AxSeL : un intergiciel pour le déploiement contextuel et autonome de services dans les environnements pervasifs

Amira Ben Hamida

N° d'ordre : 2010-ISAL-004 Année 2010

Université Manouba
Ecole Nationale des Sciences informatiques

Université de Lyon
Institut National des Sciences Appliquées

Thèse en cotutelle
Présentée pour l'obtention du Diplôme de Doctorat en Informatique
de l'INSA de Lyon et de l'ENSI (Université de La Manouba)

par

Amira Ben HAMIDA

intitulée

AxSeL : un intergiciel pour le déploiement contextuel et autonome de services dans les environnements pervasifs

Préparée au sein des laboratoires RIADI et CITI

Présentée devant

L'Institut National des Sciences Appliquées de Lyon
École Doctorale Informatique et Information pour la Société

Soutenue le 1er février 2010

Devant le jury composé de

Rapporteurs :	Yolande Berbers,	Professeur,	KULeuven, Belgique
	Isabelle Demeure,	Professeur,	ParisTech, France
	Riadh Robbana,	Professeur,	EPT de Tunis, Tunisie
Directeurs :	Mohamed Ben Ahmed,	Professeur émérite,	ENSI de Manouba, Tunisie
	Frédéric Le Mouël,	Maître de conférences,	INSA de Lyon, France
	Stéphane Ubéda,	Professeur,	INSA de Lyon, France
Examinateur :	Philippe Roose,	Maître de conférences,	UPPA, France

Thèse effectuée au sein du Centre d'Innovation en Télécommunications et Intégration de Services (CITI) de l'INSA de Lyon, équipe *AMbient Architectures: Services Oriented, Networked, Efficient and Secure* (AMAZONES) de l'INRIA Rhône-Alpes et le laboratoir RIADI-GDL de l'ENSI Université de La Manouba

À mes parents Moncef et Faouzia,
À la mémoire de mon grand père Belgacem,

Remerciements

Je remercie Yolande Berbers, Professeur à Katholieke Universiteit Leuven de Belgique de m'avoir fait l'honneur d'évaluer mon travail et de présider le jury de ma soutenance de thèse.

Je remercie Isabelle Demeure, Professeur à l'école nationale des sciences de télécommunications (ENST) de Paris, et Riadh Robbana, Professeur à l'école polytechnique (EPT) de Tunis, d'avoir accepté d'évaluer mon manuscrit, leurs remarques pertinentes, synonymes de l'intérêt porté à mon travail, m'ont permis de valoriser mes contributions.

Docteur Philippe Roose a également accepté de juger ce travail, je le remercie pour sa présence dans mon jury et pour ses questions intéressantes.

Cette thèse s'est déroulée dans le cadre d'une cotutelle entre l'institut national des sciences appliquées de Lyon (INSA) et l'école nationale des sciences informatiques de Manouba (ENSI), sous la direction respective du Professeur Stéphane Ubéda et du Professeur émérite Mohamed Ben Ahmed, je leur suis sincèrement reconnaissante de m'avoir fourni les moyens qui m'ont permis de mener à bien ce projet.

Je tiens également à citer l'équipe de recherche dans laquelle j'ai travaillé. D'abord, mon encadrant Docteur Frédéric Le Mouël maître de conférences à l'INSA de Lyon. Il a dirigé ma thèse et a su me conseiller aussi bien pour mon travail de recherche que pour mes enseignements. Je lui suis reconnaissante pour sa confiance et son ouverture d'esprit lors de nos échanges. Ensuite, je remercie Docteur Stéphane Frénot maître de conférences à l'INSA de Lyon et chef de l'équipe intergicielle pour son esprit critique et sa disponibilité.

Je tiens à remercier mes collègues du laboratoire et désormais amis pour le soutien qu'ils ont su me donner mais également pour les moments de fous rires partagés, les grilles de mots fléchés et les soirées passées ensemble, ils ont su me donner les moments de réconforts nécessaires, je cite Karel, Virgile, Anya, Ochir, Loïc, Paul, Wassim et Cédric. Une pensée particulière à Noha Ibrahim pour son amitié indéfectible, et à Fatiha Benali pour son grand coeur.

Enfin, je dédie cette thèse à mes parents Moncef et Faouzia qui ont toujours cru en moi et m'ont soutenu par leur amour inconditionnel, à mon frère en qui à mon tour je crois beaucoup, à ma famille, à ma grand mère, à mes amis en Tunisie, et enfin à Ioan qui m'a supporté pendant les moments les plus difficiles.

Résumé

Les environnements pervasifs sont apparus avec l'essor des technologies de communication couplé avec celui des terminaux mobiles. Parallèlement à cela, une évolution logicielle s'est effectuée en passant de l'informatique systémique globale à une approche modulaire et granulaire. Déployer une application sur un dispositif mobile reste toutefois une problématique ouverte, à cause de l'inadéquation de celle-ci aux contraintes matérielles et contextuelles de tels environnements. En effet, les applications usuelles nécessitent plus de ressources que ne peuvent fournir les terminaux mobiles. Pour pallier cela, nous proposons une solution qui s'inscrit à la frontière de ces deux mondes : celui des environnements pervasifs contraints et celui des applications orientées services.

Dans cette thèse nous présentons une approche pour le déploiement contextuel et autonome d'applications orientées services dans des environnements contraints : AxSeL -A conteXtual SErvice Loader- adapte le déploiement des applications en prenant en compte le contexte des services, la mémoire d'exécution disponible sur le dispositif et les préférences utilisateur. Dans les environnements pervasifs, les services fournis sont disséminés sur les dispositifs mobiles ou des serveurs de dépôts. L'accès à ceux-ci se fait grâce à des descripteurs de services intégrant l'ensemble des dépendances d'un service et sa localisation. Pour déployer une application il est nécessaire de résoudre l'ensemble de ses dépendances en termes de services. A partir de l'agrégation des descripteurs des services AxSeL construit sa propre représentation de l'application sous la forme d'un graphe de dépendances logiques de services. Les noeuds de ce graphe représentent les services et les composants les implantant et les arcs les dépendances d'exécution et de déploiement entre ceux-ci. Le graphe fournit une représentation intuitive et flexible des dépendances et nous permet de distinguer le niveau de l'exécution (services) de celui du déploiement (composants). AxSeL opère à la suite un processus de décision du chargement prenant en compte les caractéristiques des services, celles de la plate-forme matérielle et enfin les préférences utilisateurs. La décision est prise grâce à une technique de coloration sous contraintes du graphe de dépendances des services. En cas de changement du contexte, par exemple modification du dépôt ou des caractéristiques des services, de la mémoire disponible sur la machine ou des préférences utilisateurs, AxSeL le détecte et déclenche un processus d'adaptation qui intègre les changements perçus dans le processus décisionnel du déploiement. Ainsi, AxSeL propose une représentation globale et flexible pour les applications orientées services et des heuristiques permettant leur déploiement contextuel et autonome.

Nous avons validé les concepts de notre approche à travers un prototype AxSeL4OSGi en utilisant une machine virtuelle java comme support d'exécution et OSGi comme plate-forme de développement et d'exécution de services. Les performances de notre plate-forme sont évaluées et comparés à une approche similaire à travers des tests réalisés dans des contextes de variation de la structure des graphes applicatifs, de la mémoire disponible sur la machine, des caractéristiques des services et des préférences utilisateurs.

Abstract

Pervasive environments appeared as a consequence of the development of communicating technologies along with that of mobile electronic devices. In the same time software trends encouraged the development of modular approaches instead of the classical systemic ones. Thus, applications are no more considered as big monolithic systems but as a set of dependent modules. However, deploying an application on a constraint device is still be a hard task. Applications are resource greedy and not adapted to the hardware and contextual constraints encountered in pervasive environments. We propose a solution that is borderline between the ubiquitous world and the services-oriented one. In this thesis, we present AxSeL, A conteXtual SErvice Loader, for the service-oriented application deployment in constraint environments.

In pervasive environments provided services are spread over mobile devices as well as servers. Services access is fulfilled using services descriptors where services and application dependencies and location are listed. In order to deploy a given application its whole services dependencies are resolved. AxSeL makes services descriptors aggregation and build its own application representation. Applications are represented as logical dependency graphs. Graph nodes are the application services and graph edges are the links between them. Using graphs allows us an intuitive and flexible dependencies representation. Then, AxSeL proceeds to the loading decision by considering the services features and the hardware constraints and the user services preferences. The decision process is made through a graph coloring method. In case of context changes in the services repositories or contextual data (services, devices, users) AxSeL launches an adaptation process. The adaptation process includes the captured changes in the deployment decision in order to reach adaptability to such highly variable environments. In conclusion, AxSeL proposes a global and flexible representation of service oriented applications and a set of heuristics allowing their contextual and autonomic deployment.

We implemented AxSeL using the OSGi technology and the java virtual machine as a service execution environment. The above-mentioned propositions are proved through the prototype AxSeL4OSGi. Performance tests and comparative studies have been conducted considering different use cases and contexts.

Table des matières

Table des figures

Liste des tableaux

Listings

Liste des acronymes

ADL	Architecture Description Language
API	Application Programming Interface
AOS	Architecture Orientée Service
B2B	Business to Business
B2C	Business to Customer
CBD	Component-Based Development
CBSE	Component-Based Software Engineering
CLS	Common Language Specification
CLR	Common Language Runtime
COM	Component Object Model
DLL	Dynamic Link Library
ear	Enterprise ARchive
EJB	entreprise Java Beans
GPS	Global Positioning System
HTTP	HyperText Transfer Protocol
IBM	International Business Machines
IL	Intermediate Language
jar	Java ARchive
J2EE	Java to Entreprise Edition
JVM	Java Virtual Machine
Mo	Méga Octets
MSIL	MicroSoft Intermediate Language
MOM	Message to message
OBR	Oscar Bundle Repository
OLE	Object Linking and Embedding
OMG	Object Management Group
OSGi	précédemment connue sous le nom de Open Services Gateway initiative
PC	Personal Computer
SCA	Service Component Architecture
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOM	System Object Model
Spring DM	Spring Dynamic Modules
UML	Unified Modelling Language
URL	Uniform Resource Locator
war	Web ARchive
XML	eXtended Markup Language

Tab. 1 – Liste des acronymes

Chapitre 1

Introduction

L'objectif de ce chapitre est d'exposer les principes et les défis rencontrés dans l'intelligence ambiante. Nous mettons en évidence à travers l'évolution technologique et logicielle qui s'est opérée les défis apparus tout au long de cette évolution jusqu'à arriver aux environnements ambiants. La section 1.1 esquisse les contraintes de ces environnements et donne les directions de recherche engagées en vue d'améliorer l'existant ; elle décrit également le besoin de nouveaux paradigmes logiciels répondant aux défis d'adaptation dynamique et de prise en compte du contexte. En section 1.2, nous portons notre intérêt sur les architectures orientées services qui offrent des solutions modulaires et flexibles et des pistes de réponses aux défis des environnements ambiants. En référence à cela, en section 1.3 nous posons notre problématique qui est le déploiement contextuel de services dans des environnements pervasifs. Enfin, dans les sections 1.4 et 1.5 nous listons les points clefs de notre contribution et détaillons le contenu des chapitres.

1.1 Intelligence ambiante : principes et défis

Evolution chronologique Les années 90 ont marqué le début d'une évolution informatique et technologique durable et diffuse à travers le monde. D'abord, par la réduction de la taille des équipements informatiques de telle manière que de plus en plus de particuliers en possèdent. L'ordinateur par exemple n'est plus un serveur très encombrant et très difficile à manipuler, désormais il est léger, transportable et a une interface graphique conviviale. Les équipements se sont dotés au fil du temps de capacités matérielles incessamment croissantes et se sont miniaturisés pour susciter plus d'attrait chez les utilisateurs. De plus l'avènement d'Internet

leur a donné une nouvelle vision spatio-temporelle en permettant des interactions instantanées avec des personnes géographiquement éloignées.

Les efforts investis dans l'amélioration des protocoles de communication ont mené vers une avancée révolutionnaire : la communication sans fil. Dès lors, nous assistons à une inondation des marchés par des appareils électroniques nomades tels que les téléphones mobiles, PC portables, les PDA et GPS. La diffusion fulgurante de la technologie électronique et la forte concurrence économique du marché ont abouti à la démocratisation et à la baisse des prix de ces équipements. En se basant sur ces constats, Weiser [132] suggère en 1991 sa vision de l'informatique du futur. L'ubiquité désignera désormais la nouvelle ère de l'informatique où l'environnement est naturellement doté d'équipements électroniques qui y sont tellement bien intégrés que nous ne les distinguerons plus des objets du quotidien : l'informatique enfouie (est aussi appelée ubiquitaire ou pervasive). La figure 1.1 trace la chronologie de cette évolution depuis la naissance de l'informatique dans les années soixante jusqu'à l'émergence de l'intelligence ambiante en passant par les systèmes ubiquitaires.

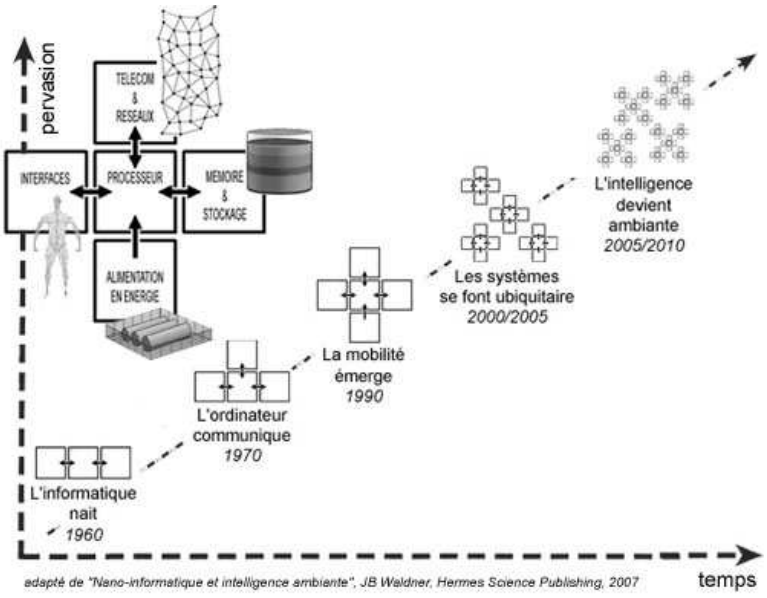

FIG. 1.1 – Chronologie de l'évolution informatique [129]

Le concept de l'informatique ubiquitaire a été le point de départ de l'intelligence ambiante [27] qui fournit une vision plus élaborée et plus tournée vers l'utilisateur. L'intelligence ambiante promet de fournir à l'utilisateur un environnement totalement personnalisé, un accès facilité à une multitude de services et de fonctionnalités, une aisance et une ergonomie de manipulation des dispositifs électroniques (mobiles, écrans, etc.) et une assistance dans la vie de tous les jours [58]. L'essence même de l'intelligence ambiante est d'améliorer la vie des personnes [26] en leur offrant une aide intuitive et transparente. Ce support omniprésent et riche ne saurait se réaliser sans la collaboration conjointe des dispositifs électroniques utilisant l'information et l'intelligence cachés dans le réseau sous-jacent.

Défis de l'informatique ambiante L'intelligence ambiante repose sur une pile de technologies superposées allant du niveau matériel au niveau logiciel couvrant des domaines tels que l'informatique ubiquitaire et embarquée, les réseaux de capteurs, les interfaces homme-machine interactives, l'intergiciel et enfin la conscience du contexte. Bien que le scénario d'un monde intelligent et réactif soit attrayant, la vérité est que l'intégration de cette panoplie de technologies avancées cache des problématiques complexes. L'intelligence ambiante hérite des problèmes des environnements mobiles et ubiquitaires [66] et [118]. Développer des systèmes pour des environnements intelligents revient à faire face à certaines de ces contraintes :

- *la limite des ressources matérielles.* En comparaison avec les dispositifs électroniques statiques, les appareils mobiles sont pauvres en ressources. L'idéal décrit par Weiser passe par l'invisibilité des machines, or la petitesse de ces éléments entraînent forcément une pénalité dans les ressources de calcul comme la rapidité de traitement, la taille mémoire, l'énergie et la capacité du disque. Bien que des avancées aient été réalisées ce sujet reste toujours d'actualité,
- *la forte mobilité des utilisateurs.* Cette caractéristique des environnements ambiants entraîne une apparition/disparition hasardeuse des utilisateurs et dispositifs mobiles de l'environnement. Outre les problèmes de sécurité inhérents cela engendre la variabilité de la connectivité qui dépendra de l'endroit où la personne se trouve. Cette contrainte est derrière la naissance du domaine de la sensibilité à la géolocalisation,
- *l'hétérogénéité matérielle et logicielle des périphériques.* Malgré l'abondance des dispositifs électroniques : téléphone, assistant personnel, ordinateur portable, GPS, ceux-ci demeurent très différents sur beaucoup de points depuis la connectivité jusqu'à la configuration matérielle et logicielle installée.

L'émergence de l'intelligence ambiante fait apparaître différentes voies de recherche. Raatikainen [112] des laboratoires Nokia en identifie certaines pour les systèmes logiciels destinés aux environnements de communication hautement dynamiques. Nous en retenons les trois suivants :

- *la reconfiguration dynamique des systèmes finaux.* Le mode de fonctionnement des dispositifs mobiles est en train de changer vers un mode plus collaboratif. En effet, les nouveaux dispositifs électroniques doivent tirer profit des ressources découvertes, créer des réseaux ad-hoc spontanément, etc. La connaissance de données sur l'environnement (dispositifs voisins, écrans, processeurs), les profils et réseaux sociaux des utilisateurs a pour but de faciliter l'auto-configuration des systèmes,
- *la reconfiguration des applications.* La mobilité des utilisateurs entraîne les disconnexions de leur accès aux services, ce qui impose de déplacer une session de service d'un périphérique à un autre pour en assurer la continuité. Une solution est de partitionner les applications en des parties coopératives et de les disséminer sur des noeuds distants. Ainsi, il est possible de redistribuer les éléments d'une application lorsqu'un changement est perçu dans l'environnement. Les approches de modularisation et d'adaptation des applications ramènent des réponses à ces objectifs,
- *la surveillance de l'environnement.* Dans le domaine de l'intelligence ambiante l'adaptation est un besoin fondamental. Elle repose sur le simple mécanisme d'action/réaction. Lorsque les circonstances changent le comportement de l'application change selon les préférences

d'un utilisateur par exemple. La surveillance de l'environnement permet l'adaptation des applications à travers la découverte des équipements, des services et des ressources matérielles disponibles.

Ainsi, pour répondre aux besoins des environnements intelligents les applications ont besoin de s'adapter aux changements contextuels. Les approches modulaires fournissent une solution pour assurer l'adaptation et l'auto-configuration des applications aux données pertinentes recueillies auprès des utilisateurs, dispositifs mobiles et ressources disponibles. Ces constats s'articulent autour de deux points majeurs : la conscience du contexte et la structure modulaire des applications.

Dans [62] le contexte est défini comme étant toute information pouvant être utilisée pour caractériser la situation d'une personne, d'un endroit ou d'un objet, celle-ci peut être pertinente lors de l'interaction entre un utilisateur et une application. Le contexte est typiquement un lieu, une identité, un état d'une personne, un groupe, des objets physiques, etc. La conscience du contexte fournit une perspective centrée autour de la personne, de ses habitudes, de ses actions et de ses mouvements afin de lui offrir des services personnalisés en accord avec ses attentes. Ceci a fait évoluer la nature de la relation qu'entreprend l'utilisateur avec les systèmes informatisés. En effet, il passe d'un mode où il ne faisait que subir l'information à un mode où il en produit constamment. Etant au coeur de l'intelligence ambiante l'utilisateur alimente constamment le système par ses données, ses mouvements et ses actions. Cette conception centrée autour de la personne fait du contexte l'une des clefs [56] de la consécration des attentes des utilisateurs et de l'obtention de gain de performances.

Enfin, la structure modulaire d'une application lui confère la flexibilité pour atteindre les capacités d'adaptation à l'environnement immédiat d'exécution et d'auto-* (configurabilité, autonomie, etc.). Les approches intergicielles modulaires [96] représentent des approches prometteuses pour les environnements ambiants [67]. Dans la section suivante, nous présentons les intergiciels orientés services et leurs apports.

1.2 Architectures orientées services pour l'intelligence ambiante

L'intergiciel (middleware) est un domaine de recherche devenu populaire dans les années 90. Il s'agit d'une classe de technologie qui aide à la gestion de la complexité et de l'hétérogénéité des systèmes distribués. Il est défini comme étant la couche intermédiaire entre le système d'exploitation et les applications. L'intergiciel est appelé aussi *plumbing* ou *the glue technology* parce qu'il connecte les applications distribuées d'une manière transparente. Il facilite la tâche de développement des applications considérablement en offrant un ensemble d'abstractions et de services dédiés. Plusieurs catégories d'intergiciels existent dont l'appel de procédure distant (RPC) [97], l'intergiciel orienté messages (MOM), l'*Object Request Broker* [89] et les architectures orientées services [81].

Nous nous intéressons à l'approche orientée service (SOA : Service-Oriented Architectures). Les SOA représentent un nouveau concept de programmation qui utilisent les services comme blocs de base pour la construction des applications distribuées d'une manière facile et peu

coûteuse. Les services sont développés indépendamment des langages de programmation et du contexte d'exécution cible. Cette approche écarte les anciennes idées où les applications logicielles étaient conçues comme étant un système monolithique statique et difficilement reconfigurable. L'intérêt porté aux SOA ne cesse d'augmenter, en effet, une étude du marché [34] effectuée par le AMR research en 2005 sur des entreprises de divers secteurs a montré l'engouement des industriels à adopter les SOA : sur 134 entreprises de divers secteurs plus de 20 % ont déjà implanté des SOA, près de 50% s'intéressent à la technologie et planifient le passage dans un futur proche (24 mois au plus) et uniquement 26 % ne considèrent pas cette option. La même étude a montré les apports observées par une centaine d'entreprises étudiées après l'adoption de SOA. Les apports décrits sont principalement la rapidité et la flexibilité lors de reconfiguration des processus métier, la réduction du coût de la technologie d'information, la sécurité et la fiabilité, la mise à jour des produits à la volée et enfin le faible couplage et l'aspect multi-plateforme. Nous présentons un histogramme issu de cette étude dans la figure 1.2.

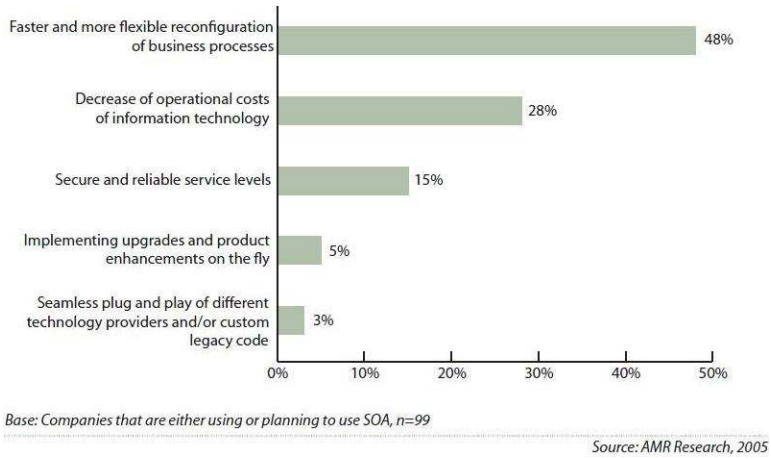

FIG. 1.2 – Les apports des SOA [34]

Les SOA sont porteurs de plusieurs paradigmes de description, de publication, de modularité et de réutilisation [102]. D'abord ils définissent une vue logique de développement des systèmes logiciels pour fournir des services aux applications ou à d'autres services distribués sur le réseau et ce à travers la découverte des interfaces publiées. Un service peut être une vue abstraite d'un programme ou d'une base de données. Ensuite grâce à la description et la publication, les détails des services sont affichés publiquement afin d'en permettre l'utilisation et la composition. A travers cet aspect les SOA permettent la visibilité et l'ouverture vers d'autres services. Pour établir la communication entre les fournisseurs et les consommateurs des services, les échanges possibles sont décrits dans la partie visible du service, masquant ainsi les détails d'implantation et réduisant le couplage entre les services.

Les SOA peuvent tirer profit des technologies web et du socle de transport sous-jacent pour permettre l'envoi des messages entre les différents services disséminés dans un réseau et ce à travers un langage indépendant de la plate-forme. Les services Web [32] par exemple utilisent le formalisme XML comme support d'interaction et les protocoles HTTP et SOAP pour la communication.

D'autre part, la granularité en terme de services rend possible l'adaptation et l'évolution des applications. En effet, l'adaptation peut soit être partielle et concerner un ou quelques services, soit totale et concerner l'ensemble de l'application. Le faible couplage entre les services permet de les administrer séparément sans que cela influence les autres services ni l'application globale. Le schéma structurel même de l'application peut être remis en cause plus aisément en lui ajoutant/retirant certains services. L'évolution est également possible grâce à des stratégies préconçues.

Les nouvelles applications ainsi conçues couplées avec des mécanismes de raisonnement contextuel peuvent disposer d'une certaine autonomie dans leur cycle de vie et anticiper les actions à entreprendre à partir de leur connaissance du contexte. Les tâches de configuration et de maintenance qui incombaient à l'utilisateur peuvent désormais être prises en charge par des mécanismes d'autonomie [66] et d'auto-reconfiguration [102], ces aspects constituent les objectifs à court terme des approches orientées services.

Malgré le manque de cadre de conception et de développement standardisé pour les services et les applications, l'idéal de Weiser d'insuffler l'intelligence dans les objets du quotidien trouve beaucoup de réponses dans l'approche orientée service. Mais tout comme Satyanarayananm [117], Weiser insiste sur le besoin d'invisibilité de l'intelligence ambiante. Or l'invisibilité est problématique car elle sous-entend la petitesse des périphériques mobiles et donc la limite de leurs ressources matérielles.

1.3 Le déploiement contextuel d'application orientée services

Le déploiement des applications logicielles sur des dispositifs contraints dans les environnements pervasifs fait face à plusieurs contraintes de l'environnement pervasif et notamment les contraintes matérielles. La limite des ressources matérielles des périphériques mobiles rend difficile l'utilisation des services à travers des dispositifs électroniques. En réalité, les performances effectives des équipements électroniques sont très limitées et dépassées par l'évolution au niveau logiciel et connectique. Pour ce qui est de la batterie, les téléphones de troisième génération proposent en théorie une panoplie de services internet, audio, vidéo, GPS et autres alors qu'en réalité ceux-ci ne disposent que de quelques heures d'autonomie à répartir entre temps de paroles et accès aux services. Quant à la mémoire vive de ce type de dispositif elle est généralement occultée (64Mo à 128Mo) par les fabricants qui, à cause de la petitesse de celle-ci, restreignent le nombre d'applications pouvant s'exécuter en même temps sur un même dispositif afin d'éviter des problèmes de mémoire limitée. D'autres part, Internet quatrième génération (Long Term Evolution) ne tardera pas à être mise en place et les dispositifs mobiles devront y faire face en intégrant le haut débit que celle-ci propose tout en tenant compte des limites matérielles (batterie, mémoire, CPU).

Dans cette thèse nous abordons la problématique du déploiement des applications orientées services sur des dispositifs contraints. Nous étudions également l'adaptation des applications aux contraintes contextuelles matérielles et provenant de l'utilisateur. Une telle approche se confronte à des problématiques telles que :

1. *L'adéquation aux contraintes contextuelles*, il s'agit d'un requis essentiel dans les environnements intelligents. Parmi ces contraintes nous relevons les limites matérielles et les préférences des utilisateurs. Grâce à des mécanismes de contextualisation il est possible de les prendre en compte lors du déploiement des applications. Cependant, contextualiser les applications à déployer doit se faire d'une manière invisible et autonome pour ne pas importuner l'usager réduisant ainsi ses interventions au moment du déploiement.
2. *La configuration à la volée des services d'une application à charger*. Dans un environnement intelligent, n'importe quel dispositif peut fournir des fonctionnalités à l'utilisateur, il est intéressant d'élargir le spectre des choix en prenant en compte l'ensemble des choix qui existent lors du déploiement d'une application. Cependant, une telle approche nous confronte à un problème NP-complet à résoudre : quels services faut-il déployer pour une application parmi la multitude offerte par l'environnement ? Faut-il changer d'avis et (re)déployer lors de changement de l'environnement ?

1.4 Une approche autonome basée sur les graphes pour le déploiement contextuel d'applications orientées services

Face aux défis énoncés dans la section précédente, nous proposons AxSeL *A conteXtual SErvice Loader*, un intergiciel pour le déploiement autonome et contextuel de services dans des environnements ambiants. AxSeL prend en charge le déploiement des applications sur le périphérique en l'adaptant aux contraintes. La prise de décision se fait sans l'intervention de l'utilisateur. Notre contribution réside en ces éléments :

1. *Un modèle d'application orientée services composants* basé sur une représentation globale et flexible sous la forme d'un graphe de dépendances. Le modèle de graphe bidimensionnel proposé est expressif et contextuel et intègre dans ses noeuds les connaissances d'exécution des services et de déploiement des composants. Il supporte également l'aspect multi-fournisseur inhérent aux approches à services par l'intégration de l'opérateur logique OU au sein des dépendances. Enfin, nous insufflons la flexibilité dans cette représentation en permettant l'ajout, le retrait et la modification des éléments du graphe. L'ensemble des données pertinentes à l'application sont absorbées dans une vue unique globale, dynamique et facilement exploitable.
2. *Un modèle de contexte dynamique* simple et basé sur la collecte à l'exécution des données contextuelles à partir du dispositif, des dépôts de services et de composants et des préférences de l'utilisateur. La notification d'AxSeL se fait grâce à un mécanisme d'écoute du contexte et de génération des actions adéquates.
3. *Des heuristiques de déploiement autonome et progressif des services et composants* qui permettent d'extraire les dépendances fonctionnelles et les caractéristiques non fonctionnelles dans une vue globale orientée graphe, et d'opérer des décisions prenant en compte plusieurs critères du contexte. Les adaptations dynamiques à l'exécution sont également

considérées. La contextualisation de l'application repose sur des mécanismes réactifs et proactifs.

4. *Un prototype AxSeL4OSGi* pour valider l'architecture AxSeL et démontrer sa faisabilité. Il est réalisé selon les spécifications OSGi et la technologie Java. Nous démontrons sa faisabilité grâce à un scénario d'illustration, évaluons ses performances et le comparons à une architecture similaire.

