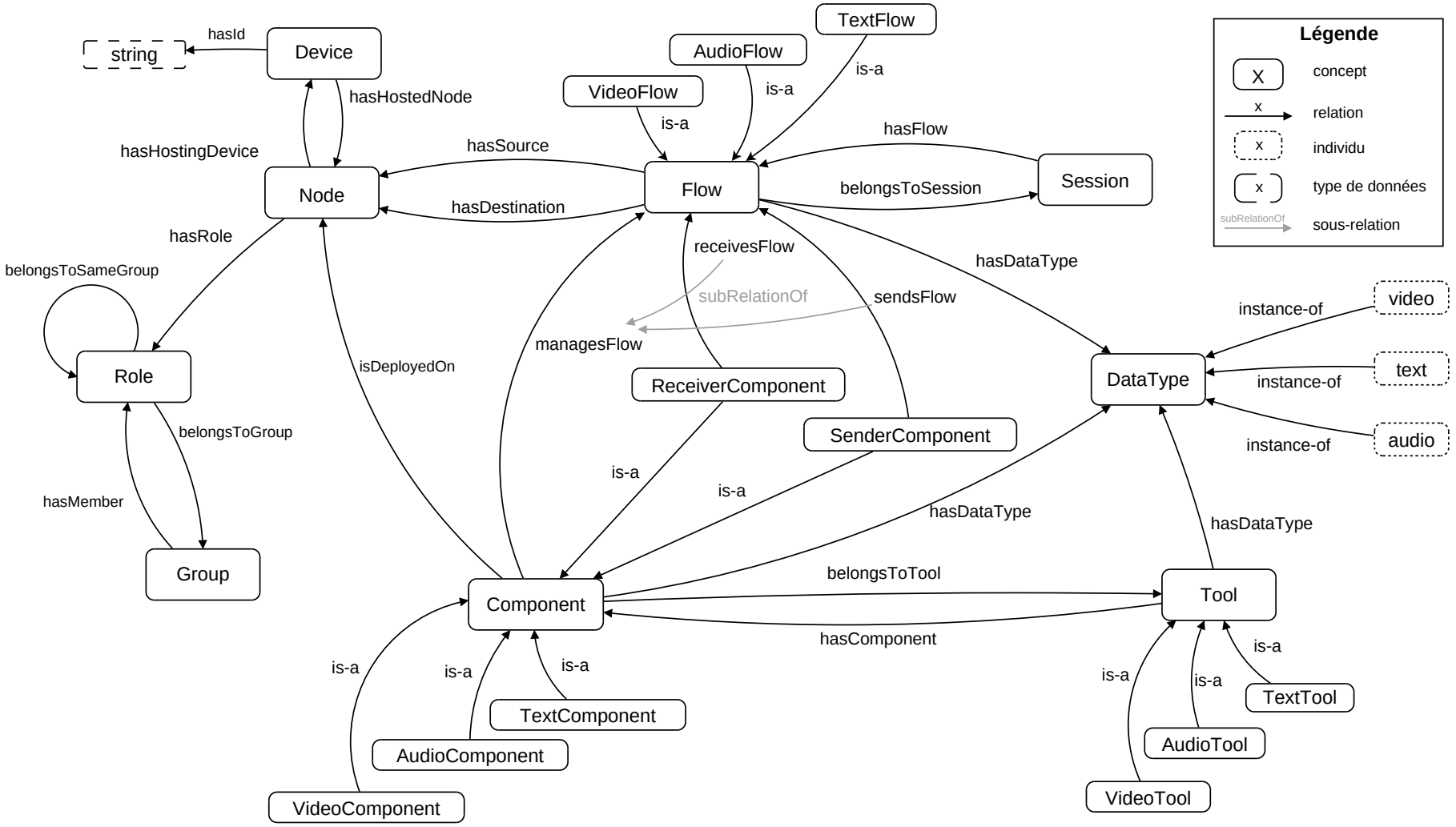


FIG. 3.3 – *Principaux concepts et relations de GCO*

multicast est prévue ; c'est pour cela que la propriété `hasDestination` n'est pas fonctionnelle (un flux peut avoir plusieurs nœuds de destination). En revanche, la propriété `hasSource` est fonctionnelle, car un flux a un et seulement un nœud comme source.

Les différents flux qui constituent une activité collaborative sont regroupés dans des *sessions*, capturées par le concept `Session`. Une session représente donc un ensemble de flux qui permet à des entités de communiquer autour d'un sujet ou d'un but spécifique. La relation `hasFlow` connecte le concept de flux à celui de session. La propriété inverse, `belongsToSession`, est fonctionnelle, car un flux appartient à une et seulement une session. Puisque les flux sont reliés aux nœuds et aux sessions, les nœuds sont indirectement reliés aux sessions auxquelles ils participent.

Les flux sont gérés par les nœuds avec des *outils* dédiés à la communication déployés dans le même dispositif, ce qui permet de séparer le code *métier* des nœuds (spécifique à l'application) du code collaboratif présent dans ces outils. Ces outils sont représentés avec le concept `Tool`. Les outils comprennent un ou plusieurs *composants* ; par exemple, un composant d'envoi et un composant de réception de données. Ces composants sont modélisés avec le concept `Component` (avec deux sous-concepts `SenderComponent` et `ReceiverComponent`), qui est relié à `Tool` par la relation `belongsToTool`. Cette relation est l'inverse fonctionnelle de `hasComponent`, fonctionnelle. La relation `hasComponent` a deux sous-relations `hasReceiverComponent` et `hasSenderComponent` selon le type du composant (nous n'avons pas inclus ces deux relations dans la figure afin qu'elle reste lisible).

Puisque les composants gèrent des flux, la relation `managesFlow` relie les concepts `Component` et `Flow`. Cette relation, ainsi que son inverse `isManagedBy`, n'est pas fonctionnelle car en principe un composant peut gérer plusieurs flux. Comme l'indique la Figure 3.3, `managesFlow` possède deux sous-relations `sendsFlow` et `receivesFlow` pour exprimer l'envoi et la réception des flux de la part des `SenderComponent` et des `ReceiverComponent`, respectivement. Les relations inverses s'appellent `isSentBy` et `isReceivedBy`, respectivement (nous n'avons pas inclus ces deux relations dans la figure afin qu'elle reste lisible). Les propriétés `receivesFlow` et `isReceivedBy` sont fonctionnelles, car un flux donné a un et seulement un composant qui le reçoit. La relation `isSentBy` est également fonctionnelle car un flux est envoyé par un et seulement un composant. En revanche `sendsFlow` n'est pas fonctionnelle car un composant peut envoyer plusieurs flux identiques à plusieurs récepteurs (ce qui peut être implanté avec un envoi *multicast*, par exemple).

La relation `isDeployedOn` relie les concepts `Component` et `Node` pour exprimer qu'un composant est déployé dans un nœud (et donc dans le dispositif qui héberge ce nœud). Cette propriété permet à GCO d'exprimer des schémas de déploiement. En effet, un ensemble d'individus qui représente des flux appartenant à une session, les composants qui les gèrent et les dispositifs dans lesquels ces composants sont déployés forment un schéma qui peut être utilisé pour implanter un déploiement qui soutient les sessions exprimées. Nous montrerons dans le chapitre

suivant comment GCO peut être exploité pour implanter de tels déploiements.

Les données gérées par les flux, les outils et les composants ont un *type* qui est modélisé par le concept `DataType`. Les différents types de données sont représentés par des individus appartenant à cette classe. Dans GCO, nous avons retenu les types audio, vidéo et texte (individus `audio`, `video` et `text`, respectivement), mais d'autres types peuvent être ajoutés dans des ontologies étendant GCO. Chacune des classes `Flow`, `Tool` et `Component` ont trois sous-classes définies selon le type de données. Ceci est exprimé avec des conditions nécessaires et suffisantes sur la propriété qui pointe vers type de données. Par exemple, la classe `AudioComponent` est définie comme :

$$AudioComponent \equiv Component \sqcap hasDataType(audio)$$

Cela veut dire que si un individu appartient à la classe `AudioComponent`, alors il est forcément un `Component` et sa propriété `hasDataType` pointe vers l'individu `audio`. Et inversement, tout individu étant un composant et dont le type de données est audio est forcément un `AudioComponent`. Nous avons utilisé le même principe pour les autres sous-classes de `Component` et pour celles de `Flow` et `Tool`.

La taxonomie des sous-classes de `Component` contient toutes les variantes selon le type de données (audio, texte, vidéo) et la direction du composant (envoi, réception). La Figure 3.4 détaille cette taxonomie telle qu'elle est représentée dans l'ontologie.

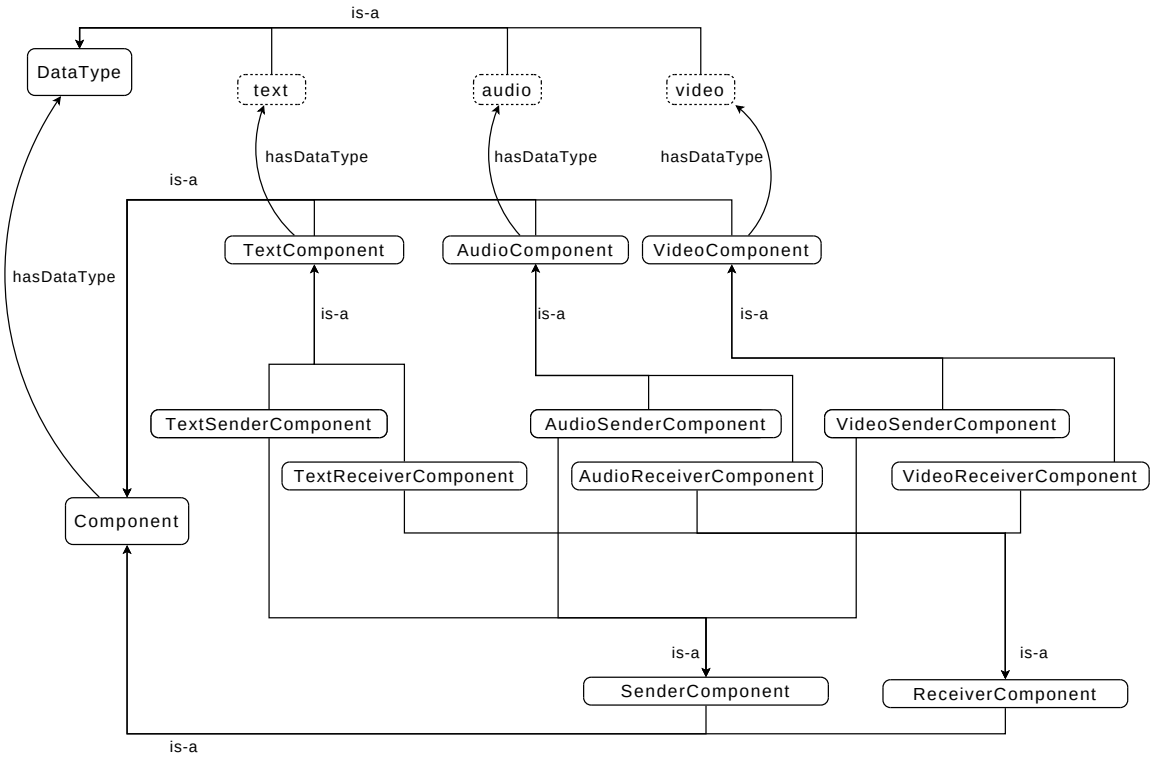


FIG. 3.4 – *Taxonomie des composants dans GCO*

3.3 Règles associées à GCO et raisonnement

Les concepts et relations de GCO constituent un modèle de la collaboration, c'est-à-dire, un canevas générique qui capture les situations de collaboration possibles. Les instances de ce modèle expriment des situations de collaboration concrètes. Ces instances sont des ensembles d'individus OWL appartenant aux concepts décrits dans GCO et liés par des instances des propriétés décrites dans GCO.

GCO et ses instances pourraient donc être utilisés comme des modèles *statiques* représentant des situations de collaboration concrètes. Bien que relevant un certain intérêt, cette utilisation n'exploite pas toutes les possibilités offertes par les ontologies. En effet, en plus de ce rôle de modèle, les ontologies offrent des capacités de traitement grâce au raisonnement et au traitement des règles SWRL. Dans cette section nous présentons, en premier lieu, les règles associées à GCO. Ensuite, nous expliquons comment exécuter ces règles et comment effectuer l'inférence sur GCO.

3.3.1 Règles associées à GCO

Nous avons ajouté des règles SWRL à GCO afin d'exprimer certaines relations, notamment celles qui permettent de déduire ou d'inférer un schéma de déploiement à partir des sessions présentes dans une instance de l'ontologie. Ces relations auraient été très difficiles, voire impossibles, à exprimer avec OWL seul. De plus, les *built-ins* fournis par SWRL sont très utiles pour implanter certains traitements, comme nous allons le voir par la suite.

Les trois premières règles de l'ontologie s'appellent, respectivement, `audio_flows_datatype`, `text_flows_datatype` et `video_flows_datatype` (figures 3.5, 3.6 et 3.7). Ces règles permettent de déduire le type de données d'un flux en fonction de la sous-classe de `Flow` à laquelle il appartient. Par exemple, la règle `audio_flows_datatype` relie tous les individus de la classe `AudioFlow` avec l'individu `audio` (appartenant à la classe `DataType`) à travers la relation `hasDataType`. Ceci veut dire que, lors du processus d'inférence, tout individu déclaré instance de la classe `AudioFlow` aura l'audio comme type de données. Les deux autres règles fonctionnent d'une manière analogue pour les flux de texte et de vidéo, respectivement. Ces trois règles permettent d'écrire les règles restantes, que nous allons voir par la suite, d'une façon générique, avec le type de données comme paramètre. Sinon, il aurait fallu faire trois versions des autres règles pour considérer les trois types de données. Si un autre type de données était ajouté, il suffirait de créer une quatrième règle analogue aux trois règles mentionnées pour le nouveau type de données. Ainsi, le nouveau type de données serait pris en compte automatiquement dans les autres règles.

```
AudioFlow(?f) → hasDataType(?f, audio)
```

Fig. 3.5 – *Règle* `audio_flows_datatype`

```
TextFlow(?f) → hasDataType(?f, text)
```

FIG. 3.6 – *Règle* `text_flows_datatype`

```
VideoFlow(?f) → hasDataType(?f, video)
```

FIG. 3.7 – *Règle* `video_flows_datatype`

La règle `same_group`, détaillée dans la Figure 3.8 infère que deux rôles appartiennent au même groupe. Cette relation simple ne peut pas être exprimée en OWL sans l'utilisation des règles. Cela évite aux utilisateurs de GCO de devoir expliciter `belongsToSameGroup` pour chaque paire de rôles appartenant au même groupe. Puisque cette relation est transitive et symétrique, le fait d'inférer que *a* appartient au même groupe que *b* implique automatiquement que *b* appartient au même groupe que *a*. Cette règle a été ajoutée dans GCO afin que les ontologies l'étendant puissent utiliser le concept `belongsToSameGroup` dans leurs règles sans avoir à déclarer explicitement toutes les instances de cette relation.

```
belongsToGroup(?r1,?g) ∧ belongsToGroup(?r2,?g)
→ belongsToSameGroup(?r1,?r2)
```

FIG. 3.8 – *Règle* `same_group`

La règle `components_manage_flow`, illustrée dans la Figure 3.9, permet de créer les composants nécessaires pour gérer les flux présents dans l'ontologie. En effet, pour chaque individu de la classe `Flow` trouvé, cette règle crée un `SenderComponent` qui envoie le flux (déployé dans le nœud source) et un `ReceiverComponent` qui reçoit le flux (déployé dans le nœud de destination). Les composants créés ont le même type de données que le flux qu'ils gèrent.

Pour créer les composants, le *built-in* SWRL `swrlx:createOWLThing` est utilisé. Dans la création du *sender*, les arguments de `createOWLThing` sont le nœud source et la session, tandis que, dans la création du *receiver*, on utilise le nœud de destination et le flux. La raison de ceci est la suivante : dans le cas des flux appartenant a la même session et dont l'émetteur est le même nœud, on veut un seul composant qui envoie ces flux aux différents récepteurs. Cette utilisation de `swrlx:createOWLThing` assure qu'un seul individu de la classe `SenderComponent` sera instantié par session et nœud source, tandis qu'il y aura autant de `ReceiverComponent`s que de nœuds de destination du flux.

Cette règle permet de faire le lien entre un schéma de collaboration (c'est-à-dire, un ensemble de sessions comprenant des flux envoyés entre des entités) et l'ensemble de composants qui permettent de gérer ces flux.

La règle `components_datatype`, détaillée dans la Figure 3.10 permet de déduire le type de données d'un composant à partir du type de données du flux géré par ce composant.

```
Flow(?f) ∧ belongsToSession(?f,?s)
∧ hasDataType(?f,?dt)
∧ hasSource(?f,?src) ∧ hasDestination(?f,?dst)
∧ swrlx:createOWLThing(?sc,?src,?s)
∧ swrlx:createOWLThing(?rc,?dst,?f)
→ SenderComponent(?sc) ∧ hasDataType(?sc,?dt)
∧ isDeployedOn(?sc,?src) ∧ sendsFlow(?sc,?f)
∧ ReceiverComponent(?rc) ∧ hasDataType(?rc,?dt)
∧ isDeployedOn(?rc,?dst) ∧ receivesFlow(?rc,?f)
```

FIG. 3.9 – *Règle* `components_manage_flow`

```
Component(?c) ∧ Flow(?f) ∧ managesFlow(?c, ?f)
∧ hasDataType(?f,?dt) → hasDataType(?c,?dt)
```

FIG. 3.10 – *Règle* `components_datatype`

3.3.2 Inférence et traitement des règles

L'exécution des règles présentées dans la section précédente sur une instance de GCO, ainsi que sa classification et son interrogation avec un moteur d'inférence, permettent d'exploiter l'information contenue dans cette instance.

Supposons que nous ayons une instance de l'ontologie qui exprime une certaine situation de collaboration. Cette instance contiendra des individus appartenant aux concepts de GCO, qui seront liés entre eux à travers les relations définies dans la GCO. L'exécution des règles permettra de :

1. Affecter à chaque individu des sous-classes de `Flow` son type de données (fait par les règles `audio_flows_datatype`, `text_flows_datatype` et `video_flows_datatype`).
2. Créer des individus de la classe `SenderComponent` et `ReceiverComponent`, représentant les composants qui permettent d'envoyer et de recevoir chaque flux. Ces composants seront liés par la relation `isDeployedOn` aux nœuds où ils sont déployés, et ils auront comme valeur de la propriété `hasDataType` un des individus de la classe `DataType` (`audio`, `video` ou `text`). Tout cela est fait par la règle `components_manage_flow`.

L'utilisation d'un moteur d'inférence permettra de :

1. Détecter s'il y a des incohérences dans les définitions des concepts, des relations, des individus, etc.
2. Obtenir (expliciter) des connaissances qui ont été renseignées d'une façon implicite. Ces connaissances peuvent être obtenues lors d'une interrogation du moteur d'inférence.

Exemple Considérons l'instance de GCO illustrée dans la Figure 3.11. Cet exemple est très simple, et il ne contient qu'une petite partie des éléments de GCO, mais il sert à illustrer comment s'effectuent le traitement

des règles et le raisonnement avec GCO. Dans cet exemple il y a deux individus de la classe `Node` (`node1` et `node2`), et un individu de la classe `AudioFlow` (`flow1`). Ce flux a comme source le nœud `node1`, et sa destination est `node2`. Nous avons également représenté l'individu `audio` de la classe `DataType`, qui est inclus dans GCO.

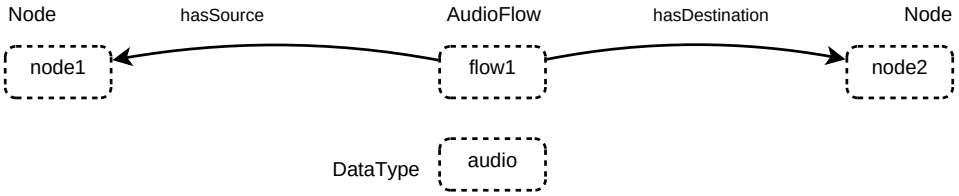


FIG. 3.11 – *Exemple de raisonnement et règles : situation initiale*

La Figure 3.12 montre l'état de l'ontologie après avoir exécuté les règles et ajouté les faits déduits par le raisonneur. Nous avons marqué à côté de chaque individu la classe à laquelle il appartient. En premier lieu, la règle `audio_flows_datatype` a déduit que `flow1` a `audio` comme type de données. Cela est pris en compte par la règle `components_manage_flow`, qui a ajouté un `SenderComponent` (`sc1`) dont le type de données est `audio`, et qui est déployé dans `node1`. Elle a également créé un `ReceiverComponent` (`rc1`) dont le type de données est `audio`, et qui est déployé dans `node2`. Le composant `sc1` envoie le flux et `rc1` le reçoit, comme l'indiquent les relations `sendsFlow` et `receivesFlow`. La règle `components_datatype` prend en compte le type de données de `flow1` afin d'affecter `audio` comme type de données des deux composants créés.

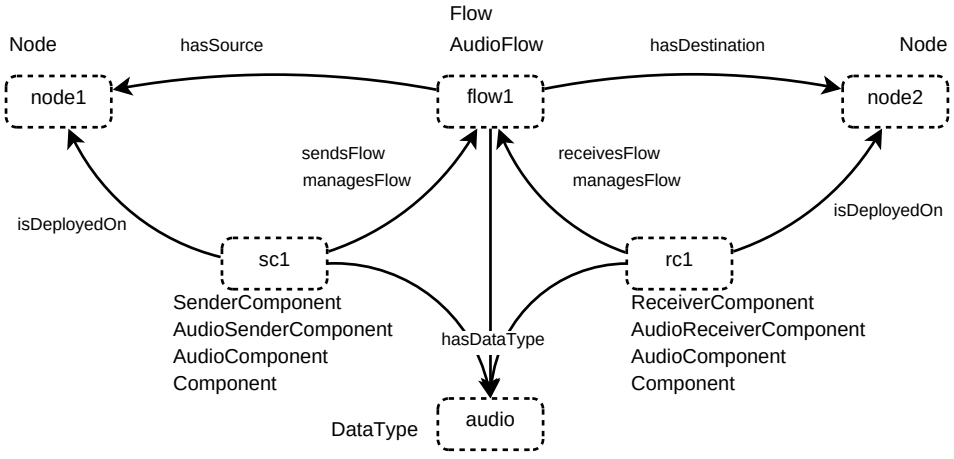


FIG. 3.12 – *Exemple de raisonnement et règles : situation finale*

Le raisonneur peut inférer que `flow1` est un `Flow`, car cette classe est une super-classe d'`AudioFlow`. Puisque `sc1` est un `SenderComponent` et qu'il a `audio` comme type de données, le raisonneur a déduit qu'il appartient également aux classes `AudioSenderComponent`, `AudioComponent` et `Component`. De façon analogue, il trouve que `rc1` appartient aux classes `AudioReceiverComponent`, `AudioComponent` et `Component`.

Le raisonnement porte aussi sur les propriétés. Par exemple, le raisonneur déduit que `sc1` gère le flux `flow1` (propriété `managesFlow`), car la propriété `sendsFlow` est une sous-propriété de la propriété `managesFlow`. De la même façon, `rc1` est lié à `flow1` par la propriété `managesFlow`.

Le raisonneur trouve aussi toutes les relations inverses, que nous n'avons pas inclues dans la figure afin qu'elle reste lisible ; par exemple `flow1` est relié à `sc1` par `isManagedBy` et `isSentBy`, et à `rc1` par `isManagedBy` et `isReceivedBy`.

Bien évidemment, si l'instance de GCO utilisée inclut des individus, des concepts, des relations ou des règles appartenant à d'autres ontologies (par exemple dans le cadre d'une ontologie qui étend GCO pour un domaine particulier), le traitement des règles et le raisonnement porteront également sur ces éléments additionnels.

3.4 Principes de conception et propriétés de GCO

La conception d'une ontologie est un processus dans lequel beaucoup de choix doivent être faits. Dans cette section nous présentons les principes de conception qui nous ont guidés dans ces choix lors de la conception de GCO, et nous soulignons les propriétés qui en résultent.

3.4.1 Langage

Comme nous l'avons expliqué dans la sous-section 3.1.4, nous avons retenu OWL 1 pour exprimer GCO. La variante d'OWL que nous avons retenue est OWL-DL car elle offre un bon compromis entre l'expressivité offerte et le fait qu'elle possède les propriétés de complétude et de décidabilité. Ces propriétés permettent son utilisation dans des implantations qui utilisent des services de raisonnement. Nous avons également bénéficié de l'expressivité apportée par SWRL afin d'exprimer des règles d'inférence et des relations qui auraient été difficiles, voire impossibles, à exprimer avec OWL-DL seul.

3.4.2 Contenu de l'ontologie

Le but essentiel de GCO est la modélisation des sessions collaboratives de haut niveau, synchrones ou asynchrones. Nous avons donc choisi le contenu de cette ontologie afin de représenter les éléments nécessaires pour cette modélisation : sessions, flux, rôles, etc. Beaucoup de ces éléments se trouvent dans les modèles de session classiques.

Le deuxième but que nous avons poursuivi dans le choix des éléments de GCO est le déploiement. En effet, l'idée est que les schémas de collaboration exprimés avec les instances de GCO puissent être utilisés pour le déploiement des composants nécessaires pour leur réalisation. Pour cela, nous avons ajouté des éléments, tels que les outils ou les composants, qui permettent d'indiquer avec quels moyens on gère les échanges de données, et sur quelle machine ils doivent être déployés.

La transition entre le schéma de collaboration exprimé explicitement par une instance de GCO et le schéma de déploiement correspondant est

fait par le processus d'inférence et de traitement des règles. Ce fait a influencé notre façon de définir les règles et les éléments de l'ontologie afin de faciliter le raisonnement.

3.4.3 Généricité et extensibilité

GCO a été conçu pour être le plus générique possible par rapport au domaine d'utilisation. Cela veut dire qu'il peut être utilisé pour modéliser la collaboration dans n'importe quelle application, indépendamment du domaine auquel elle appartient. Ceci est possible car les concepts et les relations présents dans cette ontologie représentent un ensemble minimal (le dénominateur commun) indépendant de tout domaine, et ils ne concernent que la collaboration en elle-même. Les éléments concernant des domaines spécifiques ont été exclus de l'ontologie. Ce principe a également été appliqué à la conception des règles associées à l'ontologie.

La généricité de GCO implique qu'il doit être étendu pour s'adapter au choix d'un domaine spécifique. Par exemple, dans une application de télé-enseignement, cette adaptation consistera à dériver la notion de rôle, générique, en deux notions *professeur* et *élève*, qui correspondent au domaine spécifique. Cette adaptation se fait par la création d'une deuxième ontologie qui importe GCO et qui définit, entre autres, des concepts et des relations qui héritent de ceux qui se trouvent dans GCO. Cela est possible grâce au mécanisme d'importation d'une ontologie externe, tel que nous l'avons décrit dans la section 3.1. En effet, ce mécanisme rend visibles tous les concepts et les relations de GCO depuis l'ontologie qui l'importe. Dans l'exemple de l'ontologie de télé-enseignement, elle importera GCO et elle définira les concepts *étudiant* et *professeur* comme des cas particuliers (ou sous-concepts) du concept *rôle* de GCO. On peut également déclarer des sous-relations à partir des relations de GCO. Un exemple d'ontologie de domaine étendant GCO peut être trouvé dans la sous-section 5.3.1.

Cette extensibilité permet également de prendre en compte des descriptions plus riches des dispositifs. Dans GCO, la seule propriété des dispositifs est leur identifiant. Pour des applications qui ont besoin d'une description plus riche, par exemple avec le type de CPU, la quantité de mémoire, le type de système d'exploitation, etc., elles peuvent être ajoutées au niveau de l'ontologie spécifique de l'application.

Le cas où une deuxième ontologie importe GCO est équivalent au schéma classique dans lequel il existe une ontologie *de haut niveau* et une ontologie *de domaine* qui l'étend, et qui a été évoqué dans la section 3.1.

3.4.4 Utilisation dans des architectures multi-niveaux

La généricité de GCO et le mécanisme d'extension décrits ci-dessus rendent cette ontologie apte à être utilisée dans des architectures à plusieurs niveaux où elle serait utilisée comme le modèle de référence du niveau collaboration. Dans des niveaux plus spécifiques, on pourrait trouver des ontologies spécifiques au domaine qui étendent GCO. De cette façon, il est possible d'avoir plusieurs niveaux, chacun plus spécialisé que les précédents et sur lesquels il repose. Cette séparation facilite la sépara-

tion de responsabilités dans les niveaux, ce qui résulte en une conception claire et modulaire et qui favorise la réutilisation du code.

Dans notre *framework* FACUS, qui sera présenté dans le chapitre suivant, nous avons mis en œuvre ce principe de conception multi-niveaux où GCO joue un rôle central comme modèle de niveau collaboration.

3.4.5 Simplicité et performances

Bien que GCO permette de modéliser des sessions collaboratives de plusieurs domaines et d'effectuer des déploiements pour implanter ces sessions, il reste assez simple. GCO ne regroupe qu'un petit nombre d'éléments indispensables : le nombre de concepts est de 27, tandis que le nombre de relations est de 22. Ceci a deux avantages. Premièrement, l'ontologie est abordable par des concepteurs voulant l'utiliser pour leurs applications ; elle peut être facilement appréhendée et étendue pour exprimer des connaissances collaboratives spécifiques au domaine considéré. Deuxièmement, les instances de cette ontologie exprimant des situations de collaboration concrètes restent relativement simples et légères par rapport au nombre de participants, ce qui fait que les traitements effectués (*parsing*, raisonnement, traitement des règles) restent assez performants. Dans la section 5.4 nous présentons les résultats de l'évaluation des performances de ces traitements.

3.4.6 Nomenclature

Afin d'avoir une nomenclature claire et cohérente qui facilite la compréhension de GCO, nous avons suivi les principes suivants pour nommer les éléments de cette ontologie :

1. Les noms des éléments sont en anglais.
2. Quand le nom d'un élément contient plusieurs mots, tous les mots sont collés, et chaque nouveau mot commence par une majuscule.
3. Les noms des concepts commencent par une majuscule. Exemple : `VideoFlow`.
4. Les noms des relations commencent par une minuscule. Le premier mot est un verbe et le deuxième est un complément du verbe. Le sujet du verbe est le concept duquel part la relation, tandis que le concept sur lequel elle arrive est la valeur que prend le complément. Exemple : la relation `hasSource` va de `Flow` vers `Node`, ce qui veut dire qu'un flux a comme source un nœud.
5. Les noms des individus et des types de données sont écrits en minuscule. Exemple : `audio`, `string`.

3.4.7 Autres principes suivis

Quelques principes supplémentaires nous ont guidés dans la création de GCO :

- concernant le choix de l'inclusion d'individus dans l'ontologie, dans la plupart des concepts de GCO, nous nous sommes arrêtés au niveau du concept, sans inclure des individus. Les individus seront

utilisés dans les instances de GCO pour exprimer des situations de collaboration particulières. La seule exception est celle du concept `DataType` qui représente les types de données, qui contient trois individus `audio`, `video` et `text`. Cette exception repose sur deux raisons. La première est que ces sous-concepts sont atomiques ; audio, vidéo et texte ne peuvent pas être décomposés en une granularité plus fine. La deuxième raison, plus importante, est que le fait d'avoir des individus pour les types de données facilite l'écriture des règles qui utilisent le type de données. Plus particulièrement, nous pouvons nommer directement le type de données concerné comme valeur d'une relation. Par exemple, on peut écrire `hasDataType(audio)` pour restreindre le traitement seulement aux individus qui ont le type de données audio. Ceci aurait été plus difficile avec les concepts.

- Nous avons souvent utilisé les propriétés inverses pour avoir deux « directions » dans les relations. Ceci facilite l'écriture des règles et la définition des restrictions des classes.
- Pour chaque ensemble de concepts qui héritent du même concept, nous les avons définis comme disjoints explicitement quand ceci est possible, afin de faciliter le raisonnement et l'exécution des règles.
- Afin que le raisonnement et les règles fonctionnent correctement, il faut déclarer explicitement que les individus sont différents. Autrement, puisque OWL n'applique pas l'hypothèse du nom unique, ils ne seront pas considérés comme des individus distincts. Le plus simple est d'inclure les individus dans la construction `owl:allDifferent` pour indiquer qu'ils sont tous différents entre eux.

3.5 Utilisation de GCO dans un système *runtime*

Une application peut utiliser des instances de GCO pour représenter l'état de la collaboration à un moment donné. Comme nous l'avons expliqué dans la section 3.1, l'instance de l'ontologie peut être interrogée et modifiée par le code d'une application en utilisant les différents APIs et outils disponibles. Il est également possible de la traiter avec un moteur d'inférence afin d'expliciter des informations implicites. Ainsi, l'application peut utiliser l'ontologie non seulement comme un modèle statique représentant des données (à travers les individus présents et les relations qui existent entre eux), mais elle peut aussi bénéficier des capacités de traitement offertes par le raisonnement et le traitement des règles.

Cette possibilité est très intéressante ; cependant, la nature monotonique du raisonnement avec OWL fait émerger un problème. Comme nous l'avons indiqué dans la section 3.1, le fait qu'OWL soit monotone implique que tout fait présent dans une ontologie ne peut pas être enlevé à posteriori par le raisonnement ou par le traitement des règles. Les nouveaux faits peuvent ajouter de l'information, mais pas en détruire. Il existe des alternatives à OWL qui éliminent cette limitation. Par exemple, OWL++, une extension d'OWL proposée par Hosain et Jamil [HJ09], ajoute l'inférence non-monotonique à OWL. D'autres langages logiques, comme l'*Event Calculus* [Sha99], modélisent explicitement l'aspect temporel des

connaissances, ce qui facilite la prise en compte de leur évolution dans le temps. Cependant, ces alternatives n'ont pas encore connu une diffusion similaire à celle d'OWL, notamment en termes de standardisation et d'outils qui les supportent.

