Approche déclarative pour la génération de canevas logiciels dédiés à l'informatique ubiquitaire

Wilfried Jouve

▶ **To cite this version:**

Wilfried Jouve. Approche déclarative pour la génération de canevas logiciels dédiés à l'informatique ubiquitaire. Génie logiciel [cs.SE]. Université Sciences et Technologies - Bordeaux I, 2009. Français. NNT : . tel-00402605

HAL Id: tel-00402605
https://theses.hal.science/tel-00402605

Submitted on 7 Jul 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre: 3781

THÈSE

présentée à

L'UNIVERSITÉ BORDEAUX 1

École Doctorale de Mathématiques et Informatique

par **Wilfried Jouve**

pour obtenir le grade de

DOCTEUR
Spécialité: INFORMATIQUE

Approche déclarative pour la génération de canevas logiciels dédiés à l'informatique ubiquitaire

Thèse dirigée par Charles Consel

Soutenue le : 8 avril 2009

Devant la commission d'examen formée de :

M. :	Raymond	Namyst	Professeur, Université de Bordeaux 1, Bordeaux	Président
MM. :	Yolande	Berbers	Professeur, Katholieke Universiteit Leuven, Belgique	Rapporteurs
	Didier	Donsez	Professeur, Université Joseph Fourier, Grenoble	
MM. :	Roy H	Campbell	Professeur, University of Illinois, Urbana-Champaign	Examinateurs
	Charles	Consel	Professeur, ENSEIRB, Bordeaux	

Université Bordeaux 1 – Les Sciences et les Technologies au service de l'Homme et de l'environnement

Thèse réalisée au sein de l'Équipe-Projet INRIA Phoenix

INRIA Bordeaux – Sud-Ouest
Bâtiment A 29
351 cours de la libération
F-33405 Talence Cedex

LaBRI
Unité Mixte de Recherche CNRS (UMR 5800)
351 cours de la libération
F-33405 Talence cedex

à ma mère,

Remerciements

Une thèse est constituée de moments de joie et de doute. Mais elle est en premier lieu constructive pour soi. Les remerciements exposent son coté positif car ce sont les seuls souvenirs qui méritent de rester à sa surface.

Je remercie tout d'abord les membres de mon jury :

- Je remercie Yolande Berbers, professeur à la Katholieke Universiteit Leuven, et Didier Donsez, professeur à l'Université Joseph Fourier de Grenoble, d'avoir accepté la charge de rapporteur.
- Je remercie Raymond Namyst, professeur à l'Université de Bordeaux 1, qui a présidé ce jury.
- Je remercie Roy H Campbell, professeur à la University of Illinois at Urbana-Champaign, d'être venu de si loin pour assister à ma soutenance.

Je remercie Charles Consel, mon directeur de thèse, sans qui je n'aurais sans doute pas effectué ma thèse dans de si bonnes conditions.

Je remercie les membres de l'équipe Phoenix. En particulier, Nicolas Palix a été assez courageux pour relire et corriger mes premiers brouillons. Je le remercie aussi pour son aide précieuse dans la concrétisation de l'approche DIAGEN. Julien Bruneau m'a permis de « m'abstraire de l'implémentation » de DIASIM tout en me persuadant que je devais prendre plus de cours de tennis. Je remercie Julien Lancia pour son ouverture d'esprit lors de nos discussions et pour son travail à mes cotés. Je remercie Julien Mercadal pour avoir relu une partie de ma thèse et pour ses remarques toujours pertinentes. Laurent Burgy, « ma petite pleureuse », m'a fait rire et parfois pleurer. Il a surtout toujours été là malgré tout. Je remercie Benjamin Bertran pour s'être fait lamentablement battre à notre dernier baby-foot avec un de mes tirs à 2 points et pour son investissement dans les projets du groupe, même si « ce n'était pas son bouleau ». Je remercie Nicolas Loriant pour ses conseils et Zoé Drey pour la tarte tatin.

Je remercie tous les membres de ma famille. Je remercie en particulier mon oncle, ma tante et ma cousine Amélie pour m'avoir aidé durant mes séjours à Paris et ainsi grandement faciliter la fin de ma thèse. Je remercie mon père et ma sœur pour avoir été là et ma sœur pour avoir relu ma thèse.

Enfin, je dédie cette thèse à ma mère car c'est aussi la sienne.

Approche déclarative pour la génération de canevas logiciels dédiés à l'informatique ubiquitaire

Résumé

Les applications ubiquitaires évoluent dans des environnements plus hétérogènes et plus dynamiques que ceux des systèmes distribués traditionnels. La criticité des domaines applicatifs impliqués et la vocation non intrusive de l'informatique ubiquitaire exigent de garantir la robustesse des applications avant et pendant leur déploiement en situation réelle. Les solutions proposant de gérer la dynamicité des environnements ubiquitaires offrent des canevas de programmation dont la généricité ne permet pas de garantir la fiabilité des applications développées. D'autres solutions permettent davantage de vérifications en assurant, par exemple, l'intégrité des communications. Cependant, ces vérifications, telles que proposées dans ces solutions, empêchent la prise en compte de la dynamicité, indispensable à la mise en œuvre d'applications ubiquitaires.

Dans cette thèse, nous proposons un canevas logiciel visant à concevoir, développer, vérifier et tester les applications ubiquitaires avant leur déploiement en environnements réels. Notre approche repose sur des spécifications haut niveau des applications cibles. Chaque spécification, écrites dans le langage DIASPEC, est analysée, vérifiée et compilée par le compilateur DIASPEC qui génère un canevas logiciel dédié, incluant un canevas de programmation et un canevas de simulation. Les canevas de programmation générés fournissent du support pour la programmation d'applications ubiquitaires. Ils garantissent l'intégrité des communications tout en permettant la gestion de la dynamicité des environnements ubiquitaires. Les canevas de simulation générés fournissent du support pour le test des applications ainsi développées.

Les contributions de cette thèse sont les suivantes :

- Nous proposons l'approche DIAGEN qui permet, à partir de spécifications d'architectures logicielles ubiquitaires, de générer du support de programmation et un ensemble de vérifications dédiés au développement d'applications ubiquitaires. Ces spécifications reposent sur un langage déclaratif, appelé DIASPEC, qui permet de décrire les types de services composant les applications ubiquitaires cibles. Le compilateur DIASPEC vérifie la cohérence des spécifications et génère, à partir de celles-ci, des canevas de programmation dédiés. Les canevas de programmation générés garantissent l'intégrité des communications d'applications dynamiques.
- Nous avons développé DIASIM, un simulateur pour tester à l'exécution, le comportement des applications développées avec l'approche DIAGEN. DIASIM permet de tester les applications sans les modifier et d'intégrer incrémentalement des services réels dans les scénarios de simulation.
- Nous avons montré que l'approche DIAGEN permet davantage de vérifications statiques tout en gérant la dynamicité des environnements ubiquitaires et en permettant une plus grande concision des applications développées. Nous avons analysé les canevas de programmation générés pour la gestion de réunion, l'immotique et la téléphonie.

Mots clés

Canevas de programmation, simulation, systèmes ubiquitaires, génie logiciel

A Declarative Approach for Generating Software Frameworks Dedicated to Ubiquitous Computing

Abstract

Ubiquitous systems have to cope with more heterogeneous and more dynamic environments than traditional distributed systems. The criticity of application domains related to ubiquitous computing requires to ensure the robustness of applications before their deployment in actual situations. Some approaches handle the dynamicity of ubiquitous environments but propose generic programming frameworks. This genericity is a major obstacle for producing robust applications. Other approaches enable more verifications that ensure, for example, communication integrity. However, as performed by these approaches, these verifications are not compatible with managing constant changes of ubiquitous environments.

This thesis proposes a declarative approach to generate software frameworks for ubiquitous computing. Our approach relies on high-level specifications of target applications. These specifications, written in the DIASPEC language, are analyzed, checked and compiled by the DIASPEC compiler which generates dedicated software frameworks, including programming frameworks and simulation frameworks. The generated programming frameworks provide programming support for developing ubiquitous applications while the generated simulation frameworks provide testing support for these applications. Programming frameworks ensure communication integrity while handling the dynamicity of ubiquitous environments. Furthermore, they are generated on top of a generic middleware; the layered architecture of this middleware allows ubiquitous applications to be independent of underlying technologies.

The contributions of this thesis are as follows:

- We present the DIAGEN approach which proposes to generate, from specifications of ubiquitous software architectures, programming support and verifications dedicated to the development of ubiquitous applications. These specifications are written in a declarative language, called DIASPEC, which allows to describe all service types composing the target ubiquitous applications. The DIASPEC compiler checks the specification consistency and generates dedicated programming frameworks. These programming frameworks ensure communication integrity of dynamic applications. The underlying middleware has a layered architecture, making programming frameworks independent of software buses. Thus, applications are portable without modification.

- We present the DIASIM simulator that tests, at run time, the behavior of applications developed with DIAGEN. DIASIM allows to test applications without modification and to incrementally integrate actual services in simulation scenarios.

- We show that DIAGEN applications are more concise than existing approaches while enabling more verifications and handling the dynamicity of ubiquitous environments.

Keywords

Programming Framework, Simulation, Ubiquitous Systems, Software Engineering

Table des matières

Table des figures

Chapitre 1

Introduction

La miniaturisation des objets communicants ainsi que leur banalisation dans nos activités quotidiennes permettent d'envisager de nouveaux domaines applicatifs où l'utilisateur serait au centre des préoccupations. Ces domaines applicatifs sont issus de la convergence d'écosystèmes qui sont encore aujourd'hui relativement disjoints : les télécommunications, l'informatique et l'électronique. Ils automatisent un nombre croissant d'activités, comme la gestion des bâtiments (c'est-à-dire l'*immotique*), des maisons (c'est-à-dire la *domotique*) ou le suivi des patients à l'hôpital. L'automatisation de ces activités requiert des objets communicants plus hétérogènes et plus nombreux par rapport aux systèmes distribués traditionnels. Ces objets communicants sont coordonnés par des applications dîtes ubiquitaires [Wei91] qui automatisent les tâches en s'adaptant aux changements des environnements physiques. Les changements à considérer et les adaptations à effectuer varient selon le domaine applicatif visé. Il peut s'agir d'une simple intrusion dans un bâtiment ou bien d'une situation critique pour une personne âgée. Dans tous les cas, la détection de ces changements est grandement facilitée par la généralisation des capteurs de faibles dimensions et notamment par l'émergence des technologies RFID *(Radio-frequency identification)* permettant le suivi et la localisation des personnes et des objets. Les applications ubiquitaires évoluent donc dans des environnements physiques plus difficiles à appréhender car plus riches et plus dynamiques qu'auparavant. Par ailleurs, les applications ubiquitaires doivent être robustes pour ne pas interférer négativement sur les activités quotidiennes des utilisateurs et ne pas mettre en danger ces mêmes utilisateurs dans des domaines applicatifs critiques, comme le domaine hospitalier [IBM06] ou le domaine de la gestion des autoroutes [CCG+07]. La programmation des applications orchestrant ces objets communicants doit alors prendre en compte une multitude de scénarios, chacun décrivant une évolution possible des environnements physiques. La complexité des environnements physiques nécessite des outils adaptés pour la programmation des applications ubiquitaires et pour l'intégration d'un flot constant de nouveaux objets communicants.

Les approches existantes proposent l'utilisation d'intergiciels pour assurer l'intéropérabilité entre les différents objets communicants [Dow98, OHE97, W3C, RHC+02, GDL+04, Sun00]. Les intergiciels fournissent, au-dessus des environnements physiques et des systèmes d'exploitation, une couche d'abstraction qui est composée (1) d'un canevas de programmation *(programming framework)* pour le développement des applications et (2) de services systèmes pour assurer, à l'exécution, des tâches communes à la plupart des applications distribuées, comme la gestion des communications. Les intergiciels proposés permettent de gérer la dynamicité des environnements physiques grâce à des mécanismes de découverte de services [ZMN05] et des

communications asynchrones fondées sur les événements [EFGmK03]. Cette dynamicité est nécessaire à la mise en œuvre d'applications ubiquitaires devant constamment s'adapter aux changements des environnements physiques. Cependant, la généricité des intergiciels implique que peu de vérifications statiques sont possibles. Les applications produites sont par conséquent peu fiables et ne prennent pas en compte la criticité des nouveaux domaines applicatifs. La détection et le traitement des erreurs sont ainsi effectués à l'exécution, compromettant la sécurité des utilisateurs et la stabilité de l'infrastructure logicielle. Les approches à base de composants constituent un premier pas vers un développement d'applications plus fiables. L'utilisation d'une chaîne de production standardisée et la mise en œuvre de vérifications statiques de cohérence rendent l'exécution des applications plus prévisible [Sie00, ACN02, BCS04]. Cependant, ces vérifications, telles que proposées dans les solutions existantes, empêchent toute considération de la dynamicité, indispensable au développement d'applications ubiquitaires. De plus, à l'instar des approches précédentes, certaines vérifications ne peuvent être effectuées qu'à l'exécution. En effet, la logique applicative étant une donnée subjective, la capacité des applications à accomplir leur tâche ne peut être évaluée que par des tests en environnements réels. La complexité des environnements physiques rend ces tests longs, coûteux et parfois inconcevables. Plus généralement, les canevas de programmation existants n'abstraient que partiellement les technologies sous-jacentes et requièrent donc un haut niveau d'expertise, obligeant les développeurs à manipuler des notions propres aux technologies sous-jacentes. Le portage du code applicatif devient alors une opération délicate. La diversité des domaines applicatifs, potentiellement critiques, ainsi que la complexité des environnements physiques associés nécessitent donc une approche adaptable permettant de vérifier la fiabilité des applications distribués avant leur déploiement, tout en gérant leur dynamicité.

1.1 Thèse

Dans cette thèse, nous proposons l'approche DIAGEN[1] visant à concevoir, développer, vérifier et tester les applications ubiquitaires avant leur déploiement en environnements réels. Cette approche a donné lieu au développement d'une boite à outils couvrant toutes les étapes de production d'une application ubiquitaire.

L'approche DIAGEN s'appuie sur un langage déclaratif, DIASPEC, pour décrire l'architecture logicielle d'une application et spécifier les contraintes en terme de découverte et d'interaction entre les composants de l'application. À partir de cette spécification haut niveau, le compilateur DIASPEC génère un canevas logiciel dédié, incluant un canevas de programmation et un canevas de simulation. Les canevas de programmation générés permettent l'implémentation et l'instanciation de l'architecture logicielle décrite. Ils permettent le développement d'applications distribuées dynamiques tout en garantissant statiquement la validité des applications vis-à-vis de leur spécification. Les applications développées dans l'approche DIAGEN sont testées dans un simulateur d'environnements ubiquitaires, appelé DIASIM. Les scénarios de simulation sont définis grâce au canevas de simulation généré depuis la spécification DIASPEC correspondante. Enfin, elles sont exécutées en environnements réels dans un intergiciel dont l'architecture en couche permet la portabilité des applications à travers une variété de bus logiciels.

[1] **DI**stributed **A**pplications based on a **GEN**erative approach

1.2 Contributions

Les contributions de cette thèse comportent trois volets : un langage de spécification d'architectures logicielles d'applications ubiquitaires, des canevas de programmation dédiés aux architectures logicielles spécifiées et un simulateur permettant de tester les applications développées.

DIASPEC. Nous avons conçu DIASPEC, un langage déclaratif pour la spécification d'architectures logicielles ubiquitaires. Les spécifications DIASPEC déclarent les contraintes que devront respecter les développeurs d'applications ubiquitaires. Ces contraintes s'appliquent aux descriptions, aux fonctionnalités et aux connexions des briques logicielles constituant les applications ubiquitaires. Elles répondent à des besoins de validation des applications et de gestion de la dynamicité des environnements physiques.

Canevas de programmation dédiés. Nous avons développé le compilateur DIASPEC qui vérifie la cohérence d'une spécification DIASPEC et génère, à partir de celle-ci, un canevas de programmation dédié. Les canevas de programmation générés aident au développement des applications distribuées et dynamiques tout en garantissant statiquement le respect des contraintes déclarées dans la spécification DIASPEC. Nous avons conçu un intergiciel générique pour exécuter les applications développées dans le canevas de programmation. Cet intergiciel a une architecture en couche, rendant les canevas de programmation indépendants des technologies sous-jacentes, comme les bus logiciels RMI, SIP ou services Web. Ces technologies deviennent ainsi interchangeables sans impacter le code applicatif.

DIASIM. Nous avons développé DIASIM, un simulateur pour tester le comportement des applications ubiquitaires à l'exécution. Nous nous appuyons sur l'approche DIAGEN pour spécifier l'architecture logicielle de l'environnement simulé et générer le support de programmation correspondant. Notre approche permet de tester des applications dans des environnements hybrides constitués d'entités réels et simulés. DIASIM étend notre intergiciel pour exécuter et visualiser les environnements simulés.

1.3 Organisation du document

Ce document est structuré en deux parties. La première partie introduit le contexte de l'étude. La deuxième partie décrit notre solution fondée sur une approche déclarative.

Contexte. La première partie porte sur l'étude du contexte dans lequel nous nous plaçons. Le chapitre 2 caractérise les systèmes ubiquitaires. Le chapitre 3 introduit la programmation des systèmes distribués. Le chapitre 4 décrit les solutions existantes pour la production d'applications distribuées. Dans le chapitre 5, nous présentons la démarche suivie pour répondre aux problèmes posés par le développement de systèmes ubiquitaires. Cette démarche décrit notre approche, de la problématique à la mise en oeuvre d'une implémentation.

Approche proposée. Dans la deuxième partie, nous présentons la contribution de cette thèse. Le chapitre 6 décrit notre langage de description d'architectures logicielles ubiquitaires. Le chapitre 7 décrit les canevas de programmation générés et l'architecture de l'intergiciel

associé. Le chapitre 8 illustre notre approche avec le développement d'un simulateur. Ce simulateur permet de tester le comportement du code applicatif à l'exécution. Dans le chapitre 9, nous validons notre approche en présentant l'implémentation du compilateur DIASPEC, en analysant le code généré et évaluant sa pertinence et sa concision par rapport aux approches existantes. Enfin, dans le chapitre 10, nous récapitulons les différentes contributions de notre approche et nous ébauchons des perspectives.

Première partie

Contexte

Chapitre 2

Les systèmes ubiquitaires

Les systèmes distribués traditionnels doivent faire face aux problèmes de distribution et d'hétérogénéité d'objets communicants considérés comme immobiles. La miniaturisation des objets communicants, les communications sans fil ainsi que leur banalisation dans nos activités quotidiennes ont transformé les systèmes distribués et permis l'émergence de nouveaux domaines applicatifs comme la domotique ou l'immotique. Les systèmes ubiquitaires matérialisant ces nouveaux domaines applicatifs nécessitent d'orchestrer des objets communicants mobiles, plus hétérogènes qu'auparavant et dont certains ont des ressources limitées. Les systèmes ubiquitaires doivent aussi prendre en compte la dynamicité des paramètres physiques des environnements pour adapter leur comportement et prendre des décisions à la place des utilisateurs. Dans ce chapitre, nous caractérisons les systèmes ubiquitaires pour en comprendre les spécificités et les besoins. Dans un premier temps, nous identifions les défis des systèmes ubiquitaires. Ensuite, nous caractérisons les applications ubiquitaires et les services offerts par les objets communicants.

2.1 Défis

L'hétérogénéité des objets communicants, la dynamicité des environnements physiques et la criticité des domaines applicatifs constituent trois des principaux défis des systèmes ubiquitaires. Il est nécessaire de les caractériser pour pouvoir proposer des solutions adaptées.

2.1.1 Hétérogénéité

Par rapport aux premiers systèmes distribués, les systèmes ubiquitaires sont composés d'objets communicants hétérogènes dont les domaines applicatifs étaient jusqu'à récemment disjoints. Ces objets communicants sont, par exemple, des terminaux téléphoniques, des ordinateurs, des appareils électroménagers, des appareils électroniques grand public ou des appareils médicaux. Ils hébergent une variété de *services* accessibles par des bus matériels et logiciels hétérogènes. Ces services héritent ainsi de l'hétérogénéité de leur objet communicant hôte. Par exemple, un assistant numérique personnel ou PDA *(Personal Digital Assistant)* héberge un service de notifications d'alertes qui peut interagir avec d'autres services via le bus matériel Wifi ou le bus matériel Bluetooth et le bus logiciel Jini [Sun00] ou le bus logiciel RMI [Dow98]. Pour assurer l'interopérabilité entre les objets communicants, leur hétérogénéité doit être abstraite par des outils capables d'intégrer un flot constant de nouveaux objets

communicants.

2.1.2 Dynamicité

Les systèmes ubiquitaires requièrent de considérer des environnements physiques dont l'évolution est imprévisible. Notamment, la mobilité des utilisateurs et la volatilité des objets communicants rendent les systèmes ubiquitaires difficiles à appréhender. Dans cette section, nous introduisons les deux formes de dynamicité dont les systèmes ubiquitaires sont dépendants : la dynamicité des objets communicants et la dynamicité du contexte.

2.1.2.1 Objets communicants

La disponibilité des objets communicantes varie au cours du temps. Ils sont mobiles car connectés sur des réseaux sans fil, ont des ressources limitées, ont des problèmes matériels ou des bogues logiciels les rendant inaccessibles, ou bien sont ajoutées ou supprimées des environnements physiques pour des raisons de maintenance. Les systèmes ubiquitaires doivent donc considérer la *dynamicité* des objets communicants en plus de leur hétérogénéité.

2.1.2.2 Contexte

Les systèmes ubiquitaires permettent l'automatisation de tâches auparavant attribuées aux utilisateurs. À l'instar d'un utilisateur, les systèmes ubiquitaires doivent être sensibles à leur *environnement physique* pour prendre les décisions appropriées. Ils doivent alors non seulement prendre en compte la dynamicité des objets communicants, mais aussi l'évolution des *paramètres physiques* de l'environnement physique. Ces paramètres physiques sont les phénomènes observables de l'environnement physique et inclut, par exemple, le température ambiante, la lumière du soleil ou la présence d'un utilisateur. Ils sont généralement mesurés par des objets communicants appelés *capteurs*. L'analyse de ces paramètres constitue le *contexte* des environnements. Par exemple, des valeurs élevées des paramètres physiques de température et de fumée peuvent constituer un contexte d'incendie. Dey et Abowd définissent le contexte comme l'ensemble des informations permettant de caractériser la situation des utilisateurs, des lieux ou des objets [Dey01]. Les systèmes ubiquitaires analysent les changements de contexte pour décider du comportement à adopter et ainsi s'adapter. La nature du contexte est directement liée au domaine applicatif cible ; par exemple, le contexte décrivant un incendie est nécessaire aux applications d'alerte incendie. Le contexte donne aux systèmes ubiquitaires une vision haut niveau et dirigée de l'évolution des environnements physiques. Un environnement physique est ainsi abstrait pour former un *environnement ubiquitaire* comme illustré dans la figure 2.1. Un environnement ubiquitaire est constitué des services fournis par les objets communicants et du contexte nécessaire à l'adaptation des applications.

2.1.3 Criticité

En positionnant l'utilisateur au centre des préoccupations, les systèmes ubiquitaires ont pour objectif d'investir notre quotidien. Les technologies associées doivent être non intrusives pour l'utilisateur comme illustré par cette citation de Mark Weiser : "[Technologies] *weave themselves into the fabric of everyday life until they are indistinguishable from it.*" [Wei91]. Pour cela, les systèmes ubiquitaires doivent non seulement prendre des décisions pour l'utilisateur mais aussi être robustes pour ne pas interférer négativement sur les activités quotidiennes

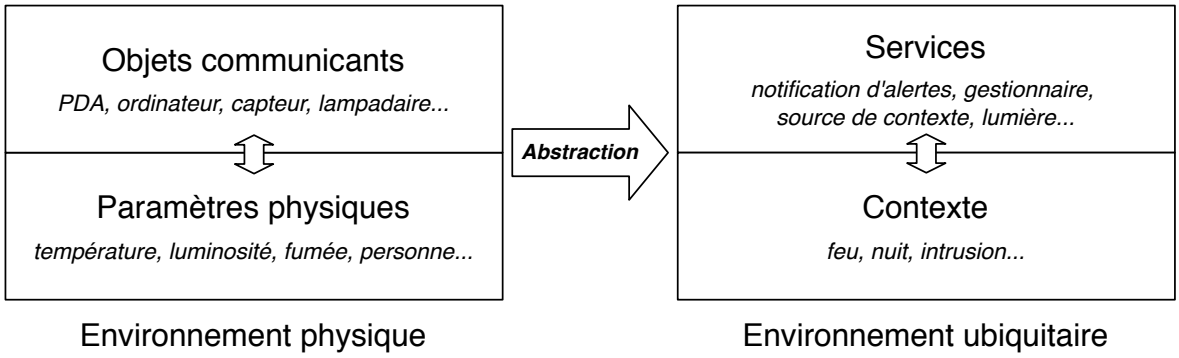


FIG. 2.1: Environnement ubiquitaire

des utilisateurs. De plus, l'informatique ubiquitaire touche des domaines applicatifs potentiellement critiques comme le domaine hospitalier [IBM06] ou le domaine de la gestion des accidents autoroutiers [CCG+07]. Par exemple, le service des urgences d'un hôpital ubiquitaire alloue automatiquement les ressources correspondant à l'arrivée de patients critiques. Il est donc primordial de s'assurer de la robustesse des systèmes ubiquitaires en situation réelle et ceci quelque soit le *scénario*, c'est-à-dire, quelque soit l'évolution particulière du contexte et des services sur une période donnée. Par exemple, l'hôpital ubiquitaire doit être capable de gérer l'important flux de blessés consécutif à un accident de la route. Les outils de production des applications doivent donc permettre de valider les comportements des applications avant leur déploiement en situation réelle.

2.2 Caractérisation des applications ubiquitaires

Une application ubiquitaire automatise des tâches pour le compte des utilisateurs. Prenons par exemple, dans le domaine de l'immotique, une application d'alerte incendie représentée dans la figure 2.2. En cas d'incendie, un détecteur de feu notifie un gestionnaire d'alerte incendie qui modifie l'environnement physique pour faciliter l'évacuation du bâtiment ; sur ordre du gestionnaire d'alerte incendie, les gestionnaires de lumières et de portes modifient respectivement l'état des lumières et des portes pour passer en configuration incendie. En configuration normale, le gestionnaire de lumières modifie l'état des lumières en fonction des informations des capteurs de luminosité se situant à l'extérieur et de l'agenda indiquant les horaires de fermeture du bâtiment. De cet exemple, nous pouvons en déduire qu'une application capture les changements de contexte grâce à des objets communicants d'acquisition (par exemple détecteurs de feu, capteurs de température et caméras) et adaptent l'environnement en conséquence grâce à des objets communicants appelées actionneurs (par exemple lumières et alarmes) et des objets communicants de rendus (par exemple téléphones, écrans et enceintes). Nous remarquons aussi que l'intelligence de l'application d'alerte incendie, c'est-à-dire l'entité logicielle menant à bien la tâche de l'application, est concentrée dans un service appelé gestionnaire d'alerte incendie. Dans cette section, nous analysons l'architecture des applications ubiquitaires et nous introduisons la découverte de services qui est le mécanisme fondamental des applications ubiquitaires dans la gestion de la volatilité des objets communicants.

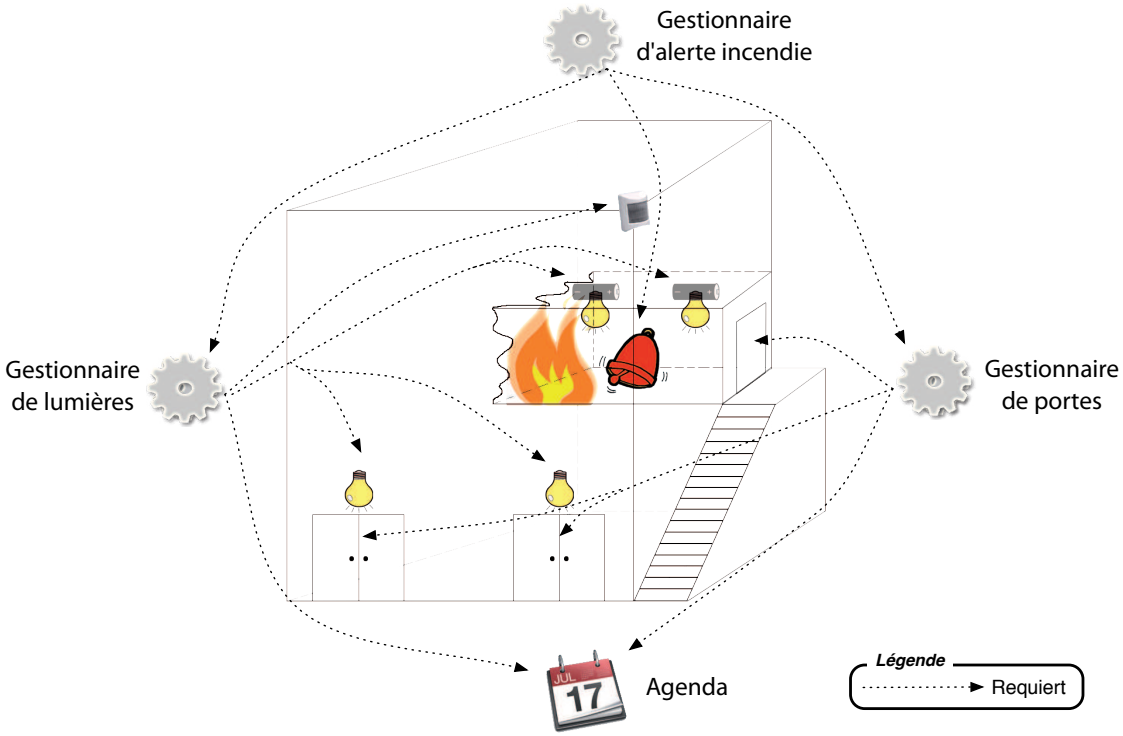


FIG. 2.2: Application d'alerte incendie

2.2.1 Architecture

Comme illustrée sur la figure 2.2, une application ubiquitaire implique un ensemble de communications entre des couples de services. Dans chacun de ces couples, un service client initie une requête vers un service serveur qui fournit une information en retour. Par exemple, un gestionnaire de lumières consulte un agenda pour savoir si le bâtiment est fermé. Nous parlerons de communications de *un vers un*. Un service serveur peut aussi avoir simultanément plusieurs services clients. Par exemple, lorsque le bâtiment passe en état fermé, le service serveur agenda notifie deux services clients, les gestionnaires de lumières et de portes. Nous parlerons de communications de *un vers plusieurs*. Les services communiquent entre eux en respectant un ensemble de règles qui définissent un *bus logiciel* par analogie au bus matériel.

Les applications ubiquitaires reposent principalement sur les réseaux *PAN (Personal Area Network)* et *LAN (Local Area Network)*. Les réseaux personnels ou PAN sont particulièrement adaptés pour les applications gérant des utilisateurs mobiles dans des grands espaces ouverts comme des villes ou des autoroutes. Ces espaces ne sont pas favorables au déploiement d'une infrastructure de communication continue et donc à l'utilisation d'autorités centralisées que des services pourraient interroger à tout moment. Les services communiquent alors directement entre eux par des réseaux ponctuels basés sur des interfaces de communication à courte portée comme le Bluetooth. Par exemple, une application de gestion des lumières d'une ville permet aux lumières de communiquer entre elles pour adapter leur luminosité en fonction de la localisation des utilisateurs [BC06]. Un autre exemple est l'application Ubibus qui aide des utilisateurs malvoyants équipés d'un téléphone à prendre les transports publiques [BCPB04].

Dans les réseau locaux ou LAN, la présence d'autorités centralisées facilite la conception et

la maintenance en concentrant les informations sur les services et leurs données. Les services sont découplés les uns des autres et les services serveurs peuvent être remplacés sans affecter les services clients. La dynamicité des objets communicants est ainsi gérer de façon transparente. Par ailleurs, contrairement aux réseaux personnels, des politiques de sécurité peuvent être mises en places via ces autorités. Ainsi, les applications ubiquitaires existantes utilisent cette architecture dès que l'environnement le permet, notamment dans les espaces fermés comme les maisons ou les bâtiments [RHC+02, GSSS02]. Dans le cadre de cette thèse, nous nous restreignons aux réseaux locaux.

2.2.2 Découverte de services

Les applications ubiquitaires résultent de la coordination de services hétérogènes dont la disponibilité varie au cours du temps. Pour gérer la dynamicité des services, les applications doivent prendre en compte l'ensemble des scénarios possibles. Cependant, il serait inapproprié de spécifier les adresses physiques des services lors de leur développement et ainsi de lier statiquement les applications à des services concrets. Pour simplifier la conception des applications et permettre leur évolution, les mécanismes de *découverte de services* ajoutent un niveau d'indirection à la localisation physique des services [ZMN05]. Pour cela, un service serveur est définie par des couples propriété-valeur et un service client exprime ses besoins en spécifiant les valeurs des propriétés des services serveurs recherchés. Les besoins du service client constituent un contrat que le service serveur devra respecter. La découverte de services peut être distribuée ou centralisée. Dans le premier cas, le contrat exprimé par le service client est envoyé à tous les services serveurs disponibles. Les services serveurs respectant le contrat envoient alors une réponse au service client. Dans le dernier cas, le contrat est envoyé vers un annuaire qui centralise les descriptions des services serveurs enregistrés et les adresses physiques associées. En retour, le service client reçoit la liste des services serveurs correspondants. Par exemple, dans la figure 2.3, les lumières s'enregistrent sur l'annuaire (étape 1). Le gestionnaire de lumières spécifie contractuellement qu'il a besoin de lumières se situant dans les couloirs et reçoit en retour la liste des services serveurs correspondants (étape 2). Enfin, il initie une requête vers une lumière (étape 3). Pour gérer la disparition des services, les protocoles de découverte de services proposent un mécanisme de bail *(leasing)* qui oblige les services serveurs à s'enregistrer à interval de temps régulier dans l'annuaire. Autrement, le service est supprimé de l'annuaire et ne peut plus être découvert. Pour maintenir à jour la liste des services découverts par les services clients, des approches proposent de mettre à jour dynamiquement cette liste [KkK06].

2.3 Caractérisation des services

Un service a des *fonctionnalités* qui lui permettent d'interagir avec les autres services. Par exemple, le service lumière permet d'éteindre et d'allumer une lumière. Le *type de services* lumière est présent sur toutes les lumières mais est implémenté différemment selon que la lumière soit un lampadaire ou une lumière de bureau. Pour certaines applications, il est nécessaire de pouvoir distinguer chaque service au sein d'un même type de services. Par exemple, l'application d'alerte incendie allume certains services lumière et en éteint d'autres pour indiquer les voies d'évacuation. Il fait ainsi une sélection des services lumières par rapport à leur localisation. Pour cela, il est nécessaire d'ajouter des propriétés non fonctionnelles, ou *attributs*, aux types de services.

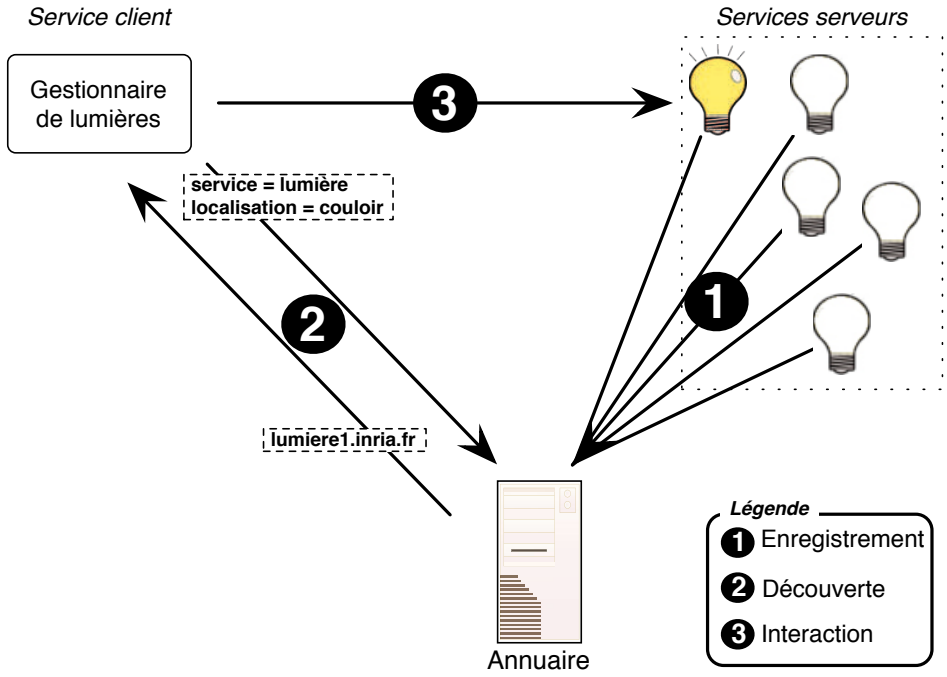


FIG. 2.3: Découverte de services

Dans cet section, nous caractérisons les services pour identifier leurs besoins fonctionnels et non fonctionnels. Dans un premier temps, nous distinguons deux catégories de services, les services primitifs et les services de coordination. Dans un deuxième temps, nous déterminons les éléments constitutifs des services, les fonctionnalités et les propriétés. Pour les fonctionnalités, nous identifions les modes de communication existants et les données échangées.

2.3.1 Aperçu

L'hétérogénéité des services nécessite des stratégies différentes d'intégration au sein d'un environnement ubiquitaire. Comme illustré sur la figure 2.4, nous distinguons deux types de services : les services primitifs et les services de coordination.

2.3.1.1 Services primitifs

Les environnements physiques sont équipés d'objets communiquants exposant des services via un bus logiciel ad hoc. Par exemple, les bâtiments sont équipés de caméras et d'alarmes tandis que les maisons sont généralement dotées d'équipements multimédias et de boitiers intégrés *(set-top boxes)*. Ces objets communicants ont des capacités de calcul variables et des fonctionnalités qui sont potentiellement très simples (par exemple allumage d'une lumière) ou très compliqués (par exemple réception du flux vidéo d'une caméra). Les bus logiciels varient aussi en fonction des capacités et des domaines applicatifs des objets communicants. Par exemple, les téléphones portables utilisent le bus logiciel Bluetooth et les caméras de sécurité utilisent le bus logiciel UPnP. Les bus logiciels ne sont pas interopérables et proposent des modes de communication variés.

Chaque objet communicant héberge un ou plusieurs services. Par exemple, un PDA peut

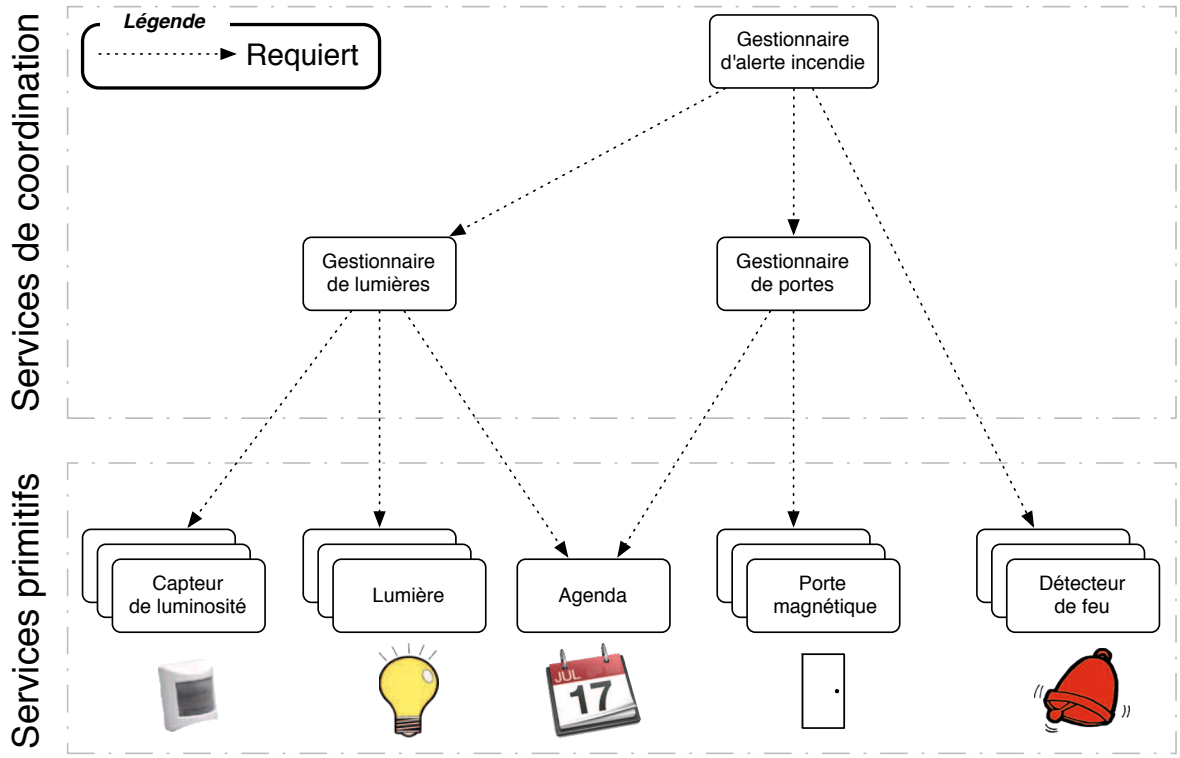


FIG. 2.4: Services composant une application d'alerte incendie

héberger un service de notification ainsi qu'un service de contrôle à distance. Ces services sont qualifiés de *primitifs*. Les services primitifs sont exclusivement des services serveurs. Ils englobent les services d'acquisition (par exemple des capteurs de luminosité), les services qui interagissent avec les utilisateurs (par exemple des lumières, des haut-parleurs et des écrans tactiles) et les bases de données (par exemple des agendas et des bases de profils utilisateurs). Pour l'application d'alerte incendie, il s'agit des capteurs de luminosité, des lumières, de l'agenda, des portes et des détecteurs de feu. L'intégration des services primitifs dans une application distribuée demande de lier les opérations des bus logiciels ad hoc des objets communicants au bus logiciel de l'infrastructure logicielle.

2.3.1.2 Services de coordination

Une application doit coordonner de façon cohérente les services pour remplir la tâche qui lui a été attribuée. Pour cela, une application ubiquitaire utilise des *services de coordination*. Pour percevoir les changements de contexte et agir sur l'environnement, les services de coordination utilisent les services primitifs. Ils sont clients de services primitifs et d'autres services de coordination. Par exemple, le gestionnaire d'alerte incendie utilise le détecteur de feu et commande les gestionnaires de lumières et de portes. Le gestionnaire de lumières est aussi un service de coordination qui agit sur les services primitifs de lumières.