1.5 Plan de la thèse

Dans ce premier chapitre nous avons présenté le contexte de l'intelligence ambiante et amené les problématiques qui y sont inhérentes. Ensuite, nous avons mis en évidence les défis à relever et les contributions de notre travail. Ce manuscrit est organisé comme suit :

- **Le chapitre 2** présente un état de l'art incluant une définition du déploiement, du contexte et une spécification des points caractéristiques à considérer. Dans ce chapitre, nous étudions également les mécanismes de déploiement au sein des approches monolithiques et modulaires selon les points caractéristiques relatifs respectivement au déploiement et au contexte. A la fin de ce chapitre nous classifions les approches étudiées selon les éléments suivants : la globalité et la flexibilité de la représentation de l'application et l'autonomie du déploiement contextuel.
- **Le chapitre 3** détaille l'architecture AxSeL avec ses modèles d'applications de services et de composants et ses services coeur. Dans ce chapitre, nous modélisons le graphe de dépendances des services composants proposé. Ensuite, nous présentons notre modèle de gestion dynamique du contexte. Enfin, nous détaillons les algorithmes d'extraction du graphe de dépendances, l'heuristique de coloration du graphe et d'adaptation aux changements contextuels.
- **Le chapitre 4** détaille l'implémentation du prototype AxSeL4OSGi. Nous en présentons l'architecture générale et réalisons l'adéquation avec la technologie OSGi. Ensuite, nous illustrons le comportement de notre intergiciel à travers un cas d'utilisation pour la visualisation des documents PDF. La dernière section détaille les évaluations des performances temps et mémoire de notre prototype en comparaison avec l'architecture similaire OBR.
- **Le chapitre 5** conclue et propose une synthèse des travaux présentés. Par la suite, nous donnons les perspectives et directions de recherche possibles à ce travail.

Chapitre 2

Déploiement logiciel contextuel

Ce chapitre définit dans une première section 2.1 le déploiement logiciel et ses étapes en se basant sur plusieurs travaux de la littérature. Dans une deuxième section 2.2 nous présentons le contexte et ses principes de modélisation, de capture et de prise en compte. Ensuite, la troisième section 2.3 est consacrée aux études des approches de déploiement des applications. Nous basons cette étude sur une caractérisation des points importants, présentés en section 2.3.1. Nous considérons les applications selon leur granularité monolithique ou modulaire. Les approches monolithiques, telles que les installateurs d'applications ou les gestionnaires de packages, sont présentées dans la section 2.3.2. Les approches modulaires font l'objet de la section 2.3.3 et sont décrits selon une évolution chronologique et à travers une granularité croissante allant de l'objet vers le service en passant par le composant. Dans le cadre de notre étude, nous abordons des contributions technologiques mais aussi des contributions académiques. Enfin, nous proposons une classification des approches étudiées dans la section 2.4.

2.1 Le déploiement logiciel

Dans cette section, nous définissons le déploiement logiciel et ses différentes étapes. Pour ce, nous étudions plusieurs définitions de la littérature et proposons notre propre définition en nous basant sur chacune des contributions.

2.1.1 Définitions et principes

Pour définir le déploiement logiciel nous nous basons sur les définitions de *Object Management Group* OMG [4], de Carzaniga [51] et de Szyperski [123] et dégageons les caractéristiques de chacune d'elles selon l'acteur, l'unité, la cible et la politique de déploiement.

Définition de l'OMG Selon la spécification OMG [4] relative au déploiement et à la configuration des applications distribuées orientées composants, le processus de déploiement débute après qu'un logiciel soit développé, packagé et publié par un fournisseur de logiciels. Les étapes suivantes sont réalisées dans la globalité ou unitairement :

1. *l'installation* est le fait de prendre le *package* logiciel publié et de le ramener dans un dépôt de composants logiciels sous le contrôle du déployeur,
2. *la configuration*, lorsque le logiciel est hébergé dans un dépôt il peut être configuré pour une exécution ultérieure,
3. *la planification* décide de la manière et de l'emplacement d'exécution d'un logiciel dans un environnement cible. Les besoins de déploiement d'un logiciel et les ressources de l'environnement cible sont pris en compte pour décider de l'implantation et de l'emplacement de son exécution dans l'environnement,
4. *la préparation* est le fait d'effectuer les tâches nécessaires dans l'environnement cible pour l'exécution du logiciel. Ces tâches peuvent être le déplacement de fichiers binaires aux ordinateurs spécifiés dans l'environnement cible,
5. *le lancement* d'une application l'amène à un état d'exécution réservant toutes les ressources requises indiquées dans la métadonnée des *packages*. Les applications orientées composants sont lancées en instanciant les composants sur les machines cibles.

Acteur	La fonction de déployeur est mentionnée, il s'agit de l'acquéreur du logiciel et celui qui le déploiera sur l'environnement cible.
Unité	Un composant logiciel a des implantations de code compilé ou des *assembly* d'autres composants. Une *assembly* est un ensemble de composants, d'interconnexions, de métadonnées et de code binaire.
Cible	Le déploiement se fait à partir d'un dépôt où les *packages* sont publiés vers des machines cibles.
Politique	Un plan de déploiement résulte de l'étape de planification. L'adéquation entre les métadonnées du *package* et les ressources de la plate-forme cible est réalisée.

Tab. 2.1 – Synthèse de la définition de l'OMG

Définition de Carzaniga et al. Dans [51] les auteurs définissent le déploiement comme étant l'ensemble des étapes suivantes :

1. *la livraison* inclut toutes les opérations nécessaires pour préparer un système à l'assemblage et au transfert au site du consommateur. Les composants et la description du système sont assemblés dans un *package*. La livraison inclut aussi la publication,
2. *l'installation* couvre l'insertion initiale d'un système dans le site du consommateur, elle traite de l'assemblage de toutes les ressources nécessaires pour utiliser le système. L'installation comporte le transfert d'un produit du site producteur à celui du consommateur, et les configurations nécessaires pour son activation dans le site du consommateur,
3. *l'activation* est le lancement des composants exécutables d'un système. Il s'agit de l'établissement de quelques commandes ou des icônes d'interfaces graphiques pour l'exécution des binaires pour les systèmes simples et du lancement des serveurs ou des démons requis pour les systèmes complexes,
4. *la désactivation* considère l'activité d'arrêt des composants en exécution d'un système installé. La désactivation est souvent requise avant que d'autres activités de déploiement ne commencent,
5. *la mise à jour* est un cas spécial d'installation à travers lequel la version courante du système est mise à jour. Le système est d'abord désactivé, une nouvelle version est ensuite installée et le système est de nouveau réactivé. Dans certains cas la désactivation n'est pas nécessaire. La mise à jour inclut le transfert et la configuration des composants nécessaires,
6. *l'adaptation* inclut le fait de modifier un système logiciel déjà installé. l'adaptation est initiée par des événements locaux tels que le changement dans l'environnement cible,
7. *la désinstallation*, à un certain moment un logiciel n'est plus requis chez un site consommateur et devrait être désinstallé. La désinstallation sous entend que le système est d'abord désactivé.
8. *le retrait* est fait lorsque le système est marqué comme obsolète et que le producteur ne fournit plus son support. Tout comme lors de la désinstallation l'attention doit être portée sur les conséquences du retrait du système.

Acteur	Le producteur, le consommateur du logiciel et le site qui représente toutes les informations non fonctionnelles sur la configuration et les ressources matérielles d'une plate-forme.
Unité	L'unité de déploiement est un composant ou un *assembly* ou un système. Les composants sont les modules interdépendants constituant un système logiciel.
Cible	Le déploiement se fait d'un site producteur à un site consommateur. Un site est relatif à un ensemble d'ordinateurs et de ressources.
Politique	Aucune mention n'est faite quant aux techniques d'adéquation entre le logiciel à déployer et la plate-forme cible.

TAB. 2.2 – Synthèse de la définition de Carzaniga et al.

Définition de Szyperski Dans [123], Szyperski définit le déploiement comme un processus d'intégration invoqué par l'utilisateur. Le déploiement est le processus par lequel un composant est préparé pour l'installation dans un environnement spécifique. Ce processus doit satisfaire les paramètres du descripteur de déploiement. Le déploiement se limite à une activité ponctuelle dans le temps et dans le cycle de vie d'un logiciel, entre l'acquisition et l'installation :

1. *l'acquisition* est le processus par lequel un composant logiciel est obtenu. Un composant arrive dans une forme prête au déploiement. Celui-ci est effectué selon un descripteur de déploiement,
2. *l'installation* est le processus qui succède au déploiement et qui est souvent automatisé. L'installation rend un composant disponible dans un hôte d'un environnement particulier,
3. *le chargement* est le processus qui permet à un composant installé d'être dans un état en exécution. Si un composant définit des variables statiques ou globales alors celles-ci sont instanciées lors du chargement. L'instanciation à fine granularité selon l'approche orientée objet suit le chargement et opère sur les classes du composant chargé.

Acteur	Une mention est faite à un producteur de logiciel.
Unité	L'unité de déploiement est le composant logiciel constitué d'une collection de modules et de ressources. Tout composant est un délivrable exécutable dans un environnement d'exécution.
Cible	L'auteur mentionne le déploiement des unités du site du producteur au site cible.
Politique	Un descripteur de déploiement est censé décrire tous les paramètres à satisfaire lors du déploiement.

TAB. 2.3 – Synthèse de la définition de Szyperski

2.1.2 Synthèse

Le tableau 2.4 récapitule les définitions selon l'unité, les étapes, la cible, la politique et l'acteur du déploiement. La politique décrit les mécanismes de prise de décision. En se référant aux étapes couvertes par le processus de déploiement, la définition fournie par Carzaniga et al. [51] a plus de portée que les deux autres. Cependant, nous identifions des points intéressants dans les deux autres définitions. Chacune des définitions apporte un support intéressant au déploiement logiciel : OMG [4] introduit son activité de planification et les précisions apportées quant à l'étape d'activation. Carzaniga et al. [51] couvre un large spectre d'activités et notamment la phase d'exécution d'un logiciel. Szyperski [123] amène de la précision quant à la phase de chargement. Dans un environnement hautement instable, l'intérêt porté à la phase d'exécution d'une application permet une adaptation à la volée de celle-ci aux contraintes contextuelles. D'autre part, l'instauration d'une politique de déploiement exprimée à travers des plans de déploiement lors de la planification permet une optimisation du déploiement. Il s'agit d'une forme de contrat à accomplir entre une demande (les logiciels à déployer) et une offre (les ressources disponibles). Pour plus de complétude et pour une adéquation avec notre problématique, nous étendons la définition de Carganiza et al. avec les éléments pertinents des deux autres définitions (figure 2.1).

Définition	Unité de déploiement	Etapes	Cible	Politique	Acteur
OMG [4]	Composant Logiciel	Installation, configuration, planification, préparation, lancement	Distribuée	Adéquation aux ressources	Déployeur
Carganiza [51]	Logiciel, *assembly*	Livraison, installation, activation, désactivation, mise à jour, adaptation, désinstallation, retrait	Locale, distribuée	Aucune	Producteur et consommateur
Szyperski [123]	Composant logiciel	Préparation avant l'installation du composant	Non indiquée	Descripteur de déploiement	Producteur et utilisateur

TAB. 2.4 – Synthèse des définitions du déploiement

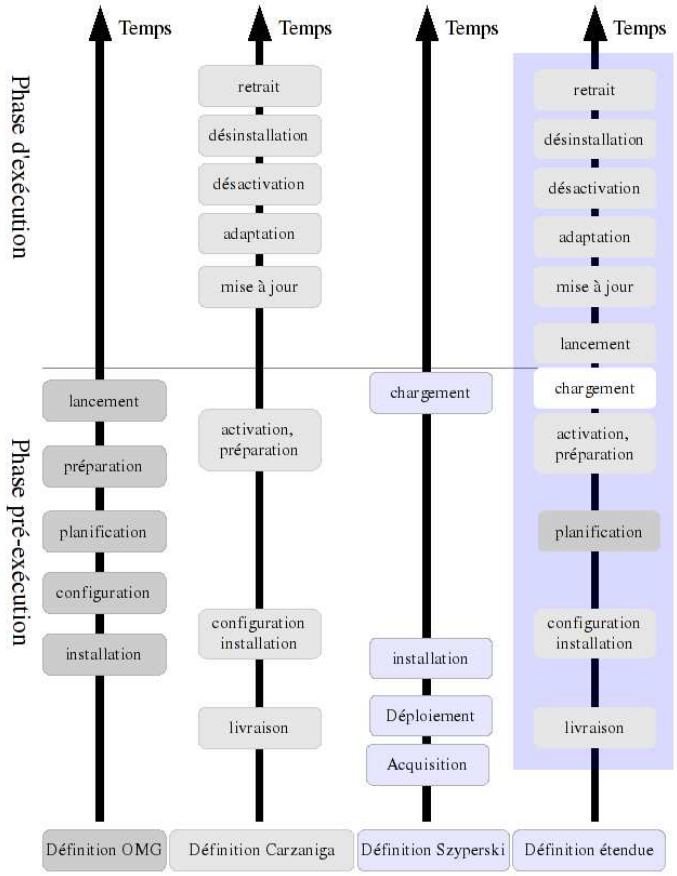

FIG. 2.1 – Définition du déploiement logiciel

2.2 La conscience du contexte

Le contexte revêt une grande importance dans les environnements pervasifs. Dans cette section nous définissons le contexte et ses principes de capture et de représentation.

2.2.1 Définitions et principes

Définition du contexte La communauté de recherche a longtemps débattu sur la définition et les usages du terme contexte sans pour autant arriver à un consensus [65]. Certains auteurs l'ont associé à l'application logicielle, en mentionnant l'état de ce qui l'entoure [131], ou encore ses paramètres [115]. D'autres l'associent à l'utilisateur même. Il est défini dans [49] comme l'ensemble des éléments faisant partie de l'environnement utilisateur et desquels l'ordinateur est conscient. Dans [83] les auteurs parlent de situation de l'utilisateur en général. Schilit et al. [120] définissent le contexte comme étant la localisation, l'identité des personnes et des objets avoisinants ainsi que les changements concernant ces derniers. Brown et al. [49] donnent une définition assez proche de la précédente en incluant l'endroit où se trouve l'utilisateur, la saison de l'année, l'heure courante, la température, etc.

Comme l'intelligence ambiante est centrée sur les utilisateurs et non seulement sur les périphériques mobiles, la notion de contexte s'étend à l'utilisateur en incluant les données qui lui sont propres. Dans cette même optique Dey [63] définit le contexte comme l'état émotionnel d'un utilisateur, l'environnement, l'identité, l'endroit et l'orientation, la date et le temps, les objets et les personnes dans l'environnement utilisateur. [119] quant à lui centre le contexte autour de trois questions essentielles : Où nous trouvons-nous ? Avec qui sommes nous ? et quelles ressources sont dans notre voisinage ? Les auteurs divisent le contexte en plusieurs sous ensembles : le contexte informatique disponible (processeurs, périphériques accessibles, capacités réseaux), le contexte utilisateur (personnes proches, situation sociale) et enfin, le contexte physique (lumières, niveaux de sonorisation).

En s'affinant, les définitions données au contexte omettent certains aspects tels que par exemple les aspects transactionnels entre les individus, les applications et les objets. Afin, d'avoir une portée plus importante les définitions considérées doivent être génériques et facilement applicables. Nous présentons ces trois définitions classées dans un ordre chronologique croissant :

1. Dey et al. [61] : *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*
2. Chen et al. [53] : *Context is the set of environmental states and settings that either determines an application's behavior or in which an application event occurs and is interesting to the user.*
3. Coutaz et al. [56] : *Context is not simply the state of a predefined environment with a fixed set of interaction resources. It's part of a process of interacting with an ever-changing environment composed of reconfigurable, migratory, distributed, and multiscale resources.*

L'ensemble de ces définitions montre la difficulté de trouver un consensus commun pour définir une "chose" qui englobe tout et n'importe quoi. Nous récapitulons dans le tableau 2.5 les apports

de chacune d'elles. Dans cette thèse, nous considérons le contexte relatif au déploiement d'une

Qui ?	(1)(2) s'accordent sur la présence d'un utilisateur auquel sont destinées les applications et qui profite du contexte.
Quoi ?	(1) Toute information caractérisant une entité (personne, endroit, objet, application), cette définition est communément admise mais omet certains éléments comme les interactions entre les applications. (2) Etats et paramètres environnementaux, événements provenant de l'environnement et de l'application.(3) Partie d'un processus d'interaction.
Où ?	(2) et (3) citent le terme environnement sans autres précisions.
Comment ?	(1) Pertinence à l'interaction entre un utilisateur et une application. (2) Intérêt pour l'utilisateur. (2) Les événements de l'environnement de l'application déterminent son comportement et ceux émanant de l'application même peuvent avoir un intérêt pour l'utilisateur. (3) Interaction, mobilité, reconfiguration et hétérogénéité des environnements pervasifs.

TAB. 2.5 – Synthèse des définitions du contexte

application comme étant l'ensemble des données non fonctionnelles pouvant être pertinentes à celui-ci. Ces données peuvent être d'ordre matériel, logiciel, provenant de l'utilisateur, ou d'autres sources elles mêmes diverses.

Types de contexte Certains auteurs ont donné une typologie des informations contextuelles. Dey [29] identifie deux types de contexte : le contexte primaire et le contexte secondaire. Le contexte primaire comporte les informations sur la localisation, l'identité, le temps et l'activité, pour caractériser la situation d'une entité particulière. Le contexte secondaire comporte par exemple l'adresse mail, le numéro de téléphone de l'utilisateur. Dey nous dessine le contexte comme une base de données dans laquelle le contexte secondaire est indexé par le contexte primaire. Il est possible de retrouver l'adresse d'une personne en cherchant dans un annuaire téléphonique son identité. Dans ce cas l'adresse de la personne est une information de second niveau qui a été retrouvée en recherchant l'identité comme un identifiant dans un espace d'informations. Les catégories de Dey se basent fortement sur la forte relation d'indexation entre la première et la deuxième catégorie.

Chen [53] fait une catégorisation hiérarchique en différenciant entre contexte de bas niveau et contexte de haut niveau : le contexte de bas niveau regroupe la localisation, le temps, l'orientation, la capacité de la bande passante, d'autres informations sur la présence, l'accélération, le niveau de lumière, etc. L'ensemble de ces informations peut être collecté à travers des capteurs ou des sondes. Le contexte de haut niveau quant à lui intègre le contexte social. Toutefois, l'acquisition de ce contexte est très complexe, par exemple il est difficile de déduire l'activité d'une personne, le recours à son agenda personnel ne garantit pas de résultat parce que l'information peut ne pas y figurer et la personne peut toujours décider à la dernière minute d'annuler son activité.

Le contexte peut aussi être classifié d'une manière non hiérarchique. Les auteurs dans [44] identifient cinq catégories de contexte : la localisation géographique, l'environnement, l'activité

de l'utilisateur, les contrats et partages entre groupes d'utilisateurs, ou les données pertinentes de fonctionnalités et de services.

Le contexte a divers aspects couvrant la localisation, le temps, l'espace, le profil utilisateur et le profil matériel. Par exemple dans [113], [28] et [130] la principale information contextuelle est la localisation, alors que dans [33] les auteurs s'intéressent au contexte du périphérique matériel : la puissance, la mémoire, le stockage, les interfaces réseaux et la batterie. Ces données sont collectées via des architectures différentes dont le choix revient au développeur de l'application.

Capture du contexte La capture des données du contexte peut se faire de plusieurs manières. Les données sont collectées à partir d'un accès direct aux capteurs tel que dans [131]. Il n'existe pas de couche intermédiaire pour le traitement des données du capteur. Les pilotes des capteurs sont incorporés dans l'application. Cette méthode impose une forte dépendance entre le contexte et l'application. Le manque de flexibilité de cette solution l'empêche d'être adaptée aux systèmes distribués.

Les approches basées sur un intergiciel s'appuient sur les méthodes d'encapsulation pour séparer la logique métier des interfaces graphiques utilisateurs. Une couche intermédiaire est ajoutée pour encapsuler les données remontées à partir de la capture. Cette technique est adoptée dans [72], elle permet l'extensibilité et la réutilisation des codes d'accès aux capteurs. La séparation entre le contexte et l'application fournit une approche intéressante car d'une part elle garantit l'extensibilité du modèle et d'autre part le contrôler des notifications des changements contextuels.

L'approche par serveur de contexte est basée sur l'architecture client/serveur, elle a été implantée dans les travaux de [80]. Elle étend celle basée sur l'intergiciel en introduisant un composant de gestion d'accès à distance. La collecte des données des capteurs est centralisée au niveau du serveur de contexte pour faciliter les accès multiples, ceci a pour avantage d'alléger la charge de traitement chez les clients.

Dans [37] les auteurs présentent d'autres approches dont celles basées sur les Widgets qui sont des composants logiciels fournissant une interface publique pour un capteur, celles basées sur les architectures orientées services, et celles basées sur le modèle orienté données tel le *Blackboard*.

Notifications des changements La notification des changements contextuels se base sur une architecture d'observateur et de fournisseur d'événements. Elle peut être mise en œuvre par les mécanismes *Push* et *Pull*. Le premier est mis en place lorsque la fréquence des changements est limitée et consiste à informer l'observateur par les changements. Il repose sur un mécanisme d'abonnement à un service de contexte et permet aux utilisateurs abonnés d'être informés des notifications qui les intéressent. Le second mécanisme *Pull* permet à un observateur de sonder à son rythme le contexte en envoyant des requêtes aux fournisseurs. Cependant, comme les sources d'informations du contexte sont différentes, la pertinence des informations ainsi que la fréquence de publication de ces événements varient également. La localisation d'une personne peut changer chaque seconde alors que celle de l'imprimante risque de ne pas bouger de l'année. La fréquence de notification des changements peut être prédéfinie par l'utilisateur de l'application pour ne pas avoir de bruit de notifications superflues. L'écoute continue des

sources contextuelles exploitent les ressources matérielles des périphériques, la planifier à des périodes précises permet de les économiser. Enfin, pour interpréter les informations recueillies il est essentiel de les modéliser selon un formalisme de représentation.

Modélisation du contexte Il est possible d'utiliser plusieurs formalismes pour représenter conceptuellement l'information contextuelle. Le développeur de l'application décide du formalisme à adopter selon ses besoins (expressivité, traitement automatique, support de raisonnement, etc.). Nous listons ci-après quelques uns de ces formalismes [44], il est possible d'en retrouver plusieurs dans une même approche :

Les premiers formalismes, sont les pairs de clé-valeur et les formalismes à balises. Pour la première catégorie la clef représente la variable de l'environnement et la valeur la donnée actuelle de cette variable. Les auteurs dans [104] se basent sur les clés-valeurs pour modéliser le contexte du système Hips/HyperAudio de guide automatique dans un musée. En pratique, cette approche est facile à mettre en place et peu coûteuse en ressources matérielles, mais elle manque de flexibilité pour modéliser des niveaux et des granularités de contextes variés.

La deuxième catégorie concerne les formalismes basés sur les balises. Dans Context-Addict [43], les auteurs façonnent les données selon le contexte qu'ils modélisent avec la structure d'un arbre. Le modèle représente la totalité de l'espace de données de tous les contextes possibles. Le premier nœud de l'arbre est l'espace de données des contextes, les nœuds de premier niveau sont les dimensions à travers lesquelles il est possible de façonner les données. Un contexte donné est spécifié à travers un ensemble d'attributs décrivant chacun un contexte autonome (utilisateur, périphérique, etc.). Les formalismes RDF et XML sont couramment utilisés pour modéliser le contexte. Ils sont facilement extensibles et permettent de représenter différents niveaux de contextes.

La modélisation du contexte se confronte à plusieurs problématiques telles que la gestion de l'hétérogénéité des informations et des sources contextuelles, la mobilité des périphériques, la gestion des relations et des dépendances entre les différents types de contexte, la facilité d'utilisation des modèles et leur flexibilité, etc. Ainsi, De nouveaux formalismes plus expressifs sont apparus pour satisfaire les besoins énoncés.

Des formalismes plus expressifs, sont apparus pour pallier le manque d'expressivité des premiers. Ces modèles exploitent des mécanismes avancées de représentation et de raisonnement. Dans [78], les auteurs s'intéressent à l'aspect temporel et à la qualité de l'information contextuelle. Les contextes possibles d'une application cible sont représentés par un graphe orienté, composé d'un ensemble d'entités et d'attributs représentant les objets et leurs propriétés. Différentes associations connectent une entité à ses attributs et à d'autres entités. Grâce à ce modèle il est possible de représenter des contextes, des contraintes variés, et des modèles à contextes multiples.

L'approche orientée objet peut également être exploitée. Dans [36], les auteurs utilisent le langage UML pour modéliser les différents contextes considérés. Une distinction est faite entre un contexte pertinent pour le déploiement et un autre à travers l'usage d'opérateurs de comparaison. Les éléments du contexte et leurs propriétés sont modélisés sous la forme de classes UML. L'approche orientée objet est intéressante car elle permet de bénéficier des paradigmes

objet comme l'héritage multiple par exemple. Elle offre aussi la flexibilité pour la représentation de plusieurs contextes à la fois et la réutilisation des modèles existants.

Enfin, les ontologies amènent une vraie puissance d'expressivité et de raisonnement. Elles sont adoptées dans le support d'apprentissage U-Learn [135]. L'apprenant et le contenu d'apprentissage sont décrits par deux ontologies. Un système basé sur les règles fournit un mécanisme de correspondance entre les deux ontologies. L'intérêt d'une telle approche réside dans la possibilité d'enrichir les données contextuelles et la puissance des outils de raisonnement.

La combinaison de plusieurs approches peut annuler quelques inconvénients de certains modèles, d'où l'apparition des approches hybrides. Dans [38] les auteurs proposent un modèle hiérarchique hybride basé sur les ontologies et une représentation spatiale du contexte. Une approche similaire basée sur les ontologies et la localisation géographique est fournie dans [121]. Enfin, [30] présente un modèle hybride couplant les ontologies aux représentations à balises. Les modèles de représentation des contextes ont beaucoup gagné en expressivité, cependant, les outils d'inférence sur les ontologies ne sont pas adaptés aux plates-formes contraintes et nécessitent d'importantes ressources matérielles pour le raisonnement. Toutefois, certains travaux tels que [109] proposent une approche qui adapte les modèles à base d'ontologie aux périphériques contraints.

2.2.2 Synthèse

Dans la section précédente nous avons présenté une définition du contexte et quelques principes de capture, de modélisation, de notification des données contextuelles, ils sont récapitulés dans le tableau 2.6.

Type : • Dispositif • Utilisateur • Localisation • Activités, etc.	**Capture** : • Capteurs • Intergiciel • Serveur de contexte	**Notification** : • Push • Pull	**Modélisation** : • Clé/valeur • langages à balises • Orientée objet • Ontologies

Tab. 2.6 – Récapitulatif des concepts du contexte

Nous retenons certains points qui nous seront utiles pour notre approche. Il est intéressant de séparer entre le contexte et l'application. La séparation des préoccupations nous permet de gérer indépendamment le fonctionnement de notre approche du contexte. L'intérêt réside dans la modularité, la flexibilité et la réutilisation de cette solution.

Ensuite, la modélisation du contexte peut être réalisée de plusieurs manières. Les approches orientées ontologies sont expressives et très intéressantes mais nécessitent des ressources matérielles importantes. Néanmoins, même si elles ne sont pas aussi expressives que les précédentes, les approches orientées objets sont intéressantes pour modéliser le contexte. Elles sont faciles à implanter, réutilisables et permettent la séparation entre le contexte et la partie applicative.

2.3 Approches de déploiement contextuel

Les sections précédentes ont présenté le déploiement et le contexte sans les lier. Dans ce qui suit, nous allons présenter les approches réalisant du déploiement logiciel et identifier leur niveau de prise en compte du contexte. Afin de caractériser ces approches, nous déterminons les points caractéristiques relatifs au déploiement et à l'adaptation contextuelle.

2.3.1 Caractérisation du déploiement contextuel d'applications

Au sein des approches de déploiement contextuel le contexte est mis au profit des mécanismes de déploiement afin de fournir une approche personnalisée. Dans cette section, nous identifions les points pertinents à étudier dans celles-ci. Nous basons notre étude sur deux éléments : les caractéristiques intrinsèques au déploiement et celles intrinsèques au contexte.

Points de caractérisation relatifs au déploiement

Le déploiement a été défini précédemment comme étant un processus de mise à disposition d'un logiciel pour en permettre l'usage. Ce processus comporte un ensemble d'étapes et concerne une unité et une cible de déploiement. D'autre part, il est important de définir une représentation des dépendances entre les éléments constituant une application. Nous identifions les points de caractérisation suivants :

- *la structure* de l'application représente son aspect modulaire ou monolithique. Elle influe sur les décisions à prendre lors du déploiement,
- *l'unité* de déploiement est le point d'entrée d'une application sur une plate-forme d'exécution et la première entité gérée par le processus de déploiement,
- *la cible* du déploiement est l'ensemble des machines hôtes destinées à l'exécution des unités de déploiement. Celui-ci peut être local sur une machine unique ou distribué sur plusieurs machines. L'aspect distribué fait apparaître des points discriminants tels que la gestion des communications entre objets distants déployés sur deux machines hétérogènes par exemple.,
- *les étapes* de déploiement sont partiellement couvertes par certaines approches de déploiement qui ne se focalisent généralement que sur quelques unes en particulier. Il est important d'identifier la contribution de ces approches par rapport aux étapes considérées,
- *la représentation* des dépendances dans les approches modulaires est une notion primordiale. Déployer un composant implique une résolution de ses dépendances. Nous nous intéressons particulièrement à la représentation de ces dépendances. Certaines représentations peuvent faciliter la gestion des dépendances et fournir le support expressif nécessaire dans des environnements pervasifs.

Points de caractérisation relatifs à l'adaptation contextuelle

Le contexte met en jeu un ensemble de paramètres liés aux entités, personnes, logiciels, périphériques matériels et environnements immédiats. Afin de fournir un déploiement qui respecte les données contextuelles recueillies à partir de diverses sources, il est nécessaire de mettre

en œuvre des mécanismes de raisonnement et de prise de décision. Nous identifions les points suivants :

- *les données contextuelles* représentent les données non fonctionnelles qui seront écoutées, modélisées et prises en compte. Toutes les approches ne considèrent pas les mêmes informations. Il est important de distinguer les données contextuelles considérées,
- *le modèle du contexte* est la façon avec laquelle les auteurs d'une approche ont modélisé le contexte. Les modèles adoptés influence le choix de la technique d'adaptation contextuelle,
- *le mécanisme de prise de décision* représente les techniques de contextualisation et d'adaptation considérées, celles-ci peuvent être réalisées par inférence sur des ontologies de contextes, par des heuristiques de déploiement, etc.

Dans ce qui suit nous étudions les approches de déploiement d'applications monolithiques et modulaires. Nous abordons aussi bien les solutions académiques que les solutions technologiques.

2.3.2 Déploiement des applications monolithiques

Les applications monolithiques sont formées d'une couche unique comportant l'interface utilisateur, la logique métier, et la manipulation des données dans un seul bloc.

Outils d'installation Pour réaliser le déploiement local des logiciels il existe des solutions telles que les installateurs. Il s'agit de technologies qui rassemblent le système logiciel en une seule archive auto-installable capable d'être distribuée à travers des supports physiques ou des réseaux. La plupart des installateurs réalisent également la désinstallation en annulant les changements faits lors de l'installation. Les autres activités de déploiement ne sont pas couvertes par ces outils.

Les installateurs se basent sur un modèle produit incluant la liste des fichiers du composant à installer ainsi que la version et des informations sur la plate-forme. Ils intègrent également le modèle du site du consommateur qui renseigne les informations de configuration du site, l'environnement utilisateur, et les fichiers systèmes. Actuellement, les installateurs de logiciels tentent de couvrir plus d'étapes de déploiement, par exemple des adaptations ou des mises à jour via Internet.