Avec OWL, la seule façon de retirer de l'information serait de changer l'ontologie en utilisant une des API pour en obtenir une nouvelle. Cela reviendrait à faire marche arrière dans la base de connaissances, à revenir jusqu'au moment où on a exprimé un certain fait et puis à changer la suite. Mais cela crée un problème dans le cas où l'on utilise du raisonnement (et aussi dans celui de l'exécution de règles), car tout ce qui a été déduit serait effacé. Si un moteur d'inférence a déduit qu'un fait *a* est vrai, il n'est pas envisageable qu'il décide plus tard qu'il est faux si les conditions ont changé. Dans ce cas là, le fait *a* serait *vrai et faux en même temps*. De la même façon, avec GCO, si à un moment donné il existe un flux qui va du `node1` au `node2`, les règles créeront des composants déployés dans ces deux nœuds qui gèrent le flux. Même si on efface le flux en utilisant une API pour modifier le modèle en mémoire correspondant, les individus qui représentent les composants créés par les règles existeront toujours dans l'ontologie. Encore plus grave, si l'on exécute la règle deux fois, ces composants seront créés deux fois. Il n'est pas possible d'écrire une règle qui détruirait ces composants. Il serait possible d'effacer ces individus avec une API qui modifie l'ontologie, mais dans la pratique il peut être très long et fastidieux de maintenir un registre avec toute l'information qui a été inférée et de décider ce qui doit être effacé ou non à chaque instant.

La solution que nous proposons ici consiste à utiliser les capacités d'inférence d'OWL dans une boucle *capture–inférence–résultats*, tel qu'illustré dans la Figure 3.13. La première étape de la boucle consiste à capturer l'*état du monde*, c'est-à-dire l'ensemble des informations que l'on veut représenter. Dans notre cas, cet état inclut les utilisateurs présents, leurs rôles, les flux qu'ils s'envoient, etc. Cette capture est faite par le code de l'application, qui représente ensuite l'état capturé dans une instance de GCO. Une telle instance est un ensemble d'individus qui appartiennent aux concepts de GCO et qui sont liés par des instances des relations définies dans GCO. Cette instance est utilisée par l'application comme modèle représentant son état à un moment donné.

L'application peut exécuter les règles associées à l'ontologie et effectuer des raisonnements sur l'ontologie avec un moteur d'inférence. Le résultat est une instance modifiée de l'ontologie contenant un nouveau groupe d'individus. Cette instance contient des informations additionnelles ; par exemple, dans le cas de GCO, des composants qui gèrent les flux existants ont été ajoutés, ils sont déployés sur certains nœuds, etc. Cette nouvelle instance contient en fait le nouvel *état du monde* que l'application doit mettre en œuvre.

Ensuite, l'application parcourt la nouvelle instance et effectue les actions nécessaires pour arriver à l'état souhaité. Dans le cas de GCO, elle va trouver tous les individus qui correspondent aux composants et elle va déployer les composants réels dans les machines correspondantes.

À cause du problème de monotonicité évoqué précédemment, quand un changement dans l'*état du monde* survient (par exemple quand un

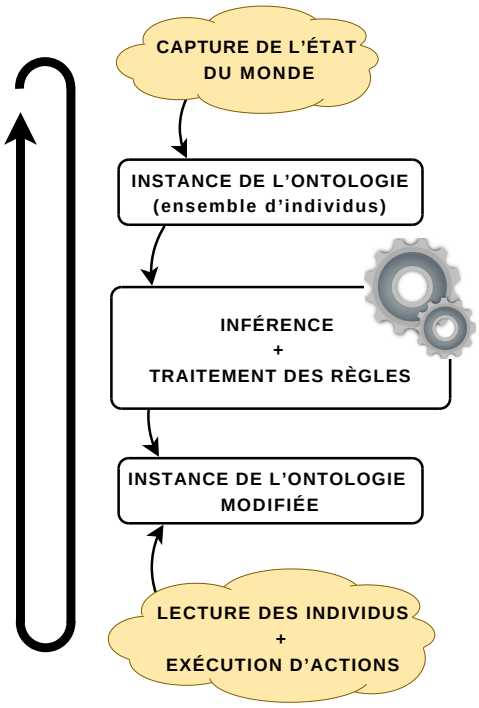


FIG. 3.13 – *Boucle* capture–inférence–résultats *proposée pour l'utilisation de GCO dans un système* runtime

nouvel utilisateur arrive), la boucle entière est répétée afin d'obtenir les nouveaux résultats. L'utilisation de la boucle est discrète : lorsque les entrées changent, tout est re-exécuté afin d'obtenir les nouveaux résultats. Dans l'étape n, l'instance obtenue lors de l'étape $n-1$ est écartée, et *une nouvelle instance est construite*. Si OWL supportait le raisonnement non-monotonique, on aurait pu modifier directement l'instance existante et avoir des règles permettant de suivre l'évolution.

Conclusion du chapitre

L'Ontologie Générique de la Collaboration (GCO) est le modèle central de notre proposition. Dans la première partie de ce chapitre, nous avons introduit les langages et les technologies de conception et de traitement de ce modèle : le langage d'ontologies OWL, le langage de règles SWRL, l'inférence, ainsi que les logiciels qui permettent de les gérer. Nous avons fourni des exemples qui permettent de comprendre comment représenter des connaissances et comment effectuer des inférences avec ces technologies issues du Web Sémantique.

Ensuite, nous avons détaillé les éléments modélisés dans GCO, ainsi que les règles associées. Nous avons montré comment les règles et le raisonnement fonctionnent sur une instance d'exemple de GCO. Nous avons détaillé ensuite les principes de conception et les propriétés qui justifient les choix que nous avons faits pour la création de GCO. Finalement, nous avons expliqué comment GCO peut être exploité dans un système *runtime*, notamment pour profiter du raisonnement et du traitement des règles.

Comme nous l'avons vu, GCO est un modèle de la collaboration qui

permet de représenter les utilisateurs, leurs rôles, et la façon dont ils s'échangent des données pendant des sessions. Ce modèle est générique car les détails dépendant des domaines particuliers ne sont pas modélisés. Cela peut être fait par d'autres ontologies qui utilisent GCO comme modèle de base de la collaboration et l'étendent pour modéliser des relations dans des domaines ou des applications concrètes. Le fait d'utiliser des langages et technologies standard facilite l'utilisation de GCO comme modèle commun de référence pour la collaboration dans des implantations réelles. GCO peut être utilisée également pour déduire un schéma de déploiement à partir d'une situation de collaboration donnée. Dans le chapitre suivant, nous présenterons l'utilisation de GCO comme modèle central de la collaboration dans un système concret, le *framework* FACUS.

4 Facus : un *Framework* Adaptatif pour les Systèmes Collaboratifs Ubiquitaires

Sommaire

Ce chapitre est consacré à notre apport pour la conception de systèmes collaboratifs ubiquitaires (SCU). Cet apport est divisé en deux parties. Tout d'abord, nous présentons notre cadre conceptuel pour la conception des SCUs, qui est basé sur une approche de modélisation à plusieurs niveaux [BSV+09]. Le but principal de cette approche est de modéliser l'architecture du système à considérer et de permettre l'adaptation de cette architecture aux changements du contexte dans lequel le système évolue. Nous appliquons cette approche générique au cas des SCUs en identifiant les niveaux de modélisation significatifs pour ce type de systèmes et en détaillant les éléments à modéliser dans chaque niveau.

La deuxième partie de notre apport consiste en une implantation de notre approche de modélisation sous la forme de *framework* que les concepteurs de SCUs peuvent utiliser comme guide de modélisation et

comme plate-forme d'exécution pour leurs applications. Cette implantation s'appelle *Framework for Adaptive Collaborative Ubiquitous Systems* (FACUS) [SBT+09]. Nous détaillons les modèles architecturaux concrets que nous avons utilisés dans FACUS, dont fait partie l'ontologie GCO présentée dans le chapitre précédent. Ensuite, nous présentons la façon dont FACUS a été conçu, son architecture et les détails de son implantation. Nous expliquons également comment FACUS peut être utilisé dans un SCU et nous fournissons une explication de son fonctionnement en ligne, notamment au cours du processus d'adaptation au contexte.

4.1 Approche d'adaptation multi-niveaux pour les systèmes collaboratifs ubiquitaires

Dans cette section nous présentons notre approche de modélisation générique qui met en œuvre une adaptation multi-niveaux, ainsi que son instanciation pour les systèmes collaboratifs ubiquitaires. Cette approche est la base théorique de Facus, mais sa généricité la rend apte à être utilisée dans d'autres *frameworks* ou applications qui utilisent d'autres modèles que ceux retenus dans Facus.

4.1.1 Approche multi-niveaux générique

Notre approche de modélisation prend en compte l'architecture d'un système logiciel, car notre but est de mettre en œuvre une adaptation architecturale. Afin de séparer clairement les différents problèmes à gérer et de simplifier la conception des systèmes, l'approche prévoit plusieurs niveaux de modélisation. Ainsi, les niveaux supérieurs peuvent faire abstraction des détails et de la complexité contenus dans les niveaux inférieurs, sur lesquels ils reposent.

Les niveaux supérieurs modélisent des éléments de haut niveau, proches des activités humaines, tandis que les niveaux inférieurs représentent des détails de bas niveau, plus proches des implantations physiques qui soutiennent ces activités. Cette séparation permet de mettre en œuvre une *adaptation multi-niveaux*, c'est-à-dire, qui tient compte des exigences des utilisateurs (dans les niveaux supérieurs) et des contraintes imposées par les ressources disponibles (dans les niveaux inférieurs).

Puisqu'ici nous parlons de l'approche générique, le nombre de niveaux n'est pas déterminé. Nous utilisons n pour désigner un niveau, avec n appartenant à $\mathbb{N}$.

À chaque niveau est associé un modèle, qui encapsule les éléments architecturaux à modéliser. Chaque instance du modèle représente donc une configuration architecturale, c'est-à-dire, un ensemble d'entités logicielles distribuées sur un ou plusieurs nœuds d'exécution et reliées entre elles avec des liens de communication. Dans la suite, nous utiliserons le terme *configuration architecturale* (ou simplement *architecture*) pour faire référence à ces instances du modèle d'un niveau. À chaque instant, une seule configuration parmi toutes les configurations possibles est retenue comme instance courante du niveau.

Une configuration architecturale est dénotée $A_{n,i}$, où n est le niveau d'abstraction considéré ($n \in \mathbb{N}$) et i est un index permettant de distinguer les configurations ($i \in \mathbb{N}$). Toute configuration $A_{n,i}$ est une instance du modèle de niveau n. Pour chaque configuration architecturale $A_{n,i}$ de niveau n, il existe plusieurs configurations architecturales $(A_{n-1,p}, \ldots, A_{n-1,q})$ qui l'*implantent* au niveau $n-1$. C'est-à-dire, ces configurations représentent un raffinement ou une vue détaillée de $A_{n,i}$ (voir Figure 4.1), dans le sens qu'elles contiennent un ensemble d'éléments de niveau $n-1$ cohérents avec les éléments de niveau n contenus dans $A_{n,i}$.

L'adaptation du système aux exigences haut niveau et aux contraintes bas niveau se fait par le biais des configurations architecturales retenues à chaque niveau. Quand l'adaptation impose un changement au niveau n,

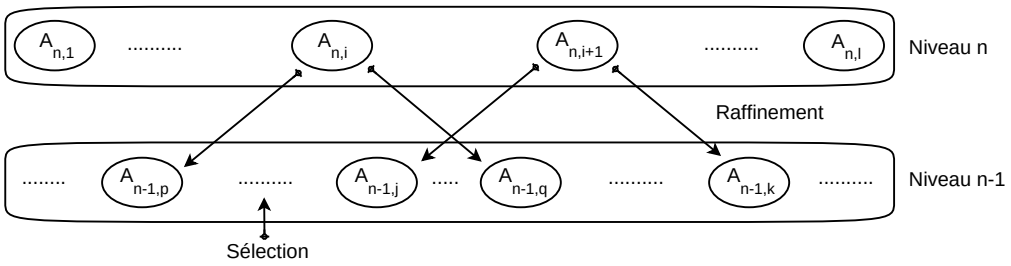


FIG. 4.1 – *Approche générique de modélisation multi-niveaux*

une configuration architecturale est choisie parmi toutes celles du niveau $n-1$ qui implantent la nouvelle configuration de niveau n. Si l'ancienne et la nouvelle configuration retenues au niveau $n-1$ sont équivalentes, cela veut dire que la nouvelle configuration de niveau n peut être implantée par la configuration courante de niveau $n-1$. Par conséquent, l'adaptation au changement peut s'arrêter là, et les niveaux inférieurs ne seront pas modifiés. Sinon, il faut continuer à choisir de nouvelles implantations dans les niveaux $n-2$, $n-3$, ..., $n-k$, ...

Cette technique d'adaptation requiert deux actions différentes : le *raffinement* et la *sélection*. Nous expliquons par la suite ces deux actions.

Raffinement et sélection

Le *raffinement* permet de calculer, à partir d'une configuration de niveau n, l'ensemble des configurations de niveau $n-1$ qui l'implantent. Cette action permet de faire la transition d'un niveau vers le niveau inférieur immédiat. La Figure 4.2 représente la version générique de cette fonction. Cette fonction est générique car les actions associées au mot *implante* (ligne 2) seront différentes pour chaque application concrète et pour chaque niveau.

```
1 Refine(A_{n,i}){
2 Retourner 𝔸^i_{n-1} = {A_{n-1,j} ∈ 𝔸_{n-1} tel que A_{n-1,j} implante
      A_{n,i}, j ∈ ℕ}
3 }
```

FIG. 4.2 – *Algorithme générique de raffinement*

Le résultat du raffinement est un ensemble de configurations qui peuvent être mises en place au niveau inférieur. Ces configurations sont des « candidats » à être la nouvelle configuration retenue dans ce niveau. La fonction de *sélection* permet de choisir *le meilleur* de ces candidats, en fonction de plusieurs paramètres.

La Figure 4.3 illustre la fonction générique de sélection. Les paramètres d'entrée de cette fonction sont l'ensemble des candidats ($\mathbb{A}_n^p$) et une politique (*Policy*). L'algorithme fonctionne de la façon suivante : en premier lieu, on calcule la valeur que la fonction *Context_Adaptation()* affecte à chacun des candidats. Les candidats ayant la plus grande valeur (positive)

```
1  Select(𝔸_n^p, Policy){
2  Soit S_1 = {A_{n,k} ∈ 𝔸_n^p, k ∈ ℕ tel que
       Context_Adaptation(A_{n,k}) ≥ 0 et
       Context_Adaptation(A_{n,k}) ≥ Context_Adaptation(X), ∀X ∈ 𝔸_n^p}
3  Si card(S_1) = 0
4    retourner ∅
5  sinon si card(S_1) = 1
6    retourner le premier élément de S_1
7  sinon
8    Si Policy = distance
9      Soit A_{n,i} l'implantation courante au niveau n
10     Soit S_2 = {A_{n,k} ∈ 𝔸_n^p, k ∈ ℕ tel que
       Distance(A_{n,i}, A_{n,k}) ≤ Distance(A_{n,i}, X), ∀X ∈ S_1}
11   sinon si Policy = dispersion
12     Soit S_2 = {A_{n,k} ∈ S_1, k ∈ ℕ tel que
       Dispersion(A_{n,k}) ≥ Dispersion(X), ∀X ∈ S_1}
13   sinon si Policy = ...
14     ...
15   Si card(S_2) = 1
16     retourner le premier élément de S_2
17   sinon
18     Sélectionner aléatoirement une configuration de S_2
19 }
```

FIG. 4.3 – *Algorithme générique de sélection*

sont introduits dans un ensemble appelé S_1. Si S_1 ne contient aucune configuration, cela veut dire qu'aucun des candidats n'est compatible avec le contexte actuel. Dans ce cas, la fonction *Select()* retourne l'ensemble vide. Si S_1 contient un seul candidat, ce candidat est retourné. Sinon, la politique fournie en entrée est utilisée pour sélectionner le meilleur candidat parmi les configurations de S_1. Ici, nous avons considéré deux politiques, mais nous pouvons facilement en ajouter d'autres selon les besoins de l'application. La première politique, appelée *distance*, utilise une fonction *Distance()* qui affecte une valeur à chaque candidat en fonction de sa distance (par exemple en nombre de redéploiements à faire) avec la configuration courante. La deuxième politique, appelée *dispersion*, utilise une fonction *Dispersion()* qui affecte une valeur à chaque candidat selon le degré de dispersion des composants sur les nœuds. Dans les deux cas, les candidats qui ont la meilleure valeur sont introduits dans un ensemble S_2. Finalement, si S_2 contient un seul candidat, il est retourné en sortie ; s'il y en a plusieurs, cela veut dire qu'ils ont tous les mêmes propriétés par rapport aux politiques, et alors l'un d'entre eux est choisi aléatoirement.

Cette fonction de sélection est générique car la fonction *Context_Adaptation()* n'est pas détaillée ; chaque application doit fournir l'algorithme utilisé dans cette fonction. Cette fonction doit tenir compte des caractéristiques de chaque configuration et doit lui affecter une valeur selon le degré d'*adaptation au contexte* qu'elle présente. Les configurations qui sont bien adaptées au contexte (par exemple celles qui ont les compo-

sants les plus gourmands en ressources dans les nœuds les plus puissants) recevront des valeurs grandes et vice-versa. Les configurations incompatibles avec le contexte actuel sont impossibles à implanter. Elles auront donc -1 comme valeur. Les fonctions *Distance()* et *Dispersion()*, ainsi que des politiques additionnelles éventuelles, sont également à définir selon les besoins des applications. Nous détaillerons nos choix pour FACUS dans la sous-section 4.2.2.

L'action conjointe du raffinement et de la sélection met en œuvre une *adaptation multi-niveaux* qui peut répondre à des changements de contexte à plusieurs niveaux. Quand un changement de contexte se produit dans un niveau, une *adaptation horizontale* ou *intra-niveau* est effectuée à travers une action de sélection permettant de choisir le meilleur candidat, selon la fonction *Context_Adaptation()* et les politiques, au niveau considéré. Si le candidat retenu est différent du précédent, alors une *adaptation verticale* ou *inter-niveaux* est faite avec le raffinement de la nouvelle configuration vers les niveaux inférieurs. De cette façon, les exigences qui proviennent des niveaux hauts, qui prennent en compte le contexte externe, se propagent vers le bas (avec le raffinement), mais à chaque niveau les contraintes imposées par le contexte des ressources sont également prises en compte (avec la sélection).

Exemple Considérons l'exemple illustré par la Figure 4.4. Dans cet exemple nous avons trois niveaux. Les niveaux 3 et 1 ont quatre configurations possibles, tandis que le niveau 2 contient seulement trois configurations. Les flèches de la figure montrent les raffinements possibles de chaque configuration.

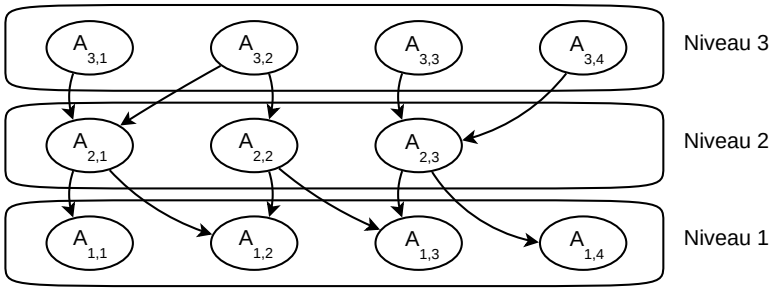


FIG. 4.4 – *Exemple d'adaptation multi-niveaux*

Imaginons qu'à un moment donné les configurations retenues aux niveaux 3, 2 et 1 sont $A_{3,4}$, $A_{2,3}$ et $A_{1,4}$ respectivement. Si un changement de contexte est détecté au niveau 3, une nouvelle configuration sera retenue. Si la configuration retenue est, par exemple, $A_{3,3}$, alors l'adaptation s'arrêtera là, car son raffinement au niveau 2 est $A_{2,3}$, qui est la configuration courante. Dans ce cas, on a une adaptation intra-niveau.

Imaginons qu'un nouveau changement de contexte est détecté au niveau 3, et que la nouvelle configuration retenue est $A_{3,2}$. Dans ce cas, il faut choisir une nouvelle configuration au niveau 2, car les raffinements au niveau 2 de $A_{3,3}$ et de $A_{3,2}$ n'ont pas de configurations communes. La nou-

velle configuration retenue au niveau 2 est, par exemple, $A_{2,2}$, tandis qu'au niveau 1 $A_{1,2}$ est retenue. Dans ce cas, on a une adaptation inter-niveau.

Imaginons que, plus tard, un changement de contexte survient au niveau 2. La fonction de sélection doit choisir une nouvelle configuration parmi les candidats possibles, $A_{2,1}$ et $A_{2,2}$. Si, par exemple, $A_{2,1}$ est retenue, alors l'adaptation peut s'arrêter là, car la configuration courante au niveau 1, $A_{1,2}$, est un raffinement possible de $A_{2,1}$.

4.1.2 Application de l'approche aux SCUs

L'approche générique présentée dans la section précédente peut être instanciée pour gérer l'adaptation multi-niveaux dans des systèmes appartenant à différents domaines et implantés avec des technologies diverses. Dans cette section, nous montrons comment nous avons appliqué cette approche au domaine des systèmes collaboratifs ubiquitaires. Pour cela, nous détaillons nos choix par rapport aux niveaux retenus, et nous listons les éléments à modéliser dans chaque niveau. Le but est de spécifier clairement les problèmes à traiter dans ces systèmes, ainsi que de proposer un cadre conceptuel pour leur conception. L'approche reste générique par rapport aux technologies, aux langages et aux outils. Les détails d'implantation seront cependant abordés dans la section suivante.

Nous avons retenu trois niveaux : le *niveau application*, le *niveau collaboration* et le *niveau intergiciel*. Ces niveaux sont illustrés par la Figure 4.5.

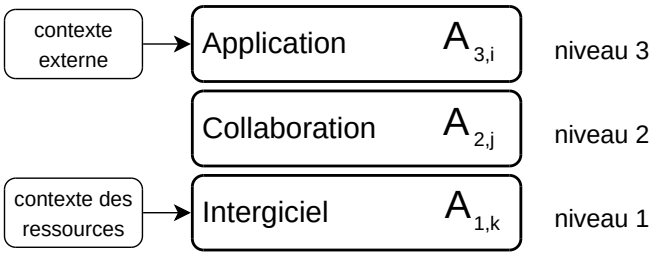


FIG. 4.5 – *Niveaux retenus pour la modélisation des SCUs*

Niveau application

Le niveau supérieur est le niveau application. Dans ce niveau sont modélisées les applications qui ont des besoins de collaboration à l'intérieur de groupes d'utilisateurs dans des environnements ubiquitaires. Le modèle correspondant à ce niveau représente une vue *métier* des entités qui collaborent et des relations existant entre elles. L'application prend en compte le contexte externe lors de l'instanciation de ce modèle. Celui-ci peut également contenir des éléments qui ne sont pas en relation directe avec la collaboration ; l'application peut s'en servir ainsi comme modèle métier et non pas comme modèle de collaboration exclusivement. En tout cas, seuls les éléments qui concernent la collaboration seront utilisés dans le raffinement vers les niveaux inférieurs.

La vue métier que le modèle de ce niveau représente est spécifique au domaine de l'application considérée. Elle doit représenter clairement les types d'entités qui peuvent intervenir dans la collaboration, la façon

dont elles sont organisées, les rapports possibles entre elles, etc. Bien que cette vue sera souvent spécifique à l'application, il est possible que des applications différentes appartenant au même domaine puissent partager un même modèle de niveau application.

Niveau collaboration

Le niveau collaboration fournit une abstraction qui représente la collaboration sous la forme de sessions regroupant des entités qui collaborent au moyen de flux de données. Une instance du modèle de ce niveau est un schéma de collaboration haut niveau qui traduit les besoins de collaboration de l'application de façon générique. Les éléments nécessaires pour mettre en œuvre ces sessions sont également représentés dans ce niveau : sessions, flux, composants, etc.

Les éléments de ce niveau sont représentés de façon abstraite, et donc indépendante de leur implantation. Ils peuvent ainsi être implantés avec diverses technologies, qui seront détaillées dans le niveau intergiciel. En conséquence, ce niveau joue un rôle charnière : il met en œuvre une interopérabilité entre les besoins de collaboration exprimés dans le niveau application et les implantations réelles décrites dans le niveau intergiciel.

Niveau intergiciel

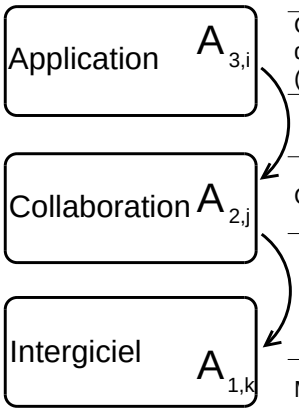
Le niveau intergiciel fournit un modèle de communication qui masque les détails de la communication, tels que les sockets TCP, les datagrammes UDP, les adresses IP, les services réseau, etc. Ceci permet de représenter d'une façon uniforme les canaux de communication qui implantent les échanges sur lesquels se base la collaboration. Ce niveau sert, donc, à traiter les problèmes de l'informatique distribuée, qui sont gérés par les intergiciels. En conséquence, ce niveau utilise un intergiciel tel que les appels de procédure distantes (*Remote Procedure Call* ou RPC) [Nel81], le pair-à-pair (*Peer-to-Peer* ou P2P) [Sch01], la communication basée sur les événements (*Event-Based Communications* ou EBC) [MC02], CORBA [coro6] ou autres. Les instances du modèle de ce niveau contiennent un ensemble d'entités propres au système choisi. Par exemple, dans le cas du pair-à-pair, l'instance contient des *pairs* et des *superpairs*, avec des liens qui les relient entre eux.

Une instance du modèle de ce niveau représente un schéma détaillé des éléments de l'intergiciel retenu et de leurs connexions, et il peut donc être utilisé pour effectuer un déploiement réel de ces éléments. Ce niveau prend en compte le contexte des ressources lors du processus de sélection de la meilleur configuration. Le contexte des ressources modélise l'ensemble des ressources matérielles et logicielles qui sont nécessaires pour l'exécution de l'intergiciel : mémoire, CPU, batterie ou sources d'énergie, connexions réseau, périphériques, dispositifs, etc.

4.2 Modèles et transformations de Facus

Afin d'implanter notre approche de modélisation pour les SCUs dans Facus, nous avons proposé le modèle correspondant à chaque niveau, et

nous avons choisi des techniques pour effectuer les transformations entre niveaux. La Figure 4.6 montre une vue globale de ces choix. Cette figure est une instance de la Figure 4.5 avec les modèles et les transformations concrets utilisés dans Facus. La colonne *Modèle* contient les modèles retenus pour les trois niveaux de modélisation. La deuxième colonne détaille les instances de ces modèles. La colonne *Transformation* contient les méthodes utilisées pour l'implantation du raffinement et de la sélection dans les deux transitions de niveau. La dernière colonne détaille le langage utilisé pour la description des modèles et de leurs instances ou la technologie utilisée pour la transformation entre niveaux, selon le type de la ligne. Dans la suite de cette section nous expliquons tous ces choix et nous les détaillons.



Modèle	Instances du Modèle	Transformation	Langage/ Technologie
Ontologie de domaine (qui étend GCO)	Instances de l'ontologie de domaine		OWL
		Raf. et Sél. : inférence + règles SWRL	Moteur d'inférence +moteur de règles
GCO	Graphes de collaboration (instances de GCO)		OWL / GraphML
		Raff. : transformation de graphes Sél : algorithme de sélection	Moteur de transformation de graphes Implantation Java de l'algorithme
Modèle EBC	Graphes de déploiement		GraphML

Fig. 4.6 – *Choix pour les modèles et les transformations dans* Facus

4.2.1 Modèles des niveaux

Dans cette sous-section, nous décrivons les modèles que nous utilisons dans Facus pour les trois niveaux de modélisation retenus : application, collaboration et intergiciel.

Niveau application

Le modèle de ce niveau est une ontologie OWL qui étend GCO. Une telle ontologie est dépendante du domaine et de l'application visée, et donc elle doit être fournie par le développeur des applications qui utilisent Facus. En général, l'ontologie de ce niveau contiendra notamment des concepts qui héritent des concepts `Role` et `Group` de GCO et quelques règles SWRL qui indiquent comment ces rôles communiquent entre eux. Néanmoins, cette ontologie peut être plus complexe en fonction des besoins de l'application. Une instance $A_{3,i}$ du modèle sera une instance de l'ontologie de collaboration. Elle contiendra des individus reliés par des propriétés qui représentent notamment les nœuds présents, les dispositifs associés, leurs rôles, les groupes et les sessions.

Grâce au mécanisme d'importation d'OWL, les individus des classes de GCO et ceux de l'ontologie de domaine, ainsi que les relations qui les

relient, seront regroupés dans les instances de ce modèle. Une instance de ce modèle est donc une vue métier, à un moment donné, de l'application, qui contient explicitement ou implicitement tous les aspects de collaboration. Il est possible, si l'application en a besoin, d'ajouter dans cette ontologie d'autres concepts et relations qui ne concernent pas la collaboration. Par exemple, on peut ajouter des éléments qui modélisent l'interface utilisateur ou les modules de sécurité de l'application. Ces ajouts ne seront pas pris en compte lors du processus de raffinement.

Niveau collaboration

Les instances $A_{2,j}$ du modèle de ce niveau sont des graphes de collaboration qui représentent explicitement les entités qui prennent partie à l'activité collaborative, ainsi que les échanges de données qui ont lieu entre ces entités. Une instance de GCO (ou d'une ontologie d'application qui étend GCO) contient un tel graphe de collaboration, si on tient compte seulement des nœuds, des dispositifs, des flux et des composants (composants d'envoi et composants de réception, avec leurs types). Les autres éléments de GCO (et de l'ontologie de niveau application) ne sont pas représentés dans le graphe de collaboration. Le modèle de ce niveau est donc GCO. Un graphe de collaboration peut être considéré comme un schéma de déploiement abstrait. Le niveau intergiciel détaille comment implanter un tel schéma avec les composants fournis par un intergiciel donné.

Un graphe de collaboration peut donc être exprimé avec une instance de GCO réduite aux individus qui représentent les nœuds, les dispositifs, les flux et les composants. Cependant, afin de partager les instances du modèle avec les niveaux inférieurs, ce graphe résultant est exprimé en XML avec GraphML. GraphML[1] [BEH+01] est un dialecte XML pour l'expression de graphes. Il permet l'expression de graphes orientés ou non, de graphes hiérarchiques, d'hypergraphes, etc. Il est extensible et il permet d'associer des étiquettes tant aux sommets qu'aux arêtes. Nous avons retenu ce langage parmi d'autres possibles (comme GXL, GML ou TGF) en raison de sa simplicité, sa syntaxe XML (qui facilite la génération et lecture des graphes) et sa flexibilité.

Nous avons utilisé GraphML pour exprimer les graphes de collaboration de la façon suivante :

1. Les sommets du graphe représentent les composants. Les sommets ont une étiquette composée de trois parties : l'identificateur du composant, son type (*sender* s'il envoie des données ou *receiver* s'il en reçoit) et l'identificateur du dispositif où il est déployé.
2. Les arêtes du graphe représentent les flux de données envoyés entre un composant d'envoi (source de l'arête) vers un composant de réception (destination de l'arête). Les arêtes ont une étiquette composée de deux parties : le type de données gérées par le flux (*audio*, *video* ou *text*) et le nom de la session à laquelle le flux appartient.