L'intégration des services de coordination est moins contraignante que celle des services primitifs car très souvent, elle s'affranchit de l'existant. En tant que services clients, les services de coordination doivent être capables de localiser et d'interagir avec les services serveurs.

2.3.2 Fonctionnalités

Les communications entre services sont effectuées dans deux plans fonctionnels disjoints : le plan de contrôle et le plan de données. Le plan de contrôle définit le mode de communication (appelé aussi *mode d'interaction*), c'est-à-dire comment un service initie, accepte et termine une communication. Le plan de données définit comment un service produit et consomme des données. Une *fonctionnalité* définit comment le plan de contrôle et le plan de données doivent être associés. Le service client et le service serveur doivent communiquer selon le même mode de communication. Ils doivent aussi utiliser la même représentation des données. Nous caractérisons ici les modes de communication et les données échangées par les services.

2.3.2.1 Synchronisme des communications

Les services communiquent de façon synchrone ou asynchrone. Dans une communication synchrone, le service client envoie une requête ; son exécution est bloquée en attendant la réponse du service serveur. Le service serveur traite la requête et envoie la réponse au service client. À la réception de la réponse, le service client reprend son exécution.

Modèle RPC. L'appel à distance de type RPC est le mode de communication synchrone le plus répandu [Mic88]. C'est un mode de communication de un vers un. Il est adapté pour récupérer des mesures de capteurs et interroger des bases de données. Par exemple, le gestionnaire de lumières envoie une requête de type RPC à l'agenda pour savoir si le bâtiment est actuellement fermé au public.

Il existe donc un couplage fort entre le service client et le service serveur dans le modèle RPC. Or, la dynamicité des environnements ubiquitaires nécessitent la propagation des changements de contexte vers de multiple services qui ne connaissent pas à l'avance l'identité des producteurs. Cette propagation demande donc un couplage lâche entre les différents services. Les communications asynchrones réalisent ce couplage lâche en rendant la requête du service client non bloquante et la réponse optionnelle.

Modèle RPC asynchrone. Une évolution du modèle RPC permet des communications asynchrones [ATK92]. Le modèle RPC asynchrone est particulièrement intéressant pour les commandes de changement de statut d'un service qui ne requiert pas de réponse, par exemple, pour allumer une lumière. Lorsqu'une réponse est nécessaire, le modèle *future RPC* permet au service client, dans un deuxième temps, d'interroger régulièrement le service serveur [ATK92, Ame87]. On parlera de mode *PULL* car le service client récupère la donnée sur le service serveur. Le service serveur et le service client sont alors respectivement producteur et consommateur. Les modèles RPC synchrones imposent que le service client et le service serveur se connaissent. Au contraire, les environnements ubiquitaires nécessitent un anonymat entre les services pour faciliter la gestion de la dynamicité des services producteurs et consommateurs de contexte.

Modèle asynchrone anonyme. Les modèles de files de messages [BCSS99] *(message queueing)* et à espaces partagés [Gel85] *(shared dataspace)* proposent d'utiliser des intermédiaires logicielles ou physiques pour assurer l'anonymat. Les services communiquent respectivement par l'intermédiaire d'une file d'attente et d'opérations de diffusion *(broadcasting)*. Comme pour les modèles RPC asynchrones, les services clients restent bloqués en attendant

la donnée (mode PULL). Le modèle de files de messages est un modèle de communication de un vers un exclusivement. Le modèle à espaces partagés permet des communications de un vers plusieurs nécessaires à la propagation du contexte. Il est particulièrement utilisé dans les réseaux personnels où aucune autorité centrale n'est disponible. Chaque service n'est alors sensible qu'au changement de contexte dans son immédiate proximité [CB03, BC06], limitant le modèle à un ensemble restreint d'applications.

Cependant, le mode *PULL* n'est pas adapté à des environnements ubiquitaires saturés de services et dont les objets communicants comme les capteurs sont potentiellement pauvres en ressources.

Modèle événementiel. Le *modèle événementiel* introduit le mode *PUSH* et permet ainsi un découplage total entre les services serveurs et les services clients [EFGmK03]. Les communications sont basées sur des échanges d'événements qui sont des transitions ponctuelles et significatives d'états de l'environnement physique. Par exemple, un départ d'incendie peut faire l'objet d'un événement. Les services clients souscrivent à un patron d'événements auprès d'un *courtier d'événements* qui tient à jour la liste des souscripteurs. Le patron d'événements joue alors le rôle de filtre. Un service serveur publie un événement sur le courtier d'événements qui le transmet aux souscripteurs correspondants de façon asynchrone grâce au mode PUSH.

Le modèle événementiel est donc plus adapté à la gestion des environnements ubiquitaires. Par ailleurs, le modèle événementiel a été évalué comme plus expressif que le modèle à espaces partagés [BZ01]. C'est pourquoi, son utilisation pour la gestion des environnements ubiquitaires est très répandue [RHC+02, GSSS02, CCG+07].

2.3.2.2 Communications à état

Les modes de communication précédemment introduits sont *sans état* : le service serveur ne peut identifier si deux requêtes sont faites par le même service client. Pour certaines applications, le service serveur doit garder un historique des échanges de façon à adapter son comportement en fonction de chaque service client. Nous parlerons de *communications à état*. Par exemple, les applications ubiquitaires nécessitent des objets communicants aux ressources limités et doivent donc contrôler l'utilisation de ces ressources ; un haut-parleur qui ne peut recevoir plus d'un flux audio à la fois doit pouvoir identifier chaque connexion pour réserver les ressources correspondantes et éventuellement refuser de nouvelles connexions. La notion de *session* a été introduite dans cet objectif [Inf81, Ros02]. Une session permet de conserver des informations spécifiques à un service client à travers plusieurs requêtes. Une session est délimitée dans le temps par une connexion et une déconnexion qui correspondent respectivement à une réservation et une libération de ressources, par exemple des ports ou la bande passante.

Les sessions sont particulièrement adaptées à la résolution des problèmes d'hétérogénéité des services. Par exemple, dans le domaine de la téléphonie, le protocole SIP *(Session Initiation Protocol)* s'appuie sur des sessions pour gérer les échanges de flux audio entre deux terminaux téléphoniques hétérogènes [Ros02]. À la connexion, les services négocient les caractéristiques de la session (par exemple le codec audio) qui seront appliquées pour l'échange du flux audio jusqu'à déconnexion de l'un des services.

2.3.2.3 Direction des données

L'orientation du plan de données est généralement indépendante de l'orientation du plan de contrôle. Ainsi, le modèle RPC permet au service client d'envoyer des données comme un message d'alerte et au service serveur de retourner des données comme une mesure de luminosité. Les modèles asynchrones mettent en place un schéma producteur/consommateur où le service serveur et le service client sont respectivement producteur et consommateur. La direction des données est alors unidirectionnelle. À l'inverse, une session ne donne pas d'indication quant à la l'orientation du plan de données et les flux de données sont unidirectionnels ou bidirectionnels.

2.3.2.4 Types des données échangées

La variété des services et de leurs fonctionnalités impliquent naturellement une diversité des types de données échangées. Cette diversité s'exprime par la nature et le format des données.

Nature des données. La nature des données varie en fonction des domaines applicatifs et des services associées. Par exemple, les données échangées peuvent être du texte (par exemple un message d'alerte affiché par un PDA), des mesures (par exemple des mesures produites par un capteur), des images (par exemple des images stockées par une base de données), un flux vidéo (par exemple un flux vidéo produit par une caméra) ou un flux audio (par exemple un flux audio produit par un téléphone).

Formats des données. Un format décrit l'encodage des données. Le type MIME (*Multipurpose Internet Mail Extensions*) liste la plupart des formats disponibles [FB96a, FB96b]. Il est notamment utilisé pour décrire le format des courriers électroniques dans le protocole SMTP [Pos01]. Le type MIME inclut le format *plain* des données textuelles non structurées, les formats pour la structuration du texte (par exemple HTML [D. 99] et XML [W3C96]), les codecs audio (par exemple MP3 et WAV), les codecs vidéo (par exemple MPEG4 et Quicktime Movie) et les formats de type images (par exemple GIF et PNG).

L'hétérogénéité des formats de données pose un problème d'interopérabilité entre les services. Pour résoudre ce problème, de nombreuses approches proposent de transcoder les données envoyées par le producteur [RCAM+05, HcFWJ05]. Le transcodage est cependant inadapté pour les communications dont la latence doit être minimale (par exemple la vidéoconférence) car c'est une solution coûteuse en temps. De plus, le transcodage peut être évité dans de nombreux cas en négociant le type de données. Si les deux participants se mettent d'accord, l'échange de données peut avoir lieu sans transcodage.

2.3.3 Propriétés

La multitude des services d'un environnement ubiquitaire constitue autant de ressources que les applications sélectionnent pour remplir leur tâche. Pour effectuer cette sélection, les services doivent être différenciés les uns par rapport aux autres. Pour cela, un service est décrit par des propriétés fonctionnelles et non fonctionnelles. Une propriété est non fonctionnelle lorsqu'elle spécifie ses capacités de calcul (par exemple la puissance d'un ordinateur) ou de rendu (par exemple la résolution d'un écran), son aspect physique (par exemple la couleur d'une lumière et le modèle d'un PDA) ou sa localisation. Enfin, une propriété non fonctionnelle peut être volatile lorsque sa valeur change à l'exécution. Par exemple, la propriété *état* d'une

lumière prend les valeurs *allumée* et *éteinte*. Nous qualifierons les propriétés non fonctionnelles d'*attributs*.

2.4 Bilan

Les systèmes ubiquitaires proposent de nouveaux défis en terme d'hétérogénéité des services, de dynamicité des environnements et de criticité des domaines applicatifs. La caractérisation des systèmes ubiquitaires met en évidence l'existence de besoins logiciels communs à la plupart des applications ubiquitaires. Ces besoins sont fonctionnels, dans le cas, par exemple des communications asynchrones ou de la gestion des flux de données. Ces besoins sont non fonctionnels, dans le cas, par exemple de la gestion de la dynamicité et de la robustesse des applications. Des outils répondant à ces besoins sont nécessaires pour factoriser les efforts de production des applications ubiquitaires. Les besoins auxquels ils répondent ne sont pas dépendants d'un domaine applicatif spécifique.

De la caractérisation des services a émergé la notion de type de services. Un type de services partage des caractéristiques communes : les attributs et les fonctionnalités. La spécification de ces types permet d'organiser et d'abstraire les ressources des environnements ubiquitaires. La représentation haut-niveau résultante facilitent alors la gestion des ressources. Elle permet la création d'outils adaptés facilitant l'intégration des services dans les applications. Ces outils sont spécialisés par rapport à un domaine applicatif spécifique. Dans le chapitre suivant, nous présentons les outils utilisés pour la production et l'exécution des systèmes distribués.

Chapitre 3

Programmation des systèmes distribués

Les systèmes distribués doivent prendre en compte la complexité des environnements ubiquitaires tout en étant suffisamment fiables pour mener à bien leur mission applicative. Ces problèmes nécessitent des solutions adaptées s'appuyant sur des technologies éprouvées. Ces technologies sont regroupées sous le terme d'*intergiciel*. Les intergiciels fournissent des outils adaptés et standardisés pour abstraire la complexité des systèmes distribués. Un intergiciel est la couche entre les services et le système d'exploitation. Il permet de faire communiquer des services hétérogènes entre eux en leur fournissant un environnement d'exécution. Cet environnement est constitué de *services systèmes* pour assurer, à l'exécution, des tâches communes à la plupart des applications ubiquitaires. Ces tâches gèrent les aspects non fonctionnels des domaines applicatifs pour lesquels les intergiciels sont dédiés. Par exemple, la persistance permet aux applications Web de perpétuer l'existence des données au-delà de l'exécution des services grâce à des systèmes de stockage non volatile comme les bases de données. D'autres services systèmes comme la découverte de services sont présents dans la plupart des intergiciels. Certains services systèmes, grâce à un processus de réification, sont accessibles dans le code applicatif via un *canevas de programmation*. Du point de vue du développeur, le canevas de programmation est la partie visible de l'intergiciel. C'est une interface de programmation ou API *(Application Programming Interface)* de haut niveau qui aide à l'implémentation des services. Les intergiciels ont pour rôle de *masquer l'hétérogénéité* des services composant les applications en permettant notamment l'intégration des services existants. L'intégration se fait par le biais d'adaptateurs qui permettent de faire le lien entre les opérations du service existant et celles du bus logiciel de l'intergiciel. La dernière fonction principale d'un intergiciel est de *masquer la distribution* des services sur les différents objets communicants de l'environnement physique.

Dans ce chapitre, nous introduisons les principes fondamentaux de la programmation des systèmes distribués. Nous présentons le modèle RPC qui définit un paradigme essentiel à la distribution des services. Nous introduisons la programmation à objets répartis et la programmation orientée composant. À travers la programmation à objets répartis, nous nous intéressons à deux mécanismes essentielles aux systèmes ubiquitaires que sont la localisation des services et le modèle événementiel. Avec la programmation orientée composant, nous présentons une évolution dans la manière de concevoir les systèmes distribués et introduisons la notion d'architecture logicielle.

3.1 Le modèle RPC

Le modèle RPC *(Remote Procedure Call)* est fondamental pour la programmation des systèmes distribués car il permet de masquer la distribution et l'hétérogénéité des services. Les différentes implémentations de ce modèle sont des dérivées de la technologie RPC [Mic88].

Dans ce modèle, chaque service doit déclaré une interface, comportant les fonctionnalités qu'il fournit. Cette interface constitue un contrat que le développeur devra respecter lors de l'implémentation du service. Les interfaces des services sont écrites dans des langages de description d'interfaces ou IDL *(Interface Description Language)*. Les IDL sont généralement indépendants du langage de programmation de la plate-forme, assurant ainsi leur portabilité. Comme illustrée dans la figure 3.1, une déclaration d'interface est composée d'attributs et de signatures d'opérations.

```
1 interface GestionnaireDeLumières {
2   attribute boolean configurationFeu;
3   readonly attribute unsigned long priorité;
4   void mettreEnConfigurationFeu(in boolean presenceFeu);
5 }
```

FIG. 3.1: Déclaration d'un service avec l'IDL CORBA

En plus d'être une référence haut niveau pour les développeurs, les déclarations d'interfaces simplifient la programmation des systèmes distribués en permettant aux développeurs de programmer son application comme si elle était locale. La compilation d'une déclaration d'interface génère des talons clients *(stubs)* et des talons serveurs *(skeletons)* qui permettent respectivement au client d'appeler le serveur distant et au serveur de recevoir l'appel et de répondre au client distant (Figure 3.2). Le talon client joue le rôle d'un *mandataire*. Ce mandataire prend en charge les détails d'implémentation de l'appel comme sa sérialisation et les erreurs du bus logiciel sous-jacent. La requête client est transportée par le bus logiciel pour être délivrée au talon serveur qui désérialise la requête et la délègue à l'implémentation du serveur. La génération des talons à partir d'un langage déclaratif permet de décharger les développeurs de tâches répétitives et sujettes aux erreurs.

Selon les approches, une déclaration d'interface prend diverses formes, chaque approche fournissant son propre compilateur d'IDL : une syntaxe à la C pour RPC [Mic88], une interface Java pour RMI [Dow98], une syntaxe à la C++ pour ORB [OHE97] *(Object Request Broker)* et un format XML pour les services Web [W3C].

3.2 Programmation orientée objet

Les *intergiciels à objets répartis* combinent la programmation orientée objet et le modèle RPC. Le développeur s'appuie sur les mécanismes de classes, d'héritage, de polymorphisme et d'encapsulation pour développer des services dont chaque instance est représentée par un objet à l'exécution. Les classes permettent de créer des types de services qui seront instanciés autant de fois que nécessaire. L'héritage facilite la réutilisation de l'existant. En permettant de substituer des services équivalents, le polymorphisme permet aux applications de gérer plus facilement la dynamicité des systèmes distribués. Enfin, l'encapsulation cache les détails d'implémentation d'un serveur à ses clients, l'interface serveur qui définit des méthodes et

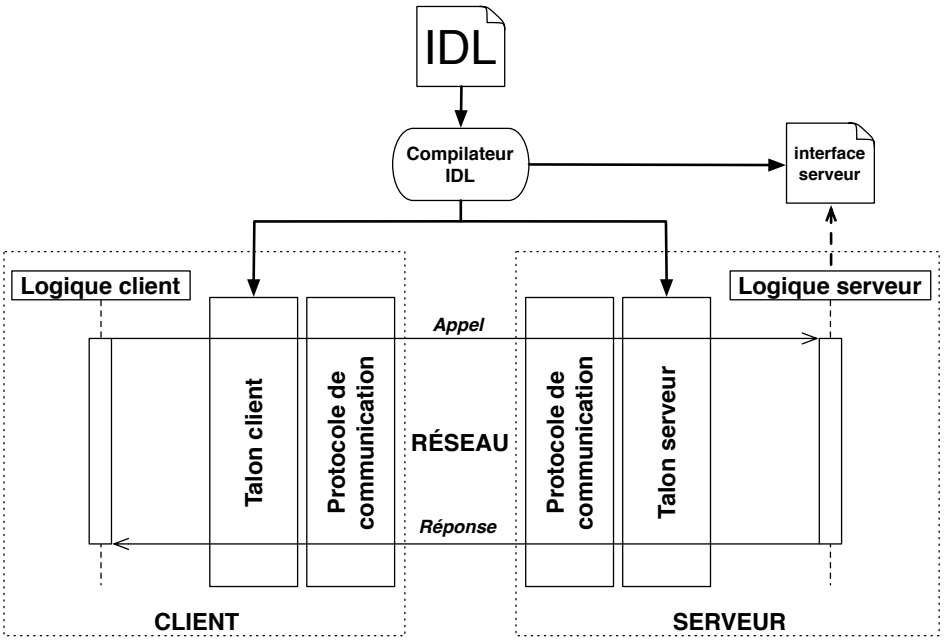


FIG. 3.2: Le modèle RPC (adapté de [BN84])

des attributs étant la seule information visible par les clients. Le modèle RPC a été étendu aux objets pour permettre un accès homogène grâce à l'encapsulation des données et à la création dynamique d'objets distants. Des technologies comme Java RMI (Remote Method Invocation [Dow98]) ou ORB (Object Request Broker [OHE97]) sont des instances de cette extension.

3.2.1 Localisation d'un service

La dynamicité des systèmes distribués rend la disponibilité des services non déterministe lors du développement. Les intergiciels proposent des services de nommage pour abstraire la localisation physique des services. Les développeurs désignent ainsi les services par des noms symboliques sous forme de chaînes de caractères. Les protocoles de découverte de services ajoutent un niveau d'indirection supplémentaire en permettant des recherches de services par rapport à leurs propriétés. Ces propriétés incluent les fonctionnalités fournies par le service, généralement son interface, ainsi que des propriétés non fonctionnelles sous la forme d'attributs. Pour éviter des sollicitations inutiles des services potentiellement pauvres en ressource, les intergiciels mettent en place un *courtier de services* qui se charge de l'enregistrement des services et résout les requêtes de découverte de services.

La programmation d'un service serveur consiste à enregistrer ses propriétés sur le courtier de services. Elle est effectuée en trois étapes : (1) connexion au courtier de services, (2) spécification des propriétés du service serveur, c'est-à-dire son interface et les valeurs de ses attributs et (3) envoi de la requête d'enregistrement au courtier de services. Un exemple de programmation de l'enregistrement de services dans le contexte de Jini est présentée dans la figure 3.3. Jini est un intergiciel Java dont l'interface de programmation pour l'enregistrement et la découverte de services est représentatif des intergiciels à objets [Sun00]. L'exemple illustre l'enregistrement d'un service d'alerte textuelle déployé sur le PDA d'Alice.

Le service se connecte au courtier de services en spécifiant l'adresse physique correspondante (« `jini ://192.168.0.12 :4160` », ligne 2) et en créant un gestionnaire de courtiers qui est en charge de résoudre l'adresse spécifiée (ligne 5). Les attributs sont spécifiés en instanciant des classes définies par l'utilisateur, c'est-à-dire `Propriétaire` et `Résolution` (lignes 10 et 11). Pour faciliter la sérialisation des données, les classes des attributs Jini ne peuvent pas contenir de champs primitifs au risque de lever une exception à l'exécution. Ainsi, la chaîne de caractères « `ALICE` » définit le propriétaire du service (ligne 10) et les chaînes de caractères « `480` » et « `640` » définissent la résolution du PDA (ligne 11). Les propriétés fonctionnelles sont spécifiées en créant un objet typé par l'interface du service (lignes 13 et 14). Dans Jini, cet objet est un mandataire utilisé par les clients pour invoquer le service `AlerteTextuelle`. Finalement, ligne 17, la requête d'enregistrement est envoyée au courtier de services.

```
1  /* (1) Connexion au courtier de services */
2  LookupLocator lookupLocator = new LookupLocator("jini://192.168.0.12:4160");
3  String[] groupes = new String[]  "" ;
4  LookupLocator[] localisations =  lookupLocator ;
5  LookupDiscoveryManager gestionnaireDeCourtiers
6     = new LookupDiscoveryManager(groupes, localisations, null);
7
8  /* (2) Spécification des propriétés du service */
9  Entry[] attributs = new Entry[2];
10 attributs[0] = new Propriétaire("ALICE");
11 attributs[1] = new Résolution("480", "640");
12
13 net.jini.export.Exporter exporter = new net.jini.jrmp.JrmpExporter();
14 IAlerteTextuelle serverProxy = (IAlerteTextuelle) exporter.export(this);
15
16 /* (3) Envoi de la requête d'enregistrement au courtier de services */
17 JoinManager joinManager =
18    new JoinManager(serverProxy, attributs, new ServiceID(...),
19                                 gestionnaireDeCourtiers, new LeaseRenewalManager());
20
```

FIG. 3.3: Enregistrement d'un service AlerteTextuelle en Jini

La programmation des services clients consiste à créer des mandataires des services serveurs pour ensuite les invoquer. Elle est effectuée en cinq étapes : (1) connexion au courtier de services, (2) spécification du patron *(template)* des services recherchés (3) envoi de la requête de découverte de services au courtier de services, (4) conversion de type des mandataires de services retournés et (5) invocation des services. Un exemple de découverte de services Jini est présenté dans la figure 3.4. Le client se connecte au courtier de services aux lignes 2 et 3. Les attributs et l'interface du service recherché sont spécifiés aux lignes 7 et 8 et le patron est créé à la ligne 10. Jini ne permet de spécifier que des égalités strictes pour les attributs recherchés ; dans la figure 3.4, pour le service recherché, la propriétaire devra être « `ALICE` ». Certaines interfaces de programmation permettent de spécifier des patrons de services plus complexes en utilisant des chaînes de caractères pour définir des relations de supériorité, d'infériorité ou d'inclusion [OHE97, FSM08]. La requête de découverte de services est paramétrée par le patron de services (ligne 13). Le type générique du mandataire retourné est converti dans le type du service recherché (ligne 16). Le service découvert est ensuite invoqué (ligne 19).

```
1  /* (1) Connexion au courtier de services */
2  LookupLocator lookupLocator = new LookupLocator("jini://192.168.0.12:4160");
3  ServiceRegistrar courtier = lookupLocator.getRegistrar();
4
5  /* (2) Spécification du patron des services recherchés */
6  Entry[] attributs = new Entry[1];
7  attributs[0] = new Propriétaire("ALICE");
8  Class[] serviceType = new Class[] {IAlerteTextuelle.class};
9
10 ServiceTemplate patron = new ServiceTemplate(null, serviceType, attributs);
11
12 /* (3) Envoi de la requête de découverte de services au courtier de services */
13 Object o = courtier.lookup(patron);
14
15 /* (4) Conversion de type de la référence retournée */
16 IAlerteTextuelle server = (IAlerteTextuelle) o;
17
18 /* (5) Invocation du service */
19 server.afficherAlerte(2, "Intrusion détectée !!");
```

FIG. 3.4: Découverte d'un service AlerteTextuelle en Jini

3.2.2 Le modèle événementiel

Comme introduit dans le chapitre 2, le modèle événementiel est adapté à la gestion des environnements ubiquitaires. Il découple totalement les producteurs des consommateurs, permet des communications de un vers plusieurs et offre des facilités de programmation, comme par exemple la gestion automatique des listes de souscripteurs.

JMS *(Java Message Service)* met en œuvre le modèle événementiel [JMS] et nous servira d'illustration pour introduire les interfaces de programmation associées. La programmation des producteurs d'événements est effectuée en trois étapes : (1) connexion au courtier d'événements, (2) création d'un événement et (3) envoi de l'événement au courtier d'événements. La figure 3.5 illustre ces trois étapes pour la publication d'un événement de luminosité. Les caractéristiques du courtier d'événements sont stockées dans l'objet `context` de la classe `InitialContext` (lignes 2 à 6). Le producteur se connecte au courtier d'événements en invoquant la méthode `lookup` de cet objet avec en argument le sujet de la publication, c'est-à-dire « `Luminosité` » (lignes 8 à 10). L'objet `topic` résulte de cette connexion. L'événement à envoyer est créé aux lignes 14 et 15. L'événement est ici encapsulé dans un type `ObjectMessage` fourni par JMS (lignes 13 et 17). Ce type permet de spécifier des propriétés, par exemple « `Localisation` » (ligne 16), qui permettront aux souscripteurs de définir des critères de filtrage. Finalement, l'événement est publié en invoquant la méthode `publish` de l'objet `publisher` créé à partir des informations fournies lors l'étape de connexion au courtier (lignes 20 et 21).

La programmation des consommateurs d'événements est effectuée en cinq étapes : (1) connexion au courtier d'événements (2) spécification du patron d'événements, (3) envoi de la souscription au courtier d'événements, (4) réception de l'événement et (5) conversion de type de l'événement reçu. La programmation de la souscription et de la réception d'un événement sont respectivement illustrées dans les figures 3.6 et 3.7. Le code développé pour se connecter au courtier d'événements est identique à celui du producteur (Figure 3.6, lignes 2 à 10). Ensuite, la spécification du patron d'événements diffère selon les approches. Dans les approches fondées sur le sujet, le patron d'événements se résume à un sujet généralement représenté

```
1  /* (1) Connexion au courtier d'événements */
2  Properties properties = new Properties();
3  ...
4  properties.put(Context.PROVIDER_URL, "jms://192.168.0.12:2345");
5  ...
6  InitialContext context = new InitialContext(properties);
7  ...
8  TopicSession session = ...
9
10 Topic topic = (Topic) context.lookup("Luminosité");
11
12 /* (2) Création d'un événement */
13 ObjectMessage event = session.createObjectMessage();
14 Luminosité luminosité = new Luminosité();
15 luminosité.setValeur(1250);
16 event.setStringProperty("Localisation", "Exterieur");
17 event.setObject(luminosité);
18
19 /* (3) Envoi de l'événement vers le courtier d'événements */
20 TopicPublisher publisher = session.createPublisher(topic);
21 publisher.publish(event);
```

FIG. 3.5: Programmation d'un producteur d'événements de luminosité en JMS

dans une chaîne de caractères. Les approches fondées sur le contenu utilisent des patron d'événements spécifiant des filtres sur les propriétés de l'événement recherché, par exemple « `(Localisation = 'Exterieur')` » (Figure 3.6, ligne 13). La souscription est effectuée en spécifiant le sujet (`topic`, figure 3.6, ligne 16), le patron d'événements (`filtre`, ligne 16) et un objet (`receiver`, Figure 3.6, ligne 17) qui sera appelé pour traiter la réception des événements. Pour recevoir les événements, le développeur doit implémenter la méthode `onMessage` (Figure 3.7, ligne 4). Enfin, le type des événements est converti pour extraire les informations nécessaires (Figure 3.7, lignes 3 à 8).

```
1  /* (1) Connexion au courtier d'événements */
2  Properties properties = new Properties();
3  ...
4  properties.put(Context.PROVIDER_URL, "jms://192.168.0.12:2345");
5  ...
6  InitialContext context = new InitialContext(properties);
7  ...
8  TopicSession session = ...
9
10 Topic topic = (Topic) context.lookup("Luminosité");
11
12 /* (2) Spécification du patron d'événements */
13 String filtre = new String("(Localisation = 'Exterieur')");
14
15 /* (3) Envoie de la souscription au courtier d'événements */
16 TopicSubscriber subscriber = session.createSubscriber(topic, filtre);
17 LuminositéReceiver receiver = new LuminositéReceiver();
18 session.setMessageListener(receiver);
19
```

FIG. 3.6: Programmation d'une souscription d'événements de luminosité en JMS

```
1  public class LuminositéReceiver implements MessageListener {
2
3     /* (4) Réception d'un événement */
4     public void onMessage(Message message) {
5        /* (5) Convertion de type de l'événement reçu */
6        ObjectMessage m = (ObjectMessage) message;
7        Luminosité luminosité = (Luminosité) message.getObject();
8     }
9  }
10
```

FIG. 3.7: Programmation de la réception d'événements de luminosité en JMS

3.3 Programmation orientée composant

Pour gérer la complexité et l'évolution des applications, il faut pouvoir s'appuyer sur une chaîne de production standardisée et faire levier sur des outils permettant l'assemblage et la réutilisation de services existants. Les intergiciels à objets ne satisfont pas à ces critères. La simplicité du langage IDL ne permet pas d'avoir une vue globale de l'architecture de l'application ni d'exprimer des besoins non fonctionnels. Les services sont considérés individuellement sans relation les uns avec les autres et l'assemblage entre les services doit être réaliser manuellement par les développeurs. Les développeurs doivent explicitement programmer les parties non fonctionnelles, en plus des parties fonctionnelles qui se retrouvent mêlées dans le code applicatif. Les services ainsi développés sont difficilement maintenables et réutilisables. Plus généralement, l'absence d'une vue globale est pénalisante pour la réalisation d'une chaîne de production cohérente et pour le développement en parallèle des services.

Les intergiciels à composants proposent une réponse à ces limitations en introduisant la notion d'architecture logicielle. L'architecture logicielle donne une vue globale d'une application dont la brique de base est le composant. Un ensemble d'outils permet la spécification, l'implémentation, l'assemblage et le déploiement des composants. Dans cette section, nous décrivons la chaîne de production des services.

Spécification. Un composant peut être défini comme une unité de composition avec des interfaces et des dépendances explicites spécifiées contractuellement [Szy02]. Un composant peut implémenter plusieurs interfaces qui représentent des ports de connexion pour les autres composants. Ces ports décrivent des fonctionnalités fournies ou requises. Le type d'un composant est défini par l'union de ses ports et de ses attributs. Les interfaces fournies sont reliées aux interfaces requises par des *connecteurs*. Les composants et les connecteurs sont déclarés dans une *architecture logicielle*. Les architectures logicielles sont généralement spécifiées dans un langage déclaratif de description d'architectures ou ADL *(Architecture Description Language)*. Les spécifications ADL permettent de générer du support de programmation pour l'implémentation et l'assemblage des composants [Sie00, ACN02].

Implémentation. Le support de programmation fourni par les intergiciels à composants permet la séparation des préoccupations fonctionnelles et non fonctionnelles. Les parties non fonctionnelles des composants sont généralement générées depuis les déclarations de l'architecture logicielle. Les développeurs peuvent alors se concentrer sur les parties fonctionnelles.

L'implémentation d'un composant implique l'implémentation de l'ensemble de ses interfaces déclarées comme fournies dans l'architecture logicielle.

Assemblage. L'assemblage est généralement spécifié dans des descripteurs XML [Sie00, BCS04] qui spécifient les connexions entre services clients et services serveurs. L'assemblage peut aussi être effectué dans le code applicatif [Sie00, ACN02, BCS04]. Le code généré depuis les spécifications ADL est alors utilisé pour instancier des connexions statiques entre les composants.

Instanciation. Les instances des composants sont exécutées dans un conteneur qui leur fournit un ensemble de services systèmes pour gérer, par exemple, le cycle de vie des composants, la persistance, les transactions et la qualité de service. Le cycle de vie d'un composant définit l'évolution du composant de sa création à sa destruction. Le code gérant le cycle de vie est généralement généré à partir de l'architecture logicielle.

3.4 Bilan

Dans ce chapitre, nous avons examiné le support apporté par les intergiciels pour la production d'applications. Les intergiciels permettent aux développeurs de gérer la dynamicité des environnements ubiquitaires à l'aide de canevas de programmation. Ces canevas de programmation simplifient la programmation de tâches répétitives et complexes comme la découverte et la gestion des événements. Pour cela, des bibliothèques génériques de code sont fournies. Ainsi, les requêtes de découverte de services sont réalisées par des opérations génériques. De même, les événements sont reçus par une méthode générique. Quant aux mandataires, nécessaires à l'invocation, ils sont générés à partir de spécifications haut niveau des services. Ces mandataires permettent des invocations statiquement typées. La programmation orientée composants apportent une évolution par rapport à la programmation orientée objet en facilitant l'assemblage des services et la programmation des aspects non fonctionnels grâce à la notion d'architecture logicielle. Il nous reste donc à examiner les solutions existantes pour la programmation des systèmes distribués et à évaluer le support proposé pour la production d'applications distribuées, dynamiques et fiables.

Chapitre 4

Solutions existantes pour la programmation des systèmes distribués

La programmation des systèmes distribués requiert des solutions différentes selon le domaine applicatif visé et la complexité des applications cibles. Le domaine d'application détermine les services systèmes nécessaires et les abstractions de programmation les plus appropriées. Ainsi, des approches plus généralistes comme Java RMI et CORBA fournissent les services systèmes essentiels à la plupart des applications distribuées (par exemple un service de nommage). Elles fournissent aussi des canevas de programmation assez expressifs pour capturer toute la généralité des systèmes distribués. Les approches pour les systèmes ubiquitaires fournissent des services systèmes et des abstractions de programmation facilitant la programmation de la mobilité et des changements de contexte. Enfin, les approches à base de composants facilitent la programmation de l'assemblage des applications.

Dans ce chapitre, nous examinons la capacité des solutions existantes à gérer la dynamicité des environnements ubiquitaires tout en assurant la fiabilité des applications développées. Nous nous intéressons plus particulièrement au support de programmation et aux vérifications que ces solutions mettent en œuvre dans leur canevas de programmation. Dans cette optique, nous illustrons la programmation des systèmes distribués traditionnels en nous appuyant sur les technologies Java RMI, CORBA, services Web et SIP. Avec les approches one.world et Olympus, nous abordons la programmation de la dynamicité des environnements ubiquitaires. Enfin, les approches à base de composants, EJB, CORBA CCM et ArchJava, nous permettent d'analyser l'utilité des architectures logicielles pour l'assemblage des applications.

4.1 Programmation des systèmes distribués traditionnels

Nous présentons trois solutions dont les interfaces de programmation sont basées sur le modèle RPC : RMI, CORBA et les services Web. RMI et CORBA sont des évolutions vers les objets tandis que les services Web sont une évolution vers l'échange de documents XML. Enfin, nous examinons à travers le protocole SIP *(Session Initiation Protocol)* les interfaces de programmation d'un protocole de communication. Nous évaluons le support de programmation et les vérifications proposés par ces solutions au travers de la programmation de mécanismes primordiaux pour la gestion de la dynamicité des environnements ubiquitaires : la localisation

des services et le modèle événementiel. Pour illustrer la localisation et l'invocation de services, nous nous appuyons sur un service d'alerte textuelle capable d'afficher un message, par exemple, sur un PDA. Ce service a deux attributs : l'attribut `propriétaire` et l'attribut `résolution` qui permettent de connaître respectivement le nom du propriétaire et la résolution de l'objet communicant sur lequel est déployé le service. Pour le modèle événementiel, nous prenons comme exemple un producteur et un consommateur d'événements de luminosité. Un événement de luminosité est caractérisé par un entier, donnant la valeur de la mesure, et par un booléen indiquant s'il s'agit de la luminosité extérieure. Par souci de concision, la gestion des exceptions est volontairement omise dans les exemples.

4.1.1 Java RMI

La technologie RMI *(Remote Method Invocation)* est une implémentation du modèle RPC dans Java. La figure 4.1 illustre une interface RMI. Le développeur définit, dans une interface étendant l'interface `Remote`, la ou les méthode(s) pouvant être appelée(s) à distance. C'est le cas par exemple de la méthode `afficherAlerte` de l'interface `AlerteTextuelle`. Le développeur doit ensuite fournir une classe serveur implémentant son interface `AlerteTextuelle` (non illustré dans l'exemple). Les classes talon sont générées dynamiquement à partir des interfaces. Le service client utilise alors l'interface `AlerteTextuelle` pour communiquer avec l'implémentation du serveur. RMI fournit des outils pour la résolution de noms et l'invocation des services.

```
1 public interface AlerteTextuelle extends java.rmi.Remote {
2   public String propriétaire;
3   public int résolution_largeur;
4   public int résolution_hauteur;
5   public void afficherAlerte(int priorité, String message);
6 }
```

FIG. 4.1: Interface RMI

La technologie RMI inclue un service de nommage qui permet d'associer un nom symbolique à l'adresse physique d'un service. Un courtier de services appelé *rmiregistry* maintient la liste de ces associations ; les services serveurs s'enregistrent auprès du *rmiregistry* afin que les services clients puissent ensuite les invoquer.

Enregistrement des services. La figure 4.2 illustre l'enregistrement du service d'alerte textuelle. Pour des raisons de sécurité, RMI oblige le courtier de services *rmiregistry* à être local, c'est-à-dire sur la même machine que les services enregistrés. La création du *rmiregistry* est illustrée à la ligne 2. L'enregistrement du service est effectué à la ligne 8. Il requiert le nom du service serveur représenté par une chaîne de caractères « `MonAlerteTextuelle` » et l'objet représentant le service serveur à enregistrer. Le courtier de services associe le nom à une référence vers cet objet. Le nom a pour but d'abstraire la localisation physique du service correspondant.

Localisation des services. RMI propose un service de nommage et une API pour résoudre les noms logiques des services comme illustré dans la figure 4.3. L'invocation est précédée de

```
1  /* création du courtier */
2  java.rmi.registry.LocateRegistry.createRegistry(1099);
3
4  /* (1) Connexion au courtier de services */
5  /* (3) Envoi de la requête d'enregistrement au courtier de services */
6  AlerteTextuelle alerte_textuelle_impl = new AlerteTextuelle_Impl();
7  Naming.bind("MonAlerteTextuelle", alerte_textuelle_impl);
```

FIG. 4.2: RMI - Enregistrement d'un service

la résolution du nom logique du serveur représenté par une chaîne de caractères (ligne 6). Elle est donc limitée en terme d'expressivité et est sujette aux erreurs de saisie. Notamment, une recherche sur les attributs n'est pas possible et si le nom d'un service est mal orthographié, la découverte échouera à l'exécution. Le type de la référence retournée doit ensuite être converti (ligne 9) pour invoquer la méthode `afficherAlerte` (ligne 12) sur l'objet distant.

```
1  /* (1) Connexion au courtier de services */
2  Registry registry = LocateRegistry.getRegistry("192.168.0.12", "1099");
3
4  /* (2) Spécification du patron des services recherchés */
5  /* (3) Envoi de la requête de découverte de services au courtier de services */
6  Object o = registry.lookup("MonAlerteTextuelle");
7
8  /* (4) Conversion de type de la référence retournée */
9  AlerteTextuelle alerte_textuel_obj = (AlerteTextuelle) o;
10
11 /* (5) Invocation du service */
12 alerte_textuel_obj.afficherAlerte(2, "Intrusion détectée !!");
```

FIG. 4.3: RMI - Résolution de nom

4.1.2 CORBA

CORBA (Common Object Request Broker Architecture) est un des premiers intergiciels standardisés à s'attaquer aux problèmes d'interopérabilité dans les systèmes distribués [OHE97]. CORBA est un intergiciel à objets répartis dont l'IDL permet de définir les interfaces des services et de générer du support pour la communication entre services. Par exemple, dans la figure 4.4, l'interface du service `AlerteTextuelle` contient les attributs `propriétaire`, `résolution_largeur` et `résolution_hauteur` ainsi que la signature de l'opération `afficherAlerte` qui servira à afficher un message d'alerte sur un PDA. CORBA définit les liens entre le langage IDL et plusieurs langages de programmation. L'interface de programmation est alors unique pour un langage de programmation donné. Annoncée il y a une dizaine d'années comme la nouvelle technologie pour le commerce électronique, CORBA a depuis perdu ce statut au bénéfice des technologies Java et services Web [Hen06]. Cette désaffection s'explique surtout par la complexité de son canevas de programmation, ses choix architecturaux et par l'imprécision de sa spécification rendant ses nombreuses implémentations non interopérables.

```
1  interface AlerteTextuelle {
2    readonly attribute string propriétaire;
3    readonly attribute unsigned long résolution_largeur;
4    readonly attribute unsigned long résolution_hauteur;
5    void afficherAlerte(in short priorité, in string message);
6  }
```

FIG. 4.4: IDL CORBA - Déclaration d'un service AlerteTextuelle

4.1.2.1 Découverte de services

Serveur. Les services s'enregistrent sur le courtier de services en spécifiant un objet mandataire, une chaîne de caractères représentant le type du service et des couples attribut-valeur ; le nom de l'attribut est spécifié avec une chaîne de caractères et sa valeur est typée. La programmation de l'enregistrement est illustrée dans la figure 4.5. Une API générique permet de spécifier les attributs et la requête d'enregistrement. Ainsi, plusieurs erreurs peuvent survenir durant le développement sans qu'aucune vérification statique ne puisse être effectuée : les noms des attributs ainsi que le type de services peuvent être mal orthographiés (ligne 9 : « `propriétaire` », ligne 14 : « `résolution_largeur` », ligne 24 : « `AlerteTextuelle` ») et les attributs spécifiés peuvent ne pas correspondre aux attributs de la déclaration d'interface du service correspondant.

```
1  ...
2  /* (1) Connexion au courtier de services */
3  org.omg.CORBA.Object trader = orb.resolve_initial_references("TradingService");
4  org.omg.CosTrading.Lookup lookup  = org.omg.CosTrading.LookupHelper.narrow(trader);
5  org.omg.CosTrading.Register register = lookup.register_if();
6
7  /* (2) Spécification des propriétés du serveur */
8  org.omg.CosTrading.Property[] attributs = new org.omg.CosTrading.Property[3];
9  attributs[0] = new org.omg.CosTrading.Property();
10 attributs[0].name = "propriétaire";
11 attributs[0].value = orb.create_any();
12 attributs[0].value.insert_string("ALICE");
13
14 attributs[1] = new org.omg.CosTrading.Property();
15 attributs[1].name = "résolution_largeur";
16 attributs[1].value = orb.create_any();
17 attributs[1].value.insert_long(640);
18
19 ...
20
21 AlerteTextuelle_Impl alerte_textuelle_impl = new AlerteTextuelle_Impl();
22 AlerteTextuelle alerte_textuelle = alerte_textuelle_impl._this(orb);
23
24 /* (3) Envoi de la requête d'enregistrement au courtier de services */
25 String id = register.export(alerte_textuel, "AlerteTextuelle", attributs);
```

FIG. 4.5: CORBA - Enregistrement d'un service

Client. La programmation d'une découverte de services est illustrée dans la figure 4.6. De même que pour l'enregistrement, l'utilisation d'une API générique oblige le développeur à

manipuler des chaînes de caractères pour spécifier le type de services (ligne 8) et les contraintes sur les attributs (ligne 9). En plus des égalités, les contraintes permises sont des relations d'ordre sur les entiers ; par exemple, ligne 9, l'attribut `résolution_largeur` doit être supérieur à 600. L'appel à `narrow()`, qui convertit le type de l'objet retourné, n'est pas vérifié statiquement. Cette conversion s'appuie sur une classe `AlerteTextuelleHelper` générée depuis la déclaration d'interface du service (lignes 20 et 21). Enfin, le service est invoqué à la ligne 23.

```
1  /* (1) Connexion au courtier de services */
2  ...
3
4  /* (2) Spécification du patron des services recherchés */
5  /* (3) Envoi de la requête de découverte de services au courtier de services */
6  org.omg.CosTrading.OfferSeqHolder services = new org.omg.CosTrading.OfferSeqHolder();
7  trader.query(
8    "AlerteTextuelle",       // type du service
9    "propriétaire = 'ALICE' and résolution_largeur > 600",  // contraintes
10   ...,
11   1,              // nombre de services voulu
12   services,         // services retournés
13   ...
14 );
15
16 org.omg.CosTrading.Offer[] service = services.value;
17 ...
18 /* (4) Conversion de type de la référence retournée */
19 AlerteTextuelle alerte_textuel_obj
20    = AlerteTextuelleHelper.narrow(service[0].reference);
21
22 /* (5) Invocation du service */
23 alerte_textuel_obj.afficherAlerte(2, "Intrusion détectée !!");
```

FIG. 4.6: CORBA - Découverte d'un service

4.1.2.2 Modèle événementiel

CORBA propose une version non typée et une version typée du modèle événementiel. Dans la version non typée du modèle événementiel CORBA, le producteur publie un évènement en invoquant une opération sur un objet du courtier d'événements ou *canal d'événements*, comme décrit dans le chapitre 3. Le canal d'événements invoque ensuite une opération similaire sur chaque consommateur connecté au canal. Dans sa version typée, le producteur utilise un mandataire qui implémente la déclaration d'interface du consommateur définie par le développeur. Par exemple, la déclaration d'interface `GestionnaireDeLumières` de la figure 4.7 étend la déclaration d'interface `LuminositéPusher` qui définit une opération typée pour la réception des événements (ligne 5 : `push_luminosité`). La déclaration d'interface `LuminositéPusher` étend la déclaration d'interface `PushConsumer` fournie par CORBA pour déclarer son statut de consommateur d'événements. De façon similaire, la déclaration d'interface du producteur d'événements implémente la déclaration d'interface `PushSupplier` fournie par CORBA. Les déclarations d'interfaces `PushConsumer` et `PushSupplier` contiennent une opération de déconnexion du canal. À partir de la déclaration d'interface du consommateur, le compilateur IDL génère le mandataire du consommateur utilisé par le producteur et le canal pour transmettre des événements typés vers respectivement

le canal et les consommateurs.