Nous citons parmi ces outils Actual Installer [9], SmInstall [1], Advanced Installer [10], IzPack [25] ou Install Shield [128]. Ce dernier est un produit Windows servant à l'installation d'applications à partir d'un support physique. Il accède aux données matérielles (CPU, mémoire, etc) fournies par la plate-forme cible du consommateur et créé un script basé sur celles-ci et sur la description du logiciel à déployer fourni par le producteur. Le script s'exécute localement dans la plate-forme cible et un mécanisme interprète les actions dictées par celui-ci. Install Shiel couvre également l'étape de désinstallation des logiciels installés.

Pour des plates-formes mobiles telles que les Smart-phones ou les Pockets PC, l'installation de logiciels dépend du système d'exploitation qui est propriétaire par exemple Windows Mobile [23], Mac OS X Mobile [17] ou le système d'exploitation orienté java d'Android [11]. Des outils tels que installAPK ou APK2EXE permettent l'installation des fichiers propriétaires Androïd de Google. Il est également possible d'installer d'autres formats de fichiers tels que les fichiers Windows Cabinet [95] ou les archives JAR dans lesquelles tous les éléments à installer

ainsi les métadonnées sont compressés. Ces outils considèrent un gros grain de déploiement incluant l'ensemble de l'application dans une unique unité de déploiement. Les applications dédiées à ce type de périphériques sont conçues d'une manière minimale destinée à l'embarqué.

Pour installer un logiciel sur un téléphone mobile par exemple nous pouvons recourir à deux manières, soit à partir des appareils mêmes et ce en accédant à un dépôt de logiciels dédiés à ces systèmes d'exploitation et en installant manuellement l'application souhaitée. Soit à travers un ordinateur plus puissant en ressources matérielles qui va en un premier temps installer le fichier à format exécutable localement, ensuite dans un deuxième temps synchroniser cette installation en copiant les fichiers requis au périphérique mobile à l'emplacement désiré. Lorsque les limites des ressources mémoires sont atteintes l'utilisateur peut être amené par un message de notification à effacer certaines applications encombrantes. Cette tâche est laissée à l'entière responsabilité de ce dernier.

Les unités de déploiement sont accompagnées de métadonnées de description sur les configurations requises ou les dépendances nécessaires à satisfaire. Cette métadonnée fournit des informations sur le processeur, la version du système d'exploitation, l'espace disque, la mémoire, la vidéo, la carte son, et les périphériques de communication. A titre d'exemple, Android utilise le formalisme XML pour la métadonnée accompagnant les logiciels à déployer. Celle-ci comporte les noms des packages java de l'application ainsi que ses composants en tant qu'activités, services, les permissions à accorder à l'application pour interagir avec d'autres applications et vice versa, ainsi que ses dépendances en termes de bibliothèques.

La métadonnée offre un support pour le déploiement et son adéquation avec les plates-formes cibles. Les installateurs couvrent une partie des activités de déploiement incluant principalement l'installation et pour certains cas la mise à jour. L'adaptation est prise en charge très partiellement par certaines solutions. Aucune prise de décision n'est faite au moment du déploiement et la seule prise en compte possible du contexte est celle du matériel. Il s'agit d'une prise en compte très basique n'incluant pas les fluctuations possibles de l'environnement.

En conclusion, ces solutions sont efficaces pour une installation locale d'applications, elles traitent de l'installation entière des applications à grosse granularité sans pour autant permettre une quelconque adaptation. Enfin, le processus d'installation de ces solutions est assez rigide et ne peut être adapté. D'autant plus que les applications possèdent une structure et un comportement statique et ce malgré la nécessité d'adaptation à l'environnement pervasif.

Gestionnaires de packages Les gestionnaires de *packages* permettent l'installation, la mise à jour et la gestion de logiciels. Le terme *package* désigne une archive qui contient les fichiers constituant l'application ainsi que la métadonnée la décrivant. Les *packages* sont référencés dans un dépôt qui renseigne l'état des packages installés. Les gestionnaires de *packages* couvrent plusieurs activités de déploiement logiciel. Ils fournissant les fonctionnalités de création, d'installation de *packages*, d'adressage de requêtes aux dépôts, de vérification de l'intégrité d'un *package* installé, de sa mise à jour et enfin de sa désinstallation. Les gestionnaires de packages réalisent l'activité de packaging dans le site du producteur et les autres activités dans le site du consommateur excepté le lancement et la désactivation.

La métadonnée d'un package est fournie par le producteur du logiciel et utilisée ensuite par

le gestionnaire de packages pour créer le package du côté du producteur. Une fois le package installé l'information provenant de celui-ci est sauvegardée dans le dépôt du consommateur. Le package manager permet l'accès à ces dépôts et la mise à jour des packages. La métadonnée fournie renferme une information riche des attributs du package. Les gestionnaires de package appliquent une sorte de politique de déploiement en modifiant le comportement des procédures de déploiement. Cette politique est toutefois très primitive et consiste en une vérification d'un certain nombre d'actions attendues.

Nous trouvons des gestionnaires de *packages* sous les systèmes d'exploitation Linux *Advanced Packaging Tool* ou *Red Hat Package Manager* ainsi que sous Windows Pkgmgr [94] pour Vista par exemple. Ces outils sont très utiles pour un déploiement local dans une machine unique ou dans un réseau partageant le même système de fichiers, toutefois, ils ne permettent pas les activités d'activation et de désactivation et ne s'adonnent pas non plus à une adaptation en phase d'exécution des *packages* installés.

Gestionnaires d'applications Les gestionnaires d'applications sont dédiés à l'écoute des interruptions qui peuvent arriver au niveau du réseau et des problèmes techniques des plates-formes matérielles, à la gestion des applications basées sur le réseau. Ils traitent certaines activités du processus de déploiement excepté l'activité de livraison du logiciel qui s'effectue par le fournisseur du logiciel. Ces outils réalisent des sous-installations sur plusieurs plates-formes et la coordination de l'installation sur des systèmes distribués ainsi que l'activation, la désactivation et le contrôle des applications. Ils assurent la qualité des services offerts en contrôlant tous les composants d'une application.

Les systèmes de gestion des applications sont généralement centralisés. Un producteur centralisé est désigné comme le site central d'administration des versions approuvées des logiciels. Les activités de gestion et de déploiement logiciel sont gérées par une station centrale d'administration. Ces systèmes sont basés sur des dépôts centralisés qui sauvegardent les métadonnées du déploiement. Celles-ci contiennent des informations sur les packages du produit, les configurations des sites cibles, et les spécifications du processus de déploiement. Ces outils fournissent également des inventaires des systèmes installés en scannant le site du consommateur et permettent aussi de superviser en temps réel les performances du système. Foglight's [110] est une plate-forme de supervision basée des agents de collecte de données distribués sur les serveurs à contrôler. Les données de disponibilité et de performance technique des composants du système d'information sont collectées à intervalles réguliers, avant d'être transmises au serveur pour stockage, analyse et traitement.

Synthèse Le tableau 2.3.2 synthétise ces approches selon les points caractéristiques identifiés en section 2.3.1.

	Déploiement				Contexte		
	Unité de déploiement	Etapes de déploiement	Cible	Dépendances	Données	Modèle	Décision
Outils d'installation	Application	installation, configuration, mise à jour, adaptation, désinstallation	Locale	Gestion automatique	Ressources matérielles et logicielles	Descripteur de métadonnée	Vérification de l'adéquation aux ressources
Outils d'installation sur plates-formes contraintes	Application	Installation, mise à jour	Locale	Gestion automatique	Ressources matérielles et logicielles	Descripteur à balises	Vérification de l'adéquation aux ressources
Gestionnaire de *packages*	Application	Livraison, installation, mise à jour, désinstallation	Locale	Gestion automatique par certains outils	Ressources matérielles et logicielles	Descripteur Clé valeur	Oui forme primitive basée sur les actions
Gestionnaire d'applications	Application	Installation, activation, désactivation	Distribuée	Gestion Automatique	Ressources matérielles à l'exécution et trafic réseau	-	-

TAB. 2.7 – Synthèse des outils d'installations des applications monolithiques

2.3.3 Déploiement des applications modulaires

Dans cette section, nous étudions des approches de déploiement destinées aux applications modulaires. D'abord, nous présentons les modèles orientés objets, ensuite, les modèles orientés composants et pour finir les modèles orientés services. Nous nous tenons au plan suivant pour chaque modèle : d'abord, nous donnons un bref historique de l'approche, ensuite, nous présentons une définition basée sur celles de la littérature, quelques approches de déploiement et enfin nous synthétisons l'ensemble selon les points caractéristiques extraits dans la section 2.3.1.

2.3.3.1 Modèle orienté objet

Historique Simula I (1962-65) et Simula 67 [47] sont les deux premiers langages orientés objets. Ils ont été créés dans le centre d'informatique norvégien à Oslo par Ole-Johan et Kristen Nygaard. Simula 67 a introduit la plupart des concepts clés de la programmation orientée objet : les objets, les classes, sous-classes (héritage), etc. Le besoin de développer de tels langages est apparu dans les années 1950 vers les début de 1960 lorsqu'il a fallu avoir des outils précis pour la description et la simulation de systèmes homme-machine complexes. Il était important que ces outils puissent fournir des descriptions des systèmes pour l'homme et la machine. Ce n'est qu'en 1970 que les compilateurs Simula sont apparus pour UNIVAC, IBM, DEC, etc. Les concepts Simula ont été d'un apport important pour les types de données abstraits et pour les modèles à programmes d'exécution concurrents. En 1970, Alan Kay du groupe Xerox PARC a utilisé Simula pour le développement de son langage SmallTalk en y intégrant les interfaces graphiques utilisateurs et les exécutions interactives de programmes. En 1980, Bjarne Stroustrup a commencé le développement du langage C++ en important les concepts de Simula dans le langage C. Plus tard, dans les années 1980 est apparu le langage ADA. D'autres suivirent comme Eiffel (B. Meyer), CLOS (D. Bobrow and G. Kiczales) et SELF (D. Ungar and others). En 1990, les langages orientés objets deviennent de plus en plus répandus. Sun Microsystems présente en 1995 le langage Java qui intègre les concepts orientés objet d'héritage, de polymorphisme, d'encapsulation, etc. Java apporte également la possibilité de développer des applications Web, des interfaces graphiques et de gérer les bases de données.

Modèle conceptuel d'un objet Afin de définir le concept d'objet nous nous basons sur la définition donnée par [47] : un objet est une abstraction des données du monde réel. Il représente un concept, une idée, ou un objet réel et est caractérisé par une identité, un état et un comportement. L'identité référence d'une manière unique un objet indépendamment des autres objets. L'état d'un objet est une valeur simple (entier, littéral) ou composée (tableau, liste). Enfin, l'ensemble des opérations applicables à un objet définit son comportement. A travers les identités des objets il est possible de faire des connexions à d'autres objets.

En factorisant les caractéristiques communes (structure, comportement) des objets, il est possible de les classifier. La classe offre un mécanisme de création d'instances donnant des objets ayant ses propriétés. Une classe est définie également par des attributs et des opérations.

L'approche par objet amène des fondements essentiels au domaine informatique comme la modularité, l'extensibilité, l'héritage à travers lesquels il est possible de passer les caractéristiques du type père au type fils et le polymorphisme permettant de représenter une

méthode sous des formes différentes en la surchargeant par exemple.

Plates-formes objets et déploiement Bien que nous nous basons dans notre approche sur les architectures orientées services, nous nous intéressons aux mécanismes sous-jacents de déploiement et plus précisément de chargement au sein des plates-formes objets considérés comme les ancêtres des services. Parmi les langages orientés objets nous citons Java, C Sharp [75], vala [114], Objective C, Eiffel, Python, C++, PHP ou Smalltalk. Nous en présentons Java et C++.

Plate-forme Java Les objets java sont gérés par une machine virtuelle (JVM) où les classes sont mises à disposition. Ceci est réalisé à travers un processus de chargement, de connexion, et d'initialisation. Le chargement est le processus de localisation de la représentation binaire d'un type et son acheminement vers le chargeur de classes. La connexion est le processus qui prend le type et l'intègre dans l'état d'exécution de la JVM pour permettre son exécution. L'initialisation se fait en exécutant les constructeurs d'un type. Lorsque l'objet ne peut plus être atteint (déréférencement), il est déchargé de la mémoire par le *garbage collector*. L'unité de déploiement est un fichier de bytecode binaire java. Les dépendances entre classes ou *packages* java ne sont pas gérées automatiquement. Le déploiement de classes java se fait sans aucune prise en compte d'un quelconque contexte ni des caractéristiques non fonctionnelles des classes ou des machines. Il existe des outils comme maven [126] et des environnements de développement comme Eclipse [127] (Integrated Development Environment) qui facilitent le développement et améliorent la gestion des dépendances.

Plate-forme C++ C++ est un langage permettant la programmation sous de multiples paradigmes procéduraux, orientés objet et génériques. La compilation d'un programme en C++ consiste en un ensemble de traitements successifs sur un fichier source dont l'étape ultime est de générer un fichier exécutable par la machine [42]. Les traitements successivement effectués sur le fichier source sont les suivants, d'abord le *pre-processing* consiste à substituer toutes les macros présentes dans le code par leur valeur. Ensuite, la compilation traite le fichier résultat du *pre-processing* et produit un fichier texte contenant du code en langage d'assemblage spécifique à la machine sur laquelle la compilation est réalisée. L'assemblage prend le fichier précédent et génère du code machine. Le fichier produit est appelé fichier objet, et se reconnaît en général par son extension ".o". Enfin, l'édition de liens de dépendances prend un ensemble de fichiers objets pour produire un programme exécutable. Elle se fait d'une manière statique ou dynamique. Lorsqu'elle est statique les codes requis sont copiés des bibliothèques à l'intérieur du programme, et lorsqu'elle est dynamique le chargement des bibliothèques se fait à l'exécution. Les dépendances ne sont pas gérées automatiquement, il faut les indiquer à la compilation du programme. L'unité de déploiement est le fichier exécutable. Si la gestion de la mémoire en Java est prise en charge par le *garbage collector*, en C++ il faut indiquer la fin de l'activité d'un objet pour libérer la mémoire qu'il occupait.

Synthèse Les approches orientées objets souffrent de certaines limites qui restreignent leur utilisation. D'abord, le manque d'expressivité, un objet ne possède pas de description contextuelle renseignant son état à l'exécution, ni les ressources qu'il requiert, ni les dépendances qu'il a avec d'autres objets. Cette limite empêche la visibilité des objets sur des réseaux plus répandus et leur réutilisation par des tiers. Les objets ne pouvant pas être publiés dans des dépôts consultables, ils sont uniquement destinés à une utilisation restreinte par des experts.

Ensuite, les dépendances entre objets permettent d'optimiser le temps de développement et de réutiliser des fonctionnalités déjà implémentées, cependant, ces dépendances de déploiement sont très fortes et ne permettent pas la flexibilité du modèle. Il est très difficile d'opérer des choix quant aux objets à déployer et d'adapter le déploiement selon les contraintes de la plate-forme ou du contexte.

Mises à part leurs limites les approches objets sont porteuses de paradigmes à travers lesquels il est possible de penser des mécanismes de prise en compte du contexte et d'amélioration du modèle existant. Des langages objets tels que java, Small Talk ou C Sharp fournissent des mécanismes de réflexivité à travers lesquels un programme peut s'introspecter pour évaluer son propre état (introspection) et par la suite le modifier à l'exécution (intercession). Ces mécanismes sont intéressants dans la mesure où ils permettent à un programme d'inspecter ses performances et de modifier son code en fonction des besoins de l'environnement.

2.3.3.2 Modèle orienté composant

Les besoins actuels en expressivité, modularité et flexibilité ont amené à l'apparition des approches orientées composants. Celles-ci sont venues comme une amélioration des modèles orientées objet. Par rapport aux objets, les composants fournissent un grain de manipulation plus gros que l'objet qui est fin (fichier, classe). Cette approche à plus grand niveau de granularité amène la possibilité de description, de publication et de réutilisation des composants, mais rend également possible l'ajout d'un niveau d'abstraction applicatif.

Historique A l'occasion de l'organisation en 1968 d'une conférence d'ingénierie informatique pour trouver une solution à ce qui était à l'époque appelé "la crise logicielle", Douglas McIlroy publie pour la première fois un article [88] sur l'utilisation de blocs logiciels préfabriqués pour construire des applications. Pour valider son concept, McIlroy avait à l'époque implanté une infrastructure basée sur l'introduction de pipelines et de filtres dans le système d'exploitation Unix. En 1986, Brad Cox publie [57] un résumé des concepts modernes de l'approche orientée composants. Il invente le langage de programmation Objective-C afin de créer une infrastructure et un marché pour ses composants. Au début des années 1990, IBM a mis en place le *System Object Model* (SOM) et Microsoft a proposé les technologies OLE et COM.

Modèle conceptuel d'un composant L'approche de programmation orientée composant est aussi appelée *component-based development* (CBD), *Component-based Software Engineering* (CBSE) ou *component-based Engineering* (CBE). Pour définir un composant logiciel il n'existe pas de consensus commun et universel. Dans la littérature, plusieurs définitions [86] ont été données, celles-ci convergent sur des points et divergent sur d'autres. Nous nous basons sur

trois définitions communément adoptées afin de dégager les propriétés communes aux modèles de composants.

1. Szyperski [122] : *a component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties,*
2. Heineman and Councill [77] : *a component is a software element that conforms to a component model and be independently deployed and composed without modification according to a composition standard.*
3. Meyer [92] : *a component is a software element (modular element), satisfying the following conditions : (i) It can be used by other software elements, its "clients". (ii) It possesses an official usage description, which is sufficient for a client author to use it. (iii) It is not tied to any fixed set of clients,*

Nous dégageons les principales propriétés d'un composant à travers les questions suivantes :

Quoi ?	Un composant est défini comme une unité logicielle de composition (1), (2) et (3). Cependant, la définition (2) inclut une forte contrainte de conception en imposant un modèle de composant.
Qui le fournit ?	Aucune des définitions ne le mentionne.
Où le trouver ?	La description du mécanisme de composition est introduite dans la définition (3). Cette nouveauté vient sans doute du besoin d'étendre l'usage des composants à divers utilisateurs. La description ramène une possibilité de publication et de diffusion des composants pour des usages plus larges.
Comment y accèder ?	(1) et (2) mentionnent un déploiement indépendant et une composition à travers des interfaces spécifiques. (2) évoque le besoin d'un standard de composition. Les contraintes imposées donnent un cadre de conception et de développement servant à homogénéiser les différents composants. (3) inclut dans la description d'un service son mode d'utilisation par les clients. Les trois définitions s'accordent sur la possibilité d'utilisation d'un composant par n'importe quel client. La composition est censée se passer sans une quelconque modification du composant en question.

Le tableau 2.8 récapitule les apports de chaque auteur selon les critères suivants : la définition donnée au composant même, la description de celui-ci, les mécanismes de composition avec des tiers et enfin le déploiement du composant. L'approche de programmation orientée composant reprend les paradigmes fournis par l'approche objet en orientant la problématique de conception vers les fonctionnalités plutôt que vers les structures. Ceci a permis d'offrir une granularité logicielle plus importante et de préconiser un niveau applicatif absent dans les approches objet.

Définition	Composant	Description	Composition	Déploiement
Szyperski [122]	Unité logicielle	Non indiqué	Interfaces spécifiques	Indépendant
Heineman and Councill [77]	Unité logicielle selon un modèle de composant	Description de la composition	Dtandard spécifique, pas de modification du composant	Indépendant
Meyer [92]	Unité logicielle	Description d'usage officielle	Pas de modification du composant	Non indiqué

TAB. 2.8 – Synthèse des définitions de composant

Plates-formes composants et déploiement Il existe beaucoup d'approches pour le déploiement de composants telles que EJB [134], CORBA [13], Microsoft .Net [8], Java Beans [107], SOFA/DCUP [105], Cortex [52], MADME [59], Sekitei [85], SDI [124], Jadabs [70], MiDUSC [79], SAMoHA [108]. Nous en présentons respectivement les technologies EJB [134], CORBA [13], Microsoft .Net [8], et les approches académiques MiDUSC [79], CADeComp [36] et SAMoHA [108].

EJB *Entreprise Java Beans* [134] est une technologie qui repose sur une architecture serveur pour les plates-formes java. Elle permet le développement des applications distribuées java. Un composant EJB est déployé sous la forme d'une archive jar contenant toutes les classes qui composent chaque EJB (interfaces home et remote, les classes qui implantent ces interfaces et toutes les autres classes nécessaires aux EJB). Les composants EJB possèdent des métadonnées qui selon les versions sont renseignées soit dans un fichier descripteur de déploiement au format XML (version 1.0 et 2.1), soit dans le code source sous forme d'annotation java (version 3.0). Les métadonnées du composant EJB permettent de donner au conteneur les caractéristiques du *bean* telles que son type, sa persistance et le type de la transaction. Pour utiliser un composant EJB un client accède à l'annuaire dans lequel il est publié, récupère sa référence et appelle ses méthodes. Un client peut être une entité de toute forme : une application avec ou sans interface graphique, un *bean*, une servlet ou une JSP ou un autre EJB. Le déploiement des composants EJB est pris en charge par le serveur sous-jacent qui effectue la création, l'installation, l'activation, la désactivation et la destruction de ceux-ci. La mise à jour n'est pas préconisée, en cas de modification dans l'EJB il est nécessaire de le redéployer manuellement dans son intégralité pour prendre en compte les changements. Une mise à jour plus granulaire n'est pas possible. Les dépendances sont gérées automatiquement dans la dernière spécification EJB. Elles sont injectées dans le code des EJB et gérées d'une manière transparente par le serveur. Les EJB possèdent un contexte d'exécution qui les informe sur leur conteneur, le *bean* et les clients. Il est possible de contrôler le processus de déploiement à travers quelques attributs du coté serveur, les fonctions *VerifyDeployments* et *StrictVerifier* permettent de s'assurer de la validité du déploiement. Cependant, la plate-forme de déploiement EJB n'opère pas de politique spécifique de déploiement.

CORBA *Common Object Request Broker Architecture* [13] est un intergciel orienté objet proposé par OMG. Ce bus d'objets répartis offre un support d'exécution masquant les couches techniques d'un système réparti (système d'exploitation, processeur et réseau) et prenant en charge les communications entre les composants logiciels qui forment les applications réparties hétérogènes. Le bus CORBA propose un modèle orienté objet client/serveur d'abstraction et de coopération entre les applications réparties. Chaque application peut exporter certaines de ses fonctionnalités (services) sous la forme d'objets CORBA. Les interactions entre les applications sont matérialisées par des invocations à distance des méthodes des objets : il s'agit de la partie coopération. La notion client/serveur intervient uniquement lors de l'utilisation d'un objet : l'application implantant et utilisant l'objet représentent respectivement le serveur et le client. Une application peut tout à fait être à la fois cliente et serveur. La définition du déploiement proposée par OMG dans la section 2.1.1 est dédiée au modèle CORBA. Les étapes de déploiement considérées sont l'installation, la configuration, la planification, la préparation et le lancement. Toutes les étapes ayant attrait à la désinstallation comme la désactivation par exemple ne sont pas prises en compte.

Microsoft .Net [8] est une plate-forme dédiée aux systèmes d'exploitation Windows, un composant est un assemblage de code binaire qui est soit une application exécutable, soit une bibliothèque DLL (*Dynamic Link Library*). Chaque assemblage est pris en charge par le CLR *Common Language Runtime*. Le CLR est l'environnement d'exécution de .NET qui charge, exécute et gère les types .NET. Il représente aussi le dépôt des composants .NET. Tous les langages .NET sont écrits d'abord selon une spécification commune (CLS) ensuite, compilés dans un langage intermédiaire (MSIL). Un composant .NET renferme la métadonnée et le code intermédiaire *Intermediate Language* (IL). La métadonnée renseigne la description d'un assemblage, des types et des attributs. La description de l'*assembly* comporte le nom d'identité, la version, les fichiers, les types, les ressources, les autres *assembly* duquel le premier dépend, et enfin un ensemble de permissions requises pour s'exécuter. La description des types inclut le nom, la visibilité, les interfaces implantées, et les membres des types comme les méthodes, champs, propriétés, événements, et les types imbriqués. Enfin, les attributs renseignent sur le nettoyage (*garbage collection*) de la mémoire et les attributs de sécurité. Lors de la construction d'un assemblage exécutable, le compilateur/éditeurs de liens renseigne quelques informations qui provoquent le chargement et l'initialisation du CLR lorsque l'exécutable est invoqué. Le CLR localise le point d'entrée de l'application et permet son exécution. Lorsqu'une méthode est appelée pour la première fois, le CLR examine le code IL de celle-ci pour connaître les types référencés, il charge ensuite tous les assemblages nécessaires définissant ces types. Si un assemblage est déjà présent le CLR ne le recharge pas.

MiDUSC est le nom que nous attribuons en référence à *Middleware for the deployment of ubiquitous software components* [79], pour en faciliter la représentation. Les auteurs présentent une approche intergicielle pour le déploiement de composants logiciels sur des réseaux dynamiques composés par des îlots de machines connectées. Les contraintes abordées sont celles de la volatilité des hôtes dans des environnements mobiles (localisation), et les ressources

matérielles et logicielles des machines. Les applications considérées sont constituées de composants hiérarchiques respectant le modèle Fractal [22], [41]. Un composant possède une ou plusieurs interfaces. Dans le modèle Fractal une interface est une sorte de port pour accéder au composant, elle implante un type particulier qui spécifie les opérations fournies. Un composant possède une membrane qui permet l'introspection et la reconfiguration de ses caractéristiques intrinsèques, et un contenu qui peut être composé par plusieurs autres composants. Dans MiDUSC l'application est décrite via un langage de description d'application (ADL) au format XML, celle-ci comporte les composants faisant partie de l'application ainsi que leurs interfaces et leurs dépendances. Les configurations matérielles et logicielles des plates-formes sont également décrites via une ADL. Chaque machine reçoit le descripteur de l'architecture et le descripteur de la plate-forme et effectue un traitement sur la possibilité d'accueillir un composant localement, le résultat de ce traitement est intégré dans un algorithme consensus pour aboutir ou pas à un accord commun entre tous les composants. Le processus de déploiement est propagatif, l'activation de l'application se fait d'une manière progressive au fur et à mesure du déploiement de ses composants. Lorsque les composants sont placés sur les hôtes, les informations concernant ce déploiement sont retranscrites sur le descripteur de déploiement pour affecter les vraies valeurs aux variables de ce dernier. Lorsque les ressources ne sont plus disponibles (baisse de la capacité mémoire par exemple), certains composants sont redéployés. Cela inclut l'arrêt du composant, la sauvegarde de son état et la publication d'un avis de redéploiement avec le nouvel état du composant. Les machines recevant cet avis remettent à jour leurs descripteurs de déploiement avec les nouvelles valeurs. En considérant cette procédure MiDUSC est aussi autonome puisque le déploiement continue jusqu'à la satisfaction du descripteur de l'application.

CADeComp *Context-Aware Deployment of COMPonents* [36] est une approche pour le déploiement contextuel des applications à base de composants. Les auteurs proposent un déploiement initial automatique à la volée. Le déploiement s'effectue d'abord en local et en cas d'épuisement des ressources certains composants sont exportés sur des plates-formes distantes. CADeComp considère les étapes de configuration, d'installation et d'activation de l'application. Cinq paramètres du déploiement sont identifiées : l'architecture de l'application, l'emplacement des instances de composants, le choix d'implantation, la valeur de leurs propriétés et leurs dépendances. Le modèle de données de CADeComp concernent les composants et les applications, il décrit le contexte qui agit sur le déploiement de l'application ainsi que les contrats de règles qui spécifie la variation des cinq paramètres identifiés. Le contexte considéré modélise essentiellement les caractéristiques des plates-formes locales et distantes et celles des utilisateurs. Ces informations sont collectées lors de la spécification, de la conception et du développement des composants par le producteur du composant. Un plan de déploiement final résulte du traitement des données contextuelles et de la prise de décision d'adaptation, ce plan indique l'architecture de l'application à déployer, ses instances de composants ainsi que leurs dépendances. La contextualisation du déploiement est réalisée à travers des contrats à respecter. En effet, un ensemble de règles d'adaptation au niveau composant et application sont spécifiées par le producteur du composant, les règles indiquent les variations des propriétés des composants, dépendances, et des implantations selon le contexte. Les rôles de producteur de composant

et d'assembleur d'application sont clairement identifiés, ils renseignent les métadonnées et les règles d'adaptation dans un format générique indépendant de la plate-forme. Le modèle des données est ensuite transformé en un format spécifique Corba [13]. En basant leur approche sur les architectures dirigées par les modèles, les auteurs de CADeComp réalisent une prise en compte d'un contexte préalablement prévu. Cette solution n'effectue pas d'adaptation ni de reconfiguration de l'application à l'exécution.

SAMoHA en référence à *Self-Adaptation for Mobile Handlheld Applications* nous attribuons le nom SAMoHA à l'approche proposée par [108]. Elle décrit une solution intergicielle pour le déploiement contextuel d'applications à base de composants sur des dispositifs contraints. Ces applications sont accompagnées par des descripteurs de déploiement listant les composants obligatoires et optionnels. SAMoHA est conçue en couches : la couche applicative, la couche dédiée à la gestion du contexte est séparée de la couche d'exécution du composant qui est par dessus la machine virtuelle java. L'auto-adaptation proposée est transversale à ces trois couches. Au niveau de chaque couche elle se fait à deux niveaux : le niveau comportemental et le niveau structurel. Dans la couche applicative l'adaptation comportementale ne change pas la composition d'une application mais sa façon de réagir aux changements du contexte. Elle se base sur l'ensemble des données collectées à partir de la couche contextuelle. L'auto-adaptation structurelle est réalisée grâce à la capacité de reconfiguration de la composition d'une application par l'ajout ou le remplacement d'un composant. Ce type d'adaptation est déclenché par des préconditions sur les messages reçus. La couche contextuelle est responsable de la collecte et de l'agrégation des données à partir des sources de données. Ces données sont stockées sous la forme de clés-valeurs et d'ontologies. SAMoHa est sensible à la localisation, la charge processeur, la batterie, la consommation mémoire et l'interface de connexion sans fil. Au niveau de la couche contextuelle, il est possible de faire des adaptations comportementales comme par exemple effacer des données obsolètes ou changer la fréquence d'acquisition des données, ou structurelles par l'ajout de composants appropriés à la chaîne de traitement du contexte par exemple. Enfin, l'adaptation est également réalisée au niveau de la couche d'exécution des composants. Elle est comportementale en baissant la fréquence du processeur en cas d'inaction du système par exemple, ou structurelle en désactivant les modules optionnels. La prise en compte de la phase d'exécution se fait grâce à des moniteurs de ressources qui sondent le processeur, la mémoire et la batterie. En se basant sur les informations recueillies à l'exécution par ces sondes, SAMoHA propose un algorithme pour réaliser le déploiement contextuel des composants. Cet algorithme choisit parmi les concepts contextuels d'une application celui qui satisfait les dépendances de déploiement et qui est le moins gourmand en ressources.

Synthèse Dans le tableau 2.9, nous classifions selon les critères mis en évidence dans la section 2.3.1 les approches étudiées.