La Figure 4.7 montre un exemple de graphe de collaboration simple en GraphML. Ce fichier GraphML contient un graphe appelé `exempleCollab` dans lequel un composant d'envoi `sender1` envoie un

[1] http://graphml.graphdrawing.org/

flux de type audio au composant de réception `receiver1`. Le composant `sender1` est déployé sur un dispositif dont l'identificateur est `dev5`, tandis que le composant `receiver1` est déployé sur le dispositif `dev4`. Le flux qui relie ces composants est de type audio et il appartient à la session dont l'identificateur est `session1`. L'attribut `edgedefault` avec la valeur `directed` veut dire qu'il s'agit d'un graphe orienté. La Figure 4.8 montre une représentation graphique du graphe `exempleCollab`. Les étiquettes des nœuds contiennent en premier lieu l'identificateur, puis les attributs du nœud, séparés par des virgules. Les étiquettes des flux contiennent les attributs du flux.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
  <graph id="exempleCollab" edgedefault="directed">
    <node id="sender1">
      <data key="attribute1">sender</data>
      <data key="attribute2">dev5</data>
    </node>
    <node id="receiver1">
      <data key="attribute1">receiver</data>
      <data key="attribute2">dev4</data>
    </node>
    <edge source="sender1" target="receiver1">
      <data key="attribute1">audio</data>
      <data key="attribute2">session1</data>
    </edge>
  </graph>
</graphml>
```

Fig. 4.7 – *Exemple de graphe de collaboration en GraphML*

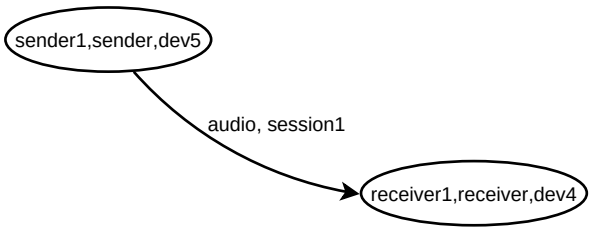


Fig. 4.8 – *Exemple de graphe de collaboration*

Niveau intergiciel

Le premier choix que nous avons fait pour ce niveau est celui du type d'intergiciel utilisé. Nous avons retenu la communication basée sur les événements ou EBC [MC02]. EBC (connu aussi sous le nom de *publish/subscribe*) est un modèle de communication qui permet l'interconnexion de composants faiblement couplés, en mode synchrone ou asynchrone. Les schémas de type *un-vers-plusieurs* et *plusieurs-vers-plusieurs* peuvent être facilement implantés avec EBC. Le coût d'initialisation des connexions est assez léger par rapport au paradigme client-serveur et il est bien adapté aux situations

où plusieurs composants distribués établissent des relations dynamiques à la volée d'une façon imprévisible. Ces caractéristiques rendent EBC très pertinent pour les environnements ubiquitaires.

Il y a trois types d'entités EBC : les *producteurs d'événements* (*event producers* ou EP en anglais), les *consommateurs d'événements* (*event consumers* ou EC en anglais) et les *gestionnaires de canal* (*channel managers* ou CM en anglais). Les EP et les EC peuvent être connectés à des CM, mais ils ne peuvent pas être directement reliés entre eux. Les EP peuvent envoyer des données vers le CM auquel ils sont connectés. Le CM renvoie une copie des données reçues à tous les EC qui sont reliés à lui. Chaque lien peut être de type *push* ou *pull*, selon si les données sont lues à initiative de l'envoyeur ou du récepteur. Les données peuvent être envoyées périodiquement ou sporadiquement. Les données envoyées appartiennent à un *topic* (ou sujet) qui permet de distinguer des « conversations » différentes.

En conséquence, le modèle de niveau intergiciel est le modèle EBC. Une instance $A_{1,k}$ de ce modèle contient un ensemble d'entités de type EP, EC et CM reliées entre elles. Une instance est représentée comme un graphe décrit en GraphML de la façon suivante :

1. Les sommets du graphe représentent les entités EBC. Les sommets ont une étiquette composée de cinq parties : l'identificateur de l'entité, son type (EP, EC ou CM), le type de données envoyées (*audio*, *video* ou *text*), l'identificateur de la session qu'il gère (c'est-à-dire le *topic* auquel les données sont affectées) et l'identificateur du dispositif où elle est déployée.
2. Les arêtes du graphe représentent les liens entre les entités EBC. La direction de l'arête indique la direction que suivent les données. Les arêtes ont une étiquette qui indique s'il s'agit d'un lien de type *push* ou de type *pull*.

Un tel graphe constitue un graphe de déploiement qui permet de connaître de façon détaillée les composants physiques qu'il faut déployer, les machines sur lesquelles ils doivent être configurés et la façon dont ils doivent être connectés afin d'implanter un schéma de collaboration donné.

La Figure 4.9 montre un exemple de graphe de déploiement simple en GraphML. Ce fichier GraphML contient un graphe appelé `exempleDepl` qui comprend trois entités EBC : un EP appelé `ep1`, un CM appelé `cm1` et un EC appelé `ec1`. Le producteur `ep1` est connecté à `cm1` par un lien de type *push*, et `cm1` est connecté à `ec1` par un autre lien *push*. Les trois entités sont de type audio et elles appartiennent à la session `session1`. Les entités `ep1` et `cm1` sont déployées sur le dispositif `dev5`, tandis que `ec1` est déployé sur `dev4`. La Figure 4.10 montre une représentation graphique du graphe `exempleDepl`.

4.2.2 Transformations entre niveaux

Dans cette sous-section nous décrivons les techniques retenues pour l'implantation du raffinement et de la sélection dans les deux transformations entre niveaux de Facus : du niveau application vers le niveau collaboration et du niveau collaboration vers le niveau intergiciel.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
  <graph id="exempleDepl" edgedefault="directed">
    <node id="ep1">
      <data key="attribute1">EP</data>
      <data key="attribute2">audio</data>
      <data key="attribute3">session1</data>
      <data key="attribute4">dev5</data>
    </node>
    <node id="cm1">
      <data key="attribute1">CM</data>
      <data key="attribute2">audio</data>
      <data key="attribute3">session1</data>
      <data key="attribute4">dev5</data>
    </node>
    <node id="ec1">
      <data key="attribute1">EC</data>
      <data key="attribute2">audio</data>
      <data key="attribute3">session1</data>
      <data key="attribute4">dev4</data>
    </node>
    <edge source="ep1" target="cm1">
      <data key="attribute1">push</data>
    </edge>
    <edge source="cm1" target="ec1">
      <data key="attribute1">push</data>
    </edge>
  </graph>
</graphml>
```

Fig. 4.9 – *Exemple de graphe de déploiement en GraphML*

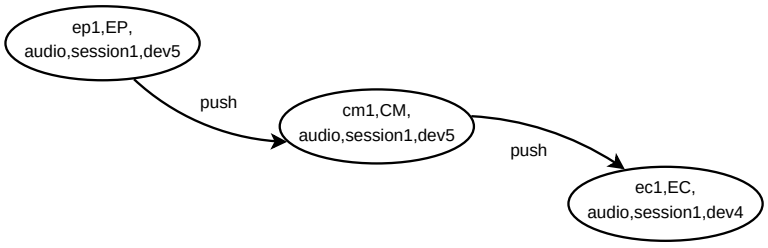


Fig. 4.10 – *Exemple de graphe de déploiement*

Transformation du niveau application vers le niveau collaboration

Le raffinement entre les niveaux application et collaboration est basé sur le raisonnement et l'exécution de règles SWRL effectués sur l'instance de l'ontologie de l'application, qui étend GCO. Comme nous l'avons expliqué dans le Chapitre 3, les éléments de GCO et les règles ont été définis de telle façon que le résultat de ce processus soit une instance de GCO qui contient toutes les informations nécessaires pour décrire un graphe de collaboration. Ce processus obtient explicitement les éléments présents

dans un graphe de collaboration : les nœuds, les dispositifs, les flux et les composants.

Une fois que l'inférence et le traitement des règles ont produit tous les éléments nécessaires dans l'instance de l'ontologie, cette instance peut être parcourue pour lire ces éléments et produire un fichier GraphML qui exprime le graphe de collaboration correspondant. La bibliothèque Protégé-OWL API, présentée dans le chapitre précédent, facilite cette tâche.

En partant d'une instance de l'ontologie de domaine, le processus de raisonnement et de traitement de règles produit une instance unique de GCO, ce qui donne un graphe de collaboration unique. En conséquence, le mécanisme de sélection dans cette première transformation est trivial : le seul graphe de collaboration candidat est retenu.

Transformation du niveau collaboration vers le niveau intergiciel

Raffinement Puisque les instances des modèles tant au niveau collaboration qu'au niveau intergiciel sont exprimées avec des graphes, nous avons décidé d'utiliser des techniques de transformation de graphes afin d'implanter le raffinement entre ces deux niveaux. Plus concrètement, nous utilisons des *Grammaires Génératives* [Cho56] de graphes. Cela nous permet de spécifier le raffinement sous la forme de règles de transformation de graphes.

Les grammaires génératives sont définies comme un système grammatical $< AX; NT; T; P >$, où AX est l'axiome, NT est l'ensemble de littéraux non-terminaux, T est l'ensemble de littéraux terminaux et P est l'ensemble de productions ou règles de transformation. Les productions permettent de transformer les littéraux non-terminaux en littéraux terminaux. En conséquence, l'application récursive des productions sur une occurrence en entrée (qui ne contient que des littéraux non-terminaux) produit en sortie une occurrence qui ne contient que des littéraux terminaux.

Pour mettre en œuvre la transformation de graphes, nous utilisons la définition de grammaires génératives de graphes fournie par K. Guennoun [Gue06]. Dans le cas des grammaires de graphes, les littéraux sont des nœuds d'un graphe. Nous avons en entrée des graphes qui contiennent uniquement des nœuds non-terminaux. L'application d'une séquence de productions de P permet d'obtenir en sortie des graphes qui contiennent uniquement des nœuds terminaux. Les productions contenues dans P sont de la forme $(L; K; R)$, où L, R et K sont des sous-graphes. Une production est applicable au graphe G s'il existe un homomorphisme de L vers G (i.e., L est un sous-graphe contenu dans G). Si une production est applicable, son application implique la destruction de l'occurrence du sous-graphe $Del = (L \setminus K)$ et la création de la copie isomorphe du sous-graphe $Add = (R \setminus K)$. Autrement dit, L représente le sous-graphe mis en correspondance par la règle, dans lequel on conserve le sous-graphe K en ajoutant le sous-graphe Add. Le résultat est le sous-graphe R.

Dans notre cas, les nœuds non-terminaux sont ceux des graphes de collaboration, tandis que les nœuds terminaux sont ceux des graphes de déploiement EBC. Nous utilisons la grammaire de graphes $GG_{collab \rightarrow ebc}$,

telle que définie dans la Figure 4.11. Cette grammaire de graphes a été élaborée en collaboration avec I. Bouassida Rodriguez [SBT+09].

$GG_{collab \to ebc} = (AX, NT, T, P)$ où :
$AX = \{\}$,
$NT = \{(id, "receiver", m), (id, "sender", m)\}$,
$T = \{(id, "CM", d, s, m), (id, "EC", d, s, m), (id, "EP", d, s, m)\}$ et
$P = \{p_1, p_2, p_3\}$

$p_1 = ($
$L = \{(sc1, "sender", m2) \xrightarrow{d,s} (rc1, "receiver", m1),$
$(sc2, "sender", m1) \xrightarrow{d,s} (rc2, "receiver", m2)\}$;
$K = \{\}$;
$R = \{(cm1, "CM", d, s, m1) \xrightarrow{push} (ec1, "EC", d, s, m1),$
$(ep1, "EP", d, s, m2) \xrightarrow{push} (cm1, "CM", d, s, m1),$
$(cm1, "CM", d, s, m1) \xrightarrow{push} (ec2, "EC", d, s, m2),$
$(ep2, "EP", d, s, m1) \xrightarrow{push} (cm1, "CM", d, s, m1)\}$
$)$

$p_2 = ($
$L = \{(sc1, "sender", m2), (cm1, "CM", d, s, m1)\}$;
$K = \{(cm1, "CM", d, s, m1)\}$;
$R = \{(ep1, "EP", d, s, m2) \xrightarrow{push} (cm1, "CM", d, s, m1)\}$
$)$

$p_3 = ($
$L = \{(rc1, "receiver", m2), (cm1, "CM", d, s, m1)\}$;
$K = \{(cm1, "CM", d, s, m1)\}$;
$R = \{(cm1, "CM", d, s, m1) \xrightarrow{push} (ec1, "EC", d, s, m2)\}$
$)$

Fig. 4.11 – *Grammaire de graphes* $GG_{collab \to ebc}$

Les nœuds non terminaux possèdent un identificateur (*id*), un type de composant (*"sender"* ou *"receiver"* en fonction de s'il s'agit d'un composant d'envoi ou de réception) et l'identificateur de la machine sur laquelle ils se trouvent (*m*). Les nœuds terminaux ont un identificateur (*id*), un type (*"CM"*, *"EC"* ou *"EP"* en fonction de s'il s'agit respectivement d'un gestionnaire de canal, d'un consommateur ou d'un producteur), un type de données (*d*), une session (*s*) à laquelle ils appartiennent et l'identificateur (*m*) de la machine sur laquelle ils sont déployés. Ces définitions correspondent à celles que nous avons données dans la sous-section précédente pour les nœuds des graphes de collaboration et de déploiement, respectivement.

La règle $p1$, illustrée par la Figure 4.12, cherche un sous-graphe L qui consiste en un composant d'envoi $sc1$ et un composant de réception $rc2$ déployés sur la machine $m2$, connectés, respectivement, à un composant de réception $rc1$ et à un composant d'envoi $sc2$ qui sont sur la machine $m1$; tous ces composants appartiennent à la session s. Ce sous graphe est remplacé par le sous-graphe R, qui consiste en un CM $cm1$, déployé sur

$m1$, auquel sont connectés deux EP $ep1$ et $ep2$ (déployés sur $m2$ et $m1$ respectivement) et deux EC $ec1$ et $ec2$ (déployés sur $m1$ et $m2$ respectivement).

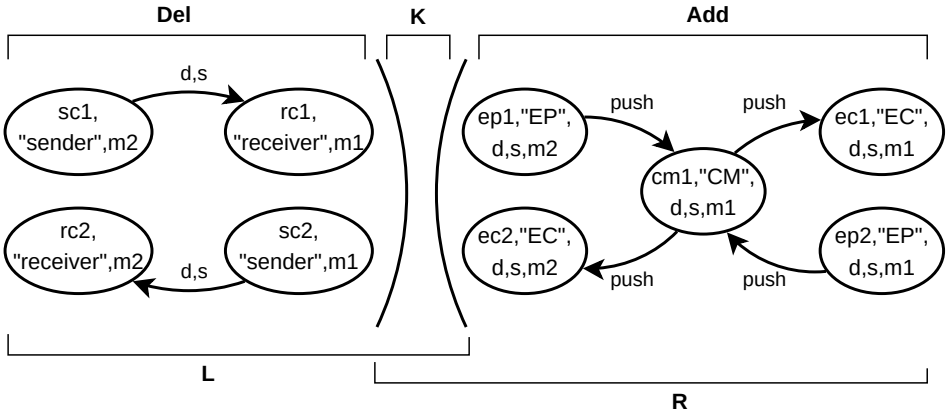


FIG. 4.12 – *Règle de transformation* $p1$

La règle $p2$, illustrée par la Figure 4.13, cherche un sous-graphe L qui consiste en un composant d'envoi $sc1$ et un CM $cm1$, non connectés, qui appartiennent à la même session s et qui sont sur deux machines $m2$ et $m1$. Ce sous-graphe est remplacé par un autre sous-graphe R dans lequel le $cm1$ est conservé, tandis que $sc1$ est remplacé par un EP $ep1$ déployé sur la même machine et connecté à $cm1$.

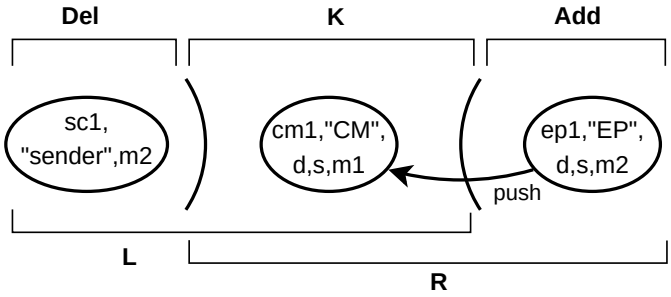


FIG. 4.13 – *Règle de transformation* $p2$

La règle $p3$, illustrée par la Figure 4.14, cherche un sous-graphe L qui consiste en un composant de réception $rc1$ et un CM $cm1$, non connectés, qui appartiennent à la même session s et qui sont sur deux machines $m2$ et $m1$. Ce sous-graphe est remplacé par un autre sous-graphe R dans lequel le $cm1$ est conservé, tandis que $rc1$ est remplacé par un EC $ec1$ déployé sur la même machine et connecté à $cm1$.

Afin d'appliquer les règles de transformation de la grammaire sur les graphes générés par la couche collaboration, nous utilisons un logiciel appelé *Graph Matching and Transformation Engine* (GMTE)[2]. Ce moteur de transformation de graphes a été développé au sein de notre groupe de recherche par K. Guennoun, I. Bouassida Rodriguez et K. Drira, notamment [DB10, BGD+08]. Lors du raffinement, le GMTE est utilisé pour appliquer chaque règle récursivement sur le graphe d'entrée sur toutes les

[2]Disponible sur http://homepages.laas.fr/khalil/GMTE/

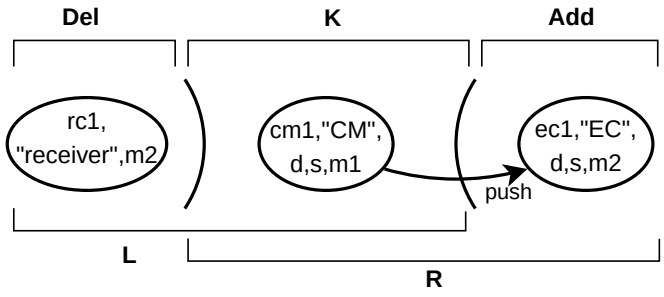


Fig. 4.14 – *Règle de transformation p*3

occurrences possibles de L jusqu'à ce qu'il ne soit plus possible de l'appliquer. Les graphes produits sont utilisés de la même façon avec la règle suivante, et ainsi de suite. Le résultat de ce processus est un ensemble contenant toutes les combinaisons possibles de graphes terminaux (i.e., de graphes de déploiement EBC) générées à partir du graphe de collaboration considéré.

Sélection Afin de sélectionner le graphe à retenir au niveau intergiciel, nous utilisons l'algorithme de sélection présenté dans la sous-section 4.1.1. Pour cela, nous devons définir notre implantation des fonctions $Context\_Adaptation()$, $Distance()$ et $Dispersion()$.

La fonction $Context\_Adaptation()$ est détaillée dans la Figure 4.15. Cette fonction prend en compte un ensemble de ressources. Chaque ressource a un *seuil* et un *poids* associés. Si un dispositif a un niveau pour une ressource qui est en dessous du seuil, on considère que ce dispositif est en état critique. On suppose que cette fonction a accès au niveau de chaque ressource pour chaque dispositif (la sous-section 4.3.2 explique la prise en compte du contexte dans Facus, quelles sont les ressources considérées et comment les niveaux sont obtenus). La fonction calcule, en un premier lieu, la valeur P_r^i pour chaque dispositif (ligne 15), qui exprime de combien la valeur de la ressource considérée est au dessus du seuil correspondant pour le dispositif (une fois la consommation des composants déployés dans le dispositif enlevée). Si cette valeur est négative pour un des dispositifs de l'architecture, alors cela veut dire que le dispositif sera en état critique si cette architecture est déployée. Dans ce cas, la fonction retourne -1 comme résultat afin que la fonction de sélection ne la retienne pas (ligne 16). Si ce n'est pas le cas, alors la fonction calcule la valeur minimale de P_r^i pour chaque ressource dans toute l'architecture et elle retourne la somme pondérée (par les poids de chaque ressource) des ces valeurs (ligne 20). Intuitivement, cela veut dire qu'on considère que les architectures les plus adaptées au contexte sont celles où les dispositifs « les plus critiques » pour chaque ressource sont le plus en dessus du seuil. Les poids β_r, fixés par l'utilisateur de Facus, servent à donner plus d'importance à certaines ressources par rapport à d'autres.

La fonction $Distance()$ est détaillée dans la Figure 4.16. Cette fonction prend comme argument deux architectures et elle retourne comme résultat le nombre de composants qui ne sont pas déployés dans le même dispositif dans les deux architectures. Intuitivement, cela représente le

```
1  Context_Adaptation(A_{1,q}){
2     Soit N = card(A_{1,q})
3     Soit d_i i ∈ [1..N] un dispositif de A_{1,q}
4     Soit L_r^i le niveau de la ressource r pour d_i
5     Soient m_{ep,i}, m_{ec,i} et m_{cm,i} le nombre d'EP, d'EC et de
      CM déployés dans d_i
6     Soit {r_1,...,r_R} l'ensemble des ressources considérées
7     Soit R = card({r_1 ... r_R})
8     Soit T_r le seuil associé à la ressource r
9     Soit β_r le poids associé à la ressource r
10    Soient C_{ep,r}, C_{ec,r} et C_{cm,r} r ∈ [1..R] les consommations
      d'EP, d'un EC et d'un CM pour la ressource r
11    Soit adapt = 0
12    Pour chaque i ∈ [1..N]
13       Pour chaque r ∈ [1..R]
14          Soit C_i = C_{ep,r} m_{ep,i} + C_{ec,r} m_{ec,i} + C_{cm,r} m_{cm,i}
15          Soit P_r^i = L_r^i − C_i − T_r
16          Si P_r^i ≤ 0, alors retourner −1
17       fin pour
18    fin pour
19    Pour chaque r ∈ [1..R]
20          adapt = adapt + β_r min_i(P_r^i)
21    fin pour
22    retourner adapt
23 }
```

FIG. 4.15 – *Fonction Context_Adaptation()*

```
1  Distance(A_{1,q}, A_{1,k}){
2     Soit d_i^q(c_j) le dispositif où est déployé le composant
      c_j ∈ A_{1,q}
3     Soit dist = 0
4     Pour chaque c_j ∈ A_{1,q} ∪ A_{1,k}
5        Si d_i^q(c_j) ≠ d_i^k(c_j) alors dist = dist + 1
6     fin pour
7     retouner dist
8  }
```

FIG. 4.16 – *Fonction Distance()*

nombre de redéploiements qu'il faut effectuer pour passer d'une architecture à l'autre. Selon la description de *Selection()* que nous avons fournie dans la Figure 4.3, les architectures ayant une valeur de distance plus grande auront moins de probabilité d'être sélectionnées.

La fonction *Dispersion()* est détaillée dans la Figure 4.17. Cette fonction affecte à l'architecture passée en paramètre sa valeur de dispersion, calculée comme le nombre de dispositifs sur lesquels nous avons des CM déployés. Intuitivement, si une architecture a une valeur de dispersion

grande, cela veut dire que les CM ne sont pas concentrés dans un petit nombre de dispositifs, ce qui la rendra plus robuste, car en cas de panne ou déconnexion d'un dispositif, peu de CM se verront affectés. Selon la description de *Selection*() que nous avons fournie dans la Figure 4.3, les architectures ayant une valeur de dispersion plus grande auront plus de probabilité d'être sélectionnées.

```
1 Dispersion(A_{1,q}){
2     Soit N = card(A_{1,q})
3     Soit d_i i ∈ [1..N] un dispositif de A_{1,q}
4     Soit disp = 0
5     Pour chaque d_i i ∈ [1..N]
6         Si ∃CM déployé dans d_i, alors disp = disp + 1
7     fin pour
8     retourner disp
9 }
```

Fig. 4.17 – *Fonction Dispersion*()

4.3 Conception et implantation de Facus

Dans cette section nous présentons l'architecture de Facus au travers de sa conception et de son implantation, ainsi que la façon dont il peut être utilisé pour la conception et l'exécution de SCUs.

Facus est un *framework* Java pour les SCUs. Rappelons la définition de *framework* pour les langages orientés objet utilisée par Gamma et al. [GHJV95] :

« *Un* framework *est un ensemble de classes qui coopèrent entre elles et qui constituent un patron de conception réutilisable pour une catégorie spécifique de logiciels.* »

Dans le cas de Facus, la catégorie spécifique de logiciels dont on parle sont les SCUs. Gamma et al. précisent l'utilisation des *frameworks* :

« *Une application spécifique utilise un* framework *en créant des sous-classes spécifiques à l'application qui héritent des classes abstraites fournies par le* framework. *Le* framework *dicte l'architecture de l'application. Il définit la structure générale, sa décomposition en classes et objets, leurs responsabilités, la façon dont les classes et les objets collaborent et le fil de contrôle. Un* framework *prédéfinit ces paramètres de conception pour que le concepteur/développeur de l'application se concentre sur les spécificités de l'application. Le* framework *capture les décisions de conception qui sont communes au domaine de l'application. Ainsi, les* frameworks *favorisent la* réutilisation de la conception *plutôt que la* réutilisation du code, *bien que, souvent, ils fournissent des sous-classes concrètes qui peuvent être utilisées directement.* »

Comme nous allons le voir dans cette section, Facus s'adapte parfaitement à cette définition. En effet, il fixe une structure générale sur laquelle les applications peuvent se greffer en créant des sous-classes, et il fournit également plusieurs sous-classes concrètes qui sont prêtes à l'emploi. Cette structure ne s'arrête pas aux classes et sous-classes, mais elle com-

prend aussi le cadre de modélisation utilisé pour la gestion de la collaboration dans les SCUs. Comme nous l'expliquons dans la sous-section 4.3.5, les concepteurs d'applications qui utilisent FACUS doivent fournir les éléments propres à l'application dans un modèle de l'application. Ensuite, lors de l'exécution du système, les instances de ce modèle devront être mises à jour au fur et à mesure que l'état de l'application change.

4.3.1 Architecture de Facus

FACUS est conçu pour être utilisé dans une machine qui centralise la logique de décision de la collaboration. Dans des applications qui ont un serveur central sur lequel se connectent les clients qui se trouvent sur les dispositifs des utilisateurs, FACUS sera utilisé par ce serveur central. En général, nous pouvons considérer que les SCUs ont un tel serveur sur chacun des espaces ubiquitaires, et que ce serveur gère l'application dans l'espace où elle se trouve. Il est également possible d'exécuter FACUS sur un des dispositifs des utilisateurs, pourvu qu'il dispose des ressources nécessaires. Afin de rendre plus claire notre explication, et sans perte de généralité, nous considérerons que FACUS est utilisé par l'application sur une machine que nous appellerons *serveur* et que les machines des utilisateurs (ou *dispositifs*) s'y connectent dans les environnements ubiquitaires.

La Figure 4.18 est un diagramme de classes UML qui présente les principales classes qui constituent FACUS, ainsi que leurs relations. La conception de FACUS est centrée sur la notion de couche (*layer* en anglais). Les couches correspondent aux niveaux de notre approche de modélisation, auxquels sont ajoutés les traitements nécessaires pour la production de l'instance du modèle du niveau considéré à partir de celle du niveau supérieur. La classe `Layer`, abstraite, contient les attributs et les méthodes communs à toutes les couches. Les classes qui correspondent aux couches spécifiques héritent de `Layer` : `ApplicationLayer`, `CollaborationLayer` et `MiddlewareLayer`.

Les instances des modèles de chaque niveau sont encapsulées dans des objets des classes qui héritent de la classe abstraite `Descriptor` : `ApplicationDescriptor`, `CollaborationDescriptor` et `MiddlewareDescriptor`. La classe `Layer` possède deux attributs `descriptor` et `upperDescriptor` qui correspondent à l'instance retenue dans la couche considérée et dans la couche supérieure, respectivement. Cette classe déclare une méthode abstraite `generateDescriptor()` qui produit le descripteur du niveau considéré à partir de celui du niveau supérieur. Cette méthode est implantée par les classes concrètes qui héritent de `Layer`.

Les couches sont coordonnées grâce à un objet de la classe `LayerManager`. Cette classe a une référence vers chaque couche, et les couches ont également une référence vers le `LayerManager`. Quand une couche produit le descripteur de son niveau, elle le passe au `LayerManager`, qui, en fonction du niveau du descripteur reçu, le passe à la couche inférieure, ou, dans le cas du descripteur du niveau intergiciel, au service de déploiement (classe `DeploymentService`, voir la sous-section 4.3.3).

Puisque le modèle de niveau application dépend de l'applica-

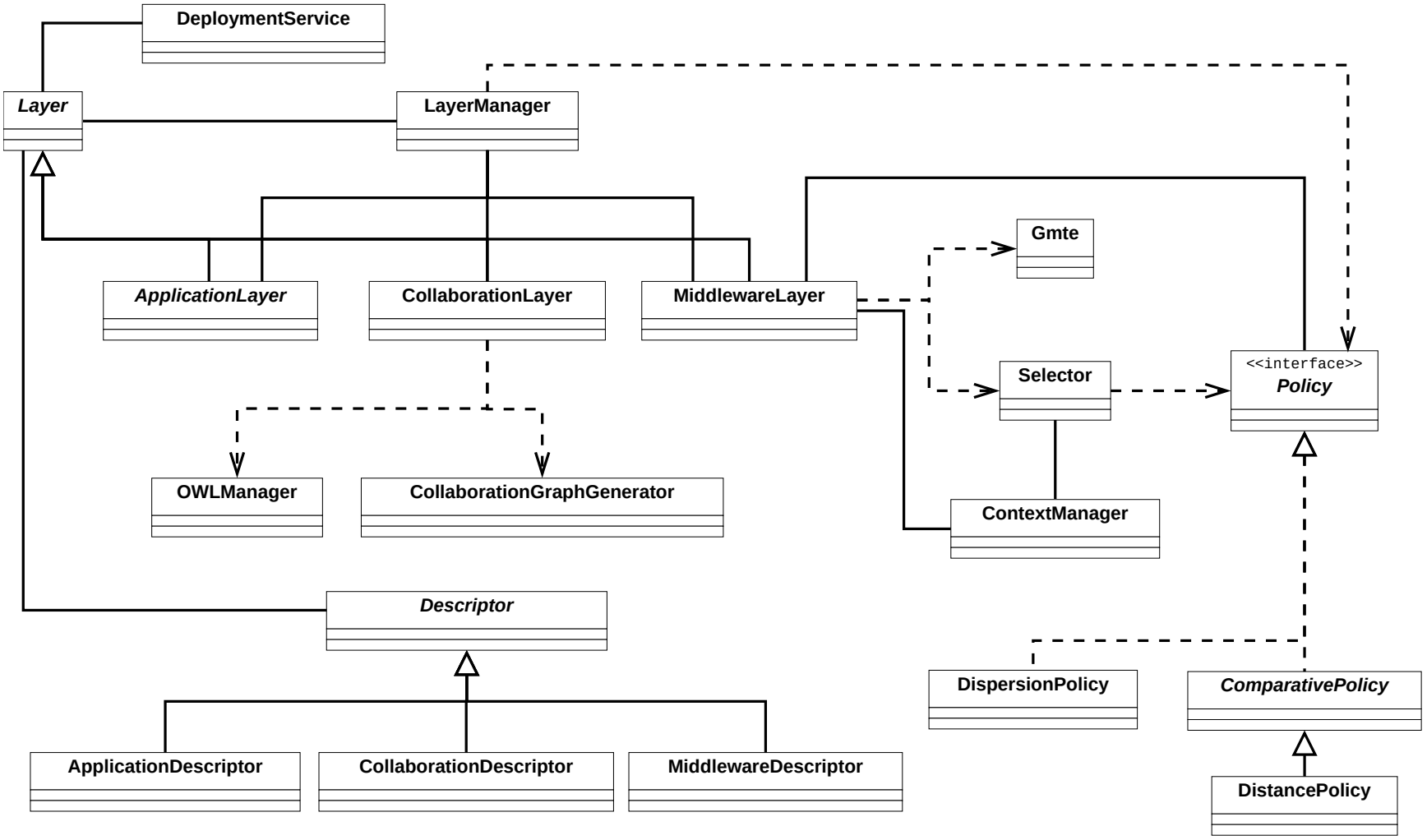


Fig. 4.18 – *Diagramme de classes UML de* Facus

tion concrète qui utilise FACUS, la classe qui correspond à ce niveau, `ApplicationLayer`, est abstraite. Les concepteurs des applications doivent fournir une classe concrète qui hérite de cette classe afin d'utiliser FACUS (voir la sous-section 4.3.5).

Afin d'implanter le raffinement du niveau application vers le niveau collaboration, la classe `CollaborationLayer` utilise la classe `OWLManager`, qui lui permet de charger une ontologie OWL, de la traiter avec le moteur d'inférence Pellet et d'exécuter les règles SWRL avec le moteur de règles Jess. Afin de traduire une instance de GCO sous forme de graphe de collaboration, la couche collaboration utilise la classe `CollaborationGraphGenerator`. Cette classe utilise Pellet pour parcourir l'ontologie et extraire les éléments à représenter dans le graphe.

Pour la génération de l'instance du modèle de niveau intergiciel, la classe `MiddlewareLayer` utilise le moteur de transformation de graphes (classe `Gmte`[3]) pour le raffinement, et la classe `Selector` pour la sélection. La classe `Selector` est une implantation en Java de l'algorithme de sélection présenté dans la sous-section 4.1.1. Cette classe a donc besoin d'accéder aux données de contexte, ce qu'elle fait à travers la classe `ContextManager` (voir sous-section 4.3.2), et d'utiliser les politiques. Une liste de politiques lui est fournie par le `LayerManager`. Les politiques sont représentées par l'interface `Policy`. Les politiques concrètes, comme `DispersionPolicy`, implantent cette interface. `DistancePolicy` hérite de la classe abstraite `ComparativePolicy`, dont doivent hériter toutes les politiques qui comparent des configurations. Nous avons fourni les implantations de ces deux politiques, mais d'autres peuvent être facilement ajoutées.

Quand le `LayerManager` reçoit le descripteur de niveau application, il appelle le raffinement de la couche collaboration dans un *thread* séparé, afin que l'application puisse continuer son exécution sans rester bloquée par l'appel à FACUS. Ceci a un deuxième avantage : si l'application lance un nouveau processus de raffinement lorsque le précédent n'est pas encore fini, on peut annuler facilement l'exécution du *thread* en cours et lancer le nouveau. La classe `LevelChangeThread`, qui n'est pas représentée dans la figure, implante ce *thread* qui lance le raffinement au niveau collaboration et qui finit avec l'appel au service de déploiement.

4.3.2 Acquisition des données de contexte

Comme nous l'avons vu, le mécanisme de sélection implanté dans le niveau intergiciel est sensible au contexte. En effet, il utilise la fonction *Context_Adaptation*(), qui affecte à chaque configuration candidate du niveau intergiciel (qui sont des graphes de déploiement) une valeur numérique qui indique son « degré d'adaptation au contexte ». Afin de calculer cette valeur, la fonction *Context_Adaptation*() a besoin de connaître, entre autres, le niveau de chaque paramètre de contexte sur chaque dispositif. Pour cela, nous avons utilisé la boite à outils *Context Toolkit*, que nous avons introduite dans la section 2.3.

[3]Puisque le GMTE est développé en C++, nous avons écrit une passerelle en Java à l'aide de la bibliothèque *Java Native Interface* (JNI) ; la classe `Gmte` implante cette passerelle.

Context Toolkit fournit une architecture élémentaire dans laquelle des capteurs sont encapsulés par des composants appelés *widgets*. Les *widgets* exposent les données des capteurs à travers une interface de programmation. Context Toolkit fournit également des outils qui permettent d'interroger des *widgets* distants à travers le réseau.