Client. La programmation d'un consommateur est illustrée dans la figure 4.8. Les trois premières étapes utilisent des structures de code générique fournies par CORBA. Ainsi, la spécification du patron d'événements est effectuée avec la méthode générique `obtain_typed_push_supplier` et la chaîne de caractères « `IDL :LuminositéPusher :1.0` », ne permettant pas de vérifier statiquement le sujet de la souscription (lignes 9 et 10). La souscription est effectuée via la méthode générique `connect_push_consumer()` qui prend en argument le type générique `PushConsumer` (ligne 13) ; l'API CORBA permet donc à un consommateur d'événements de souscrire à des événements qu'il n'est pas capable de recevoir. Au contraire, la réception des événements s'effectue par une méthode typée `push_luminosité` et évite une conversion de types potentiellement erronée (ligne 17).

```
1  #include <omg/CosEventComm.idl>
2  #include <ConfigurationFeu.idl>
3
4  interface LuminositéPusher : PushConsumer {
5     void push_luminosité(in long valeur, in boolean extérieur);
6  };
7
8  interface GestionnaireDeLumières :
9     LuminositéPusher, ConfigurationFeu { };
```

FIG. 4.7: IDL CORBA - Déclaration d'un consommateur d'événements de luminosité

```
1  ... {
2     /* (1) Connexion au courtier d'événements */
3     Object obj = orb.resolve_initial_references("EventService");
4     TypedEventChannelFactory m_factory = TypedEventChannelFactoryHelper.narrow(obj);
5     TypedEventChannel tec = m_factory.create_typed_channel("TypedChannel", id);
6
7     /* (2) Spécification du patron d'événements */
8     TypedConsumerAdmin tca = tec.for_consumers();
9     ProxyPushSupplier pps
10       = tca.obtain_typed_push_supplier("IDL:LuminositéPusher:1.0");
11
12    /* (3) Envoie de la souscription au courtier d'événements */
13    pps.connect_push_consumer(this);
14 }
15
16 /* (4) Réception d'un événement */
17 void push_luminosité(in long valeur, in boolean extérieur) {
18    /* traitement de l'événement */
19 }
```

FIG. 4.8: CORBA - Programmation d'un consommateur d'événements de luminosité

Serveur. La programmation d'un producteur d'événements est illustrée dans la figure 4.9. Le sujet de la publication est spécifié par la chaîne de caractères « `IDL :Luminosité-Pusher :1.0` » (ligne 7). Si le producteur et le consommateur ne font pas référence à la même chaîne de caractères, une exception est levée. À partir de cette référence, le producteur

obtient un objet générique `obj` de la part du canal (lignes 6 à 12). À la ligne 13, la classe `LuminositéPusherHelper` générée depuis l'interface IDL du consommateur permet d'appeler l'opération `narrow` pour convertir l'objet `obj` en un mandataire typé du consommateur. Si le consommateur n'implémente pas l'interface `LuminositéPusher`, une exception est levée à l'exécution lors de l'appel à l'opération `narrow`. Enfin, l'envoi des événements s'effectue avec une méthode typée d'un mandataire dont la classe est générée depuis l'interface IDL du consommateur (ligne 14).

```
1  /* (1) Connexion au courtier d'événements */
2  Object obj = orb.resolve_initial_references("EventService");
3  m_factory = TypedEventChannelFactoryHelper.narrow(obj);
4  tec = m_factory.create_typed_channel("TypedChannel", id);
5
6  org.omg.CosTypedEventChannelAdmin.TypedSupplierAdmin tsa = tec.for_suppliers();
7  TypedProxyConsumer tpc = tsa.obtain_typed_push_consumer("IDL:LuminositéPusher:1.0");
8  tpc.connect_push_supplier(this);
9
10 /* (2) Création d'un événement */
11 /* (3) Envoi de l'événement vers le courtier d'événements */
12 Object obj = tpc.get_typed_consumer();
13 LuminositéPusher pusher = LuminositéPusherHelper.narrow(obj);
14 pusher.push_luminosité(1250, true);
```

FIG. 4.9: CORBA - Programmation d'un producteur d'événements de luminosité

4.1.3 Programmation des services Web

D'après le consortium W3C, les services Web sont des programmes informatiques qui permettent des interactions machine à machine dans des environnements distribués hétérogènes [W3C]. Les services Web diffèrent des technologies pour objets répartis comme CORBA [Vog03] ; les services Web utilisent des documents XML pour échanger des informations et n'imposent pas d'interface de programmation. Un service Web n'est pas obligatoirement un objet et peut être défini comme une entité logicielle traitant de documents XML normalisés.

Un service Web est décrit dans un langage XML appelé WSDL *(Web Services Description Language)*. La figure 4.10 définit le service d'alerte textuelle en WSDL. Les messages échangées sont spécifiés dans les balises `<message>` ; les lignes 5 à 8 définissent la requête de l'opération `afficherAlerte` qui est composée d'un entier pour la priorité et d'une chaîne de caractères pour le message d'alerte. Des messages plus complexes peuvent être définis à partir de schémas XML. Les opérations fournies par le service sont énumérées dans la balise `<portType>` ; les lignes 13 à 16 définissent la signature de l'opération `afficherAlerte`. Le protocole de transport et le format des messages des opérations sont spécifiés dans la balise `<binding>` ; les lignes 20 à 29 définissent le protocole HTTP pour le transport et le format SOAP *(Simple Objet Access Protocol)* pour le format des messages. Enfin, l'adresse du service est spécifiée dans la balise `port` (lignes 33 à 35). À l'instar des documents XML, les déclarations WSDL sont verbeuses et les développeurs utilisent des outils pour simplifier leur création. Par exemple, l'outil Java2WSDL génère des déclarations WSDL à partir du code Java [jav].

De façon similaire à CORBA, un compilateur WSDL génère du support de code pour l'invocation et l'implémentation des services ainsi que la gestion des messages échangées [jav]. Cette

approche est qualifiée de descendante *(top-down)*. À l'inverse, l'approche montante *(bottom-up)* permet de générer la déclaration WSDL à partir de l'implémentation, comme introduit précédemment. Les spécifications des services Web n'imposent pas de contraintes sur le code à générer et chaque implémentation est libre de fournir son propre compilateur. Les services Web sont ainsi indépendants de tous langages et systèmes. L'utilisation du protocole HTTP et de protocoles fondés sur XML permet de réaliser cette indépendance. Il existe une multitude d'extensions aux services Web qui permettent de traiter des aspects non fonctionnels (par exemple l'assemblage [IBMa, KBR05], la sécurité [OAS, NGGH07]) ou fonctionnels (par exemple la gestion des événements [ws-b, ws-a]). Cependant, ces spécifications sont parfois redondantes. Par exemple, WS-Eventing [ws-b] et WS-Notification [ws-a] constituent deux solutions concurrentes pour la gestion des événements dans les services Web.

Contrairement à RMI ou CORBA, les services Web ne spécifient pas d'interface de programmation. Les spécifications sont ainsi implémentées par divers API, essentiellement en C# et Java. Les interfaces de programmation ont des niveaux d'abstraction qui varient beaucoup d'une implémentation à une autre ; le développeur a généralement le choix entre une interface de programmation bas niveau où il devra directement manipuler les documents XML, souvent complexes, et une interface de programmation plus haut niveau qui abstrait la complexité de ces documents. Par ailleurs, ces interfaces de programmation haut niveau permettent d'être plus concis tout en ayant un vocabulaire plus proche de l'objectif fonctionnel, par exemple la publication d'événements. Cependant, ces abstractions réduisent l'expressivité, si bien que plusieurs interfaces de programmation sont nécessaires pour couvrir tous les besoins d'une application ubiquitaire. À travers les mécanismes de découverte de services et de notification d'événements, nous examinons les particularités des services Web en terme de programmation.

4.1.3.1 Découverte de services

La spécification UDDI *(Universal Description, Discovery, and Integration)* standardise l'enregistrement et la découverte des services Web [Bel02]. UDDI se focalise sur la découverte de partenaires commerciaux dans le but de réaliser des transactions. Cependant, le mécanisme UDDI et son modèle de description sont assez génériques pour s'appliquer aux systèmes distribués en général et aux systèmes ubiquitaires en particulier [KK04]. Le serveur d'enregistrement UDDI permet trois niveaux de recherche : les pages blanches qui contiennent des informations sur les entreprises (par exemple le nom, l'adresse, le contact), les pages jaunes qui permettent une catégorisation des services Web et les pages vertes qui décrivent le comportement et les fonctions d'un service Web. La spécification UDDI contient un ensemble de schémas XML qui définissent la représentation des services enregistrés, les opérations d'enregistrement et de découverte et la représentation des messages associés.

Pour interagir avec le serveur d'enregistrement UDDI, le développeur a le choix entre (1) une interface de programmation bas niveau pour SOAP, comme Axis [jav], qui demande une expertise dans la spécification UDDI et oblige à spécifier manuellement des messages XML potentiellement très complexes et (2) une interface plus haut niveau, comme UDDI4J [udd], conçue spécifiquement pour la spécification UDDI. Nous utiliserons UDDI4J, une implémentation Java de UDDI, dans nos exemples.

Serveur. La figure 4.11 contient le code nécessaire à l'enregistrement du service d'alerte textuelle. Bien que les schémas XML de la spécification UDDI définissent des contraintes sur les types, l'interface de programmation utilise des structures de code génériques. Par exemple, à

```
1  <?xml version="1.0"?>
2  <definitions name="AlerteTextuelleService"
3     targetNamespace="urn:AlerteTextuelle">
4
5     <message name="afficherAlerteRequete">
6        <part name="priorité" type="xsd:int" />
7        <part name="message" type="xsd:string" />
8     </message>
9
10    <message name="afficherAlerteReponse" />
11
12    <portType name="AlerteTextuelleInterface">
13       <operation name="afficherAlerte">
14          <input message="tns:afficherAlerteRequete" />
15          <output message="tns:afficherAlerteReponse" />
16       </operation>
17    </portType>
18
19    <binding name="AlerteTextuelleBinding" type="tns:AlerteTextuelleInterface">
20       <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
21       <operation name="afficherAlerte">
22          <soap:operation soapAction="urn:AlerteTextuelleAction" />
23          <input>
24             <soap:body use="literal"
25                    namespace="urn:AlerteTextuelle"
26                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
27          </input>
28          <output/>
29       </operation>
30    </binding>
31
32    <service name="AlerteTextuelleService">
33       <port name="AlerteTextuellePort" binding="tns:AlerteTextuelleBinding">
34          <soap:address location="http://phoenix.labri.fr:8080/alerteTextuelle" />
35       </port>
36    </service>
37
38 </definitions>
```

FIG. 4.10: Déclaration d'un service AlerteTextuelle avec WSDL (AlerteTextuelle.wsdl)

la ligne 65, l'enregistrement des services est effectué par la méthode `save_service(String authInfo, Vector businessServices)` où les deux arguments, bien qu'ayant des types contraints dans la spécification, ont des types génériques dans l'API. Au sein du document XML décrivant les services enregistrés, les attributs sont contenus dans l'élément `<categoryBag>` définissant les catégories. Ces catégories sont définies dans des taxonomies, listant les valeurs possibles. Les attributs sont pourtant définis avec des chaînes de caractères (par exemple ligne 24, la valeur de l'attribut résolution est « `640x480` »), rendant impossible toutes vérifications statiques par rapport à ces taxonomies. Il est donc indispensable pour le développeur de se référer en permanence aux documents XML. En outre, l'enregistrement d'un service nécessite de faire référence à des ressources extérieures, c'est-à-dire la description WSDL du service (ligne 55). Finalement, les nombreuses lignes de codes nécessaires pour l'enregistrement s'expliquent par la complexité des documents XML sous-jacents qui n'a pas été totalement abstraite par l'interface de programmation.

```
/* (1) Connexion au courtier de services */
UDDIProxy proxy = new UDDIProxy();
proxy.setPublishURL("http://phoenix.labri.fr/publishapi");

/* (2) Spécification des propriétés du serveur */
TModel tModel = new TModel();
tModel.setName("Alerte Textuel Interface");

OverviewDoc odoc = new OverviewDoc();
odoc.setOverviewURL("http://localhost/AlerteTextuelleInterface.wsdl");
tModel.setOverviewDoc(odoc);

CategoryBag cbag = new CategoryBag();
Vector  krVector = new Vector();
KeyedReference kr = new KeyedReference();
kr.setTModelKey("uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4");
kr.setKeyName("uddi-org:types");
kr.setKeyValue("wsdlSpec");
krVector.add(kr);
kr.setKeyName("propriétaire");
kr.setKeyValue("ALICE");
krVector.add(kr);
kr.setKeyName("résolution");
kr.setKeyValue("640x480");
krVector.add(kr);
cbag.setKeyedReferenceVector(krVector);

Vector tModelsVector = new Vector();
tModelsVector.add(tModel);
TModelDetail detail = proxy.save_tModel("", tModelsVector);

tModel = (TModel)detail.getTModelVector().elementAt(0);
String tModelKey = tModel.getTModelKey();

BusinessService service = new BusinessService();
service.setBusinessKey("uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40");
service.setDefaultName(new Name("AlerteTextuelleService"));

BindingTemplates templates = new BindingTemplates();
BindingTemplate template = new BindingTemplate();
templates.getBindingTemplateVector().add(template);
service.setBindingTemplates(templates);

AccessPoint accessPoint = new AccessPoint();
accessPoint.setURLType("HTTP");
accessPoint.setText("http://phoenix.labri.fr:8080/alerteTextuelle");
template.setAccessPoint(accessPoint);

TModelInstanceDetails details = new TModelInstanceDetails();
TModelInstanceInfo instance = new TModelInstanceInfo();
instance.setTModelKey(tModelKey);

InstanceDetails instanceDetails = new InstanceDetails();
OverviewDoc odoc2 = new OverviewDoc();
odoc2.setOverviewURL("http:////phoenix.labri.fr/AlerteTextuelle.wsdl");
instanceDetails.setOverviewDoc(odoc2);
instance.setInstanceDetails(instanceDetails);
details.getTModelInstanceInfoVector().add(instance);
template.setTModelInstanceDetails(details);

Vector services = new Vector();
services.add(service);

/* (3) Envoi de la requête d'enregistrement au courtier de services */
proxy.save_service("", services);
```

FIG. 4.11: Services Web (UDDI4J) - Enregistrement d'un service

Client. La découverte et l'invocation d'un service sont illustrées dans la figure 4.12. À l'instar de l'enregistrement, les structures de code sont génériques. Par exemple, l'envoi de la requête de découverte de services au serveur d'enregistrement UDDI est effectué à la ligne 23 par la méthode `find_service` qui prend comme arguments la chaîne de caractères `businessKey` définissant l'identifiant de l'entreprise et un vecteur de chaînes de caractères définissant les types de services recherchés. La méthode `find_service` renvoie une liste de services dont est extrait l'adresse de la description WSDL du premier service découvert (lignes 33 à 51). À partir de cette adresse WSDL, un mandataire du service est instancié en utilisant l'API JAX-RPC [jax] (lignes 54 à 62). JAX-RPC fournit un compilateur WSDL pour générer les mandataires et une API pour instancier ces mandataires à partir d'adresses WSDL. Comme illustré dans l'exemple, l'instanciation d'un mandataire requiert de manipuler des chaînes de caractères (lignes 55, 56 et 61) qui spécifient les données XML de la description WSDL du service. Cette instanciation demande aussi une conversion de type à la ligne 62. Le service est finalement invoqué grâce à une méthode typée (ligne 67).

4.1.3.2 Modèle événementiel

La spécification WS-Notification définit une approche pour introduire le modèle événementiel dans les services Web. Elle définit les fonctionnalités et messages que doivent gérer les fournisseurs d'événements, les courtiers d'événements, les souscripteurs et les consommateurs. Les figures 4.13, 4.14 et 4.15 illustrent respectivement la programmation de la publication, souscription et consommation d'événements. Le code développé utilise l'implémentation Apache Muse [mus].

Serveur. Le producteur utilise des structures de code génériques pour construire et publier les événements (Figure 4.13). Le sujet de l'événement ainsi que son contenu sont spécifiés avec des chaînes de caractères (ligne 9 : « `Luminosité` » et ligne 10 : « `1250` »). L'événement est envoyé en appelant la méthode `publish` de l'objet `producteur` créé lors de la connexion au courtier d'événements (ligne 13). La spécification et la publication des événements requièrent l'utilisation d'une bibliothèque de code gérant les structures XML (ligne 6 : `QName`, ligne 10 : `Element`).

Client. La programmation du souscripteur est présentée dans la figure 4.14. La création de l'objet `souscripteur` à partir de l'adresse du courtier d'événements permet de se connecter au courtier d'événements (lignes 2 à 6). Le sujet « `Luminosité` » et l'adresse du souscripteur sont ensuite envoyés au courtier d'événements en utilisant la méthode `subscribe` de l'objet `consommateur` (ligne 12). Un objet est finalement créé pour recevoir les événements correspondants (lignes 14 à 17). La classe de cet objet est définie dans la figure 4.15. Pour extraire les informations sur l'événement reçu, le développeur doit manipuler la structure XML du message correspondant à l'événement (lignes 6 à 8).

4.1.4 Programmation du protocole SIP

SIP *(Session Initiation Protocol)* est un protocole de communication ouvert et standardisé pour la téléphonie sur IP [Ros02]. Il permet l'établissement et la gestion de sessions multimédia entre des terminaux téléphoniques. SIP gère la mobilité des terminaux grâce à un service de nommage. En tant que protocole de signalisation, SIP n'impose pas de protocole

```
/* (1) Connexion au courtier de services */
UDDIProxy proxy = new UDDIProxy();
proxy.setInquiryURL("http://phoenix.labri.fr/inquiryapi");

/* (2) Spécification du patron des services recherchés */
String businessKey = ...;

Vector serviceNames = new Vector();
serviceNames.add("AlerteTextuelleService");

CategoryBag cbag = new CategoryBag();
Vector  krVector = new Vector();
KeyedReference kr = new KeyedReference();
kr.setKeyName("propriétaire");
kr.setKeyValue("ALICE");
krVector.add(kr);
kr.setKeyName("résolution");
kr.setKeyValue("640x480");
krVector.add(kr);
cbag.setKeyedReferenceVector(krVector);

/* (3) Envoi de la requête de découverte de services au courtier de services */
ServiceList list = proxy.find_service(
                    businessKey, // fournisseur du service
                    serviceNames, // type de service
                     cbag, // attributs
                     ...,
                     1 // nombre de services
);

/* (4) Conversion de type de la référence retournée */
/* (4a) Extraction de l'adresse WSDL de la référence retournée */
ServiceInfos infos = list.getServiceInfos();

ServiceInfo info = (ServiceInfo)infos.getServiceInfoVector().elementAt(0);
String serviceKey = info.getServiceKey();

ServiceDetail detail = proxy.get_serviceDetail(serviceKey);
BusinessService service;
service = (BusinessService) detail.getBusinessServiceVector().elementAt(0);

BindingTemplate template;
template = (BindingTemplate) service.getBindingTemplates()
                                    .getBindingTemplateVector().elementAt(0);
TModelInstanceDetails details = template.getTModelInstanceDetails();
TModelInstanceInfo instance;
instance = (TModelInstanceInfo) details.getTModelInstanceInfoVector().elementAt(0);
InstanceDetails instanceDetails  = instance.getInstanceDetails();

OverviewDoc odoc = instanceDetails.getOverviewDoc();
String wsdlpath = odoc.getOverviewURLString();

/* (4b) Création du mandataire */
QName serviceName = new QName(
                    "urn:AlerteTextuelle",
                    "AlerteTextuelleService");
URL wsdlLocation = new URL(wsdlpath);
ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(wsdlLocation, serviceName);

QName portName = new QName("", "AlerteTextuellePort");
AlerteTextuelleInterface alerte_textuel = (AlerteTextuelleInterface) service.getPort(
                    portName,
                    AlerteTextuelleInterface.class);

/* (5) Invocation du service */
alerte_textuel.afficherAlerte(2, "Intrusion détectée !!");
```

FIG. 4.12: Services Web (UDDI4J et JAX-RPC) - Découverte et invocation

```
1  /* (1) Connexion au courtier d'événements */
2  NotificationProducer producteur =
3     (NotificationProducer) getResource().getCapability(WsnConstants.PRODUCER_URI);
4
5  /* (2) Création d'un événement */
6  QName topicName = new QName("http://phoenix.labri.fr/ubiquitaire", "Luminosité");
7  wsn.addTopic(topicName);
8
9  QName messageName = new QName("http://phoenix.labri.fr/ubiquitaire", "valeur");
10 Element payload = XmlUtils.createElement(messageName, "1250");
11
12 /* (3) Envoi de l'événement vers le courtier d'événements */
13 producteur.publish(topicName, payload);
```

FIG. 4.13: Services Web (Apache Muse) - Publication d'événements

```
1  /* (1) Connexion au courtier d'événements */
2  URI address = URI.create("http://phoenix.labri.fr:8080/monCourtier");
3  EndpointReference courtierAdresse = new EndpointReference(address);
4  EndpointReference souscripteurAdresse = ...;
5  NotificationProducerClient souscripteur
6     = new NotificationProducerClient(courtierAdresse, souscripteurAdresse);
7
8  /* (2) Spécification du patron d'événements */
9  QName sujet = new QName("http://phoenix.labri.fr/ubiquitaire", "Luminosité");
10
11 /* (3) Envoie de la souscription au courtier d'événements */
12 souscripteur.subscribe(souscripteurEPR, new TopicFilter(sujet), null);
13
14 MonConsommateur consommateur = new MonConsommateur(topicName);
15 NotificationConsumer consommateurCap
16    = (NotificationConsumer)getResource().getCapability(WsnConstants.CONSUMER_URI);
17 consommateurCap.addMessageListener(consommateur);
```

FIG. 4.14: Services Web (Apache Muse) - Souscription d'événements

```
1  private class MonConsommateur implements NotificationMessageListener {
2     ...
3
4     /* (4) Réception d'un événement */
5     public void process(NotificationMessage message) {
6        Element event = message.getMessageContent(WefConstants.MGMT_EVENT_QNAME);
7        /* (5) Conversion de type de l'événement reçu */
8        String valeur = XmlUtils.toString(event);
9     }
10 }
```

FIG. 4.15: Services Web (Apache Muse) - Réception d'événements

pour l'échange des flux de données. Les extensions au protocole SIP incluent un modèle événementiel [Roa02, Nie04] et un modèle de communication synchrone utilisé notamment pour la messagerie instantanée [CRS+02]. SIP n'impose pas non plus de protocoles pour les données échangées dans le corps des messages. Ses divers modes de communication, sa flexibilité et sa capacité à gérer la mobilité des utilisateurs ont permis à SIP d'élargir ses domaines applicatifs

aux objets communicants [BSSW03, JPCK08].

La programmation d'un protocole est effectuée grâce à un support protocolaire qui réalise l'analyse et la construction des messages du protocole comme définies dans le standard. Le support de programmation des piles de protocoles est généralement très bas niveau. Pour fournir des abstractions plus hauts niveaux, des interfaces de programmation proposent d'ajouter une surcouche logicielle au support du protocole SIP [jaia, Jav03, jaib, osi]. Ces interfaces facilitent certaines tâches comme la création des messages.

Par exemple, l'API JAIN SIP [jaia] *(Java Advanced Intelligent Network SIP)* permet aux développeurs Java de gérer les messages SIP en s'appuyant sur des interfaces standardisées et un modèle de programmation à événements. Un service SIP utilisant JAIN SIP implémente une interface qui contient un ensemble de méthodes appelées par la pile de protocole pour le traitement des requêtes et réponses. L'API JAIN SIP 1.2 contient 102 classes Java et plus de 500 méthodes. Par ailleurs, l'implémentation de référence comporte plus de 300 classes pour environ 2 600 méthodes [Pal08]. En plus d'exiger une expertise dans le protocole SIP, l'API JAIN SIP demande un temps d'apprentissage non négligeable. Il existe des API plus haut niveau que JAIN SIP comme SIP servlet [Jav03]. Cependant, étant dédiés au routage des appels, elles ne sont pas adaptées à la programmation des applications ubiquitaires. À travers les mécanismes de découverte de services et de notification d'événements, nous discutons des particularités de l'API JAIN SIP.

4.1.4.1 Localisation de services

SIP propose d'abstraire la mobilité des utilisateurs. Pour cela, le terminal d'un utilisateur enregistre sur le courtier de services son adresse physique et un nom symbolique appelé URI SIP *(Uniform Resource Identifier)* représentant l'utilisateur. L'URI SIP est une adresse email (par exemple `alice@phoenix.fr`) utilisée par les contacts de l'utilisateur pour le contacter. L'utilisateur peut ainsi changer de terminal sans affecter ses contacts. Pour les systèmes distribués, l'URI abstrait la localisation des services. À l'instar de RMI, SIP ne propose pas de mécanisme de découverte de services.

Serveur. La figure 4.16 montre l'enregistrement du service d'alerte textuelle d'URI « `Alerte -Textuelle@inria.fr` » dans l'API JAIN SIP. L'enregistrement du service consiste à construire une requête SIP `REGISTER` (ligne 11) en spécifiant notamment l'adresse du courtier de services en tant que destinataire (ligne 9), l'adresse physique du service (lignes 23 et 24) et l'URI SIP correspondante (lignes 13 à 15). À la ligne 34, la requête est envoyée. La création d'une requête d'enregistrement avec JAIN SIP implique la manipulation d'un grand nombre de concepts (par exemple transaction et en tête) qui demande une expertise dans le protocole SIP. Par ailleurs, le développeur doit utiliser des chaînes de caractères pour spécifier les adresses physiques, les URI et le protocole de transport. JAIN SIP ne donne aucune garantie quant à la validité de la requête par rapport aux spécifications SIP. L'oubli d'un champs provoque ainsi une exception à l'exécution.

Client. Pour appeler les fonctionnalités d'un serveur, le client construit une requête similaire à celle utilisée pour l'enregistrement. Par exemple, pour inviter un serveur à échanger un flux de données audio, le service client crée une requête SIP `INVITE` en indiquant en destinataire l'URI SIP du service serveur à contacter. L'URI SIP est ensuite résolue par le courtier de services et la requête est envoyée au serveur. JAIN SIP ne fournit pas d'interface de programmation

```
SipAgent sa = ...; // étend javax.sip.SipListener
SipFactory sipFactory = SipFactory.getInstance();
sipFactory.setPathName("gov.nist");

HeaderFactory hf = sipFactory.createHeaderFactory();
AddressFactory af = sipFactory.createAddressFactory();
MessageFactory mf = sipFactory.createMessageFactory();

URI courtierAdresse = af.createSipURI(null, "courtier.inria.fr");
CallIdHeader callId = sa.getSipProvider().getNewCallId();
CSeqHeader cSeq = hf.createCSeqHeader(1L, Request.REGISTER);

SipUri entity
   = (SipUri) addressFactory.createSipURI("AlerteTextuelle", "inria.fr");
FromHeader from = hf.createFromHeader(af.createAddress(entity), Utils.generateTag());

ToHeader to = hf.createToHeader(af.createAddress(entity), null);
ArrayList<ViaHeader> viaList  = new ArrayList<ViaHeader>();
viaList.add(
   hf.createViaHeader("192.168.0.12", 5060, "udp", Utils.generateBranchId()));
MaxForwardsHeader maxForwards = hf.createMaxForwardsHeader(70);

ContactHeader contact = hf.createContactHeader(
   af.createAddress("sip:AlerteTextuelle@192.168.0.12"));
ExpiresHeader expires = hf.createExpiresHeader(600);

Request reg;
reg = mf.createRequest(
   courtierAdresse, Request.REGISTER, callId, cSeq, from, to, viaList, maxForwards);
reg.addHeader(contact);
reg.addHeader(expires);

/* (3) Envoi de la requête d'enregistrement au courtier de services */
sa.getSipProvider().getNewClientTransaction(reg).sendRequest();
```

FIG. 4.16: SIP (JAIN-SIP) - Enregistrement d'un service

pour la sérialisation et la désérialisation des événements. Le développeur doit alors opter pour une API externe comme Castor [cas].

4.1.4.2 Modèle événementiel

La programmation des événements dans SIP implique la création des requêtes `SUBSCRIBE`, `PUBLISH` et `NOTIFY`. La requête `SUBSCRIBE` est émise par le souscripteur et contient le type d'événements spécifié par une chaîne de caractères dans JAIN SIP. La requête `PUBLISH` est envoyée par le producteur d'événements vers le courtier d'événements qui envoie ensuite la requête `NOTIFY` à l'ensemble des souscripteurs. À la réception de la requête `NOTIFY`, la pile de protocole du souscripteur appelle une méthode de l'API JAIN SIP pour traiter l'événement.

4.1.5 Synthèse

L'analyse des interfaces de programmation précédentes nous permet de lister les lacunes concernant les canevas de programmation pour les systèmes distribués.

Manque d'abstraction. Les canevas de programmation n'abstraient que partiellement les technologies sous-jacentes. Dans le but de respecter au mieux les spécifications de ces techno-

logies, les canevas de programmation incluent des structures de code et du vocabulaire spécifique aux technologies sous-jacentes. Un haut niveau d'expertise dans ces technologies est alors nécessaire. Par ailleurs Les canevas de programmation manquent d'ambition. Certaines tâches répétitives et sujettes à erreurs pourraient être simplifiées. Cependant, les canevas de programmation proposés implémentent les spécifications des technologies associées sans offrir d'abstraction supplémentaire. Enfin, les canevas de programmation ne sont pas suffisants. Ils obligent les développeurs à se référer en permanence aux documents de spécification des applications et des technologies. En outre, ces documents doivent être référencés dans le code applicatif (par exemple pour la souscription dans CORBA et l'invocation dans les services Web). L'exigence de séparation entre conception et implémentation introduite par les langages IDL n'est donc pas respectée.

Manque de vérifications. La généricité des canevas de programmation implique que peu de vérifications statiques sont possibles. Les canevas de programmation favorisent l'utilisation de structures génériques de code, de chaînes de caractères et de conversions de type. De plus, ils n'assurent que partiellement la cohérence des implémentations par rapport aux déclarations d'interfaces des services. Par ailleurs, les canevas de programmation n'assurent pas la cohérence globale des applications. Les services sont considérés individuellement. Les compilateurs IDL ne considèrent pas un ensemble d'interfaces mais une seule interface à la fois. Par exemple, un type de services peut être déclaré comme souscripteur d'un type d'événements sans qu'aucun fournisseur correspondant n'existe.

Manque de portabilité du code applicatif. Les applications développées ne sont pas portables. Elles ne sont pas portables d'un bus logiciel à un autre à cause du manque d'abstraction des API vis-à-vis de la spécification. Elles ne sont pas non plus portables au sein même d'une technologie. Par exemple, les implémentations de la technologie CORBA ne sont pas interopérables [Hen06]. Les implémentations des spécifications services Web sont très variées et souvent redondantes. Le changement d'implémentation implique alors de réécrire tout ou parties des applications. De plus, pour la programmation d'une même application, les services Web et le protocole SIP nécessitent l'utilisation de plusieurs interfaces de programmation répondant à autant de préoccupations (par exemple transport des données, signalisation, invocation). L'hétérogénéité de ces interfaces complique l'apprentissage des canevas de programmation.

4.2 Programmation des systèmes ubiquitaires

La variété et la dynamicité des environnements ubiquitaires multiplient les paramètres à prendre en compte lors de la programmation des applications. Les intergiciels pour systèmes distribués sont adaptés à des environnements peu dynamiques. Ils ne fournissent donc ni les services systèmes ni le support de programmation adaptés à la gestion de changements constants. Les développeurs doivent alors combler le fossé existant par la programmation de tâches complexes et répétitives. Pour répondre à ces problèmes, les intergiciels pour systèmes ubiquitaires proposent d'augmenter le niveau d'abstraction des intergiciels pour systèmes distribués. Pour cela, ils fournissent de nouveaux services systèmes s'appuyant sur les services systèmes existants. Notamment, les services systèmes réflexifs sont proposés pour adapter les applications aux changements des environnements ubiquitaires sans programmation explicite

des développeurs. La plupart des intergiciels pour systèmes ubiquitaires ne propose pas de nouvelle solution pour la programmation des systèmes ubiquitaires. En revanche, ils proposent des langages à scripts [RHC+02, GSSS02] dont le niveau réduit d'expressivité ne permet que de faciliter la configuration des environnements ubiquitaires. D'autres approches proposent des boites à outils *(toolkits)* pour faciliter la programmation de certains aspects des applications ubiquitaires. Par exemple, la boite à outils Magic Broker aide à la programmation des interactions entre de grands écrans et des objets communicants mobiles comme des téléphones [EBF+08]. Dey *et al.* proposent une boite à outils pour la gestion du contexte [DAS01]. Les boîtes à outils fournissent un ensemble figé d'abstractions haut niveau spécialisées dans la programmation d'une tâche. La variété et la dynamicité des domaines applicatifs ubiquitaires exigent au contraire des solutions adaptables, c'est-à-dire capables d'intégrer un flot constant de nouveaux objets communicants et d'aider au développement d'applications dont les besoins ne cessent de varier. Par ailleurs, les API proposés sont génériques et des erreurs potentiellement critiques sont susceptibles de se produire à l'exécution.

Dans cette section, nous présentons deux intergiciels qui proposent des canevas de programmation pour le développement d'applications ubiquitaires. Plus particulièrement, nous analysons les abstractions fournies pour gérer la mobilité des utilisateurs dans les environnements ubiquitaires.

4.2.1 One.world

L'approche one.world propose un canevas de programmation pour la gestion de la dynamicité des environnements ubiquitaires [GDL+04]. Elle simplifie la programmation des tâches de découverte, de migration et de gestion des erreurs. Les communications entre les services sont exclusivement effectuées selon un modèle événementiel. Les événements sont spécifiés par une chaîne de caractères et un objet, c'est-à-dire `set(String name, Object value)`, et sont envoyés via une méthode générique, `handle(Event e)`. Ils sont reçus par la méthode générique `handle1(Event e)`.

L'approche one.world fournit des services systèmes pour gérer le changement (par exemple la migration, la sauvegarde et la reprise d'états des applications), la composition (par exemple la découverte de services et les communications asynchrones) et la persistance. Elle fournit des opérations pour sauvegarder, restaurer ou migrer les ressources de l'espace physique comme illustré dans la figure 4.17. La programmation de la migration d'une application dans one.world nécessite de spécifier l'adresse physique de l'objet communicant vers laquelle la migration doit s'effectuer (par exemple ligne 4, « `sio ://pda.inria.fr/alice` »).

```
1 ...
2 operation.handle(
3    new MoveRequest(null, utilisateur, utilisateur.env.getId(),
4       "sio://pda.inria.fr/alice", false)
5 );
```

FIG. 4.17: One.World - Migration de la ressource « env » d'un utilisateur sur l'objet pda

One.world fournit un canevas de programmation générique. Les structures génériques de code représentant l'environnement (par exemple `Event`) et l'emploi des chaînes de caractères (par exemple pour nommer les propriétés des services) rendent les applications vulnérables aux erreurs à l'exécution.

4.2.2 Gaia et Olympus

Gaia est un intergiciel réflexif qui aide au développement et à l'exécution d'applications ubiquitaires reconfigurables [RHC+02]. Elle repose sur le concept d'espace actif *(active space)*. Un espace actif abstrait un espace physique en le dotant d'une infrastructure logicielle sensible à la mobilité des utilisateurs. Gaia permet de migrer dynamiquement une application à travers différents environnements physiques.

Gaia étend CORBA pour gérer la dynamicité des environnements. Il fournit des services systèmes pour la gestion des événements, la persistance et la localisation des services. La gestion des événements étend le service d'événements de CORBA et permet de notifier les applications à propos des nouveaux services ou du déplacement des utilisateurs. Gaia maintient une liste de propriétés non fonctionnelles sur chacun des services de l'espace actif pour faciliter le déploiement ou le redéploiement des applications. Cette liste est maintenue et interrogée grâce au service de localisation de CORBA. Gaia utilise un langage à script appelé LuaOrb [CCI99] pour programmer et configurer les espaces actifs et pour coordonner les objets communicants correspondants. Le langage LuaOrb est peu expressif et interprété. Il ne peut donc satisfaire au développement d'applications ubiquitaires complexes et critiques.

Le canevas de programmation Olympus [RCAM+05] est une surcouche logiciel à Gaia. Olympus fournit des abstractions hauts niveaux pour la programmation des espaces actifs. Olympus utilise les ontologies pour concevoir une représentation indépendante des environnements physiques. Les ontologies permettent de définir des concepts, des relations entre ces concepts et des instances de ces concepts.

Dans Olympus, les ontologies décrivent les types de services, d'objets communicants, de personnes et l'espace physique. Elles spécifient aussi les relations d'héritage et de proximité sémantique entre ces types. À l'instar des services CORBA et des services Web qui se réfèrent aux déclarations d'interfaces, les applications Olympus expriment leurs besoins en se référant aux ontologies. À chaque changement d'environnement, ces besoins sont réévalués par rapport aux ressources disponibles et l'application est redéployée sur les objets communicants correspondants. Un algorithme se charge de trouver les objets et les services les plus appropriés par rapport aux informations disponibles dans les ontologies. Olympus fournit aux développeurs des opérations haut niveau et génériques pour modifier l'état des services et des applications dans les espaces actifs. Par exemple, le développeur peut activer ou désactiver les services avec les opérations `on` et `off`. Cependant, Olympus ne propose pas de solution pour ajouter de nouvelles opérations. La figure 4.18 illustre la découverte et l'invocation de l'opération `on` sur des lumières. La requête de découverte est spécifiée par des méthodes génériques, par exemple, `hasProp()` dont les arguments qui font références aux ontologies sont définis avec des chaînes de caractères. L'évaluation de la requête par rapport aux ontologies est effectuée à l'exécution, produisant potentiellement une erreur. De plus, les services sont manipulés avec le type générique `Device`, restreignant les opérations possibles à celles définies par défaut.

D'un point de vue programmation, la vraie contribution d'Olympus vient de ses opérations de déploiement et de migration qui permettent d'augmenter le niveau d'abstraction des requêtes de découverte de services. Olympus permet de suspendre l'exécution d'une application avec l'opération `suspend` et reprendre cette exécution dans un nouvel environnement avec l'opération `resume`. Par exemple, dans la figure 4.19, les applications d'Alice sont migrées dans l'espace actif dans lequel elle se trouve. L'opération `resume` masque alors un ensemble d'opérations de découverte de services qui se chargent de trouver les meilleurs services nécessaires aux applications.

```
1  DeviceList lumières;
2  lumières.hasProp("classe", "Lumière");
3  lumières.hasProp("localisation", "Etage1/Couloir/Nord");
4  lumières.instantiate();
5
6  for (Device lumière : lumières)
7     lumière.on();
```

FIG. 4.18: Olympus - Découverte et invocation de services (pseudo-code)

```
1  Person alice;
2  alice.hasProp("name", "ALICE");
3
4  ActiveSpace espaceActif;
5  espaceActif.hasProp("containsPerson", alice);
6
7  ApplicationList applications;
8  applications.hasProperty("propriétaire", alice);
9  applications.hasProperty("etat", "suspended");
10 applications.instantiate();
11
12 for (Application application : applications)
13    application.resume(espaceActif);
```

FIG. 4.19: Olympus - Migration d'applications (pseudo code)

4.2.3 Synthèse

Olympus et one.world proposent des solutions pour gérer la dynamicité des environnements ubiquitaires. Ils permettent notamment de programmer la migration des applications des utilisateurs en fournissant aux développeurs des interfaces de programmation haut niveau. Cependant, les canevas de programmation ne sont pas liés aux spécifications de l'environnement, quand elles existent, ce qui oblige les développeurs à manipuler des structures génériques de code. Les applications sont alors dynamiquement évaluées et des erreurs peuvent compromettre des applications ubiquitaires potentiellement critiques.