	Déploiement				Contexte		
	Unité de déploiement	Etapes de déploiement	Cible	Dépendances	Données	Modèle	Décision
EJB [134]	Assemblage de composants	Installation, activation, désinstallation	Locale, Distribuée	XML ou annotation java, Résolution automatique par injection	Type du composant, persistance de la transaction, état du composant	Descripteur de déploiement	-
Corba [13]	Assemblage de composants	Installation, configuration, planification, préparation, lancement	Locale, Distribuée	Pas de représentation, Résolution manuelle	Catégorie et comportement du composant, transaction	Descripteur de déploiement	-
.Net [8]	Assemblage de composants	Chargement, initialisation, désinstallation	Locale, Distribuée	Pas de représentation, Résolution manuelle	-	Descripteur de déploiement	-
MiDSUC [79]	Composant hiérarchiques	Installation, activation et redéploiement	Distribuée	ADL	Ressources matérielles, localisation	ADL	Collaborative basée sur un algorithme consensus
CADeComp [36]	Composant hiérarchiques	Configuration, installation et activation	Locale et distribuée	Descripteur de composants, applications	Ressources matérielles, localisation, préférences utilisateurs	Orienté objet/XML	Prédéfinie basée sur des règles d'adaptation
SAMoHA [108]	Composant	Adaptation	Locale	Descripteur de déploiement	Ressources matérielles à l'exécution et localisation	Clé-Valeur et ontologie	Algorithme d'adaptation à l'exécution

TAB. 2.9 – Synthèse des approches de déploiement orientées composants

2.3.3.3 Modèle orienté service

L'étymologie du mot service nous renvoie vers la fonction de quelqu'un qui sert une cause ou qui aide une personne. Un service a plusieurs synonymes dont aide, assistance, bien, charité, secours, travail et utilité. Dans ce qui suit, nous définissons le concept de services en nous référant à plusieurs définitions de la littérature, ensuite nous étudions certaines approches de déploiement de services.

Historique L'approche à services est un concept assez récent [81]. Elle préconise l'usage du service comme bloc de construction d'une application. Elle se base sur des paradigmes de couplage, de publication, de découverte et de liaison entre fournisseur et consommateur de services. Ses principes de base reposent sur un mécanisme de médiation. Un fournisseur de services, un consommateur de services et enfin un médiateur (broker) entre les deux. les services sont publiés sur un annuaire décrivant les contrats de composition. Les architectures SOA ont été popularisées avec l'apparition de standards comme les Services Web dans l'e-commerce (commerce électronique) (B2B, inter-entreprise, ou B2C, d'entreprise à consommateur), basés sur des plates-formes comme J2EE ou .NET.

Modèle conceptuel d'un service La programmation orientée service se base sur les services comme unités de construction des applications distribuées [35]. Des travaux comme [136] font un état de l'art des différentes architectures orientées services. Cependant, si beaucoup de travaux s'accordent sur la pertinence de cette approche, les définitions données sont souvent contradictoires et très relatives aux technologies d'implantation. Il n'existe pas de consensus commun et universel pour définir un service. Nous nous basons sur trois définitions de la littérature pour dégager les propriétés principales d'un service :

1. Bieber [39] (1) *A service is a contractually defined behavior that can be implemented and provided by any component for use by any component, based solely on the contract.*
2. OASIS [87] (2) *A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. A service is provided by an entity the service provider for use by others, but the eventual consumers of the service may not be known to the service provider and may demonstrate uses of the service beyond the scope originally conceived by the provider.*
3. Papazoglou [101] (3) *Services are autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in novel ways. They perform functions that range from answering simple requests to executing sophisticated business processes requiring peer-to-peer relationships among multiple layers of service consumers and providers.*

Dans ce qui suit nous dégageons les principales propriétés d'un service. Le tableau 2.10 récapitule les apports de chaque auteur selon les critères suivants : la définition donnée au service même, la description de celui-ci, les mécanismes de composition avec des tiers et enfin le déploiement du composant.

Qui ?	(1) présente le service comme étant un comportement fourni par un composant à n'importe quel autre contractuellement. (2) définit quant à elle le service comme étant un mécanisme d'accès à des fonctionnalités. Enfin, (3) représente le service comme une entité autonome indépendante de la plate-forme.
Qui le fournit ?	La définition (1) présente un service comme étant fourni par n'importe quel composant, alors que les définitions (2) et (3) citent des fournisseurs de services.
Où le trouver ?	La définition (3) mentionne qu'il peut être publié et découvert, nous en déduisons qu'il se trouve probablement sur un annuaire ou un dépôt en ligne.
Comment y accéder ?	(1) La notion de contrat est très intéressante pour les paradigmes services car elle mentionne la relation entre deux services à un instant donné. L'accès est à travers des interfaces définies et se fait selon une politique et des contraintes (2). (3) évoque le couplage faible pour y accéder.

Définition	Service	Fournisseur	Découverte	Composition
Bieber [39]	Comportement fourni	Composant fournisseur	-	Selon un contrat
OASIS [87]	Mécanisme d'accès à des fonctionnalités	Fournisseur de services	-	Interfaces et politique et contraintes
Papazoglou [101]	Entité logicielle	Fournisseur de services	Citée	Couplage faible

TAB. 2.10 – Synthèse des définitions de service

Plates-formes services et déploiement Les architectures orientées services ont amené des paradigmes de haut niveau migrant l'attention plus vers le niveau fonctionnel des applications. Le déploiement au sein de ces architectures est pris en charge de plusieurs manières. Certaines plates-formes orientées services abordent uniquement le niveau contractuel de ceux-ci laissant de côté les dimensions de déploiement, nous en citons les Web Services [32], Jini [2], UPnP [93] et ReMMoC [73]. D'autres ont abordé les problématiques de déploiement des services tels que OSGi [100], Spring Dynamic Modules [19] et SCA [7]. Un grand nombre d'approches intergicielles se sont intéressées à la conscience du contexte tels que MobiPads [54], Poladian [106], CASM [103], CARISMA [50], SOCAM [74]. Nous présentons d'abord OSGi [100], quelques solutions autour de cette approche, Spring Dynamic Modules [19] et SCA [7] pour leur apport en matière de déploiement, ensuite, CARISMA [50] et SOCAM [74] pour leur apport en terme de déploiement contextuel.

OSGi [100] est une spécification à base de services destinée au départ aux plates-formes résidentielles. Il s'agit d'une approche non distribuée basée sur les paradigmes de services et de composants les implantant. La plate-forme OSGi fournit un environnement d'exécution basé sur la technologie java, et la possibilité de déployer et administrer les services et les *bundles*. Elle est répartie en plusieurs couches : sécurité, module, cycle de vie et service. La couche de sécurité

impose certaines contraintes de déploiement et se base sur la couche sous-jacente de sécurité java. Les couches module et cycle de vie fournissent respectivement un support de déploiement et un environnement d'exécution pour les *bundles*. Enfin, la couche service offre un mécanisme de gestion des services à l'exécution sans avoir à se préoccuper de leurs implantations. Un service OSGi est fourni par un ou plusieurs *bundles* OSGi. Le *bundle* est l'unité de déploiement de cette plate-forme. Il peut fournir zéro ou plusieurs services ou *packages*. Un service peut être implanté différemment par plusieurs *bundles*. Il s'agit d'une archive jar renfermant les classes compilées java, des ressources (librairies, fichier, etc.) et un descripteur du *bundle* (manifest). Le descripteur renseigne essentiellement les données relatives aux dépendances avec des librairies, et la classe Activator responsable du lancement et arrêt du service. Lors du développement des *bundles* il est important de spécifier les dépendances de *pakages*, celles de services ne sont pas requises. OSGi permet la gestion du cycle de vie d'un *bundle*, l'installation, l'activation, la mise à jour, l'arrêt et la désinstallation. Lors de l'installation et de l'activation d'un *bundle* sur la plate-forme OSGi, celui-ci ne démarrera que si ses dépendances sont présentes. Une fois ce dernier installé et démarré, il peut enregistrer les services qu'ils fournit dans le contexte d'exécution. Ceux-ci sont alors visibles pour d'éventuels consommateurs qui doivent les chercher dans le contexte. Les dépendances de services n'étant pas obligatoires à préciser, il n'est possible de les apercevoir réellement que lors de la phase d'exécution. Pour apercevoir un service il est obligatoire de le charger localement sur la machine même s'il n'est pas utilisé. Bien que basé sur les services, OSGi ne considère pas les échanges et les dépendances entre services, ceux-ci sont gérés par l'utilisateur du service à la phase d'exécution. Le grain de manipulation proposé est le *bundle*, OSGi ne donne pas de précisions quant à la notion d'applications orientées services ni, à la granularité de celles-ci. Le mécanisme de composition entre *packages* est supporté d'une manière déclarative à la phase de développement. OSGi conçoit l'approche à services comme composants orientées services. Le déploiement de composants au dessus d'OSGi est manuel.

Autour d'OSGi plusieurs solutions ont été proposées pour enrichir le modèle. OSGi *Bundle Repository* [3] est une proposition pour la gestion du déploiement de *bundles* et de dépendances au sein d'OSGi, elle permet l'installation et la résolution des dépendances d'un *bundle* pour permettre son chargement. OBR définit un dépôt de *bundles* pouvant être hébergé localement ou à distance. Il s'agit d'un descripteur XML listant l'ensemble des *bundles* et des ressources (*bundles* ou *packages*) requis. Il fournit un mécanisme de gestion d'une fédération de dépôt et la résolution automatique des dépendances d'un *bundle*. Pour retrouver l'ensemble des dépendances d'un *bundle* en termes de *bundles* et de *packages* OBR effectue un traitement récursif. Enfin, il n'impose pas de protocoles particuliers pour décrire les dépendances de services.

Le *Service Binder* [76] est un mécanisme facilitant le développement et le déploiement de services OSGi. Il se base sur l'extraction de la logique de gestion des dépendances entre services qui incombait au développeur des *bundles*. Les dépendances de services sont décrites dans un descripteur XML et injectées dans le code du *bundle*. iPOJO [68] [64] est une suite au *Service Binder* dans laquelle les auteurs proposent un modèle de composants services pour la construction d'applications dynamiques au dessus d'OSGi, les dépendances de services y

sont injectés dans le *bundle*. De la même manière est conçue la gestion des dépendances de services dans les services déclaratifs présentées dans la nouvelle spécification R.4.1 OSGi [6]. Cependant, ces derniers proposent un modèle de dépendances plus riche définissant une politique de liaison entre services dynamique (changer de fournisseur s'il n'est plus disponible) ou statique (instance détruite si le fournisseur disparaît). Enfin, nous citons le *Dependency Manager* [99] qui simplifie le développement de services OSGi pour permettre la construction d'applications orientées services. Il sépare le code métier du code de gestion des dépendances et utilise un mode déclaratif basé sur une API dans laquelle sont décrits les *bundles*. Des efforts ont été fournis autour de la plate-forme OSGi pour automatiser le déploiement des services, nous notons cependant, que les solutions fournies n'opèrent pas de prise en compte du contexte d'exécution, ni de politiques spécifiques de chargement, toutefois, elle offrent des outils qu'il est possible d'exploiter pour réaliser cela.

Spring Dynamic Modules [19] est issu de la réunion du patron de conception Spring et du modèle OSGi. Spring fournit un conteneur léger et un modèle de programmation non intrusif basé sur l'injection des dépendances, l'approche par aspects, et les abstractions de services. La plate-forme de services OSGi offre un environnement d'exécution et de déploiement dynamique de *bundles*. L'unité de déploiement sur cette plate-forme est le *bundle* OSGi. Des annotations supplémentaires destinées aux composants Spring et OSGi de la plate-forme accompagnent le *bundle* à déployer, celles-ci renferment respectivement les définitions des *beans* et les dépendances OSGi. Il est également possible de spécifier certains paramètres d'exécution au sein de ces métadonnées, par exemple le temps d'attente de réponse d'un service, ou le niveau de satisfaction des dépendances exigées. Spring Dynamic Modules distingue le niveau d'exécution de celui du déploiement, et conçoit un contexte d'application créé automatiquement pour chaque *bundle* à partir des métadonnées renseignées. Le contexte d'application est l'unité de modularité primaire de Spring DM. Il est responsable de l'instanciation, la configuration, l'assemblage et la décoration des objets Spring (*beans*) au sein du *bundle*. Il contient des *beans* (objets gérés par Spring), et peut être configuré hiérarchiquement de telle sorte que le contexte d'application enfant peut voir les *beans* définis par son père mais pas l'inverse. Les usines *bean factory* exportent les références des *beans* aux clients externes au contexte de l'application, et injectent les références dans les services définis à l'extérieur du contexte de l'application. Il est possible d'exporter certains *beans* comme des services OSGi, ou de les injecter d'une manière transparente avec des références à des services OSGi. Le contexte d'application est enregistré comme un service dans l'annuaire de services OSGi pour en faciliter le test, l'administration et la gestion. La possibilité d'accéder à un *bean* lors de l'exécution est intéressante mais il est déconseillé d'y accéder directement, il est préférable d'utiliser les mécanismes d'imports exports. Le déploiement au dessus de Spring DM repose sur l'environnement d'exécution OSGi et étend les mécanismes existants : lorsqu'un bundle Spring arrêté est réactivé alors un contexte d'application lui est automatiquement créé, lorsqu'il est arrêté tous les services qu'il exportait sont ôtés de l'annuaire de services, et le contexte d'application et les *beans* détruits. Par rapport à OSGi, Spring DM ramène une notion de gestion à l'exécution des services et des applications orientées services et ce à travers l'allocation d'un contexte pour l'application et une gestion du cycle

de vie des services. Spring DM gère d'une manière autonome l'intergiciel, lorsqu'un contexte d'application n'exporte pas de services, ou exportent des services qui ne sont pas couramment référencés, ceux-ci sont arrêtés selon un ordre chronologique (les plus récents sont arrêtés en premier).

SCA *Service Component Architecture* [7] est un ensemble de spécifications pour construire les applications orientées services indépendemment du langage de programmation. Le composant est l'élément basique dans un assemblage SCA. Les composants sont ensuite combinés dans des composites SCA pour fournir des fonctionnalités. Les composants sont des instances configurées d'implantations. Les composants fournissent et consomment des services. Un composant contient zéro ou plusieurs services et/ou références. Plusieurs composants peuvent utiliser une même implantation mais configurée différemment via la référence proposée. Les dépendances des composants envers d'autres composants ou services sont réalisées à travers des références. Les composants orientés services SCA sont d'abord développés ensuite assemblés. L'assemblage correspond à la correspondance entre les références proposées et les services fournis. Les implantations possèdent également des propriétés telles que des données nécessaires aux fonctionnalités fournies. Un domaine fournit une configuration à l'exécution pouvant être distribuée sur un ensemble de nœuds connectés. Il s'agit d'un ensemble de services fournissant des fonctionnalités métiers nécessaires aux composites et qui sont contrôlés par une organisation centralisée. L'unité de déploiement dans le domaine SCA est la contribution, il s'agit d'un composite packagé sous la forme d'un zip, jar, ear, war, etc. Une contribution peut être accompagnée d'un descripteur XML listant les composants exécutables, les dépendances en imports et exports de services et de composants. L'installation et l'activation d'une contribution impose la présence et l'installation des éléments desquels elle dépend (décrits dans le descripteur). La spécification SCA réalise partiellement la gestion et la résolution automatique des dépendances, cependant, il est préférable de déléguer la résolution des dépendances aux technologies adoptées. Un mécanisme d'auto-résolution est préconisé *auto-wire* qui permet de résoudre des dépendances non explicitées par le développeur. A l'exécution, l'ajout d'éléments composites et la mise à jour d'une contribution déployée sont possibles. SCA prévoit aussi la désinstallation des contributions déployées du domaine d'exécution. Cependant, aucune politique de déploiement, ni de prise en compte de données non fonctionnelles ne sont précisées. SCA a été implantée pour J2EE, C++, BPEL, et COBOL.

CARISMA *Context Aware Reflective Middleware System for Mobile Applications* [50] est un intergiciel orienté services s'adaptant par réflexivité aux changements d'un contexte d'exécution pervasif. La prise en compte du contexte se fait à la phase de déploiement et d'exécution. Les auteurs considèrent le contexte comme tout ce qui peut influencer le comportement d'une application. Cela inclut les périphériques matériels (batterie, mémoire, taille de l'écran), ou le contexte utilisateur (humeur, activité courante). Les données sont collectées directement à partir des sondes spécifiques ensuite enregistrées dans le modèle du contexte sous un format XML simple. Les auteurs s'appuient sur la notion de profil d'application représentant le comportement de l'application dans un contexte particulier. Le comportement d'une application est défini par l'association entre les services configurés par l'intergiciel, les politiques

qui peuvent être appliquées pour fournir les services, et les conditions contextuelles requises pour appliquer une politique donnée. La reconfiguration de l'application est dirigée par une politique prenant en compte les métadonnées recueillies. L'ensemble des profils de l'application sont transférés à l'intergiciel. A chaque fois qu'un service est invoqué, l'intergiciel prend en charge de consulter le profil de l'application, de se renseigner des ressources requises par l'application même, et de déterminer quelle politique appliquer au contexte courant. Les applications peuvent introspecter et modifier leur profil à travers une API réflexive. Pour ce qui est des conflits pouvant être engendrés par la réflexivité, les auteurs appliquent une méthode inspirée du monde économique : les enchères, ceux-ci se basent sur des priorités accordées aux ressources selon leur importance et pertinence dans le contexte. CARISMA est constitué d'un gestionnaire de contexte responsable de la capture des variations contextuelles, d'un ensemble de services responsables de satisfaire aux requêtes des services selon les niveaux de qualité de services prédéfinis, et d'un modèle d'application donnant un cadre standard pour la création et l'exécution d'applications au dessus du middleware. CARISMA fournit une représentation du contexte basé sur un modèle simple d'encodage du contexte en vue de préserver les ressources des périphériques. La pertinence du choix de la politique est évaluée selon des mécanismes d'enchères et de fonction d'utilité.

SOCAM *Service-oriented Context-Aware Middleware* [74] est un intergiciel qui permet de construire et de prototyper des services mobiles et conscients du contexte. Il est composé d'un fournisseur de contexte, d'un interpréteur de contexte, d'une base de données contextuelles, d'un service de localisation de services, et de services mobiles conscients du contexte. Afin de profiter des paradigmes services, SOCAM est elle même conçue en tant qu'une architecture orientée services où chaque élément représente un service indépendant pouvant être déployé sur des périphériques distants. Le fournisseur de contexte offre une abstraction qui sépare le bas niveau de capture du haut niveau de manipulation du contexte. L'interpréteur de contexte agit également comme un fournisseur de services en interprétant des contextes de bas niveau en contextes de haut niveau. Il est composé d'un raisonneur de contexte et d'une base de connaissances. Le raisonneur fournit des contextes déduits à partir des contextes directs en résolvant les conflits et en maintenant la consistance de la base de connaissance. Il est possible de gérer, modifier, ajouter, ou retirer à travers des API des connaissances de la base de contexte. Les auteurs modélisent le contexte sous la forme d'ontologies, leurs instances sont spécifiées par les utilisateurs ou acquises à partir d'autres fournisseurs de contexte. Le service de localisation de services permet aux utilisateurs et aux applications de localiser les différents fournisseur de contexte. L'adaptation et la conscience au contexte sont réalisées avec un ensemble de règles prédéfinies par les développeurs de services et déclenchant un comportement souhaité des services en question. La prise de décision du comportement à avoir est réalisée grâce à un mécanisme de raisonnement sur le modèle du contexte.

Synthèse Dans le tableau 2.11, nous classifions selon les critères mis en évidence dans la section 2.3.1 les approches étudiées.

	Déploiement				Contexte		
	Unité de déploiement	Etapes de déploiement	Cible	Dépendances	Données	Modèle	Décision
OSGi [100]	*Bundle*, *package*	Installation, activation, mise à jour, arrêt et désinstallation	Locale	Descripteur de *bundle* (clé-valeur), Automatique dans (OBR, *Service Binder*, *Dependency Manager*, *Services déclaratifs*)	Ressources logicielles (service,*bundle*, *package*)	-	-
Spring DM [19]	*Bundle* OSGi	Installation, activation, mise à jour, arrêt et désinstallation	Locale et distribuée	Descripteur XML, Partiellement gérée	Ressources logicielles (service, composite, composant)	-	Désinstallation des services non utilisés
SCA [7]	Contribution	Installation, activation, mise à jour, arrêt et désinstallation	Locale et distribuée	Descripteur XML, manifest OSGi (clé-valeur), paramètres *runtime*	Ressources logicielles (service, *bean*, *bundle*, *package*)	-	Partielle lors de la désinstallation
CARISMA [50]	Service	Adaptation	Locale et distribuée	Descripteur XML	Ressources logicielles, matérielles,	Représentation XML	Politiques d'adaptation, réflexivité
SOCAM [74]	Service	Adaptation	Distribuée	-	Ressources matérielles et préférences utilisateur	Ontologies, contexte capturé, dérivé, interprété	Inférence sur ontologie contextuelle

TAB. 2.11 – Synthèse des approches de déploiement orientées services

2.4 Classification

Dans cette section, nous classifions et discutons les approches de déploiement présentées. D'abord, nous spécifions les critères de classification, ensuite, nous abordons dans une première partie les approches technologiques et dans une seconde partie les approches académiques suivant les critères suivants :

Vue globale Le cycle de vie d'une application passe par différentes étapes relatives à son déploiement et son exécution sur une plate-forme. Les approches orientées services et composants reposent sur un modèle dans lequel les services et composants sont publiés dans des annuaires en vue de leur utilisation par des clients. Plusieurs fournisseurs peuvent offrir un même service ou composant. Par vue globale nous relevons l'importance d'une représentation pluridimensionnelle de l'application qui rassemble les aspects de déploiement (composant), d'exécution (service) et multi-fournisseurs. Chacune des applications étudiées considère à sa manière un ou plusieurs de ces aspects. Nous les classifions selon leur prise en compte totale ou partielle de ces aspects.

Flexibilité Une application orientée services ou composants est construite sur un modèle de dépendances entre ceux-ci. La rigidité de ces dépendances fige la structure de l'application. La flexibilité des applications est un besoin essentiel dans les environnements intelligents. Nous définissons la flexibilité de la structure d'une application par sa capacité d'extensibilité qui se traduit par la possibilité d'ajout et de retrait d'éléments de la structure, et sa dynamicité qui dénote du caractère changeant de ceux-ci.

Autonomie Les environnements intelligents promettent de fournir les fonctionalités d'une manière transparente et non intrusive. L'autonomie de l'intergiciel à prendre les décisions nécessaires au déploiement évite d'envahir l'utilisateur par des sollicitations de prise de décision. Les décisions de déploiement peuvent être régies par des événements déclencheurs (réactivité), ou bien par des comportements prévus (proactivité) comme par exemple l'anticipation du chargement des éléments pour de futures usages. Nous classifions les cinq approches par rapport aux critères de proactivité et de réactivité.

Approches technologiques Dans ce qui suit nous classifions et discutons les approches technologiques orientées applications monolithiques et celles orientées composants et services : EJB [134], CORBA [13], Microsoft .Net [8], OSGi [100], Spring DM [19] et SCA [7]. Le tableau 2.4 fournit une classification selon les critères de vue globale, de flexibilité et d'autonomie. Les signes + et - mentionnent respectivement si ce dernier est supporté ou non.

L'ensemble des outils présentés et destinés aux applications monolithiques comme les installateurs d'applications (A) , les gestionnaires de packages (B) et les gestionnaires des applications (C), couvre partiellement le cycle de déploiement tel que défini dans la section 2.1. Au sein de ces applications, les prises de décision réalisées sont pour la plupart basiques et considèrent une correspondance entre des ressources requises et des ressources offertes par la plate-forme. Les données contextuelles considérées sont pour la plupart statiques et concernent les ressources

Critères	Propriétés	A	B	C	EJB	CORBA	.NET	OSGi	Spring	SCA
Vue globale	Multi-fournisseur	-	-	-	-	-	-	-	-	-
	Déploiement	+	+	+	+	+	+	+	+	+
	Exécution	-	-	+	-	-	-	+	+	+
Flexibilité	Dynamicité	-	-	+	-	-	-	-	+	+
	Extensibilité	-	-	-	-	-	-	+	+	+
Autonomie	Proactivité	-	-	-	-	-	-	-	+	-
	Réactivité	-	-	+	-	-	-	-	-	-

TAB. 2.12 – Classification des approches technologiques de déploiement

matérielles uniquement, à l'exception du dernier cas de gestionnaires d'applications où les performances des applications et des services sont sondées à l'exécution. Ces systèmes ne sont pas autonomes et ne suivent pas de stratégies de déploiement spécifiques.

EJB [134] prend en compte un contexte basique renseigné d'une manière statique dans des descripteurs de composants. Les données non fonctionnelles considérées sont uniquement à destination du serveur d'exécution des composants et ne concerne pas un contexte matériel ni des préférences utilisateur. Le support de déploiement fourni par cette technologie effectue la création, l'installation, l'activation, la désactivation et la destruction des composants, mais pas la mise à jour. La mise à jour se fait en redéployant manuellement le composant en entier. La gestion des dépendances est prise en compte automatiquement dans les dernières versions. Cependant, cette technologie n'opère pas de politique spécifique de déploiement. Les composants EJB et leurs dépendances sont installés intégralement et systématiquement.

CORBA [13] considère les étapes de déploiement suivantes : l'installation, la configuration, la planification, la préparation et le lancement. Toutes les étapes ayant attrait à la désinstallation comme la désactivation par exemple ne sont pas prises en compte. Cette plate-forme ne propose pas de représentation particulière des dépendances d'un composant, ni de résolution automatique de celles-ci.

Microsoft .Net [8] supporte les étapes de chargement, d'initialisation et de désinstallation. Le contexte n'est pas géré par cette plate-forme et seules les propriétés fonctionnelles sont renseignées. Aucune prise de décision autonome n'est réalisée.

OSGi [100] ramène le paradigme service mais les dépendances d'exécution sont laissées au soin du développeur, seules les dépendances de déploiement sont renseignées "en dur" à la phase de développement. OSGi fournit un support fiable au déploiement et à l'exécution des services, néanmoins, il souffre de quelques limites telles que l'absence d'un niveau applicatif, l'absence d'un support de gestion et de contrôle des services ou des *bundles* à l'exécution et le manque d'expressivité du modèle (pas de paramètres non fonctionnels). iPOJO [68] est une solution au dessus d'OSGi qui étend le modèle de service actuel à un modèle plus expressif. Il fournit également un modèle de dépendances flexibles entre composants et services, mais pas de mécanismes automatiques de déploiement.

Spring DM [19] améliore les apports amenés par OSGi, et fournit un support d'exécution et un niveau applicatif au dessus des *bundles*. Toutefois, même s'il gère d'une manière autonome la suppression de services non utilisés, il ne fournit aucune politique pour le chargement autonome

et contextuel d'application.

SCA [7] fournit des spécifications pour construire les applications orientées services indépendamment du langage de programmation. L'installation et l'activation d'une contribution SCA nécessite la présence et l'installation des éléments desquels elle dépend (décrits dans le descripteur). La gestion et la résolution automatique des dépendances sont partiellement réalisées. Le schéma de la contribution SCA peut être étendu par l'ajout à l'exécution d'éléments composites. L'étape de mise à jour d'une contribution déployée est prise en charge. SCA prévoit aussi la désinstallation des contributions déployées du domaine d'exécution. Cependant, aucune politique de déploiement, ni de prise en compte de données non fonctionnelles ne sont précisées.

Ces plates-formes fournissent des modèles de composants et des supports de déploiement couvrant partiellement les étapes de déploiement telles que définies dans la section 2.1.1. Le déploiement fourni est systématique et impose la présence de l'intégralité des composants d'une unité de déploiement. Les composants et services proposés ne sont pas accompagnés par des propriétés non fonctionnelles concernant les plates-formes matérielles ou les préférences utilisateurs, seuls les contextes basiques sont pris en compte. D'autant plus qu'il n'y a pas de prise de décision contextuelle appropriée fournie par ces approches. D'autre part, le niveau d'expressivité des besoins non fonctionnels des composants et services au sein des approches technologiques se limite aux données basiques n'ayant pas attrait aux utilisateurs ni aux plates-formes matérielles. Le support de la flexibilité des schémas structurels des applications est très limité.

Approches académiques Dans le tableau 2.4, nous classifions les approches académiques suivantes MiDUSC [79], CADeComp [36], SAMoHA [108], CARISMA [50] et SOCAM [74] par rapport aux critères énoncés. Les signes + et - mentionnent respectivement si ce dernier est supporté ou non.

Critères	Propriétés	MiDUSC	CADeComp	SAMoHA	CARISMA	SOCAM
Vue globale	Multi-fournisseur	-	-	-	-	-
	Déploiement	+	+	+	-	-
	Exécution	-	-	+	+	+
Flexibilité	Dynamicité	+	-	+	-	-
	Extensibilité	-	-	+	-	-
Autonomie	Proactivité	-	-	+	+	-
	Réactivité	+	+	+	+	+

TAB. 2.13 – Classification des approches académiques de déploiement

MiDUSC [79] considère les étapes d'activation et de redéploiement. L'application ainsi que les configurations matérielles et logicielles sont décrites par un descripteur à travers un langage déclaratif à balises. La prise de décision est réalisée avec un algorithme consensus, elle est réactive et propagative. MiDUSC modifie les valeurs contextuelles se trouvant dans le descripteur avec de nouvelles valeurs et fournit ainsi une représentation dynamique des dépendances. Cependant, les aspects multi-fournisseur inhérents aux environnements intelligents n'est pas supporté.

CADeComp [36] traite uniquement les étapes de déploiement initial et d'activation. CADeComp part d'une vue statique décrivant les dépendances des composants et grâce à des règles d'adaptation détermine le contexte pertinent à exploiter, et génère un plan de déploiement destiné à placer les instances des composants. Cette approche repose sur le modèle MDA pour être indépendante de la technologie sous-jacente. L'adaptation est réactive et est réalisée grâce à des règles d'adaptation. Cependant, cette approche considère des descripteurs statiques d'applications et ne supporte pas de représentation bidimensionnelle et flexible intégrant plusieurs niveaux.

SAMoHA [108] s'apparente à notre approche car l'adaptation considérée est transversale à plusieurs domaines. Elle concerne les différentes couches d'exécution, de gestion du contexte et de déploiement de composants. L'adaptation effectuée est comportementale et structurelle. Cependant, cette approche ne fournit pas l'aspect multi-fournisseur nécessaire à la vue bidimensionnelle. L'adaptation est prise en charge d'une manière réactive à travers des préconditions sur les événements recueillis et ne supporte pas de proactivité. SAMoHA couvre principalement l'étape d'adaptation dans le déploiement.

CARISMA [50] prend en compte le contexte à la phase de déploiement et d'exécution de l'application. Les auteurs considèrent l'adaptation comportementale des profils des applications déployées. La structure de l'application est représentée sous la forme d'un descripteur statique et n'intègre pas l'aspect multi-fournisseur, dynamique et extensible. L'adaptation contextuelle de CARISMA est réactive et proactive. En effet, des écouteurs de contexte sont déployés, les valeurs contextuelles sondées influencent le processus de décision. D'autre part, CARISMA préconise un ensemble de comportements prévus par l'application et le transmet à l'intergiciel.