La Figure 4.19 illustre les principales classes impliquées dans la gestion du contexte dans FACUS. Dans cette figure nous avons pris l'exemple du niveau de batterie des dispositifs comme paramètre de contexte. La classe `BatteryContextWidget` est le *widget* qui dialogue avec la sonde qui détecte le niveau de batterie. Cette classe hérite de la classe `Widget` fournie par le Context Toolkit. La sonde du niveau de batterie est un objet de la classe `BatteryGenerator`. Ces deux classes s'échangent des objets de la classe `BatteryData`. Le `BatteryContextWidget` peut être interrogé à distance par un `ContextMonitor`. Cette classe hérite de la classe `Handler` de Context Toolkit. Chaque `ContextMonitor` peut surveiller un ou plusieurs *widgets*. Un *widget* peut également informer de façon spontanée son moniteur quand des changements du niveau de batterie surviennent. Les échanges entre ces classes contiennent des objets de la classe `Attribute`, fournie par le Context Toolkit. Cette classe permet l'expression des données de contexte sous la forme de paires clé-valeur.

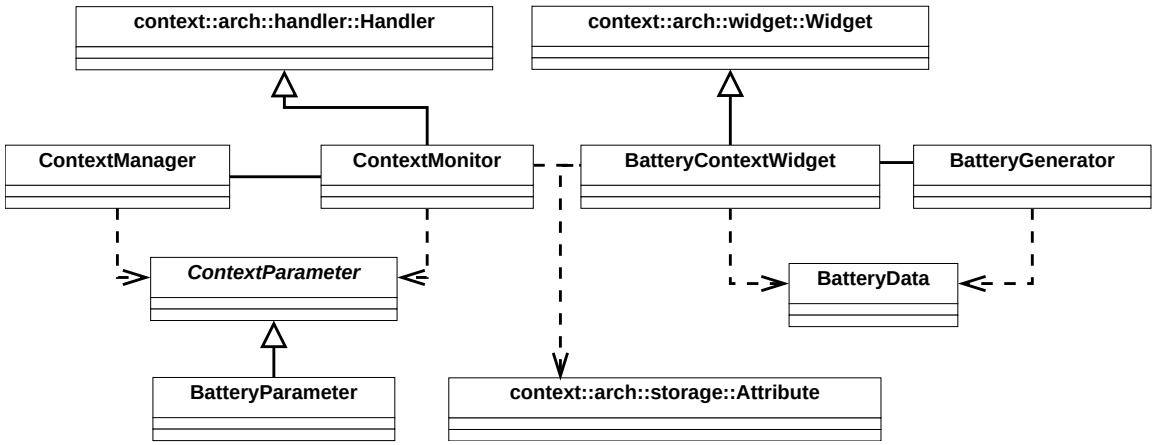


FIG. 4.19 – *Principales classes pour la gestion du contexte dans* FACUS

Les instances de la classe `ContextMonitor` sont pilotées par un objet de la classe `ContextManager`. Comme nous l'avons vu, cet objet est le point d'entrée utilisé par la couche intergiciel afin de récupérer les données de contexte. La couche intergiciel indique au `ContextManager` quelles sont les entités à surveiller et quels paramètres de contexte doivent être pris en compte, et le `ContextManager` s'occupe d'instancier les objets `ContextMonitor` nécessaires. Les différents paramètres de contexte pris en compte sont représentés par les classes qui héritent de la classe abstraite `ContextParameter`, dans notre cas `BatteryParameter`. Les paramètres considérés peuvent appartenir aux deux types de contexte que nous avons définis dans la section 2.3 : le *contexte des ressources* (par exemple, la batterie d'un dispositif, sa mémoire, la bande passante disponible, etc.) et le *contexte externe* (par exemple, la luminosité d'une pièce, la température, etc.).

La Figure 4.20 montre un exemple de l'architecture d'acquisition des données de contexte lors d'une exécution de FACUS. On peut voir comment l'instance du `ContextManager` contrôle celles des `ContextMonitor`. Toutes ces instances se trouvent dans la machine serveur où FACUS s'exécute. Les `ContextMonitor` communiquent à travers le réseau avec les dispositifs des utilisateurs qui collaborent afin de récupérer les données de contexte qui sont fournies par les *widgets* qui interrogent les capteurs de contexte.

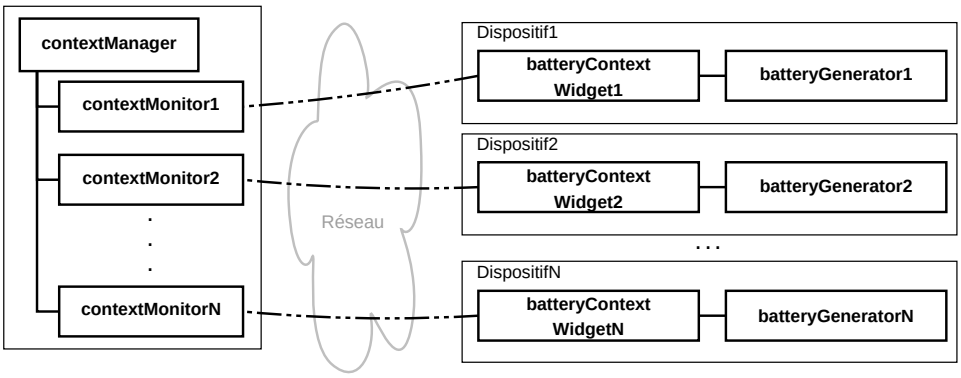


FIG. 4.20 – *Architecture d'acquisition des données de contexte dans* FACUS

La communication entre la couche intergiciel et le `ContextManager` est bidirectionnelle : la couche intergiciel peut interroger le `ContextManager` afin de connaître la valeur d'un certain paramètre pour un dispositif donné, notamment lors de l'exécution de la fonction *Context_Adaptation*(), mais le `ContextManager` peut également informer la couche intergiciel du fait que la valeur d'un paramètre a changé dans un dispositif. Ceci permet de déclencher un nouveau processus de sélection au niveau intergiciel afin de sélectionner la configuration de ce niveau qui est la mieux adaptée au nouveau contexte.

Extensibilité

Dans le cadre de FACUS nous avons développé les classes qui permettent de prendre en compte le niveau de batterie afin de montrer l'acquisition et la prise en compte du contexte. Cependant, il est possible d'ajouter d'autres paramètres de contexte de façon simple. Pour cela, il faut :

1. créer une classe qui hérite de `ContextParameter` qui représente le paramètre en question ;
2. créer la classe qui hérite de `Widget` qui communiquera les valeurs du paramètre aux moniteurs qui la surveillent ; et
3. créer les méthodes ou classes qui permettent au *widget* d'acquérir la valeur mesurée du paramètre (par exemple une classe qui s'interface avec un capteur physique).

Finalement, il faut noter que, bien que dans FACUS le contexte est pris en compte par la couche intergiciel, les applications qui utilisent FACUS auront peut-être besoin d'acquérir des données contextuelles des dispositifs des utilisateurs aussi, et ce tant pour le contexte des ressources que pour

le contexte externe. Dans ce cas, ces applications peuvent bénéficier de l'architecture d'acquisition de contexte en utilisant le `ContextManager` depuis leur couche application spécifique.

4.3.3 Service de déploiement de composants

Une fois que la couche intergiciel a produit le graphe de déploiement, instance du modèle de niveau intergiciel, le déploiement effectif des composants est réalisé à partir de ce graphe. Le but de cette action est que les composants nécessaires à la mise en œuvre du schéma de collaboration décidé soient installés sur les dispositifs des utilisateurs, et que les liens entre ces composants soient établis correctement.

Dans Facus, ce travail est effectué par le *service de déploiement*, implanté par la classe `DeploymentService` de la Figure 4.18. Dans cette sous-section, nous décrivons le principe de fonctionnement de ce service de déploiement, dont nous avons fourni une implantation de base. Cette implantation va être étendue par la suite dans le cadre des travaux de notre groupe de recherche.

L'architecture du service de déploiement est illustrée par la Figure 4.21. Dans le même serveur que Facus, nous avons l'objet principal, `deploymentService`, et un registre qui contient les composants disponibles. On trouve un *agent de déploiement* dans chacune des machines des utilisateurs. Quand le service de déploiement reçoit un graphe de déploiement, il le parcourt afin de trouver les composants nécessaires pour chaque dispositif. Ensuite, il se connecte à chacun des agents, connexion matérialisée par les lignes hachurées de la figure, et il lui envoie un message qui lui signale la liste des composants qu'il doit installer. Il lui indique également à quels composants distants doivent être connectés chacun de ces composants. Ce message suit un protocole simple spécifique exprimé en XML.

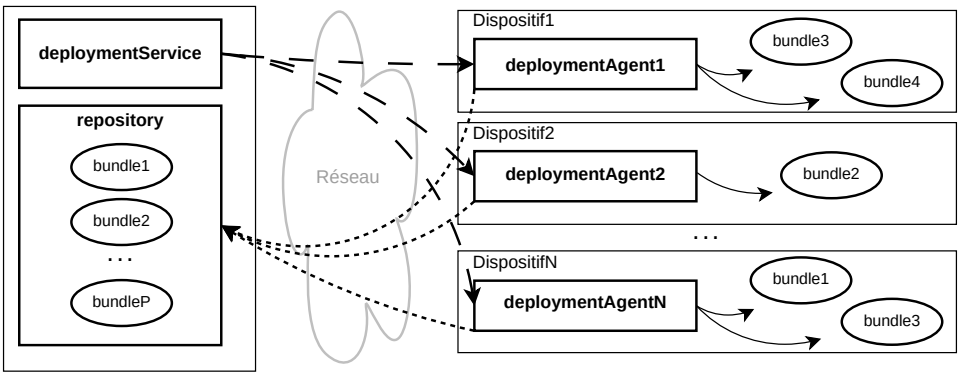


Fig. 4.21 – *Architecture du service de déploiement de composants dans* Facus

L'agent de déploiement de chaque dispositif utilisateur interprète le message reçu et déploie ensuite les composants indiqués. Si l'agent de déploiement détecte un composant manquant, il le récupère à partir du *repository* qui se trouve dans Facus. Ce téléchargement se fait par le protocole HTTP. Il est matérialisé par les lignes en pointillés de la figure. L'agent de déploiement peut notifier à d'autres composants, notamment au com-

posant principal de l'application, ces déploiements, afin qu'ils puissent se servir des composants nouvellement installés. Nous pouvons classer ce service de déploiement, par rapport aux critères cités dans la sous-section 2.2.2, comme dynamique, centralisé, et de type *push*.

La récupération, l'installation et le démarrage à chaud des composants sont basés sur la technologie *Open Services Gateway initiative* (OSGi, [OSG07]). La spécification OSGi fournit une plate-forme de gestion de modules et de services pour Java. Outre une gestion riche des services (avec un registre dynamique de services qui permet la détection, l'invocation, l'addition et la suppression de services en temps d'exécution), OSGi offre également un modèle de composants très complet.

Les composants de ce modèle s'appellent *bundles*. Un *bundle* est un groupe de classes Java (et éventuellement d'autres fichiers) empaquetées dans un fichier *jar*. Un *bundle* déclare notamment un ensemble de classes qui sont exportées (c'est-à-dire, qui seront visibles par les autres *bundles*) et un *activator*, qui est la classe qui est appelée lors du démarrage du *bundle*. Les *bundles* sont gérés par un conteneur OSGi.

L'apport le plus intéressant du modèle de composants d'OSGi est que les *bundles* peuvent être récupérés, installés, démarrés, arrêtés, mis à jour et désinstallés à chaud, c'est-à-dire, sans redémarrage du conteneur OSGi ni de la machine virtuelle Java. De plus, le répertoire de services est mis à jour lors de ces événements, ce qui veut dire, par exemple, que lors de l'ajout d'un nouveau *bundle*, les autres *bundles* seront informés des services offerts par le nouveau *bundle*, et ils pourront commencer à les utiliser.

Il existe plusieurs implantations de la spécification OSGi, telles que Equinox[4] (qui appartient au projet Eclipse), Felix[5] (de l'Apache Foundation) ou Knopplerfish[6]. La plupart de ces implantations sont des logiciels libres et très matures.

L'utilisation d'OSGi pour le service de déploiement dans FACUS comporte plusieurs avantages. Le fait de se trouver dans des environnements ubiquitaires rend très difficile d'avoir, au préalable et dans chaque dispositif, les outils qui seront nécessaires pour soutenir la collaboration. L'encapsulation des composants dans des *bundles* OSGi permet le fait que, si un utilisateur a besoin d'un certain composant, alors son agent de déploiement peut le récupérer, l'installer et le démarrer à chaud lors de l'exécution du système, d'une façon totalement transparente à l'utilisateur. Puisque les dispositifs dans ce type d'environnements ont souvent des capacités restreintes, les agents de déploiement peuvent décider de désinstaller des *bundles* qui ne sont pas nécessaires à un moment donné afin de libérer des ressources pour d'autres composants.

Cette utilisation d'OSGi implique que les composants déployés dans les machines des utilisateurs d'applications qui se servent de FACUS sont des *bundles* OSGi, notamment l'agent de déploiement lui-même et les *widgets* pour l'acquisition de données du contexte. Le code de l'application lui-même peut être distribué sous forme de *bundle*. Cela suppose que les dispositifs des clients doivent avoir au préalable une installation minimale

[4]Disponible sur http://www.eclipse.org/equinox/
[5]Disponible sur http://felix.apache.org/
[6]Disponible sur http://www.knopplerfish.org/

avec le conteneur OSGi et l'agent de déploiement afin que le reste des composants puissent être déployés dynamiquement.

Le `repository` de la Figure 4.21 est très simple à implanter avec les outils fournis par OSGi. Il suffit d'avoir un serveur HTTP qui contient les fichiers `.jar` des *bundles* et un descripteur XML appelé `repository.xml` qui contient la description des *bundles* disponibles. Ces éléments sont conformes à ceux décrits dans la spécification OSGi.

4.3.4 Outils, langages et technologies utilisés dans Facus

Facus est implanté en Java et donc les applications qui souhaitent l'utiliser doivent être également implantées en Java ou fournir une passerelle entre leur langage et l'API Java de Facus. L'implantation de Facus a été particulièrement complexe du fait que nous avons eu recours à divers outils et bibliothèques externes, la plupart d'entre eux écrits en Java. Nous avons également utilisé plusieurs langages de modélisation et les technologies qui leur sont associées.

Le Tableau 4.1 résume tous les langages, les outils, les bibliothèques et les technologies qui interviennent dans la conception et dans l'exécution de Facus.

Tâche	**Outils/langages/technologies**
Programmation	Java 1.6
Gestion d'ontologies	OWL 1, SWRL, Protégé-OWL API 3.4.1, Pellet 1.5.2, Jess 7.0
Gestion de graphes	GraphML, GMTE (passerelle en JNI pour accéder à l'API C++)
Gestion du contexte	Context Toolkit
Gestion de composants	OSGi (Felix)

Tab. 4.1 – *Outils, langages et technologies utilisés dans* Facus

4.3.5 Utilisation de Facus

Nous montrons ici les étapes que doit suivre un concepteur d'applications ubiquitaires afin d'utiliser Facus pour la prise en compte et l'implantation de la collaboration. Pour cela, une tâche de modélisation est à accomplir, ainsi que deux tâches d'implantation. Dans le chapitre suivant, nous illustrons ces étapes par la réalisation d'une application complète.

Étape 1. Création du modèle de niveau application

La première étape est dédiée à la modélisation de la collaboration dans l'application considérée. Cette modélisation produit une ontologie OWL de l'application qui contient au moins les éléments qui concernent la collaboration. Si le concepteur le souhaite, cette ontologie peut contenir également d'autres éléments, et il peut s'en servir pour contrôler la totalité de l'application. Cette ontologie doit importer et étendre GCO, tel que nous l'avons expliqué dans la sous-section 4.2.1. Cette ontologie peut contenir également des règles SWRL qui seront traitées par Facus.

Étape 2. Implantation du niveau application spécifique

Ensuite, il faut implanter le niveau application, sous la forme d'une classe concrète qui hérite de la classe `ApplicationLayer`. Cette classe doit créer une instance de l'ontologie de l'application et l'alimenter avec des individus qui représentent l'état de l'application à tout moment. Cette classe doit notamment implanter la méthode `generateDescriptor()`, qui encapsule l'instance de l'ontologie de niveau application dans un descripteur de niveau application (instance d'`ApplicationDescriptor`). Cette méthode devra être appelée à chaque fois que l'état de l'application change, afin de lancer le processus de raffinement dans FACUS qui finit par le déploiement des nouveaux composants nécessaires dans les dispositifs des utilisateurs.

Le niveau application peut utiliser l'architecture de contexte fournie par FACUS afin de prendre en compte des données contextuelles des dispositifs au niveau application.

Étape 3. Instanciation de Facus

Afin que l'application lance FACUS, il suffit de créer une instance du `LayerManager`, puis un objet de la classe qui correspond au niveau application. Cet objet doit recevoir comme paramètre le `LayerManager` instancié afin qu'ils puissent communiquer. Une fois cette mise en place terminée, FACUS est opérationnel. Il suffit d'appeler la méthode `generateDescriptor()` de la couche application pour que le processus de raffinement, puis le déploiement, se déclenchent.

L'application peut indiquer au `LayerManager` les paramètres de contexte à prendre en compte (sous-classes de `ContextParameter`) à travers la méthode `addContextParameter()` de `LayerManager`. Ces paramètres seront automatiquement pris en compte dans les `ContextMonitor` et les *widgets* nécessaires seront déployés dans les dispositifs des utilisateurs. Si le concepteur le souhaite, il peut ajouter de nouveaux paramètres de contexte et les *widgets* correspondants.

4.3.6 Fonctionnement de Facus et processus d'adaptation

Ici nous montrons d'une façon globale le fonctionnement de FACUS avec une application qui l'utilise. Cette explication intègre tous les concepts expliqués tout au long de ce chapitre.

La Figure 4.22 illustre l'architecture d'une application qui utilise FACUS et de FACUS lui-même lors de leur exécution. Dans la partie de gauche nous avons la machine sur laquelle la partie serveur de l'application et FACUS s'exécutent. Le code de FACUS et de l'application s'exécutent dans une même machine virtuelle Java. Le rectangle correspondant à cette machine virtuelle est divisé en deux parties afin de voir plus clairement les éléments qui appartiennent à l'application (en haut) et à FACUS (en bas). Dans la partie de droite nous avons les dispositifs des utilisateurs (`dispositif1` à `dispositifN`). Le réseau, au milieu du dessin, permet que toutes les machines puissent communiquer entre elles.

Dans chacun des dispositifs des utilisateurs se trouve un conteneur OSGi qui s'exécute dans une machine virtuelle Java. Ce conteneur permet

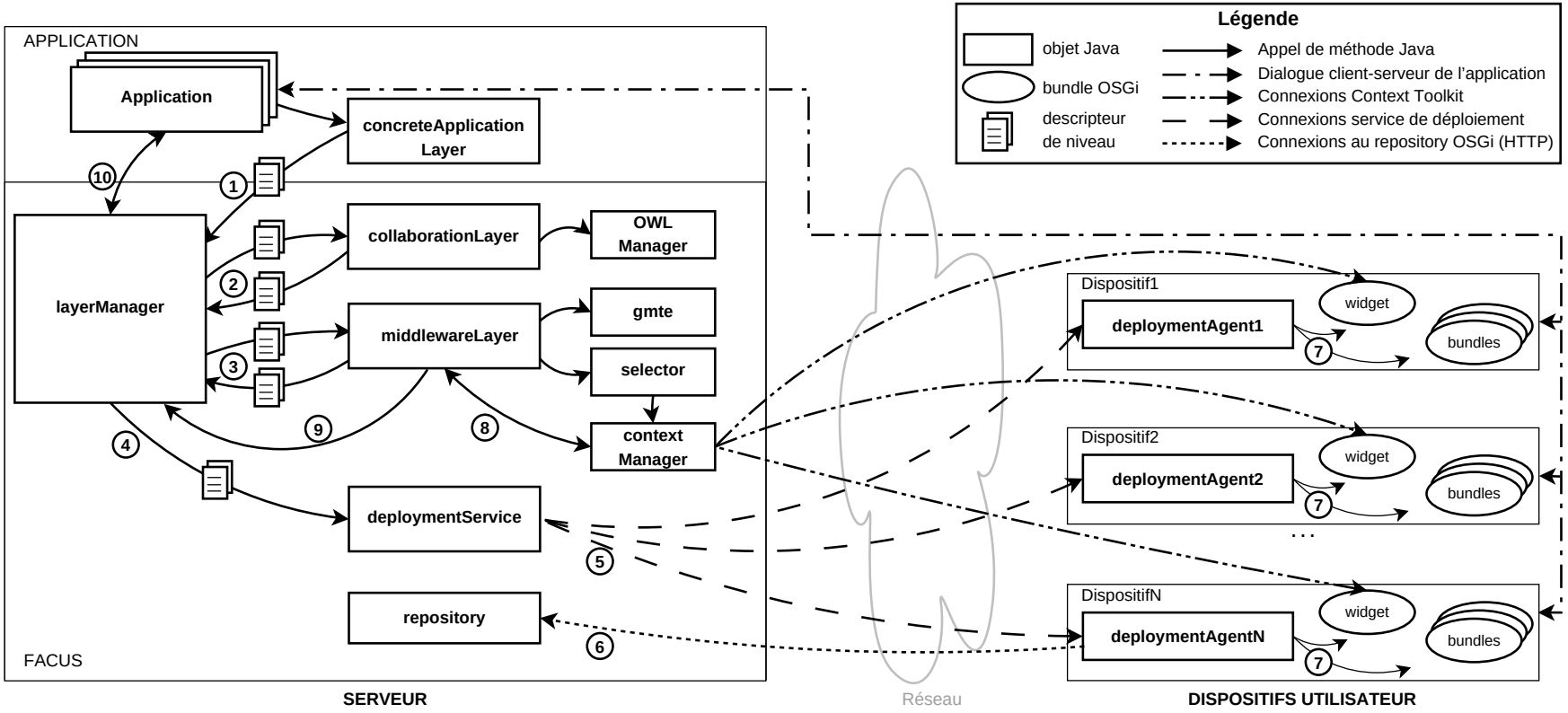


Fig. 4.22 – *Architecture de* Facus *et d'une application qui l'utilise*

l'exécution des composants qui sont encapsulés dans des *bundles* OSGi. Le composant principal est l'agent de déploiement (*deploymentAgent*) qui gère le cycle de vie des autres *bundles*. Parmi les autres *bundles* nous trouvons notamment celui qui correspond à la partie utilisateur de l'application et les *widgets* qui servent à l'acquisition du contexte. Les composants qui mettent en œuvre la collaboration (EP, EC et CM) font également partie de cet ensemble de *bundles*.

Dans la partie serveur, le code de l'application est représenté par l'ensemble d'objets étiqueté `Application`. Ce code de l'application dialogue avec les *bundles* client de l'application qui sont déployés sur les dispositifs des utilisateurs. Ces dialogues sont matérialisés par des lignes composées de traits et de points dans la figure.

Afin d'interagir avec FACUS, l'application instancie un objet de la classe `LayerManager` et un objet de la classe qui implante la couche application spécifique à l'application (objet `concreteApplicationManager` dans la figure). Cet objet maintient une référence vers l'instance de l'ontologie de l'application. À chaque fois que la structure de l'application change, par exemple quand un nouvel utilisateur arrive, cette instance est mise à jour, et la couche application déclenche le processus de raffinement. Elle le fait en encapsulant l'instance de l'application dans un `ApplicationDescriptor` qu'elle passe au `layerManager` (étape ① dans la figure). Cette utilisation de l'ontologie suit le schéma de boucle *capture–inférence–résultats* que nous avons proposé dans la section 3.5.

Le `layerManager` reçoit ce descripteur, il voit qu'il est de niveau 3, et donc il le passe à la couche collaboration pour qu'elle le traite (étape ②). La couche collaboration raffine la configuration de niveau 3 en appliquant le raisonnement et en exécutant les règles SWRL, avec Pellet et Jess, respectivement, qui sont utilisés à travers le `OWLManager`. Puis elle parcourt les individus de l'ontologie résultante pour générer le graphe de collaboration correspondant. Finalement, elle encapsule ce graphe dans un descripteur de niveau 2 (classe `CollaborationDescriptor`), qu'elle envoie au `layerManager`.

Le `layerManager` reçoit ce descripteur de niveau 2 et ensuite il l'aiguille vers la couche intergiciel afin qu'elle produise le descripteur de niveau 1 retenu (étape ③). La couche intergiciel lance, premièrement, le processus de raffinement en s'aidant du moteur de transformation de graphes (`gmte`) et de la grammaire $GG_{collab \rightarrow ebc}$. Elle obtient comme résultat un ensemble de graphes de déploiement qui implantent le graphe de collaboration reçu. En deuxième lieu elle lance le processus de sélection (avec le `selector`) qui choisit le meilleur des candidats en fonction du contexte actuel. Pour cela, le `selector` se sert des données contextuelles fournies par le `contextManager` et des politiques fournies par le `layerManager`. Le `contextManager` possède les données contextuelles car il surveille en permanence les *widgets* déployés dans les dispositifs à travers des `contextMonitors`. Une fois que le `selecteur` a choisi le candidat à retenir, la couche intergiciel l'encapsule dans un descripteur de niveau 1 qu'elle rend au `layerManager`.

Quand le `layerManager` reçoit le descripteur de niveau 1, il le passe au service de déploiement (étape ④). Le service de déploiement (`deploymentService`) parcourt le graphe de déploiement reçu et

contacte l'agent de déploiement de chacun des dispositifs présents dans le graphe (étape ⑤). Pour cela il utilise l'identifiant réseau des dispositifs contenu dans le graphe. Il indique aux `deploymentAgent` la liste de composants à mettre en place et les adresses des composants distants auxquels ils doivent être connectés.

Suite aux ordres reçus depuis le `deploymentService`, l'agent de déploiement de chacun des dispositifs des utilisateurs met en place les *bundles* correspondants aux composants nécessaires. Si des *bundles* sont absents du conteneur OSGi local, il les récupère depuis le `repository` de Facus (étape ⑥). Ensuite il installe, configure et démarre les *bundles* dans le conteneur OSGi local (étape ⑦), puis il signale au *bundle* principal de l'application que les nouveaux composants sont disponibles. Le *bundle* de l'application peut ainsi commencer à utiliser ces *bundles* pour envoyer et recevoir les données transmises à travers le réseau et qui implantent la collaboration entre les dispositifs des utilisateurs.

Adaptation aux changements de contexte

Quand le contexte change, l'architecture de collaboration mise en place doit s'adapter afin de répondre de la meilleure façon possible.

Les changements dans le *contexte externe* sont détectés par la couche application. Elle peut avoir connaissance de ces changements soit directement, par exemple lors de l'arrivée d'un joueur, soit à travers les notifications du `contextManager`, par exemple un changement de luminosité dans une pièce. La couche application réagit en effectuant des modifications dans l'état de l'application et puis elle met à jour l'instance du modèle de niveau application. Ensuite elle encapsule la nouvelle instance générée dans un descripteur de niveau 3 et elle le passe au `layerManager` (étape ①), ce qui déclenche un nouveau processus de raffinement et sélection qui provoque la mise en place d'un nouveau déploiement (étapes ② à ⑦). La réponse au changement de contexte produit donc une *adaptation verticale* ou *inter-niveau* car elle opère des modifications dans les configurations de plusieurs niveaux.

Les changements dans le *contexte de ressources* sont pris en compte, dans un premier temps, par la couche intergiciel. Le `contextManager` signale à cette couche les changements détectés par les `widgets` de contexte dans les dispositifs des utilisateurs (étape ⑧). Afin de s'adapter à ces changements, la couche intergiciel relance le processus de sélection sur l'ensemble de candidats qu'elle avait gardé lors du raffinement précédent afin de trouver le meilleur candidat adapté au nouveau contexte. Si la nouvelle configuration retenue est la même que la précédente, l'adaptation s'arrête là. Sinon, la nouvelle configuration de niveau intergiciel est encapsulée dans un descripteur de niveau 1 que la couche intergiciel passe au `layerManager` (étape ③). Celui-ci appellera le service de déploiement (étape ④) et le nouveau schéma de déploiement sera mis en place (étapes ⑤ à ⑦). Cette action est donc une *adaptation horizontale* ou *intra-niveau* car elle est gérée dans le seul niveau où le changement se produit.

Dans le cas précédent, si le processus de sélection détecte qu'aucune des configurations disponibles ne peut être implantée avec les niveaux actuels des ressources, cela veut dire que les exigences de la couche applica-

tion ne peuvent pas être satisfaites. En conséquence, la couche intergiciel signale ce cas au `layerManager` (étape ⑨), qui, à son tour, demande à la couche application de s'adapter à ces contraintes imposées par les ressources (étape ⑩). La couche application doit alors modifier l'instance du modèle de niveau application, ce qui résulte en une nouvelle adaptation verticale.

Nous constatons que, par rapport aux critères que nous avons présentés dans la sous-section 2.4.2, nous pouvons caractériser l'adaptation mise en œuvre dans Facus comme architecturale, dynamique, externe, basée sur les modèles et réactive.

Conclusion du chapitre

Dans ce chapitre, nous avons présenté notre approche de modélisation et son implantation, qui visent à combler, au moins partiellement, le manque de cadres conceptuels et de modélisation pour les SCUs.

En premier lieu nous avons proposé une approche de modélisation qui utilise des abstractions à plusieurs niveaux afin de soutenir une adaptation multi-niveaux, qui prend en compte tant les exigences des niveaux supérieurs que les contraintes des niveaux inférieurs. Nous avons également décliné cette approche, qui est générique, pour le cas des SCUs. Pour cela, nous avons identifié les niveaux de modélisation à prendre en compte dans ce type de systèmes et les éléments à considérer dans chacun de ces niveaux.

Ensuite, nous avons proposé le *framework* Facus, qui implante notre approche et qui peut servir de guide de modélisation et de conception, ainsi que de cadre d'exécution, pour des applications de type SCU. Nous avons détaillé les choix de modélisation mis en œuvre dans Facus et nous avons exposé son architecture et la façon dont nous l'avons implanté. Finalement nous avons expliqué comment il peut être utilisé par une application et nous avons détaillé le fonctionnement conjoint en ligne d'une telle application et de Facus.

Facus, avec l'approche qui le soutient, représente un pas important dans la conception de systèmes collaboratifs pour les systèmes ubiquitaires dans lequel l'accent est mis sur la prise en compte des données contextuelles et sur l'adaptation architecturale du système aux changements de ces données. L'effet ultime de cette adaptation est la mise en œuvre, lors de l'exécution du système, d'un schéma de déploiement capable de répondre aux exigences des applications et de leurs utilisateurs, tout en respectant les limitations imposées par les ressources. Ceci met en œuvre une interopérabilité sémantique entre les exigences et les contraintes de l'application.

Dans le chapitre suivant nous présentons un exemple d'application afin d'illustrer la relation entre Facus et les applications qui l'utilisent, ainsi que les instances concrètes des modèles utilisés à chaque niveau. A partir de cette application, nous évaluerons Facus tant d'un point de vue fonctionnel que de celui de ses performances.

Expérimentations

5

Sommaire

Ce chapitre porte sur les expérimentations que nous avons réalisées avec Facus afin de vérifier la validité de notre approche et de mesurer les performances de notre implantation. Nous présentons d'abord le contexte dans lequel nous avons réalisé ces expérimentations : le projet européen UseNet. Ensuite nous décrivons une application illustrative qui utilise Facus afin de mettre en œuvre une collaboration adaptative dans un environnement ubiquitaire. Cette application nous servira à instancier avec des modèles concrets le processus d'adaptation implanté dans Facus, ainsi que la façon dont il peut être utilisé depuis une application.

Une fois l'application développée, nous l'utilisons pour évaluer Facus selon le point de vue fonctionnel et selon les performances de la réalisation. Pour l'évaluation fonctionnelle, nous expliquons la façon dont l'application utilise Facus, avec notamment l'ontologie de niveau application spécifique, puis nous étudions les modèles produits dans chaque niveau de l'approche pour diverses configurations de l'application. Nous mettrons l'accent sur la réponse du système aux changements de contexte, tant externe que des ressources, qui se produisent lors du fonctionnement de l'application. Pour l'évaluation des performances, nous étudions le temps de réponse de Facus lors du lancement du processus d'adaptation par l'application. Ce temps de réponse déterminera la faisabilité de notre approche dans d'autres situations réelles.

5.1 Présentation du projet européen UseNet

Le cadre dans lequel nous avons conçu et développé l'application prototype que nous utilisons dans ce chapitre pour illustrer l'utilisation et le fonctionnement de Facus est le projet *Ubiquitous M2M Service Networks* (UseNet[1]).

Le projet UseNet appartient au programme de recherche et développement *Information Terchnology for European Advancement* 2 (Itea2). Les programmes Itea2 sont orientés vers la recherche et l'innovation dans le domaine du logiciel. Ils sont financés par les pays membres du réseau pan-européen de R&D *Eureka*, auquel Itea2 appartient.

Fig. 5.1 – *Logo du projet* UseNet

La durée du projet UseNet est de 36 mois. Il implique des acteurs tant du monde académique que du monde industriel qui proviennent de la France, de la Finlande, de l'Espagne et de la Belgique. Les principaux partenaires sont VTT (l'agence de recherche finlandaise, *leader* du projet), le LAAS-CNRS, Bull, Alcatel-Lucent, Thales, Ikerlan et Ideko.

La motivation de UseNet naît du besoin de gérer la grande quantité de dispositifs (par exemple des capteurs et des actionneurs) qui doivent être surveillés et contrôlés dans un monde d'informatique ubiquitaire. Ces dispositifs sont très nombreux et très hétérogènes en termes de connectivité et de capacité, ce qui rend leur gestion très complexe. Le terme *machine-to-machine* (M2M), qui apparaît dans le nom du projet, englobe tous les systèmes qui mettent en œuvre des techniques pour la gestion automatique de ces dispositifs et de leurs communications. Le but principal du projet est de fournir une plate-forme permettant l'accès aux services M2M d'une façon ubiquitaire et de montrer les usages possibles des applications basées sur ces services [LKH+08, RSM+10]. Les services M2M émergent à partir de l'acquisition, la transmission et le traitement de l'information produite ou consommée par les dispositifs. Le fonctionnement de ces services doit être le plus autonome possible afin de minimiser le besoin de contrôle humain.