4.3 Programmation orientée composant

Dans cette section, nous étudions à travers deux intergiciels à composant, EJB 3 et CORBA CCM, la méthodologie de production des applications. À travers CORBA CCM et ArchJava, nous nous intéressons à la notion d'architecture logicielle et son impact sur la production des applications. Les architectures logicielles permettent, contrairement aux approches précédentes, de vérifier statiquement la validité des connexions entre services.

4.3.1 EJB 3 : Enterprise Java Beans

EJB 3 *(Enterprise Java Beans)* est une plate-forme Java pour le développement et le déploiement d'applications à base de composants [PRL07]. EJB est une technologie Java standardisée reposant sur une spécification publique de son interface de programmation. Elle est soutenue par un grand nombre d'entreprises et de communautés open source qui proposent

des implémentations concurrentes et interopérables de la spécification. Nous allons détailler la chaîne de production des composants illustrée dans la figure 4.20.

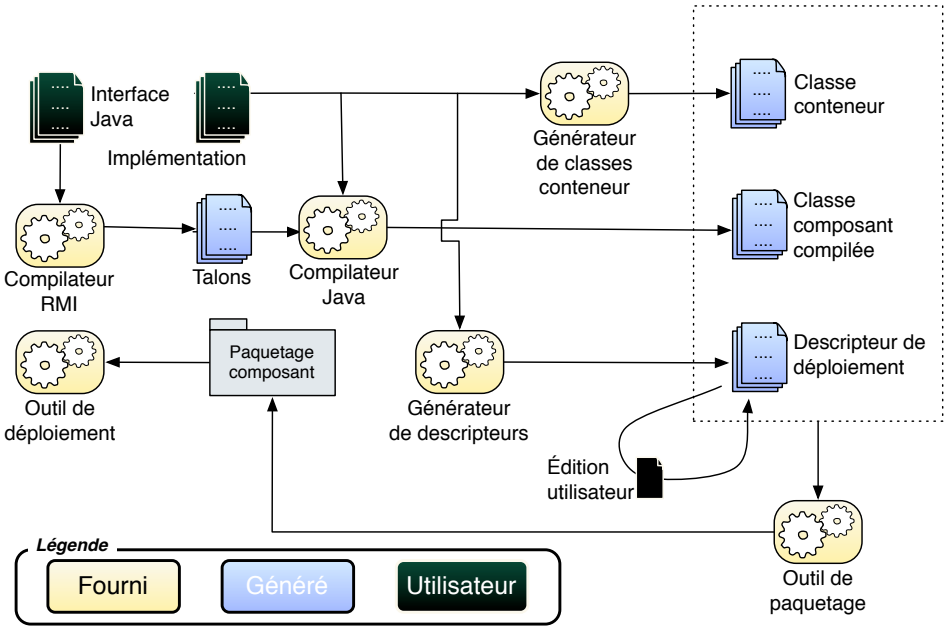


FIG. 4.20: La chaîne de production des composants EJB 3

4.3.1.1 Spécification et génération de code

Chaque composant EJB a une interface fonctionnelle Java. Elle permet également de déclarer par quel protocole de communication le composant peut être contacté (par exemple localement, via RMI ou via des services Web). Des talons clients et serveurs sont générés depuis les interfaces des composants EJB.

```
1 @Remote
2 public interface AlerteTextuelle {
3    public void afficherAlerte(int priorité, String message);
4 }
```

FIG. 4.21: Interface d'un composant EJB

4.3.1.2 Programmation des applications

Trois catégories de composants (ou *beans*) sont utilisés : les composants sessions (*session beans*), les composants messages (*message-driven beans*) et les composants entités (*entities*). Les composants sessions traitent des communications synchrones à état ou sans état. Les composants messages traitent des communications asynchrones en mode point à point ou suivant le modèle événementiel. Les composants entités assurent la persistance des données en fournissant une représentation objet des données stockées dans les bases de données.

L'interface de programmation d'EJB utilise les annotations introduites dans Java 5 pour simplifier le développement des applications. Les annotations permettent d'ajouter des informations sur les interfaces, classes, méthodes et variables. Elles rendent plus concises des tâches répétitives comme la découverte de services. Elles permettent aussi de remplacer les fichiers XML de configuration verbeux et non vérifiés statiquement. Par exemple, l'annotation `@stateless` d'une classe spécifie que le type de composant implémenté est un composant session sans état, évitant ainsi la création d'un fichier XML de 9 lignes comme illustré dans la figure 4.22. Un composant EJB est donc une classe Java avec des annotations.

```
<enterprise-beans>
   <session>
      <ejb-name>AlerteTextuelleBean</ejb-name>
      <local>fr.labri.phoenix.AlerteTextuelle</local>
      <ejb-class>fr.labri.phoenix.AlerteTextuelleBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
   </session>
<enterprise-beans>
```

```
@stateless
public class AlerteTextuelleBean implements AlerteTextuelle {
...
}
```

FIG. 4.22: Substitution des fichiers XML par des annotations pour le typage des composants

EJB fournit un ensemble complet d'interfaces de programmation, évitant aux développeurs la recherche fastidieuse d'outils externes comme pour les services Web. EJB intègre des solutions existantes comme JNDI [JND] *(Java Naming and Directory Interface)* pour la découverte de services, RMI pour les communications synchrones et JMS pour les communications asynchrones. En s'appuyant sur des technologies existantes, EJB hérite aussi de leurs faiblesses et notamment de leur incapacité à vérifier statiquement le code développé comme illustré dans la section 4.1. Les annotations utilisées dans EJB simplifient la lisibilité du code mais ne permettent pas davantage de vérifications.

Les annotations permettent de masquer les requêtes de localisation de ressources, comme par exemple lors d'une publication d'événements ou d'une découverte de services. Cependant, le développeur spécifie la ressource avec la même chaîne de caractères et une conversion de type, même si elle est masquée, est toujours effectuée à l'exécution. Par exemple, les figures 4.23 et 4.24 montrent la spécification du sujet `Luminosité` d'une publication, respectivement sans annotation et avec annotations. Enfin, les connexions entre composants sont effectuées manuellement dans le code applicatif.

Comme illustré dans la figure 4.20, un outil spécifique à la plate-forme EJB génère des classes conteneurs destinées à configurer les services systèmes du conteneur à l'exécution pour gérer, par exemple, le cycle de vie du composant.

4.3.1.3 Paquetage et déploiement

Pour simplifier le déploiement, EJB paquète l'implémentation du composant, c'est-à-dire les classes conteneur et les classes composant compilées. Pour cela, EJB suit la norme JAVA EE

```
1 InitialContext context = new InitialContext();
2 Topic topic = (Topic) context.lookup("Luminosité");
```

FIG. 4.23: EJB 3 - Déclaration d'une ressource sans annotation

```
1 @Resource(name="Luminosité")
2 private Topic topic;
```

FIG. 4.24: EJB 3 - Déclaration d'une ressource avec une annotation

(Java platform, Enterprise Edition) pour le déploiement. Les implémentations de composants sont contenues dans des modules EJB-JAR qui sont des archives Java JAR *(Java ARchive)* au format zip. Chaque module EJB-JAR contient un *descripteur de déploiement* qui est un fichier XML décrivant le contenu du module, les ressources utilisées et des contraintes sur le conteneur. Les descripteurs de déploiement sont en partie générés depuis le code applicatif et doivent être complétés manuellement. Le but est de séparer les préoccupations de développement et de déploiement. Le descripteur de déploiement écrase les annotations si nécessaire, chaque annotation ayant son équivalent XML dans le descripteur. Le composant ainsi paqueté est déployé dans un environnement d'exécution où il peut être utilisé dans l'application.

4.3.1.4 Exécution

De la même façon qu'un objet Java s'exécute dans une machine virtuel Java, les composants EJB sont exécutés dans des conteneurs EJB. Un conteneur EJB fournit des services aux composants comme la gestion du cycle de vie.

4.3.1.5 Évaluation

L'implémentation est simplifiée grâce aux conteneurs qui gèrent les aspects non fonctionnels des composants et permettent aux développeurs de se concentrer sur les aspects fonctionnels. À l'exécution, les conteneurs fournissent un ensemble figé de services systèmes aux instances de composants. Une approche générative est adoptée pour générer les talons et les classes spécialisant les conteneurs. Le support de programmation est identique à celui fourni par les technologies des systèmes distribués comme CORBA ou JMS. Par exemple, des structures génériques de code et des conversions de type sont employées pour la gestion des événements dans l'implémentation des composants. Les annotations Java permettent de séparer les déclarations des ressources du code applicatif et de donner des informations descriptives sur le composant comme son type. Les annotations ne fournissent pas de garantie supplémentaire quant à la fiabilité du code développé. En revanche, elles soulagent les développeurs en spécifiant des informations contenues habituellement dans des descripteurs XML. Les descripteurs décrivent le contenu des paquetages, les propriétés de configuration du conteneur et des composants. L'outil de déploiement utilise les descripteurs pour vérifier que les ressources nécessaires soient disponibles. Certains descripteurs ne sont pas générés et doivent donc être spécifiés par les utilisateurs qui utilisent généralement des outils graphiques permettant d'assurer la validité par rapport au schéma XML du descripteur. Cependant, comme ces descripteurs ne sont pas

liés au code applicatif, des erreurs à l'exécution peuvent se produire. Par exemple, l'erreur `ClassNotFoundException` est levée si une classe est manquante.

EJB ne permet pas de spécifier les interfaces requises ou la composition entre les composants. De plus, EJB ne fournit aucun outil ou méthodologie pour assembler les implémentations ou les instances de composants, la gestion des connexions restant à la charge des développeurs. La pauvreté du langage de spécification fragilise donc la chaîne de production des composants EJB.

4.3.2 CCM : CORBA Component Model

Le modèle à composant CORBA (CCM) est une extension de CORBA. Il spécifie un modèle abstrait de composants, un modèle de programmation et un modèle d'exécution. Les principales avancées par rapport à CORBA sont l'introduction de la notion d'architecture logicielle et la séparation entre la programmation des parties fonctionnelles et non fonctionnelles. Contrairement à EJB, CCM est indépendant du langage de programmation. CCM définit une méthodologie de production des applications où les composants CORBA passent par cinq étapes : spécification, implémentation, paquetage, assemblage et déploiement. L'étape de paquetage étant similaire à celle de EJB, nous ne détaillons pas ce point. Comme illustré dans la figure 4.25, l'implémentation et l'assemblage des composants requièrent de spécifier et de générer un ensemble de fichiers que nous détaillons. Pour la programmation des composants, nous nous appuyons sur l'implémentation openCCM. OpenCCM est une implémentation Java qui fonctionne avec la plupart des implémentations Java de CORBA.

4.3.2.1 Spécification et génération de code

Le modèle abstrait de composants définit comment déclarer les interfaces fonctionnelles d'un composant. Pour déclarer des *types de composants*, il introduit le langage IDL3 qui est une version enrichie du langage IDL CORBA (c'est-à-dire IDL2). Le langage IDL3 permet de spécifier des *ports* qui sont des fonctionnalités fournies ou requises d'un composant. CCM distingue quatre sortes de ports correspondant aux modes de communication synchrone (c'est-à-dire port client et port serveur) et asynchrone (c'est-à-dire port consommateur et port producteur). La figure 4.26 illustre un exemple de spécification IDL3 ; les types de composants `GestionnaireLumières` et `CapteurLuminosité` sont respectivement déclarés client et serveur d'une interface `LireLuminosité` et consommateur et fournisseur d'événements `Luminosité`. La spécification IDL3 définit aussi des *maisons de composants (home)* qui gèrent les aspects non fonctionnels des composants.

Un compilateur CCM génère la projection des interfaces IDL3 dans des interfaces IDL2. Un compilateur d'IDL CORBA génère ensuite les talons et les interfaces du langage cible. Pour l'implémentation des composants, il génère l'interface serveur de `CapteurLuminosité` (Figure 4.27) et l'interface consommateur de `GestionnaireLumières` (Figure 4.28). Il génère également des interfaces spécifiant les opérations de connexion entre les ports des composants (Figure 4.29). Un deuxième compilateur génère, à partir des interfaces IDL3, des classes conteneurs pour gérer, grâce aux maisons, les aspects non fonctionnels des composants, c'est-à-dire l'implémentation des opérations de connexion et la gestion du cycle de vie du composant.

La structure de l'implémentation d'un type de composant est spécifiée dans le langage CIDL. Le langage CIDL permet notamment de déclarer la catégorie d'un type de composants,

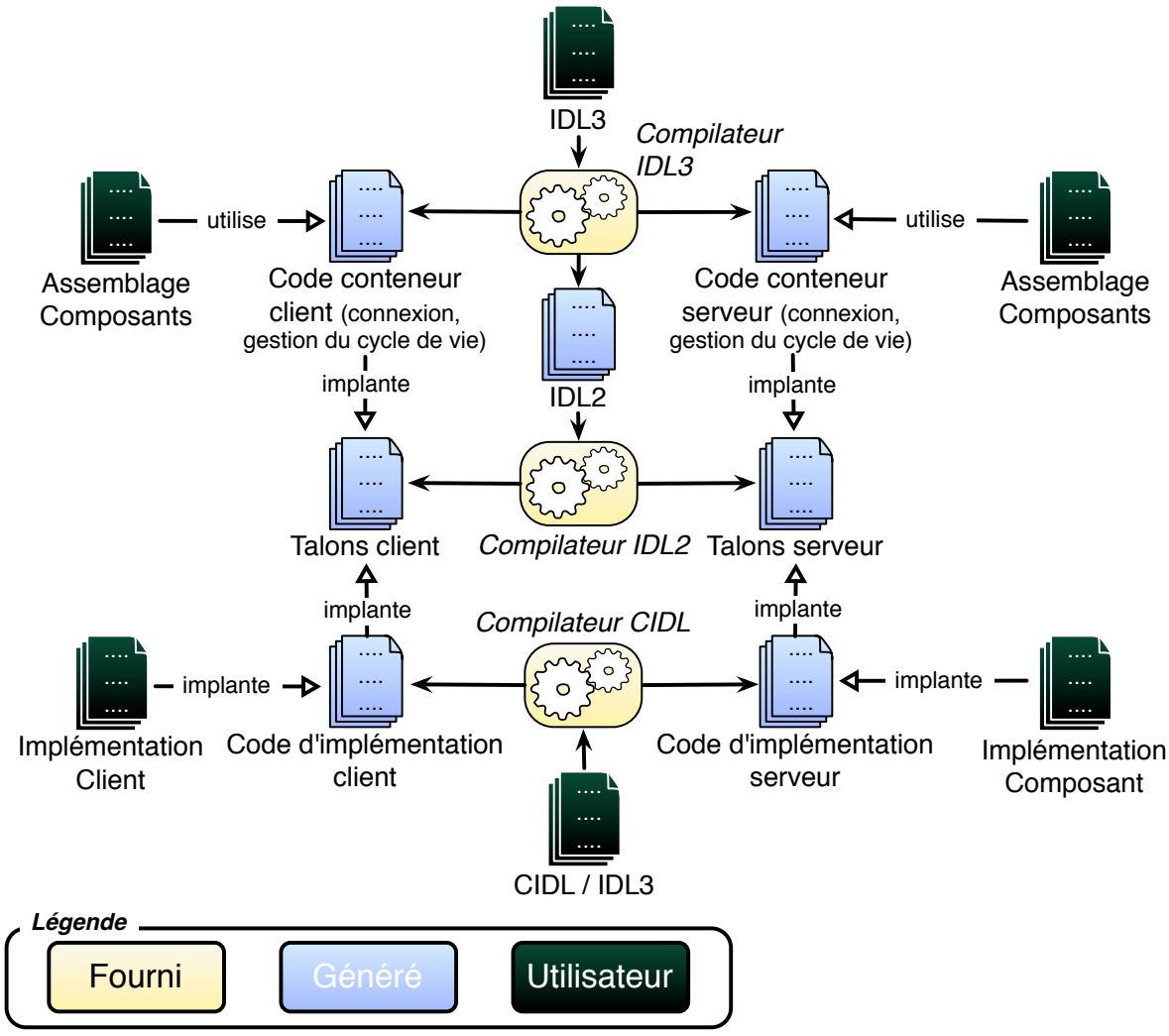


FIG. 4.25: Spécification, implémentation et assemblage des composants dans CCM

```
enum LuminositéValeur {
   NUIT, SOMBRE, JOUR
};

interface LireLuminosité {
   LuminositéValeur lireLuminosité();
};

eventtype Luminosité {
   public LuminositéValeur valeur;
   public boolean extérieur;
};

component GestionnaireLumières {
   uses LireLuminosité lireLuminosité;
   consumes Luminosité luminosité;
};
home GestionnaireLumièresMaison manages GestionnaireLumières {};

component CapteurLumière {
   provides LireLuminosité lireLuminosité;
   publishes Luminosité luminosité;
};
home CapteurLumièreMaison manages CapteurLumière {};
```

FIG. 4.26: IDL3 - Déclaration de types de composants

```
public interface LireLuminositéOperations {
   LuminositéValeur lireLuminosité();
}
```

FIG. 4.27: OpenCCM - Interface serveur d'un capteur de luminosité

```
public interface CCM_LuminositéConsumerOperations {
        void push(Luminosité event);
}
```

FIG. 4.28: OpenCCM - Interface d'un consommateur d'événements Luminosité

c'est-à-dire s'il est à état et/ou persistant. La figure 4.30 illustre la spécification CIDL des types `GestionnaireLumières` et `CapteurLuminosité` en tant que type `session`, c'est-à-dire capable d'effectuer des communications à état. Nous retiendrons que c'est à partir des spécifications CIDL que le code pour l'implémentation des composants est généré. Le code généré permet de lier la partie fonctionnelle (c'est-à-dire le type de composants) à la partie non fonctionnelle (c'est-à-dire la maison de composants).

4.3.2.2 Programmation des applications

Les classes générées depuis les spécifications permettent d'implémenter les composants et d'assembler les instances de composants pour créer l'application. Le canevas de programmation CCM simplifie la programmation des applications en séparant les préoccupations d'implémen-

```
1  public interface CapteurLuminositéOperations {
2          LireLuminosité provide_lireLuminosité();
3          Cookie subscribe_luminosité(LuminositéConsumer consumer);
4          LuminositéConsumer unsubscribe_luminosité(Cookie ck);
5  }
6
7  public interface GestionnaireLumièresOperations {
8          void connect_lireLuminosité(LireLuminosité connexion);
9          LireLuminosité disconnect_lireLuminosité();
10         LireLuminosité get_connection_lireLuminosité();
11
12         LuminositéConsumer get_consumer_luminosité();
13 }
```

FIG. 4.29: OpenCCM - Interfaces de connexion

```
1  composition session GestionnaireLumièresImpl {
2     home executor GestionnaireLumièresMaisonImpl {
3        implements GestionnaireLumièresMaison;
4        manages GestionnaireLumièresImpl;
5     };
6  };
7
8  composition session CapteurLuminositéImpl {
9     home executor CapteurLuminositéMaisonImpl {
10       implements CapteurLuminositéMaison;
11       manages CapteurLuminositéImpl;
12    };
13 };
```

FIG. 4.30: CIDL - Déclaration de l'implémentation des types de composants

tation des composants et les préocuppations d'assemblage des composants.

Implémentation. De façon similaire à CORBA, les serveurs et les consommateurs d'événements doivent implémenter les opérations définies dans les interfaces des figures 4.27 et 4.28 pour recevoir respectivement des invocations et des événements. L'invocation des serveurs et la publication d'événements sont programmées en utilisant des méthodes `get_context()` héritées des classes générées. Chaque type de composants a une méthode `get_context()` qui lui est spécifique et qui lui permet d'accéder uniquement aux ports déclarés dans la spécification. Dans la figure 4.31, la méthode `get_context()` du type de composants `Gestionnaire-Lumières` permet de récupérer une connexion sur l'interface `LireLuminosité` d'un composant de type `CapteurLuminosité`. L'invocation de la méthode `lireLuminosité` est ensuite effectuée sur le composant connecté. De même dans la figure 4.32, la publication d'un événement est effectuée par la méthode `get_context()` du type de composant `Capteur-Luminosité`. En plus d'aider les développeurs à utiliser correctement les ports spécifiés, les méthodes `get_context()` permettent aux développeurs de ne pas se préoccuper des connexions entre composants lors de leur implémentation.

Assemblage. La spécification de l'ensemble des connexions entre composants, c'est-à-dire l'assemblage des composants, est effectuée dans un programme dédié. L'assemblage est pré-

```
1 /* get_context() généré et hérité par le gestionnaire de lumières */
2 LireLuminosité capteurLuminosité = get_context().get_connection_lireLuminosité();
3 ...
4 LuminositéValeur valeur = capteurLuminosité.lireLuminosité();
```

FIG. 4.31: OpenCCM - Invocation d'un capteur de luminosité par un composant GestionnaireLumières

```
1 /* get_context() généré et hérité par le capteur de luminosité */
2 get_context().push_luminosité( new Luminosité(LuminositéValeur.JOUR, true) );
```

FIG. 4.32: OpenCCM - Publication d'un événement Luminosité par un composant CapteurLuminosité

cédé de la création ou de la découverte des conteneurs, des maisons et des instances de composants à assembler. Par exemple, dans la figure 4.33, des instances de composants de types `GestionnaireLumières` et `CapteurLuminosité` sont créées à partir des maisons correspondantes (lignes 1 à 5). La connexion entre le consommateur et le producteur d'événements `Luminosité` est effectuée grâce aux méthodes générées `get_consumer_luminosité` et `subscribe_luminosité` (lignes 7 et 8). L'assemblage des instances de composants peut aussi être spécifié dans un fichier XML. Des outils graphiques sont alors utilisés pour s'assurer du respect du document DTD *(Document Type Definition)* correspondant.

```
1 GestionnaireLumièresMaison glMaison = ...;
2 GestionnaireLumières gl = glMaison.create();
3
4 CapteurLumièreMaison cMaison = ...;
5 CapteurLumière c = cMaison.create();
6
7 LuminositéConsumer lConsumer= gl.get_consumer_luminosité();
8 c.subscribe_luminosité(lConsumer);
```

FIG. 4.33: OpenCCM - Programmation d'une connexion entre composants

4.3.2.3 Évaluation

CCM s'efforce d'isoler des tâches répétitives de CORBA pour en automatiser la génération. Pour cela, CCM privilégie la déclaration des besoins plutôt que leur programmation. Il fournit une méthodologie de production d'applications et permet, contrairement à EJB, de distribuer les composants. Il fournit des interfaces de programmation qui exposent exactement les ports déclarés dans la spécification, facilitant l'implémentation et l'assemblage des composants. Cependant, des méthodes génériques sont toujours disponibles pour utiliser des fonctionnalités ou effectuer des connexions. Par exemple, l'interface `org.omg.Components.Receptacles-Operations` contient des opérations génériques de connexion et déconnexion des composants. L'existence de ces opérations génériques ne permet pas de garantir la *cohérence des communications* vis-à-vis des déclarations IDL3. De plus, la *cohérence des spécifications* n'est pas vérifiée ; des composants peuvent alors fournir (c'est-à-dire `provides` ou `publishes`) une

fonctionnalité sans qu'aucun autre composant ne la requière. Par ailleurs, aucune solution pour gérer la *dynamicité* des composants n'est proposée. Les connexions sont effectuées entre des instances de composants. Elles sont donc statiques. Par exemple, la souscription n'est pas effectuée par rapport à un patron d'événements mais par rapport à une instance d'un composant fournisseur d'un événement (Figure 4.33). Pour gérer la dynamicité, le développeur doit utiliser les API CORBA qui n'offrent aucune garantie sur la fiabilité du code développé comme expliqué dans la section 4.1.2. Enfin, la spécification de l'assemblage des composants dans des fichiers XML n'est pas vérifiée avant l'exécution des applications résultantes.

4.3.3 ArchJava

L'approche ArchJava [ACN02, ASCN03] intègre directement les architectures logicielles dans le langage de programmation Java. Contrairement à l'approche CCM, elle assure l'*intégrité des communications* en garantissant statiquement que les connexions entre les instances des composants respectent la spécification.

4.3.3.1 Programmation

Pour illustrer la programmation dans ArchJava, nous prenons l'exemple d'une application d'alerte incendie où un gestionnaire d'alerte incendie commande un gestionnaire de lumières qui actionne les lumières selon la luminosité extérieure. La figure 4.34 illustre la spécification et la programmation d'un composant `GestionnaireLumières`. La spécification du composant indique qu'il peut être configuré pour les situations d'incendie (ligne 3), qu'il peut envoyer un acquittement correspondant à cette nouvelle configuration (ligne 4) et qu'il peut lire des valeurs de luminosité (ligne 7). À l'instar de CCM, les fonctionnalités sont organisées dans des ports, comme par exemple le port `in` (ligne 2) et le port `out` (ligne 6). Dans le reste du composant, le développeur doit implémenter les fonctionnalités déclarées comme fournies (ligne 10).

```
1  public component class GestionnaireLumières {
2    public port in {
3      provides void mettreEnConfigurationFeu(boolean presenseFeu);
4      requires void acquittement();
5    }
6    public port out {
7      requires Luminosité lireLuminosité();
8    }
9
10   void mettreEnConfigurationFeu(boolean presenceFeu) { ... }
11 }
```

FIG. 4.34: ArchJava - Le composant GestionnaireLumières

L'assemblage est effectué dans le composant `ApplicationAlerteIncendie` (Figure 4.35). Les ports des instances des composants `GestionnaireAlerteIncendie`, `GestionnaireLumières` et `CapteurLuminosité` sont connectés en utilisant le mot clé `connect` (lignes 6 et 7). L'application correspondante est initialisée dans la méthode `main` (lignes 9 à 11). Les programmes ArchJava sont vérifiés et transformés en Java à l'aide d'un pré-processeur.

```
1  public component class ApplicationAlerteIncendie {
2    private final GestionnaireAlerteIncendie gAI = ...;
3    private final GestionnaireLumières gL = ...;
4    private final CapteurLuminosité cL = ...;
5
6    connect gAI.out, gL.in;
7    connect gL.out, cL.in;
8
9    public static void main(String args[]) {
10     new ApplicationAlerteIncendie().test();
11   }
12
13   public void test() {
14     gAI.alert();
15   }
16 }
```

FIG. 4.35: ArchJava - Le composant ApplicationAlerteIncendie définissant l'assemblage

4.3.3.2 Évaluation

ArchJava ne cible pas les systèmes distribués mais permet de définir de nouveaux connecteurs dédiés aux communications distantes. Les développeurs doivent alors eux-mêmes programmer ces connecteurs et les vérifications associées. De plus, ArchJava mélange la spécification des architectures logicielles avec le code applicatif ce qui rend les changements d'architecture logicielle plus difficiles et pénalise l'implémentation en parallèle de plusieurs composants d'une même application. Enfin, ArchJava ne fournit pas de support de programmation et ne permet pas de gérer la dynamicité des applications.

4.4 Bilan

Les solutions existantes pour la programmation des systèmes distribués permettent la spécification des applications, la composition des services, des aides à la programmation et des vérifications statiques des applications. Le tableau 4.36 récapitule les caractéristiques des solutions présentées dans ce chapitre. Chacun des trois aspects abordés dans ce chapitre, c'est-à-dire la programmation des systèmes distribués traditionnels, la programmation des systèmes ubiquitaires et la programmation orientée composant, apporte une contribution différente pour la programmation des systèmes distribués. Les approches pour la programmation des systèmes distribués traditionnels offrent un support de programmation générique permettant de simplifier la programmation de tâches essentielles aux systèmes distribués, c'est-à-dire les communications synchrones et la gestion de la dynamicité grâce aux communications asynchrones, la résolution de noms et la découverte de services. Ce support de programmation générique est en partie spécialisé par la génération de talons depuis les interfaces IDL des services. L'assemblage des services est réalisé dans le code applicatif. La généricité du support de programmation ne permet pas d'effectuer les vérifications statiques nécessaires à la production d'applications fiables. Les approches pour la programmation des systèmes ubiquitaires n'apporte pas d'avancée en terme d'assemblage ou de vérification des applications. En revanche, elles fournissent de nouvelles abstractions de programmation pour gérer la dynamicité des environnements ubiquitaires. Enfin, les approches à base de composants génèrent, à partir d'une architecture logicielle, un canevas de programmation plus riche. Notamment, elles permettent de générer

du support d'implémentation et de connexion dédié à l'architecture logicielle spécifiée. Par ailleurs, elles fournissent des méthodologies pour la production d'applications, incluant du support pour le paquetage et l'assemblage des composants. Seule l'approche ArchJava permet de garantir l'intégrité des communications. Cependant, elle ne propose pas de solution pour programmer des applications distribuées et pour gérer la dynamicité des environnements ubiquitaires. L'approche CCM permet de programmer des applications distribuées en connectant statiquement des instances de composants. Pour gérer la dynamicité, les développeurs doivent utiliser les API CORBA dont la généricité est pénalisante pour la fiabilité des applications. De plus, CCM ne garantit pas l'intégrité des communications.

	RMI	CORBA	Services Web	SIP	One.world	Olympus	EJB 3	CCM	ArchJava
Technologie distribuée	+	+	+	+	+	+	+	+	-
Résolution de noms	+	+	+	+	+	+	+	-	-
Découverte de services	-	+	+	-	+	+	+	-	-
Découverte de services typée	-	-	-	-	-	-	-	-	-
Communication synchrone	+	+	+	+	-	+	+	+	+
Communication asynchrone	-	+	+	+	+	-	+	+	-
Cohérence des spécifications	-	-	-	-	-	-	-	-	-
Intégrité des communications	-	-	-	-	-	-	-	-	+
Indépendance des bus logiciels	-	-	-	-	-	-	-	-	-

FIG. 4.36: Tableau comparatif : (-) et (+) correspondent respectivement à une absence et une présence de la caractéristique

De façon plus générale, les canevas de programmation proposés sont dépendants des technologies sous-jacentes. Les développeurs doivent avoir une expertise dans ses technologies pour produire des applications dont la portabilité est pénalisée. Aucune des approches ne vérifie la cohérence des spécifications des services, ce qui peut entraîner la production d'un support de programmation inadapté et ainsi impacter la fiabilité des applications développées. Aucune solution ne fournit de découverte de services typée, obligeant les développeurs à manipuler des chaînes de caractères et à effectuer des conversions de types potentiellement erronées. Finalement, nous remarquons l'importance des spécifications et en particulier des architectures logicielles. Elles permettent, grâce à des approches génératives, de spécialiser un ensemble d'outils génériques et ainsi fournir davantage de support de programmation ou de vérifications.

Chapitre 5

Démarche suivie

Dans le chapitre 2, nous avons caractérisé les systèmes ubiquitaires et mis en évidence les besoins des applications ubiquitaires. Nous avons ainsi montré que les environnements ubiquitaires sont dynamiques et qu'ils sont composés d'objets communicants hétérogènes. Nous en avons déduit que les systèmes ubiquitaires, potentiellement critiques, requièrent des outils pour l'intégration des services, la gestion de la dynamicité et la vérification des applications à la compilation. Dans le chapitre 3, nous avons introduit les intergiciels à objets répartis et à composants. Les intergiciels fournissent des canevas de programmation spécialisés par des outils de génération depuis des spécifications haut niveau dans le but de faciliter l'implémentation des services constituant les applications. Dans le chapitre 4, nous avons montré que certaines approches satisfont au besoin de dynamicité des systèmes ubiquitaires mais obligent les développeurs à utiliser des canevas de programmation génériques. À l'inverse, d'autres approches permettent d'effectuer des vérifications statiques en garantissant, par exemple, l'intégrité des communications mais ne gèrent pas la dynamicité. Nous récapitulons dans ce chapitre les problèmes rencontrés par les développeurs d'applications ubiquitaires. Par la suite, nous présentons notre approche déclarative et les outils associés pour la gestion de la dynamicité des environnements ubiquitaires et production d'applications fiables.

5.1 Problématique

Les systèmes ubiquitaires imposent de nouveaux défis aux développeurs et nécessitent donc une solution originale. Dans cette section, nous décrivons le cahier des charges de la solution correspondante.

Adaptation des outils aux domaines applicatifs cibles. Les systèmes ubiquitaires impliquent une grande variété de domaines applicatifs et nécessitent des outils de production d'applications capables de s'adapter à de multiples environnements ubiquitaires cibles.

Gestion de la dynamicité des environnements ubiquitaires. La dynamicité des environnements ubiquitaires demande aux applications de s'adapter en permanence. Les développeurs ont besoin d'abstractions pour gérer les changements de contexte et la dynamicité des services.

Vérifications statiques des applications. Les développeurs doivent s'assurer de la fiabilité d'applications potentiellement critiques. Les approches s'appuyant sur le modèle RPC comme RMI et CORBA génèrent du support d'invocation depuis les déclarations d'interfaces des services. Ce support est typé et spécialise en partie les canevas de programmation par rapport aux services modélisés. Cependant, la généricité du support de programmation ne permet pas de vérifier statiquement la validité des connexions et des interactions entre services. D'autres approches comme CCM et ArchJava soutiennent l'assemblage des services. Bien que l'approche ArchJava permette de garantir l'intégrité des communications, elle ne donne aucune solution pour programmer des applications distribuées et gérer la dynamicité des environnements ubiquitaires. L'approche CCM permet de programmer des applications distribuées en connectant statiquement des instances de composants. Pour gérer la dynamicité, les développeurs doivent utiliser les API CORBA dont la généricité est pénalisante pour la fiabilité des applications. De plus, l'intégrité des communications n'est pas garantie. À l'instar de l'approche EJB, CCM permet l'utilisation des descripteurs XML pour décrire le contenu des paquetages, les propriétés de configuration et les assemblages de composants. Ces descripteurs n'étant pas liés au code applicatif, des erreurs peuvent se produire à l'exécution. Il n'existe donc aucune approche gérant la dynamicité des environnements ubiquitaires tout en assurant l'intégrité des communications.

Indépendance des technologies sous-jacentes. Les canevas de programmation pour les systèmes distribués sont fortement liés aux technologies sous-jacentes compliquant la tâche des développeurs et impactant la portabilité des applications. Au contraire, l'indépendance des technologies sous-jacentes permettrait de programmer de la même manière des domaines applicatifs pourtant disjoints. Elle favoriserait ainsi la réutilisation du code applicatif à travers les différents domaines applicatifs des systèmes ubiquitaires.

Validation des applications avant leur déploiement en environnements réels. Aucune solution ne permet d'assurer, avant déploiement en environnement réel, que la logique applicative puisse accomplir la tâche assignée à l'application. Pour valider leur comportement, les applications doivent alors être testées en environnements réels. Ces tests impliquent des scénarios variés nécessitant l'acquisition et le déploiement d'un nombre important d'objets communicants potentiellement onéreux. De plus, certaines applications impliquent des scénarios qui ne peuvent être reproduits en environnement réel, comme par exemple un incendie. Il est donc nécessaire de fournir des outils pour garantir l'accomplissement des tâches attribuées aux applications avant leur déploiement en environnements réels.

Pour répondre à ce cahier des charges, nous proposons l'approche déclarative DIAGEN pour la génération de canevas logiciels ubiquitaires. Nous proposons d'intégrer une spécification haut niveau dans un canevas de programmation et dans un simulateur d'environnements ubiquitaires pour faciliter le développement d'applications distribuées fiables.

5.2 Démarche globale

L'approche que nous proposons pour faciliter le développement d'applications ubiquitaires fiables est fondée sur l'utilisation du langage déclaratif DIASPEC pour décrire l'architecture logicielle des applications ubiquitaires. La figure 5.1 illustre le processus de développement des applications. Notre approche est composée de trois étapes.

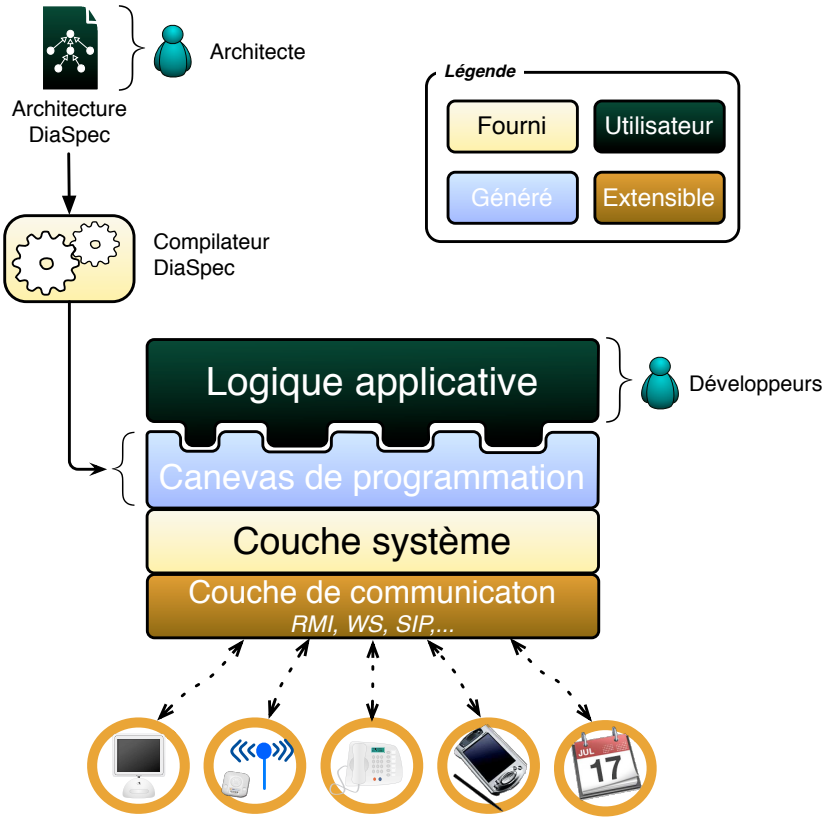


FIG. 5.1: Processus de développement des applications DIAGEN

Dans un premier temps, un architecte modélise l'environnement d'exécution d'une application. Un environnement d'exécution est l'ensemble des services nécessaires à la réalisation d'une application. Les environnements d'exécution d'un même domaine applicatif partagent certaines caractéristiques. Par exemple, toutes les applications de téléphonie contiennent des téléphones. La modélisation de ces environnements s'appuie sur DIASPEC, un langage de description d'architectures logicielles. DIASPEC permet de déclarer les types des services constituants une application. Un type de services est caractérisé par des attributs, des fonctionnalités fournies ou requises et les types de services avec lesquels il a le droit d'interagir. Les déclarations des services sont compilées par le compilateur DIASPEC qui vérifie la cohérence de l'architecture logicielle et génère un canevas de programmation dédié à l'environnement modélisé.

Dans un deuxième temps, le canevas de programmation généré permet la programmation des services de coordination grâce à des structures de code typées pour découvrir des services et interagir avec eux. Il facilite aussi l'implémentation des services primitifs et garantit la cohérence des services développés par rapport à la spécification DIASPEC. À partir du canevas de programmation généré, les développeurs peuvent implémenter une variété d'environnements d'exécution et ainsi s'adapter aux environnements présents et à leur évolution. Les canevas de programmation générés garantissent l'intégrité des communications tout en fournissant les structures de code nécessaires à la gestion de la dynamicité des environnements ubiquitaires. Cette dynamicité est gérée par une découverte typée des services et des mécanismes à événements typés. La *couche système* fournie permet aux canevas de programmation d'abstraire la complexité des interfaces de programmation des bus logiciels sous-jacents. La *couche de communication* contient des implémentations de ces interfaces de programmation et permet d'interagir avec les entités communicantes du bus logiciel associé. L'indépendance du canevas de programmation vis-à-vis des technologies sous-jacentes facilite le travail du développeur et permet la portabilité des applications entre les différents bus logiciels.

Dans un troisième temps, les applications développées sont testées et déboguées dans le simulateur DIASIM. Le simulateur DIASIM est paramétré par la spécification DIASPEC des applications à tester. DIAGEN permet au simulateur DIASIM d'être extensible et d'intégrer, sans modification, des services réels dans les scénarios de simulation. Une application peut alors être exécutée sans modification dans des environnements de simulation hybrides, contenant à la fois des services simulés et des services réels.

Dans la suite de ce document, nous détaillons les différentes étapes de la démarche que nous venons de présenter. Dans un premier temps, nous présentons DIASPEC, le langage de description d'architectures logicielles. Dans un deuxième temps, nous présentons les canevas de programmation dédiés et l'intergiciel générique sous-jacent. Dans un troisième temps, nous présentons une application de l'approche DIAGEN à la simulation. Enfin, nous validons l'approche dans un quatrième temps. Nous décrivons l'implémentation de notre approche, analysons le code généré et comparons l'approche DIAGEN aux solutions existantes.