SOCAM [74] porte l'intérêt principalement aux aspects contextuels. Les auteurs n'y fournissent pas de contribution d'une quelconque représentation de la structure d'une application. La prise en compte du contexte est réalisée d'une manière réactive selon des règles d'adaptation prédéfinies. SOCAM n'a pas un comportement proactif.

Conclusion L'étude des approches académiques et technologiques de déploiement permet de mettre en relief trois limites principales : l'absence de l'aspect multi-fournisseur et la limite de la flexibilité de la vue globale ainsi que le faible support de la proactivité du déploiement. Afin d'intégrer plusieurs niveaux d'abstraction incluant l'exécution, le déploiement, l'aspect multi-fournisseur et les propriétés non fonctionnelles, il est nécessaire de fournir une vue globale, flexible et expressive pour représenter les applications. La prise de décision du déploiement est possible à travers l'ajout d'un niveau de décision basé sur les contraintes contextuelles collectées et le raisonnement sur la structure globale de dépendances. Dans le chapitre 3, nous présentons notre architecture AxSEL un intergiciel contextuel et autonome pour le chargement de services dans des environnements pervasifs. AxSEL repose sur une représentation des dépendances flexible et expressive intégrant les niveaux d'exécution et de déploiement, l'aspect multi-fournisseur et les propriétés non fonctionnelles des services et composants. Au dessus de ce modèle, nous fournissons un ensemble d'heuristiques de déploiement autonome et contextuel.

Chapitre 3

AxSel, un intergiciel pour le déploiement autonome et contextuel des applications orientées services

Dans le chapitre précédent, nous avons mis en relief trois besoins fondamentaux à satisfaire : la globalité et la flexibilité de la vue ainsi que l'autonomie de la décision. Notre objectif est de fournir un modèle de déploiement autonome et contextuel pour les applications orientées services, qui répond à ces trois besoins. Dans ce chapitre, nous présentons notre contribution dans le domaine du déploiement des applications orientées services composants qui consiste d'abord à fournir un modèle de dépendances global et expressif, ensuite un modèle de contexte dynamique pour capturer et représenter les données pertinentes et enfin sur la base de ces deux derniers proposer des heuristiques pour le déploiement autonome. Dans la section 3.1, nous axons notre problématique autour des entités du déploiement, et fixons nos objectifs et hypothèses. Ensuite, nous proposons une architecture orientée services pour le chargement contextuel et en détaillons les services cœur et les interactions dans la section 3.2. Dans la section 3.3, nous introduisons un modèle d'application fournissant une vue globale, flexible et expressive. Ensuite, dans la section 3.4 nous présentons le modèle de contexte dynamique considéré. Sur la base de ces modèles nous proposons en section 3.5 un ensemble d'heuristiques de déploiement contextuel d'application. Enfin, la section 3.6 récapitule nos apports par rapport aux objectifs de recherche initiaux.

3.1 Autonomie et contextualisation en environnements intelligents

Dans cette section, nous présentons notre contribution dans le domaine du déploiement contextuel d'applications dans les environnements intelligents. D'abord, nous donnons les entités du déploiement logiciel d'une manière générale et les propriétés qui lui sont conférées lorsqu'il est appliqué dans un environnement intelligent. Ensuite, nous détaillons les paradigmes et les principes proposés pour réaliser la contextualisation des applications à déployer. Nous adoptons une approche intergicielle prenant en charge la capture, la représentation, la notification et le raisonnement sur le contexte et libérant l'utilisateur et l'application de ces tâches. Enfin, nous énumérons les apports de notre approche et énonçons les hypothèses considérées.

3.1.1 Entités du déploiement

Le déploiement logiciel est au cœur d'un mécanisme mettant en jeu un ensemble d'entités. Les activités de déploiement ont été identifiées dans la section 2.1.2 du chapitre 2. Pour rappel, il s'agit de l'ensemble des étapes suivantes : la livraison, l'installation, la configuration, la planification, la préparation, l'activation, le lancement, le chargement, la mise à jour, l'adaptation, la désactivation, la désinstallation et le retrait. Nous nous focalisons principalement sur les activités de chargement et d'adaptation, le reste est délégué à la plate-forme de déploiement sous-jacente. Nous illustrons dans la figure 3.1 les principales entités du déploiement.

- *Unité.* L'unité de déploiement est la ressource logicielle qui est au cœur des mécanismes de déploiement et qui subit ses actions.
- *Cible.* La cible du déploiement est l'ensemble des ressources matérielles locales ou distantes concernées par le déploiement. Dans notre cas, nous nous intéressons principalement au

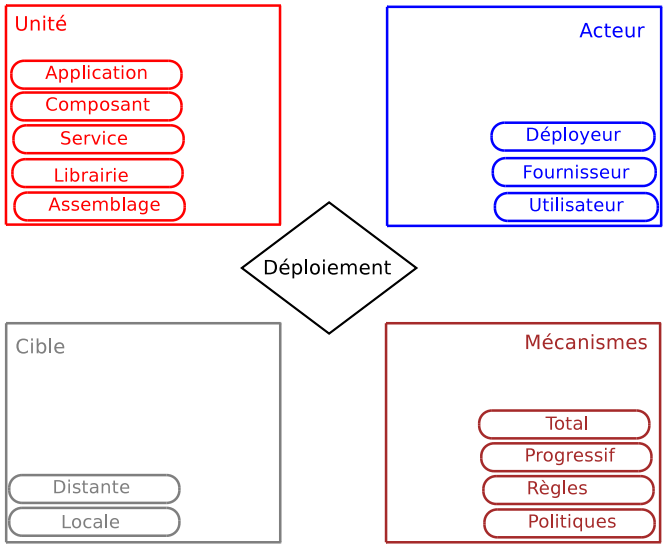

FIG. 3.1 – Entités du déploiement

chargement sur un périphérique local à partir de sources distantes.

- *Acteur.* L'acteur du déploiement représente les catégories de personnes, ou d'applications qui sont impliqués dans les activités du déploiement. Il peut s'agir de l'utilisateur et propriétaire du périphérique qui décide d'installer une application sur sa machine, ou du fournisseur du logiciel qui procède à des opérations de mise à jour chez les clients.
- *Mécanismes.* La manière avec laquelle est réalisée le déploiement repose sur l'existence de mécanismes particuliers ou de politiques. Ceux-ci répondent aux questions quelles unités déployer ? sur quelles cibles ? compte tenu des circonstances ?

Le déploiement logiciel met en jeu des unités de déploiement dans un domaine cible, selon des mécanismes qui décident selon les contraintes contextuelles des unités à déployer et de leur emplacement.

3.1.2 Propriétés du déploiement en environnements intelligents

Dans cette thèse, nous abordons cette problématique au sein des environnements intelligents et explorons une solution qui se situe à la frontière de trois mondes. Nous proposons d'associer les architectures orientées services, les architectures orientées composants et les environnements pervasifs. En les combinant ensemble, nous mettons à profit les apports des deux premiers en faveur de la réalisation de l'intelligence ambiante.

D'abord, l'intelligence ambiante aspire à un environnement dans lequel les fonctionalités sont accessibles à n'importe qui, n'importe où, d'une manière non intrusive, invisible et adaptée aux besoins de l'utilisateur [132], [117]. Ensuite, les apports amenés par le développement logiciel orienté composants et services comme la séparation des préoccupations, la modularité, la facilité de configuration et de mise à jour en font le meilleur élu pour s'adapter aux environnements intelligents. Cependant, afin de fournir les fonctionalités souhaitées, plusieurs contraintes contextuelles telles que la haute mobilité, la variabilité des contextes, la rareté des ressources

matérielles, et l'hétérogénéité des ressources, sont à prendre en compte [66], [118].

Enfin, réaliser des adaptations dynamiques nécessite la flexibilité et la dynamicité du système. Par ailleurs, ces adaptations doivent demeurer invisibles à l'utilisateur et ne pas solliciter son intervention, ce qui implique le besoin de soustraire les tâches de collecte des données pertinentes et de prise de décision à l'utilisateur et à l'application. Pour relever ces défis de sensibilité au contexte, d'optimisation des ressources matérielles et d'invisibilité, nous greffons des propriétés sur les modèles de déploiement extraits. Nous positionnons le déploiement au sein de cette configuration et illustrons dans la figure 3.2 les propriétés de chacune des entités préalablement présentées.

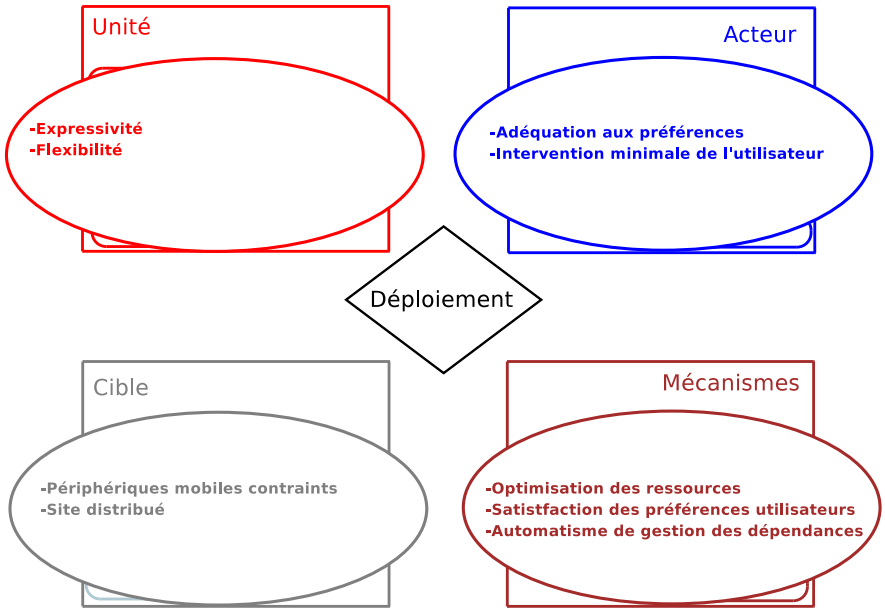

FIG. 3.2 – Propriétés du déploiement en environnements intelligents

Expressivité et flexibilité des unités de déploiement Par expressivité nous entendons l'intégration des propriétés fonctionnelles et non fonctionnelles, mais aussi, des dépendances à l'exécution avec d'autres services et de déploiement avec d'autres composants. La flexibilité est relative aux degrés de liberté de manipulation de la structure de l'application : ajout, suppression et modification des composants et services. Un support flexible permet de manipuler simultanément plusieurs aspects de l'application et de modifier sa structure dynamiquement en prenant les décisions adéquates. Les unités de déploiement et les applications nécessitent un support de représentation expressif et flexible adapté à l'intelligence ambiante.

Optimisation des ressources des cibles de déploiement Dans un contexte pervasif les cibles de déploiement sont petites pour en faciliter le transport, leur petite taille implique une **contrainte des ressources matérielles** de ces périphériques. Une prise en compte du contexte matériel est donc nécessaire pour une adéquation avec les environnements pervasifs. Les mécanismes de déploiement peuvent considérer des politiques d'économie de ressources.

Satisfaction des préférences utilisateurs Dans un environnement intelligent l'**adéquation aux préférences des utilisateurs** est un requis essentiel. En effet, cela permet de fournir une approche personnalisée plus axée sur les demandes et attentes des personnes. L'utilisateur peut renseigner ses préférences dans ses profils ou ses métadonnées en précisant par exemple ses applications et ses services préférés. Elles peuvent également être déduite automatiquement à partir de son comportement. Grâce à des mécanismes d'adaptation ces préférences sont considérées lors des prises de décision de déploiement pour éviter d'importuner l'usager. Cela permet de réduire l'intervention de l'utilisateur du terminal à des tâches ponctuelles de lancement du déploiement d'une application ou de modification de ses préférences.

Autonomie et adaptation dynamique au contexte Un système autonome est capable de prendre ses propres décisions avec **une intervention minimale de l'utilisateur** voire absente. L'autonomie est une solution pour atteindre la non intrusion d'un système. Elle permet d'alléger la charge de l'utilisateur et de lui soustraire certains traitements. Pour réaliser cela, il est nécessaire de fournir des mécanismes pour **automatiser la gestion des dépendances** et le processus de déploiement d'une application.

D'autre part, il est essentiel de fournir des mécanismes d'adaptation contextuelle prenant en compte les conditions du déploiement telles que la limite des ressources matérielles et les préférences utilisateurs. Des mécanismes d'écoute et d'interprétation du contexte sont exploités pour sonder le contexte à l'exécution et réagir aux changements pertinents. Ces traitements doivent également être transparents à l'utilisateur.

Dans ce qui suit, nous répondons aux problématiques du déploiement dans les environnements intelligents et aux objectifs de recherche sous-jacents en donnant les paradigmes et les principes que nous proposons pour construire notre approche de déploiement autonome et sensible au contexte. Ensuite, nous présentons les contributions apportées et les hypothèses considérées.

3.1.3 Déploiement contextuel et autonome d'applications

Afin de réaliser un déploiement contextuel d'applications plusieurs paramètres sont pris en compte. D'abord, la structure de l'application doit être expressive et flexible pour se prêter aux adaptations dynamiques souhaitées. Ensuite, les événements du contexte doivent être représentés de manière adéquate et remontés dynamiquement à travers des mécanismes de capture. Enfin, les mécanismes de contextualisation doivent assurer une prise de décision autonome sur la base de la vue globale et des contraintes contextuelles.

Vue globale et flexible pour toutes les dépendances d'une application Les approches orientées services et composants reposent sur un modèle selon lequel des fournisseurs publient leurs composants et services dans des dépôts. Les clients y accèdent et y puisent les services ou composants qu'ils désirent. Plusieurs fournisseurs peuvent offrir un même service ou composant. Ceux-ci possèdent des propriétés non fonctionnelles différentes relatives respectivement aux contextes de déploiement et d'exécution. D'autre part, les paradigmes composant et service

se basent sur les concepts de dépendances entre ceux-ci. Un composant dépend d'autres composants et services et inversement. Nous proposons de représenter la structure de dépendances d'une application d'une manière globale.

Globalité de la vue L'intégration de plusieurs aspects dans une unique vue pluridimensionnelle nous permet d'abord de répondre au besoin d'expressivité, ensuite de considérer les multiples facettes de l'application et d'appliquer les mécanismes de déploiement contextuel adéquats. Nous proposons une vue globale rassemblant les aspects de déploiement relatifs aux composants, d'exécution relatifs aux services, mais aussi les propriétés non fonctionnelles de ceux-ci, essentielles pour une adaptation au contexte, enfin, nous représentons également l'aspect multi-fournisseur inhérent aux architectures orientées services et composants.

- *Aspects liés aux fournisseurs multiples.* Plusieurs fournisseurs peuvent offrir un même service ou composant. Ceux-ci peuvent avoir des propriétés différentes comme la version, le nom symbolique, la taille mémoire, etc. L'aspect multi-fournisseur offre une information pertinente qui fournit une opportunité de choix multiple à la place d'un choix déterministe des services et composants à charger sur le périphérique. Nous proposons de prendre en compte cette information essentielle et d'intégrer l'aspect multi-fournisseur et multi-dépôt dans la vue globale de l'application.
- *Aspects liés au déploiement.* Les composants logiciels sont fournis dans des dépôts pour être déployés sur des machines cibles clientes. Le développeur ou fournisseur d'un composant le décrit avec des métadonnées. Celles-ci incorporent des données fonctionnelles telles que les dépendances envers d'autres composants, mais aussi des données non fonctionnelles relatives au contexte de déploiement telles que la taille mémoire du composant.
- *Aspects liés à l'exécution.* Les services apparaissent à l'exécution une fois les composants déployées. Un service possède des informations pertinentes nous renseignant sur son contexte d'exécution, relevant des aspects fonctionnels et non fonctionnels de dépendances avec d'autres services et composants. L'intégration des aspects fonctionnels et non fonctionnels liés à l'exécution des services permet d'avoir une approche verticale dans une même vue globale, cette approche part d'une base de déploiement et se prolonge pour exprimer l'exécution.

Flexibilité de la vue Par flexibilité nous entendons l'aptitude des éléments d'une application à être modifiée et sa capacité d'extensibilité par ajout ou retrait de ses éléments. Dans un environnement intelligent, le contexte change constamment, cela inclut l'apparition, la disparition de nouveaux dispositifs électroniques, le changement de données des composants ou des services, la disponibilité des ressources matérielles sur les plates-formes cibles de déploiement et la variation des préférences utilisateurs. Afin de s'adapter aux fluctuations des environnements intelligents il est essentiel que les applications soient flexibles.

- *Dynamicité.* Les variations observées dans un environnement pervasif sont multiples, elles proviennent de plusieurs sources et ont de l'influence sur les composants, les services et les machines cibles. Au sein d'un dépôt de composants de services les données peuvent changer, ainsi, lorsqu'un composant a ses données contextuelles modifiées cette variation est répercutée dynamiquement sur la vue globale des dépendances de l'application. De

la même manière lorsque les données d'un service sont modifiées dans un annuaire ou dans un environnement d'exécution, ses données sont modifiées dans la vue globale. La dynamicité concernent également les dépendances entre services et composants. Grâce au caractère dynamique de la vue globale nous pouvons réaliser des adaptations contextuelles et dynamiques.

- *Extensibilité.* Les changements observés peuvent concerner les dépôts de composants et les services. L'apparition d'un nouveau composant ou d'un nouveau service au sein du dépôt est une information pertinente qui peut influencer le mécanisme de décision. La répercussion de cette apparition et de la répercuter sur l'application déjà déployée peut être réalisée si la structure de l'application est extensible et supporte l'ajout de nouveaux éléments tels que des composants, des services ou des dépendances. Il en est de même pour le retrait d'élément du dépôt.

Une gestion dynamique du contexte Dans le but de libérer l'utilisateur des tâches répétitives et de faciliter le déploiement de l'application, l'intergiciel se charge des opérations de capture des données contextuelles à partir des sondes et des métadonnées, et de la représentation de celles-ci sous un format compréhensible et exploitable. Les sources de ces données sont les ressources matérielles et principalement la mémoire virtuelle, le profil utilisateur, et le dépôt de composants et de services. Les sources et les données contextuelles sont connues de l'intergiciel uniquement et non pas de l'application. Les contraintes de déploiement proviennent de l'utilisateur lorsqu'il affiche ses préférences des services souhaités et les limites des ressources du périphérique électronique à partir desquelles il sera notifié. A travers l'utilisation des données collectées à partir de ces sources l'intergiciel opère une prise de décision qui respecte les contraintes énoncées.

Une contextualisation autonome et dynamique Par contextualisation nous entendons la capacité d'une application à changer selon les contraintes contextuelles. La contextualisation dynamique de l'application repose sur sa réactivité aux événements de l'environnement, et sa proactivité par anticipation du chargement dans les cas pertinents de prise de décision. Dans un contexte intelligent l'adaptation au contexte inclut l'optimisation des ressources matérielles et le respect des préférences utilisateur. L'approche intergicielle est intéressante car elle permet d'abstraire le raisonnement contextuel à l'application et de rendre la tâche invisible à l'utilisateur. Nous portons notre intérêt principalement à la contextualisation structurelle de l'application et ce à travers le choix de services et de composants à charger parmi l'ensemble des éléments de l'application. Lorsque l'application est inadaptée aux contraintes car trop volumineuse par exemple, certains de ses services et composants sont chargés localement, le reste sera chargé progressivement.

- *Réactivité.* Les mécanismes de surveillance, d'introspection et de notification du contexte permettent de tenir au courant l'intergiciel des changements contextuels pertinents. Des sondes déployés sur la mémoire du dispositif, le dépôt de composants et de services et le profil utilisateur renseignent à chaque changement des modifications observés. Le comportement de l'intergiciel est influencé selon la nature et la pertinence de la donnée collectée.

Lorsqu'un événement notifiant de l'atteinte de la limite de la mémoire est levé l'intergiciel peut choisir de décharger certains services et composants déjà chargés. Inversement, lorsque la mémoire est libérée un événement notifie l'intergiciel, celui-ci peut décider de charger le reste des services et composants de l'application. La réactivité obéit au modèle événement condition action et déclenche à chaque événement capturé une action.

- *Proactivité.* La variabilité du contexte nécessite de l'intergiciel une adaptation adéquate à chaque cas rencontré. Ainsi, le comportement peut être dicté par l'importance des contraintes contextuelles. Par exemple, dans les cas de contraintes de ressources matérielles une stratégie les optimisant est mise en place, dans d'autres cas, d'autres paramètres tels que les préférences utilisateurs sont considérées. D'autre part, anticiper les décisions ultérieures permet de réaliser des gains de performances. L'anticipation du chargement consiste à précharger les composants et services sur le dispositif en vue d'une utilisation future. Grâce à cette technique nous évitons à l'intergiciel de réaliser les traitements de recherche et de rapatriement de service et ce que cela engendre comme coût.

3.1.4 Contributions et hypothèses

Contributions Nous proposons une architecture basée sur l'ensemble des paradigmes énoncés, soient la vue globale et flexible et la contextualisation autonome. AxSeL fournit les apports suivants :

- *Une architecture intergicielle pour le déploiement autonome d'applications orientées services composants.* AxSeL repose sur un niveau contextuel responsable de la capture de données de bas niveau à partir des sondes sur les ressources matérielles, les métadonnées des services, des composants, et de l'utilisateur. Le raisonnement sur les données collectées est intégré au niveau de l'intergiciel de manière à libérer l'application et l'utilisateur de cette charge.
- *Une vue globale expressive et flexible pour la représentation de l'application.* Nous représentons les dépendances en services et en composants d'une application sous la forme d'un graphe bidimensionnel où les dimensions représentent les niveaux d'exécution et de déploiement, les nœuds les services et les composants, et les arcs les dépendances. Les propriétés contextuelles des services et composants sont exprimées au niveau des nœuds du graphe. Cette vue est également flexible, ainsi, lorsque des variations sont perçues dans l'environnement, il est possible d'ajouter, de retirer et de modifier dynamiquement les nœuds, les dépendances et les niveaux du graphe.
- *Une gestion dynamique du contexte.* Elle est basée sur l'introspection et l'écoute des événements générés par les sondes déployées sur les sources de données, et la génération des événements pertinents qui déclenchent les prises de décision. Les sources considérées incluent l'utilisateur, le terminal et le dépôt à partir duquel les services et composants sont puisés. AxSeL fournit un modèle de représentation et de capture du contexte.
- *Une contextualisation autonome et dynamique réactive et proactive.* Sur la base des données collectées par la couche contextuelle, de la représentation de l'application et des heuristiques de raisonnement, AxSeL opère une prise de décision autonome et multi-critères visant à choisir progressivement les services et composants à charger localement.

AxSeL opère aux étapes de chargement et d'adaptation.

Hypothèses

1. Nous supposons l'existence d'une couche responsable de la communication entre les périphériques et les fournisseurs de services. Nous ne nous intéressons pas aux protocoles de communication mais supposons que le protocole supporté gère également les connexions à distance.
2. Notre intergiciel capture le contexte matériel mémoire du périphérique, les préférences utilisateur, et les propriétés fonctionnelles des applications et des services. AxSeL exploite des API permettant le dialogue avec la machine virtuelle et le système d'exploitation pour réaliser la capture du contexte matériel.
3. Dans le but de déployer sur un dispositif contraint une application, AxSeL réalise un chargement progressif. Au fur et à mesure les composants et services d'une application sont ramenés localement. Celle-ci fonctionnera lorsque l'intégralité de ses dépendances est résolue. Par conséquent, la panne de l'application en cours de chargement est un comportement attendue par notre approche.
4. Nous supposons que chaque terminal est utilisé par une seule personne. Ceci est réalisable principalement pour les terminaux mobiles qui sont généralement dédiés à l'usage par un seul individu. En supposant cela, nous évitons les problèmes de conflits lors de l'application des stratégies de déploiement sur les applications à déployer.
5. Notre intergiciel réalise une adaptation structurelle des applications orientées services. Cela entraîne le choix de services à la place d'autres selon des propriétés non fonctionnelles. Dans AxSeL nous n'avons pas étudié les aspects de cohérence de l'application.

Conclusion Dans cette section, nous avons identifié les propriétés du déploiement logiciel dans un environnement intelligent. Ensuite, nous avons dégagé les objectifs de recherche poursuivis ainsi que les principes et les concepts proposés. Enfin, nous avons fixé les hypothèses sur lesquelles nous basons notre approche. Dans la section suivante, nous proposons AxSeL un intergiciel autonome et contextuel pour le déploiement des applications orientées services. Nous présentons d'abord une vue d'ensemble, ensuite nos modèles de services, de composants, d'applications, et contexte.

3.2 Architecture de chargement contextuel de services

Dans cette section, nous esquissons une vue générale de l'architecture AxSeL. Elle repose sur un modèle de service, de composant et de contexte et sur une couche d'heuristiques de déploiement. Nous donnons d'abord les détails des services cœur d'AxSeL ainsi que les interactions opérées afin de réaliser le chargement et l'adaptation des applications. Ensuite, nous présentons les modèles relatifs au service, au composant, à l'application et au contexte.

La figure 3.3 positionne AxSeL dans une architecture générale en couches. En partant du bas nous trouvons le système d'exploitation par dessus duquel vient se superposer l'intergiciel

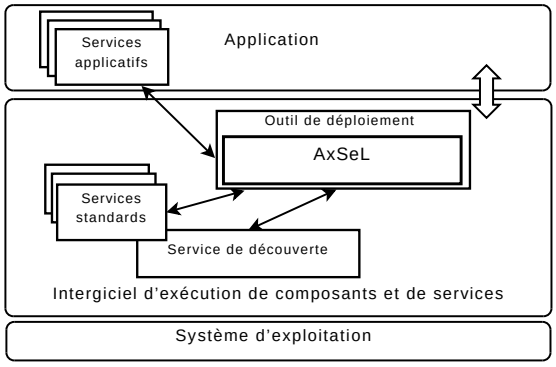

FIG. 3.3 – Architecture générale en couche

de déploiement et d'exécution de services et de composants. AxSeL fait partie de l'outil de déploiement de cet intergiciel et bénéficie des services standards en exécution fournis par celui-ci tels que le service de découverte et d'autres services applicatifs. L'application se trouve au niveau de la plus haute couche et communique avec l'intergiciel d'exécution des services et composants.

3.2.1 Description des services d'AxSeL

AxSeL est conçue selon le paradigme orienté service afin de permettre plus de flexibilité lors de sa gestion et de son déploiement. La figure 3.4 illustre en détail les modèles et services d'AxSeL. Elle est composée de trois services : le service d'extraction des dépendances, le ser-

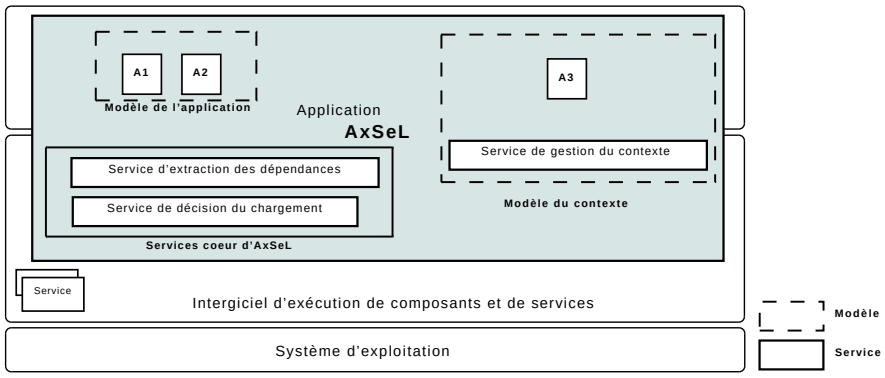

FIG. 3.4 – Architecture générale détaillée

vice de gestion du contexte et le service de décision du chargement. AxSeL possède aussi deux modèles : le modèle d'application et de contexte. Le service d'extraction des dépendances trace l'ensemble des services et composants nécessaires à l'application à partir d'une spécification statique en une représentation multidimensionnelle, expressive et flexible : le modèle d'application. Ensuite, le service de gestion du contexte déploie un ensemble de mécanismes de capture et de représentation des données pertinentes pour fournir un modèle de contexte dynamique et extensible. Enfin, le service de décision encapsule les stratégies de déploiement essentielles pour contextualiser l'application d'une manière autonome et dynamique. En optant pour une

approche intergicielle, nous concentrons les tâches de déploiement et de prise de décision au sein de l'intergiciel même, libérant ainsi l'utilisateur de les effectuer et l'application de gérer les données propres à l'exécution. Dans ce qui suit, nous détaillons brièvement les modèles d'application et de contexte, ensuite, les services d'AxSeL.

Un modèle d'application Les applications orientées services composants sont généralement fournies dans des dépôts auxquels les clients accèdent. Elles y sont décrites grâce à des descripteurs statiques qui renseignent leurs dépendances fonctionnelles et non fonctionnelles. Ces applications offrent une conception hybride en combinant les composants relatifs à la phase de déploiement avec les services relatifs à la phase d'exécution. AxSeL représente les applications orientées services composants sous la forme d'un graphe bidimensionnel intégrant les aspects de déploiement et d'exécution, les données contextuelles non fonctionnelles et l'aspect multi-fournisseur. Le graphe permet une manipulation intuitive des dépendances mais aussi la possibilité de gérer l'application dans son intégralité. Les nœuds du graphe représentent les services et composants, les arcs les dépendances, et les niveaux l'environnement auquel appartient le nœud. AxSeL puisent les services et composants de chaque application à partir de plusieurs dépôts à la fois, cet aspect multi-fournisseur est exprimé dans le graphe de dépendances grâce à des opérateurs logiques. Le modèle d'application d'AxSeL est extensible et dynamique et permet l'ajout, le retrait et la modification des nœuds, des arcs ou des niveaux. Cette vision globale fournit un choix non déterministe de services et de composants à charger selon les contraintes données. Le modèle d'application AxSeL est présenté dans la section 3.3.

Un modèle du contexte Le contexte que nous considérons est à la fois statiquement renseigné et dynamiquement recueilli en cours d'exécution. Nous évitons à l'application la gestion de l'hétérogénéité impliquée par les données recueillies à travers un modèle du contexte capable de capturer les données, de les interpréter et de communiquer uniquement celles qui sont compréhensibles et pertinentes pour l'application. AxSeL utilise un modèle orienté objet pour représenter le contexte. Les données sont sondées à partir des préférences utilisateurs, des dépôts de services et de composants, et du terminal, à travers des API spécifiques capables de remonter l'information capturée. Les événements générés peuvent avoir plusieurs types selon la source contextuelle. Le détail du modèle du contexte est donné dans la section 3.4.

Service d'extraction des dépendances Les approches orientées services composants sont basées sur un concept fort qui est la dépendance entre ceux-ci. Ces dépendances sont renseignées dans les métadonnées des services et des composants ou au sein des descripteurs de dépôts où ils sont hébergés. Ces derniers décrivent d'une manière unitaire les services et composants en ne mentionnant que leurs dépendances immédiates. Afin de déployer une application il est nécessaire de résoudre l'ensemble de ses dépendances en services et en composants. En parcourant le descripteur de dépôt, AxSeL procède à l'extraction récursive des dépendances de chaque service et composant et en les inclue au fur et à mesure dans une représentation orientée graphe. Lorsque l'ensemble des dépendances est tracé, la construction du graphe de dépendances global est alors finie. Le détail de ce service est présenté dans la section 3.5.1.