Le projet se focalise sur plusieurs domaines où le M2M est applicable, tels que la surveillance et le contrôle à distance d'environnements ubiquitaires, la domotique, la maintenance et le contrôle de machines industrielles, les dispositifs personnels portables immergés dans les environnements ubiquitaires, les systèmes de télémesure fixes et mobiles, etc. La

[1] http://usenet.erve.vtt.fi/

partie commune du projet vise à construire une plate-forme de services générique, interopérable, réutilisable et valable pour tous ces domaines, ainsi qu'à fournir les infrastructures nécessaires pour son fonctionnement. Par exemple, la couche réseau virtuel permet la communication entre des dispositifs qui sont connectés à des réseaux différents. Les groupes de travail spécifiques se focalisent sur les applications et les services spécifiques aux domaines cités.

Dans le cas du LAAS-CNRS, nous sommes impliqués dans le groupe de travail dédié à la gestion des dispositifs personnels dans les environnements ubiquitaires. Ces dispositifs sont les machines personnelles qui accompagnent les utilisateurs immergés dans les environnements ubiquitaires que nous avons décrits dans ce mémoire : ordinateurs portables, *smartphones*, PDA, etc. La problématique principale est d'exploiter les caractéristiques qu'offrent les environnements ubiquitaires pour les utilisateurs qui possèdent ce genre de dispositifs.

Dans ce groupe de travail nous avons défini un scénario pour l'illustration des possibilités de ces environnements ainsi que des applications possibles [RSM+08, RSM+09]. Ce scénario se déroule dans le cadre des entreprises de transport urbain et inter-urbain. On considère que les bus de ces entreprises sont équipés avec des dispositifs tels que des capteurs, des actionneurs, des serveurs de contenus et d'applications, des passerelles (*gateways*), etc. Les passerelles fournissent une connectivité réseau qui peut être locale (par exemple avec des réseaux Wifi en mode infrastructure) mais aussi plus globale vers l'extérieur (autres bus et internet). Les dispositifs des utilisateurs peuvent communiquer directement entre eux et avec ceux qui sont déployés dans les bus. Tout cela fait que les bus deviennent des environnements ubiquitaires dans lesquels on peut construire des services pour l'entreprise de transport et pour les utilisateurs.

Dans ce scénario, nous avons proposé plusieurs applications, comme par exemple, la gestion de la flotte de bus et de la circulation, des systèmes de gestion du parcours des utilisateurs dans le réseau de transports, des systèmes d'information contextuelle (par exemple pour proposer des activités à proximité du bus), etc. Certaines de ces applications mettent en avant le côté collaboratif. Par exemple, nous trouvons les systèmes d'aide à la gestion du covoiturage entre les usagers des bus, qui leur permettent d'atteindre leurs domiciles après la descente du bus dans des zones interurbaines. D'autres applications fournissent des systèmes de divertissement, comme par exemple des jeux multi-joueur. L'application que nous avons utilisée pour illustrer Facus, que nous présentons dans la section suivante, est un de ces jeux collaboratifs.

5.2 Application d'illustration : jeu de bataille navale

Dans cette section nous présentons l'application que nous avons utilisée comme cas d'étude pour l'utilisation de Facus, ainsi que pour sa validation fonctionnelle et pour l'évaluation de ses performances. Cette application est un jeu collaboratif qui s'appelle *Collaborative Warships*, dont le prototype a été utilisé dans le cadre du scénario du projet UseNet. Nous avons spécifié et conçu cette application, qui a été ensuite développée par

M. Aymen Kamoun pour son stage de fin d'études [Kam10], réalisé sous notre encadrement.

5.2.1 Description de l'application

L'application *Collaborative Warships* est un jeu de bataille navale multi-joueur. Dans ce jeu, les utilisateurs contrôlent des bateaux regroupés dans des équipes. Chaque équipe essaye de faire couler les bateaux de l'équipe adverse.

Types de bateaux

Nous rencontrons trois types de bateaux : les *bateaux d'attaque* (*battle ships* ou BS en anglais), les *bateaux de réparation* (*repair ships* ou RS en anglais) et les *bateaux espion* (*spy ships* ou SS en anglais). Chaque bateau a deux attributs principaux : les *points de vie* et la *vitesse*. Selon le type de bateau, d'autres attributs sont considérés :

- pour les BS : la *portée de tir* (la portée maximale du feu du BS, en nombre de cases) et les *points de dégât* (nombre de points de vie que perd un bateau atteint par le feu du BS) ;
- pour les RS : la *vitesse de réparation* (nombre de points de vie que le RS peut restituer à un bateau de la même équipe par tour) ;
- pour les SS : la *portée de vue* (la distance maximale à laquelle le SS peut voir des bateaux ennemis).

À tout moment, un des BS d'une équipe est leur *chef* ; on l'appelle *battle ship leader* ou BSL. De la même façon, les RS ont un *chef* appelé *rescue ship leader* ou RSL.

Règles du jeu

Le champ de bataille est représenté par un tableau de 20×20 cases. Chaque bateau occupe une case unique à chaque instant.

Dans le jeu, deux équipes son présentes. Chaque équipe doit avoir au moins un BS pour jouer. À tout moment, chaque équipe a exactement un BSL, et, si l'équipe possède des RS, l'un d'entre eux est le RSL. Une équipe contient au plus 6 BS, 3 RS et 3 SS.

Pendant le jeu, un joueur connaît les positions de tous les bateaux de son équipe, et il ignore les positions des bateaux de l'équipe adverse (sauf pour le cas des SS ayant des ennemis dans leur portée de vue).

Les équipes jouent à tour de rôle. Pendant le tour d'une équipe, chaque bateau de l'équipe peut réaliser une seule action. Les actions possibles sont : mouvement (tous les bateaux), tir (pour les BS), et réparation (pour les RS). Seul un des BS de l'équipe peut effectuer une action de feu pendant le tour.

Un *mouvement* change la position d'un bateau. La case de destination doit se trouver dans un carré centré dans sa position et dont le rayon est la vitesse du bateau. Le joueur ne peut pas choisir comme case de destination une case occupée par un bateau de son équipe. S'il choisit une case qui est occupée par un bateau ennemi, celui des deux qui possède le moins de points de vie coule automatiquement.

Avec une action de *réparation*, un RS peut rendre à un autre bateau de son équipe des points de vie qu'il aurait perdus. Le nombre maximal de points de vie restitués par tour est indiqué par la vitesse de réparation du RS. Le bateau réparé ne peut pas dépasser son nombre initial de points de vie. La réparation ne peut être effectuée que si le RS et le bateau réparé se trouvent dans deux cases adjacentes.

Quand un BS effectue un *tir*, il doit indiquer la case cible (qui doit être à l'intérieur d'un rayon égal à sa portée de tir). Si un bateau se trouve dans la case cible, ce bateau perdra un nombre de points égal aux points de dégât du BS qui tire, et il sera informé du fait qu'il a été atteint. Le bateau qui tire sera informé du résultat de son action. Quand un bateau perd tous ses points de vie, il coule, et donc il arrête de jouer (mais il peut continuer à regarder la partie). La partie se termine lorsqu'une équipe perd tous ses BS, ce qui résulte dans la victoire de l'autre équipe.

Dynamique du jeu

Au cours de la partie, plusieurs actions peuvent déclencher un changement dans la configuration du jeu :

1. *Arrivée d'un joueur* : quand un joueur rejoint la partie, il faut décider dans quelle équipe et avec quel type de bateau il va jouer. Pour choisir l'équipe, on applique ces règles :
   – si les deux équipes ont le même nombre de joueurs, le joueur peut choisir l'équipe qu'il veut rejoindre ;
   – sinon, il jouera avec l'équipe qui a moins de joueurs.
   Pour décider du type de bateau du nouveau joueur, on applique ces règles :
   – s'il s'agit du premier joueur de l'équipe, alors le bateau sera un BSL ;
   – si le nombre de bateaux dans l'équipe est 1 ou 2, alors le bateau est un BS ;
   – sinon, le joueur peut choisir librement le type de son bateau. S'il choisit un RS et qu'il n'y en a pas d'autres dans l'équipe, alors il sera le RSL.
2. *Naufrage d'un bateau* : quand un bateau perd tous ses points de vie, il coule. Les cas particuliers où le bateau coulé est un BSL ou un RSL impliquent des changements dans le jeu :
   – BSL : si aucun autre BS n'est présent dans l'équipe, alors cette équipe perd. Sinon, le BS ayant le plus de points de vie sera le nouveau BSL.
   – RSL : si d'autres RS sont présents dans l'équipe, alors celui qui a le plus de points de vie sera le nouveau RSL. Sinon, l'équipe reste sans RSL.
3. *Départ d'un joueur* : si un joueur décide de quitter la partie, alors son bateau perd tous ses points de vie et il coule.
4. *Désistement d'un chef* : si un joueur qui est chef (BSL ou RSL) ne veut plus l'être, il peut se désister à condition que d'autres bateaux du même type soient présents dans son équipe. Dans ce cas, un nouveau BSL (resp. RSL) est choisi parmi les autres BS (RS) de l'équipe, et le BSL (RSL) précédent devient un BS (RS) normal.

Ces règles assurent que le nombre de bateaux des deux équipes reste équilibré, que chaque équipe a toujours un BSL, et que chaque équipe ayant des RS a un RSL.

Coordination au sein d'une équipe

Afin de maximiser les chances de gagner la partie, les membres d'une équipe doivent collaborer entre eux. En conséquence, il faut mettre en place des flux de données entre les joueurs. Selon le type de bateau de chaque joueur, il aura besoin de communiquer avec certains membres de l'équipe. Par exemple, si un BS atteint un ennemi avec son feu, il peut informer son BSL de la position de l'ennemi. De cette façon, le BSL pourra transmettre cette position à d'autres BS mieux placés ou avec plus de points de dégât pour qu'ils tirent à leur tour sur le bateau ennemi. De la même façon, le BSL peut demander à un SS d'aller sur une zone spécifique afin de traquer les ennemis, etc.

La Figure 5.2 représente les liens de communication possibles entre les membres d'une équipe. Chaque RS est relié au RSL de son équipe, tandis que chaque BS et SS sont liés au BSL. Les deux chefs de l'équipe sont également reliés entre eux, afin qu'ils puissent coordonner les tâches offensives et d'espionnage avec celles de réparation.

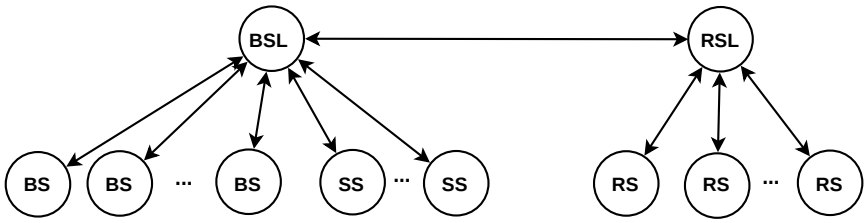


FIG. 5.2 – *Collaboration au sein d'une équipe*

Intérêt de l'application

L'application *Collaborative Warships* est un exemple d'application collaborative destinée à être exécutée dans des environnements ubiquitaires, en l'occurrence dans un contexte de transports en commun comme les autobus décrits dans la section précédente. En plus d'illustrer l'utilisation de FACUS par une application tierce, cette application présente deux besoins communs aux SCUs :

1. La sensibilité au contexte est nécessaire afin de détecter des changements dans le contexte externe (par exemple l'arrivée d'un joueur ou un changement de chef) et dans le contexte interne (par exemple dans le niveau de batterie des dispositifs personnels des joueurs).
2. L'adaptation, et plus spécifiquement l'adaptation de l'architecture de l'application (et du déploiement) est nécessaire afin de répondre aux changements du contexte. Par exemple, si un RSL se désiste, il faut changer tous les flux qui le reliaient aux autres RS de son équipe et mettre en place les flux entre ces RS et le nouveau RSL. Il faut également déployer les composants qui gèrent les nouveaux flux,

détruire les composants qui ne sont plus nécessaires pour libérer des ressources, etc.

5.2.2 Conception et implantation de l'application

L'architecture de l'application *Collaborative Warships* suit le principe que nous avons présenté dans la section 4.3, et qui est décrit par la Figure 4.22, pour les applications qui utilisent Facus : nous trouvons un serveur qui centralise les données du jeu, auquel se connectent les clients déployés dans les dispositifs des utilisateurs.

La partie serveur contrôle la logique du jeu. Elle gère les clients connectés, leurs actions, le temps de chaque tour, etc. Cette partie se sert de Facus pour implanter la collaboration et le déploiement de composants.

La partie client inclut l'interface utilisateur, qui gère la représentation graphique du jeu et l'interaction avec l'utilisateur, et la partie qui gère les échanges avec le serveur. Sur le même dispositif que cette partie client sera déployé l'agent de déploiement de Facus afin de gérer le déploiement des composants nécessaires pour la collaboration. L'agent de déploiement gère également les *widgets* qui obtiennent et transmettent les données de contexte.

Dans cette sous-section nous présentons la conception et l'implantation des parties serveur et client. Les détails de l'utilisation de Facus par le serveur, ainsi que l'ontologie de l'application, seront présentés dans la section suivante.

Partie serveur

La Figure 5.3 montre le diagramme de classes UML qui définit les classes principales du serveur de l'application *Collaborative Warships*. La classe principale est `WSGameManager`. Cette classe coordonne le fonctionnement des autres classes.

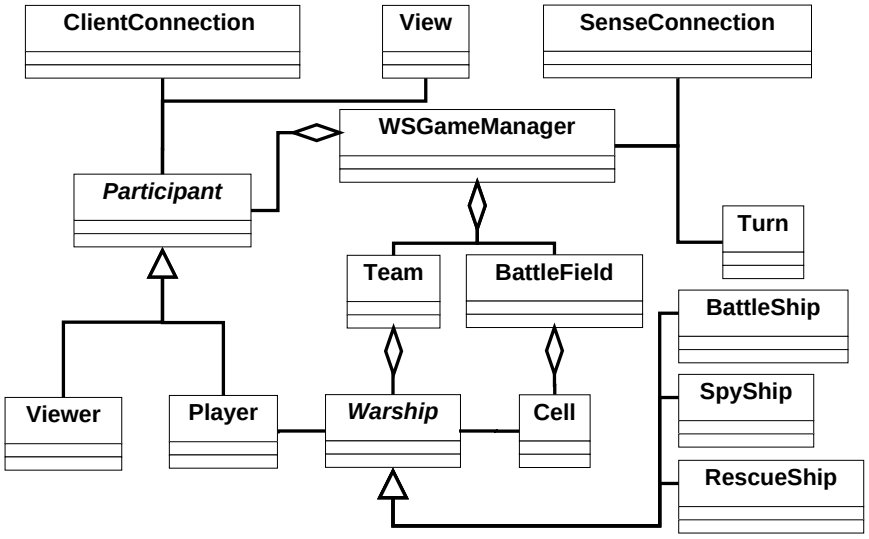


Fig. 5.3 – *Classes principales du serveur de l'application* Collaborative Warships

La classe `SenseConnection` est à l'écoute des connexions qui arrivent des clients. Quand une connexion est acceptée, le `WSGameManager` instancie un objet d'une des sous-classes de la classe `Participant`, qui encapsule les données du participant. Les participants peuvent jouer (classe `Player`) ou juste regarder le déroulement de la partie (classe `Viewer`). Les objets de la classe `Player` ont une référence vers un objet de la classe `Warship`, qui représente le bateau du joueur. Les trois sous-classes concrètes de `Warship` représentent les trois types de bateaux (`BattleShip`, `RescueShip` et `SpyShip`). Les classes `BattleShip` et `RescueShip` ont un booléen `isLeader` qui indique s'il s'agit d'un chef ou non. Les instances de la classe `Warship` sont liées a une instance de la classe `Team`, qui représente l'équipe à laquelle les bateaux appartiennent.

La classe `WSGameManager` possède une instance de la classe `BattleField` qui représente le terrain de jeu. Ce terrain de jeu est composé de 20×20 instances de la classe `Cell`, qui représentent les cases. La classe `Warship` a une référence vers l'instance de la classe `Cell` qui représente la case où se trouve le bateau.

Le `WSGameManager` possède également une référence vers la classe `Turn`, qui sert à gérer les tours de jeu. Cette classe possède un *timer* qui impose les changements de tour de jeu.

Partie client

Le diagramme de classes UML de la Figure 5.4 montre les classes principales du client de l'application *Collaborative Warships*. La classe principale, qui coordonne le fonctionnement des autres classes, est `WSGameClient`. La connexion avec le serveur est gérée par la classe `ServerConnection`.

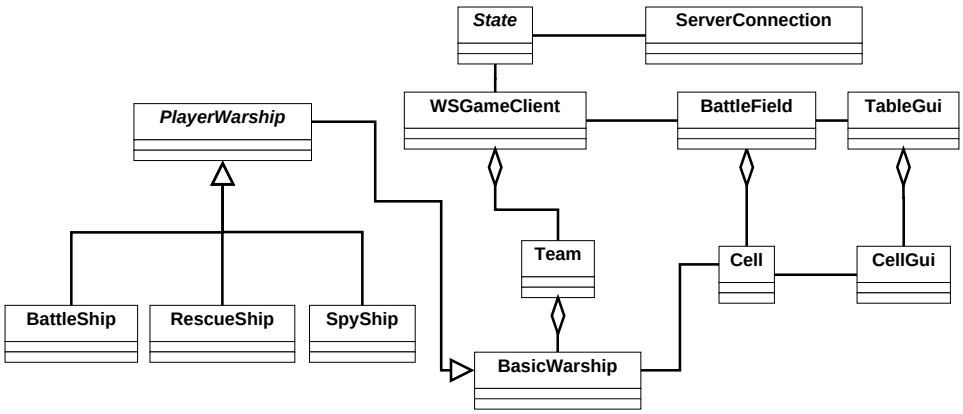


FIG. 5.4 – *Classes principales du client de l'application* Collaborative Warships

Les classes présentes dans le client représentent les informations et les traitements propres au joueur qui l'utilise, ainsi que les informations qu'il possède sur le reste du jeu. En conséquence, nous trouvons dans le client des classes équivalentes à celles qui utilise le serveur, notamment `Team`, `BattleField` et `Cell`. La classe équivalente à la classe `Warship` du serveur (qui est abstraite) est `BasicWarship`. Cette classe est concrète et elle sert à représenter les bateaux des autres joueurs, dont on ne possède que quelques informations. Le bateau du joueur en

question est représenté par la classe `PlayerWarship` (abstraite), qui hérite de `BasicWarship` et dont les sous classes correspondent aux trois types de bateaux (`BattleShip`, `RescueShip` et `SpyShip`). Ces classes contiennent toutes les informations du bateau du joueur.

Les classes qui représentent des objets qui s'affichent dans l'interface graphique du client ont un nom qui finit par `GUI`. Ces classes utilisent le *framework* graphique *Standart Widget Toolkit*[2] (SWT) de Java, qui fait partie du projet Eclipse. La Figure 5.5 montre l'interface graphique du client de l'application lors d'une partie. On y trouve, notamment, le tableau de jeu (à gauche), avec le bateau du joueur en surbrillance, et les boutons des actions (*fire*, *move*, etc.) en bas. Dans la partie de droite, on trouve les informations du joueur et du tour, ainsi qu'une zone où le joueur peut gérer les sessions audio et texte auxquelles il appartient.

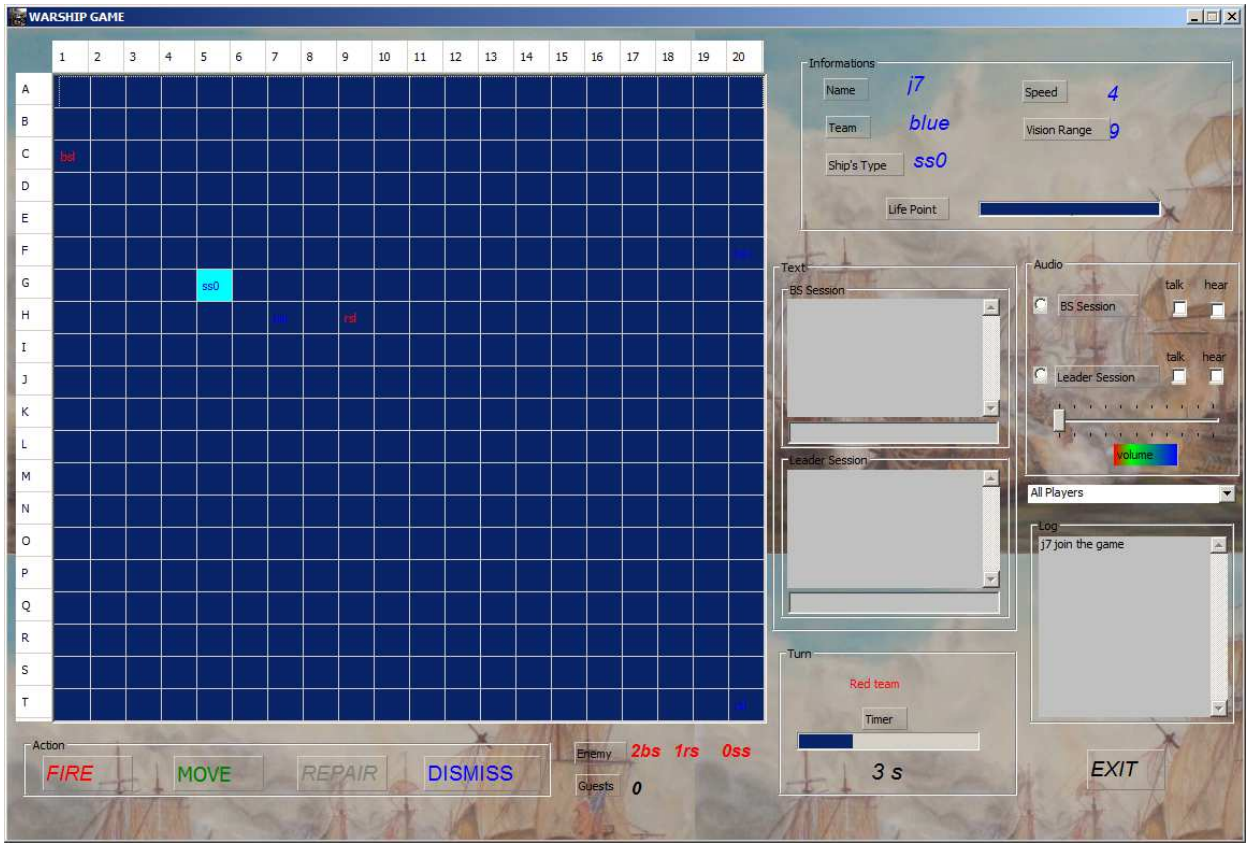


FIG. 5.5 – *Interface graphique du client de l'application* Collaborative Warships

Communication client–serveur

L'implantation des communications entre le client et le serveur est basée sur l'API *Java Message Service*[3] (JMS) ; nous avons utilisé l'implantation OpenJMS[4]. JMS est un intergiciel de type *Event-Based Computing* (EBC) mais qui peut être utilisé aussi en mode point-à-point. Les envois de don-

[2]Disponible sur `http://www.eclipse.org/swt/`
[3]`http://www.oracle.com/technetwork/java/index-jsp-142945.html`
[4]Disponible sur `http://openjms.sourceforge.net/`

nées se font vers des files d'attente dans lesquelles les producteurs écrivent et que les consommateurs lisent. Un des grands avantages de JMS est qu'il permet l'envoi de messages typés, c'est-à-dire que l'on peut construire des messages qui sont des objets Java. Le protocole de communication entre le serveur et le client est donc composé de plusieurs classes qui représentent les messages possibles entre le client et le serveur. Ces classes sont représentées dans la Figure 5.6. Toutes les classes de messages concrètes héritent de la classe `Message` qui définit les attributs et les méthodes communs. Parmi ces classes, on trouve notamment le message `NotifyMSG` qui est envoyé par le serveur vers le client afin de l'informer de l'état de la partie. Cette classe contient un objet de la classe `View` qui fournit au client sa « vue » de la partie avec toutes les informations qu'il a le droit de connaître.

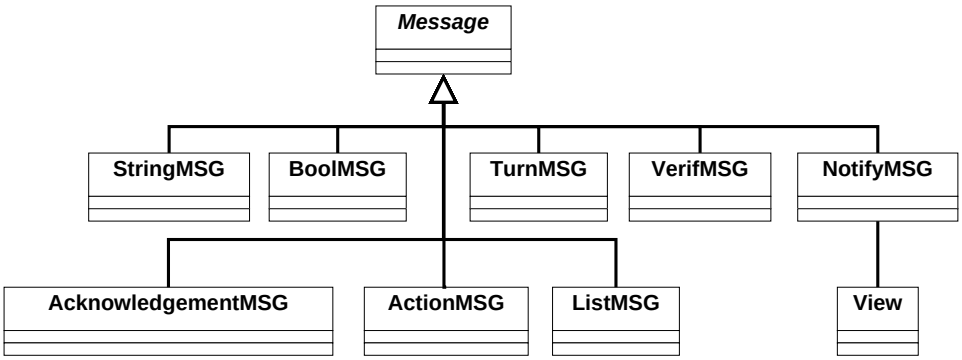


Fig. 5.6 – *Classes qui représentent les messages échangés entre le client et le serveur de l'application* Collaborative Warships

5.3 Évaluation fonctionnelle de Facus

Dans cette section nous évaluons le fonctionnement de Facus et son interaction avec l'application d'illustration. En premier lieu, nous décrivons l'ontologie de niveau application utilisée par l'application *Collaborative Warships*. Ensuite, nous montrons la façon dont l'application interagit avec Facus, entre autres avec cette ontologie, afin de lancer le processus d'adaptation multi-niveaux. Finalement, nous examinons les différents modèles produits à chaque niveau lors du déroulement d'un scénario de l'application.

5.3.1 Ontologie de niveau application

Afin de pouvoir utiliser Facus avec l'application *Collaborative Warships*, il est nécessaire de définir une ontologie de niveau application qui étend GCO et qui représente la collaboration entre les joueurs. Nous avons donc créé cette ontologie[5], qui est représentée par la Figure 5.7. Dans cette figure, nous avons représenté en gris les concepts et les relations de GCO, qui sont importés dans cette ontologie avec le préfixe `sessions`, tandis que ceux qui sont spécifiques à l'application le sont en noir.

[5]Le fichier OWL de cette ontologie est disponible sur `http://homepages.laas.fr/gsancho/ontologies/warships.owl`

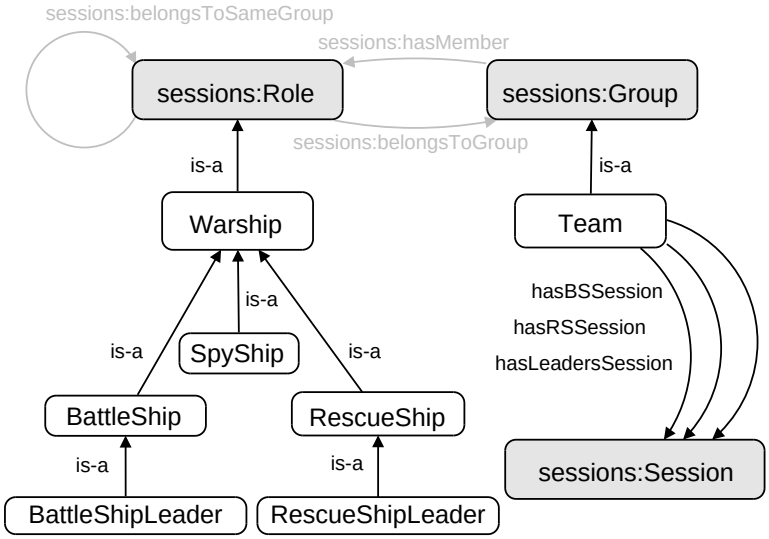


FIG. 5.7 – *Ontologie de l'application* Collaborative Warships

Cette ontologie est très simple : les concepts principaux représentent les bateaux (`Warship`) et les équipes (`Team`). Les trois types de bateaux possibles sont représentés par les sous-concepts de `Warship` : `BattleShip`, `RescueShip` et `SpyShip`. Les concepts `BattleShipLeader` et `RescueShipLeader`, sous-concepts de `BattleShip` et de `RescueShip` respectivement, servent à représenter les deux types de chefs possibles (BSL et RSL).

L'ontologie de l'application importe GCO, ce qui fait que les éléments de GCO sont *visibles* depuis cette ontologie. Cela est utilisé pour étendre GCO, car les concepts et les relations nouvellement définis ont des relations avec des éléments de GCO. Le concept `Warship` hérite du concept `Role` de GCO. En conséquence, les bateaux, ainsi que tous les types spécifiques de bateaux, sont des *rôles* au sens GCO : ils appartiennent à un nœud, ils peuvent être membres d'un groupe, etc. De la même façon, le concept `Team` hérite du concept `Group` de GCO, ce qui permet aux groupes d'avoir des membres, notamment. Les concepts `Warship` et `Team` ne sont pas reliés directement, mais ils le sont indirectement à travers leurs super-concepts. En conséquence, un individu de la classe `Team` peut être relié à un individu de la classe `Warship` (ou d'une de ses sous-classes) par la relation `hasMember`, et inversement avec la propriété `belongsToGroup`.

Trois propriétés relient le concept `Team` avec le concept `Session` de GCO : `hasBSSession`, `hasRSSession` et `hasLeadersSession`. Ces trois propriétés, qui sont des sous-propriétés de `hasSession`, représentent les trois types de conversations qui se déroulent à l'intérieur d'une équipe : celle entre les BS, les SS et le BSL, celle entre les RS et le RSL et celle entre le BSL et le RSL.

Règles SWRL de l'application

Les règles associées à l'ontologie de l'application *Collaborative Warships*, détaillées par la suite, servent à établir les flux de communication néces-

saires pour la collaboration entre les joueurs, selon le schéma que nous avons montré sur la Figure 5.2.

La règle `BS-BSL_audio_flows`, présentée par la Figure 5.8, sert à créer les flux audio entre les BS et le BSL. Cette règle trouve tous les couples de BS et BSL qui appartiennent à la même équipe, avec la propriété `sessions:belongsToSameGroup`. Le BS et le BSL trouvés ne doivent pas être le même individu, ce qui est imposé par la propriété `differentFrom`. Pour chaque couple trouvé, la règle crée deux flux audio qui relient le nœud du BS à celui du BSL et vice-versa. Les flux créés appartiendront à la session indiquée par la propriété `hasBSSession` de l'équipe.

```
BattleShip(?bs) ∧ sessions:Node(?nbs) ∧
sessions:hasRole(?nbs,?bs) ∧
BattleShipLeader(?bsl) ∧ sessions:Node(?nbsl) ∧
sessions:hasRole(?nbsl,?bsl) ∧
sessions:belongsToSameGroup(?bs,?bsl) ∧
differentFrom(?bs,?bsl) ∧
sessions:belongsToGroup(?bsl,?t) ∧
hasBSSession(?t,?s) ∧
swrlx:createOWLThing(?af1,?bs) ∧
swrlx:createOWLThing(?af2,?bs)
→ sessions:AudioFlow(?af1) ∧
sessions:hasSource(?af1,?nbsl) ∧
sessions:hasDestination(?af1,?nbs) ∧
sessions:belongsToSession(?af1,?s) ∧
sessions:AudioFlow(?af2) ∧
sessions:hasSource(?af2,?nbs) ∧
sessions:hasDestination(?af2,?nbsl) ∧
sessions:belongsToSession(?af2,?s)
```

FIG. 5.8 – *Règle `BS-BSL-audio_flows`*

Il existe trois autres règles analogues à celle que nous avons expliquée : `SS-BSL_audio_flows` relie les SS au BSL, `RS-RSL_audio_flows` fait de même pour les RS et le RSL, et finalement `BSL-RSL_audio_flows` relie le BSL et le RSL dans la session des chefs. La façon dont nous avons défini les éléments de l'ontologie rend la description des règles plus simple. En effet, si au lieu de caractériser les bateaux chefs en créant des sous-concepts des BS et RS, nous avions utilisé une propriété booléenne `isLeader`, l'écriture des règles aurait été plus longue, car il aurait fallu spécifier la valeur de cette propriété à chaque fois que l'on veut indiquer qu'un bateau est chef.

La mise en place de flux vidéo et texte est similaire à celle des flux audio que nous venons de décrire. L'utilisation de ces règles permet de mettre en œuvre le schéma de collaboration nécessaire à partir seulement des utilisateurs présents et de leurs rôles.

5.3.2 Utilisation de Facus par l'application

Afin d'utiliser Facus, l'application *Collaborative Warships* doit suivre les étapes présentées dans la sous-section 4.3.5.

1. définir une ontologie de niveau application ;
2. définir une classe concrète qui implante la couche application ;
3. lors de l'exécution de l'application, instancier un objet de la classe `LayerManager` et déclencher le processus d'adaptation à chaque fois que l'ontologie de l'application change.

Pour la première étape, nous avons détaillé l'ontologie de l'application dans la sous-section précédente. Pour la deuxième et la troisième étapes, la Figure 5.9 illustre les éléments concernés qui implantent ces étapes dans *Collaborative Warships*.