Deuxième partie

Approche proposée

Chapitre 6

Spécification d'architectures logicielles avec DIASPEC

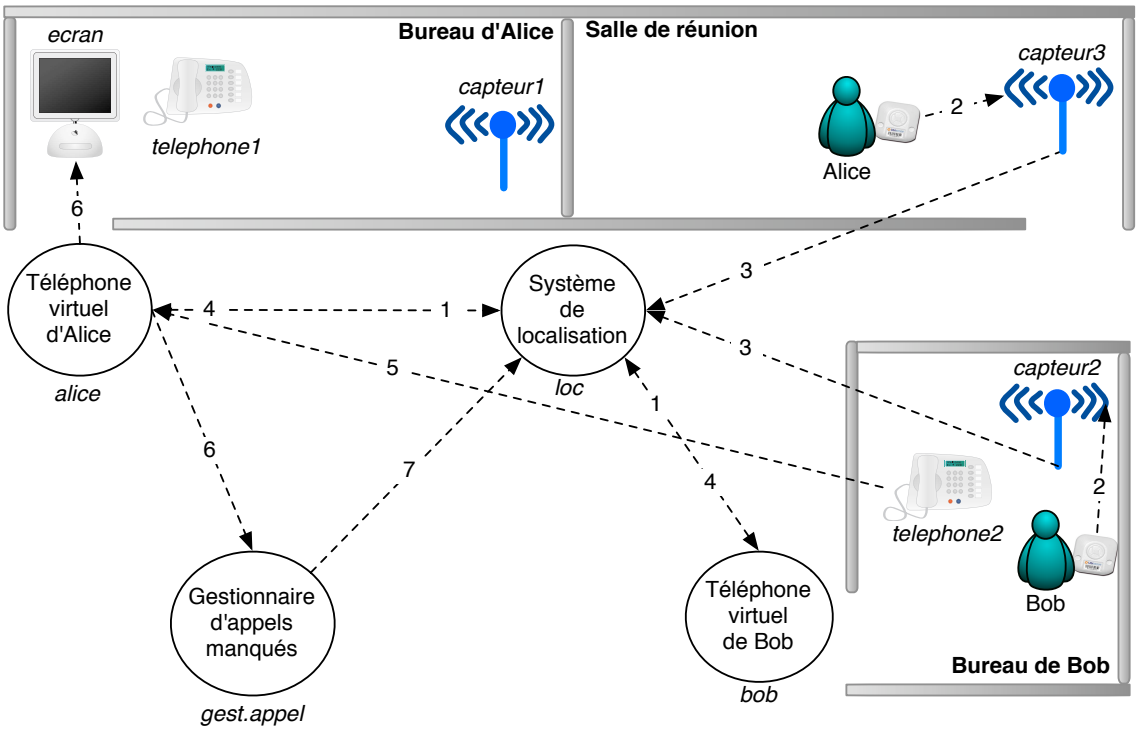
La production des applications, dans l'approche DIAGEN, est fondée sur des taxonomies. Ces taxonomies organisent hiérarchiquement les types de services nécessaires à la construction des applications. Chacune de ces taxonomies contient l'architecture logicielle d'une ou plusieurs applications selon les besoins des environnements cibles. Elles caractérisent des types de services et des connexions entre ces types. Un type de services spécifie l'ensemble des attributs et des fonctionnalités partagés par un groupe nommé de services. Ces informations sont essentielles à la mise en œuvre de vérifications et pour l'exécution d'applications hétérogènes, distribuées et dynamiques. Dans ce chapitre, nous décrivons DIASPEC, un langage de description d'architectures logicielles permettant de spécifier les besoins de ces applications.

6.1 Exemple

Pour motiver notre approche, nous considérons une application permettant aux employés d'une entreprise de ne pas perdre les appels téléphoniques provenant de leurs collègues. Actuellement, un appel doit être répété tant que l'appelé est absent. Quand la personne appelée est de nouveau disponible, l'appelant peut ne plus l'être. Pour résoudre cette situation, un service stocke les appels manqués et rappelle les deux interlocuteurs dès qu'ils deviennent simultanément disponibles. Nous nommerons cette application *l'application de rappel intelligent*.

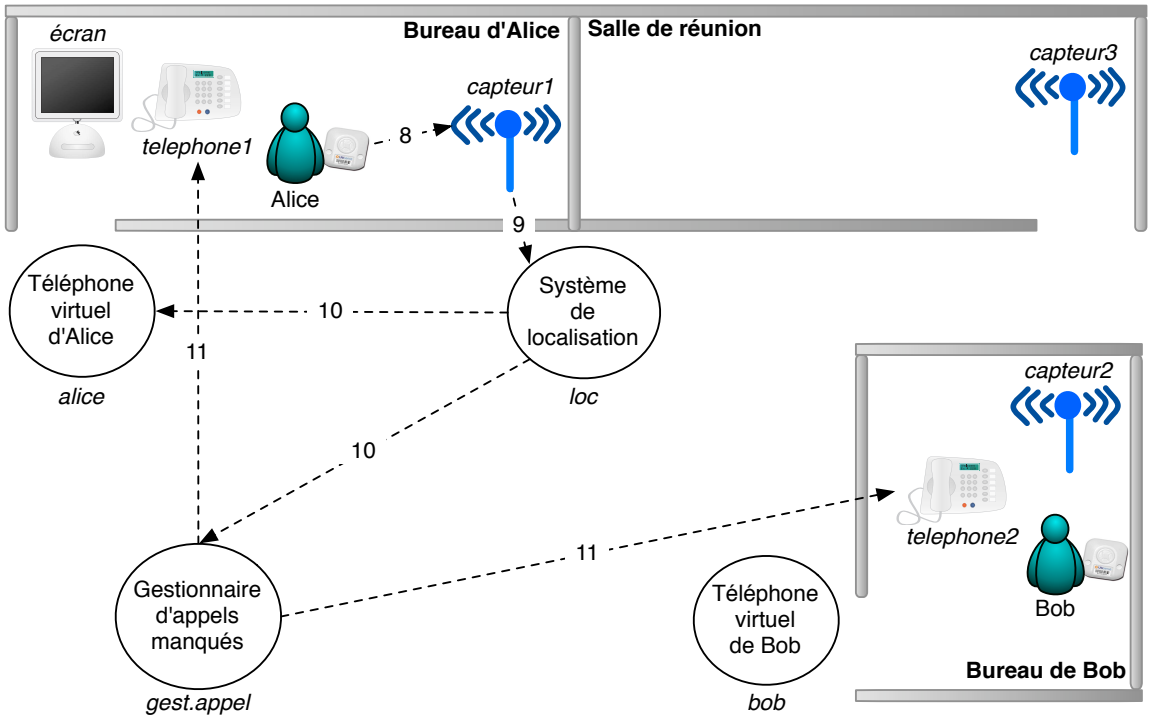
Cette application coordonne une variété de services. Un *système de localisation* localise les utilisateurs en agrégeant les informations provenant de *capteurs de localisation* répartis de façon stratégique dans l'immeuble. Un *téléphone virtuel* gère les appels entrants d'un utilisateur en fonction de sa présence : quand il est disponible, l'utilisateur reçoit l'appel, sinon l'appel est envoyé au *gestionnaire d'appels manqués*. Les figures 6.1 et 6.2 présentent un cas d'usage de notre application. Dans la figure 6.1, Bob appelle Alice alors qu'elle n'est pas dans son bureau. Son appel est alors stocké au niveau du gestionnaire d'appels manqués et Alice est avertie par un message sur son ordinateur ou son PDA. Dans la figure 6.2, Alice et Bob sont mis en contact, lorsqu'Alice est de nouveau dans son bureau.

L'application de rappel intelligent est composée de services primitifs fournis par des téléphones, des écrans ou des capteurs de localisation. Elle est aussi composée de services de coordination comme le système de localisation ou le gestionnaire d'appels manqués.



1. Les téléphones virtuels d'Alice et de Bob souscrivent à la « Présence d'Alice (resp. Bob) dans son bureau ».
2. Les tags d'Alice et de Bob sont détectés par les **Capteurs de localisation**.
3. Les **Capteurs de localisation** publient un événement de « localisation » vers le **Système de localisation**.
4. Le **Système de localisation** publie l'événement « en dehors de son bureau » vers le **Téléphone virtuel** d'Alice.
   Le **Système de localisation** publie l'événement « dans son bureau » vers le **Téléphone virtuel** de Bob.
5. Bob appelle Alice.
   Le **Téléphone** de Bob se connecte au **Téléphone virtuel** d'Alice qui est absente.
6. Le **Téléphone virtuel** d'Alice met l'appel de Bob dans la file d'attente du **Gestionnaire d'appels manqués**.
   Le **Téléphone virtuel** d'Alice affiche un message sur un **Ecran** pour avertir Alice.
7. Le **Gestionnaire d'appels manqués** souscrit à la « Présence d'Alice et de Bob dans leur bureau ».

FIG. 6.1: L'application de rappel automatique basée sur la présence : initialisation



8. Le tag d'Alice est détecté par le **Capteur de localisation**.
9. Le **Capteur de localisation** publie un événement de « Présence » au **Système de localisation**.
10. Le **Système de localisation** publie l'événement « Dans son bureau » au **Gestionnaire d'appels manqués**.
10. Le **Système de localisation** publie l'événement « Dans son bureau » au **Téléphone virtuel** d'Alice.
11. Le **Gestionnaire d'appels manqués** crée une session Audio entre les **Téléphones** d'Alice et de Bob.

FIG. 6.2: L'application de rappel automatique basée sur la présence : un scénario

6.2 Architecture logicielle

Nous allons voir comment spécifier l'architecture logicielle de l'application de rappel intelligent dans le langage de description d'architectures logicielles DIASPEC. Une spécification DIASPEC se compose d'un ensemble cohérent de types de services. Un type de services est un *service DIASPEC* et représente un ensemble de services partageant des caractéristiques communes. Ainsi, un type de services est décrit par un ensemble d'attributs et de fonctionnalités définissant la façon dont les services associés interagissent les uns avec les autres. Enfin, les types de services sont organisés hiérarchiquement, permettant, grâce au polymorphisme, à des instances plus spécifiques d'être utilisées quand des instances moins spécifiques sont nécessaires. La figure 6.3 illustre la hiérarchie de types de services spécifiant l'application de rappel intelligent. Le langage DIASPEC, dont la grammaire est donnée Figure 6.5, est mis en œuvre pour spécifier l'application de rappel intelligent, comme illustré Figure 6.4. Dans le reste de cette section, nous décrivons la déclaration des attributs, des fonctionnalités et des relations hiérarchiques en utilisant la spécification de l'application de rappel intelligent détaillée dans les figures 6.3 et 6.4.

6.2.1 Attributs

Les attributs décrivent les propriétés non fonctionnelles d'un service DIASPEC, comme par exemple la localisation. Dans DIASPEC, les attributs sont spécifiés dans une liste parenthésée après le nom du type de services. Par exemple, dans la figure 6.4, le type de services `EntitéMatérielle` a des attributs `localisation` et `autonomie` (ligne 3). Un attribut est associé à un type Java arbitrairement complexe qui décrit l'ensemble des valeurs possibles pour cet attribut. Les types des attributs sont définis en dehors des spécifications DIASPEC. Par exemple, l'attribut `priorité` (ligne 17) est défini par une classe Java `Priorité` qui est une énumération {`FAIBLE`, `MOYENNE`, `HAUTE`}.

6.2.2 Fonctionnalités

Les fonctionnalités d'un service DIASPEC déterminent comment ce service interagit avec les autres services. Une fonctionnalité est réalisée par la mise en œuvre d'un des trois modes d'interaction fournis par DIASPEC : commande, événement et session. Le mode d'interaction commande permet des communications de type RPC. Le mode d'interaction événement permet des communications asynchrones suivant le modèle événementiel, c'est-à-dire le paradigme publication/souscription [EFGmK03] dans lequel un producteur d'événements envoie des événements à un ensemble de souscripteurs. Le mode d'interaction session permet l'échange de flux de données après négociation. Une fonctionnalité peut être déclarée comme fournie (`provides`) ou requise (`requires`) par les services. Ces déclarations ont les formes suivantes :

`provides` *fonctionnalité*
`requires` *fonctionnalité*

Dans le premier cas, le service est un service serveur et doit implanter la fonctionnalité correspondante. Dans le deuxième cas, le service est un service client qui peut utiliser la fonctionnalité requise.

Une fonctionnalité associe un mode d'interaction à un type de données Java. Pour une commande, le type Java est une interface contenant une liste de méthodes. Par exemple, le

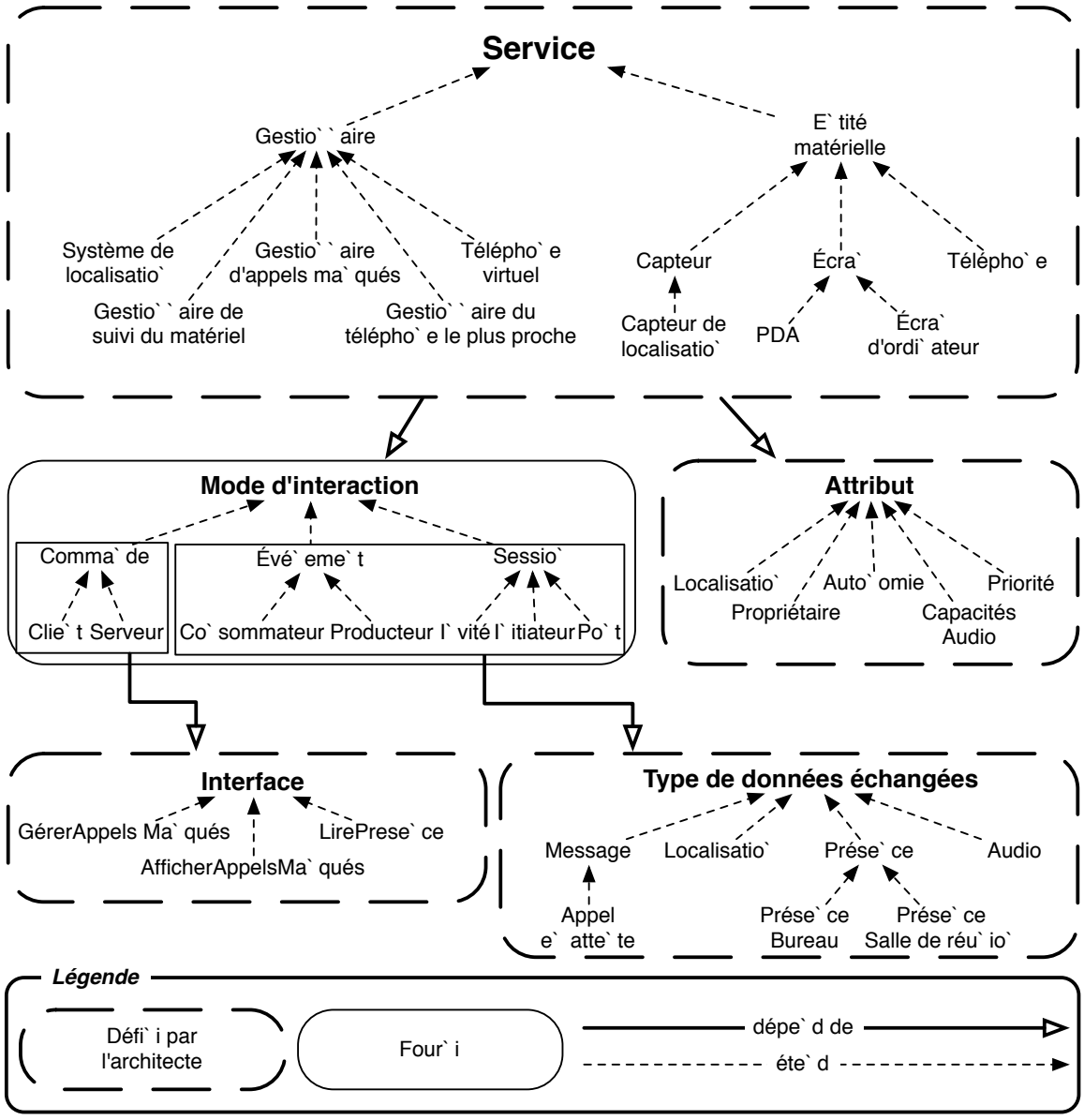


FIG. 6.3: Spécification de l'architecture logicielle de l'application de rappel intelligent

```
1  icommand GererAppelsManqués {
2   void ajouterAppelManqué(Utilisateur appelant, Utilisateur appelé);
3   void supprimerAppelManqué(int appelManquéID);
4  }
5  icommand AfficherAppelsManqués {
6   void afficher(AppelManqué appelManqué);
7  }
8  icommand LirePrésence {
9   Présence lirePrésence(Utilisateur utilisateur);
10 }
11
12 service Service(Propriétaire propriétaire) {
13 }
14 service EntitéMatérielle(Localisation localisation, Autonomie autonomie)
15    extends Service {
16 }
17 service CapteurLocalisation() extends Capteur {
18   provides event Localisation to SystèmeLocalisation;
19 }
20 service Téléphone(CapacitésAudio capacitésAudio) extends EntitéMatérielle {
21   provides session Audio to GestionnaireAppelsManqués;
22   requires session Audio from TéléphoneVirtuel;
23 }
24 service Écran() extends EntitéMatérielle {
25   provides command AfficherAppelsManqués to GestionnaireAppelsManqués;
26   requires command GérerAppelsManqués from GestionnaireAppelsManqués;
27 }
28 service Gestionnaire(Priorite priorite) extends Service {
29   provides command ActiverDesactiver to GestionnaireServices;
30 }
31 service TéléphoneVirtuel() extends Gestionnaire {
32   requires session Audio from Téléphone;
33   requires event PrésenceBureau from SystèmeLocalisation;
34   requires command GérerAppelsManqués from GestionnaireAppelsManqués;
35   provides session Audio to Téléphone;
36 }
37 service SystèmeLocalisation(Localisation localisation) extends Gestionnaire {
38   requires event Localisation from CapteurLocalisation;
39   provides command LirePrésence to GestionnaireTéléphoneLePlusProche;
40   provides session Présence to GestionnaireSuiviMateriel;
41   provides event PrésenceBureau to TéléphoneVirtuel;
42   provides event PrésenceBureau to GestionnaireAppelsManqués;
43 }
44 service GestionnaireAppelsManqués() extends Gestionnaire {
45   requires event PrésenceBureau from SystèmeLocalisation;
46   requires command AfficherAppelsManqués from Écran;
47   provides command GérerAppelsManqués to TéléphoneVirtuel;
48   binds session Audio from (Téléphone, Téléphone);
49 }
```

FIG. 6.4: Extrait de la spécification DIASPEC de l'application de rappel intelligent

service DIASPEC `GestionnaireAppelsManqués` fournit une fonctionnalité de commande pour ajouter et supprimer des appels manqués (ligne 35). L'interface `GérerAppelsManqués` liste les méthodes correspondantes (lignes 39 à 42). Pour le mode d'interaction événement, le type Java est le type de la donnée échangée, c'est-à-dire le type de l'événement. Par exemple, le service DIASPEC `SystèmeLocalisation` est producteur d'événements `Présence` informant sur l'entrée et la sortie des utilisateurs de certains lieux comme les salles de réunion (ligne 30). Pour le mode d'interaction session, le type Java définit les paramètres nécessaires à

```
softwareArchitecture ::= attribute* datatype* commandInterface* serviceDefinition*
constructor          ::= serviceid ( (attributeid (, attributeid)* )? )
serviceDefinition    ::= service constructor (extends serviceid)? { interaction* }
interaction          ::= requires functionality from serviceid ;
                       | provides functionality to serviceid ;
                       | binds session classid from (serviceid, serviceid) ;
functionality        ::= command interfaceid
                       | event classid
                       | session classid
```

FIG. 6.5: Extrait de la grammaire du langage DIASPEC

la négociation du flux de données. Par exemple, le type `Audio` définit le paramètre `codec` sur lequel les services de type `Téléphone` doivent se mettre d'accord pour établir un échange de flux audio (lignes 10 et 11). Les types de données échangées sont organisés hiérarchiquement. Par exemple, la figure 6.3 présente la hiérarchie des types de données échangées pour l'application de rappel intelligent. Le type `Présence` contient deux sous-noeuds `PrésenceBureau` et `PrésenceSalleRéunion`. Un service peut alors souscrire à l'un de ces deux types d'événements ou au type générique `Présence`.

Finalement, l'utilisation d'une fonctionnalité est restreinte par une contrainte, appelée *visibilité*, sur le type des services avec lesquels l'interaction est permise. La visibilité est exprimée après les mots clés `to`, spécifiant les services clients des fonctionnalités fournies, et `from`, spécifiant les services serveurs des fonctionnalités requises. Les visibilités définissent des *connexions* entre types de services pour une fonctionnalité donnée.

Pour illustrer ces déclarations de façon plus détaillée, nous considérons les services DIASPEC `SystèmeLocalisation` et `CapteurLocalisation` déclarés dans la figure 6.4. Le service DIASPEC `CapteurLocalisation` produit des événements `Localisation` (par exemple des coordonnées cartésiennes) vers les services de type `SystèmeLocalisation`. De façon complémentaire, le service DIASPEC `SystèmeLocalisation` requiert des événements de type `Localisation`. À partir des événements `Localisation`, il fournit des informations sur la présence des utilisateurs dans un ensemble de lieux prédéfinis, comme des bureaux ou des salles de réunion, en tant que commande de type `LirePrésence` (ligne 28), session de données de type `Présence` (ligne 29) et événement de type `Présence` (ligne 30).

6.2.3 Hiérarchie

Les services DIASPEC sont structurés dans une hiérarchie, comme illustré pour l'application de rappel intelligent de la figure 6.4. Chaque service DIASPEC spécifie le type de services qu'il étend avec le mot-clé `extends`. Partant du noeud racine `Service`, la hiérarchie contient des types de services de plus en plus spécifiques. Chaque noeud successif ajoute de nouveaux attributs et de nouvelles fonctionnalités qui sont spécifiques au type de services qu'ils caractérisent. Un service DIASPEC hérite des attributs et des fonctionnalités de ses ancêtres.

Dans notre approche, l'héritage joue un rôle important pour la découverte de services. Conceptuellement, un développeur voulant accéder à un type de services désigne le noeud correspondant dans la hiérarchie et reçoit tous les services correspondant aux types de services contenus dans le sous-arbre. Cette stratégie implique que le code qui implante un type de

services soit associé au noeud le plus bas possible dans la hiérarchie pour exposer précisément le maximum de ses fonctionnalités fournies aux autres services. À l'inverse, le code qui utilise un type de services doit spécifier le type de services le plus générique satisfaisant ses besoins. En procédant ainsi, (1) une requête de découverte de services a une probabilité plus importante de retourner un plus grand nombre de services et (2) l'application résultante expose seulement les fonctionnalités nécessaires, améliorant ainsi sa portabilité et la rendant compatible avec les évolutions futures des types de services demandés.

6.3 Vérifications

Le compilateur DIASPEC vérifie la cohérence de la spécification DIASPEC et génère un canevas de programmation dédié à l'architecture logicielle spécifiée. Le compilateur DIASPEC considère l'ensemble des types de services d'une architecture logicielle ce qui lui permet d'effectuer des vérifications de cohérence. Le compilateur DIASPEC garantit ainsi qu'à partir des services DIASPEC déclarés, le développeur peut programmer une application où chaque type de services joue pleinement son rôle, c'est-à-dire où chaque donnée produite est consommée (et vice versa). Nous détaillons ici les trois principales vérifications effectuées par le compilateur DIASPEC : la cohérence des services serveurs, la complétude des services clients et l'intégrité des visibilités. Nous ne traitons pas du cas `binds` qui est similaire au cas `requires`.

Nous notons A $\sqsubseteq$ B si et seulement si A est B ou A est une sous classe de B, où A et B sont soit des types de services soit des types de données.

Cohérence des services serveurs. Le compilateur DIASPEC assure que pour chaque type de services clients, il existe un type de services serveurs correspondant. La vérification correspondante est énoncée comme suit :

Soit M un mode d'interaction, D_0 et D_1 des types de données et S_1 un type de services.
Pour chaque type de services S_0 tel que

S_0 *requires* M D_0 *from* S_1

$\exists$ un type de services S_2 tel que

S_2 *provides* M D_1 *to* S_3

avec $S_0 \sqsubseteq S_3$, $D_0 \sqsubseteq D_1$ et $S_1 \sqsubseteq S_2$.

Complétude des services clients. Pour chaque type de services serveurs, il doit exister au moins un type de services clients capable de consommer le type exact de données produites par le type de services serveur. La vérification correspondante est énoncée comme suit :

Soit M un mode d'interaction, D un type de données et S_1 un type de services.
Pour chaque type de services S_0 tel que

S_0 *provides* M D *to* S_1

$\exists$ un type de services S_2 tel que

S_2 *requires* M D *from* S_3

avec $S_2 \sqsubseteq S_1$ et $S_3 \sqsubseteq S_0$.

Intégrité des visibilités. Un type de services peut restreindre la visibilité d'une fonctionnalité du type de services qu'il étend. En revanche, il ne peut pas augmenter cette visibilité. La vérification correspondante est énoncée comme suit :

Soit F une fonctionnalité et S_0 un type de services tel que

S_0 `requires/provides` F `from/to` S_1

Soit S_3 un type de services.
$\forall$ service S_2 tel que $S_2 \sqsubseteq S_0$ et

S_2 `requires/provides` F `from/to` S_3

alors $S_3 \sqsubseteq S_1$.

6.4 Bilan

Nous avons présenté dans ce chapitre un langage de descriptions d'architectures logicielles nommé DIASPEC. Ce langage permet de spécifier les types de services nécessaires à la programmation d'une application. Le compilateur DIASPEC garantit la cohérence des spécifications DIASPEC. Une fois les vérifications effectuées, le compilateur DIASPEC génère le code nécessaire au développement d'applications valides par rapport aux spécifications DIASPEC. Le code généré spécialise des outils génériques : un intergiciel et un simulateur destinés respectivement à l'exécution des applications en environnements réels et simulés. Nous décrivons dans les prochains chapitres la structure de ces outils, leur paramétrage par les spécifications DIASPEC et les garanties apportées en terme de fiabilité.

Chapitre 7

Canevas de programmation dédiés

Une spécification DIASPEC capture les caractéristiques des types de services nécessaires au développement d'une application. À partir de cette spécification, le compilateur DIASPEC génère un canevas de programmation qui fournit du support et des vérifications dédiés à l'architecture logicielle spécifiée. Comme illustré dans la figure 7.1, le canevas de programmation spécialise l'API générique d'une couche système fourni. Cette couche système contient les services systèmes communs à toutes les architectures logicielles. Les développeurs implémentent ensuite les services DIASPEC en s'appuyant sur le canevas de programmation généré. Enfin, ils instancient les services dans des programmes de déploiement qui référencent le bus logiciel à utiliser.

Dans ce chapitre, nous détaillons le support et les vérifications apportés par les canevas de programmation générés. Nous montrons l'intérêt des différentes couches composant l'intergiciel.

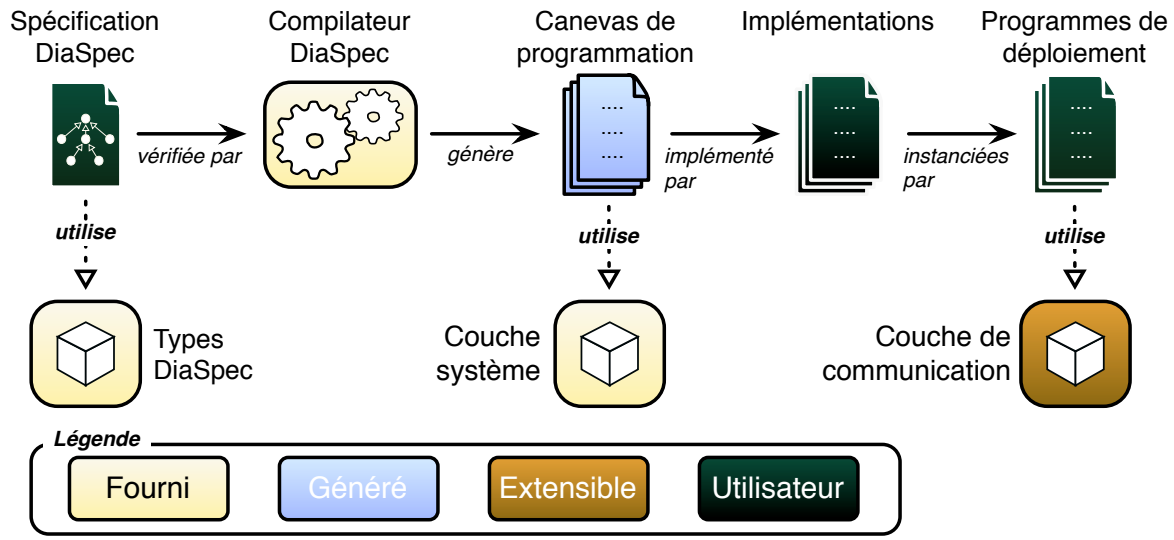


FIG. 7.1: Chaîne de production des services

7.1 Architecture de l'intergiciel

Le compilateur DIASPEC génère une surcouche logicielle dédiée à l'architecture logicielle spécifiée spécialisant un intergiciel générique. Comme illustré dans la figure 7.2, l'intergiciel résultant est composé de trois couches. Le *canevas de programmation* fournit les abstractions nécessaires au développement des applications. La *couche système* permet au canevas de programmation d'être indépendant vis-à-vis des protocoles de communication sous-jacents. Pour cela, elle fournit une API qui masque la complexité des API du bus logiciel implémenté par la *couche de communication* sous-jacente. La couche de communication est extensible et un nouveau bus logiciel peut être ajouté sans impacter les couches supérieures de l'intergiciel. La *couche système* contient les mécanismes génériques essentiels à la communication entre les services. Il s'agit de la découverte de services et des modes d'interaction introduits dans le chapitre 6. Nous détaillons ces mécanismes dans les sous-sections suivantes.

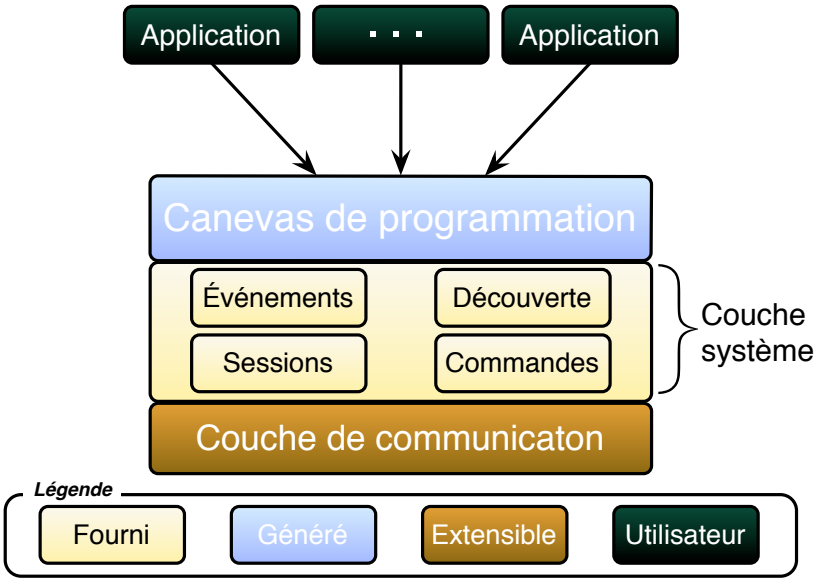


FIG. 7.2: Architecture de l'intergiciel

7.1.1 Découverte

Les services serveurs s'enregistrent sur un courtier de services en spécifiant les valeurs de leurs attributs et le service DIASPEC qui décrit leurs fonctionnalités. Comme illustré dans la figure 7.3, le courtier de services attache chaque service enregistré à un service DIASPEC. Un service client exprime ses besoins au courtier de services en lui indiquant le service DIASPEC voulu. En retour, le courtier lui envoie la liste des services contenus dans le sous-arbre de la hiérarchie des services DIASPEC. Dans la figure 7.3, les services recherchés étant de type 3, la liste des services découverts contient non seulement les services attachés au noeud 3 mais aussi les services attachés à ses noeuds fils, c'est-à-dire les noeuds 4 et 5. En plus du service DIASPEC, un client peut spécifier les valeurs des attributs des services recherchés. Une deuxième sélection est alors effectuée sur les services du sous-arbre de la hiérarchie.

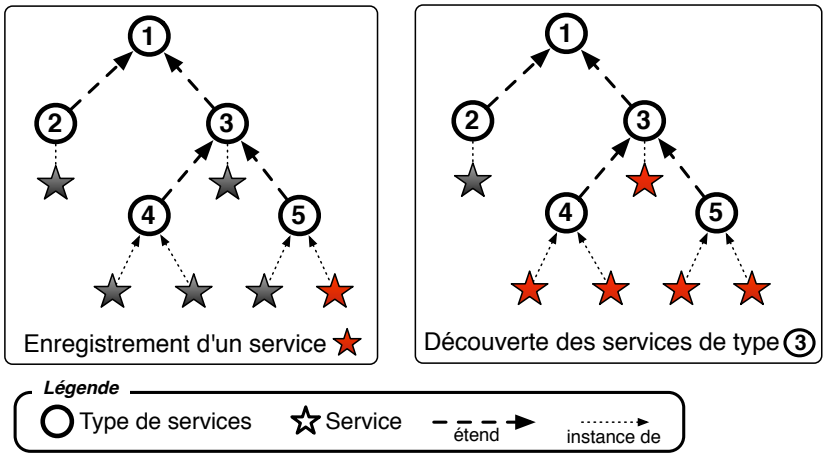


FIG. 7.3: Organisation des services dans le courtier de services

7.1.2 Modes d'interaction

Commande. Les commandes suivent le modèle RPC. Les commandes sont invoquées par l'intermédiaire de mandataires ou talons clients qui transmettent l'appel au service correspondant. La réponse est envoyée par l'intermédiaire de talons serveurs.

Événement. La gestion des événements repose sur le modèle événementiel fondé sur le sujet. Le souscripteur, qui est aussi le consommateur, envoie une demande de souscription à un courtier d'événements. Le sujet de la demande est composé du service DIASPEC du fournisseur, des valeurs de ses attributs et du type d'événements. Les types d'événements étant organisés hiérarchiquement dans la spécification DIASPEC, les événements reçus par le consommateur correspondent au type d'événements souscrit et à ses sous-types. De même, les événements reçus sont produits par les services correspondant au service DIASPEC souscrit et à ses sous-types. Le filtrage des événements publiés est effectué dynamiquement par le courtier d'événements.

Session. La couche système est responsable de la négociation des paramètres des flux de données. Le processus de négociation implique deux participants et aboutit à la création d'un contrat ou accord qui devra être respecté jusqu'à la déconnexion de l'un des participants. Une négociation est généralement composée des étapes suivantes [Hua03, Pic03] : dans un premier temps, chaque participant annonce sa propre vision du contrat. Ensuite, un algorithme détermine un compromis en fonction de préoccupations fonctionnelles et/ou non fonctionnelles, comme par exemple le temps, les ressources et le coût financier. Finalement, quand une solution est acceptée par les deux participants, un contrat est créé. Il existe deux stratégies de négociation distinctes : la négociation concurrente et la négociation collaborative. La négociation concurrente implique un conflit d'intérêt entre le service client et le service serveur et nécessite potentiellement plusieurs transactions avant d'arriver à un compromis. À l'inverse, la négociation collaborative est généralement réalisée en une seule transaction entre le service client et le service serveur car il s'agit d'un processus fondé sur l'intérêt mutuel qui aboutit à une solution gagnant-gagnant pour les deux participants.

Dans le cadre de la négociation des paramètres d'un flux de données, les objets communicants ont des capacités statiques et ne peuvent moduler leur offre de négociation. L'approche DIAGEN adopte un processus de négociation collaborative car n'importe quel sous-ensemble des capacités communes au service client et au service serveur est acceptable. La gestion des sessions dans DIAGEN est similaire à la gestion, par le protocole SIP, des sessions multimédia entre des terminaux téléphoniques de voix sur IP. À l'instar de SIP, DIAGEN utilise le modèle offre/réponse pour parvenir à un accord sur les paramètres de la session [RS02]. La gestion des sessions DIAGEN est illustrée dans la figure 7.4. L'initiateur envoie une demande de connexion à l'invité pour un type de session donné, comme par exemple une session vidéo. La demande de connexion contient les paramètres souhaités pour le flux de données comme la résolution ou l'encodage d'un flux vidéo. Ces paramètres sont généralement liés aux capacités de l'objet communicant (par exemple, un PDA a une résolution maximale de 640x480) et peuvent inclure plusieurs choix (par exemple, un PDA peut lire des flux encodés en MPEG1 et MPEG4). En retour, l'invité envoie un sous-ensemble des paramètres souhaités (dans le message OK), correspondant à l'intersection des capacités de l'initiateur et de l'invité. Finalement, l'initiateur envoie un message d'acquittement pour démarrer l'échange des données en accord avec les paramètres négociés. Les flux de données peuvent être unidirectionnels comme sur la figure 7.4 ou bidirectionnels comme pour une conversation téléphonique.

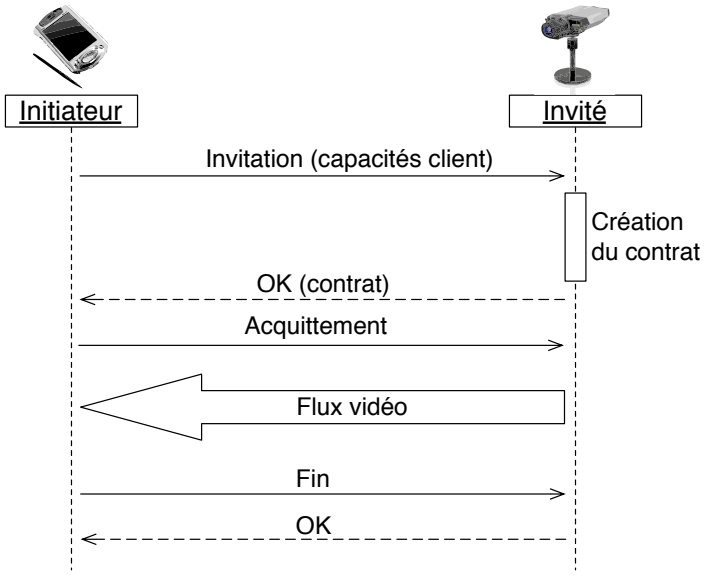


FIG. 7.4: Gestion des sessions dans DIAGEN

Les applications ubiquitaires font intervenir des services beaucoup plus hétérogènes que ceux du domaine de la téléphonie. Pour résoudre cette hétérogénéité, DIAGEN permet de gérer, en plus des flux audio et vidéo, des flux de données provenant par exemple de capteurs de mesures.

7.2 Programmation des services

Pour programmer un service, le développeur détermine dans un premier temps le type de services qu'il souhaite développer. La déclaration du type des services fournit ensuite au développeur une interface de programmation dédiée à l'implémentation des fonctionnalités du service. L'interface de programmation est fournie par un canevas de programmation généré depuis la spécification DIASPEC. Dans cette section, nous décrivons le support et les vérifications fournis par les canevas de programmation générés.

7.2.1 Support de programmation

Pour chaque service DIASPEC, le compilateur génère une classe abstraite qui contient des méthodes et un constructeur correspondant respectivement aux fonctionnalités et aux attributs caractérisant le service. Le compilateur DIASPEC génère, dans la classe abstraite, une implémentation de ces méthodes lorsqu'elles dépendent exclusivement des informations contenues dans le service DIASPEC associé. Par exemple, une méthode pour publier un événement ne dépend que du type d'événements et est donc définie dans la classe abstraite. À l'inverse, les commandes sont représentées par des méthodes abstraites que le développeur doit implémenter dans le service. De cette façon, le développeur peut se concentrer sur la programmation de la logique applicative. Par ailleurs, l'utilisation de classes abstraites permet la génération de support de programmation dans des IDE *(Integrated Development Environment)* comme Eclipse [ecl]. Pour créer un service, le développeur étend la classe abstraite correspondant à un service DIASPEC. Dans les IDE, cette action produit un squelette de classe que le développeur complète avec la logique applicative du service. Nous décrivons maintenant la programmation de l'implémentation, de l'utilisation et de la découverte de services dans les canevas de programmation générés.

7.2.1.1 Fonctionnalités des services

Les fonctionnalités sont représentées par des méthodes implémentées et des méthodes abstraites. Comme illustration, nous présentons la programmation d'un gestionnaire d'appels manqués implémentant le service DIASPEC `GestionnaireAppelsManqués` introduit dans la section 6.1 à la page 63. La figure 7.5 montre le squelette d'un service `MonGestionnaire-AppelsManqués` après extension de la classe abstraite `GestionnaireAppelsManqués`. La figure 7.6 montre l'implémentation correspondante.

Commande Les commandes sont représentées par des méthodes abstraites dans la classe abstraite. Eclipse génère les squelettes des méthodes correspondantes dans la classe du service à implémenter. Comme illustré dans la figure 7.5 pour le gestionnaire d'appels manqués, les méthodes représentant les commandes sont `ajouterAppelManque`, `supprimerAppelManque`, `activer`, `desactiver`. Les interfaces `IGererAppelsManqués` et `IActiverDesactiver` déclarant ces méthodes sont respectivement fournies par les types de services `Gestionnaire-AppelsManqués` et `Gestionnaire`, comme illustré dans la spécification DIASPEC de la figure 6.4 à la page 68.

Les services qui requièrent une commande invoquent la méthode correspondante via un appel de type RPC. Par exemple, le service `MonGestionnaireAppelsManqués` appelle la méthode `afficher` pour envoyer un message sur un écran (figure 7.6, ligne 37).

```
1  public class MonGestionnaireAppelsManqués extends GestionnaireAppelsManqués {
2
3    public MonGestionnaireAppelsManqués(Priorité priorité, Propriétaire propriétaire) {
4      super(priorité, propriétaire);
5      // TODO Auto-generated constructor stub
6    }
7
8    public void receive(PrésenceBureau event) {
9      // TODO Auto-generated method stub
10   }
11
12   public void ajouterAppelManqué(Utilisateur appelant, Utilisateur Appelé) {
13     // TODO Auto-generated method stub
14   }
15
16   public void supprimerAppelManqué(int appelManquéID) {
17     // TODO Auto-generated method stub
18   }
19
20   public void activer() {
21     // TODO Auto-generated method stub
22   }
23
24   public void desactiver() {
25     // TODO Auto-generated method stub
26   }
27
28 }
```

FIG. 7.5: Le squelette de la classe MonGestionnaireAppelsManqués (produit par Eclipse)

Événement Pour un service DIASPEC qui déclare fournir un type d'événements, la classe abstraite définit une méthode de publication pour chaque type d'événements à publier. Les producteurs d'événements invoquent ces méthodes pour publier les événements correspondants. Par exemple, le service de type `SystèmeLocalisation` publie un événement de localisation quand une personne change de lieu, en invoquant la méthode `publish(Localisation localisation)`. Cet événement est reçu par tous les services souscripteurs.

Pour un service DIASPEC déclarant consommer un type d'événements, la classe abstraite définit une méthode abstraite `receive` pour chaque type d'événements. Cette méthode a le type d'événements en guise d'argument. À partir de cette définition, Eclipse produit un squelette de la méthode qui doit être implémenté par le développeur. Les lignes 13 à 27 de la figure 7.6 montrent l'implémentation d'une méthode `receive` qui permet au gestionnaire d'appels manqués de recevoir et traiter les événements de type `PrésenceBureau`. La classe abstraite définit une méthode `subscribePrésenceBureau` pour que le gestionnaire d'appels manqués puisse souscrire au type d'événements `PrésenceBureau` comme déclaré dans la spécification DIASPEC. Le gestionnaire d'appels manqués souscrit aux événements `PrésenceBureau` à la ligne 10 (Figure 7.6). La classe abstraite définit aussi une méthode `unsubscribe` pour annuler une souscription.