Service de gestion du contexte Dans un environnement intelligent les sources contextuelles sont hétérogènes et multiples. AxSeL propose le service de gestion du contexte pour effectuer dynamiquement l'observation, l'introspection de ces sources et l'interprétation du contexte afin d'informer des modifications pertinentes. Dans le but de fournir un déploiement contextuel, AxSeL considère les limites matérielles des terminaux, les préférences utilisateurs et le contexte requis par les services et les composants. Ces informations sont puisées à partir du descripteur de services et de composants, des préférences utilisateurs et du terminal. Celles-ci sont capturées à travers des API spécifiques pour chacune des sources contextuelles. Le service de gestion du contexte collecte les données à partir des API sondes, et réagit en cas de changements pertinents en générant un événement contextuel. Les aspects de gestion dynamique du contexte sont présentés dans la section 3.4.

Service de décision du chargement Ce service confronte un contexte requis à satisfaire avec un contexte fourni à respecter. AxSeL parcourt le graphe de dépendances et évalue les données contextuelles des services et des composants par rapport aux contraintes du terminal et de l'utilisateur. Pour chaque service du graphe de dépendances, une décision de chargement est prise. Une nouvelle décision de chargement peut avoir comme conséquences d'éventuels chargements/déchargements de services et de composants. Le chargement se fait d'une manière progressive jusqu'au rapatriement et exécution de toutes les dépendances de l'application. La décision de chargement peut obéir à plusieurs stratégies relatives aux politiques adoptées, par exemple, dans le cas d'une politique d'économie de ressources matérielles une stratégie mettant en priorité celles-ci est choisie. Enfin, AxSeL anticipe le chargement de certains services en vue d'une utilisation future ce qui optimise les performances du dispositif en lui économisant le coût d'accès, de rapatriement et de résolution des dépendances des composants et services. Le détail des mécanismes de décision est donné dans la section 3.5.2.

3.2.2 Interactions entre les services d'AxSeL

Nous avons préalablement détaillé le rôle de chaque service d'AxSeL. A présent, nous illustrons les interactions prenant place entre ces services pour réaliser le chargement autonome et contextuel des applications. D'abord, AxSeL réalise l'extraction des dépendances pour obtenir une vue globale sous la forme d'un graphe contextuel bidimensionnel de services et de composants. L'extraction des dépendances est suivie par l'étape de décision contextuelle de chargement. La collecte du contexte pertinent et essentiel à la prise de décision est réalisée par le service de gestion du contexte.

La figure 3.5 illustre les services et modèles d'AxSeL qui sont impliqués dans l'étape d'extraction des dépendances. Les services et composants peuvent être hébérgés sur des dispositifs mobiles tels qu'un assistant personnel ou un ordinateur portable. Le service d'extraction des dépendances accède à un ou plusieurs dépôts de services locaux ou distants, et construit à partir des descripteurs des dépôts un modèle d'application sous la forme d'un graphe flexible et contextuel. Le graphe représente d'une manière globale l'application à charger incluant les services, les composants, les liens entre eux, et leurs données fonctionnelles et non fonctionnelles.

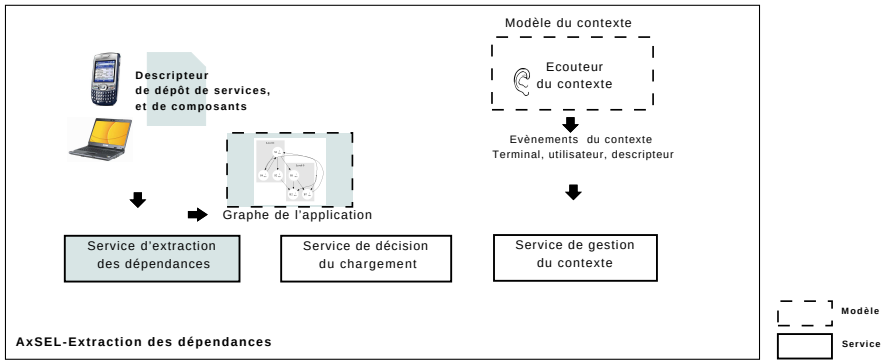

FIG. 3.5 – Extraction des dépendances d'une application

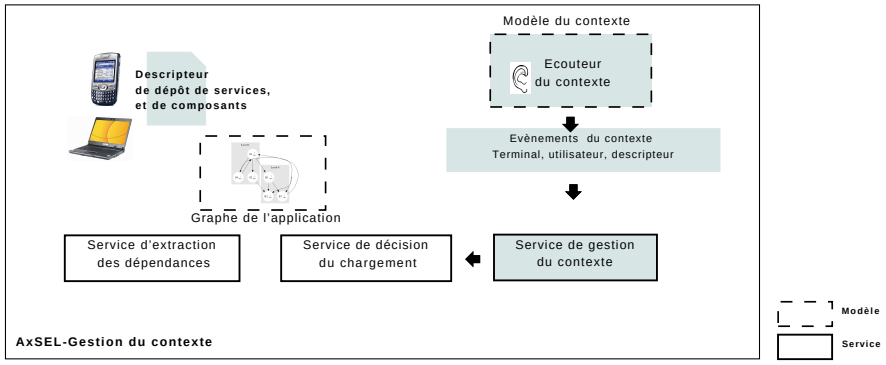

FIG. 3.6 – Gestion du contexte

Dans la figure 3.6 nous illustrons les services d'AxSeL qui sont impliqués dans la gestion du contexte. Le service de gestion du contexte emploie un écouteur du contexte pour collecter dynamiquement les données à partir des métadonnées de l'utilisateur, du descripteur du dépôt et du API sondes sur le terminal. Le modèle de contexte fournit la représentation des données de toutes les sources. Lorsqu'une modification est observée au niveau de ces sources de données des événements sont générés. Ceux-ci sont dépendants de la source qui les déclenchent. Tous les événements contextuels de nature matérielle ou autre sont acheminés vers le service de décision du chargement.

Au sein d'AxSeL, la prise de décision du chargement est réalisée par le service de décision du chargement. Ce dernier prend le graphe extrait à partir de l'étape précédente, les données contextuelles fournies par le service de gestion du contexte et applique une coloration du graphe dirigée par les contraintes. Le graphe contextuel de l'application est communiqué à l'outil de déploiement de l'intergiciel d'exécution des services et composants pour qu'il réalise son chargement. Le mécanisme de la prise de décision est donné figure 3.7. AxSeL réalise le déploiement initial de l'application mais aussi son adaptation, ainsi lorsqu'au cours de l'exécution d'une application le service de gestion du contexte notifie le service de décision du chargement d'un changement, celui-ci engage une nouvelle prise de décision afin d'offrir la configuration la plus optimisée de l'application.

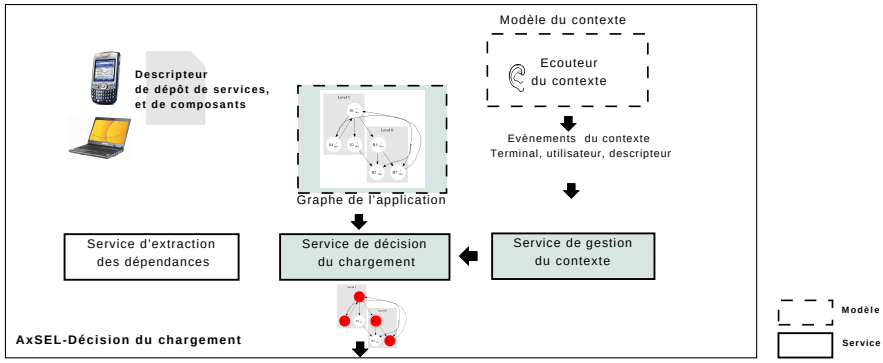

FIG. 3.7 – Décision du chargement

Conclusion Dans cette section nous avons présenté une vue d'ensemble de l'architecture AxSeL pour le déploiement autonome d'applications orientées services. Nous avons également détaillé chacun des services et modèles constituant notre architecture et les principales interactions prenant place pour réaliser le chargement contextuel. Ainsi nous avons mis en évidence trois principaux services : le service d'extraction des dépendances, le service de gestion du contexte et le service de décision du chargement, et deux modèles : le modèle d'application et le modèle du contexte. Dans la section suivante, nous présentons la couche de modélisation haut niveau sur laquelle AxSeL repose. Nous proposons un modèle expressif et flexible pour représenter les applications orientées services et composants, ainsi qu'un modèle dynamique pour représenter les données du contexte.

3.3 Une vue globale expressive et flexible

L'approche à base de services amène une vue d'exécution où les services sont mis à disposition des clients dans des annuaires pour être utilisés. Quant à l'approche à base de composants elle amène une vision de déploiement où ceux-ci sont hébergés dans des dépôts en vue de leur déploiement sur les plates-formes clientes. Les services et composants possèdent des dépendances contextuelles fonctionnelles et non fonctionnelles avec d'autres services et composants. Le paradigme des dépendances est essentiel dans ces approches puisqu'il leur permet d'importer des fonctionnalités qu'ils ne possèdent pas. Pour déployer un composant il est nécessaire de résoudre et de satisfaire ses dépendances. Enfin, les environnements intelligents permettent à n'importe qui de fournir des services ou des composants n'importe où et à n'importe qui d'autre entraînant ainsi la multiplicité des fournisseurs. Cela consiste en une information pertinente à considérer. En effet, si un service ne respecte pas les contraintes de déploiement, il est possible de retrouver son équivalent qui soit adéquat.

Unifier l'ensemble de ces aspects dans un paradigme commun fournit une vue bidimensionnelle et expressive grâce à laquelle il est possible de gérer simultanément les aspects de déploiement et d'exécution d'une application et de représenter les aspects contextuels fonctionnels, non fonctionnels et multi-fournisseur. Outre l'expressivité, il est essentiel que la vue

globale soit flexible. En effet, il est pertinent de considérer les possibles variations de l'environnement telles que l'apparition, la disparition des services et composants et la modification de leur métadonnées. Ces variations nécessitent l'extensibilité de la vue de l'application à l'exécution, par ajout et retrait de ses éléments ainsi que sa dynamique par modification de leurs valeurs.

Dans cette section, nous proposons une nouvelle approche expressive et flexible orientée graphe pour la représentation des dépendances des applications orientées services composants. D'abord, nous définissons les concepts de service, de composant et d'application en nous basant sur la recherche bibliographique préalablement réalisée et présentons un modèle indépendant de la plate-forme et basé sur les paradigmes des architectures orientées services. Ensuite, nous proposons notre graphe bidimensionnel et flexible. Pour modéliser les services, les composants et les applications nous utilisons les diagrammes de classes UML [45].

3.3.1 Modélisation de composant, service

Nous nous basons sur des modèles qui répondent aux contraintes de séparation des préoccupations, de flexibilité et d'expressivité nécessaires dans les environnements intelligents. D'abord, la séparation des préoccupations [40] permet de distinguer les fonctionnalités logicielles permettant ainsi des capacités de réutilisation, de modularité, d'injection des dépendances, d'inversion de contrôle et de souplesse des dépendances. En se basant sur une architecture logicielle orientée services, AxSeL bénéficie de ces paradigmes. Ensuite, le besoin d'expressivité amène la nécessité de rompre le principe de transparence en mettant en avant les données contextuelles des services et des composants. Il est alors possible d'exploiter ces données contextuelles pour influencer le processus de prise de décision, pour le choix des services et composants à déployer selon les contraintes contextuelles. Enfin, la flexibilité offre aux applications la possibilité d'adaptation aux changements du contexte par modification ou extensibilité dynamique de ses services et composants.

Nous avons préalablement étudié dans les sections 2.3.3.2 et 2.3.3.3 du chapitre 2 les définitions proposées par la littérature pour les composants et les services en nous référant à chaque fois à trois travaux. A travers les tableaux synthétiques mis en évidence nous dégageons les définitions adoptées par AxSeL pour les composants et les services. Ensuite, nous proposons notre propre modèle de service composant et de dépendances entre services composants.

Modèle de composant *Un composant est une unité logicielle conçue selon un modèle de conception. Il peut être décrit en vue de publication et de réutilisation par des tiers. Il est déployé d'une manière indépendante. Il possède des dépendances de contexte. L'accès se fait à travers des interfaces et selon un standard de composition spécifique. La composition se fait avec n'importe quel client, et ne nécessite pas de modification du composant* [122] [77] [92].

Nous représentons dans la figure 3.8 le diagramme de classes d'un composant et listons ci-après ses éléments :

- *Repository* : un dépôt de composants est un terminal hébergeant des composants mis à disposition des tiers. Ceux-ci y sont déposés accompagnés d'une description contextuelle fonctionnelle et non fonctionnelle précisant leurs caractéristiques et leurs dépendances de déploiement.

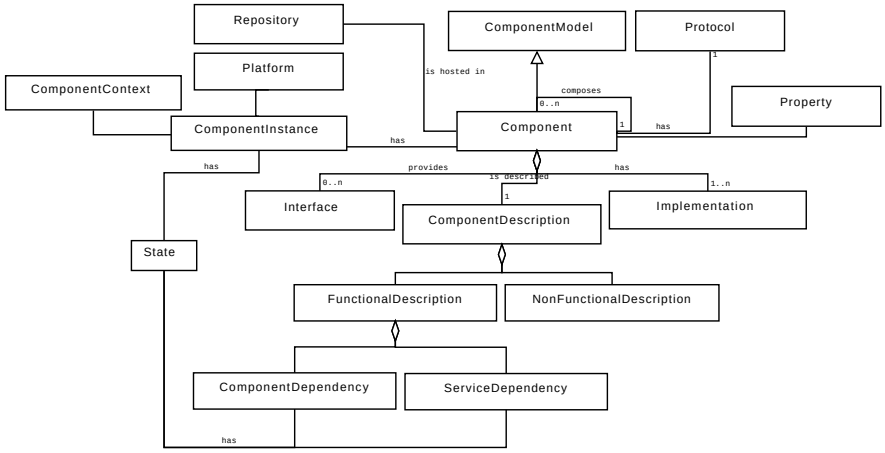

FIG. 3.8 – Représentation UML d'un composant

- *Protocol* : le protocole est le standard de communication utilisé par les composants pour communiquer, en local via la machine virtuelle par exemple ou à distance via les standards SOAP ou RPC.
- *Platform* : il s'agit de la plate-forme d'exécution sur laquelle est déployé le composant, elle obéit au modèle de composant et inclut l'ensemble de mécanismes du protocole de communication entre les instances de composants.
- *Implementation* : est l'implantation et l'ensemble du code encapsulés par le composant. L'implantation est le code métier des fonctionalités exportées par le composant à travers ses services.
- *Interface* : une interface permet l'accès aux fonctionalités enfouies dans le composant. Elle comporte l'ensemble d'opérations publiquement visibles du composant.
- *ComponentModel* : il s'agit du standard et de la technologie selon lesquels le composant est conçu. Cela impose un cadre de conception, des standards de développement et de composition, ainsi que la nécessité d'usage de la plate-forme d'exécution adéquate. Les technologies OSGi, EJB, CCM sont des modèles de composant.
- *ComponentInstance* : lorsqu'un composant est installé sur une plate-forme d'exécution, une instance de celui-ci est créé, il s'agit de l'objet relatif au composant. L'instance du composant déclare les interfaces fournis et les dépendances vers celles requises.
- *State* : cela correspond à l'état d'une instance de composant ou des dépendances. Une dépendance peut être satisfaite ou pas. Une instance de composant peut être active ou inactive.
- *ComponentContext* : lorsqu'un composant est déployé sur une plate-forme d'exécution, un contexte qui lui est propre est créé, ce contexte héberge l'instance du composant et gère ses états et offre une visibilité dessus. Il s'agit d'une sorte d'annuaire des objets en cours d'exécution sur la plate-forme.
- *ComponentDependency* : les dépendances de composants portent sur d'autres composants ou sur des interfaces en incluant leurs descriptions. Les interfaces peuvent être internes

au composant ou exporté par des composants externes. Lors du déploiement d'un composant ses dépendances peuvent avoir deux états : satisfaites ou non satisfaites. Une dépendance est satisfaite si l'élément de laquelle elle dépend est présent sur la plate-forme d'exécution, sinon elle est non satisfaite. Un composant ne pourra s'exécuter avec succès que si l'ensemble de ses dépendances sont satisfaites. Dans le cas où ses dépendances sont non satisfaites celui-ci ne pourra pas publier d'interfaces et ne sera pas visible dans le contexte du composant.

- *ComponentDescription* : il s'agit de la description fonctionnelle et non fonctionnelle accompagnant le composant. Elle précise leurs dépendances, le modèle du composant, le protocole de communication. Cette description peut être réalisée à travers des modèles de clé-valeur, des ontologies, des formats XML, etc. Elle est renseignée par le développeur du composant.
- *FunctionalDescription* : la description fonctionnelle est l'ensemble des données nécessaires à l'usage du composant. Cela inclut l'interface du composant et les dépendances nécessaires à son usage.
- *NonFunctionalDescription* : la description non fonctionnelle d'un composant est l'ensemble des données qualitatives ou quantitatives qui lui sont relatives. Cela peut inclure la confiance, la sécurité (qualitative), ou les ressources matérielles nécessaires à l'exécution du déploiement (quantitatives).
- *Property* : la propriété désigne une donnée non fonctionnelle accompagnant le composant et servant à le caractériser dans un environnement d'exécution. Lors de l'enregistrement d'un composant dans une plate-forme d'exécution, la spécification d'une propriété permet de le distinguer des autres.

Modèle de service *Un service est une entité autonome et indépendante de la plate-forme. Il désigne un comportement fourni par un composant à n'importe quel autre contractuellement. Le service est un mécanisme d'accès à des fonctionnalités. Il est fourni par n'importe quel composant ou fournisseur de services. Il est publié et découvert dans un annuaire de services. La relation entre deux services est un contrat réalisé à un instant donné. L'accès au service est réalisé par un couplage faible, à travers des interfaces et une politique de composition* [39] [87] [101].

Nous illustrons dans la figure 3.9 le modèle de service et listons ci-après ses éléments :

- *ServiceRegistry* : est un annuaire de services référençant l'ensemble des services publiés. L'instance du composant exportant le service se charge de l'enregistrer dans l'annuaire. La publication des services permet leur visibilité en vue d'utilisation par des tiers.
- *Protocol* : le protocole désigne un ensemble de mécanismes de composition et d'accès aux services.
- *ServiceProperty* : l'instance du composant enregistre le service dans l'annuaire de services en précisant des propriétés non fonctionnelles qui permettent d'identifier le service dans l'environnement d'exécution.
- *ServiceContext* : le contexte du service comporte toutes les données relatives à l'exécution du service. Ces données peuvent être fonctionnelles ou non fonctionnelles. telles que la fréquence d'utilisation d'un service ou les ressources matérielles qu'il consomme.

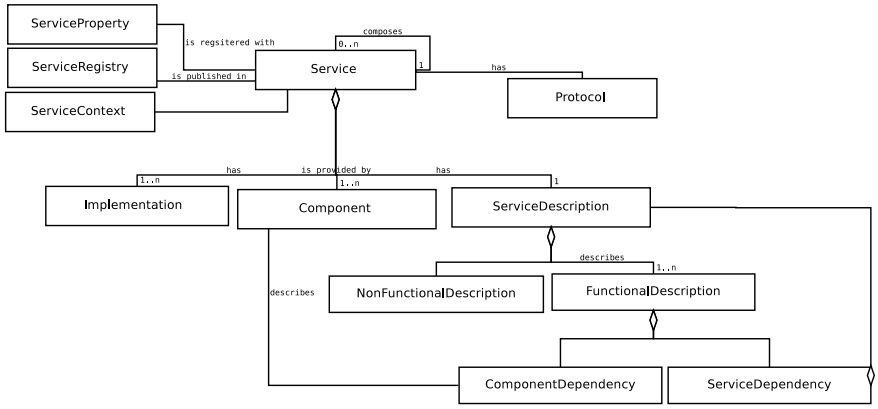

FIG. 3.9 – Représentation UML d'un service

- *ServiceDescription* : il s'agit de la description fonctionnelle et non fonctionnelle accompagnant le service. Elle précise les dépendances en composants et en services, le protocole de communication. Cette description peut être réalisée à travers des modèles de clé-valeur, des ontologies, des formats XML, etc. Cette description est censé être renseignée par le fournisseur du service.
- *FunctionalDescription* : la description fonctionnelle est l'ensemble des données nécessaires à l'usage du service. Cela inclut les dépendances nécessaires à son usage et les méthodes exportées.
- *NonFunctionalDescription* : tout comme pour le composant la description non fonctionnelle d'un service est l'ensemble des données qualitatives ou quantitatives qui lui sont relatives. Cela peut inclure la confiance, la sécurité (qualitative), ou les ressources matérielles nécessaires à l'exécution du déploiement (quantitatives).
- *Implementation* : est l'implantation du composant. Il s'agit du code métier des fonctionalités exportées par le composant à travers ses services.

Pour les usagers des services (applications, clients, etc.), un service est une interface permettant l'accès aux fonctionnalités fournies par le composant. Il renferme une liste de fonctions, une description spécifiant ses dépendances, les fonctionalités qu'il exporte, et le protocole d'accès nécessaire. D'autres données peuvent venir s'ajouter à ce descripteur. Ces données sont de deux types : fonctionnelles (méthodes, accès) et non fonctionnelles (ressources matérielles, qualité de service, version). Les descriptions sont généralement fournies par le développeur ou le fournisseur des services et des composants.

Modèle de service composant *Un service composant est un concept hybride qui réunit le composant et le service. Un composant est une unité logicielle encapsulant des fonctionalités. Il peut être déployé et exécuté. Il possède une description et des interfaces. Une interface correspond à un service. Les services permettent l'import et l'export des fonctionalités. Un composant est sujet à composition avec d'autres composants à travers des services importés et exportés. Un composant et ses services ne sont pas dédiés à un utilisateur spécifique.*

Dans la figure 3.10, nous présentons un diagramme des classes du modèle service composant

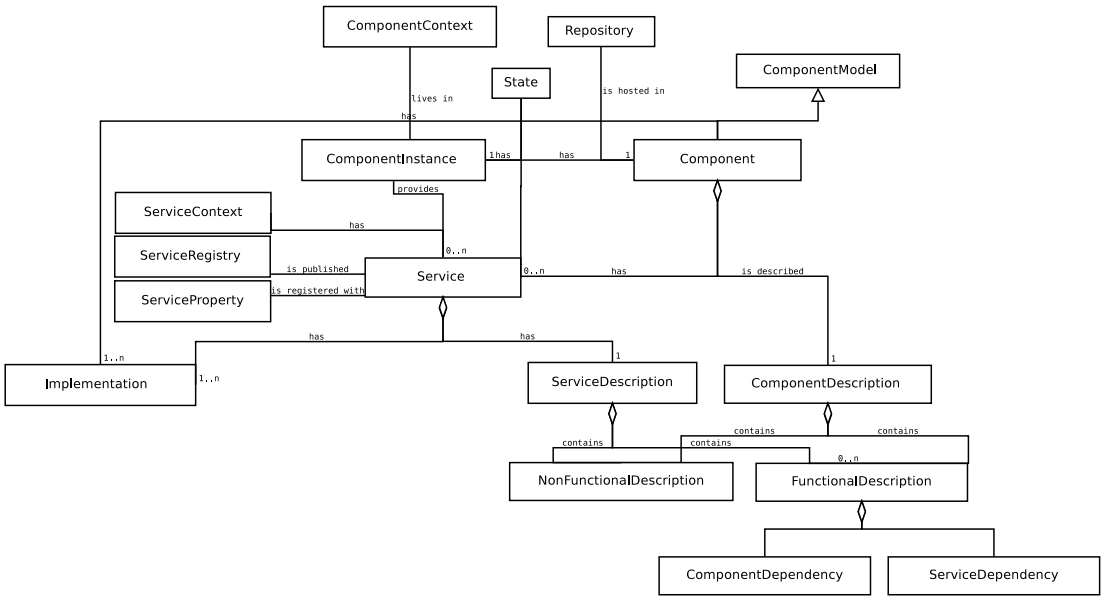

FIG. 3.10 – Modèle du service composant

proposé par AxSeL. Le composant apparaît comme une unité de déploiement alors que le service apparaît comme une unité de contractualisation plutôt orientée vers le comportement à fournir à l'étape d'exécution. Nous rassemblons les deux concepts en un seul : le service composant. Le composant intègre le code métier exporté par le service. Ce modèle hybride nous permet de gérer simultanément les aspects de déploiement liés aux composants et les aspects d'exécution liés aux services.

Dans cette section, nous avons introduit les modèles de services et de composants, et proposé notre propre modèle de service composant. Dans la section suivante, nous présentons la notion de dépendances entre services et composants.

3.3.2 Dépendances de services et de composants

Définition *Une dépendance est une relation mettant en jeu deux ou plusieurs éléments, et où le changement d'état d'un ou plusieurs de ces éléments peut générer le changement d'état des autres éléments. Les éléments de la dépendance peuvent être abstraits ou concrets.* [60]

AxSeL introduit le niveau de dépendances entre services dans le modèle de services composants. La dépendance entre services peut être assimilée à un contrat établi entre deux services où un service fournit une fonctionalité à un autre. La figure 3.11 présente une application AxSeL. Nous mettons en évidence les différentes dépendances possibles, et distinguons entre le niveau des composants et le niveau des services. Cette distinction nous permet de gérer indépendamment les services des composants en considérant leurs caractéristiques respectives. Les services fournissent une vue de haut niveau, il est possible de leur assigner des propriétés non fonctionnelles telles que la confiance, la sécurité, la qualité de service, alors que les composants sont immédiatement liés au déploiement sur la plate-forme et peuvent dépendre de propriétés matérielles telles que la mémoire ou l'usage processeur. Ceci nous permet également

de porter une vue orientée service sans nous soucier des composants les exportant ni des détails d'implémentation.

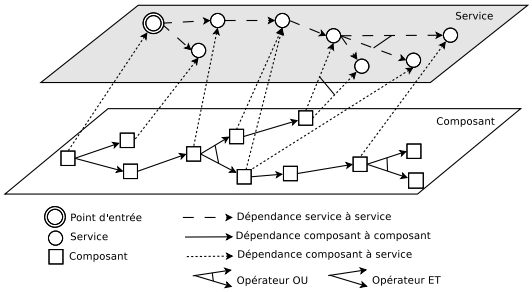

FIG. 3.11 – Illustration d'un exemple d'application orientée service composant

Dans la figure 3.12 nous illustrons la classification de Keller et al. [84] qui représentent les attributs des dépendances sur six axes orthogonaux. Nous nous inspirons de cette classification pour présenter chaque axe et l'étendre avec nos propres attributs. Les attributs entourés sont considérés par notre approche.

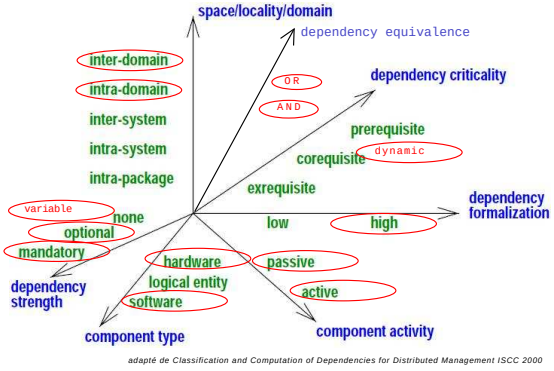

FIG. 3.12 – Extension de la classification des dépendances de [84]

Description de services AxSeL suppose qu'une description de service doit inclure les dépendances de celui-ci. Cette connaissance supplémentaire nous permet de structurer les applications sans nous soucier des implémentations sous-jacentes et de remplacer également un service par un autre. L'expressivité d'un composant est fonctionnelle mais très limitée, en effet, il n'est pas obligatoire au niveau composant de préciser les services fournis et surtout requis. Sans la précision des dépendances dans la description du service, notre connaissance se limiterait au niveau implémentation et déploiement.

Type de composant *(Component type)* Nous considérons principalement les dépendances entre entités logicielles : services et composants. Le contexte environnant peut aussi influencer le changement d'état des services et des composants et peut être considéré comme une dépendance. Le détail de cette dépendance fait l'objet de la section 3.4.

Type de dépendance *(Space/locality/domain)* Dans [84], les auteurs distinguent entre dépendances fonctionnelles et dépendances structurelles. Les premières sont précisées lors de la conception et du développement d'un composant, les secondes apparaissent au moment du déploiement et de l'installation. Les dépendances entre services et composants peuvent être de deux types, si elles sont précisées au développement elles sont fonctionnelles, sinon elles apparaissent dans la phase de déploiement et sont structurelles. Les dépendances entre services sont également structurelles et sont visibles à la phase d'exécution. Ainsi nous distinguons entre dépendances de déploiement et d'exécution. La satisfaction des dépendances de déploiement est requise par l'intergiciel lors du déploiement du composant.

Obligatoire ou optionnelle *(Dependency strength)* La dépendance obligatoire doit être satisfaite pour assurer le fonctionnement de l'application ou du service, alors qu'il n'est pas obligatoire de satisfaire une dépendance optionnelle. Celle-ci peut apporter une amélioration par une fonctionalité supplémentaire par exemple. Ces propriétés peuvent changer dans le temps, il est intéressant pour l'utilisateur de satisfaire une dépendance optionnelle annotée par un coefficient fort. En effet, AxSeL assigne à chaque dépendance une caractéristique non fonctionnelle qui s'incrémente avec la fréquence d'usage des éléments qui la composent. A travers cela, il est possible d'évaluer des statistiques d'usage d'un service donné et de prédire des comportements de chargement à l'avance.

Déterministe ou plurielle *(Dependency equivalence)* Nous ajoutons cet axe à la figure initiale. AxSeL fournit un modèle de dépendances basé sur la collecte de l'ensemble des fournisseurs de services susceptibles de satisfaire une dépendance. Ainsi, une dépendance AxSeL peut être équivalente avec une autre dépendance, dans le cas où un service est fourni avec deux versions différentes par exemple. Nous reposons sur une hypothèse simple d'équivalence de spécification. Deux services sont équivalents s'ils ont le même nom dans la description de services. D'autres travaux tels que [82] ont proposé des formalismes d'évaluation des équivalences entre services.

Formalisme de la dépendance *(Dependency formalization)* Plusieurs travaux se sont intéressés à la représentation des dépendances logicielles comme [60] où les auteurs exploitent les graphes conceptuels ou UML [45] qui est un langage formel. AxSeL repose sur un formalisme orienté objet et basé sur les graphes pour garder en mémoire la structure globale de dépendances d'une application donnée, le détail est donné en section 3.3.3. Ce formalisme offre la possibilité de réutilisation et d'évolution du modèle de dépendances, et permet également d'observer sur l'échelle temporelle les variations possibles d'une application en vue de faire des statistiques d'utilisation et de prédire les services favoris d'un utilisateur.