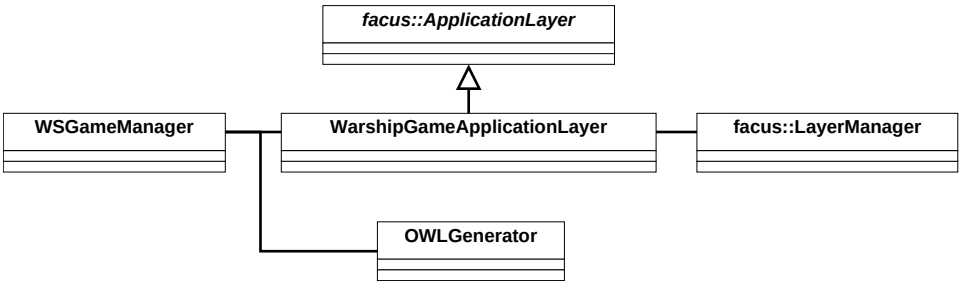


Fig. 5.9 – *Classes qui interviennent dans la communication entre l'application* Collaborative Warships *et* Facus

La classe `WarshipGameApplicationLayer` est la couche application spécifique qui hérite de la classe abstraite `ApplicationLayer`. Cette classe fournit une méthode `modelChanged()`, qui reçoit un modèle OWL en mémoire qu'elle encapsule dans un `ApplicationDescriptor` pour le passer à son `LayerManager`. Comme nous l'avons expliqué dans le chapitre précédent, cet appel lance le processus d'adaptation qui se termine par la mise en place d'un nouveau déploiement de composants de collaboration.

La classe principale de l'application, `WarshipGameManager`, possède une référence vers un objet de la classe `OWLGenerator`. Cet objet gère l'instance de l'ontologie de collaboration qui reflète l'état de l'application à tout moment. Les classes qui effectuent des modifications dans l'état de l'application délèguent à cet objet les modifications nécessaires dans l'instance de l'ontologie.

Lors de l'initialisation de l'application, `WarshipGameManager` instancie un objet de la classe `LayerManager`, qui est le point d'entrée vers Facus, et un autre objet de la classe `WarshipGameApplicationLayer`, à qui elle passe comme paramètre le `LayerManager`. Ensuite, elle instancie `OWLGenerator` et lui demande de construire l'instance initiale de l'ontologie d'application. Cette instance contient uniquement les deux équipes (individus du concept `Team`) et les trois sessions par défaut de chaque équipe (individus du concept `sessions:Session`). Les équipes sont reliées aux sessions par les propriétés `hasBSSession`, `hasRSSession` et `hasLeadersSession`.

À chaque fois qu'un objet modifie l'état de l'application, par exemple quand on ajoute un bateau à une équipe, cet objet demande à la classe `OWLGenerator` de modifier l'ontologie en utilisant les méthodes `createWarship()`, `changeRole()`, etc. fournies par cette classe. La nouvelle instance de l'ontologie obtenue est passée à l'instance de `WarshipGameApplicationLayer`, qui l'introduit dans un descripteur de niveau application et le passe ensuite à l'instance de `LayerManager`. Ceci déclenche le processus d'adaptation.

5.3.3 Déroulement de l'adaptation multi-niveaux

Nous décrivons ici les expérimentations que nous avons effectuées afin de valider le comportement de FACUS avec l'application *Collaborative Warships*. L'expérimentation consiste à étudier, pour plusieurs situations, le déroulement de l'adaptation depuis le moment où un changement se produit, au niveau application ou au niveau intergiciel, jusqu'à la production par FACUS du descripteur de déploiement à mettre en place. Ce déroulement a un but double : en premier lieu, il servira à illustrer le processus d'adaptation précisé dans le chapitre précédent avec une application concrète. En deuxième lieu il servira à vérifier si l'exécution de FACUS produit les résultats attendus.

Scénario et méthodologie

Pour évaluer le processus d'adaptation mis en œuvre dans FACUS, nous avons utilisé un scénario réaliste d'utilisation de FACUS depuis l'application *Collaborative Warships*. Ce scénario, décrit par la Figure 5.10, considère l'évolution d'une équipe dans le temps. Nous centrerons notre évaluation sur une seule équipe afin de rendre l'explication et les résultats plus clairs, sans perte de généralité car les équipes sont indépendantes entre elles et elles ont un comportement similaire.

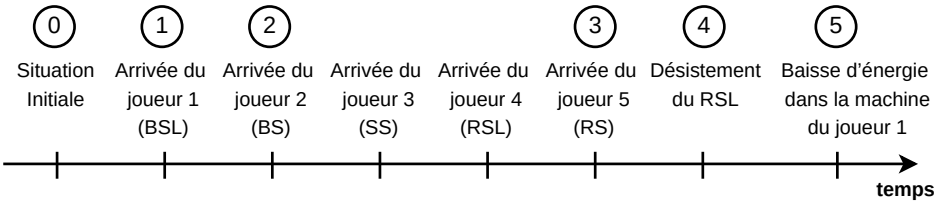


FIG. 5.10 – *Déroulement du scénario de référence*

L'axe horizontal de cette figure représente le temps. Sur cet axe, nous avons marqué quelques repères qui représentent les événements qui se produisent dans l'équipe. Ces événements correspondent à des changements du contexte externe (arrivées des joueurs, désistement des chefs) ou du contexte des ressources. Nous avons retenu quelques uns de ces événements ou *étapes* pour étudier l'adaptation du système. Les éléments retenus sont marqués de ⓪ à ⑤ dans la figure.

Pour chacune de ces étapes, nous avons créé l'ontologie de niveau application qui correspond à la configuration de l'équipe et nous l'avons

fournie comme entrée à FACUS, ce qui simule l'utilisation que fait l'application de FACUS. Ensuite, nous avons laissé FACUS générer les modèles de chaque niveau et nous les avons enregistrés dans des fichiers. Au niveau application, nous avons l'instance de l'ontologie de l'application. Au niveau collaboration, nous trouvons l'instance de GCO générée par l'application des règles et du raisonnement, ainsi que le graphe de collaboration correspondant. Au niveau intergiciel, apparaît le graphe de déploiement retenu après avoir appliqué la grammaire de graphes et le processus de sélection. Nous avons généré des images qui représentent ces modèles automatiquement à partir de la bibliothèque *Java Universal Network/Graph Framework*[6] (JUNG). Toutes les images que nous présentons dans la suite de cette section ont été automatiquement générées par des programmes Java qui utilisent JUNG pour afficher les modèles contenus dans les fichiers produits par FACUS.

Dans les ontologies de niveau application que nous avons fournies à FACUS, nous avons nommé les éléments avec des noms qui rendent plus facile la lecture des modèles générés. Le nœud correspondant à chaque joueur a comme nom `nodeN`, où N est le numéro du joueur. De la même façon, les identifiants des machines des joueurs sont de la forme `devN`. Le rôle joué par un joueur N sera de la forme `bslN`, `bsN`, `ssN`, `rslN` ou `rsN` en fonction du type de bateau. L'équipe considérée s'appelle `red_team`.

Afin de tester la sélection au niveau intergiciel, dont font partie les algorithmes de la fonction *Context_Adaptation*() et des politiques de distance et de dispersion, nous avons pris en compte le niveau de la batterie comme paramètre de contexte. Nous avons fixé la valeur de ce paramètre pour le dispositif de chaque joueur. Le Tableau 5.1 contient ces valeurs, qui sont exprimées en pourcentage de batterie restante. Afin d'isoler clairement les effets des changements dans le contexte externe et dans celui de la batterie, ces valeurs restent constantes entre les étapes 0 et 4. Dans l'étape 5, il se produit une baisse dans le niveau de batterie du dispositif de l'utilisateur 1. FACUS est paramétré pour considérer que le déploiement d'un CM consomme 4% de batterie, tandis que les EP et les EC consomment 3%.

Machine	Batterie (%) – étapes 0 à 4	Batterie (%) – étape 5
dev1	76	60
dev2	62	62
dev3	62	62
dev4	68	68
dev5	83	83

TAB. 5.1 – *Niveau de batterie des dispositifs des joueurs du scénario*

Résultats

Étape 0 L'étape 0 correspond à la situation de l'application juste après l'initialisation. Dans cette étape, l'ontologie de l'application ne contient que l'équipe `red_team` et les trois sessions de l'équipe (celle des BS et

[6]Disponible sur http://jung.sourceforge.net/

des SS, celle des RS et celle des chefs). La Figure 5.11 représente cette ontologie. Les individus sont représentés sous la forme de cercles gris, et leur nom est précédé du nom de la classe à laquelle ils appartiennent. Les instances des relations sont représentées comme des flèches entre les individus. Leur étiquette affiche le nom de la relation.

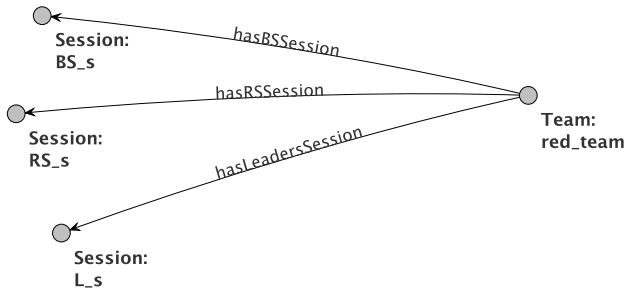


FIG. 5.11 – *Étape 0 – ontologie de niveau application*

Puisque l'équipe ne contient pas de membres, le traitement des règles produit un graphe de collaboration vide et le processus d'adaptation s'arrête là.

Étape 1 L'ontologie de niveau application de l'étape 1 (Figure 5.12) contient, en plus des éléments présents dans l'étape 0, ceux qui correspondent au joueur 1, qui a rejoint l'équipe. Ces éléments sont le nœud qui représente le joueur (`node1`), son dispositif (`dev1`) et le rôle qu'il joue (`bsl1`). Ce rôle appartient à la classe `BattleShipLeader`, car il s'agit d'un BSL. Le nœud est lié au dispositif avec la relation `hasHostingDevice`, et au rôle avec la relation `hasRole`. Puisque le rôle appartient à l'équipe, la relation `hasMember` relie ces deux individus.

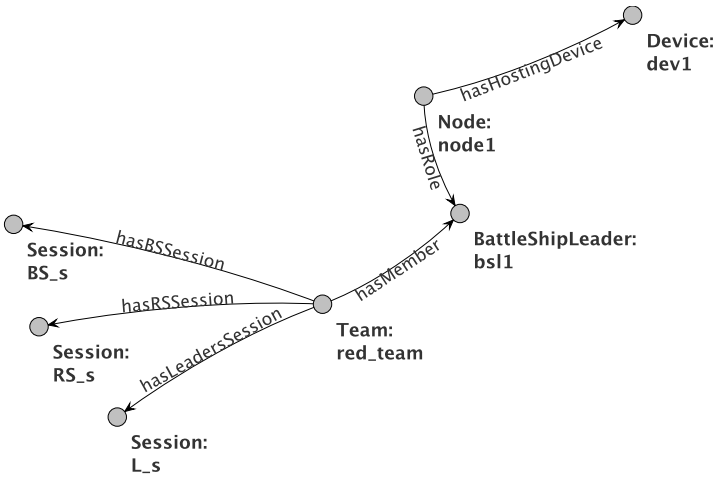


FIG. 5.12 – *Étape 1 – ontologie de niveau application*

Tout comme dans l'étape 0, le graphe de collaboration de cette étape est vide, car l'équipe ne contient qu'un seul membre, donc il n'y a pas de collaboration possible.

Étape 2 L'ontologie de niveau application de cette étape, représentée dans la Figure 5.13, ajoute le nœud du nouveau joueur arrivé (`node2`), son rôle (qui est un BS, `bs2`) et son dispositif (`dev2`).

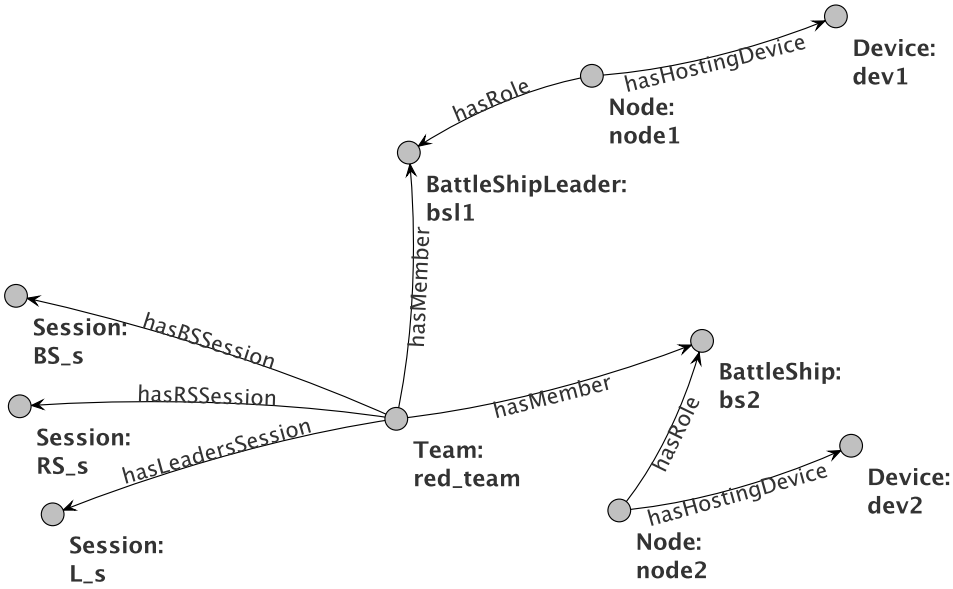


FIG. 5.13 – *Étape 2 – ontologie de niveau application*

Après le processus de raffinement du niveau application vers le niveau collaboration, nous avons une nouvelle instance de GCO, représentée par la Figure 5.14[7]. Comme cette figure le montre, les règles SWRL provenant de l'ontologie de l'application et de GCO ont créé des individus qui représentent les flux envoyés entre le nœuds (qui appartiennent à la session des BS, `BS_s`) et les composants qui gèrent ces flux. Les noms des individus générés par les règles sont de la forme `SWRLInjected_N`. Dans ce cas, `node1` envoie un flux `SWRLInjected_1` à `node2`, tandis que ce dernier envoie un flux `SWRLInjected_2` à `node1`. Dans `node1`, le composant d'envoi `SWRLInjected_5` envoie le premier flux, et le composant de réception `SWRLInjected_4` reçoit le deuxième flux. De la même façon, dans `node2` deux composants gèrent l'envoi et la réception des deux flux.

La couche collaboration extrait de cette instance de GCO les éléments nécessaires pour créer le graphe de collaboration, qui est représenté dans la Figure 5.15. On trouve les deux composants déployés dans `dev1` connectés aux deux composants déployés dans `dev2` par deux flux audio de la session `BS_s`.

La Figure 5.16 montre le graphe de déploiement retenu au niveau intergiciel. Dans ce graphe, un EC et un EP sont déployés dans le dispositif `dev1`. Ils correspondent aux composants de réception et d'émission présents sur ce dispositif dans le graphe de collaboration. De la même façon, un EP et un EC sont présents dans `dev2`. Tous ces composants échangent des données à travers un CM qui appartient à la même session et qui est déployé dans `dev1`. Cette configuration, avec le CM déployé

[7]L'ontologie de niveau collaboration contient également tous les éléments présents dans celle de niveau application (équipe, sessions, rôles, etc.), mais nous ne les avons pas inclus dans les figures pour les rendre plus lisibles.

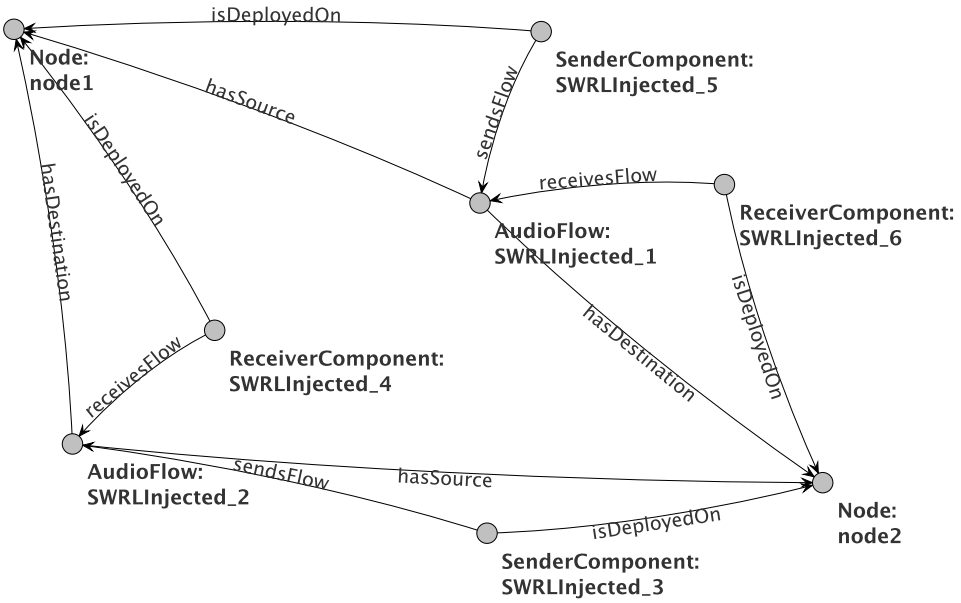


FIG. 5.14 – *Étape 2 – ontologie de niveau collaboration*

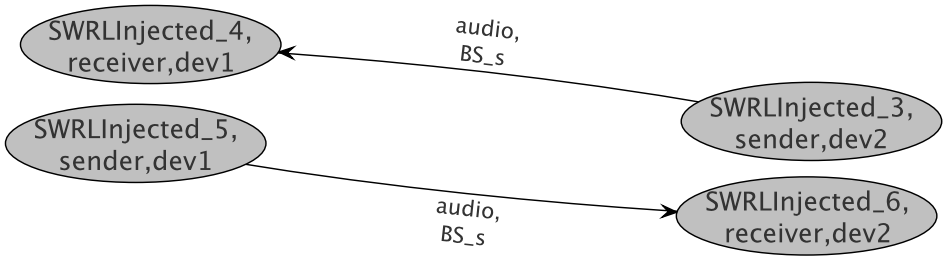


FIG. 5.15 – *Étape 2 – graphe de collaboration*

sur `dev1` plutôt que sur `dev2`, a été retenue parce que `dev1` a un niveau de batterie plus élevé que celui de `dev2`, ce qui implique que la fonction *Context_Adaptation*() lui donne la valeur la plus élevée.

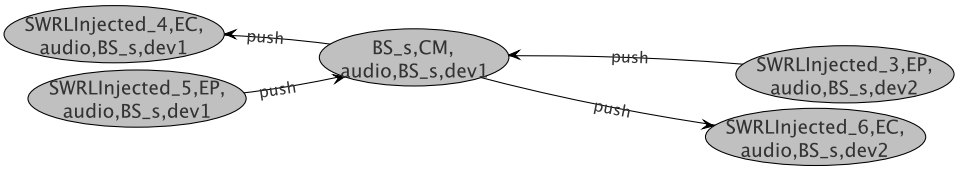


FIG. 5.16 – *Étape 2 – graphe de déploiement retenu*

Étape 3 Lorsqu'on arrive à l'étape 3, l'équipe contient un BSL, un BS, un SS, un RSL et un RS (`bsl1`, `bs2`, `ss3`, `rsl4` et `rs5`, respectivement). La Figure 5.17 représente l'ontologie de niveau application qui donne l'état courant de l'application.

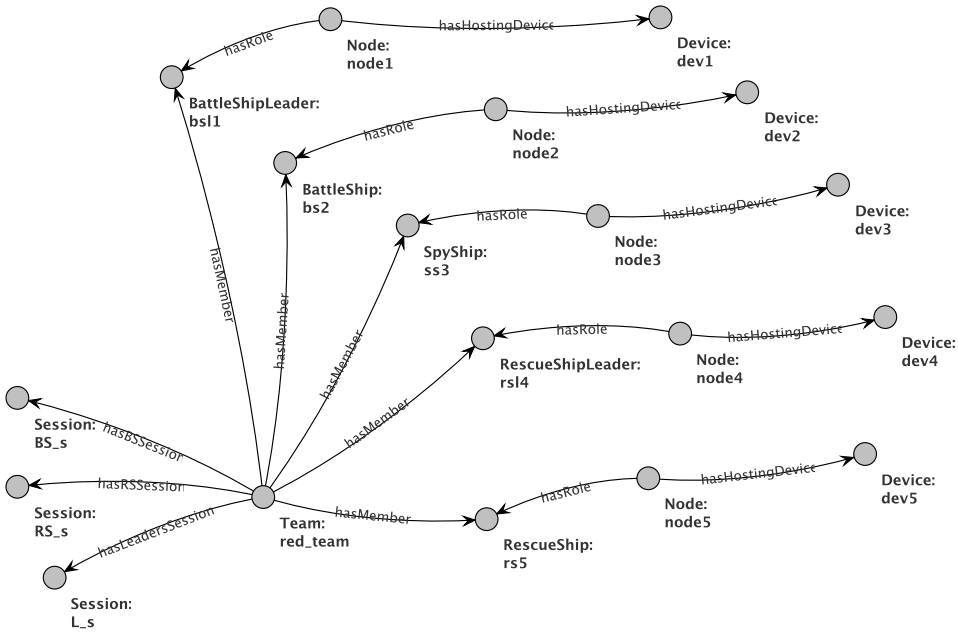


FIG. 5.17 – *Étape 3 – ontologie de niveau application*

La Figure 5.18 représente les individus de GCO créés par les règles SWRL du niveau collaboration. Cette figure est assez complexe car elle contient les 8 flux envoyés entre les 5 joueurs et les 15 composants qui gèrent ces flux. Nous ne trouvons pas 16 composants car les deux flux que `node1` envoie à `node2` et `node3` respectivement, qui appartiennent tous les deux à la session `BS_s`, sont gérés par le même composant d'envoi multiple `SWRLInjected_11`. La Figure 5.19 représente le graphe de collaboration équivalent, qui est beaucoup plus lisible car il ne contient que les nœuds, les flux et les composants.

Le processus de raffinement au niveau intergiciel produit 192 graphes de déploiement. Le graphe retenu par la sélection est représenté par la Figure 5.20. Puisque le dispositif `dev1` a beaucoup plus de batterie que

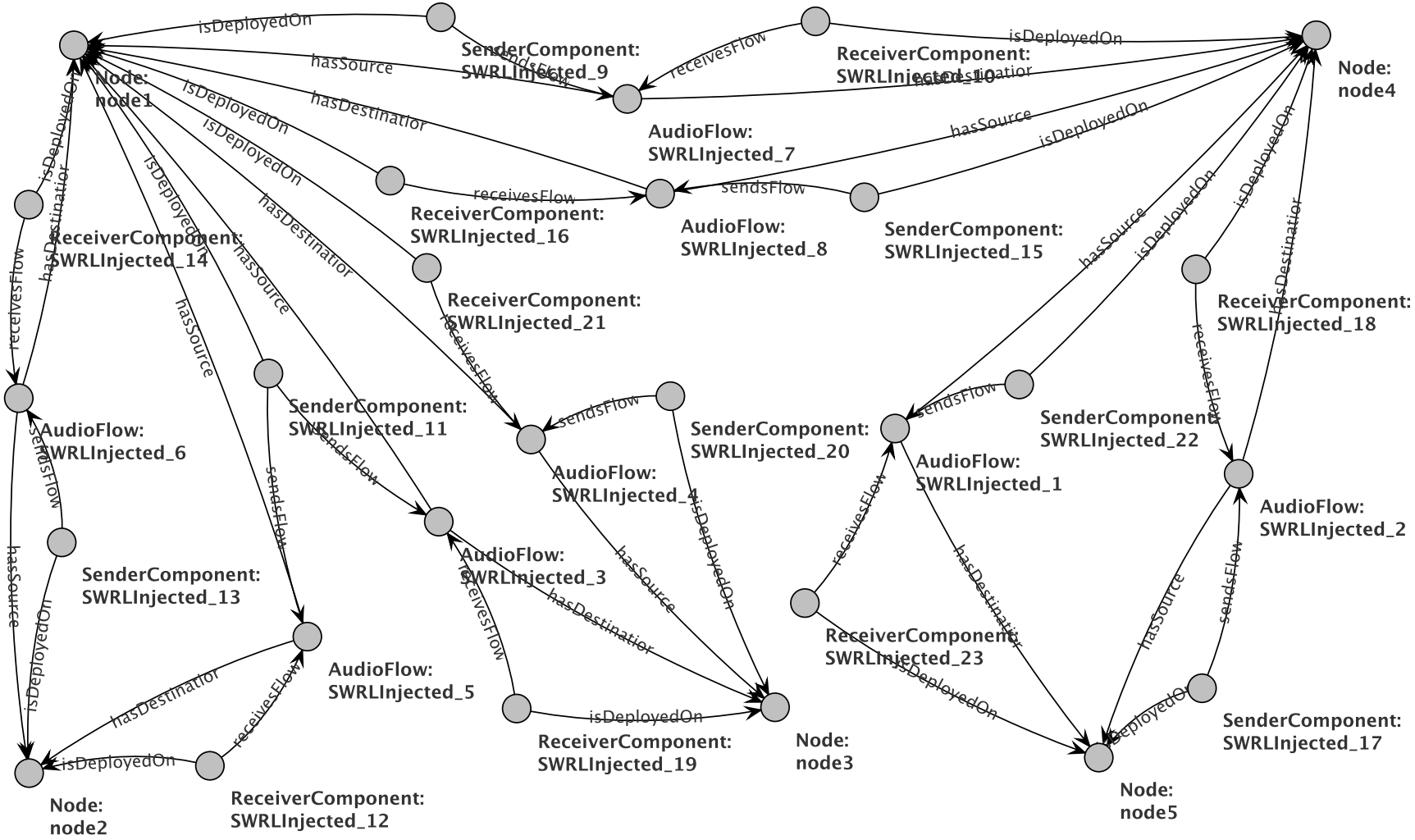


Fig. 5.18 – *Étape 3 – ontologie de niveau collaboration*

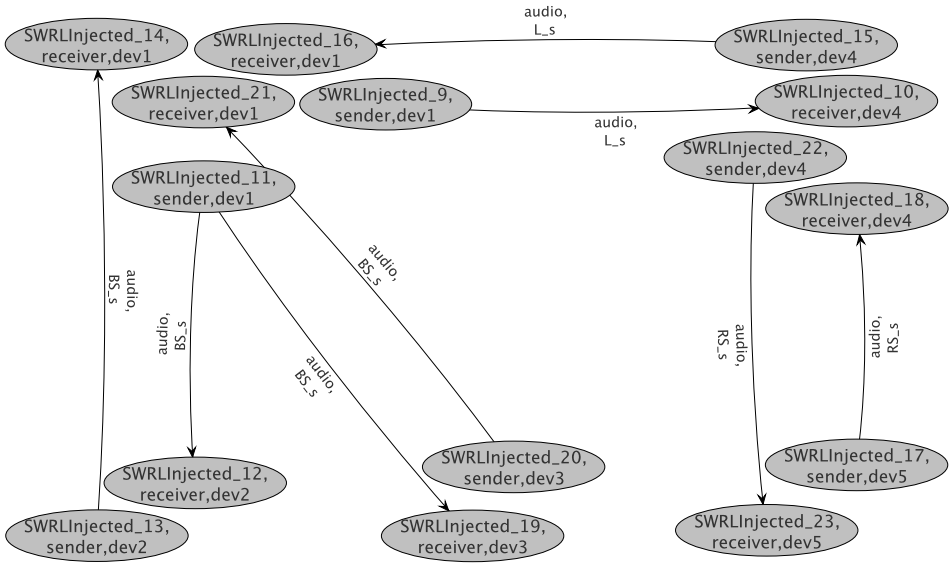


FIG. 5.19 – *Étape 3 – graphe de collaboration*

`dev2`, `dev3` et `dev4`, la configuration retenue a les CM des sessions `BS_s` et `L_s` sur `dev1`. De la même façon, le CM de la session `RS_s` est déployé sur `dev5`, qui a plus d'énergie que `dev4`.

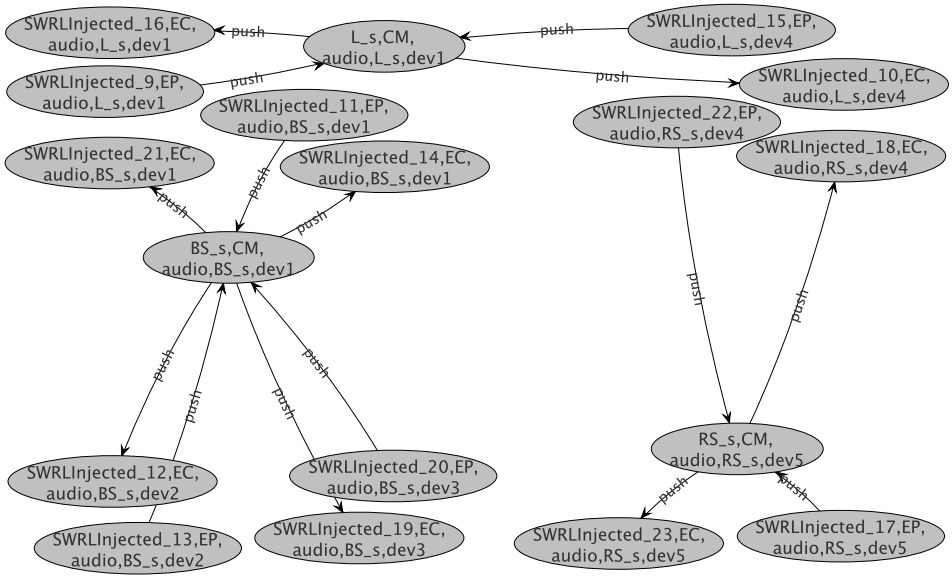


FIG. 5.20 – *Étape 3 – graphe de déploiement retenu*

Étape 4 Le passage de l'étape 3 à l'étape 4 résulte d'un changement dans le contexte externe de l'application : le joueur 4 se désiste de sa position de chef des RS pour devenir un RS normal. L'application choisit alors le joueur 5 comme nouveau RSL. La Figure 5.21 représente la nouvelle instance de l'ontologie de niveau application que l'application fournit à FACUS. On constate que le `node4` a maintenant comme rôle l'individu `rs4`,

qui est un `RescueShip` normal, tandis que le nouveau rôle de `node5` est le `RescueShipLeader` `rsl5`.

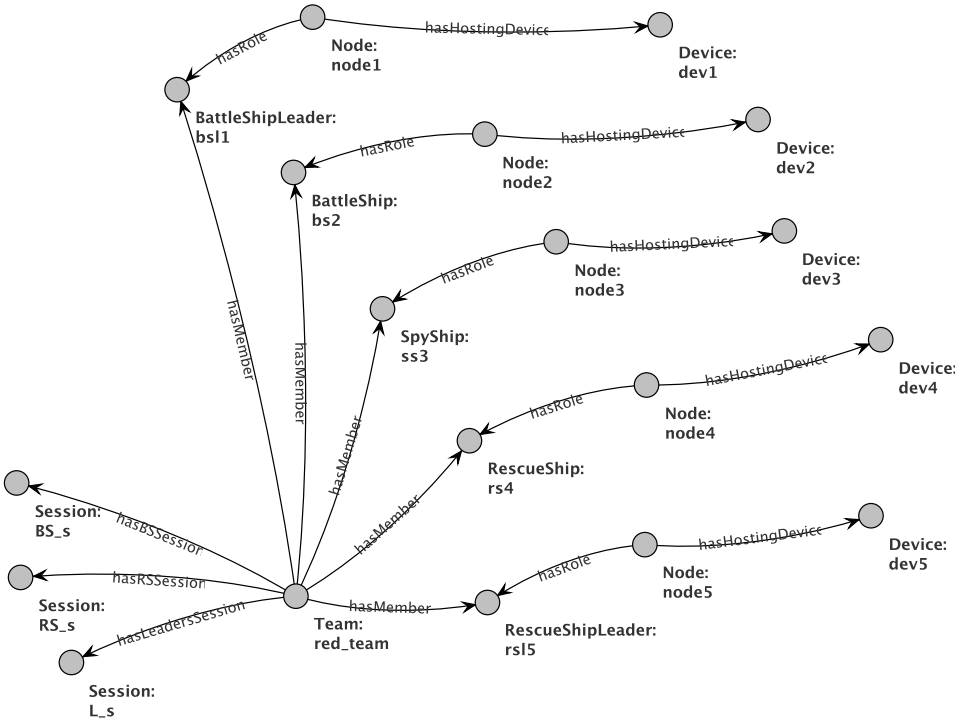


FIG. 5.21 – *Étape 4 – ontologie de niveau application*

La nouvelle instance de GCO générée au niveau collaboration est illustrée par la Figure 5.22 ; la figure Figure 5.23 montre le graphe de collaboration extrait. On constate que, par rapport à la Figure 5.19 de l'étape 3, les flux qui existaient entre `dev1` et `dev4` on été remplacés par ceux qui relient `dev1` et `dev5`. L'architecture de niveau collaboration s'est donc adaptée au changement de contexte externe.

Comme dans l'étape 3, le raffinement génère 192 graphes candidats au niveau intergiciel. On retrouve le même nombre car la taille de l'équipe reste la même. Le graphe de déploiement retenu par la sélection est celui que présente la Figure 5.24. Par rapport à la Figure 5.20, on remarque que, comme `dev5` a un niveau de batterie élevé et est maintenant RSL, alors on peut déployer sur cette machine le CM de la session des chefs, qui avant était sur `dev1`. Le CM de la session `BS_s` reste sur `dev1`, et le CM de la session `RS_s` est déployé sur `dev4`. La fonction *Context_Adaptation*() a choisi cette configuration car elle répartit les CM de telle façon que le dispositif dont le niveau de batterie est le plus proche du seuil est moins chargé que celui des autres candidats.