Session Pour un service DIASPEC déclaré fournisseur pour le mode d'interaction session (c'est-à-dire invité), la classe abstraite définit les méthodes abstraites `connect` et `disconnect` qui, une fois implémentées par les développeurs, permettent de respectivement négocier un flux de données et arrêter une session. Les services initiateurs de sessions invoquent

```
1  public class MonGestionnaireAppelsManqués extends GestionnaireAppelsManqués {
2     private IAfficherAppelsManqués[] ecrans;
3     [...]
4     public MonGestionnaireAppelsManqués() {
5       super(Priorite.HAUTE, Proprietaire.ADMIN);
6
7       SystèmeLocalisationFilter filtre = SystèmeLocalisation.getFilter();
8       filtre.setLocalisationFilter(Localisation.BâtimentA);
9
10      subscribePrésenceBureau(filtre);
11    }
12
13    public void receive(PrésenceBureau event) {
14       AppelManque appelManque = mettreAjourAppelsManqués(
15            event.lireUtilisateur(),
16            event.estDansSonBureau()
17       );
18
19       TelephoneFilter filtre1 = Telephone.getFilter();
20       filtre1.setProprietaireFilter(appelManque.lireAppeleID());
21
22       TelephoneFilter filtre2 = Telephone.getFilter();
23       filtre2.setProprietaireFilter(appelManque.lireAppelantID());
24
25       AudioSession session = bindAudio(filtre1, filtre2);
26       [...]
27    }
28
29    public void ajouterAppelManque(Utilisateur appelant, Utilisateur appele) {
30       AppelManque appelManque = stockerAppelManque(appelant, appele);
31
32       ÉcranFilter filtre = Écran.getFilter();
33       filtre.setProprietaireFilter(appele.getID());
34
35       ecrans = getIAfficherAppelsManquésServices(filtre);
36       for (IAfficherAppelsManqués ecranAppele: écrans)
37          ecranAppele.afficher(appelManque);
38    }
39
40    public void supprimerAppelManque(int appelManqueID) {
41         [...]
42    }
43
44    // Mise à jour des appels manqués avec la présence des appelants et appelés.
45    // Retourne le prochain appel manqué pour lequel l'appelant et l'appelé
46    // sont dans leur bureau.
47    private AppelManque mettreAjourAppelsManqués
48                               (Utilisateur utilisateur, boolean estDansSonBureau) {
49         [...]
50    }
51
52    // Stockage des appels.
53    private AppelManque stockerAppelManque(Utilisateur appelant, Utilisateur appele) {
54         [...]
55    }
56    [...]
57 }
```

FIG. 7.6: L'implémentation d'un gestionnaire d'appels manqués

ces méthodes pour débuter et stopper un flux de données.

Enfin, pour un service DIASPEC se déclarant *pont de signalisation* (`binds` dans la spécification DIASPEC) pour le mode d'interaction session, la classe abstraite définit une méthode `bind` pour chaque type de sessions, par exemple `bindAudio` pour une session audio. Cette méthode permet de créer un flux de données entre deux autres services. Dans notre exemple, les invités sont des services de type `Téléphone` et le gestionnaire d'appels manqués `MonGestionnaireAppelsManqués` est un pont de signalisation pour les sessions de type audio. `MonGestionnaireAppelsManqués` établit une session audio entre les téléphones de l'appelant et de l'appelé (ligne 25 de la figure 7.6). La négociation des paramètres de la session est gérée automatiquement par la couche système.

7.2.1.2 Découverte de services

Enregistrement Le développeur spécifie les valeurs des attributs d'un service en appelant le constructeur de la classe abstraite avec le mot-clé Java `super`. Le constructeur de la classe abstraite est ensuite responsable de l'enregistrement du service sur le courtier de services. Dans la figure 7.6, le service `MonGestionnaireAppelsManqués` est enregistré avec une priorité `HAUTE` et un propriétaire `ADMIN` (ligne 5).

Localisation Le canevas de programmation fournit aux développeurs des méthodes pour sélectionner des noeuds dans la hiérarchie des services DIASPEC. Le résultat de cette sélection est un *filtre* qui désigne l'ensemble de tous les services correspondant au noeud sélectionné et à ses sous-noeuds. Le développeur peut ensuite raffiner ce filtre en spécifiant les valeurs des attributs des services recherchés. Par exemple, le développeur crée un filtre désignant l'ensemble des services de type `SystèmeLocalisation` localisés dans le `BâtimentA` aux lignes 7 à 8 de la figure 7.6.

Un service utilise les filtres pour désigner les services avec lesquels il veut interagir. Pour les commandes, des méthodes `getService` et `getServices` sont définies dans la classe abstraite des services clients. Ces méthodes sont invoquées avec un filtre en argument et retournent respectivement un service ou tous les services respectant les contraintes du filtre. Par exemple, le gestionnaire d'appels manqués appelle la méthode `getIAfficherAppels-ManquésServices` sur un filtre désignant les services de type `Écran` dont le propriétaire est l'appelé (ligne 35). La méthode `afficher` est ensuite appelée sur les services découverts pour afficher un message concernant l'appel manqué (ligne 37).

Pour les événements, le filtre est l'argument de la méthode de souscription et permet au courtier d'événements de filtrer les événements non seulement par rapport à leur type, mais aussi par rapport aux caractéristiques des fournisseurs. Dans notre exemple, le gestionnaire d'appels manqués souscrit aux événements de type `PrésenceBureau` provenant des services de type `SystèmeLocalisation` localisés dans le `BâtimentA` (lignes 7 à 10).

Pour les sessions, les filtres sont les arguments des méthodes `connect` et `bind`. Dans notre exemple, la méthode `bindAudio` prend en arguments deux filtres correspondant à deux services entre lesquels le gestionnaire d'appels manqués veut créer une session audio. Les filtres désignent respectivement le téléphone de l'appelé et de l'appelant (lignes 19 à 25).

7.2.2 Vérifications

Les canevas de programmation sont générés de telle sorte que les développeurs soient contraints de respecter les spécifications DIASPEC associées. Plusieurs vérifications sont alors fournies à la compilation et à l'exécution. L'approche DIAGEN met en œuvre ces vérifications tout en prenant en compte la dynamicité des environnements ubiquitaires grâce à l'utilisation des filtres.

7.2.2.1 Compilation

Implémentation des fonctionnalités. Pour les commandes fournies, les sessions fournies et la consommation des événements, les méthodes abstraites définies dans les classes abstraites obligent les développeurs à implémenter les fonctionnalités correspondantes.

Intégrité des communications. Le canevas de programmation garantit l'intégrité des communications entre les services [ACN02]. Il vérifie que les fonctionnalités utilisées par un service sont déclarées dans le service DIASPEC correspondant. Pour cela, le compilateur DIASPEC génère uniquement les méthodes nécessaires à l'accomplissement des fonctionnalités déclarées (par exemple `subscribePrésenceBureau`, `bindAudio` et `getIAfficherAppels-ManquésServices` dans la figure 7.6). Il vérifie aussi que les interactions entre services respectent les visibilités déclarées dans la spécification DIASPEC. Pour cela, le compilateur DIASPEC génère des méthodes dont les types des arguments obligent le développeur à respecter les visibilités des fonctionnalités déclarées dans la spécification. Par exemple, le service DIASPEC `GestionnaireAppelsManqués` est déclaré en tant que souscripteur d'événements `PrésenceBureau` auprès du service DIASPEC `SystèmeLocalisation` (ligne 33 de la figure 6.4). Une méthode `subscribePrésenceBureau` prenant en argument un filtre de type `SystèmeLocalisationFilter` est alors générée dans la classe abstraite correspondante et garantit que la souscription est restreinte au type de services `SystèmeLocalisation` ou à ses descendants (ligne 10 de la figure 7.6). De même, les types des filtres donnés en argument sont restreints pour la méthode `bindAudio` utilisée pour les sessions (ligne 25) et la méthode `getIAfficherAppelsManquésServices` utilisée pour les commandes (ligne 35).

Typage des structures de code. Le canevas de programmation est généré de telle façon qu'aucune chaîne de caractères ou conversion de type ne soit nécessaire. Par exemple, les attributs des services et leurs valeurs sont typés (lignes 4 et 7), les événements reçus sont typés (par exemple `PrésenceBureau event`, ligne 12) et les découvertes de services renvoient des références typées (par exemple `IAfficherAppelsManqués`, ligne 34). DIAGEN permet ainsi d'éviter les erreurs de conversion de type et les chaînes de caractères erronées.

7.2.2.2 Exécution

À l'exécution, le courtier de services contrôle périodiquement les services enregistrés et vérifie que chaque commande a au moins un service client, que chaque producteur d'événements a au moins un consommateur, et que chaque invité en session a au moins un service client et *vice-versa*. Si ces conditions ne sont pas remplies, le courtier affiche des avertissements.

7.2.3 Évolution

La dynamicité des systèmes ubiquitaires demande des capacités d'évolution adaptées. DIAGEN permet l'évolution des applications et des environnements d'exécution à chaque étape de production des services.

7.2.3.1 Service

À l'exécution, l'ajout d'un nouveau service est pris en compte de façon transparente par les services existants. Les implémentations des services déclarent les ressources nécessaires par rapport à la spécification DIASPEC et sont donc indépendantes des instances de services. Par exemple, si un utilisateur allume son PDA, il recevra sur son PDA des messages l'avertissant des appels manqués (lignes 32 à 37 de la figure 7.6).

7.2.3.2 Implémentation

Une nouvelle implémentation peut être créée et instanciée sans nécessité de modifications des services existants. De manière similaire à l'ajout de services, le code applicatif ne fait pas référence aux instances ou aux implémentations mais aux services DIASPEC. Il prend donc en compte les instances de nouvelles implémentations de façon transparente. Par exemple, une nouvelle implémentation du service DIASPEC `SystèmeLocalisation` peut être programmée pour gérer un nouvel étage de l'immeuble `BâtimentA`. Le gestionnaire des appels manqués recevra alors les événements provenant des instances de cette nouvelle implémentation sans nécessiter de modifications (lignes 7 à 10).

7.2.3.3 Type de services

Les modifications de la spécification DIASPEC nécessitent la regénération du canevas de programmation. À l'instar d'une hiérarchie de classes dans les langages à objets, une modification de la hiérarchie de services DIASPEC peut nécessiter des modifications des implémentations existantes pour être compatibles avec le nouveau canevas de programmation.

Ajout d'un type de services. L'ajout d'un type de services n'a pas de conséquence sur les implémentations existantes si le nouveau type est ajouté en tant que feuille de la hiérarchie des services DIASPEC, comme par exemple, lors de l'ajout d'un type `ÉcranPlasma` étendant le type `Écran` (figures 6.3 et 6.4 aux pages 67 et 68). En revanche, si le nouveau type est inséré comme ancêtre de services DIASPEC existants, les implémentations correspondantes devront être modifiées pour prendre en compte les nouveaux attributs et fonctionnalités hérités du nouveau type. Par exemple, l'ajout d'un type `CapteurMouvement` qui produit des événements `Mouvement` entre les types `Capteur` et `CapteurLocalisation` impliquera de modifier les implémentations de `CapteurLocalisation` pour qu'elles produisent aussi des événements `Mouvement`.

Modification d'un type de services. La modification d'un service DIASPEC existant par l'ajout d'attributs ou de fonctionnalités nécessite la modification des implémentations de ce service DIASPEC et de ses descendants. Par exemple, l'ajout d'un fonctionnalité de commande `ObtenirStatut` au type `EntitéMatérielle` implique l'implémentation de cette commande sur toutes les implémentations de type `EntitéMatérielle` et de ses descendants.

Au contraire, la suppression d'attributs et de fonctionnalités au niveau de la spécification DIASPEC va impliquer l'apparition d'erreurs pour l'enregistrement et de code mort pour les fonctionnalités supprimées. Des analyses des implémentations de services pourraient alors être effectuées pour aider à la maintenance du code applicatif.

7.3 Couche de communication

Un inconvénient récurrent des intergiciels existants est la forte dépendance des interfaces de programmation avec des concepts spécifiques aux technologies sous-jacentes. Non seulement, la complexité des technologies complique la tâche des développeurs mais elle rend délicate la portabilité des applications. L'approche DIAGEN propose un intergiciel dont l'architecture en couches permet de substituer le bus logiciel sans impacter le code applicatif. Une même application peut ainsi indifféremment utiliser les bus logiciels RMI, SIP ou CORBA. Le choix du bus logiciel est alors effectué au déploiement de l'application. L'indépendance par rapport au bus logiciel facilite le test de la logique applicative. Par exemple, en utilisant un bus logiciel dédié à la simulation, une application peut être testée sans modification selon diverses conditions d'exécution.

La plupart des bus logiciels ont des domaines applicatifs de prédilection. Par exemple, SIP et les services Web sont utilisés respectivement pour la téléphonie et le commerce électronique. De ce fait, l'indépendance par rapport à la couche de communication permet de programmer de la même manière des domaines applicatifs pourtant disjoints. Elle favorise ainsi la réutilisation du code applicatif à travers différents domaines applicatifs. Par exemple, le système de localisation de l'application de rappel intelligent peut être utilisé dans une application de domotique pour surveiller les enfants. Enfin, le développeur peut choisir le bus logiciel le plus adapté par rapport au domaine applicatif visé et à ses entités existantes sans avoir à récrire entièrement les applications.

L'extensibilité de la couche de communication permet d'ajouter de nouveaux bus logiciels sans impacter les couches supérieures, c'est-à-dire la couche système et le canevas de programmation. Pour cela, les bus logiciels de la couche de communication doivent s'interfacer sur la couche système et doivent soutenir les mécanismes génériques de la couche système. L'ajout d'un nouveau bus logiciel dans la couche de communication implique alors de combler le fossé existant entre le bus logiciel visé et la couche système DIAGEN. Pour cela, le bus logiciel doit fournir au moins les mécanismes suivants :

1. Le transport de données arbitrairement complexes. DIASPEC permet de décrire des données arbitrairement complexes. Le bus logiciel doit donc proposer un moyen de transporter ces données.
2. Un service de nommage pour abstraire les adresses physiques des services.
3. Un mode d'interaction de type RPC.

Nous avons développé plusieurs bus logiciels au sein de la couche de communication, notamment les bus logiciels Java RMI et SIP [JPCK08]. Nous allons voir plus en détails comment les besoins de l'approche DIAGEN sont satisfaits par ces bus logiciels.

7.3.1 Étude de cas : Java RMI

Java RMI est l'équivalent objet du modèle RPC. Il fournit un service de nommage pour abstraire la localisation physique des services.

Transport des données. RMI fournit un sérialiseur/désérialiseur pour les objets Java et permet ainsi aux données de circuler entre les services de façon transparente pour le développeur. La sérialisation des données utilise le protocole JRMP *(Java Remote Method Protocol)* pour faire transiter les objets et les appels de méthodes sur le réseau.

Commande. RMI propose un mode d'interaction synchrone de type RPC qui est utilisé pour la gestion des commandes.

Événement. RMI ne propose pas de communication asynchrone et *a fortiori*, pas de modèle événementiel. Pour faire dialoguer de façon asynchrone les services avec le courtier d'événements de la couche système, les communications synchrones RMI de la couche de communication sont associées aux threads Java de la couche système.

Session. Les sessions DIAGEN sont construites à partir des communications synchrones RMI.

Découverte de services. RMI propose un service de nommage pour lequel le courtier `rmiregistry` associe un identifiant à une adresse physique. Nous avons développé une surcouche logiciel au `rmiregistry` de RMI pour gérer les couples propriété/valeur des services. Chaque service s'enregistre localement sur un `rmiregistry`. Le courtier de services DIAGEN stocke pour chaque service, ses caractéristiques, son identifiant RMI et l'adresse physique de son `rmiregistry`. À partir des caractéristiques d'un filtre de services, le courtier de services retourne l'ensemble des identifiants et adresses physiques correspondantes. En utilisant ces informations et l'API RMI, le client requête le `rmiregistry` du serveur recherché pour obtenir un mandataire sur les opérations de ce serveur.

7.3.2 Étude de cas : Le protocole SIP

Le protocole SIP propose des modes d'interaction variés : session, événement et envoi de messages synchrones. SIP fournit aussi un service de nommage pour abstraire la localisation physique des utilisateurs ou des services.

Transport des données. Plusieurs propositions ont été faites pour décrire les données des corps des messages SIP. Elles incluent des formats pour décrire les capacités multimédia des services (par exemple SDP [HJ98]), l'invocation des fonctionnalités (par exemple DMP [KGD00]) et les messages instantanés (par exemple CPIM [KA04]). Cependant, ces propositions ne sont pas interopérables et forcent le concepteur de la couche de communication à utiliser de nombreuses API pour sérialiser et désérialiser les corps des messages. Par ailleurs, les formats proposés ne sont pas suffisamment extensibles et ne permettent pas de transporter des données arbitrairement complexes. Pour résoudre ces problèmes, un sérialiseur/désérialiseur qui s'appuie sur le format SOAP est utilisé pour encoder les données dans les messages SIP et les décoder [kso]. SOAP est un format XML standardisé, extensible et très utilisé, notamment dans les services Web.

Commande. SIP propose le message `MESSAGE` pour transporter les messages instantanés [CRS+02]. La couche de communication SIP étend son utilisation pour gérer le mode d'interaction commande. Les commandes et leur valeur de retour sont décrites dans le format SOAP.

Événement. SIP gère le modèle événementiel grâce aux messages `SUBSCRIBE`, `PUBLISH` et `NOTIFY` [Roa02, Nie04]. Il gère principalement les événements de présence. La couche de communication SIP étend son utilisation à tout type d'événements en utilisant le format SOAP.

Session. SIP gère nativement les sessions grâce aux messages `INVITE`, `OK`, `BYE` et `CANCEL` [Ros02]. La négociation s'effectue suivant le modèle offre/réponse [RS02] qui est aussi utilisé par la couche système DIAGEN. Les paramètres de la négociation sont décrits par le protocole SDP [HJ98] *(Session Description Protocol)*. SDP décrit les flux audio et vidéo. Utilisant le format SOAP, la couche de communication SIP généralise SDP à n'importe quel type de données pour gérer notamment des flux de mesures. Le transport des données négociées est ensuite effectué par le protocole RTP.

Découverte de services. De façon similaire à RMI, nous avons développé une surcouche au service SIP de nommage. Un serveur d'enregistrement gère les enregistrements et stocke les associations entre les URI SIP et les adresses physiques. Le courtier de services est notifié par le serveur d'enregistrement chaque fois qu'un service enregistre son URI (message SIP `REGISTER`). Le courtier de services envoie alors une requête SIP `OPTIONS` sur le nouveau service pour obtenir ses caractéristiques. Pour découvrir les serveurs, les clients envoient des opérations `getServices` dans une requête SIP `MESSAGE` contenant le filtre des services recherchés. Lors de l'invocation des services distants, un serveur mandataire résout les URI SIP.

L'intégration du protocole SIP dans la couche de communication respecte l'utilisation des messages telle que définie dans les spécifications du protocole SIP. Le but est ici de permettre l'interopérabilité des services DIAGEN avec les infrastructures SIP existantes, constituées notamment de téléphones [lin, cis], de caméras [sip] et de serveurs gérant la mobilité [ope].

7.4 Bilan

Nous avons présenté dans ce chapitre la structure des canevas de programmation générés depuis une spécification DIASPEC. Nous avons montré comment les canevas de programmation garantissent statiquement la cohérence des applications vis-à-vis de leur spécification DIASPEC. Les vérifications mises en œuvre ne compromettent pas la dynamicité des applications ubiquitaires qui est gérée par les développeurs grâce à l'utilisation de filtres typés. L'intégrité des communications est garantie par notre approche générative. Le compilateur DIASPEC spécialise un intergiciel générique qui gère les mécanismes essentiels à la réalisation d'applications ubiquitaires : la découverte de services et les modes d'interaction commande, événement et session. Le code généré fournit des abstractions haut niveau et typées pour intégrer ces mécanismes dans le développement des applications. Par ailleurs, les canevas de programmation produits abstraient la complexité des technologies sous-jacentes grâce à une architecture en couches de l'intergiciel. Le code applicatif est alors indépendant des bus logiciels, favorisant ainsi sa portabilité et sa réutilisation au travers des divers domaines applicatifs constituant

l'informatique ubiquitaire. De nouveaux bus logiciels peuvent être ajoutés à la couche de communication sans impacter la couche système et les canevas de programmation générés. Le choix du bus logiciel est alors effectué lors du déploiement.

Chapitre 8

Application à la simulation

Les applications ubiquitaires coordonnent une variété d'objets communicants en collectant des informations de contexte produites par des capteurs et en activant des actionneurs. Pour mettre à disposition des informations de contexte, les capteurs traitent de stimuli qui sont des changements observables de l'environnement comme un feu ou un mouvement. Les applications doivent implémenter des stratégies pour gérer un ensemble de scénarios, comme par exemple un départ de feu, une intrusion ou une évacuation d'urgence. Le nombre de paramètres à considérer lors du développement peut alors être très important. Dans les chapitres précédents, nous avons présenté l'approche DIAGEN qui facilite le développement des applications et garantit statiquement leur validité par rapport aux spécifications DIASPEC. Cependant, la logique applicative étant une donnée subjective, la capacité des applications à accomplir leur tâche ne peut être vérifiée statiquement. Par exemple, les développeurs doivent résoudre des conflits potentiels entre les applications développées ; un gestionnaire d'alerte incendie et un gestionnaire de portes pourraient émettre des commandes contradictoires sur la porte d'entrée d'un bâtiment pour respectivement permettre l'évacuation et assurer la sécurité. Il est donc indispensable de procéder à des tests en environnements réels pour valider le comportement des applications par rapport à leur cahier des charges. Ces tests impliquent un processus itératif long et approximatif où les développeurs doivent modifier alternativement la configuration des environnements physiques et le code des applications.

Le développement d'une application ubiquitaire est similaire au développement d'une application d'un objet communicant mobile comme un PDA ou un téléphone. Nous qualifierons ces applications, d'applications mobiles. À l'instar des applications ubiquitaires, les applications mobiles coordonnent des composants matériels hétérogènes qui peuvent être vus comme des capteurs (par exemple microphones et boutons) ou des actionneurs (par exemple écrans et haut-parleurs). Certaines applications mobiles sont capables de découvrir des composants dynamiquement, par exemple, un téléphone intelligent *(smartphone)* peut détecter des composants Bluetooth. Comme pour les applications ubiquitaires, les développeurs doivent anticiper un nombre important de scénarios. Malgré ces similarités, les applications ubiquitaires ne bénéficient pas des mêmes outils pour faciliter leur test. En effet, les applications mobiles sont testées et déboguées dans des simulateurs [emub, emua]. Les composants matériels sont *simulés* via des composants logiciels qui copient fidèlement leur comportement observable. Les composants logiciels des objets communicants mobiles sont *émulés*, s'exécutant sans modification sur les composants matériels simulés.

Nous proposons d'appliquer la simulation, une solution éprouvée par les développeurs d'ap-

plications mobiles, aux systèmes ubiquitaires. Notre objectif est double. D'une part, le domaine de la simulation fournit une illustration pratique de l'approche DIAGEN. D'autre part, la simulation est complémentaire à l'approche DIAGEN car elle permet de tester le comportement des applications à l'exécution. Dans ce chapitre, nous décrivons DIASIM, un simulateur d'applications ubiquitaires. Nous nous appuyons sur l'approche DIAGEN pour spécifier l'architecture logicielle de l'environnement simulé et générer le support de programmation correspondant. DIASIM étend l'intergiciel DIAGEN pour exécuter et visualiser les environnements simulés.

8.1 Analyse du domaine

Les objets communicants mobiles constituent une base pratique pour identifier les besoins de la simulation d'applications ubiquitaires. Dans cette section, nous analysons le domaine applicatif de la simulation d'applications ubiquitaires et nous en déduisons les contraintes que les simulateurs d'applications ubiquitaires doivent respecter.

8.1.1 Simulateurs dédiés

Les systèmes ubiquitaires ciblent un ensemble de domaines applicatifs qui sont instanciés dans une variété d'environnements d'exécution contenant des types de services spécifiques à ces domaines. De même, les types de stimuli varient fortement suivant le domaine applicatif cible. Chaque objet communicant mobile propose un simulateur correspondant à son environnement d'exécution. L'hétérogénéité des environnements ubiquitaires ainsi que le nombre important de scénarios associés nécessitent des outils pour instancier facilement des simulateurs dédiés à un environnement ubiquitaire donné.

8.1.2 Transparence de la simulation

Une caractéristique essentielle des simulateurs d'objets communicants mobiles est la non modification des applications lors des tests en environnements simulés. Une fois la phase de test terminée, l'application est déployée telle quelle sur l'objet communicant mobile. Ne pas modifier le code applicatif facilite les tests et garantit leur précision. Une telle caractéristique doit être fournie par les simulateurs d'environnements ubiquitaires. Pour cela, les environnements ubiquitaires réels et simulés doivent être similaires vis-à-vis des applications ubiquitaires.

8.1.3 Scénarios

Certaines applications ubiquitaires gèrent des scénarios dont la nature des stimuli associés empêchent les tests en environnements réels, comme par exemple les stimuli de feu et de fumée. De façon similaire, certains scénarios sont trop contraignants à cause du nombre de stimuli, du nombre d'objets communicants ou de la superficie des espaces physiques visés. La simulation permet de s'abstraire de ces contraintes et de préciser les besoins en terme d'objets communicants. Par ailleurs, le débogage et la validation de la logique applicative nécessitent de tester les applications au travers d'un nombre important de scénarios. Il faut donc fournir des outils pour créer et reconfigurer rapidement des scénarios de simulation.

8.1.4 Visualisation de la simulation

À l'instar des simulateurs d'objets communicants mobiles, les simulateurs de systèmes ubiquitaires doivent fournir des outils pour visualiser le déroulement de la simulation et ainsi prendre les décisions appropriées quant aux modifications à effectuer sur les applications testées. Le visualiseur doit pouvoir représenter les divers scénarios de simulation impliquant un ensemble ouvert d'entités et de stimuli. Il doit aussi fournir des aides au déboguage.

8.2 Définition des scénarios de simulation

Comme les environnements d'exécution, un environnement simulé est constitué d'un ensemble de services. Ces services simulent les paramètres physiques et les services réels. Ils réalisent ainsi un scénario de simulation. Dans cette section, nous caractérisons les environnements simulés pour pouvoir les spécifier dans le langage DIASPEC et les développer dans les canevas de programmation générés.

8.2.1 Caractérisation des environnements simulés

Nous présentons ici les éléments constitutifs des environnements simulés que nous caractérisons. Pour chacun de ces éléments, nous en décrivons un modèle.

8.2.1.1 Contexte

Les *stimuli* sont les changements de l'environnement observés par les capteurs. D'un point de vue simulation, les stimuli sont produits par des *producteurs de stimuli* et sont consommés par des capteurs, comme par exemple des capteurs de mouvement. Chaque stimulus a un type (par exemple le type mouvement) qui contraint ses valeurs et qui correspond à un ou plusieurs types de capteurs. Un producteur de stimuli représente une source de stimuli qui lui est propre. Cette source peut être un groupe ou une instance d'un objet (par exemple une lampe et un radiateur), d'une personne ou d'un phénomène naturel (par exemple le soleil et un feu). Une source est liée à un unique type de stimuli. Un producteur de stimuli produit ainsi un seul type de stimuli. Une même source de stimuli pouvant être captée par plusieurs capteurs, les producteurs de stimuli sont découplés des capteurs.

Pour une valeur temporelle et une localisation données, un producteur de stimuli donne une valeur du stimulus. L'évolution de la valeur du stimulus peut être modélisée par une fonction mathématique. Cette fonction peut être périodique (par exemple une voiture se déplaçant vers un lieu de travail tous les jours) ou discrète (par exemple une personne de déplaçant d'une pièce à une autre). Ces fonctions sont généralement fournies par les experts d'un domaine applicatif. Par exemple, une approche pour la synthèse d'images modélise la lumière du jour [PSS99]. Les stimuli peuvent aussi être introduits en utilisant des traces de mesures construites à partir d'environnements réels. Par exemple, pour concevoir des immeubles à énergie positive, des mesures sont effectuées et stockées pour la température, la luminosité et le vent sur des périodes d'un an [IG08]. Ces travaux contribuent à construire des bibliothèques de mesures, facilitant la simulation sans compromettre la précision.

Cependant, les traces de mesures ne sont pas disponibles en général. Il est donc nécessaire de définir un modèle pour approximer un environnement réel aussi précisément que nécessaire. Pour y parvenir, un modèle est défini par rapport à la précision des capteurs de

l'environnement cible. Par exemple, les capteurs de localisation peuvent être simulés comme des consommateurs de stimuli dont le type définit des coordonnées cartésiennes `(x,y)`. Si les capteurs de localisation produisent l'information de localisation sous forme de noms de pièces, les producteurs de stimuli correspondants doivent générer des informations avec le même degré de précision, par exemple, un unique point `(x,y)` par pièce.

Certains types de stimuli ont une influence sur l'évolution d'autres stimuli. Par exemple, le feu influence la valeur des stimuli de température et de fumée. Ces stimuli sont qualifiés de *causaux* par opposition aux stimuli *simples* qui sont directement consommés par les capteurs.

8.2.1.2 Environnement d'exécution

L'environnement d'exécution simulé est composé de l'ensemble des services primitifs nécessaires à l'application testée. Les services simulés désignent les services primitifs dont la logique applicative et le support matériel est reproduit par un modèle de simulation. Comme un service primitif réel, un service simulé interagit avec un environnement simulé en traitant des stimuli, en effectuant des actions et en échangeant des données avec d'autres services. Deux familles de services jouent un rôle primordial dans la simulation : les capteurs et les actionneurs. La version simulée d'un capteur reproduit le comportement d'un capteur réel, réagissant aux stimuli produits par les producteurs de stimuli. Par exemple, un détecteur de mouvement simulé ignore les stimuli de mouvement quand il est éteint. Lorsqu'il est allumé, il reçoit des stimuli de mouvement et publie un événement de mouvement. Les actionneurs acceptent généralement des commandes qui modifient leur état ainsi que leur environnement. Par exemple, invoquer un service de lumière pour l'activer change son état et augmente localement la luminosité. La version simulée du service de lumière doit donc maintenir un état (c'est-à-dire allumé/éteint) et créer un producteur de stimuli de luminosité correspondant à la source lumière. Les clients interagissent indifféremment avec des services simulés ou réels. Par exemple, un gestionnaire de lumière actionne des lumières qu'elles soient simulées ou non.

8.2.1.3 Application

Dans le cadre de la simulation, une application est l'ensemble des services à tester. Contrairement aux services simulés, les service testés sont exécutés sans modification. Il s'agit généralement des services de coordination. Comme illustré dans la figure 8.1, une application s'appuie sur l'environnement d'exécution simulé pour recevoir des informations de contexte des capteurs et réagir en envoyant des commandes aux actionneurs.

8.2.1.4 Espace physique

La modélisation de l'espace physique est essentielle à la caractérisation de l'évolution des stimuli et des services. Chaque valeur de stimulus est associée à sa localisation dans l'espace simulé. De même, les services sont positionnés dans l'espace simulé par rapport à une configuration existante (ou souhaitée) à tester. Les services simulés peuvent être mobiles ou apparaître/disparaître dynamiquement. Il est donc nécessaire de pouvoir situer les services dans l'environnement simulé.

L'espace simulé est décomposé en régions. Cette décomposition est hiérarchique, définissant des régions de plus en plus petites. Par exemple, un immeuble est constitué d'étages qui sont eux-mêmes constitués de couloirs et de pièces. Chaque région est associée à la notion de perméabilité définissant comment un type particulier de stimuli se propage entre deux régions.

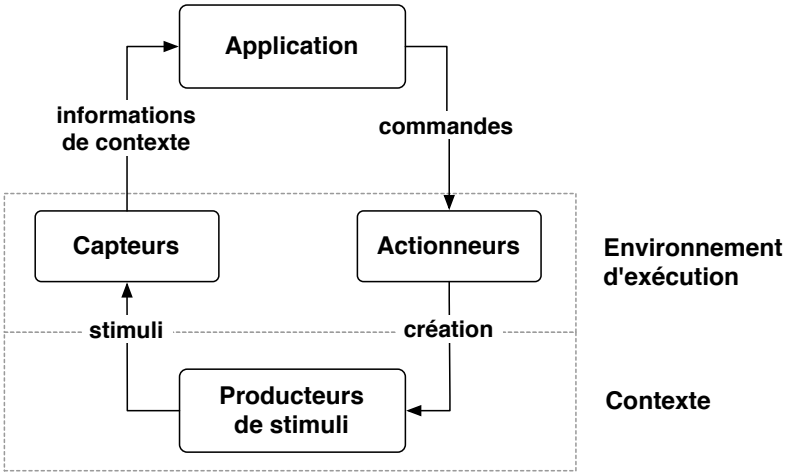


FIG. 8.1: Modèle de simulation

La valeur d'un stimulus est supposée uniforme dans les régions feuilles de la hiérarchie. Par exemple, une lumière qui est allumée dans une pièce définie en tant que région feuille propage sa luminosité de façon uniforme dans cette pièce. Si les murs et portes de la pièce sont déclarés bloquants pour ce type de stimuli, la luminosité ne se propagera pas au-delà de la pièce. La perméabilité permet ainsi d'instancier une même source de stimuli (par exemple le service de lumière) dans plusieurs régions de l'espace simulé sans nécessité de l'adapter explicitement.

8.2.2 Spécification des environnements simulés

Du modèle des environnements simulés, nous pouvons en déduire les contraintes sur les spécifications des environnements simulés. La spécification DIASPEC d'un environnement de simulation repose sur la spécification DIASPEC de l'environnement réel à simuler. Les environnements de simulation résultants ont ainsi les mêmes caractéristiques que leurs homologues réels renforçant la précision de la simulation et permettant aux applications d'être testées sans modification.

Espace Physique. L'espace physique est représenté dans la spécification DIASPEC par un attribut de type `Localisation`. Cette attribut est utilisé par les services DIASPEC et caractérise les stimuli. L'attribut `Localisation` énumère les régions de l'espace physique cible ainsi que les perméabilités de chaque stimulus.

Producteurs de stimuli. Les producteurs de stimuli sont déclarés comme des services serveurs de la commande `getStimuli(Time time)` qui retourne un ensemble de stimuli. Chaque stimulus est composé d'une localisation et d'une valeur. Par exemple, comme déclaré dans la figure 8.2, les producteurs de stimuli de luminosité doivent implémenter une méthode `getStimuli(Time time)` qui retourne un tableau de stimuli de luminosité. Un stimulus de luminosité a une localisation de type `Localisation` et une valeur de type `Luminosité`.

Capteurs simulés. Le type `Luminosité` correspond au type d'événements produits par le capteur de luminosité `CapteurLuminosité` déclaré dans la figure 8.3. Comme illustré dans

```
service ProducteurStimuliLuminosité() {
  provides command IProducteurStimulusLuminosité to [...];
}

icommand IProducteurStimuliLuminosité {
  StimulusLuminosité[] getStimuli(Time time);
}
```

FIG. 8.2: Déclaration d'un producteur de stimuli dans DIASPEC

la figure 8.3, la version simulée du capteur de luminosité `CapteurLuminositéSimulé` étend le capteur de luminosité `CapteurLuminosité` et ajoute une fonctionnalité pour recevoir des stimuli de luminosité.

```
service CapteurLuminosité() extends [...] {
  provides event Luminosité to [...];
}

service CapteurLuminositéSimulé() extends CapteurLuminosité {
  requires event StimulusLuminosité from [...];
}
```

FIG. 8.3: Déclaration d'un capteur simulé dans DIASPEC

Actionneurs simulés. De façon similaire aux capteurs, les actionneurs simulés ajoutent aux actionneurs réels une fonctionnalité pour produire des événements indiquant leur état comme illustré dans la figure 8.4. Dans le cas d'une lumière simulé, cet événement permettra de créer ou supprimer un producteur de stimuli de luminosité.

```
service Lumière() extends [...] {
  provides command ILumière to [...];
}

service LumièreSimulé() extends Lumière {
  provides event Etat to [...];
}
```

FIG. 8.4: Déclaration d'un actionneur simulé dans DIASPEC

Pour faciliter la spécification d'un environnement de simulation, des annotations permettent d'attacher des informations aux noeuds existants des hiérarchies de la spécification DIASPEC de l'environnement réel. À partir de ces annotations, le compilateur DIASPEC ajoute automatiquement les types de stimuli, de producteurs de stimuli et de services simulés à la spécification et génère le support correspondant dans le canevas de programmation.

8.2.3 Développement des environnements simulés

Le canevas de programmation d'un environnement simulé étend celui de l'environnement réel d'exécution correspondant. Le canevas de programmation d'un environnement simulé est

donc compatible avec son homologue réel. Les services développés dans le canevas de programmation d'un environnement réel sont par conséquent compatibles avec l'environnement simulé correspondant. Cette caractéristique favorise la réutilisation des services réels dans l'environnement simulé et permet de tester des services réels sans aucune modification. Par ailleurs, les canevas de programmation des environnements simulés ajoutent du support pour développer des services simulés et des producteurs de stimuli.

8.2.3.1 Développement des services simulés

Le support de programmation pour les services simulés est similaire à celui pour les services réels. Le développeur doit implémenter les fonctionnalités déclarées comme fournies et peut utiliser les fonctionnalités déclarées comme requises. Pour les capteurs simulés, le développeur doit implémenter une méthode `receive` pour traiter les stimuli. Contrairement aux consommateurs réels d'événements, la souscription est automatiquement générée dans le constructeur de la classe abstraite. Pour les actionneurs simulés, une méthode `publish` est définie dans les classes abstraites correspondantes pour rendre compte de leur état. Par exemple, le développeur appelle la méthode `publish` dans un service lumière pour indiquer son activation et la valeur de sa luminosité. Enfin, les canevas de programmation des environnements simulés définissent dans les classes abstraites des appels pour automatiquement rendre compte des interactions entre services dans le but de déboguer et valider le comportement des applications. Pour faciliter le développement des services simulés, une bibliothèque de capteurs et d'actionneurs génériques est fournie. Par exemple, une classe de capteur à seuil envoie des événements dès que la valeur des stimuli est plus haute ou plus basse qu'une certaine limite. Les services simulés et l'intergiciel DIAGEN sous-jacent constitue l'*émulateur* de l'application.

Les classes abstraites des services simulés étendent les classes abstraites des services réels comme déclaré dans la spécification DIASPEC. Grâce à cet héritage, une application peut être exécutée dans des environnements hybrides, contenant à la fois des services simulés et des services réels. En effet, le mécanisme de découverte de services DIAGEN permet aux applications de découvrir des services simulés comme si ils étaient réels. À partir d'un type de services réels (par exemple `CapteurLuminosité`) le courtier de services retourne les services correspondants à ce type de services et à ses descendants, y compris les services simulés si ils existent (par exemple `CapteurLuminositéSimulé`). L'application invoque ensuite le service découvert via un mandataire qui abstrait l'implémentation du service et sa nature simulée. Les applications sont ainsi testées dans un émulateur et déployées dans un environnement réel sans modification.

8.2.3.2 Développement des producteurs de stimuli

La programmation des producteurs de stimuli est effectuée dans le canevas de programmation généré. Elle consiste en l'implémentation de l'unique méthode `getStimuli` déclarée dans la spécification de l'environnement simulé. La logique de cette méthode définit l'évolution d'une source de stimuli. Par exemple, pour simuler un feu gagnant en intensité, un producteur de stimuli augmente graduellement la valeur des stimuli retournés par la méthode `getStimuli`. À l'instar des services simulés, le développement des producteurs de stimuli est facilité par une bibliothèque de fonctions génériques paramétrables.

8.3 Test des applications

Le simulateur DIASIM exécute les environnements simulés, donne un rendu graphique des simulations et aide au déboguage des applications. Dans cette section, nous expliquons comment les applications sont testées dans DIASIM.

8.3.1 Architecture du simulateur

L'architecture du simulateur est illustrée dans la figure 8.5. Elle est composée d'un émulateur pour l'exécution des applications et d'un simulateur de contexte pour gérer les stimuli. Le simulateur de contexte communique les informations de simulation à un moniteur pour la visualisation et le débogage des applications.

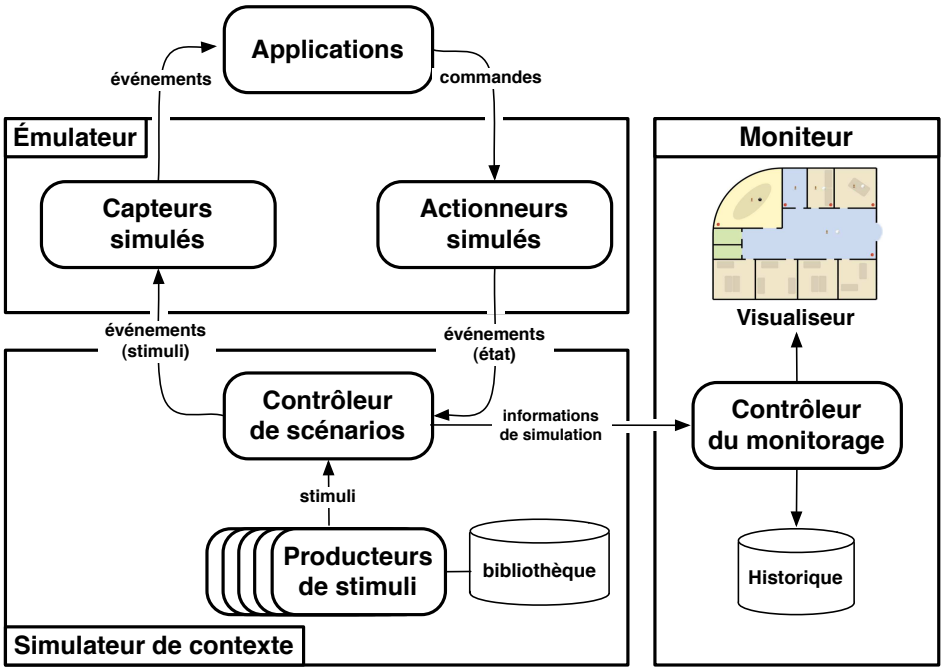


FIG. 8.5: Architecture du simulateur

Exécution des scénarios de simulation. Le simulateur de contexte prend en entrée le scénario de simulation défini précédemment et produit les stimuli correspondants. Il est composé de producteurs de stimuli et d'un *contrôleur de scénarios* qui envoie les stimuli vers les services correspondants. Le contrôleur de scénarios est un médiateur qui requête périodiquement les producteurs de stimuli pour approvisionner les capteurs simulés. C'est un service DIASPEC déclaré fournisseur de stimuli et client des producteurs de stimuli. Il est ainsi implémenté à l'aide du canevas de programmation. Il est aussi déclaré en tant que consommateur d'événement `Etat` provenant des actionneurs. Il crée ou supprime des producteurs de stimuli en fonction de la valeur de cet événement. Par exemple, si un feu est détecté, un arroseur est activé et projette de l'eau. Il en informe le contrôleur de scénarios qui crée un producteur de stimuli `eau`. Dès que l'arroseur est désactivé, le producteur de stimuli `eau` est supprimé par le contrôleur de scénario.