Politique de liaison *(Dependency criticality)* Il est possible de retrouver plusieurs types de politiques de liaison comme les dépendances statiques et les dépendances dynamiques. Les premières sont insatisfaites lorsque le fournisseur de services disparait. Les dépendances dynamiques se basent sur un mécanisme de recherche d'un fournisseur similaire lors d'une rupture de liaison. L'insatisfaction d'une dépendance influence l'état de l'instance des services et

composants lui sont relatifs. Lors d'une rupture d'une dépendance, AxSeL cherche d'éventuels fournisseurs de service similaires.

Instances de composants *(Component activity)* Les états d'une instance de composant dépendent étroitement du niveau de satisfaction de leurs dépendances. Elles sont présentés dans la figure 3.13. Le composant passe à l'état installé si toutes ses dépendances obligatoires sont satisfaites. Il devient ensuite actif lorsqu'il est lancé. Dès son activation l'instance de composant enregistre les services qu'elle désire fournir dans le contexte. Dans cet état il est possible de passer à l'état inactif ou désinstallé. La mise à jour et la désinstallation de l'instance sont possibles quand l'instance est active ou inactive. Quand une instance de composant est désinstallée les services qu'elle a auparavant enregistrés sont retirés du contexte.

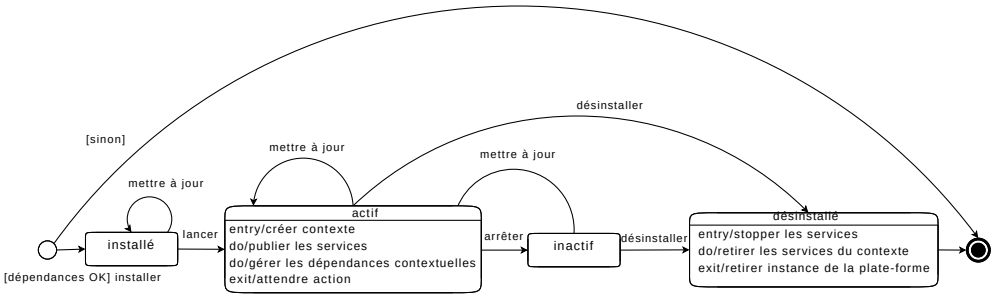

FIG. 3.13 – Diagramme des états transitions d'un composant

Conclusion Nous avons présenté séparément les concepts de services et de composants. Ensuite, nous les avons réuni dans un modèle unique de service composant qui associe l'aspect d'exécution avec celui du déploiement. Enfin, nous avons détaillé le paradigme de dépendance entre services et composants et l'avons exposé sous plusieurs aspects. Dans ce qui suit, nous introduisons la notion d'application et proposons notre représentation expressive et flexible.

3.3.3 Graphe bidimensionnel d'application orientée service composant

La considération d'un niveau applicatif orienté service composant permet la gestion de l'application dans sa totalité, et fournit également la possibilité de la contextualiser selon les préférences de l'utilisateur et les limites matérielles. Ceci constitue un apport majeur pour les environnements intelligents où la conscience du contexte et l'adaptation sont fortement requis. Malgré l'importance de ce concept, la notion d'application n'apparaît que partiellement dans certains travaux de la littérature, par exemple, le niveau applicatif n'existe pas dans les plates-formes OSGi [100] et SCA [7].

AxSeL considère des applications orientées services dans le but de les adapter aux contraintes de déploiement. Une application est définie comme étant l'ensemble des dépendances de services et de composants qui les implantent. La figure 3.14 illustre la composition d'une application orientée services composants.

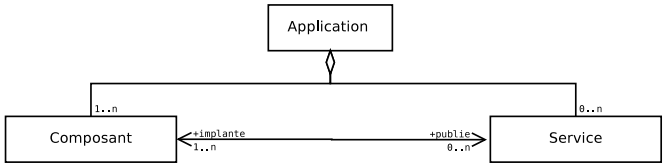

FIG. 3.14 – Représentation UML d'une application

La représentation de l'architecture d'une application s'apparente à plusieurs domaines dont les ADL (Architecture Description Languages) [55], les formalismes de représentation algébriques [111], logiques [31] et les approches orientées graphes [48]. Plusieurs travaux se basent sur la formalisation orientée graphe pour modéliser les applications logicielles dynamiques, Le Métayer et al. [91] et [90] proposent une grammaire orientée graphe et indépendante du contexte où les composants et les connecteurs sont assimilés respectivement aux nœuds et aux arcs du graphe. Les actions de reconfiguration de l'architecture modélisée sont réalisées par des règles de réécriture du graphe. Taentzer et al. [125] modélisent les graphes de réseaux distribués et assimilent chaque nœud du réseau à un graphe local. Les connecteurs sont assimilés à des arcs du graphe. Tout comme la précédente cette approche préconise des reconfigurations à travers des règles de réécriture du graphe. Wermelinger et al. [133] s'inspirent de la chimie pour modéliser les applications orientées composants, ainsi un composant est assimilé à une molécule et les connecteurs à des liens entre molécules. Les reconfigurations de l'architecture suivent des règles d'évolution.

L'utilisation des graphes offre un moyen simple pour représenter les éléments et les relations entre ceux-ci dans le domaine des applications orientées services composants. L'aspect dynamique est intégré par l'ajout d'opérations de manipulation du graphe pour avoir une structure extensible qu'il est possible de modifier par ajout ou retrait de services et de composants. D'autre part, les graphes supportent les heuristiques d'inférence et la modélisation des contraintes d'une manière générale. Il est également possible d'utiliser des outils graphiques afin de visualiser ces graphes d'une manière humainement compréhensible. Enfin, les graphes ramènent de l'expressivité à travers l'assignation de poids sur les nœuds et les arcs.

Pour la représentation des graphes de dépendances entre services et composants, nous proposons une approche basée sur une sémantique simple et un modèle orienté objet. Cette représentation doit satisfaire un certain nombre d'attributs. Il est important que la représentation adoptée soit compacte, efficace et sans ambiguïté voire même standardisée. Les coûts d'adaptation et de maintenance doivent être minimaux. En plus de ces attributs génériques il est important que notre représentation fournisse : une adéquation avec une définition standard dans le sens et la syntaxe supportée, la représentation des différents types de ressources (composants, services), l'expression de la relation d'import/export entre services et composants, l'expression de la cardinalité des éléments de la dépendances, l'équivalence des dépendances à travers l'opérateur logique OU, la représentation des concepts de contexte relatifs à chaque ressource ou dépendance, et enfin les opérations nécessaires pour permettre la flexibilité du graphe.

Acteurs et sens de la dépendance Les dépendances abordées décrivent des relations entre une entité de départ et un ensemble d'entités d'arrivée. Pour chaque dépendance il existe une entité qui dépend d'une autre, ceci exprime le sens de la dépendance qui va d'un élément de départ à un ou plusieurs éléments d'arrivée. Dans le domaine des architectures logicielles, le sens de la dépendance représente une information pertinente. Dans [84], les auteurs utilisent une sémantique particulière pour distinguer les éléments de départs de ceux de l'arrivée. En effet, les éléments de départ sont les dépendants et ceux d'arrivée sont les antécédents. Par analogie à cette sémantique, les dépendants et les antécédents considérés par AxSeL peuvent être des services ou des composants, nous les dénoterons invariablement par le terme "Ressource". La distinction entre services et composants se fait selon le niveau auquel appartient chaque ressource. Entre les différentes ressources il existe des liaisons d'import et d'export. La figure 3.15(a) montre une dépendance unidirectionnelle simple entre deux ressources. Les composants importent d'autres composants, ou d'autres services, qui eux aussi peuvent importer d'autres services. La figure 3.15(b) illustre les dépendances possibles. Une dépendance bidirectionnelle dénoterait d'une erreur de boucle fermée. Cela peut être possible si les spécifications des dépendances sont erronées.

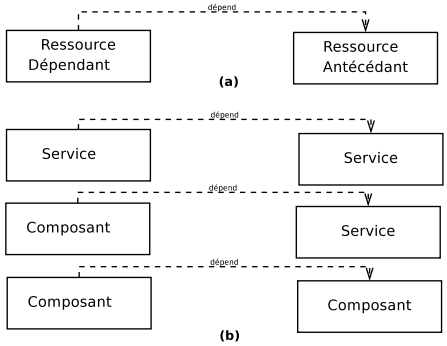

FIG. 3.15 – Dépendances entre ressources

Grammaire et terminologie utilisées Nous nous basons sur la sémantique des graphes pour mettre en évidence l'ensemble des dépendances entre les services et les composants d'une application. Pour ce, nous empruntons une sémantique simple qui adhère à nos besoins. Une application orientée services composants possède un ou plusieurs services ou composants initiaux et plusieurs services et composants interdépendants. La grammaire utilisée doit intégrer ces éléments en considérant les apports d'AxSeL. En effet, une application AxSeL est assimilée à un graphe de dépendances de services et de composants, comportant des niveaux distinctifs relatifs aux environnements de déploiement et d'exécution, un ou plusieurs points d'entrée et les opérateurs logiques ET et OU. Nous désignons les ressources services ou composants par *nœud*. La dépendance pouvant exister entre eux est un *arc*. Les environnements sont distingués par la notation *niveau*. Le sens de la dépendance est inclu dans la définition de l'arc qui a un nœud de départ et un nœud d'arrivée. La figure 3.16 (a) illustre la correspondance de la sémantique entre le monde réel et la représentation que nous proposons. Le sens de la dépendance ainsi que les opérateurs logiques sont représentés dans la figure 3.16 (b), le sens est indiqué au niveau

de l'arc par un nœud de départ (nœudDe) vers un nœud d'arrivée (nœudA). Pour l'opérateur logique OU désignant les dépendances équivalentes du graphe, nous faisons correspondre à un nœud dépendant de départ plusieurs nœuds antécédents d'arrivée, et enfin, l'opérateur ET est représenté par un arc ayant un seul nœud à arrivée.

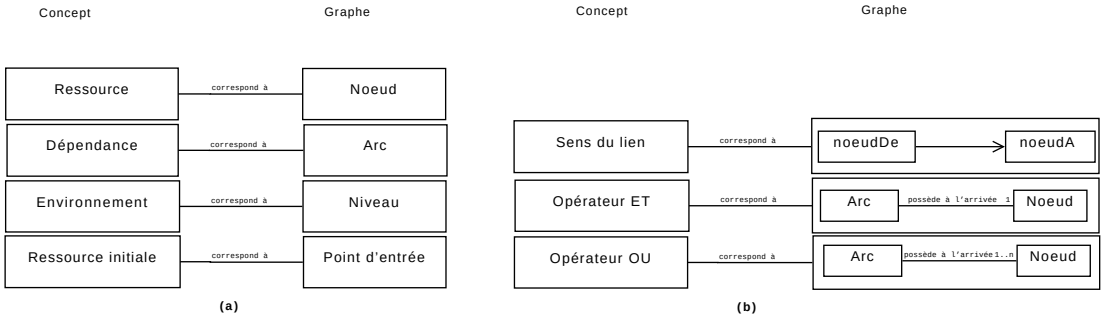

FIG. 3.16 – Sémantique de représentation

Graphe de dépendances AxSeL regroupe dans une même structure les services, les composants et les dépendances d'une application orientée services composants sous la forme d'un graphe de dépendances pondéré, orienté et bidimensionnel. Les dimensions correspondent aux niveaux de déploiement et d'exécution à considérer. La figure 3.17 présente le modèle du graphe d'application :

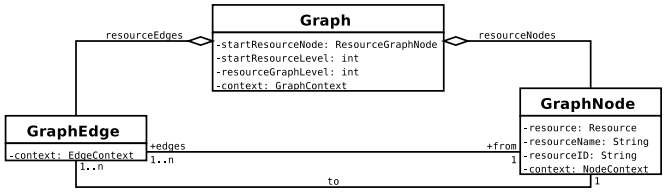

FIG. 3.17 – Modèle du graphe de dépendance

- *point d'entrée* : (startResourceNode) le déploiement d'une application est initiée par le déploiement de son point d'entrée qui entraîne celui de ses dépendances. Dans le cas des applications orientées services, le point d'entrée correspond au service ou composant qu'il va falloir exécuter ou déployer en premier. Nous distinguons le service ou le composant d'entrée des autres de la même application. Il est représenté dans le graphe par un nœud d'entrée appartenant au niveau correspondant à son type (service ou composant). AxSeL fournit un graphe extensible qui supporte l'existence de plusieurs points d'entrées.
- *niveau* : (GraphLevel) le graphe que nous proposons regroupe dans une même vue deux concepts différents : les services qui font partie des environnements d'exécution et les composants qui appartiennent plutôt aux environnements de déploiement. La représentation globale nous fournit une manière simple de gérer les deux niveaux simultanément. D'autre part, la distinction entre les niveaux permet d'assigner aux éléments appartenant à chacun les propriétés qui leurs sont intrinsèques. Dans le graphe, nous distinguons entre le niveau

d'exécution et celui du déploiement. Les nœuds correspondant aux services sont ajoutés au niveau d'exécution et ceux correspondants aux composants au niveau de déploiement. Nous avons conçu notre graphe pour supporter la définition et l'ajout d'autres niveaux selon les besoins d'utilisation. Un graphe peut posséder un ou plusieurs niveaux.

- *nœuds* : (GraphNode) les services et les composants constituant une application représentent des ressources logicielles (*Resource*). Ils possèdent de la même manière des propriétés fonctionnelles et non fonctionnelles qu'il est pertinent de représenter en vue de prendre une décision de déploiement. Ainsi, nous représentons communément les services et les composants par des nœuds du graphe. La distinction entre les nœuds services se fait par leur type et leur appartenance à un niveau. Chaque nœud est caractérisé par sa désignation, son identifiant et la ressource qu'il représente. L'expressivité du graphe permet également d'associer à chaque nœud service ou composant un contexte (NodeContext). La représentation UML du nœud du graphe est illustrée figure 3.18.

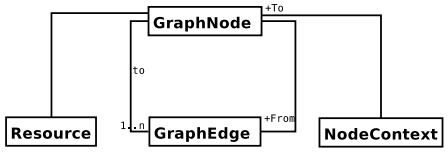

FIG. 3.18 – Représentation UML du nœud du graphe de dépendances

- *arcs* : (GraphEdge) les dépendances entre services et composants au sein d'une application correspondent aux relations d'import/export de fonctionalités. Ce concept important permet aux services et aux composants de bénéficier de fonctionalités qu'ils ne possèdent pas sans avoir à les implanter. Nous représentons dans le graphe ces dépendances par des arcs. Par analogie à une dépendance un arc lie deux nœuds. La figure 3.19 illustre la représentation UML d'un arc du graphe de dépendances. Nous distinguons entre dépendances d'exécution et dépendances de déploiement. Les services dépendent des composants qui les implantent, les composants peuvent importer d'autres composants. Enfin, les services importent d'autres services. Un arc est orienté et va d'un service à un autre, ou d'un composant à un autre ou d'un composant à un service. Les arcs possèdent des annotations spécifiques et un contexte. Un arc est caractérisé par le nœud de départ (GraphNodeFrom) et les nœuds de destination (GraphNodeTo).

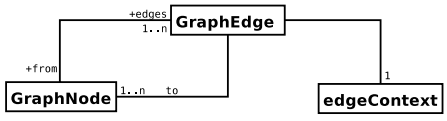

FIG. 3.19 – Représentation UML de l'arc du graphe de dépendances

Un arc peut avoir un seul nœud destination dans le cas où le nœud de départ ne possède pas de relations optionnelles avec d'autres nœuds (opérateur logique ET). Il a plusieurs nœuds destinations lorsque le nœud de départ importe optionnellement plusieurs autres nœuds (opérateur logique OU).

- *opérateurs logiques* : la sélection d'un service ou d'un composant à déployer est remise à la

charge du client qui choisit manuellement dans le dépôt ce qu'il souhaite. Dans un même dépôt, il est possible de trouver des représentations, des versions et des descriptions contextuelles différentes d'un même composant ou service offert par des fournisseurs différents. Grâce à l'intégration de cet aspect multi-fournisseur, AxSeL rompt le déterminisme des dépendances et offre un choix multiple entre plusieurs chemins possibles. Le choix des services et composants à déployer est réalisé automatiquement et par adéquation aux contraintes contextuelles. Pour représenter le déterminisme des choix de services et de composants et l'alternative entre une dépendance et une autre, nous étendons le graphe avec les opérateurs logiques ET et OU entre les arcs. Un arc portant l'opérateur ET renseigne sur la nécessité de charger les nœuds de cette dépendance. Une dépendance OU mentionne la multiplicité des choix.

- *opérations* : les descriptions des applications à déployer sont puisées dans des descripteurs de dépôts. Cependant, ces données peuvent changer lors de la modification des dépôts. La modification peut consister en l'apparition ou la disparition de nouveaux fournisseurs de services ou de composants ou la livraision de nouvelles mises à jour. Quelle que soit sa source, il est pertinent de répercuter dynamiquement les modifications observées à ce niveau afin de garder une vue actualisée d'une part et d'autre part, d'améliorer la prise de décision du chargement en intégrant de nouveaux paramètres. Pour offrir une représentation supportant la dynamique de l'environnement un ensemble d'opérations de manipulation du graphe est fourni. Celles-ci concernent les nœuds, les arcs et les niveaux, et permettent l'extensibilité du graphe en ajoutant et retirant un nœud, un arc ou un niveau. La dynamicité du graphe est assurée grâce à des fonctions qui modifient les données contextuelles au niveau des nœuds ou des arcs et des niveaux. L'extensibilité du graphe est réalisée à l'exécution. Lorsqu'un service ou un composant est ajouté dans la représentation de l'application ceci est pris en compte dynamiquement par l'ajout du nœud correspondant au niveau adéquat selon son type. Cette opération recherche dans le graphe les dépendances du nœud et les relie ensemble. De la même manière lors du retrait d'un service ou d'un composant, cela entraîne la suppression du graphe du nœud correspondant et de ses dépendances.

Adéquation avec les besoins de départ Nous avons proposé une structure de représentation globale pour les applications orientées services composants qui répond aux besoins d'expressivité et de flexibilité. Nous intégrons dans cette structure l'apport d'AxSeL tels que l'inclusion des environnements de déploiement relatif aux composants et d'exécution relatif aux services. D'autre part, les services ou composants peuvent être fournis par plusieurs fournisseurs avec des versions différentes ou des améliorations, cet aspect multi-fournisseur est pris en compte avec l'introduction de l'opérateur logique OU. Enfin, pour inclure des données non fonctionnelles pertinentes pour la prise de décision de chargement future, nous associons aux nœuds et aux arcs des contextes qui leurs sont propres. Ces contextes incluent des données relatives à la nature de la ressource et sont détaillés dans la section 3.4. Le tableau 3.1 récapitule les apports d'AxSeL par rapport aux objectifs de départ.

Objectifs	Solutions
Représentation de l'application	Usage de graphes (Graph)
Représentation des services et composants	Assignation de nœuds (GraphNode)
Représentation des dépendances	Représentation sous forme d'arcs (GraphEdge)
Aspect Multi-fournisseur	Introduction de l'opérateur logique OR
Aspect contextuel et expressivité	Association d'un contexte aux nœuds et aux arcs (Contexte)
Prise en compte des environnements	Distinction des niveaux déploiement et exécution (GraphLevel)
Flexibilité du graphe	Opérations de manipulation du graphe

TAB. 3.1 – Adéquation entre les besoins et les solutions proposées

Conclusion Cette section a abordé en détails les modèles de service, de composant et d'application et présenté le modèle de service composant proposé par AxSeL. Ensuite, nous avons proposé une représentation orientée graphe pour les applications orientées services composants et détaillé la sémantique de cette représentation ainsi que ses éléments. Dans la section suivante 3.4, nous abordons la modélisation du contexte, élément essentiel de l'architecture AxSeL et sur lequel repose l'adaptation des applications à déployer.

3.4 Une gestion dynamique du contexte

La conscience au contexte est un requis fondamental en intelligence ambiante car les paramètres contextuels peuvent influencer le comportement des services et des applications disséminés dans l'environnement. AxSeL fournit un déploiement contextuel en prenant en compte un contexte statique renseigné par les fournisseurs des services et des composants, et l'utilisateur, et un contexte dynamique sondé à partir du terminal. Il est essentiel de définir les sources contextuelles, la nature des valeurs recueillies, le contexte pertinent à considérer, les mécanismes de capture et enfin son modèle de représentation.

Dans cette section, nous présentons le modèle de contexte sur lequel AxSeL repose pour fournir le déploiement contextuel. D'abord, nous abordons les aspects généraux du service de gestion du contexte incluant la définition et l'architecture adoptées. Ensuite, nous détaillons les aspects de capture des données contextuelles en précisant les sources d'informations et les critères considérés, puis, les aspects statiques à travers la représentation et le stockage de contexte. Enfin, nous illustrons les aspects dynamiques déployés pour écouter le contexte et réagir.

Définition étendue Bien que la définition du contexte proposée par Dey et al. [61] soit communément admise, elle présente uniquement le caractère statique du contexte et l'aspect descriptif des informations pertinentes à l'utilisateur. Chen et al. [53] et principalement Coutaz et al. [56] présentent le contexte d'une manière plus dynamique en intégrant les interactions possibles avec un environnement en perpétuel changement. En nous référant à ces trois définitions nous étendons celle de Dey avec les éléments pertinents apportés par les deux autres. *Le contexte*

est toute information caractérisant une entité. Une entité est une personne, un endroit, un objet ou une application. Le contexte inclut également l'ensemble des états, paramètres et événements de l'environnement et de l'application. Il peut également être une partie d'un processus d'interaction avec un environnement constamment changeant et composé par des ressources mobiles, reconfigurables, distribuées. Le contexte est pertinent pour l'utilisateur, ou pour son interaction avec l'application.

Architecture en couches La séparation entre la capture du contexte et l'application fournit la possibilité d'extension de l'application et diminue le coût supplémentaire engendré par l'interaction directe avec les capteurs de bas niveau. Ainsi, l'utilisation d'une couche intergicielle intermédiaire est profitable car elle permet la gestion, l'interprétation et le traitement du contexte. La brique contextuelle d'AxSeL est conçue en couches (figure 3.20) et repose sur une infrastructure intergicielle centralisée qui masque la gestion des données de bas niveau collectées à partir des sondes. En comparaison avec les techniques de capture directe du contexte, celle-ci permet d'abord l'extensibilité de l'application sans modifier le code, ensuite la réutilisation des codes de capture du contexte à partir des ressources matérielles grâce à l'encapsulation [37].

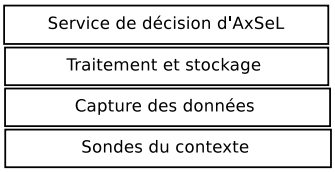

FIG. 3.20 – Architecture contextuelle en couche

3.4.1 Représentation et capture du contexte

Sources d'informations Le modèle de contexte doit satisfaire les besoins inhérents aux environnements ambiants telles la limite matérielle des terminaux, l'expressivité des applications et la représentation des données relatives à l'utilisateur. De par leur mobilité et leur petite taille, les terminaux mobiles sont contraints en ressources matérielles. La connaissance de l'état actuel d'usage en ressource au sein d'un dispositif est une information pertinente pour prendre une décision optimale de chargement. Ensuite, les applications considérées sont composées de services et de composants qui possèdent eux-mêmes des données caractéristiques contextuelles qu'il est pertinent d'intégrer dans le processus de décision. Enfin, l'adéquation aux préférences utilisateurs est un objectif important dans les environnements ambiants et intelligents afin de fournir des services personnalisés aux individus. Ces informations contextuelles peuvent influencer la décision du chargement. AxSeL considère comme sources d'informations contextuelles : le terminal, les services et les composants, et l'utilisateur. Ce schéma contextuel est ensuite projeté sur l'application orientée service composant afin d'obtenir une vue personnalisée et contextuelle de l'application à déployer (figure 3.21).

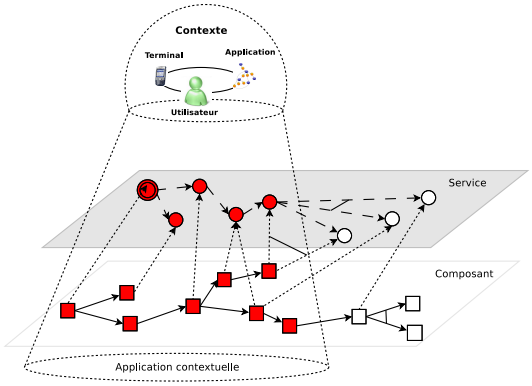

FIG. 3.21 – Vue contextuelle de l'application

Types de données Nous considérons deux types de données. D'abord, celles renseignées d'une manière statique dans les descripteurs de services et de composants ou dans le profil utilisateur. Ensuite, celles qui sont sondées à l'exécution à partir des ressources matérielles ou de la plate-forme d'exécution. Ces données peuvent provenir des terminaux, de l'utilisateur et des services et composants :

Terminal un terminal est une machine hôte destinée à l'usage de l'utilisateur. Dans la figure 3.22 nous décrivons le dispositif incluant la plate-forme logicielle et la configuration matérielle sous-jacente. Les ressources matérielles sont celles supportées par le dispositif mais aussi celles qui sont réellement mises à disposition de l'utilisateur ou de la plate-forme d'exécution de services. Par exemple, un dispositif peut avoir 256 Mo de RAM et ne laisser à la disposition de la plateforme d'exécution que 100 Mo. Les données matérielles sont déclinées sur plusieurs éléments notamment le processeur, le disque dur, la mémoire virtuelle et la batterie. Les ressources matérielles d'un terminal sont étroitement liées, par exemple l'exploitation de la mémoire engendre celle du processeur et puise dans les réserves de la batterie, inversement optimiser la mémoire permet d'optimiser les deux autres. Par conséquent, AxSeL considère l'optimisation de l'espace mémoire disponible sur le terminal. Toutefois, le modèle du contexte est extensible et peut inclure d'autres ressources.

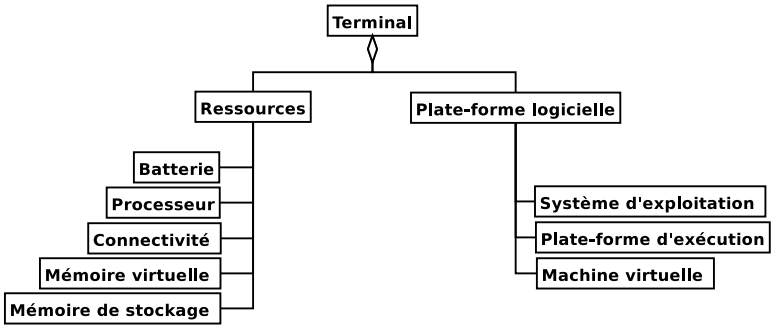

FIG. 3.22 – Modèle de description d'un terminal

Utilisateur le profil utilisateur peut inclure un grand nombre d'informations complexes. Par soucis de clarté et de simplicité nous considérons un contexte utilisateur statique qui renseigne ses préférences d'applications et de services. Ainsi, l'utilisateur peut renseigner une liste d'applications ou de services qu'il désire avoir sur son terminal en priorité. Cette liste de préférences nous permet de prendre en compte les désirs de l'utilisateur lors du chargement concomitant de plusieurs applications et services.

Services et composants AxSeL opère une adaptation qui est consciente des ressources matérielles fournies par le dispositif. La taille mémoire nécessaire à l'exécution d'un service ou d'un composant donnés peut être renseignée par le développeur lors de la livraison, sinon, si elle ne l'est pas nous supposons que la quantité mémoire estimée nécessaire est égale à la taille de stockage de l'unité de déploiement même (l'archive jar par exemple). Il aurait été possible de déduire par nous même l'usage prévu par un service, cependant, ce processus d'évaluation nous coûterait en ressources matérielles.

Modélisation du contexte Nous modélisons le contexte d'AxSeL en utilisant les diagrammes de classes UML. Un contexte est constitué des fournisseurs de données : le terminal (Device), le dépôt (Repository) hébergeant les services et les composants et l'utilisateur (User). L'adaptation réalisée par AxSeL peut se faire en concordance avec des changements observés au niveau de plusieurs sources. Ainsi, nous proposons un modèle hiérarchique [116] de classes où chaque source d'information représentée dans la figure 3.23 définit ses éléments et un ensemble de fonctions à travers une interface de manipulation exportant les fonctions d'affectation et de récupération des données.

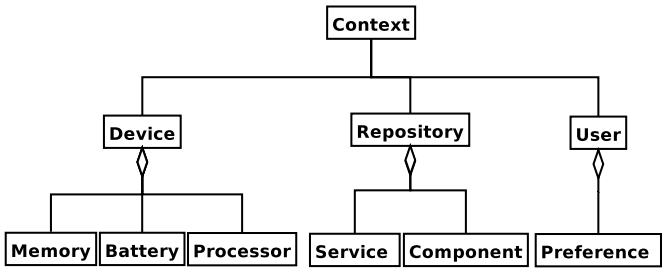

FIG. 3.23 – Modèle du contexte

Support de stockage A cause de la rareté des ressources matérielles, le stockage du contexte n'est pas nécessaire mais peut être utile. Les données stockées permettent de réaliser des historiques. Le choix du support de stockage et d'acquisition des données enregistrées dépend des besoins de l'utilisateur. AxSeL opte pour le format XML et utilise un outil optimisé de traitement de ce format de fichiers (kXML [16]) afin de minimiser l'exploitation des ressources matérielles. Pour enregistrer les données contextuelles relatives aux services et composants nous étendons le descripteur à partir duquel ils ont été installés initialement par des entrées contextuelles non fonctionnelles. Les données sont représentées dans ces descripteurs sous la forme de clé-valeur, les clés étant les attributs. Les lignes 5 et 6 du listing 3.1 mentionnent les entrées

contextuelles de taille mémoire et de priorité d'un service fourni par un composant. Le profil utilisateur ainsi que les données du terminal sont également enregistrés dans des descripteurs XML. Cependant à long terme, stocker des données peut entraîner la redondance et saturer les supports de stockage et ainsi l'utilisation des ressources du terminal. Préconiser un système de nettoyage de données est aussi coûteux en ressources, cependant, il est possible de réaliser des nettoyages de la mémoire des valeurs obsolètes, à des intervalles spécifiques sans décider des données à garder.

```
1 <resource id='1' presentationname='composant' symbolicname='composant'>
2    <capability name='service'>
3      <property name='version' value='1.0.0'/>
4      <property name='size' value='2500'/>
5      <property name='priority' value='2'/>
6    </capability>
7 </resource>
```

Listing 3.1 – Exemple de descripteur de composant

3.4.2 Gestion dynamique du contexte

Gestion homogène Afin d'éviter qu'AxSeL ne gère les données hétérogènes recueillies à partir de chaque source d'information nous utilisons un mécanisme qui masque les sondes. En effet, pour chaque source nous associons une enveloppe permettant à travers l'interaction avec des API spécifiques de récupérer les données à partir des sondes. L'enveloppe fournit des méthodes qui permettent de récupérer les données courantes. Pour les ressources matérielles, les API utilisées sont celles fournies par la plate-forme d'exécution, dans notre cas nous reposons sur une machine virtuelle permettant la récupération des valeurs de la mémoire. Les données relatives aux services et aux composants sont extraites à partir de l'API correspondante au dépôt et sont affectées au niveau des nœuds du graphe de dépendances. La figure 3.24 illustre les mécanismes mis en place pour assurer l'interaction entre les sondes et leurs représentants respectifs. Ceux-ci communiquent également avec le service de gestion du contexte.