Étape 5 Lors du passage de l'étape 4 à l'étape 5, un changement de contexte des ressources, plus concrètement une baisse du niveau de batterie dans `dev1`, se produit. Comme nous l'avons expliqué dans le chapitre précédent, le *widget* qui surveille cette ressource dans `dev1` notifie ce changement au `ContextManager` à travers son `ContextMonitor`. Le `ContextMangager` relaie cette information à la couche intergiciel, qui re-

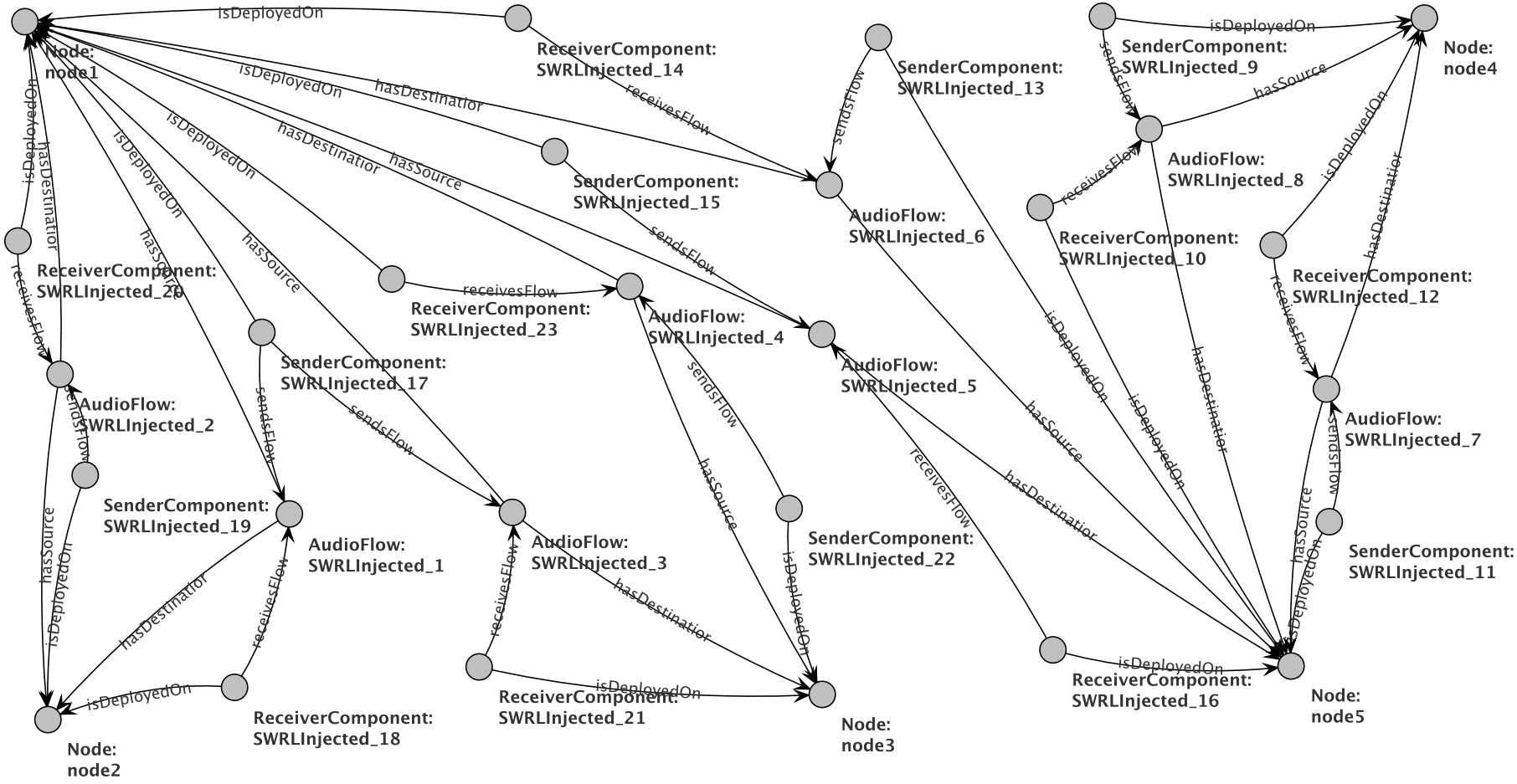


FIG. 5.22 – *Étape 4 – ontologie de niveau collaboration*

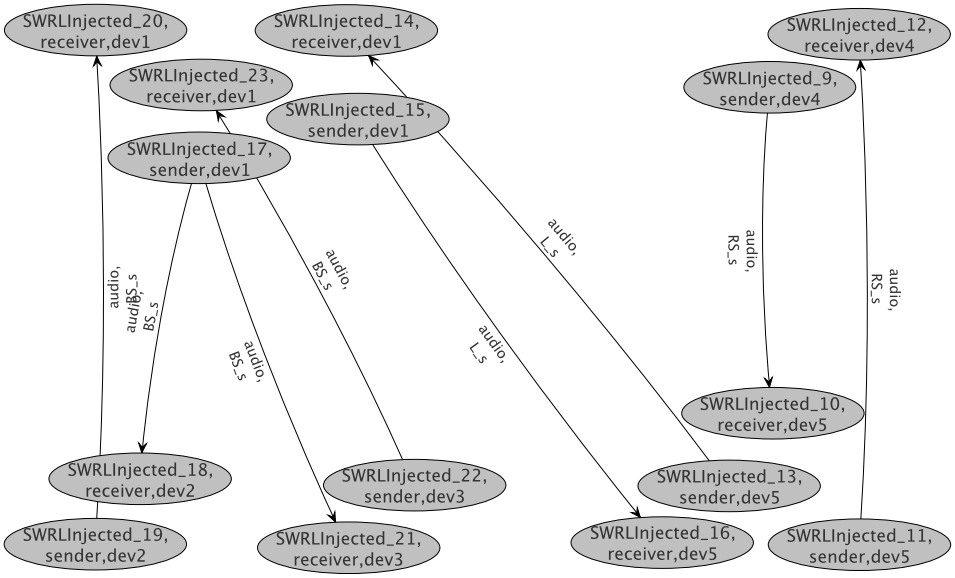


Fig. 5.23 – *Étape 4 – graphe de collaboration*

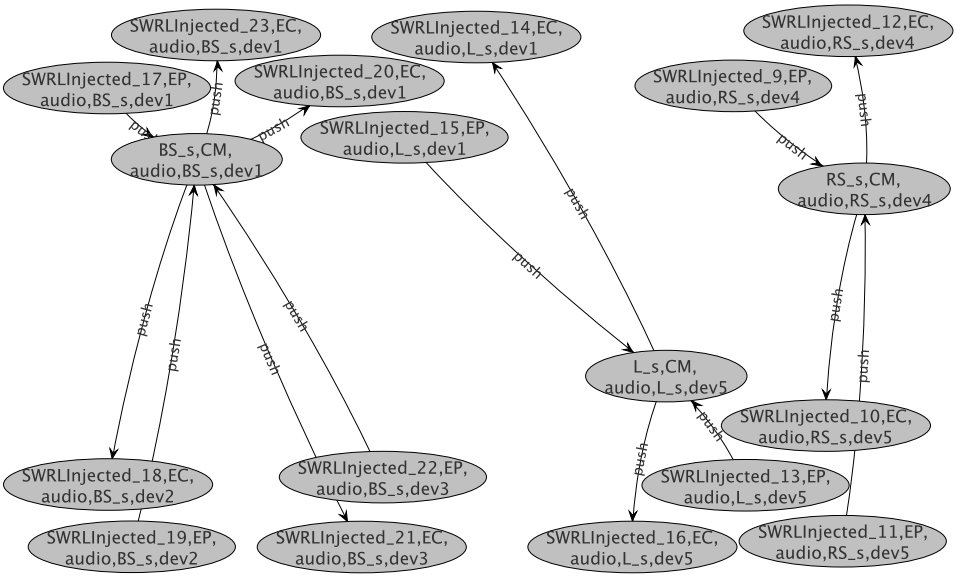


Fig. 5.24 – *Étape 4 – graphe de déploiement retenu*

lance le processus de sélection sur l'ensemble des 192 candidats qu'elle avait généré dans l'étape 4, afin de trouver celui qui est le plus adapté au nouveau contexte. La Figure 5.25 montre le nouveau candidat retenu. Par rapport à la Figure 5.24, le CM de la session `BS_s` a été déplacé de `dev1` vers `dev2`. Ceci est logique car `dev2` a maintenant un niveau de batterie qui lui permet de faire fonctionner ce CM plus longtemps que `dev1`. De cette façon, la couche intergiciel implante une adaptation intra-niveau pour répondre au changement du contexte de ressources ; les modèles des niveaux collaboration et application restent les mêmes.

Puisque le dispositif `dev3` a le même niveau de batterie que `dev2`, la configuration retenue aurait pu être également celle dans laquelle le CM de la session `BS_s` est déployé sur `dev3`. Ces deux configurations ont la même valeur de *Context_Adaptation*(), et elles sont identiques du point de vue des politiques de distance et de dispersion. Dans ce cas, l'une de ces deux configurations est choisie au hasard.

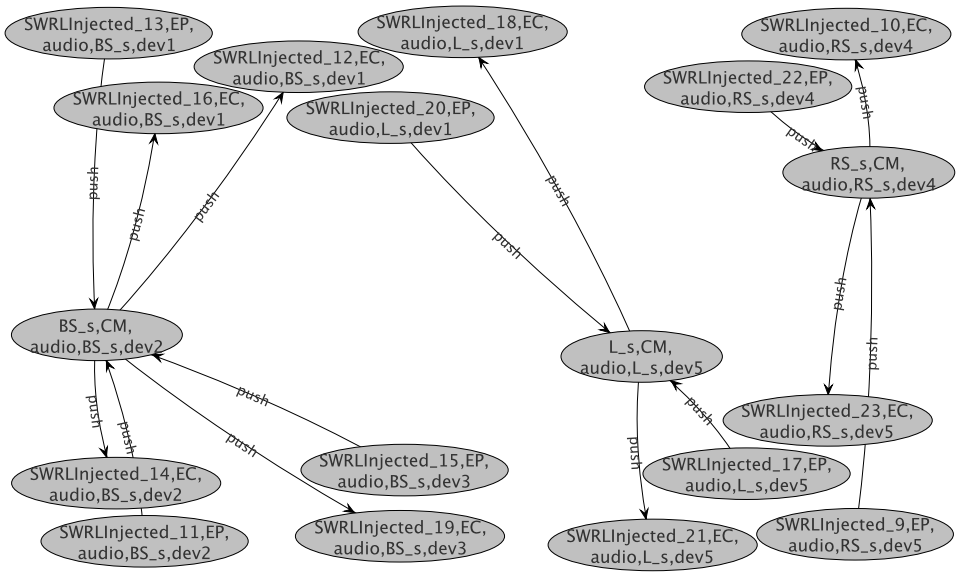


Fig. 5.25 – *Étape 5 – graphe de déploiement retenu*

5.4 Évaluation des performances de Facus

Le but de Facus étant de servir de prototype pour notre approche de modélisation et de conception des SCUs, nous avons mis l'accent sur ses fonctionnalités. Néanmoins, nous avons effectué une évaluation de ses performances afin de caractériser sommairement les temps de réaction de Facus dans des applications réelles.

Pour cette évaluation, nous avons considéré le temps d'exécution du processus d'adaptation multi-niveaux comme paramètre de référence. En effet, lors de l'exécution d'un SCU qui utilise Facus, ce paramètre déterminera si le déploiement mis en œuvre peut s'adapter assez rapidement aux changements du contexte. Si le temps de réponse reste raisonnable, le système pourra bénéficier des fonctionnalités de Facus sans que l'usabilité globale ne se voit compromise.

5.4.1 Temps d'exécution de l'adaptation

Scénario et méthodologie

Afin d'évaluer le temps d'exécution du processus d'adaptation dans Facus, nous utilisons le même scénario de référence que nous avons utilisé pour l'évaluation fonctionnelle, illustré par la Figure 5.10.

Dans chacune de étapes du scénario, nous avons mesuré quatre temps de référence :

- t_1 : le temps de génération de l'ontologie de niveau collaboration qui contient le résultat du traitement des règles SWRL et du raisonnement sur l'ontologie fournie par la couche application ;
- t_2 : le temps de génération du graphe de collaboration à partir de l'ontologie de collaboration ;
- t_3 : le temps de génération de l'ensemble des graphes de déploiement candidats au niveau intergiciel avec le moteur de transformation de graphes (c'est-à-dire, le temps du raffinement du graphe de collaboration en graphes de niveau intergiciel) ;
- t_4 : le temps de sélection du candidat retenu au niveau intergiciel avec la fonction de sélection.

Pour mesurer ces temps d'une façon qui n'affecte pas l'évaluation de Facus, nous avons créé une couche application de test qui lit des ontologies générées à l'avance dans des fichiers qui correspondent aux cinq étapes du scénario. Cette couche application lance ensuite le processus d'adaptation dans Facus. Dans ce test, nous avons obligé Facus à lancer ce processus dans un seul *thread* pour pouvoir mesurer tous les temps d'une façon séquentielle. Si nous n'avions pas fait ceci, Facus aurait rendu la main à la couche application juste après le lancement du processus d'adaptation, et l'application serait passée à l'étape suivante sans attendre la fin de l'exécution. Cela aurait annulé l'exécution en cours et dont invalidé les mesures. Nous avons également simulé l'existence des dispositifs clients en câblant leurs paramètres de contexte (dans ce cas, le niveau de batterie) dans le `ContextManager`. Ceci permet d'avoir des mesures fiables, qui ne sont pas polluées par les communications réseau ni par les performances de ces dispositifs.

Pour chacune des étapes, nous avons exécuté le processus d'adaptation 10 fois et nous avons calculé la valeur moyenne des temps mesurés. Ces valeurs moyennes sont utilisées dans notre évaluation.

Pour l'exécution des tests, nous avons utilisé une machine *Dell*® dotée d'un processeur *Intel Xeon*® *E5520* cadencé à 2.27 GHz, avec 8 Go de mémoire vive. L'utilisation moyenne de la mémoire lors de l'exécution des tests a été de l'ordre de 800 Mo.

Résultats

La Figure 5.26 montre les résultats globaux de notre expérimentation. Dans cette figure nous avons représenté, pour chacune des étapes, le temps total de l'exécution (c'est-à-dire, la somme des quatre temps de référence). Les figures suivantes se focalisent sur les divers temps intermédiaires. La Figure 5.27 montre le détail de t_1 et de t_2, c'est-à-dire le temps de génération de l'ontologie de collaboration et du graphe de collaboration. Afin

de voir clairement le coût de chacune de ces parties, ces deux temps sont empilés dans la figure. La somme de ces deux temps correspond en fait au temps total de traitement des ontologies. La Figure 5.28 montre le détail pour t_3, c'est-à-dire le temps du raffinement du niveau collaboration vers le niveau intergiciel. Finalement, la Figure 5.29 montre le détail pour t_4, c'est-à-dire le temps d'exécution de la sélection au niveau intergiciel.

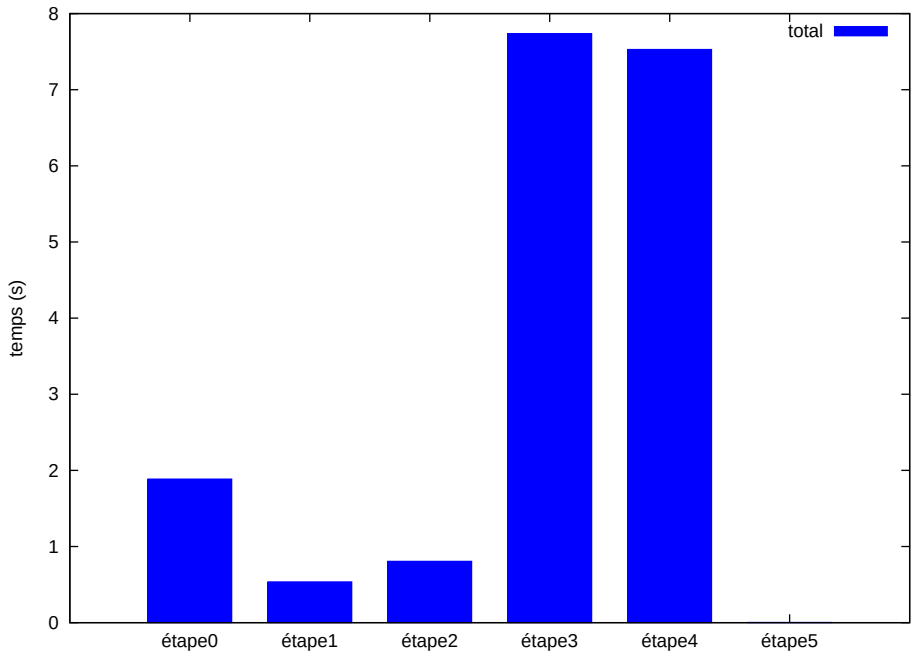


FIG. 5.26 – *Temps d'exécution – temps total*

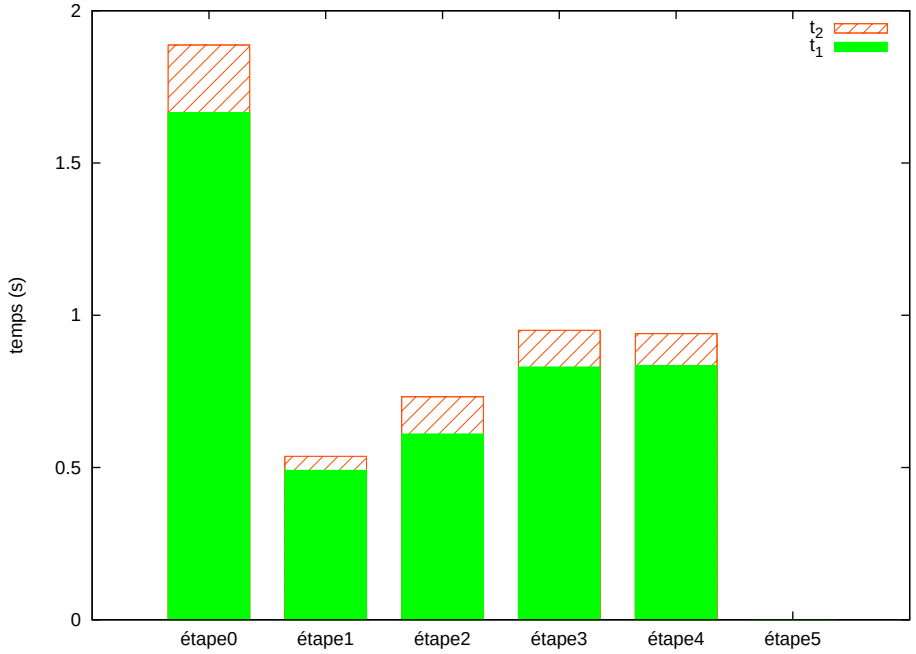


FIG. 5.27 – *Temps d'exécution – génération du graphe de collaboration*

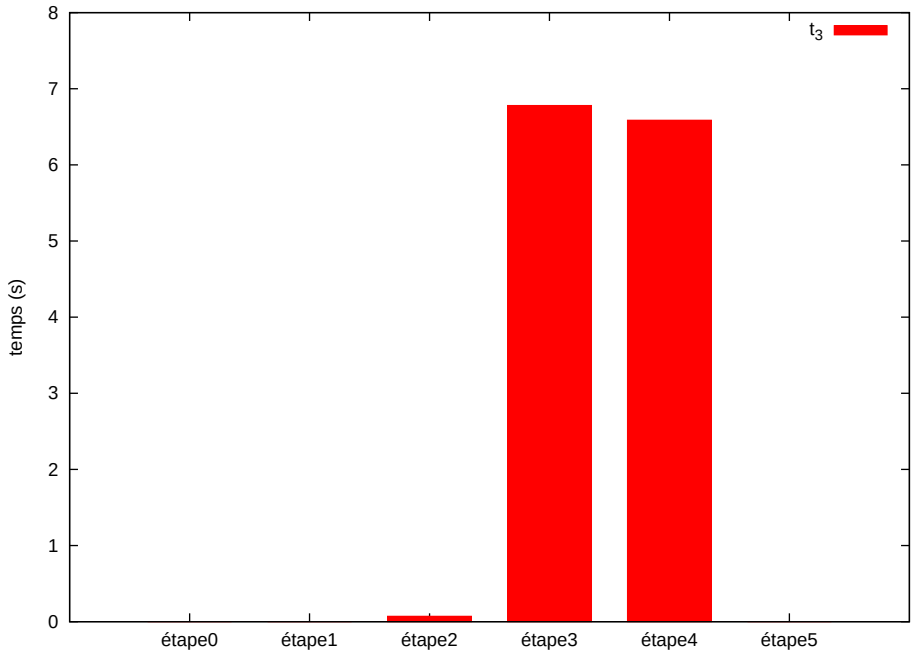


FIG. 5.28 – *Temps d'exécution – génération des graphes de niveau intergiciel*

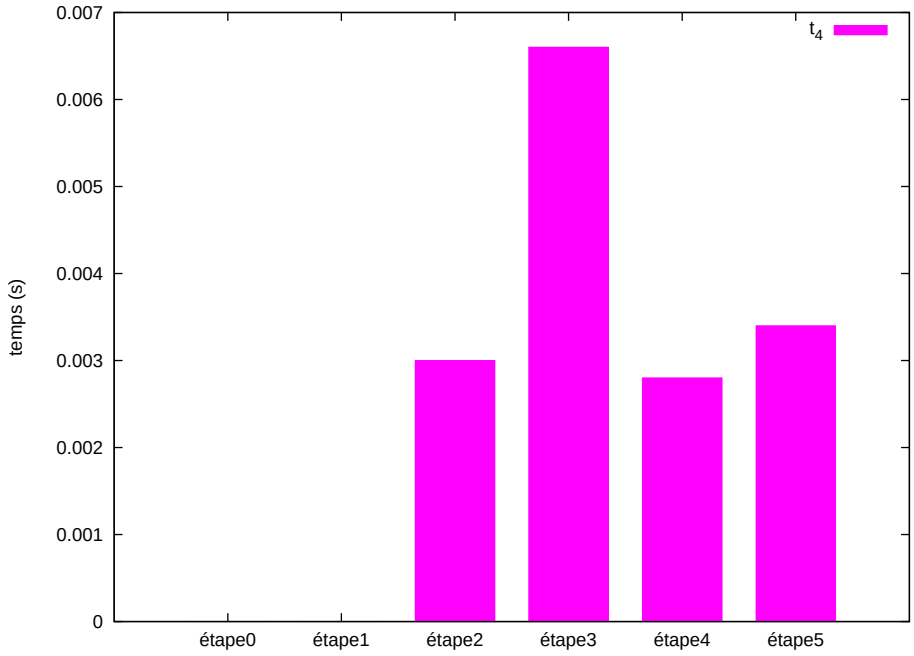


FIG. 5.29 – *Temps d'exécution – sélection au niveau intergiciel*

Comme les figures le montrent, dans les trois premières étapes, le temps prédominant est t_1. En effet, dans les étapes 1 et 2 le graphe de collaboration généré est vide, ce qui fait que le raffinement et la sélection au niveau intergiciel ne sont pas nécessaires, et donc $t_3 = t_4 = 0$. Dans l'étape 3, le graphe de collaboration est très simple ; en conséquence le raffinement et la sélection au niveau intergiciel sont très rapides et t_3 et t_4 restent petits par rapport à t_2.

Dans les étapes 3 et 4, la taille du graphe de collaboration est plus importante, ce qui fait que le processus de raffinement avec le moteur de transformation de graphes est beaucoup plus lent. En effet, l'application récursive des règles sur les graphes intermédiaires produit une explosion combinatoire qui ralentit le processus. Comme nous l'avons dit dans la section précédente, ce processus produit 192 graphes dans chacune de ces étapes.

Dans l'étape 5, le changement de contexte se produit au niveau intergiciel, ce qui veut dire que l'ontologie et le graphe du niveau collaboration, ainsi que l'ensemble des candidats du niveau intergiciel, restent inchangés, et donc $t_1 = t_2 = t_3 = 0$. En conséquence, le temps d'exécution de l'étape 5 correspond à t_4, qui est très petit.

Nous remarquons que le temps de génération de l'ontologie de collaboration (t_1) reste en dessous d'une seconde pour toutes les étapes sauf pour l'étape 0, où il est presque doublé. Nous avons pu réduire ce temps dans les autres étapes grâce à une optimisation dans la copie d'instances de l'ontologie de l'application. En effet, FACUS doit effectuer dans chaque étape une copie de l'ontologie de niveau application qui lui est fournie. Ceci est fait afin de laisser inchangée l'ontologie que l'application gère ; autrement, comme nous avons expliqué dans la section 3.5, la monotonicité d'OWL empêcherait le bon fonctionnement de l'inférence. Cette copie de l'ontologie est très lourde car elle porte sur tous les éléments de l'ontologie et elle implique des écritures et des lectures sur le disque. L'optimisation que nous avons mise en œuvre consiste à utiliser la copie stockée en mémoire lors de l'exécution précédente, sur laquelle on fait une copie des individus présents dans l'ontologie fournie par l'application. Le fait de copier seulement les individus, sans copier les définitions des concepts, des relations, etc. et sans passer par le disque, réduit d'environ 6 secondes le temps de la copie. La toute première exécution de l'étape 0 ne peut pas bénéficier de cette optimisation, car on ne dispose pas d'une copie précédente. Cela explique pourquoi l'étape 0 a une valeur d'environ 2 secondes (la moyenne sur les 10 échantillons réduit l'effet de la première mesure, qui est de plus de 6 secondes). Pour les autres étapes, t_1 grandit avec la taille du groupe, mais l'optimisation permet de rester en dessous de 1 seconde dans tous les cas.

Concernant le temps de sélection au niveau intergiciel, nous constatons qu'il est extrêmement petit (de l'ordre de quelques millièmes de seconde) y compris le cas des étapes 3, 4 et 5, dans lesquelles le nombre de candidats à traiter est de 192.

Conclusion du chapitre

Dans ce chapitre nous avons présenté un exemple d'application collaborative ubiquitaire qui se sert de Facus pour mettre en œuvre une collaboration adaptative. Cela nous a permis d'illustrer comment Facus peut être utilisé et comment créer une ontologie de niveau application qui étend GCO et qui peut être exploitée par Facus dans le processus d'adaptation. Ensuite, nous nous sommes appuyés sur cet exemple pour étudier le fonctionnement de Facus et caractériser sommairement ses performances.

L'évaluation fonctionnelle de Facus a été satisfaisante : les modèles produits à chaque niveau sont ceux attendus. Lors des changements de contexte, tant externe que des ressources, le système a répondu avec des adaptations pertinentes dans les modèles retenus à chaque niveau, qui garantissent à chaque moment que le déploiement répond tant aux exigences de l'application qu'aux contraintes matérielles. Les changements du contexte externe ont déclenché une adaptation inter-niveaux avec des modifications dans tous les niveaux de modélisation, tandis que les changements de contexte des ressources ont été gérés au niveau intergiciel, sans que les niveaux supérieurs ne se voient affectés.

Quant aux performances, le temps d'exécution du processus d'adaptation est resté dans des bornes raisonnables dans les expérimentations réalisées. Dans la plus lente des exécutions, ce temps est resté inférieur à 8 secondes, avec les autres en dessous des 2 secondes. Ce temps dépend de la taille du groupe et du type d'adaptation à réaliser (l'adaptation aux changements du contexte des ressources étant en dessous de la centième de seconde). Considérant que dans les groupes d'utilisateurs humains on n'aura que rarement des changements du contexte externe plus fréquents qu'un toutes les 15 secondes, ces temps sont suffisants pour que le système puisse suivre le rythme des changements.

La partie qui dégrade le plus le temps de réponse est celle du raffinement entre les niveaux collaboration et intergiciel, qui est dépendante du moteur de transformation de graphes utilisé dans Facus. Avec des tailles de groupe importantes, le nombre de combinaisons à gérer alourdit son exécution. Cependant, il est possible de réaliser des optimisations qui améliorent fortement les performances de cette partie. Nous esquissons ces optimisations dans le chapitre suivant.

Pour la partie collaboration, le temps est resté en général en dessous de 1 seconde, grâce aux techniques d'optimisation dans la copie d'ontologies que nous avons implantées. Ceci est un bon résultat par rapport aux plus de 6 secondes en moyenne que prendrait la solution sans optimisation. Le traitement des règles et le raisonnement sont assez rapides, en dessous de 1 seconde, grâce aux choix de conception que nous avons faits dans GCO et dans les règles.

La sélection au niveau intergiciel, avec l'algorithme d'adaptation au contexte et les politiques, est extrêmement rapide (de l'ordre de la milliseconde), même dans les cas qui comportent beaucoup de candidats. Cette partie est donc négligeable par rapport aux autres et le temps d'adaptation aux changements du contexte des ressources est imperceptible.

Conclusion générale

Dans ce travail de thèse, nous avons étudié la collaboration dans les environnements d'informatique ubiquitaire. L'informatique ubiquitaire propose une véritable révolution dans l'interaction entre les hommes et les ordinateurs. Ce nouveau paradigme induit de nombreux changements dans la conception et dans l'implantation des matériels et des logiciels qui sont à la base des ordinateurs. Bien qu'un grand chemin reste à parcourir pour arriver à la vision de l'informatique ubiquitaire de Weiser, une grande partie des techniques nécessaires pour ce changement, comme par exemple les communications sans fil, les interfaces tactiles ou la miniaturisation des composants, ont été étudiées dans les dernières années.

L'un des aspects de l'informatique ubiquitaire dans lesquels les défis de recherche sont encore nombreux est celui de la collaboration. Pourtant, les applications collaboratives sont parmi celles qui peuvent bénéficier le plus des caractéristiques des environnements ubiquitaires, comme leur grande dynamique, la disponibilité d'informations de contexte, la sensibilité à ces informations ou l'adaptabilité. Afin que les applications collaboratives puissent profiter de ces caractéristiques et être utilisées dans les environnements ubiquitaires, de nouveaux modèles et de nouvelles architectures sont nécessaires. Notre travail se situe au carrefour de ces deux domaines, l'informatique ubiquitaire et la collaboration, afin de réduire l'écart entre les techniques utilisées dans les applications collaboratives classiques, de type « bureau », et celles qui sont nécessaires dans les systèmes collaboratifs ubiquitaires.

Nous pensons que, pour effectuer cette transition vers les systèmes collaboratifs ubiquitaires, il faut utiliser deux éléments clés de l'informatique ubiquitaire : la sensibilité au contexte et l'adaptation. Dans notre travail, nous avons utilisé la sensibilité au contexte pour mettre en œuvre l'adaptation architecturale des logiciels collaboratifs. Nous avons eu recours à des techniques de représentation de connaissances, plus concrètement à des ontologies, afin de faire interopérer des exigences de haut niveau avec des contraintes de bas niveau dans le processus d'adaptation.

Bilan des contributions

Dans le Chapitre 2 nous avons introduit les principaux concepts abordés par les travaux existants dans les domaines qui concernent notre travail : l'informatique ubiquitaire, la collaboration, la sensibilité au contexte, l'adaptation et la sémantique. Nous avons également positionné nos contributions par rapport aux travaux qui existent dans la littérature et qui s'attaquent aux problèmes que nous avons abordés. Cela nous a permis de constater que peu de travaux s'intéressent, d'un côté, à la modélisation sé-

mantique de la collaboration, et, d'un autre côté, à l'adaptation architecturale des applications collaboratives dans les environnements ubiquitaires.

Dans le Chapitre 3 nous avons présenté notre modèle sémantique de la collaboration : *Generic Collaboration Ontology* (GCO). Ce modèle est une ontologie décrite avec le langage *Web Ontology Language* (OWL), dont nous avons expliqué les bases au début du chapitre. GCO permet d'exprimer, d'une façon indépendante du domaine considéré, des schémas de collaboration basés sur les sessions entre des entités au sein de groupes. L'atout majeur de ce modèle est que, grâce aux capacités de raisonnement et de traitement des règles qu'offrent les outils associés à OWL, on peut inférer, à partir d'une instance de GCO, un schéma de déploiement abstrait. Ce schéma facilite la gestion de la collaboration avec des composants qui s'envoient des flux de données. De plus, GCO est générique, et donc il peut être utilisé pour des domaines divers. Il suffit de définir des ontologies spécifiques au domaine considéré qui se basent sur GCO.

Le Chapitre 4 a été dédié à la présentation de notre *framework* pour la gestion adaptative de la collaboration dans les systèmes ubiquitaires, Facus. Facus facilite le déploiement adaptatif des composants qui instancient les sessions collaboratives pour les applications ubiquitaires. Nous avons présenté une approche générique de modélisation architecturale qui sous-tend Facus et qui prend en compte l'adaptation à plusieurs niveaux. L'apport principal de cette approche est qu'elle isole la prise en compte des données de contexte dans le niveau adéquat, ce qui permet, grâce aux mécanismes de *raffinement* et de *sélection*, d'implanter une adaptation multi-niveaux. Nous avons instancié cette approche générique pour le cas des systèmes collaboratifs ubiquitaires, pour lesquels nous avons retenu trois niveaux : *application*, *collaboration* et *intergiciel*.

Ensuite, nous avons détaillé les modèles que nous avons utilisés dans Facus pour les trois niveaux, ainsi que les transformations qui implantent les transitions entre ces niveaux. Les deux niveaux supérieurs, les niveaux application et collaboration, ainsi que la transition entre ces niveaux, sont basés sur l'ontologie GCO et sur les inférences associées. Le niveau intergiciel est basé sur le modèle producteur/consommateur du paradigme *Event-Based Computing*. La transition du niveau collaboration vers le niveau intergiciel est basée sur la transformation de graphes avec des grammaires de graphes. Grâce au rôle charnière joué par GCO, Facus implante une interopérabilité sémantique entre le niveau application, qui considère les exigences de l'application et des utilisateurs, et le niveau intergiciel, qui tient compte des contraintes imposées par les ressources disponibles. Cela est spécialement bénéfique pour les environnements ubiquitaires, où l'optimisation des ressources est nécessaire.

L'implantation de Facus prend en compte les modèles proposés pour mettre en œuvre un déploiement de composants automatique qui respecte les exigences et les contraintes. Les applications collaboratives ubiquitaires peuvent donc utiliser Facus pour bénéficier de ce déploiement qui s'adapte au contexte.