Surveillance des simulations. Les producteurs de stimuli et les services primitifs envoient au contrôleur de scénarios des informations sur la simulation. Ces informations sont envoyées vers le *moniteur* qui donne une représentation graphique de l'environnement simulé. Par l'intermédiaire du moniteur, le développeur peut modifier le déroulement de la simulation en mettant en pause la simulation ou en modifiant à la volée le scénario (par exemple en ajoutant de nouveaux producteurs de stimuli). Au-delà de la représentation graphique de la simulation, DIASIM fournit des fonctionnalités supplémentaires pour aider le développeur à déboguer et valider le comportement des applications.

8.3.2 Support de test

Surveiller une simulation implique de mesurer, collecter et restituer un flux d'informations de simulation. À cause de leur volume, ces informations seraient impossibles à observer si elles n'étaient pas approximées avant d'être restituées. L'environnement simulé est ainsi approximé en temps et en espace. L'approximation en espace fournit une vue idéalisée de l'espace physique, où l'évolution des services primitifs (par exemple une alarme qui sonne, un événement publié) et des stimuli (par exemple une propagation de feu, des déplacements de personnes) est représentée visuellement. Les environnements simulés sont aussi approximés en temps, découplant le temps de visualisation du temps réel de simulation. En effet, le volume d'informations à restituer est bien trop important pour que l'utilisateur puisse tout analyser en temps réel. Pour analyser la séquence des événements menant à une erreur, le moniteur permet de rejouer une partie de la simulation grâce à une fonctionnalité de *time shifting*. Les informations de simulation sont stockées pour constituer un historique de la simulation. À l'image des traces réseaux dans les analyseurs de réseaux [wir], cet historique peut être directement consulté. L'historique de simulation contient des informations concernant les interactions entre les services, notamment le temps, la source, la destination, le type d'interaction et les paramètres d'interaction. Il contient aussi des informations concernant les interactions entre les services et les producteurs de stimuli, notamment le temps, la source, la destination, le type d'interaction et les paramètres d'interaction. Rejouer une simulation permet d'isoler les erreurs mais ne garantit pas que l'application a été corrigée. Seules des conditions déterministes de test permettent de valider le comportement d'une nouvelle version de l'application. Pour cela, un environnement simulé est défini de manière exhaustive, rendant les conditions de test déterministes et reproductibles.

8.4 Mise en oeuvre

Pour illustrer l'approche DIASIM, nous présentons dans cette section la simulation de diverses applications de gestion de bâtiments comme illustrées dans la figure 8.6. DIASIM a permis de valider le comportement de ces applications et la faisabilité d'un déploiement à l'ENSEIRB [1], une école d'ingénieurs en électronique, informatique et télécommunications.

8.4.1 Applications

L'ENSEIRB est un bâtiment de trois étages de 13 500 m^2 composé de plusieurs amphithéâtres, salles de laboratoire et salles de détente. L'ENSEIRB accueille plus de 900 personnes,

[1] École Nationale Supérieure d'Électronique, Informatique et Radiocommunications de Bordeaux, http ://www.enseirb.fr

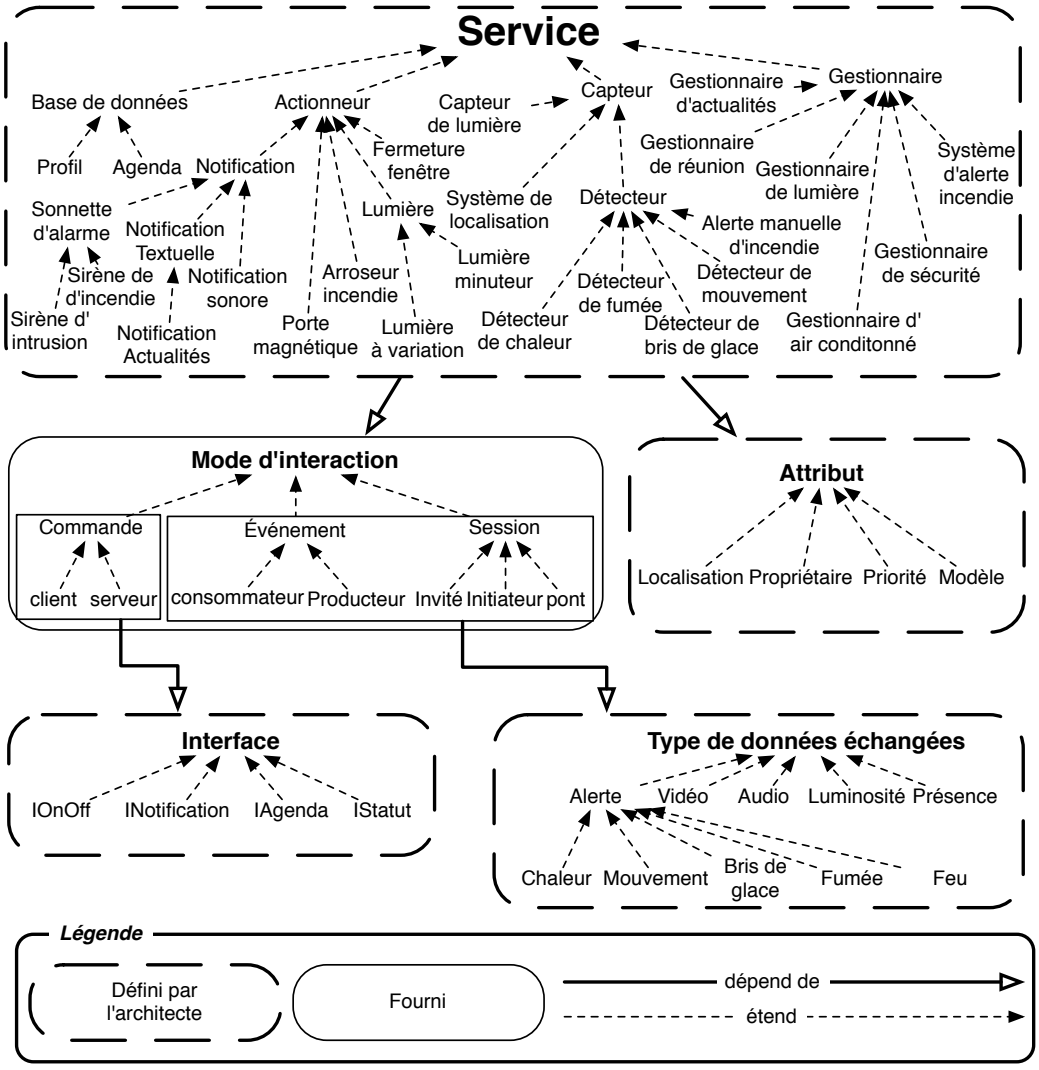


FIG. 8.6: Extrait de la spécification d'un environnement de gestion de bâtiments

incluant des étudiants et enseignants de divers pays.

Nous avons développé plusieurs applications à partir du canevas de programmation généré. L'application de diffusion d'actualités affiche des actualités et les emplois du temps sur des écrans LCD de l'école et adapte le contenu par rapport à l'affiliation et la nationalité des personnes autour des écrans. Le gestionnaire de sécurité alerte le personnel de sécurité lorsqu'une intrusion ou un vol est détecté. Le gestionnaire de réunions notifie les utilisateurs à propos de leurs réunions s'ils ne sont pas présents dans la salle de réunion correspondante. Il affiche aussi des informations à propos des réunions sur les écrans LCD de l'école si elles impliquent des groupes d'étudiants. Le gestionnaire de lumière contrôle les lumières en fonction de la luminosité extérieure, de l'agenda de l'école et de la présence des utilisateurs.

Chacune de ces applications implique au moins un service de coordination illustré par les services DIASPEC étendant le type `Gestionnaire` dans la figure 8.6. Ces services de coordination sont les principales cibles de nos tests. Cependant d'autres services sont testés sans modification dans le simulateur. Il s'agit principalement des interfaces graphiques, comme

par exemple les interfaces Web des services de type `NotificationActualités` qui diffuse les actualités sur les écrans LCD de l'école.

8.4.2 Définition des scénarios de simulation

Pour tester les applications, nous avons simulé trois scénarios : une journée en semaine, une journée en week-end et une journée de rencontre entre des entreprises et les étudiants intitulée *le forum des entreprises*. Chaque scénario est simulé avec des variations, comme par exemple des défaillances de services primitifs ou des variations dans le nombre et la localisation des services primitifs et des personnes. L'*éditeur de scénarios* permet de définir les scénarios dans une interface graphique Java illustrée dans la figure 8.7.

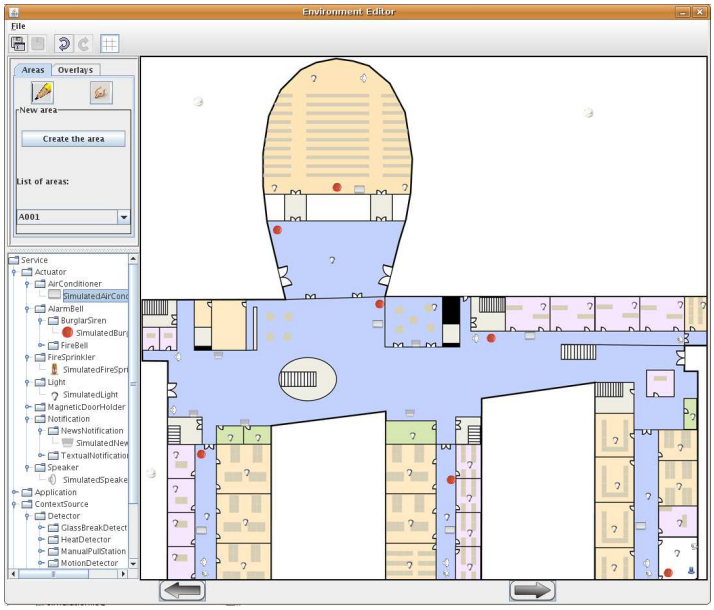


FIG. 8.7: L'éditeur de scénarios (ENSEIRB)

À partir de la spécification DIASPEC, les services simulés sont soient définis graphiquement avec un assistant *(wizard)* ou développés dans le canevas de programmation généré par le compilateur DIASPEC. Dans le premier cas, les attributs des services sont définis dans un formulaire à trous et la localisation est définie en glissant-déposant les icônes des services sur une image représentant l'espace physique. Le comportement des services simulés est ensuite défini graphiquement en sélectionnant et paramétrant une classe de comportement fournie. Par exemple, les services DIASPEC `NotificationSonore` embarqués dans les haut-parleurs sont simulés avec une classe de services de synthèse vocale. Les services DIASPEC `SirèneIntrusion` sont quant à eux simulés avec une classe de lecteur de fichiers audio paramétrée par un fichier audio. Une variété de capteurs, services de notification et lumières ont été simulées. Par ailleurs, des services primitifs réels ont été intégrés dans la simulation soit pour faciliter la définition des scénarios (par exemple un agenda et une base de données de profils) ou pour valider leur comportement (par exemple les services `NotificationActualités` embarqués dans les écrans LCD).

La seconde partie de la définition des scénarios est la configuration des producteurs de stimuli. L'éditeur de scénarios aide à la définition des producteurs de stimuli et de leur com-

portement en permettant à l'utilisateur de définir la valeur des stimuli dans les différentes régions de l'image de l'espace physique à des moments spécifiques. Par exemple, un producteur de stimuli `Mouvement` simule un professeur se déplaçant d'une salle de cours à une salle de détente à 15h30. Le producteur de stimuli de `Luminosité` extérieure est défini à partir d'une fonction définissant l'intensité lumineuse pour une durée de 24 heures. Les producteurs de stimuli simulant la présence des étudiants sont basés sur les emplois du temps.

8.4.3 Observation de la simulation

Le scénario de simulation est sauvegardé dans un fichier XML. Le fichier XML configure le simulateur DIASIM avec le scénario. DIASIM initialise et exécute l'environnement de simulation. Il inclut également un visualiseur, actuellement basé sur le logiciel Siafu [MN06]. Le simulateur DIASIM s'interface avec Siafu pour utiliser ses capacités de rendu. Au dessus d'une image représentant l'espace simulé, le visualiseur affiche les icônes des services et la valeur des stimuli comme illustré dans la figure 8.8.

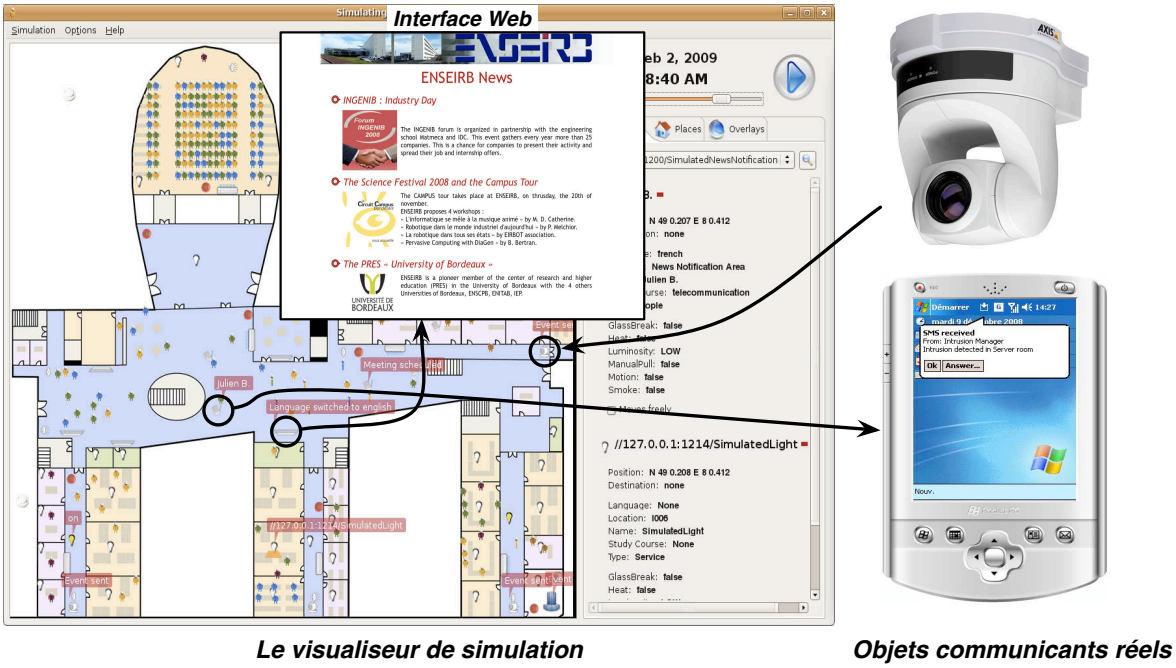


FIG. 8.8: Environnement simulé hybride

Le visualiseur de simulation affiche l'état des services primitifs sur des bulles de texte au-dessus des services (par exemple quand les capteurs publient des événements) et/ou en modifiant l'icône représentant le service (par exemple une lumière jaune pour une lumière activée). Pour compléter les vues macroscopiques, nous avons ajouté aux capacités de rendu de Siafu la possibilité d'afficher des interfaces graphiques Java et Web ainsi que la lecture des flux audio. Ces vues enrichies sont particulièrement utiles pour les services réels reposant sur des interfaces graphiques. Par exemple, dans la simulation de l'ENSEIRB, cliquer sur les écrans LCD lance l'interface Web des services réels `NotificationActualités` (Figure 8.8). Nous avons aussi utilisé les vues enrichies pour des services simulés, par exemple cliquer sur les haut-parleurs diffuse le flux audio correspondant.

DIAGEN permet à DIASIM de construire les environnements simulés comme des extensions des environnements réels. Grâce à cette relation d'héritage, une application peut être exécutée dans un environnement simulé hybride comme illustré dans la figure 8.8. Le scénario de simulation peut par exemple intégrer un PDA pour recevoir des messages d'alerte et une caméra pour envoyer des événements de détection de mouvements. Cette particularité est particulièrement utile pour effectuer des tests unitaires de services réels. De plus, elle permet d'intégrer les services réels au fur et à mesure de leur développement rendant la simulation d'autant plus réaliste.

8.5 Bilan

Dans ce chapitre, nous avons présenté DIASIM, une approche pour tester des applications DIAGEN dans des environnements simulés. Alors que DIAGEN permet de garantir à la compilation la validité des applications ubiquitaires par rapport à leurs spécifications DIASPEC, DIASIM permet de tester le comportement des applications à l'exécution en s'abstrayant des contraintes des environnements physiques. DIASIM s'appuie sur l'approche DIAGEN pour la spécification des environnements simulés et pour la spécialisation du simulateur DIASIM par rapport à ces environnements. En plus de rendre le simulateur DIASIM extensible, DIAGEN permet à DIASIM de tester les applications sans les modifier et permet d'intégrer, sans modification, des services réels dans les scénarios de simulation. Une application peut être exécutée dans des environnements hybrides, contenant à la fois des services simulés et des services réels. Le simulateur DIASIM permet la visualisation des environnements ubiquitaires simulés et facilite le débogage des applications testées.

Chapitre 9

Validation

Dans ce chapitre nous décrivons l'implémentation de notre approche, analysons le code généré et comparons l'approche DIAGEN aux solutions existantes. Nous analysons notamment la nature et la taille du code généré. Nous comparons ensuite les caractéristiques de notre approche et la concision des applications développées aux solutions existantes. Enfin, nous listons les limitations de notre approche.

9.1 Implémentation

La chaîne de production des services DIAGEN est représentée dans la figure 9.1. Elle est fondée sur le langage Java à l'exception des spécifications d'architectures, qui reposent sur le langage DIASPEC, une extension du langage Java.

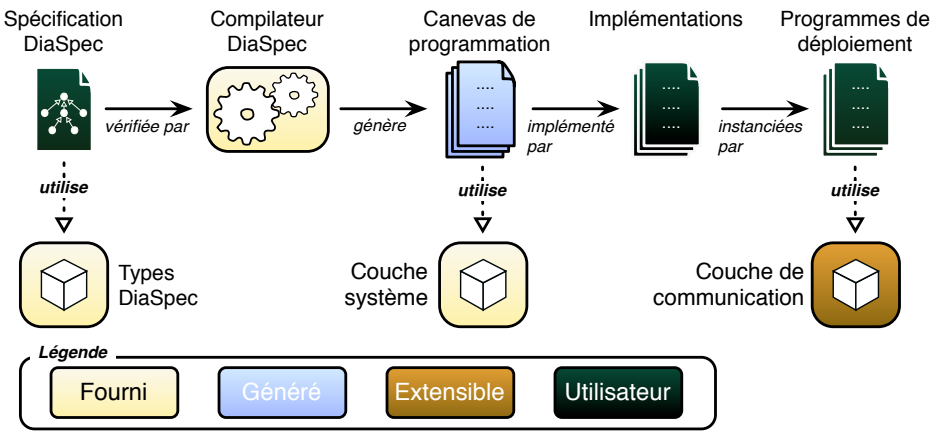


FIG. 9.1: Chaîne de production des services DIAGEN

9.1.1 Compilateur DIASPEC

Le compilateur DIASPEC prend en entrée une spécification écrite dans le langage DIASPEC qui est une extension du langage Java. Les types de services, leurs fonctionnalités et les interfaces des commandes sont définis dans le langage DIASPEC, alors que les attributs et les types de données échangées sont directement déclarés dans le langage Java. Le compilateur

DIASPEC intègre les outils JastAdd [HM03, EH07b] et JastAddJ [EH07a]. Ces outils permettent la manipulation de l'arbre abstrait des programmes Java. Ils facilitent l'introduction de nouvelles constructions dans le langage et de nouvelles analyses statiques associées à ces constructions. Le compilateur DIASPEC analyse ainsi syntaxiquement une spécification, vérifie sa cohérence et génère plusieurs classes et interfaces Java pour chaque service DIASPEC.

9.1.2 Environnement de développement

Le code généré est constitué de structures typées afin de tirer profit des environnements de développement intégrés comme Eclipse et d'assister les développeurs. Notamment, les IDE s'appuient sur les classes abstraites nécessaires à l'implémentation des services pour fournir par génération ou complétion le code nécessaire à la définition des services et à leur interaction. De façon similaire, la définition des filtres est guidée par la complétion qui indique les attributs autorisés. Enfin, le code généré étant typé, l'analyse dynamique de types des IDE indique instantanément si les types utilisés sont autorisés, par exemple, lors de l'affectation des valeurs des attributs.

9.1.3 Environnement de déploiement et d'exécution

Trois bus logiciels sont actuellement disponibles au sein de la couche de communication : un bus logiciel RMI, un bus logiciel SIP et un bus logiciel *local*. Un quatrième bus logiciel fondé sur les services Web a été partiellement implémenté. Le bus logiciel est spécifié dans les *programmes de déploiement* qui instancient l'ensemble des services nécessaires sur chaque objet communicant. Par exemple, le programme de déploiement d'un PDA pourrait instancier des services de notification et de contrôle à distance. Comme pour les services, les courtiers de services et d'événements sont instanciés dans un programme de déploiement où est spécifié le bus logiciel utilisé. De même, le simulateur spécifie le bus logiciel utilisé au déploiement. Pour des raisons de performance, le simulateur utilise le bus logiciel local lorsque cela est possible. Le bus logiciel RMI peut être aussi utilisé pour intégrer des services réels et pour distribuer la charge sur plusieurs machines quand de nombreux services sont nécessaires. Les bus logiciels sont, pour l'instant, mutuellement exclusifs.

9.2 Analyse des canevas de programmation

Nous avons spécifié plusieurs applications dans le langage DIASPEC et généré les canevas de programmation correspondants. Dans cette section, nous analysons la nature et la taille du code généré.

9.2.1 Code généré

Pour chaque service DIASPEC, le compilateur génère une classe abstraite pour l'implémentation du service, des classes mandataires pour son invocation et des classes filtres pour sa découverte. La figure 9.2 donne un aperçu des classes et interfaces générées pour le service DIASPEC `GestionnaireAppelsManqués`. Le paquetage `local` contient les classes abstraites nécessaires à l'implémentation des services. Chacune de ces classes abstraites contient des méthodes abstraites et concrètes pour implémenter ou appeler les fonctionnalités définies dans le service DIASPEC correspondant. Les méthodes concrètes sont directement utilisées par les

développeurs et permettent de découvrir des services (par exemple `getIAfficherAppels-ManquésServices`), de souscrire à un type d'événements (par exemple `subscribePrésenceBureau`), de publier un événement et de gérer une session entre deux services (par exemple `bindAudio`, `disconnect`). Les méthodes concrètes sont implémentées par les développeurs et représentent les interfaces de commande fournies (par exemple `IGérerAppelsManqués`), la réception d'événements (par exemple `receive`) et la connexion et déconnexion d'une session pour les services invités et invitants.

Le compilateur DIASPEC génère du code pour permettre aux services dédiés d'interagir avec les courtiers de services et d'événements génériques. Par exemple, la classe abstraite des consommateurs d'événements contient la méthode concrète générique `eventReceived` qui est invoquée par le courtier d'événements pour appeler ensuite la méthode `receive` implémentée par l'utilisateur et dédiée à un type d'événements. De façon similaire, l'envoi des requêtes d'enregistrement, de découverte, de souscription et de publication est effectué par le développeur via une méthode dédiée qui invoque, dans la classe abstraite, une méthode générique implémentée par les courtiers.

Le compilateur DIASPEC génère dans le paquetage `proxy` les classes mandataires permettant l'invocation des commandes. Les classes mandataires et les classes abstraites permettent d'abstraire le code applicatif de la complexité des bus logiciels sous-jacents et de rendre les canevas de programmation indépendants de la couche de communication.

Enfin, le compilateur DIASPEC génère dans le paquetage `filter` les classes filtres permettant aux développeurs de spécifier des requêtes de services statiquement vérifiées.

9.2.2 Nature du code généré

Nous analysons la nature du code généré à partir de trois spécifications. La première spécification décrit une application de notification de réunions qui notifie les utilisateurs d'une réunion imminente s'ils ne sont pas encore dans la salle de réunion correspondante. Elle est composée d'un service d'annuaire, de services de notification, d'un agenda et d'un système de localisation. La deuxième spécification correspond à l'application de rappel intelligent présentée dans le chapitre 6. La dernière spécification correspondant aux applications d'immotique décrites dans le chapitre 8. Ces trois spécifications sont caractérisées dans la figure 9.3 et déclarent respectivement 6, 11 et 33 services. Le nombre de fonctionnalités (requises et fournies) par service augmente avec le nombre de services et montre la complexité des services impliqués. Il est aussi à noter que seule la spécification de l'application de rappel intelligent déclare des fonctionnalités de session.

La figure 9.4 montre la proportion de code généré à partir des trois spécifications pour la découverte de services et les modes d'interaction, c'est-à-dire commande, événement et session. La découverte de services correspond à la majorité du code généré, et ce, quelque soit le domaine applicatif ou la taille de la spécification. Pour les applications, nous obtenons respectivement que 75%, 65% et 69% du code généré est consacré à la découverte de services. Le code généré pour la découverte de services permet non seulement la construction des requêtes de découverte de services, de souscription et de connexion en session, mais aussi le respect des visibilités déclarées dans les spécifications DIASPEC grâce aux filtres typés. La découverte de services étant étroitement liée aux fonctionnalités déclarées dans les spécifications DIASPEC, la taille du code généré correspondant varie proportionnellement à la taille du code généré pour la gestion des modes d'interaction. La programmation des interactions entre services fait intervenir notamment les mandataires, les méthodes de publication et de réception d'événe-

```
1  service  Service(Proprietaire proprietaire) {
2  }
3
4  service  Gestionnaire(Priorite priorite) extends  Service {
5    provides command ActiverDesactiver to GestionnaireServices;
6  }
7
8  service  GestionnaireAppelsManques extends  Gestionnaire {
9    requires event PresenceBureau from SystemeLocalisation;
10   requires command AfficherAppelsManques from Ecran;
11   provides command GererAppelsManques to TelephoneVirtuel;
12   binds session Audio from (Telephone, Telephone);
13 }
14
15 icommand  GererAppelsManques {
16  void ajouterAppelManque(Utilisateur appelant, Utilisateur appele);
17  void supprimerAppelManque(int appelManqueID);
18 }
19
20 icommand  ActiverDesactiver {
21  void activer();
22  void desactiver();
23 }
```

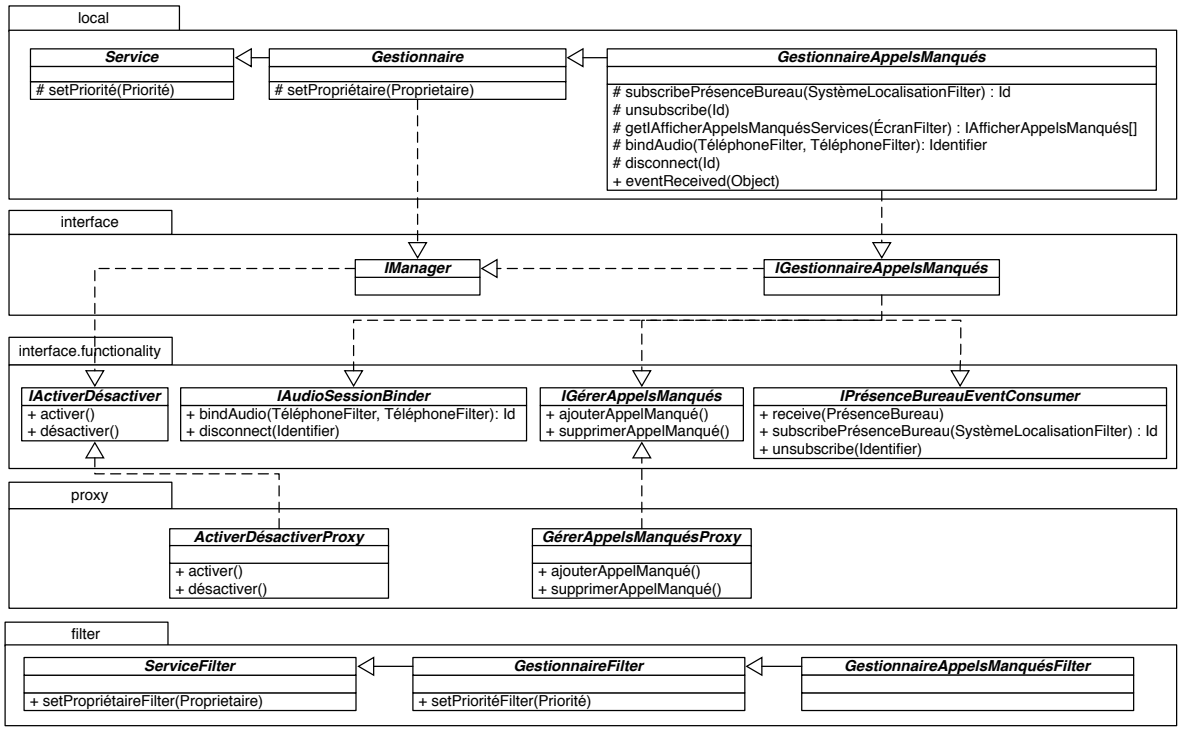


FIG. 9.2: Classes et interfaces générées pour le service DIASPEC GestionnaireAppelsManqués

	Services	Fonctionnalités par service (déclarées et héritées)	Attributs par service (déclarées et héritées)	Fonctionnalités de type commande	Fonctionnalités de type événement	Fonctionnalités de type session
Application de notification de réunion	6	1,33	0,17	6	2	0
Application de rappel intelligent	11	3	2,25	9	5	4
Applications d'immotique	33	4,32	2,12	40	26	0

FIG. 9.3: Caractérisation de trois spécifications DIASPEC

ments ainsi que les méthodes de gestion des sessions. Enfin, les proportions de code généré pour les modes d'interaction commande, événement et session illustrent la taille de la couche d'abstraction fournie par le canevas de programmation pour chacun de ces modes. Ainsi, nous pouvons observer que la proportion de code dédié aux événements est supérieure à celle des commandes pour deux de nos cas d'étude, et ceci malgré le fait que le nombre de fonctionnalités commande soit bien supérieur au nombre de fonctionnalités événement : 9 contre 5 et 40 contre 26 (Figure 9.3). De même, dans le cas de l'application de rappel intelligent, la proportion de code généré pour les fonctionnalités session est supérieure à celles des fonctionnalités commande et événement réunies : 18% contre 8% et 9% dans la figure 9.4, 4 contre 5 et 9 dans la figure 9.3. De ces constations, nous pouvons conclure que le mode d'interaction session est plus complexe à abstraire, car plus riche, que les deux autres modes d'interaction.

Nous pouvons tirer la même conclusion pour le mode d'interaction événement vis-à-vis du mode d'interaction commande.

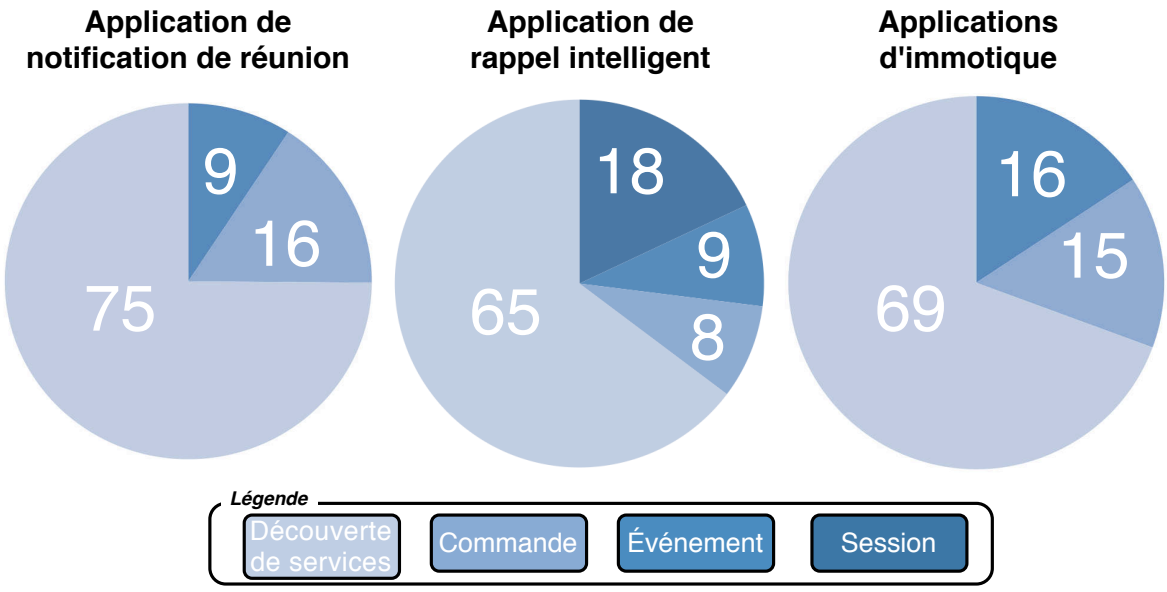


FIG. 9.4: Nature du code généré

9.2.3 Taille du code généré

À partir des trois spécifications décrites précédemment, nous avons généré les canevas de programmation correspondants. La taille du code généré est comptabilisée dans la figure 9.5. En dépit des différences en terme de complexité et de nature des applications modélisées, les ratio entre les spécifications et les canevas de programmation générés sont très proches. Ils sont même pratiquement identique si nous considérons uniquement le nombre de lignes. Ces ratio sont de quatorze et signifient que pour une ligne de spécification DIASPEC, quatorze lignes de code sont générées dans le canevas de programmation. Si nous considérons le nombre de caractères, nous pouvons établir une constatation similaire avec un ratio d'un moins vingt-quatre. Au delà des vérifications sur les spécifications, ces chiffres prouvent l'utilité d'un compilateur pour automatiser la production de canevas de programmation d'applications susceptibles d'évoluer rapidement.

		Spécification	Canevas de programmation	Ratio
Application de notification de réunion	Nombre de lignes	**30**	**412**	**14**
	Nombre de caractères	**1 051**	**26 256**	**25**
Application de rappel intelligent	Nombre de lignes	**62**	**854**	**14**
	Nombre de caractères	**2 365**	**68 832**	**29**
Applications d'immotique	Nombre de lignes	**179**	**2 469**	**14**
	Nombre de caractères	**7 005**	**167 459**	**24**

FIG. 9.5: Ratio entre les spécifications et les canevas de programmation générés

9.3 Comparaison avec les approches existantes

Nous avons développé une application de gestion de réunions à partir du canevas de programmation généré depuis la spécification décrite dans la section précédente. Dans le but d'évaluer la concision des applications développées avec l'approche DIAGEN, nous avons développé cette même application à l'aide de différentes solutions existantes. Le tableau de la figure 9.6 récapitule les caractéristiques de ces solutions telles qu'introduites dans le chapitre 4. L'application de gestion de réunions nécessite de connecter dynamiquement les services qui la composent et notamment de découvrir les services de notification pour notifier uniquement les personnes qui ne sont pas dans la salle de réunion lors du début de la réunion. Elle nécessite aussi des communications asynchrones pour l'envoi d'événements de `Réunion` par l'agenda.

Une surcouche logicielle a été développée pour réaliser la découverte de services et les communications asynchrones lorsque les solutions existantes le permettaient. Ainsi, nous avons développé une surcouche gérant la découverte de services non typée en RMI et les communications asynchrones en RMI et Jini. CORBA étant déjà doté des caractéristiques nécessaires à la réalisation de l'application aucune surcouche n'a été nécessaire. Pour ArchJava et CCM,

l'ajout d'un mécanisme de découverte de services compromettrait l'approche originale qui permet respectivement de garantir l'intégrité des communications et de connecter statiquement des composants. La comparaison avec les solutions existantes reste cependant pertinente puisque DIAGEN fournit davantage de vérifications statiques (c'est-à-dire l'intégrité des communication et la découverte de services typée) et fournit l'ensemble des caractéristiques nécessaires à la réalisation complète de l'application cible (c'est-à-dire application distribuée, découverte de services et communications asynchrones). De plus, l'application cible n'a pas été choisie pour exploiter particulièrement notre approche puisqu'elle contient une proportion faible de fonctionnalités de type événement et qu'elle n'inclue pas de fonctionnalité de type session (Figure 9.3).

	RMI	Jini	CORBA	ArchJava	CCM	DiaGen
Technologie distribuée	+	+	+	-	+	+
Découverte de services	-	+	+	-	-	+
Communication asynchrone	-	-	+	-	+	+
Cohérence des spécifications	-	-	-	-	-	+
Intégrité des communications	-	-	-	+	-	+
Découverte de services typée	-	-	-	-	-	+
Indépendance des bus logiciels	-	-	-	-	-	+

Fig. 9.6: Comparaison qualitative

Les nombres de lignes des spécifications, des surcouches et du code applicatif pour les différentes approches sont répertoriés dans la figure 9.7. Si l'on considère le code applicatif, l'application développée dans DIAGEN est plus concise que toutes les autres approches. La spécification est aussi plus concise comparée aux solutions existantes à l'exception de ArchJava dont la spécification est mélangée dans le code applicatif. Cette concision s'explique notamment par le fait que la spécification est écrite dans un seul fichier ce qui permet des factorisations de code contrairement à RMI et Jini et par le fait que l'approche DIAGEN n'intègre pas le concept de maisons contrairement à CCM. Enfin, DIAGEN permet de produire des applications dynamiques plus fiables et plus concises.

9.4 Limitations

Les connexions entre services sont effectuées dans le code applicatif des services. Contrairement aux approches à composants comme CCM, la séparation entre les parties fonctionnelles du services et ses parties non fonctionnelles n'est donc pas réalisée. Cette limitation de notre approche impacte la maintenance des applications complexes et la réutilisation des services. Cependant, DIAGEN cible plus particulièrement des applications distribuées et des environnements dynamiques alors que les approches à composants nécessitent, dans la plupart des cas, une adaptation non triviale pour développer des applications distribuées sans pour autant pouvoir résoudre la dynamicité des environnements. Nous devons alors nous demander s'il est possible de séparer les parties fonctionnelle (c'est-à-dire l'implémentation des services) et non

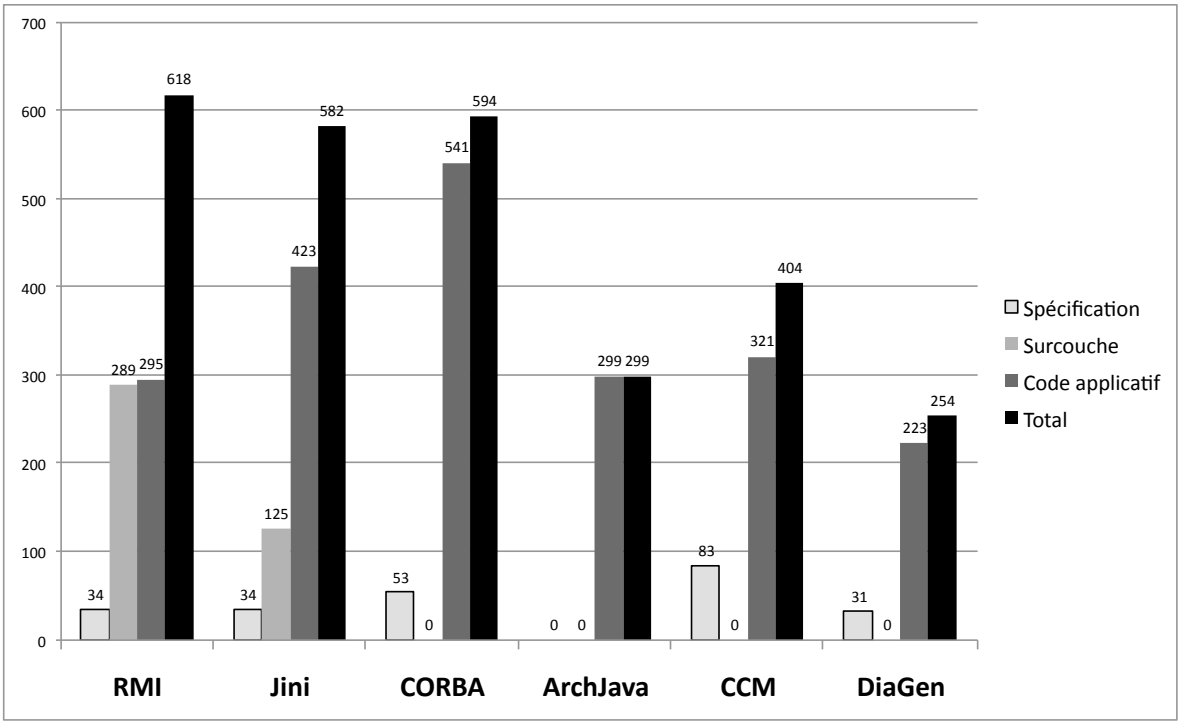


FIG. 9.7: L'application de notification de réunions en nombre de lignes

fonctionnelle (c'est-à-dire la connexion des services) tout en gérant la dynamicité des environnements ubiquitaires. Pour réaliser une application ubiquitaire, les services de coordination doivent dynamiquement modifier les connexions des services en réaction à un changement de contexte matérialisé par la réception d'un événement. Par exemple, dans DIAGEN, à la réception d'un événement (c'est-à-dire correspondant à la partie fonctionnelle), un service de coordination modifie les connexions entre services (c'est-à-dire correspondant à la partie non fonctionnelle) pour adapter dynamiquement le comportement de l'application concernée. Dans une approche à composants comme CCM, ArchJava ou Fractal [BCS04], la notion de services de coordination n'est pas concevable. Notre approche résulte donc d'un compromis entre la nécessité de gérer la dynamicité des environnements ubiquitaires et la nécessité de vérifier statiquement les connexions.