Finalement, dans le Chapitre 5, nous avons évalué Facus et l'approche sur laquelle il repose. Pour cela, nous avons introduit une application ubiquitaire de type ludique qui utilise Facus pour mettre en œuvre la collaboration. Nous avons montré comment cette application, développée dans

le cadre du projet UseNet, utilise Facus. Pour cela, elle fournit une ontologie spécifique qui étend GCO, et elle utilise l'API de Facus. Cela nous a servi à évaluer les fonctionnalités de Facus. Nous avons vérifié que les instances du modèle de chaque niveau sont celles attendues dans toutes les phases d'un scénario de déroulement que nous avons proposé. Quant à l'évaluation des performances, elle a montré que le temps d'exécution du processus d'adaptation dépend de la taille du groupe qui collabore. Le temps d'exécution de la partie ontologie est resté en dessous de 1 seconde, grâce aux principes suivis dans la conception de GCO et à l'utilisation de techniques d'optimisation dans le traitement des ontologies. La partie qui pénalise le plus le temps d'exécution est le raffinement entre les niveaux collaboration et intergiciel, avec les transformations de graphes. En revanche, la sélection sensible au contexte avec des politiques au niveau intergiciel est très performante. En effet, elle permet de traiter des centaines de graphes en quelques millisecondes.

Perspectives

Le travail que nous avons présenté dans ce mémoire a permis, d'un côté, d'aborder la modélisation des systèmes collaboratifs ubiquitaires, et d'un autre côté, de proposer une solution pour la mise en place de sessions collaboratives adaptatives dans ce type de systèmes. Ce travail pourrait être amélioré de plusieurs façons, et il ouvre plusieurs perspectives de recherche, que nous listons par la suite. Nous avons classé ces perspectives selon trois catégories : améliorations dans la réalisation, ajout de nouvelles fonctionnalités et généralisation de l'approche.

Réalisation

Amélioration des performances de Facus Comme nous l'avons montré dans le Chapitre 5, quand la taille du groupe augmente, l'exécution du moteur de transformation de graphes pénalise le temps total d'exécution du processus d'adaptation. Ceci est dû au fait que l'application récursive des règles de transformation sur les graphes produit une multitude de graphes intermédiaires. Parmi ces graphes, beaucoup sont équivalents du point de vue du déploiement. Une première idée pour réduire le temps d'exécution est d'implanter une méthode de comparaison de graphes, par exemple à partir d'une fonction de hachage, et de l'utiliser afin d'éliminer les graphes équivalents. Cela réduirait le nombre de graphes à gérer dans chaque itération et donc le temps de traitement serait moindre.

Une deuxième possibilité, plus complexe, serait de générer toutes les combinaisons possibles (jusqu'à une certaine taille de graphe) hors ligne et de les stocker. Ceci permettrait, lors de l'exécution de Facus, de chercher le graphe fourni en entrée parmi l'ensemble de graphes stockés. Une fois le graphe trouvé, on obtiendrait automatiquement l'ensemble pré-calculé de graphes qui le raffinent au lieu de le calculer en ligne. Cette méthode substitue le problème de génération de graphes par un problème de recherche de graphe, moins coûteux. Ainsi, cela éviterait l'explosion combinatoire que provoque l'application récursive des règles de transformation de graphes, et donc le temps d'exécution serait fortement réduit.

Amélioration du service de déploiement et des composants Dans l'implantation de Facus que nous avons réalisée, nous avons privilégié l'aspect modèle et l'obtention des données de contexte ; en revanche, l'implantation du service de déploiement est élémentaire. En effet, l'implantation actuelle de ce service se limite à un protocole de communication simple et aux interfaces du serveur de déploiement et de l'agent de déploiement, ainsi qu'aux squelettes des composants OSGi. Cette implantation va être perfectionnée dans le cadre de la thèse de H. Arous, en cours au groupe OLC du LAAS-CNRS. Les travaux vont porter principalement sur l'implantation des producteurs, des consommateurs et des gestionnaires de canal. L'envoi de données dans ces composants va être implanté en un premier temps avec des sockets TCP, pour essayer plus tard une deuxième implantation basée sur *Data Distribution Service*[8] (DDS). DDS est une spécification de l'*Object Management Group* (OMG) pour des intergiciels de type EBC avancés qui permet l'envoi de données et d'événements tant en mode synchrone qu'en mode asynchrone. Les caractéristiques les plus intéressantes de DDS sont la possibilité de spécifier des paramètres de qualité de service à respecter dans les envois de données et la découverte automatique de services. Il existe plusieurs implantations de DDS en Java, C++ et Ada, notamment.

Une deuxième amélioration consisterait à utiliser un langage de description d'architectures (*Architecture Description Language* ou ADL) pour l'expression du descripteur de déploiement. En effet, dans notre implantation, nous avons utilisé une description *ad-hoc* basée sur GraphML, mais l'utilisation d'un ADL constituerait une solution plus standard qui permettrait l'utilisation des descripteurs générés par Facus dans d'autres systèmes. Par exemple, le langage *Architecture Analysis and Desing Language*[9] (AADL), très utilisé actuellement, permettrait une description très détaillée des schémas de déploiement.

Implantation décentralisée de Facus Dans le Chapitre 4, nous avons dit que Facus centralise la logique de gestion de l'architecture de collaboration. Dans l'exemple présenté dans le Chapitre 5, Facus est placé dans un serveur central auquel se connectent les dispositifs des utilisateurs qui collaborent. Même si la collaboration se déroule d'une façon décentralisée entre les membres du groupe, la gestion de cette collaboration se fait en mode centralisé, ce qui résulte en une configuration hybride.

Il serait très intéressant de faire évoluer cette architecture vers une version décentralisée où la logique de gestion est répartie. Le dispositif de chaque participant exécuterait une partie des traitements sur une partie des données. Les algorithmes répartis en général, et plus particulièrement les algorithmes pair-à-pair, permettent ce genre d'implantations. Cela améliorerait, en premier lieu, le passage à l'échelle de la solution, car l'arrivée de nouveaux participants impliquerait aussi l'augmentation de la capacité totale de calcul. En deuxième lieu, cela augmenterait la robustesse du système, car il n'y aurait pas un seul point central qui, en cas de panne, rendrait le système inutilisable. Finalement, cela effacerait

[8] http://portals.omg.org/dds/
[9] http://www.aadl.info/

le besoin d'avoir une infrastructure préalable dans l'environnement où les utilisateurs collaborent ; ils pourraient collaborer de façon *ad-hoc*.

Fonctionnalités

Utilisation d'autres modèles au niveau intergiciel Dans Facus, nous avons retenu le paradigme EBC pour le niveau intergiciel. EBC fournit un modèle de communication simple et performant qui permet la mise en place de communications synchrones et asynchrones. Cependant, comme nous l'avons mentionné dans le Chapitre 4, d'autres modèles et d'autres implantations sont possibles. Il serait très intéressant de prendre en compte plusieurs modèles différents au niveau intergiciel (par exemple basés sur CORBA, le pair-à-pair, etc.). Cela permettrait de choisir, parmi la gamme de modèles disponibles, le(s) plus adapté(s) à l'application (lors de sa conception) ou même au contexte (dynamiquement lors de l'exécution du système). Cela apporterait une grande flexibilité à Facus et lui permettrait d'atteindre un degré d'adaptabilité plus important.

Prise en compte explicite des communications asynchrones Le modèle GCO représente les sessions collaboratives comme des flux échangés entre les membres d'un groupe. Bien que cette modélisation soit valable tant pour les communications synchrones que pour les communications asynchrones, il serait souhaitable de fournir des éléments spécifiques pour les communications asynchrones, comme le courrier électronique, les forums, l'édition collaborative de documents (notamment les systèmes de type *wiki*), etc. La façon de collaborer avec ce type de communications, très fréquentes dans l'internet, n'est pas la même qu'avec des interactions synchrones. Ils serait donc intéressant de spécialiser GCO pour modéliser explicitement ce type de moyens de communication, par exemple en partant d'un concept `AsyncFlow`, sous-concept de `Flow`.

Dans l'implantation de Facus, nous avons privilégié le mode de communication synchrone avec des flux de données, mais il serait facile de prendre en compte le cas asynchrone, car les intergiciels EBC sont un moyen très adapté à ce type de communications. En effet, la décorrélation entre les producteurs et les consommateurs de données introduite par les gestionnaires de canal permet d'implanter d'une façon naturelle les communications asynchrones.

Généralisation de l'approche

Utilisation de GCO et de Facus dans d'autres applications Dans le Chapitre 5 nous avons présenté un exemple d'application qui utilise GCO et Facus pour mettre en œuvre des sessions dans des environnements ubiquitaires. Il serait intéressant d'utiliser Facus dans d'autres applications pour étudier son fonctionnement et son applicabilité. Il est facile d'imaginer une multitude d'applications intéressantes utilisant la collaboration dans les environnements ubiquitaires : campus intelligents, collaboration entre médecins dans des hôpitaux intelligents, domotique, etc. Ici, nous décrivons deux projets dans lesquels le LAAS-CNRS est impliqué et qui utilisent nos contributions.

En premier lieu, dans le cadre du projet ROSACE[10], Bouassida et al. [BLD10] ont proposé un système de réponse à des situations de crise qui utilise GCO comme modèle collaboratif. Ce type de systèmes soutiennent la collaboration des policiers, des pompiers et des services de secours lors de situations critiques telles que des tremblements de terre, des attentats, des incendies, etc. Ces auteurs proposent une ontologie spécifique qui étend GCO en ajoutant des attributs de qualité de service aux composants et aux dispositifs. Une deuxième ontologie modélise les spécificités de la collaboration dans le domaine de la gestion de crise.

En deuxième lieu, Facus va être utilisé dans le cadre du projet GALAXY[11]. Ce projet s'intéresse à l'édition collaborative de modèles pour l'ingénierie de systèmes complexes. Le modèle GCO va être étendu pour la prise en compte des collaborations, majoritairement asynchrones, dans ce domaine. Même si dans ce projet on ne considère pas des environnements ubiquitaires, Facus reste utile pour implanter des sessions de travail autour des tâches d'ingénierie de modèles. Par exemple, les ingénieurs peuvent travailler sur un modèle du système, en envoyant leurs modifications de façon asynchrone. Ensuite, ils organisent une réunion virtuelle où ils communiquent de façon synchrone pour intégrer les contributions et pour se mettre d'accord sur la suite des travaux.

Plus généralement, il serait pertinent d'étudier l'applicabilité de l'approche d'adaptation architecturale multi-niveaux que nous avons proposée à d'autres applications ubiquitaires qui ne sont pas forcément collaboratives. Pour cela, il faudrait définir les niveaux spécifiques pour ces applications, ainsi que les modèles et les transformations nécessaires pour ces niveaux.

Sessions spontanées Nous avons évoqué dans le Chapitre 2 le concept de *sessions implicites* qui permet de mettre en œuvre des sessions automatiquement à partir du contexte des utilisateurs. Notre contribution constitue un pas dans cette direction, mais nous sommes encore loin d'une véritable création spontanée de sessions. Pour cela, il faudrait rajouter une couche par dessus la couche application qui serait capable d'interpréter le contexte externe et les préférences des utilisateurs qui se trouvent dans l'environnement ubiquitaire afin de leur proposer une application à utiliser, parmi un ensemble d'applications existantes basées sur Facus. Dans cette tâche d'interprétation du contexte, les technologies sémantiques comme les ontologies peuvent s'avérer à nouveau très utiles.

Ce point, délicat et complexe, constitue l'extension la plus ouverte de notre travail, extension qui pourrait déboucher sur d'autres travaux de thèse succédant à notre approche.

[10] http://www.irit.fr/Rosace,737
[11] http://www.irit.fr/GALAXY

Liste des publications personnelles

Revues Scientifiques

[SVT10] G. Sancho, T. Villemur et S. Tazi : An ontology-driven approach for collaborative ubiquitous systems. *International Journal of Autonomic Computing*, 1(3):263–279, 2010.

Conférences Internationales

[BSV+09] I. Bouassida, G. Sancho, T. Villemur, S. Tazi et K. Drira : A model-driven adaptive approach for collaborative ubiquitous systems. *In AUPC 09 : Proceedings of the 3rd workshop on Agent-oriented software engineering challenges for ubiquitous and pervasive computing*, pages 15–20, London, United Kingdom, juillet 2009. ACM.

[PSW+10] A. C. Prudencio, M. L. Sheibel, R. Willrich, S. Tazi et G. Sancho : Application and Network QoS Mapping Using an Ontology-Based Approach. *International Conference on Communication Theory, Reliability, and Quality of Service*, 0:214–219, juin 2010.

[SBT+09] G. Sancho, I. Bouassida, T.Villemur, S. Tazi et K. Drira : A model-driven adaptive framework for collaborative ubiquitous systems. *In Proceedings of the 9th Annual International Conference on New Technologies of Distributed Systems, NOTERE 2009*, pages 233–244, Montréal, Canada, juillet 2009.

[SBVT10] G. Sancho, I. Bouassida, T. Villemur et S. Tazi : What about collaboration in ubiquitous environments ? *In NOTERE 2010, Annual International Conference on New Technologies of Distributed Systems, Touzeur, Tunisia, May 31 - June 2, 2010, Proceedings*, pages 143–150. IEEE, juin 2010.

[STV08] G. Sancho, S. Tazi et T. Villemur : A Semantic-driven Auto-adaptive Architecture for Collaborative Ubiquitous Systems. *In 5th International Conference on Soft Computing as Transdisciplinary Science and Technology (CSTST'2008)*, pages 650–655, Cergy Pontoise (France), octobre 2008.

[STV10] G. Sancho, S. Tazi et T. Villemur : GCO : a Generic Collaboration Ontology. *In Proceedings of the Fourth International Conference on Advances in Semantic Processing (SEMPARO 2010), Florence, Italy*, octobre 2010.

Conférences Nationales

[San08] G. Sancho : Déploiement dynamique de composants coopératifs pour la reconfiguration architecturale. *Ecole d'été RESCOM 2008, Saint-Jean-Cap-Ferrat (France)*, juin 2008. 2p.

[San09] G. Sancho : Modélisation multi-niveau pour des systèmes ubiquitaires collaboratifs. *EDSYS 2009. 10ème Congrès de Doctorants, Toulouse (France)*, mars 2009. 6p.

Rapports de contrat

[LKH+08] J. Latvakoski, J. Kallio, T. Hautakoski, T. Vaisanen, T. P. J. Toivonen, A. Peterzens, A. Deckers, P. Dobbelaere, P. Dedecker, J. Hoebeke, B. Meekers, F. Martinez, L. Manero, I. Lucas, D. Pirez, A. Zabala, T. V. K. Drira, S. Tazi, V. Baudin, T. Chaari, G. Sancho, I. Bouassida et P. Rossignol : M2M Requirements. Délivrable UseNet D1.1, 129p., juin 2008.

[RSM+08] P. Rossignol, C. Sauvignac, P. Meurice, J. M. Tran, P. Maseres, B. Tesniere, S. Ramos, V. Baudin, I. Bouassida, T. Chaari, K. Drira, G. Sancho, S. Tazi, T. Villemur, D. Pirez et Y. Lopez : Uses Cases Components. Délivrable UseNet D5.1, 36p., 2008.

[RSM+09] P. Rossignol, C. Sauvignac, P. Meurice, J. M. Tran, P. Maseres, B. Tesniere, V. Baudin, I. Bouassida, K. Drira, G. Sancho, S. Tazi, T. Villemur, D. Pirez et Y. Lopez : Specification of smart usage demonstrator. Délivrable UseNet D5.2, 41p., 2009.

[RSM+10] P. Rossignol, C. Sauvignac, P. Meurice, J. M. Tran, V. Baudin, I. Bouassida, K. Drira, G. Sancho, S. Tazi, T. Villemur, D. Pirez et Y. Lopez : Full framework specification of M2M service network. Délivrable UseNet D1.4, 126p., 2010.

Bibliographie

[ADB+99] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith et P. 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.

[AGSGHPR07] M. Anzures-García, L. A. Sánchez-Gálvez, M. J. Hornos et P. Paderewski-Rodríguez : Ontology-Based Modelling of Session Management Policies for Groupware Applications. *In* R. Moreno-Díaz, F. Pichler et A. Quesada-Arencibia, éditeurs : *EUROCAST*, volume 4739 de *Lecture Notes in Computer Science*, pages 57–64. Springer, 2007.

[BBC97] P. J. Brown, J. D. Bovey et X. Chen : Context-aware Applications : from the Laboratory to the Marketplace. *IEEE Personal Communications*, 4(5):58–64, October 1997.

[BCM+] D. F. Baader, D. Calvanese, D. L. McGuinness, P. Patel-Schneider et D. Nardi : *The Description Logic Handbook. Theory, Implementation, and Applications*.

[BEH+01] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt et M. S. Marshall : GraphML Progress Report. *In Graph Drawing*, pages 501–512, 2001.

[BGD+08] I. Bouassida, K. Guennoun, K. Drira, C. Chassot et M. Jmaiel : Implementing a rule-driven approach for architectural self configuration in collaborative activities using a graph rewriting formalism. *In 5th International Conference on Soft Computing as Tarnsdisciplinary Science and Technology (CSTST'2008)*, pages 484–491, Cergy Pontoise (France), 2008.

[BL04] R. J. Brachman et H. J. Levesque : *Knowledge Representation and Reasoning*. Computing. Morgan Kaufmann, 2004.

[BLD10] I. Bouassida, J. Lacouture et K. Drira : Semantic Driven Self-Adaptation of Communications Applied to ERCMS. *In The 24th IEEE International Conference on Advanced Information Networking and Applications*, Perth (Australia), avril 2010.

[BLHL01] T. Berners-Lee, J. Hendler et O. Lassila : The Semantic Web. *Scientific American*, mai 2001.

[BOPV10] J. G. Breslin, D. O'Sullivan, A. Passant et L. Vasiliu : Semantic Web computing in industry. *Computers in Industry*, 61(8):729–741, 2010.

[Cha07] T. Chaari : *Adaptation d'applications pervasives dans des environnements multi-contextes*. Thèse de doctorat, INSA de Lyon, septembre 2007.

[Che04] H. Chen : *An Intelligent Broker Architecture for Pervasive Context-Aware Systems*. Thèse de doctorat, University of Maryland, Baltimore County, décembre 2004.

[Cho56] N. Chomsky : Three models for the description of language. *Information Theory, IEEE Transactions on*, 2(3):113–124, 1956.

[CK00] G. Chen et D. Kotz : A survey of context-aware mobile computing research. Rapport technique, Dept. of Computer Science, Dartmouth College, 2000.

[cor06] CORBA Component Model 4.0 Specification. Specification Version 4.0, Object Management Group, April 2006.

[CS99] P. H. Carstensen et K. Schmidt : Computer supported cooperative work : New challenges to systems design. *In K. Itoh (Ed.), Handbook of Human Factors*, pages 619–636, 1999.

[DAS01] A. K. Dey, G. D. Abowd et D. Salber : A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, 2001.

[DB10] K. Drira et I. Bouassida : A Demonstration of an Efficient Tool for Graph Matching and Tarnsformation. *In 10ème Conférence Internationale Francophone sur l'Extraction et la Gestion des Connaissances (EGC 2010)*, pages 71–73, Hammamet (Tunisie), 2010.

[DGLA99] H.-P. Dommel et J. Garcia-Luna-Aceves : Group coordination support for synchronous internet collaboration. *Internet Computing, IEEE*, 3(2):74 –80, mar. 1999.

[Edw94] W. K. Edwards : Session management for collaborative applications. *In CSCW '94 : Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 323–330, New York, NY, USA, 1994. ACM.

[Edw05] W. K. Edwards : Putting computing in context : An infrastructure to support extensible context-enhanced collaborative applications. *ACM Trans. Comput.-Hum. Interact.*, 12(4):446–474, 2005.

[EGR91] C. A. Ellis, S. J. Gibbs et G. Rein : Groupware : some issues and experiences. *Commun. ACM*, 34(1):39–58, 1991.

[ESB07] D. Ejigu, M. Scuturici et L. Brunie : CoCA : A Collaborative Context-Aware Service Platform for Pervasive Computing. *In Information Technology, 2007. ITNG '07. Fourth International Conference on*, pages 297–302, avril 2007.

[FNBS07] Y. Fawaz, A. Negash, L. Brunie et V.-M. Scuturici : ConAMi : Collaboration Based Content Adaptation Middleware for Pervasive Computing Environment. *In Pervasive Services, IEEE International Conference on*, pages 189–192, juillet 2007.

[GCH+04] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl et P. Steenkiste : Rainbow : architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10): 46–54, octobre 2004.

[GHJV95] E. Gamma, R. Helm, R. E. Johnson et J. Vlissides : *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[GLT08] I. Gorton, Y. Liu et N. Trivedi : An extensible and lightweight architecture for adaptive server applications. *Softw. Pract. Exper.*, 38(8):853–883, 2008.

[GPZ04] T. Gu, H. K. Pung et D. Q. Zhang : A Middleware for Building Context-Aware Mobile Services. *In In Proceedings of IEEE Vehicular Technology Conference (VTC)*, 2004.

[Gru93] T. R. Gruber : A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[Gue06] K. Guennoun : *Architectures dynamiques dans le contexte des applications à base de composants et orientées service*. Thèse de doctorat, Université Paul Sabatier, Toulouse, 183p., 2006.

[GXW+05] N. Gu, J. Xu, X. Wu, J. Yang et W. Ye : Ontology based semantic conflicts resolution in collaborative editing of design documents. *Advanced Engineering Informatics*, 19(2):103–111, 2005.

[Ham07] E. Hammami : *Déploiement sensible au contexte et reconfiguration des applications dans les sessions collaboratives*. Thèse de doctorat, Université Paul Sabatier, Toulouse, 121p., 2007.

[HJ09] S. Hosain et H. Jamil : Empowering OWL with overriding inheritance, conflict resolution and non-monotonic reasoning. *In Proceedings id the AAAI 2009 Spring Symposium on Social Semantic Web : Where Web 2.0 meets Web 3.0, Stanford, California*, pages 53–58, mars 2009.

[HL09] K. Hamadache et L. Lancieri : Role-based collaboration extended to pervasive computing. *In Intelligent Networking and Collaborative Systems, 2009. INCOS '09. International Conference on*, pages 9–15, Nov. 2009.

[Hor51] A. Horn : On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[HPSB+04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof et M. Dean : SWRL : A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission 21 May 2004, 2004. Url : http://www.w3.org/Submission/SWRL/, accédé le 23/07/2010.

[Ign91] J. P. Ignizio : *An Introduction To Expert Systems*. Mcgraw-Hill College, Har/Dsk édition, janvier 1991.

[IIS+08] R. Iqbal, K. Iqbal, N. Shah, J. Siddiqui et A. James : Development of context-aware systems to support human-human communication and collaboration. *In ITNG '08 : Proceedings of the Fifth International Conference on Information Technology : New Generations*, pages 206–211, Washington, DC, USA, 2008. IEEE Computer Society.

[Kam10] A. Kamoun : Application de jeu de bataille navale dans un environnement d'informatique diffuse. Rapport de projet de fin d'études, École Nationale des Sciences de l'Informatique. Université de la Manouba, Tunisie, septembre 2010.

[Kar94] A. Karsenty : Le collecticiel : de l'interaction homme-machine à la communication homme-homme. *Technique et Science Informatique (TSI)*, 13(1):105–127, 1994.

[KC03] J. O. Kephart et D. M. Chess : The Vision of Autonomic Computing. *Computer*, 36:41–50, 2003.

[KFNM04] H. Knublauch, R. W. Fergerson, N. F. Noy et M. A. Musen : The Protégé OWL plugin : An open development environment for semantic web applications. pages 229–243. Springer, 2004.

[KK88] K. L. Kraemer et J. L. King : Computer-based systems for cooperative work and group decision making. *ACM Comput. Surv.*, 20(2):115–146, 1988.

[Lad97] R. Laddaga : Self-adaptive software. Rapport technique, DARPA BAA, 1997.

[Lad01] R. Laddaga : Active Software. *In* P. Robertson, H. Shrobe et R. Laddaga, éditeurs : *Self-Adaptive Software*, volume 1936 de *Lecture Notes in Computer Science*, pages 11–26. Springer Berlin / Heidelberg, 2001.

[Las05] O. Lassila : Applying Semantic Web in Mobile and Ubiquitous Computing : Will Policy-Awareness Help ? *In proceedings of the Semantic Web and Policy Workshop, 4th International Semantic Web Conference*, 2005.

[LB87] H. J. Levesque et R. J. Brachman : Expressiveness and tractability in knowledge representation and reasoning. *Computational intelligence*, 3(2):78–93, 1987.

[LV07] M. P. Locatelli et G. Vizzari : Awareness in collaborative ubiquitous environments : The multilayered multi-agent situated system approach. *ACM Trans. Auton. Adapt. Syst.*, 2(4), 2007.

[Maz95] B. Mazlish : The man-machine and artificial intelligence. *Stanford Humanities Review*, 4(2):21–45, 1995.

[MC02] R. Meier et V. Cahill : Taxonomy of distributed event-based programming systems. *In ICDCSW '02 : Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 585–588, Washington, DC, USA, 2002. IEEE Computer Society.

[Min75] M. Minsky : A Framework for Representing Knowledge. *In* P. Winston, éditeur : *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, New York, 1975.

[MSKC04] P. K. McKinley, S. M. Sadjadi, E. P. Kasten et B. H. C. Cheng : Composing Adaptive Software. *Computer*, 37(7): 56–64, 2004.

[Nel81] B. J. Nelson : *Remote procedure call*. Thèse de doctorat, Pittsburgh, PA, USA, 1981.

[OGT+99] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum et A. Wolf : An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54–62, mai 1999.

[OSG07] OSGi Alliance. *OSGi Service Platform Release 4*, May 2007.

[Pas98] J. Pascoe : Adding generic contextual capabilities to wearable computers. *In Wearable Computers, 1998. Digest of Papers. Second International Symposium on*, pages 92–99, octobre 1998.

[PLZ08] A. Padovitz, S. Loke et A. Zaslavsky : Multiple-agent perspectives in reasoning about situations for context-aware pervasive computing systems. *Systems, Man and Cybernetics, Part A : Systems and Humans, IEEE Transactions on*, 38(4):729–742, juillet 2008.

[PSG+05] B. Parsia, E. Sirin, B. C. Grau, E. Ruckhaus et D. Hewlett : Cautiously approaching SWRL. Url : http://www.mindswap.org/papers/CautiousSWRL.pdf, accédé le 23/07/2010, 2005.

[Qui67] M. R. Quillian : Word concepts : a theory and simulation of some basic semantic capabilities. *Behavioral Science*, 12(5):410–430, September 1967.

[Rai08] C. Raibulet : Facets of Adaptivity. *In* R. Morrison, D. Balasubramaniam et K. Falkner, éditeurs : *Software Architecture*, volume 5292 de *Lecture Notes in Computer Science*, pages 342–345. Springer Berlin / Heidelberg, 2008.

[RKT+95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch et P. Muth : Towards a cooperative transaction model - the cooperative activity model. *In VLDB '95 : Proceedings of the 21th International Conference on Very Large Data Bases*, pages 194–205, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[RLBP08] V. Rajsiri, J.-P. Lorré, F. Bénaben et H. Pingaud : Collaborative Process Definition Using An Ontology-Based Approach. *In* L. Camarinha-Matos et W. Picard, éditeurs : *Pervasive Collaborative Networks*, volume 283 de *IFIP International Federation for Information Processing*, pages 205–212. Springer Boston, 2008.

[RN03] S. J. Russell et P. Norvig : *Artificial Intelligence : A Modern Approach*. Pearson Education, 2003.

[RPM98] N. S. Ryan, J. Pascoe et D. R. Morse : Enhanced Reality Fieldwork : the Context-aware Archaeological Assistant. *In* V. Gaffney, M. van Leusen et S. Exxon, éditeurs : *Computer Applications in Archaeology 1997*, British Archaeological Reports, Oxford, October 1998. Tempus Reparatum.

[RPVD01] L. Rodriguez Peralta, T. Villemur et K. Drira : An XML on-line session model based on graphs for synchronous cooperative groups. *In 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 1257–1263, Las Vegas (USA), 2001.

[Sat01] M. Satyanarayanan : Pervasive computing : Vision and challenges. *IEEE Personal Communications*, pages 10–17, août 2001.

[Sat04] M. Satyanarayanan : The many faces of adaptation. *IEEE Pervasive Computing*, 3:4–5, 2004.

[SAW94] B. Schilit, N. Adams et R. Want : Context-aware computing applications. *In Mobile Computing Systems and Applications, WMCSA 1994. First Workshop on*, pages 85–90, décembre 1994.

[SBT05] S. Suwanmanee, D. Benslimane et P. Thiran : OWL-based approach for semantic interoperability. *In Advanced Information Networking and Applications, AINA 2005. 19th International Conference on*, volume 1, pages 145–150, mars 2005.

[Sch01] R. Schollmeier : A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. *In P2P '01 : Proceedings of the First International Conference on Peer-to-Peer Computing*, page 101, Washington, DC, USA, 2001. IEEE Computer Society.

[Sha99] M. Shanahan : The event calculus explained. *Artificial intelligence today : recent trends and developments*, pages 409–430, 1999.

[SM03] D. Saha et A. Mukherjee : Pervasive Computing : A Paradigm for the 21st Century. *Computer*, 36:25–31, 2003.

[SSS91] M. Schmidt-Schaubss et G. Smolka : Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26, 1991.

[ST94] B. Schilit et M. Theimer : Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5):22–32, 1994.

[ST09] M. Salehie et L. Tahvildari : Self-adaptive software : Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[SWM04] M. K. Smith, C. Welty et D. L. McGuinness : OWL Web Ontology Language Guide. W3C Recommendation, février 2004. Url : http://www.w3.org/TR/owl-guide/, accédé le 21/07/2010.

[TE01] S. Tazi et F. Evrard : Intentional structures of documents. *In HYPERTEXT '01 : Proceedings of the 12th ACM conference on Hypertext and Hypermedia*, pages 39–40, New York, NY, USA, 2001. ACM.

[TP03] G. Texier et N. Plouzeau : Automatic Management of Sessions in Shared Spaces. *The Journal of Supercomputing*, 24(2):173–181, 2003.

[Vil06] T. Villemur : *Modèles et services logiciels pour le travail collaboratif*. Habilitation à diriger des recherches, Université Paul Sabatier, Toulouse, France, septembre 2006.

[Wei91] M. Weiser : The Computer for the 21st Century. *Scientific American*, 265(3):94–104, septembre 1991.

[Wei93] M. Weiser : Hot topics-ubiquitous computing. *Computer*, 26(10):71–72, oct. 1993.

[Wei94] M. Weiser : The world is not a desktop. *Interactions*, 1(1): 7–8, 1994.

[Wei99] M. Weiser : Some computer science issues in ubiquitous computing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3): 12, 1999.

[Win01] T. Winograd : Architectures for context. *Human-Computer Interaction*, 16:401–419, 2001.

Ce document a été préparé à l'aide de l'éditeur de texte *vim* et du logiciel de composition typographique LaTeX 2_ε.

Titre Adaptation d'architectures logicielles collaboratives dans les environnements ubiquitaires. Contribution à l'interopérabilité par la sémantique.

Résumé Ce travail de thèse explore la conception d'applications collaboratives destinées a être exécutées dans des environnements ubiquitaires. Les applications collaboratives fournissent des moyens pour que des utilisateurs organisés dans des groupes puissent coopérer afin d'effectuer une tâche commune. Jusqu'à présent, ces applications ont été conçues pour une utilisation avec des systèmes de type *bureau*. Dans ce travail nous proposons de faire profiter ces applications des possibilités offertes par les environnements issus de l'informatique ubiquitaire, notamment la sensibilité au contexte et l'adaptation. En premier lieu, un modèle sémantique de la collaboration, appelé GCO, est proposé afin de permettre l'expression des besoins de collaboration indépendamment du domaine d'application. Ce modèle est une ontologie de la collaboration incorporant des règles permettant d'effectuer d'inférences pour déduire un schéma de déploiement à partir des besoins de collaboration. En deuxième lieu, une approche de modélisation est proposée afin de faciliter la conception d'applications collaboratives ubiquitaires. Cette approche se base sur une décomposition multi-niveaux permettant la mise en œuvre d'une adaptation de l'architecture du système aux changements du contexte haut niveau (exigences de l'application) et bas niveau (contraintes liées aux ressources). Ensuite, le *framework* FACUS, qui est une implantation de l'approche utilisant GCO comme modèle pivot pour l'interopérabilité entre les exigences et les contraintes, est proposé. Enfin, FACUS est évalué tant du point de vue fonctionnel que de celui des performances, montrant la faisabilité de notre approche.

Mots-clés Adaptation d'architectures logicielles, modèles sémantiques, sensibilité au contexte, applications collaboratives, environnements ubiquitaires

Title Adapatation of collaborative software architetures in ubiquitous environments. Contribution to semantic interoperability.

Abstract This work explores the design of collaborative applications for ubiquitous environments. Collaborative applications support the cooperation of groups of users that want to achieve a common goal. Until now, collaborative applications have been designed for *desktop* systems. This work proposes the use collaborative applications in ubiquitous computing environments, which offer new characteristics such as context awareness and adaptation. Firstly, a semantic model of collaboration, called GCO (Generic Collaboration Ontology), is proposed. This model allows expressing collaboration needs independently of the application domain. Secondly, a modeling approach is proposed for the design of collaborative ubiquitous applications. This approach is based on a multi-layer decomposition that allows implementing the adaptation of the system's architecture to changes in the context. These changes can be high-level (application requirements) and/or low-level (resource constraints). Then, the FACUS framework, that implements the approach using GCO as pivot model for the interoperation of requirements and constraints, is proposed. Finally, FACUS functionalities and performances are evaluated, thus showing the feasibility of the approach.

Keywords Architectural adaptation, semantic models, context awareness, collaborative applications, ubiquitous environments