Les premiers tests de performance ont montré que l'approche DIAGEN, configurée avec le bus logiciel RMI avait un temps de réponse supérieur comparé à un intergiciel RMI. Pour effectuer cette comparaison, l'application de notification de réunions décrites dans la section précédente a été utilisée. Le temps entre l'envoi d'un événement indiquant une nouvelle réunion et la notification effective des personnes a été mesuré. Les services composant l'application ont tous été exécutés sur la même machine pour abstraire les contraintes réseaux. Les temps moyens mesurés sont de 14,60 millisecondes pour l'intergiciel RMI (écart type de 0,63 pour 750 mesures) contre 20,55 millisecondes pour l'application s'exécutant dans l'intergiciel DIAGEN (écart type de 0,15 pour 200 mesures). Cette perte de performance est due à la couche système responsable de l'indépendance du code applicatif par rapport au bus logiciel. Ce code contient des appels de méthodes supplémentaires pour adapter dynamiquement le code applicatif au bus logiciel choisi. Les tests ont été réalisés sur la première version de DIA-

GEN intégrant l'indépendance de la couche de communication. Cette version a été réalisée comme une preuve de faisabilité et non pas dans une optique d'optimisation des performances. Enfin, il est intéressant de noter que les vérifications statiques fournies par les canevas de programmation générées n'impactent pas les performances puisqu'elles sont effectuées à la compilation.

9.5 Bilan

Dans ce chapitre, nous avons analysé les spécifications DIASPEC, le code généré et le code applicatif. Nous avons montré l'utilité du compilateur DIASPEC pour l'analyse, la vérification et l'automatisation de la production des canevas de programmation. Nous avons comparé l'approche DIAGEN par rapport aux approches existantes. Nous en avons conclu que les applications distribuées développées avec l'approche DIAGEN sont plus concises. Cette concision ne se fait pas au détriment de la fiabilité des applications ou de leur dynamicité. Comparée aux approches existantes, DIAGEN permet davantage de vérifications statiques tout en gérant la dynamicité des environnements ubiquitaires. Au-delà des applications présentées dans ce chapitre, plusieurs applications ont été développées dans le cadre du projet de domotique HomeSIP en collaboration avec France Telecom. De plus, l'approche DIAGEN fait office de support pour plusieurs autres approches dont le simulateur d'environnements ubiquitaires DIASIM, le langage dédié Pantaxou [MPCL08, Pal08] et le langage graphique Pantagruel [1].

[1] http ://phoenix.labri.fr/projects/pantagruel.html

Chapitre 10

Conclusion

L'informatique ubiquitaire est une évolution naturelle de l'informatique. À l'origine, les objets communicants devaient être partagés entre les utilisateurs. Jusqu'à récemment, chaque utilisateur avait à sa disposition un voir deux objets communicants, le plus souvent un ordinateur ou un téléphone. Avec leur banalisation dans notre vie quotidienne, la proportion d'objets communicants liés à chaque utilisateur augmente progressivement et ouvre de nouvelles opportunités d'applications. Ces applications coordonnent des objets communicants hétérogènes, mobiles, volatiles et potentiellement invisibles pour les utilisateurs. Elles investissent le quotidien des utilisateurs pour automatiser des tâches potentiellement critiques. Elles remplacent alors les capacités sensorielles et décisionnelles des utilisateurs pour capter les changements de contexte et s'adapter en conséquence. Ces applications sont déjà une réalité avec des expérimentations dans la ville ubiquitaire de Hwaseong-Dongtan [ubi], dans le CHU de Nice [IBM06] ou un supermarché Stop & Shop aux États-Unis [IBMb] qui préfigurent l'avènement de l'informatique ubiquitaire dans un future proche. Moins lointain de nous, les technologies de communication machines à machines sont déjà pleinement fonctionnelles et permettent l'automatisation des chaînes d'approvisionnement, de la télésurveillance et du télépaiement. Elles mettent en place des solutions ponctuelles et statiques qui ne sont pas réutilisables dans d'autres domaines applicatifs. L'essor de la domotique et de l'immotique montrent le besoin d'infrastructures logicielles permettant l'intégration d'un flot constant d'objets communicants et le développement et déploiement d'une variété d'applications. Les nouveaux domaines applicatifs demandent de garantir le comportement des applications avant leur déploiement en situation réelle tout en permettant de gérer la dynamicité des environnements physiques. Les approches existantes ne parviennent pas à atteindre ces deux objectifs.

Nous avons présenté dans cette thèse une approche déclarative pour la génération de canevas logiciels ubiquitaires. Cette approche repose sur des spécifications haut niveau de l'ensemble des types de services constituant les applications cibles. Ces spécifications, écrites dans le langage DIASPEC, sont analysées, vérifiées et compilées par le compilateur DIASPEC qui génère des canevas de programmation dédiés. Ces canevas de programmation fournissent du support pour la programmation d'applications distribuées, garantissent l'intégrité des communications et permettent de gérer la dynamicité des environnements ubiquitaires. Par ailleurs, l'architecture en couche de l'intergiciel générique permet l'indépendance des applications développées vis-à-vis des bus logiciels sous-jacents.

Nous dressons dans ce chapitre un bilan des différentes contributions de cette thèse, puis présentons des perspectives à notre approche.

10.1 Contributions

Les contributions de cette thèse se décomposent en plusieurs volets : le langage déclaratif DIASPEC pour la modélisation des applications cibles, les canevas de programmation générés par le compilateur DIASPEC et le simulateur d'environnements ubiquitaires DIASIM fondé sur l'approche DIAGEN.

DIASPEC. Le langage DIASPEC est un langage déclaratif pour la spécification d'architectures logicielles ubiquitaires. DIASPEC permet de décrire les types de services constituant les applications cibles, leur connexion, leur mode d'interaction et leur attribut. Il permet de déclarer des interactions de types commande, événement et session. Le compilateur DIASPEC vérifient la cohérence des spécifications et génère, à partir de celles-ci, des canevas de programmation dédiés.

Canevas de programmation dédiés. Les canevas de programmation générés aident au développement d'applications ubiquitaires statiquement valides vis-à-vis des spécifications DIASPEC. Pour cela, le compilateur DIASPEC spécialise les mécanismes génériques de découverte et d'interaction en produisant du code haut niveau et typé. Les canevas de programmation générés garantissent ainsi l'intégrité des communications d'applications dynamiques. Par ailleurs, nous avons conçu un intergiciel générique pour exécuter les applications développées. Cet intergiciel a une architecture en couche, rendant les canevas de programmation indépendants des bus logiciels. Le bus logiciel peut être ainsi substitué sans impacter le code applicatif. Trois bus logiciels sont actuellement disponibles : un bus logiciel RMI, un bus logiciel SIP et un bus logiciel local.

DIASIM. Nous avons développé DIASIM, un simulateur pour tester, à l'exécution, le comportement des applications développées avec l'approche DIAGEN. DIASIM permet de valider le comportement des applications à l'exécution tout en s'abstrayant des contraintes physiques. DIASIM s'appuie sur l'approche DIAGEN pour la spécification des environnements simulés et la construction d'environnements de simulation dédiés. En plus de rendre le simulateur DIASIM extensible, DIAGEN permet à DIASIM de tester les applications sans les modifier et permet d'intégrer, sans modification, des services réels dans les scénarios de simulation. Le simulateur DIASIM permet la visualisation des environnements ubiquitaires simulés et facilite le débogage des applications testées.

Validation. Nous avons montré l'utilité du compilateur DIASPEC pour l'analyse, la vérification et l'automatisation de la production des canevas de programmation. Nous avons montré que les applications DIAGEN sont plus concises que les applications développées dans les approches existantes. Comparée aux approches existantes, DIAGEN permet davantage de vérifications statiques tout en gérant la dynamicité des environnements ubiquitaires.

10.2 Perspectives

Aspects non fonctionnels. Nous envisageons d'enrichir le langage DIASPEC pour spécifier des aspects non fonctionnels. Par exemple, nous pourrions spécifier des paramètres de sécurité et de qualité de services que devront respecter certaines connexions. Le compilateur DIASPEC

générerait le code nécessaire au respect de ces contraintes. La gestion des conflits d'accès aux ressources est un problème essentiel dans des environnements où les utilisateurs et les applications se partagent un grand nombre de ressources. De même, le langage DIASPEC pourrait être enrichi pour spécifier le nombre de connexions simultanées que supporte un service. Les travaux sur la programmation orientée aspects pourraient être utilisés pour simplifier l'introduction des aspects non fonctionnels dans les applications [WJD03, DEM02, JPS07]. Enfin, un mécanisme de reconfiguration dynamique pourrait être intégré [KkK06]. Il profiterait alors des filtres DIAGEN pour remplacer dynamiquement les services découverts devenant indisponibles.

Gestion du contexte. L'introduction de la gestion du contexte dans les canevas de programmation générés permettrait de souscrire à des changements de contexte au lieu des événements actuelles. Les changements de contexte correspondent à l'agrégation de plusieurs événements dotés de valeurs spécifiques. Les règles ECA *(Event Condition Action)* permettent de modéliser les changements de contexte [CPvS05]. Plusieurs approches utilisent ces règles pour créer des applications ubiquitaires [ECB06, JPHL07] mais aucune, à notre connaissance, ne permet de vérifier statiquement la validité des souscriptions aux règles ECA. En enrichissant le langage DIASPEC, nous pourrions générer du support de programmation typé pour créer des règles ECA dans les canevas de programmation générés.

Déploiement et cycle de vie. Le déploiement d'applications distribuées nécessite la distribution de services et la gestion de leur cycle de vie. Le canevas de déploiement et d'exécution OSGi [osg] répond en partie à ces exigences en permettant de gérer le cycle de vie des services à distance. Par ailleurs, OpenCCM permet de distribuer les composants d'une application en téléchargeant le code applicatif des implémentations de composants sur les objets communicants correspondants. Une perspective serait donc de faciliter le déploiement et la gestion du cycle de vie des services DIAGEN en s'inspirant ou en réutilisant les solutions proposées par OSGi et OpenCCM. Pour cela, nous pourrions nous référer à l'approche FROGi qui propose une intégration du modèle à composants Fractal dans OSGi [HMD]. Par ailleurs, l'utilisation de l'éditeur de scénarios du simulateur DIASIM pourrait permettre de définir graphiquement la configuration des services dans un environnement physique donné et ainsi faciliter leur déploiement.

Optimisation. L'intergiciel DIAGEN pourrait bénéficier d'optimisation pour améliorer les performances. En s'appuyant sur l'indépendance du bus logiciel, un bus logiciel local pourrait être utilisé dès que deux services voulant communiquer s'exécutent sur le même objet communicant.

La sélection du bus logiciel est actuellement effectuée dans le programme déploiement ; elle pourrait être effectuée plus tôt, lors de la génération du canevas de programmation pour spécialiser l'intergiciel statiquement et éviter ainsi les appels de méthodes nécessaires à l'adaptation dynamique des applications sur le bus logiciel sélectionné. DIAGEN pourrait s'inspirer d'approches qui utilisent la programmation orientée aspects pour spécialiser les intergiciels [SXG+04, KG06].

Par ailleurs, le développement des services sur des systèmes embarqués nécessiterait la génération d'un canevas de programmation dans le langage C. De façon similaire, un canevas de programmation dans le langage C# simplifierait la programmation des services sur le système d'exploitation Windows Mobile et l'intégration de logiciels existants comme le serveur du

système de localisation Ubisense [SG05] et le logiciel Windows Media Center qui fournissent des interfaces de programmation en C#.

Publications

Approche DIAGEN

1. W. Jouve, N. Palix, C. Consel et P. Kadionik. « A SIP-based Programming Framework for Advanced Telephony Applications ». Dans *The 2nd Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'08)*, Heidelberg, Germany, pages 1-20, Juillet 2008. Récompensé meilleur papier étudiant.
2. W. Jouve, J. Lancia, N. Palix, C. Consel, et J. Lawall. « High-level Programming Support for Robust Pervasive Computing Applications ». Dans *Proceedings of the 6th IEEE Conference on Pervasive Computing and Communications (PerCom'08)*, Hong Kong, China, pages 252–255, Mars 2008 (Session WiP).
3. C. Consel, W. Jouve, J. Lancia, et N. Palix. « Ontology-Directed Generation of Frameworks For Pervasive Service Development ». Dans *Proceedings of the 4th IEEE Workshop on Middleware Support for Pervasive Computing (PerWare'07)*, White Plains, New York, USA,pages 501–506, Mars 2007.

Approche DIASIM

1. W. Jouve, J. Bruneau, C. Consel. « DiaSim : A Parameterized Simulator for Pervasive Computing Applications ». Dans *Proceedings of the 7th IEEE Conference on Pervasive Computing and Communications (PerCom'09)*, Galveston, Texas, Mars 2009 (Demo).

Application à l'informatique ubiquitaire

1. W. Jouve, N. Ibrahim, L. Réveillère, F. Le Mouel, et C. Consel. « Building Home Monitoring Applications : From Design to Implementation into The Amigo Middleware ». Dans *Proceedings of The Second International Conference on Pervasive Computing and Applications (ICPCA'07)*, Birmingham, UK, July 2007.
2. Y.-D. Bromberg, C. Consel, W. Jouve, S. Ben Mokhtar, N. Georgantas, V. Issarny, et P.-G. Raverdy. « Middleware for ubiquitous computing ». Dans *« ARAGO 31 : Ubiquitous Computing » Rapport de l'OFTA. ISBN 2-906028-17-7*, Mai 2007.

Mode d'interaction de type session dans les services Web

1. W. Jouve, J. Lancia, C. Consel, et C. Pu. « A Multimedia-Specific Approach to WS-Agreement ». Dans *Proceedings of The 4th IEEE European Conference on Web Services (ECOWS'06)*, Zurich, Switzerland, December 2006.

Dépôt logiciel

1. Enregistrement logiciel auprès de l'Agence de Protection des Programmes (APP) du logiciel DIAGEN version 0.2.

Bibliographie

[ACN02] J. Aldrich, C. Chambers, and D. Notkin. « ArchJava : Connecting Software Architecture to Implementation ». In *Proc. International Conference on Software Engineering*, pages 187–197. ACM Press, 2002.

[Ame87] P. America. POOL/T : A Parallel Object-Oriented Language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[ASCN03] J. Aldrich, V. Sazawal, C. Chambers, and David Notkin. « Language Support for Connector Abstractions ». In *Proceedings 17th European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[ATK92] A. L. Ananda, B. H. Tay, and E. K. Koh. « A survey of asynchronous remote procedure calls ». *SIGOPS Oper. Syst. Rev.*, 26(2) :92–109, 1992.

[BC06] P. Barron and V. Cahill. « YABS : a domain-specific language for pervasive computing based on stigmergy ». In *GPCE'06 : Proceedings of the 5th international conference on Generative programming and component engineering*, pages 285–294, Portland, OR, USA, 2006. ACM.

[BCPB04] M. Banâtre, P. Couderc, J. Pauty, and M. Becus. « Ubibus : Ubiquitous Computing to Help Blind People in Public Transport ». In *MobileHCI'04 : Proceedings of the International Conference on Mobile Human-Computer Interaction*, pages 5–12, Berlin / Heidelberg, Allemagne, 2004. LNCS.

[BCS04] E. Bruneton, T. Coupaye, and J.-B. Stefani. « The Fractal Component Model », 2004. The ObjectWeb Consortium, http ://www.objectweb.org.

[BCSS99] G. Banavar, T. Deepak Chandra, R. E. Strom, and D. C. Sturman. « A Case for Message Oriented Middleware ». In Prasad Jayanti, editor, *Distributed algorithms*, volume 1693 of *Lecture Notes in Computer Science*, pages 1–18, 1999.

[Bel02] T. Bellwood, 2002. UDDI Version 2.04 API Specification.http ://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm.

[BN84] A. D. Birrell and B. Jay Nelson. « Implementing remote procedure calls ». *ACM Trans. Comput. Syst.*, 2(1) :39–59, 1984.

[BSSW03] S. Berger, H. Schulzrinne, S. Sidiroglou, and X. Wu. « Ubiquitous computing using SIP ». In *NOSSDAV '03 : Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 82–89, New York, NY, USA, 2003. ACM.

[BZ01] N. Busi and G. Zavattaro. « Publish/Subscribe vs. Shared Dataspace Coordination Infrastructures : Is It Just a Matter of Taste ? ». In *WETICE '01 :*

Proceedings of the 10th IEEE International Workshops on Enabling Technologies, pages 328–333, Washington, DC, USA, 2001. IEEE Computer Society.

[cas] « The Castor Project ». http ://www.castor.org.

[CB03] P. Couderc and M. Banatre. « Ambient computing applications : an experience with the SPREAD approach ». *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 9–16, 2003.

[CCG+07] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. « The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario ». *Pervasive Computing and Communications, 2007. PerCom '07. Fifth Annual IEEE International Conference on*, pages 69–78, 2007.

[CCI99] R. Cerqueira, C. Cassino, and R. Ierusalimschy. « Dynamic component gluing across different componentware systems ». *Distributed Objects and Applications, 1999. Proceedings of the International Symposium on*, pages 362–371, 1999.

[cis] « Cisco Unified IP Phones 7900 Series, http ://www.cisco.com/en/US/products /hw/phones/ps379/ ».

[CPvS05] P. Dockhorn Costa, L. Ferreira Pires, and M. J. van Sinderen. « Architectural Patterns for Context-Aware Services Platforms ». In S. Kouadri Mostefaoui and Z. Maamar, editors, *Second International Workshop on Ubiquitous Computing, Miami*, pages 3–18, Portugal, 2005. INSTICC Press.

[CRS+02] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle. « Session Initiation Protocol (SIP) Extension for Instant Messaging ». RFC 3428, IETF, 2002.

[D. 99] D. Raggett, et al. « HTML 4.01 Specification ». Specification, http ://www.w3.org/TR/1999/REC-html401-19991224, W3C HTML Working Group., W3C Recommendation, 1999.

[DAS01] A. K. Dey, G. D. Abowd, and 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.

[DEM02] F. Duclos, J. Estublier, and P. Morat. « Describing and using non functional aspects in component based applications ». In *AOSD '02 : Proceedings of the 1st international conference on Aspect-oriented software development*, pages 65–75, New York, NY, USA, 2002. ACM.

[Dey01] A. K. Dey. « Understanding and Using Context. ». *Personal and Ubiquitous Computing*, 5(1) :4–7, 2001.

[Dow98] T. B. Downing. *JAVA RMI : Remote Method Invocation.* IDG Books Worldwide, Inc., 1998.

[EBF+08] A. Erbad, M. Blackstock, A. Friday, R. Lea, and J. Al-Muhtadi. « MAGIC Broker : A Middleware Toolkit for Interactive Public Displays ». In *PERCOM '08 : Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 509–514, Washington, DC, USA, 2008. IEEE Computer Society.

[ECB06] R. Etter, P.D. Costa, and T. Broens. « A Rule-Based Approach Towards Context-Aware User Notification Services ». *Pervasive Services, 2006 ACS/IEEE International Conference on*, pages 281–284, 2006.

[ecl] « Eclipse.org : open extensible IDE ». Eclipse Foundation Web site. http://www.eclipse.org/.

[EFGmK03] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A. m. Kermarrec. « The many faces of publish/subscribe ». *ACM Computing Surveys*, 35 :114–131, 2003.

[EH07a] T. Ekman and G. Hedin. « The JastAdd extensible Java compiler ». In *OOPSLA '07 : Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007.

[EH07b] T. Ekman and G. Hedin. « The JastAdd system – modular extensible compiler construction ». *Science of Computer Programming*, 69(1-3) :14–26, 2007.

[emua] « iPhone SDK, http ://developer.apple.com/iphone/program/download.html ».

[emub] « Windows Mobile 5.0 SDK for Pocket PC, Visual Studio 2005 plugin, http ://www.microsoft.com/downloads/details.aspx ?familyid=83A52AF2-F524-4EC5-9155-717CBE5D25ED&displaylang=en#Overview ».

[FB96a] N. Freed and N. Borenstein. « Multipurpose Internet Mail Extensions (MIME) Part One : Format of Internet Message Bodies ». RFC 2045, Internet Engineering Task Force, 1996. http ://www.ietf.org/rfc/rfc2045.txt.

[FB96b] N. Freed and N. Borenstein. « Multipurpose Internet Mail Extensions (MIME) Part Two : Media Types ». RFC 2046, Internet Engineering Task Force, 1996. http ://www.ietf.org/rfc/rfc2046.txt.

[FSM08] K. Frank, V. Suraci, and J. Mitic. « Personalizable Service Discovery in Pervasive Systems ». *Networking and Services, 2008. ICNS 2008. Fourth International Conference on*, pages 182–187, 2008.

[GDL+04] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. « System support for pervasive applications ». *ACM Trans. Comput. Syst.*, 22(4) :421–486, 2004.

[Gel85] D. Gelernter. « Generative Communication in Linda ». In *ACM Transactions on Programming Languages and Systems*, volume 7(1), pages 80–112, 1985.

[GSSS02] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste. « Project Aura : Toward Distraction-Free Pervasive Computing ». *IEEE PERVASIVE COMPUTING*, 1(2) :22–31, 2002.

[HcFWJ05] J. Huang, W. c. Feng, J. Walpole, and W. Jouve. « An experimental analysis of DCT-based approaches for fine-grain multi-resolution video ». In *MMCN '05 : Multimedia Computing and Networking*, San Jose, CA, USA, 2005. SPIE/ACM.

[Hen06] M. Henning. « The Rise and Fall of CORBA ». 4(5) :28–34, 2006.

[HJ98] M. Handley and V. Jacobson. « SDP : Session Description Protocol ». RFC 2327, IETF, 1998.

[HM03] G. Hedin and E. Magnusson. « JastAdd : an aspect-oriented compiler construction system ». *Science of Computer Programming*, 47(1) :37–58, 2003.

[HMD] Cervantes H., Désertot M., and Donsez D.. « FROGi : Fractal over OSGi ». http://www-adele.imag.fr/frogi/.

[Hua03] R. Huang. « Negotiation Modeling and E-Shopping Agents ». In *ICCIMA '03 : Proceedings of the 5th International Conference on Computational Intelligence and Multimedia Applications*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

[IBMa] IBM. « Business Process Execution Language ». http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[IBMb] IBM and Stop & Shop in the USA. « *Stop & Shop grocery drives sales and boosts customer loyalty with IBM Personal Shopping Assistant* ».

[IBM06] IBM and CHU in Nice, http ://www.ibm.com/news/fr/fr/2006/03/cp1851.html. « *New Generation Hospital project* », 2006.

[IG08] R. E. Frechette III and R. Gilchrist. « Towards Zero Energy, A Case Study : Pearl River Tower, Guangzhou, China ». In *CTBUH : Proceedings of the Council on Tall Buildings and Urban Habitat's 8th World Congress*, pages 7–16, 2008.

[Inf81] Information Sciences Institute, University of Southern California. « Transmission Control Protocol ». RFC 793, 1981.

[jaia] « JAIN-SIP, JAVA API for SIP Signaling, https ://jain-sip.dev.java.net ».

[jaib] « JAIN SLEE ». http ://java.sun.com/products/jain.

[jav] « Apache Web Services - Axis ». http ://ws.apache.org/axis/java.

[Jav03] Java Community Process. « *SIP Servlet API* », 2003. http ://-jcp.org/en/jsr/detail ?id=116.

[jax] « JAX-RPC Reference Implementation ». https ://jax-rpc.dev.java.net.

[JMS] « Sun Microsystems. Java Message Service (JMS) ».

[JND] « Java Naming and Directory Interface (JNDI) ». http ://-java.sun.com/products/jndi.

[JPCK08] W. Jouve, N. Palix, C. Consel, and P. Kadionik. « A SIP-based Programming Framework for Advanced Telephony Applications ». In *Proceedings of The 2nd LNCS Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'08)*, pages 1–20, Heidelberg, Germany, 2008. Best Student Paper Award.

[JPHL07] J. Jung, J. Park, S.-K. Han, and K. Lee. « An ECA-based framework for decentralized coordination of ubiquitous web services ». *Inf. Softw. Technol.*, 49(11-12) :1141–1161, 2007.

[JPS07] G. Jung, C. Pu, and G. Swint. « Mulini : an automated staging framework for QoS of distributed multi-tier applications ». In *WRASQ '07 : Proceedings of the 2007 workshop on Automating service quality*, pages 10–15, New York, NY, USA, 2007. ACM.

[KA04] G. Klyne and D. Atkins. « Common Presence and Instant Messaging (CPIM) : Message Format ». RFC 3862, IETF, 2004.

[KBR05] N. Kavantzas, D. Burdett, and G. Ritzinger. « Web Services Choreography Description Language Version 1.0 ». Working draft, W3C, http ://www.w3.org/TR/ws-cdl-10/, 2005.

[KG06] D. Kaul and A. Gokhale. « Middleware specialization using aspect oriented programming ». In *ACM-SE 44 : Proceedings of the 44th annual Southeast regional conference*, pages 319–324, New York, NY, USA, 2006. ACM.

[KGD00] S. Khurana, P. Gurung, and A. Dutta. « Device Message Protocol (DMP) : An XML based format for Wide Area Communication with Networked Appliances ». Internet draft, IETF, 2000.

[KK04] M. Keidl and A. Kemper. « Towards context-aware adaptable web services ». In *WWW Alt. '04 : Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 55–65, New York, NY, USA, 2004. ACM.

[KkK06] Y. Kim and E. k. Kim. « Autonomic Service Reconfiguration in a Ubiquitous Computing Environment ». In M. Guo, L. Tianruo Yang, B. Di Martino, H. P. Zima, J. Dongarra, and F. Tang, editors, *ISPA*, volume 4330 of *Lecture Notes in Computer Science*, pages 584–593. Springer, 2006.

[kso] « kSOAP 2, http ://ksoap2.sourceforge.net ».

[lin] « The linphone project, http ://www.linphone.org/ ».

[Mic88] Sun Microsystems. « RPC : Remote Procedure Call Protocol specification ». RFC 1050 (Historic), 1988. Obsoleted by RFC 1057.

[MN06] M. Martin and P. Nurmi. « A Generic Large Scale Simulator for Ubiquitous Computing ». In *Third Annual International Conference on Mobile and Ubiquitous Systems : Networking & Services, 2006 (MobiQuitous 2006)*, San Jose, California, USA, 2006. IEEE Computer Society.

[MPCL08] J. Mercadal, N. Palix, C. Consel, and J. Lawall. « Pantaxou : a Domain-Specific Language for Developing Safe Coordination Services ». In *Proceedings of the Seventh International Conference on Generative Programming and Component Engineering (GPCE)*, pages 149–160, Nashville, Tennessee, USA, 2008. ACM.

[mus] « Apache Muse ». http ://ws.apache.org/muse.

[NGGH07] A. Nadalin, M. Goodner, M. Gudgin, and Granqvist H.. « Web Services SecurityPolicy Version 1.2 ». Working draft, Oasis, http ://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html, 2007.

[Nie04] A. Niemi. « Session Initiation Protocol (SIP) Extension for Event State Publication ». RFC 3903, IETF, 2004.

[OAS] OASIS. « Web Services Security : SOAP Message Security 1.1 ».

[OHE97] R. Orfali, D. Harkey, and J. Edwards. *Instant CORBA*. John Wiley and Sons, Inc., 1997.

[ope] « OpenSER - the Open Source SIP Server, http ://www.openser.org ».

[osg] « OSGI, "Open Service Gateway Initiative Homepage", http ://www.osgi.org. ».

[osi] « The GNU oSIP library ». http ://www.gnu.org/software/osip.

[Pal08] N. Palix. « *Langages dédiés au développement de services de communications* ». PhD thesis, Université de Bordeaux I - LaBRI / INRIA Bordeaux - Sud-Ouest, 2008.

[Pic03] W. Picard. « NeSSy : Enabling Mass E-Negotiations of Complex Contracts ». In *DEXA '03 : Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, pages 829–833, Washington, DC, USA, 2003. IEEE Computer Society.

[Pos01] J. Postel. « Simple Mail Transfer Protocol ». IETF RFC 2821 (also STD0010), 2001. http://www.rfc-editor.org/rfc/rfc2821.txt.

[PRL07] D. Panda, R. Rahman, and D. Lane. *EJB 3 In Action*. Manning, 2007.

[PSS99] A. J. Preetham, Peter Shirley, and Brian Smits. « A practical analytic model for daylight ». In *SIGGRAPH '99 : Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 91–100, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[RCAM+05] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. Dennis Mickunas. « Olympus : A High-Level Programming Model for Pervasive Computing Environments ». In *PERCOM '05 : Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 7–16, Washington, DC, USA, 2005. IEEE Computer Society.

[RHC+02] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. « A Middleware Infrastructure for Active Spaces ». *IEEE Pervasive Computing*, 1(4) :74–83, 2002.

[Roa02] A. B. Roach. « Session Initiation Protocol (SIP)-Specific Event Notification ». RFC 3265, IETF, 2002.

[Ros02] Rosenberg, J. et al.. « SIP : Session Initiation Protocol ». RFC 3261, IETF, 2002.

[RS02] J. Rosenberg and H. Schulzrinne. « An Offer/Answer Model with the Session Description Protocol (SDP) ». RFC 3264, IETF, 2002.

[SG05] P. Steggles and S. Gschwind. « The Ubisense Smart Space Platform ». In *Adjunct Proceedings of the Third International Conference on Pervasive Computing*, 2005.

[Sie00] J. Siegel. *CORBA 3 : Fundamentals and Programming*. New York : John Wiley & Sons, 2000.

[sip] « Mobotix SIP Cameras, http ://www.abptech.com/products/Mobotix/ ».

[Sun00] Sun Microsystems. « Jini Techonology Core Platform Specification ». Java specifications, Sun Microsystems, 2000.

[SXG+04] V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron. « Middleware Specialization for Memory-Constrained Networked Embedded Systems ». In *RTAS '04 : Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 306–313, Washington, DC, USA, 2004. IEEE Computer Society.

[Szy02] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[ubi] « Hwaseong-Dongtan, U-City, http ://www.udongtan.or.kr/english/cyber /cyb_01_1.aspx ».

[udd] « IBM : UDDI4J Project ». http ://oss.software.ibm.com/developerworks/ projects/uddi4j.

[Vog03] W. Vogels. « Web Services are not Distributed Objects ». 7(6) :59–66, 2003.

[W3C] W3C. « Web Services Activity ». http ://www.w3.org/2002/ws/.

[W3C96] W3C. « Extensible Markup Language (XML) », 1996.

[Wei91] M. Weiser. « The Computer for the 21st Century. ». *Scientific American*, 265(3) :66–75, 1991.

[wir] « Wireshark : A Network Protocol Analyzer ». http ://www.wireshark.org/.

[WJD03] E. Wohlstadter, S. Jackson, and P. Devanbu. « DADO : enhancing middleware to support crosscutting features in distributed, heterogeneous systems ». In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, pages 174–186, Washington, DC, USA, 2003. IEEE Computer Society.

[ws-a] « OASIS Web Services Notification ». http ://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=wsn.

[ws-b] « Web Services Eventing ». http ://www.w3.org/Submission/WS-Eventing.

[ZMN05] F. Zhu, M.W. Mutka, and L.M. Ni. « Service discovery in pervasive computing environments ». *Pervasive Computing, IEEE*, 4(4) :81–90, 2005.

Troisième partie

Annexes

Annexe A

Exemple de spécifications DIASPEC

```
1  package  fr.inria.diagen.sample.applicationReunion;
2  import  fr.inria.diagen.sample.applicationReunion.spec.datatype.*;
3  import fr.inria.diagen.diaspecTypes.Service;
4
5  icommand LireAnnuaire {
6     Groupe LireMembres(String groupe);
7  }
8
9  icommand Notifier {
10    void notifier(Reunion reunion);
11 }
12
13 icommand LireLocalisation {
14    String LireLocalisation(String utilisateur);
15 }
16
17 component Service {}
18
19 component Agenda extends Service {
20    provides event Reunion to GestionnaireReunion;
21 }
22
23 component Annuaire extends Service {
24    provides command LireAnnuaire to GestionnaireReunion;
25 }
26
27 component Notification(String proprietaire) extends Service {
28    provides command Notifier to GestionnaireReunion;
29 }
30
31 component SystemeLocalisation extends Service {
32    provides command LireLocalisation to GestionnaireReunion;
33 }
34
35 component GestionnaireReunion extends Service {
36    requires command LireAnnuaire from Annuaire;
37    requires command Notifier from Notification;
38    requires command LireLocalisation from SystemeLocalisation;
39    requires event Reunion from Agenda;
40 }
```

FIG. A.1: Spécification DIASPEC de l'application de gestion de réunions

```
1  package  fr.inria.diagen.sample.applicationRappelIntelligent;
2  import  fr.inria.diagen.sample.applicationRappelIntelligent.spec.datatype.*;
3  import fr.inria.diagen.diaspecTypes.Service;
4
5  icommand GererAppelsManques {
6          void ajouterAppelManque(Utilisateur appelant, Utilisateur appele);
7          void supprimerAppelManque(int appelManqueID);
8  }
9
10 icommand AfficherMessage {
11         void afficher(String message);
12 }
13 icommand LireStatut {
14         Statut lireStatut();
15 }
16 icommand AfficherAppelsManques {
17         void afficher(AppelManque appelManque);
18 }
19 icommand ActiverDesactiver {
20         void Activer();
21         void Desactiver();
22 }
23 component Service(Utilisateur proprietaire) {
24         provides command LireStatut to GestionnaireServices;
25 }
26 component EntiteMaterielle(Localisation localisation, Autonomie autonomie)
27   extends Service {}
28 component CapteurLocalisation extends EntiteMaterielle {
29    provides event Localisation to SystemeLocalisation;
30 }
31 component Telephone(CapacitesAudio capacitesAudio) extends EntiteMaterielle {
32    provides session Audio to GestionnaireAppelsManques;
33    requires session Audio from TelephoneVirtuel;
34 }
35 component Ecran extends EntiteMaterielle {
36    provides command AfficherAppelsManques to GestionnaireAppelsManques;
37 }
38 component Gestionnaire(Priorite priorite) extends Service {
39    provides command ActiverDesactiver to GestionnaireServices;
40 }
41 component GestionnaireServices extends Gestionnaire {
42    requires command ActiverDesactiver from Gestionnaire;
43    requires command LireStatut from Service;
44 }
45 component TelephoneVirtuel extends Gestionnaire {
46    requires event PresenceBureau from SystemeLocalisation;
47    requires command GererAppelsManques from GestionnaireAppelsManques;
48    provides session Audio to Telephone;
49 }
50 component SystemeLocalisation(Localisation localisation) extends Gestionnaire {
51    provides event PresenceBureau to GestionnaireAppelsManques;
52    requires event Localisation from CapteurLocalisation;
53    provides event PresenceBureau to TelephoneVirtuel;
54 }
55 component GestionnaireAppelsManques extends Gestionnaire {
56    requires event PresenceBureau from SystemeLocalisation;
57    requires command AfficherAppelsManques from Ecran;
58    provides command GererAppelsManques to TelephoneVirtuel;
59    requires session Audio from Telephone;
60 }
61 component PDA extends Ecran {
62
63 }
64 component EcranOrdinateur extends Ecran {
65    requires command GererAppelsManques from GestionnaireAppelsManques;
66 }
```

FIG. A.2: Spécification DIASPEC de l'application de rappel intelligent

```
1  package  fr.inria.diagen.sample.immotique;
2  import  fr.inria.diagen.sample.immotique.spec.datatype.*;
3  import fr.inria.diagen.diaspecTypes.Service;
4
5  icommand ActiverDesactiver {
6          void Activer();
7          void Desactiver();
8  }
9  icommand Faire {
10         void faire();
11 }
12 icommand EnvoyerNotification {
13         void envoyerNotification(String message);
14 }
15 icommand MySQL {
16    Ressource envoyerRequete(String requete);
17 }
18 icommand LireCapteur {
19    String lireTypeDeDonneeCaptees();
20 }
21 icommand LireLocalisation {
22    Localisation lireLocalisation(Utilisateur utilisateur);
23 }
24 icommand LireProfil {
25    Nationalite lireNationalite(Utilisateur utilisateur);
26    Filiere lireFiliere(Utilisateur utilisateur);
27 }
28 icommand LireAgenda {
29    EmploiDuTemps lireEmploiDuTemps(Filiere filiere);
30    Reunion lireReunion(Utilisateur utilisateur);
31    Reunion lireReunion(Filiere filiere);
32 }
33 icommand Variation {
34    void setPourcentage(Pourcentage pourcentage);
35 }
36 icommand Minuteur {
37    void setMinuteur(int secondes);
38 }
39 icommand GererActualites {
40    void ajouterActualite(Actualite actualite);
41    void supprimerActualite(int ActualiteID);
42    void changerLangage(Langage langage);
43    void changerFiliere(Filiere Filiere);
44    void afficherEmploiDuTempsDuJour(EmploiDuTemps emploiDuTemps);
45 }
46 icommand Supprimer {
47    void supprimer(String serviceURI);
48 }
49 component Service(Utilisateur proprietaire) {
50    provides command Supprimer to MonitorageInfrastructure;
51 }
52 component BaseDeDonnees extends Service {
53    provides command MySQL to MonitorageInfrastructure;
54 }
55 component Actionneur(Localisation localisation) extends Service {
56    provides command ActiverDesactiver to Gestionnaire;
57    provides command Faire to MonitorageInfrastructure;
58 }
59 component Capteur(Localisation localisation) extends Service {
60    provides command ActiverDesactiver to Gestionnaire;
61    provides command LireCapteur to MonitorageInfrastructure;
62 }
63 component Gestionnaire(Priorite priorite) extends Service {
64 }
```

FIG. A.3: Spécification DIASPEC des applications d'immotique (partie 1)

```
1  component Profil extends BaseDeDonnees {
2     provides command LireProfil to GestionnaireActualites;
3  }
4  component Agenda extends BaseDeDonnees {
5     provides command LireAgenda to GestionnaireActualites;
6     provides event EtatBatiment to Gestionnaire;
7     provides event MiseAJour to GestionnaireActualites;
8     provides event Reunion to GestionnaireReunion;
9  }
10 component Notification extends Actionneur {
11    provides command EnvoyerNotification to Gestionnaire;
12 }
13 component PorteMagnetique extends Actionneur {
14    provides command ActiverDesactiver to GestionnaireIncendie;
15 }
16 component ArroseurIncendie extends Actionneur {
17   provides command ActiverDesactiver to GestionnaireIncendie;
18 }
19 component Lumiere(Puissance puissance) extends Actionneur {
20    provides command ActiverDesactiver to GestionnaireLumieres;
21 }
22 component FermetureFenetre(Orientation orientation) extends Actionneur {}
23 component CapteurLuminosite(Orientation orientation) extends Capteur {
24    provides event Luminosite to GestionnaireLumieres;
25 }
26 component SystemeLocalisation extends Capteur {
27    provides event Localisation to Gestionnaire;
28    provides event PresenceEcran to GestionnaireActualites;
29    provides command LireLocalisation to GestionnaireReunion;
30    provides event LocalisationObjet to GestionnaireSecurite;
31 }
32 component Detecteur extends Capteur {
33    provides event Alerte to MonitorageInfrastructure;
34 }
35 component GestionnaireActualites extends Gestionnaire {
36    requires event PresenceEcran from SystemeLocalisation;
37    requires command GererActualites from NotificationActualites;
38    requires command LireProfil from Profil;
39    requires command LireAgenda from Agenda;
40    requires event MiseAJour from Agenda;
41 }
42 component GestionnaireReunion extends Gestionnaire {
43    requires event Reunion from Agenda;
44    requires command LireLocalisation from SystemeLocalisation;
45    requires command EnvoyerNotification from NotificationTextuelle;
46    requires command ActiverDesactiver from NotificationSonore;
47 }
48 component GestionnaireLumieres extends Gestionnaire {
49    requires event Luminosite from CapteurLuminosite;
50    requires command ActiverDesactiver from Lumiere;
51    requires event EtatBatiment from Agenda;
52    requires command Minuteur from LumiereMinuteur;
53    requires command Variation from LumiereVariation;
54 }
55 component GestionnaireSecurite extends Gestionnaire {
56    requires event Mouvement from DetecteurMouvement;
57    requires event LocalisationObjet from SystemeLocalisation;
58    requires command ActiverDesactiver from SireneIntrusion;
59    requires event BrisGlasse from DetecteurBrisGlasse;
60    requires event EtatBatiment from Agenda;
61 }
```

FIG. A.4: Spécification DIASPEC des applications d'immotique (partie 2)

```
1  component GestionnaireIncendie extends Gestionnaire {
2    requires command ActiverDesactiver from SireneIncendie;
3    requires command EnvoyerNotification from NotificationTextuelle;
4    requires event Chaleur from DetecteurChaleur;
5    requires event Fumee from DetecteurFumee;
6    requires event AlerteIncendie from AlerteManuelIncendie;
7    requires command ActiverDesactiver from ArroseurIncendie;
8    requires command ActiverDesactiver from PorteMagnetique;
9  }
10 component MonitorageInfrastructure extends Gestionnaire {
11    requires command Supprimer from Service;
12    requires command MySQL from BaseDeDonnees;
13    requires command Faire from Actionneur;
14    requires command ActiverDesactiver from Capteur;
15    requires command LireCapteur from Capteur;
16    requires event Alerte from MonitorageInfrastructure;
17 }
18 component SonnetteAlarme extends Notification {}
19 component NotificationTextuelle(Resolution resolution) extends Notification {
20    provides command EnvoyerNotification to GestionnaireIncendie;
21    provides command EnvoyerNotification to GestionnaireReunion;
22 }
23 component NotificationSonore extends Notification {
24    provides command ActiverDesactiver to GestionnaireReunion;
25 }
26 component LumiereVariation extends Lumiere {
27    provides command Variation to GestionnaireLumieres;
28 }
29 component LumiereMinuteur extends Lumiere {
30    provides command Minuteur to GestionnaireLumieres;
31 }
32 component DetecteurChaleur extends Detecteur {
33    provides event Chaleur to GestionnaireIncendie;
34 }
35 component DetecteurFumee extends Detecteur {
36    provides event Fumee to GestionnaireIncendie;
37 }
38 component DetecteurBrisGlasse extends Detecteur {
39    provides event BrisGlasse to GestionnaireSecurite;
40 }
41 component DetecteurMouvement extends Detecteur {
42    provides event Mouvement to GestionnaireSecurite;
43 }
44 component AlerteManuelleIncendie extends Detecteur {
45    provides event AlerteIncendie to GestionnaireIncendie;
46 }
47 component SireneIntrusion extends SonnetteAlarme {
48    provides command ActiverDesactiver to GestionnaireSecurite;
49 }
50 component SireneIncendie extends SonnetteAlarme {
51    provides command ActiverDesactiver to GestionnaireIncendie;
52 }
53 component NotificationActualites extends NotificationTextuelle {
54    provides command GererActualites to GestionnaireActualites;
55 }
```

FIG. A.5: Spécification DIASPEC des applications d'immotique (partie 3)