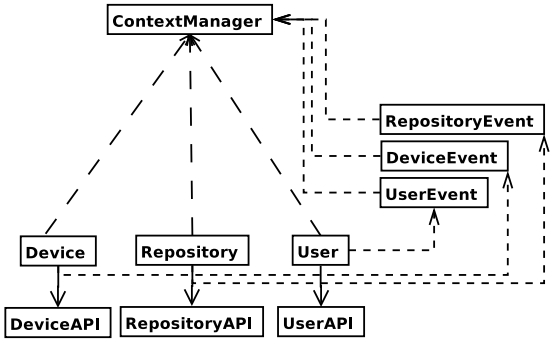

FIG. 3.24 – Mécanisme de gestion homogène du contexte

Ecouteurs de contexte Nous utilisons un système basé sur le mécanisme de notification événementielle. Ce dernier repose sur une source produisant un objet événement et un écouteur. Les différentes sources de données agissent comme des fournisseurs d'événements, elles ajoutent une liste d'écouteurs sur les ressources choisies au niveau du terminal, de l'utilisateur ou du dépôt, et produisent des événements. Lorsqu'un événement se produit l'écouteur en est notifié. AxSeL utilise ces mécanismes pour abonner le service de gestion du contexte à l'ensemble des événements provenant des différentes sources d'informations contextuelles qui l'intéressent, ainsi, chacune des sources *User*, *Repository*, et *Device* fournit des événements. A la réception de ces événements le service de gestion du contexte peut procéder à des traitements des données collectées et les passer au service de décision du chargement. Dans la figure 3.25, nous illustrons le mécanisme déployé pour écouter les changements au niveau de la mémoire. Chaque terminal ajoute des écouteurs sur ses sources d'informations. Lorsque le terminal est notifié du nouvel événement celui-ci est passé au service de gestion du contexte. Nous utilisons également le patron de conception *Adapter* qui permet à des classes hétérogènes de collaborer. L'interface de l'adapter est passé au client et ceux-ci continuent à l'utiliser comme s'il manipulaient l'interface d'origine. L'*adapter* traduit les appels passés à l'interface en des appels à l'interface originale.

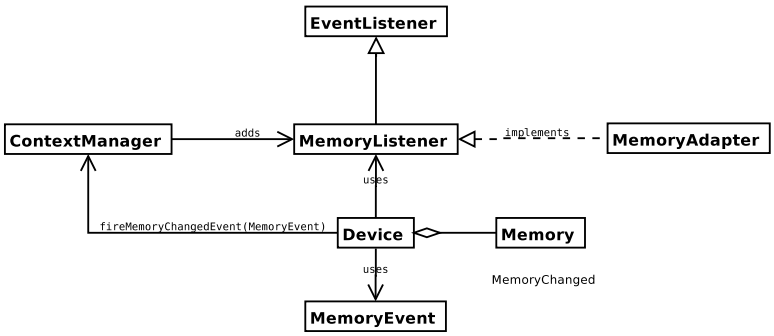

FIG. 3.25 – Exemple d'écouteur déployé sur la mémoire

Le service de gestion du contexte passe les événements générés dynamiquement au service de prise de décision qui effectue un raisonnement sur le contexte. A ce niveau, AxSeL opère une confrontation entre un contexte requis par les services, les composants et l'utilisateur avec un contexte fourni par le terminal. Cette confrontation est opérée par le service de décision du chargement qui prend un ensemble de contraintes contextuelles en entrée telles que la capacité mémoire maximale d'un dispositif et les services et composants auxquels l'utilisateur a donné une haute priorité et évalue au moment du chargement l'adéquation de chaque nœud aux contraintes. La prise en compte du contexte est réalisée initialement au moment du déploiement de l'application mais est aussi déclenchée au cours de l'exécution de celle-ci par le système d'écouteurs qui permet de capturer les événements pertinents. Les mécanismes de prise de décision et d'adaptation contextuelle sont détaillés dans la section 3.5.

Conclusion Dans cette section nous avons présenté les aspects contextuels proposés par AxSeL. D'abord, nous avons défini le contexte et esquissé l'architecture générale de capture et de représentation contextuelle. Ensuite, nous avons abordé les aspects de représentation et de

capture du contexte en précisant le terminal, l'utilisateur et les services et composants comme sources de données pertinentes. En adoptant une modélisation hiérarchique orienté objet nous facilitons l'extensibilité de notre modèle par ajouts de nouvelles sources. Enfin, sur la base de ce modèle AxSeL fournit une gestion dynamique et autonome du contexte grâce à un mécanisme de notification par événements. Dans la section suivante, nous présentons les mécanismes de contextualisation proposés par AxSeL.

3.5 Une contextualisation autonome dynamique

Un client voulant déployer une application sur un dispositif électronique accède à un dépôt de services composants et choisit l'application à déployer. Afin de réaliser cela il est nécessaire de résoudre toutes les dépendances de services et de composants de cette application et les rapatrier localement sur la plate-forme. Dans un cas classique le déploiement de l'ensemble des dépendances est systématique et aucune contextualisation servant à adapter ce processus au contexte de l'utilisateur ou de la plate-forme n'est adoptée.

AxSeL propose de réaliser un déploiement contextuel et autonome des applications à travers un ensemble d'heuristiques. Ces heuristiques reposent sur des mécanismes d'automatisation de la gestion des dépendances de l'application, de contextualisation du déploiement et d'adaptation à l'exécution. La figure 3.26 illustre le schéma de déploiement général d'AxSeL.

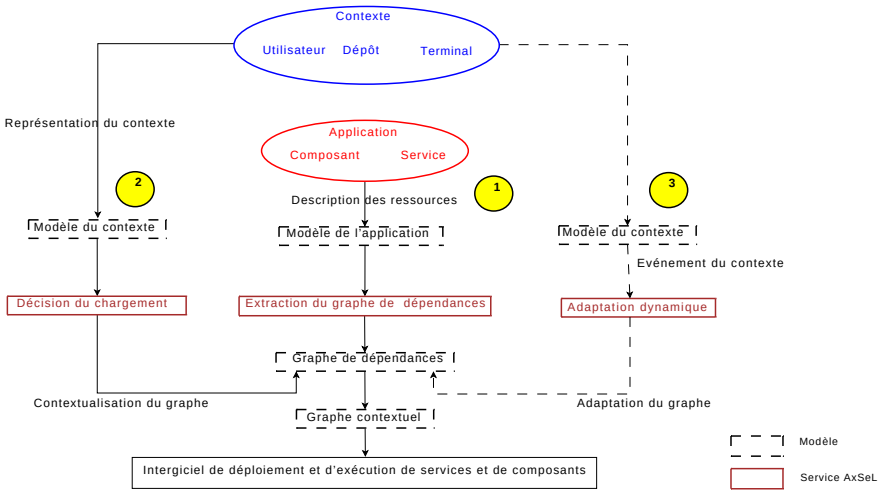

FIG. 3.26 – Schéma du déploiement d'AxSeL

L'extraction du graphe de dépendances trace l'ensemble des dépendances d'une application à partir du descripteur de ressources fourni et les représente sous la forme d'un graphe de dépendances orienté bidimensionnel. Ensuite, la décision du chargement évalue la faisabilité du déploiement initial en se basant sur les contraintes. Durant la phase d'exécution de l'application, les événements contextuels remontés à partir des sources de données utilisateur, terminal et dépôt de services composants, sont pris en compte par un processus d'adaptation contextuelle dynamique. Un graphe de dépendances contextuel et offrant une vue actualisée et dynamique est fourni à la fin des phases de contextualisation et d'adaptation.

Dans cette section nous présentons nos heuristiques de déploiement. Celles-ci se basent sur le modèle d'application orientée graphe et le modèle de contexte présentés dans la section précédente. D'abord, nous présentons l'étape d'extraction du graphe des dépendances en section 3.5.1, ensuite, la décision contextuelle en section 3.5.2. Enfin, nous montrons les mécanismes d'adaptation contextuelle à l'exécution en section 3.5.3. A la fin de cette section, nous effectuons une évaluation théorique des performances de nos heuristiques.

3.5.1 Extraction du graphe de dépendances

L'extraction des dépendances entre services et composants est une phase nécessaire avant le déploiement d'une application. Notre intergiciel fournit un support automatique pour cette phase et construit un graphe de dépendances recensant toutes les dépendances possibles des services et composants d'une application à déployer. La figure 3.27 illustre l'accès d'un client aux dépôts découverts. Chacun des périphériques de l'environnement peut constituer un dépôt de services et de composants. Plusieurs informations sont renseignées dans un dépôt, cela inclut notamment les dépendances des services et des composants ainsi que leurs propriétés non fonctionnelles. Afin de construire une vue globale et multi-fournisseur de l'application, AxSeL accède à un ou plusieurs descripteurs de dépôts et y puise les dépendances de l'application et ses propriétés non fonctionnelles. AxSeL considère également le cas où un service ou un composant est fourni par plusieurs fournisseurs avec des propriétés différentes. L'intégration de cet aspect multi-fournisseur fournit la possibilité de choisir entre plusieurs services et composants selon leurs propriétés non fonctionnelles. La construction du graphe de dépendances consiste à retrouver les sous-graphes de dépendances de chacun des services et des composants dans l'ensemble des dépôts découverts et à les intégrer progressivement dans le graphe global de dépendances.

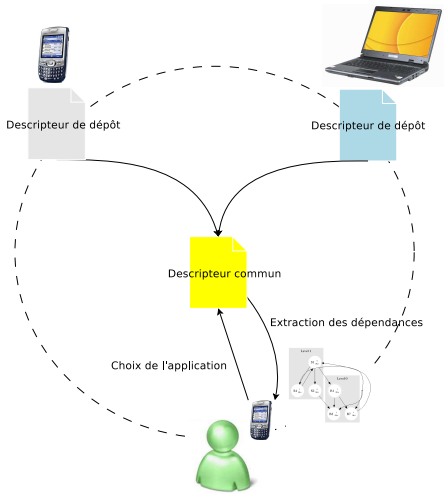

FIG. 3.27 – Extraction du graphe de dépendances

Dans ce qui suit, nous présentons le modèle de dépendance récursif entre service et composant. Ensuite, nous précisons la grammaire et la terminologie adoptées dans cette partie.

Enfin, nous détaillons les opérations de l'algorithme d'extraction des dépendances et l'illustrons à travers un exemple d'utilisation.

Un modèle de dépendance récursif Une application orientée service composant est constituée par un ensemble de services composants. Chaque service composant possède des exigences et fournit des capacités. Les exigences et les capacités représentent respectivement l'ensemble des services et des composants importés et exportés par celui-ci. La figure 3.28 illustre le schéma d'exigence et de capacité d'un service composant. Un composant importe zéro ou plusieurs composants et/ou services. De même, il exporte zéro ou plusieurs services et un ou plusieurs composants. Ce dernier doit au moins se fournir lui même pour être utilisé par des tiers. Dégager l'ensemble des dépendances d'une application en se basant sur ce schéma

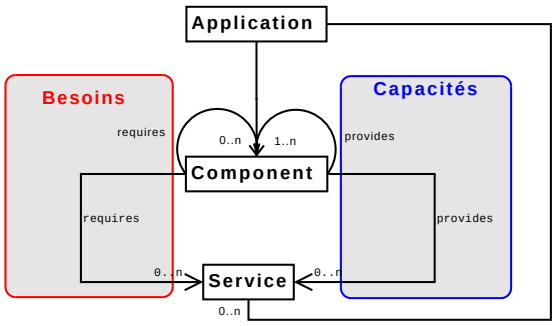

FIG. 3.28 – Schéma récursif du service composant

récursif revient à retrouver l'ensemble des besoins de chacun des composants constituant l'application et de les faire correspondre aux capacités des autres composants. Un composant a besoin d'un autre composant s'il importe des services ou des composants fournis par celui-ci. Résoudre cette dépendance revient à affecter les besoins d'un composant aux capacités des autres composants.

Grammaire et terminologie utilisées Les services et les composants sont hébergés dans un dépôt *repository* caractérisé par des propriétés non fonctionnelles. Les unités de déploiement considérées y sont hébergées sous la forme de ressources dénotées par le terme *resource*. Une ressource importe et exporte des services ou des composants et possède des propriétés non fonctionnelles. Les relations d'import et d'export sont définies par un ensemble de propriétés non fonctionnelles, et l'ensemble des services et composants importés ou exportés. Les services et les composants sont référencés par leurs noms respectifs et sont décrits par un ensemble de propriétés non fonctionnelles. Nous proposons la grammaire suivante pour représenter ces concepts :

- repository = {PNF, (resource)*}, PNF = Propriétés non fonctionnelles,
- resource = {PNF, (import)*, (export)*}
- import = {PNF, composanti→servicei |composanti→composantj}, i,j∈N
- export = {PNF, composanti→servicei}
- service = {PNF, NomService}

- component= {PNF, NomComposant}

Algorithme L'algorithme d'extraction du graphe de dépendances repose sur deux parties principales : d'abord, la construction du graphe et ensuite la recherche des dépendances. Dans ce qui suit nous présentons le principe d'extraction et de construction du graphe de dépendances de services composants et détaillons les fonctions sous-jacentes à ces mécanismes. Le modèle du graphe d'AxSeL est flexible et permet grâce aux fonctions fournies la construction du graphe de dépendance par l'ajout, la modification et la suppression des nœuds et des arcs. La construction du graphe est réalisée conjointement avec la fonction d'extraction de dépendances qui est responsable de retrouver l'ensemble des dépendances d'une application à partir d'un service ou d'un composant en entrée. L'extraction se base sur le modèle de service et de composant présenté dans la section 3.3.1.

Anticipation des dépendances de services AxSeL fait apparaître la relation de dépendance entre services qui n'est pas mentionnée dans les schémas traditionnels de descripteurs de déploiement des composants mais plutôt découverte au moment de l'exécution. Nous obtenons cette relation par extrapolation par rapport à la relation entre deux ressources dépendantes. En effet, nous supposons que si deux ressources sont dépendantes et que chacune d'elle exporte un ou plusieurs services il est probable qu'une relation de dépendances entre leurs services respectifs existe. En ajoutant l'expressivité à ce niveau nous obtenons une vue orientée services de l'application. Les dépendances entre services sont inclus dans le graphe de dépendances pour permettre ultérieurement leur chargement par anticipation.

Extraction du graphe de dépendances La fonction *extractGraph* permet d'abord de calculer le graphe de dépendances d'une ressource donnée. Ensuite de construire une structure de graphe pour représenter ses dépendances. Les dépendances représentent les relations d'import et d'export entre services et composants. Les ressources services ou composants retrouvées et satisfaisant les dépendances sont représentées par des nœuds. Ceux-ci sont créés et inclus dans le graphe selon leur type au niveau adéquat. Un service correspond à un nœud appartenant au niveau des services (*ServiceLayer*) et un composant correspond à un nœud du niveau des composants (*ComponentLayer*). Les dépendances entre services et composants sont représentées par des arcs du graphe de dépendances reliant le niveau des services à celui des composants. Les liens entre ressources de même type sont représentés par des arcs horizontaux appartenant au même niveau auquel appartiennent les ressources. A chaque étape de l'algorithme, l'inclusion entre le graphe courant et le graphe de dépendances de ses fils est enrichi d'une dépendance entre le point d'entrée du graphe courant et celui de ses fils. La construction du graphe de dépendances est réalisée d'une manière récursive pour chacune des ressources importées. Les nœuds visités sont marqués pour ne pas être revisités. L'algorithme d'extraction du graphe est présentée d'une manière simplifiée et par le pseudo code 1.

1. **Traitement des exports** : Récupérer la ressource d'entrée de l'application.
2. Créer le nœud correspondant à cette ressource et le graphe l'incluant.
3. Extraire la liste de ses exports si le nœud n'a pas été visité.

4. Parcourir la liste des exports et créer pour chaque service les nœuds correspondants et les ajouter aux graphes.
5. **Traitement des imports** : Récupérer les imports de la ressource.
6. Extraire les dépendances en ressources pour chacun des éléments de la liste des imports.
7. Parcourir la liste des dépendances en ressources et extraire récursivement le graphe qui correspond à chaque ressource.
8. Créer les nœuds correspondant aux services et les inclure dans un graphe temporaire.
9. Fusionner les graphes extraits à la première et à la deuxième phase dans un graphe global.

Algorithm 1 graph extractGraph(resource)

Require: *resource*
Ensure: *resourceGraph*

```
1: initNode ← new Node(resource)
2: graph ← new Graph(initNode)
3: if resource.isNotTagged then
4:    exports ← resources.getExports()
5:    if exports not null then
6:       for i = 0 to exports.length do
7:          if exports[i].getType() is Service then
8:             serviceNode ← new Node(exports[i])
9:             edge ← new Edge(serviceNode, initNode)
10:            graph.addEdgeBetweenLevel(edge)
11:         end if
12:      end for
13:   end if
14:   imports ← resources.getImports()
15:   initialize List resourceImports
16:   for j = 0 to imports.length do
17:      resourceImports ← extractDependency(imports[j])
18:      if resourceImports not null then
19:         while resourceImportshasElements do
20:            initialize List providerNodes
21:            providerNodes ← resourceImports.getElement()
22:            while provideNodes.hasElements do
23:               graphImports ← extractGraph(providerNodes.getElement())
24:               if providerNodes.getElement().getType() is Service then
25:                  serviceProviderNode ← new Node(providerNodes.getElement())
26:                  serviceGraph ← new Graph(serviceProviderNode)
27:                  serviceGraph.include(graphImports)
28:                  graphImports ← serviceGraph
29:               end if
30:            end while
31:            graph.include(graphImports)
32:         end while
33:      end if
34:   end for
35: end if
36: return graph
```

Extraction itérative des dépendances L'extraction des dépendances d'un service ou d'un composant revient à satisfaire ses besoins par les capacités des autres services et composants dans le dépôt. La fonction *extractDependency(Import)* automatise ce traitement par un parcours itératif du dépôt de services et de composants *repository*. Dans un dépôt de services composants, les composants sont représentés sous la forme de ressources et les services par les noms uniquement. Les lectures de données à partir des structures manipulées se font grâce à des fonctions *get()*. Ainsi, l'algorithme récupère la liste des ressources à travers la fonction *getResources()*. Cette dernière est parcourue d'une manière itérative dans le but de retrouver les fournisseurs de l'élément importé (*import*). Chaque *resource* a une liste d'imports et une liste exports. L'algorithme évalue chacun des exports des ressources du dépôt à la recherche d'une adéquation avec l'élément importé. L'adéquation de l'import avec les éventuelles ressources l'exportant est évaluée à travers la comparaison du nom de l'élément importé avec les noms de chacun des éléments exportés par les ressources du dépôt. Pour représenter le résultat de cette fonction l'algorithme manipule deux structures de données sous forme de liste : la première est la liste finale des fournisseurs *result*, et la seconde est la liste provisoire des fournisseurs trouvés à chaque itération. A la fin de chaque itération les ressources et les services fournisseurs sont ajoutés à la liste provisoire. Un test évaluant le type de l'export est réalisé afin de considérer les services, lorsque le fournisseur de l'import initial est un service il est ajouté dans la liste des fournisseurs provisoires. Celle-ci est ensuite additionnée à la liste finale. L'algorithme s'arrête d'itérer lorsque tout le dépôt est parcouru et que tous les fournisseurs sont retrouvés. Le détail du processus est présenté dans le pseudo code de l'algorithme 2.

Algorithm 2 resourceList extractDependency(import)

Require: *import*
Ensure: *resourceList*

```
1: exportValue ← import.getValue()
2: Resource[ ] resources ← repository.getResources()
3: initialize resourceList
4: for i = 0 to resources.length do
5:    exports ← resources[i].getExports()
6:    for j = 0 to exports.length do
7:       if exports[j].contains(exportValue) then
8:          providerList initialize
9:          providerList.add(resources[i])
10:         exportName ← import.getName()
11:         exportType ← import.getType()
12:         if exportType is Service then
13:            providerList.add(exportName)
14:         end if
15:         resourceList.add(providerList)
16:      end if
17:   end for
18: end for
19: return resourceList
```

L'algorithme d'extraction de dépendances se termine lorsque toutes les ressources du dépôt sont parcourus. Le nombre de boucle est borné par *resources.length* qui représente la taille du

dépôt de ressources, cette valeur entière est obligatoirement finie. Il en est de même pour le nombre d'exports parcourus au niveau de chaque ressource *exports.length* qui est également une valeur entière bornée. L'algorithme retourne une liste résultat *resourceList* même dans le cas où le dépôt est vide.

Construction incrémentale du graphe La construction du graphe correspond à la représentation des composants et des nœuds composant une application sous la forme d'une structure de graphe qui obéit au modèle présenté dans la section 3.3.3. La construction du graphe se fait d'abord par la création de la structure *Resourcegraph*. Ensuite, il est possible d'ajouter des nœuds *GraphNode* ou des arcs *GraphEdge*. Les arcs peuvent lier des nœuds d'un même niveau ou de niveaux différents (*graphLevel*). Lors du déroulement des étapes d'extraction des dépendances, des graphes temporaires sont créés et ajoutés au fur et à mesure à un graphe global représentant l'ensemble des dépendances d'une application. Le point d'entrée du graphe global peut être un service ou un composant. Dans ce qui suit nous présentons un des algorithmes faisant partie des étapes de construction du graphe de dépendances : *includeGraph()*, il prend en entrée un graphe et l'inclut dans le graphe général. Pour chaque niveau du graphe de dépendances l'ensemble des nœuds est extrait. L'algorithme itère ensuite sur les nœuds de cette liste jusqu'à sa fin et ajoute un à un les nœuds dans le graphe global au niveau correspondant. Il en est de même pour les arcs qui sont ajoutés un à un dans le graphe au niveau correspondant. Une fois l'ensemble des nœuds et des arcs ajoutés, l'algorithme ajoute les arcs reliant des nœuds de niveaux différents *edgesBetweenLevel*. Le sens de l'arc est indiqué selon son nœud d'origine (*From*), ainsi si ce dernier appartient à un niveau l'arc aura pour destination un nœud dans le niveau précédent ou suivant. Nous supposons qu'entre deux niveaux il existe une distance de un. Enfin, si les nœuds d'entrée des deux graphes sont au même niveau un arc reliant les deux est créé et ajouté dans le même niveau, sinon dans le cas où ils appartiennent à des niveaux différents un arc entre les niveaux les reliant est crée. Les étapes suivantes et le pseudo code de l'algorithme 3 donnent le détail du processus d'inclusion du graphe.

1. **Ajout des nœuds.** Récupérer les nœuds et les arcs de chaque niveau du graphe et les ajouter au graphe global.
2. **Connexion des nœuds.** Récupérer les arcs entre les niveaux du graphe et les ajouter au graphe global selon leurs origines.
3. **Inclusion du sous-graphe.** Relier les sommets des deux graphes.

Algorithm 3 includeGraph(graph)

Require: *graph*

```
1: for i = 0 to graph.levels do
2:    nodesList ← graph.nodesLevel(i)
3:    while nodesList.hasElements do
4:       graph.addNode(nodes.getElement())
5:    end while
6:    edgesList ← graph.edgesLevel(i)
7:    while edgesList.hasElements do
8:       graph.addEdge(edges.getElement())
9:    end while
10: end for
11: for j = 0 to graph.levels -1 do
12:    edgesBetweenLevelList ← graph.edgesBetweenLevel
13:    while edgesBetweenLevelList.hasElements do
14:       if edgesBetweenLevelList.getElement().getFrom() in graph.level(j) then
15:          graph.addEdgeBetweenLevels(j, j+1)
16:       else
17:          graph.addEdgeBetweenLevels(j+1, j)
18:       end if
19:    end while
20: end for
21: if entryGraphLevel = graph.getEntryGraphLevel then
22:    graph.addEdge((entryGraph, graph.getEntryGraph()), entryGraphLevel)
23: else
24:    graph.addEdgeBetweenLevels((entryGraph, graph.getEntryGraph()),
       graph.getEntryGraphLevel())
25: end if
26: return
```

Exemple d'illustration Afin d'illustrer les mécanismes présentés nous utilisons un exemple. La figure 3.29 trace les différentes itérations de l'algorithme d'extraction du graphe de dépendances.

L'algorithme prend en entrée deux descripteurs renseignant chacun les dépendances de chaque ressource d'une manière unitaire ce qu'elle exporte et ce qu'elle importe. Une première phase d'extraction des dépendances résout les dépendances de chacune des ressources, services et composants. Le résultat de cette phase est présenté dans le tableau 3.2.

1. *Itération 1 : traitement du point d'entrée S1.* Dans une première itération le nœud correspondant au service S1 est crée. Le graphe est initialisé avec ce point d'entrée. Le point d'entrée du graphe S1 est mentionné par une bordure double.
2. *Itération 2 : dépendances de S1.* Une deuxième itération traite les dépendances du composant S1. Celui-ci dépend du service S2 et du composant C1. Les nœuds correspondant à C1 et S2 sont créés et inclus dans le graphe initialisé dans l'étape précédente. Ensuite, ils sont reliés par des arcs au service S1.
3. *Itération 3 : traitement des dépendances de C1.* Le composant C1 dépend du composant C2. Le nœud correspondant à C2 est crée est inclu dans le graphe via un arc reliant C1 et C2.

Eléments de déparst	Dépendances
S1	C1, S2
C1	C2
S2	C2, S3
C2	C3
C3	C4
S3	C3,S4
S4	C4
C4	C5
S3	C3, S'4
S'4	C4
C4	C5

TAB. 3.2 – Tableau des dépendances entre services et composants

4. *Itération 4 : traitement des dépendances de S2.* Le service S2 dépend du service S3. Le nœud correspondant à S3 est crée est relié au service S2. L'ajout au graphe se fait au niveau des services.
5. *Itération 5 : traitement des dépendances de C2.* Le composant C2 dépend du composant C3, il est créé et ajouté au graphe au niveau des composant. Il est également relié par un arc au nœud du composant C2.
6. *Itération 6 : traitement des dépendances de C3.* Le composant C3 dépend du composant C4, il est créé et ajouté au graphe au niveau des composant. Il est également relié par un arc au nœud du composant C3.
7. *Itération 7 : traitement des dépendances de S3.* Le service S3 est décrit par deux descripteurs distincts. Dans le premier descripteur S3 dépend de S4 et dans le second de S'4. Les nœuds correspondants à ces services sont créés et reliés au service S3. L'opérateur logique OU est renseigné dans l'illustration par un trait entre les deux arcs S3-S4 et S3-S'4.
8. *Itération 8 : traitement des dépendances de S4 et S'4.* Les services S4 et S'4 sont décrits dans leur descripteur respectif comme dépendant du composant C4. Les arcs correspondants à ces liens sont créés et ajoutés au graphe.
9. *Itération 9 : traitement des dépendances de C4.* Le composant C4 dépend du composant C5. Le nœud correspondant à ce dernier est créé et relié au composant C4 par un arc de dépendance.

Lorsque toutes les dépendances sont parcourues et incluses dans le graphe, l'algorithme s'arrête d'itérer et fournit en sortie le graphe de dépendances global du service S1.

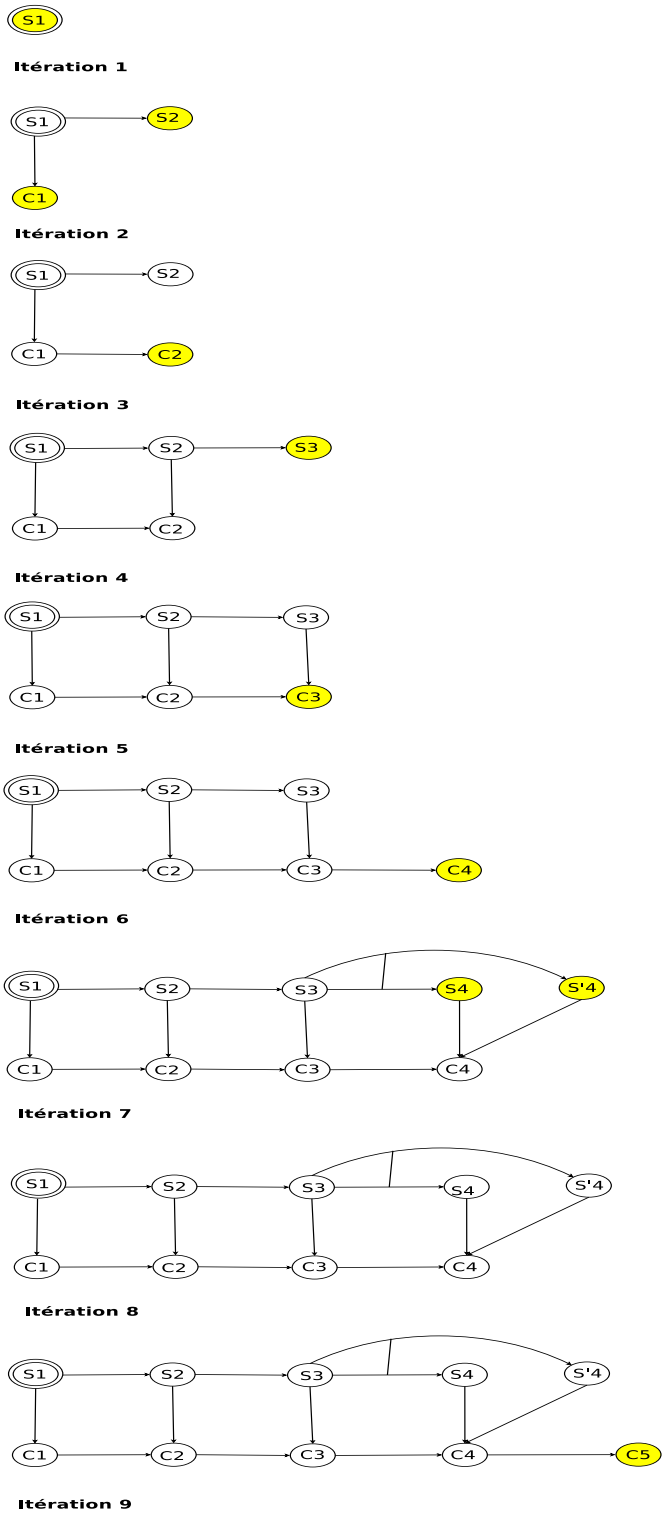

FIG. 3.29 – Illustration de la phase d'extraction du graphe de dépendances

3.5.2 Déploiement autonome et extensible

La prise de décision contextuelle proposée par AxSeL consiste en une adéquation à réaliser entre le contexte fourni par le dispositif mobile et le contexte requis par le graphe de dépendances à déployer et les préférences utilisateurs (figure 3.30). Afin de réaliser cela, nous proposons des heuristiques de déploiement multi-critère. AxSeL applique les stratégies définies sur le graphe de dépendances représentant l'application et extrait à l'étape précédente, et opère une adaptation structurelle de l'application. L'adaptation est réalisée grâce à des heuristiques de parcours et d'évaluation de la faisabilité du déploiement selon les contraintes prédéfinies. Le processus de décision repose sur le modèle de contexte dynamique et extensible présenté dans la section 3.4 et le modèle de graphe flexible présenté dans la section 3.3.3.

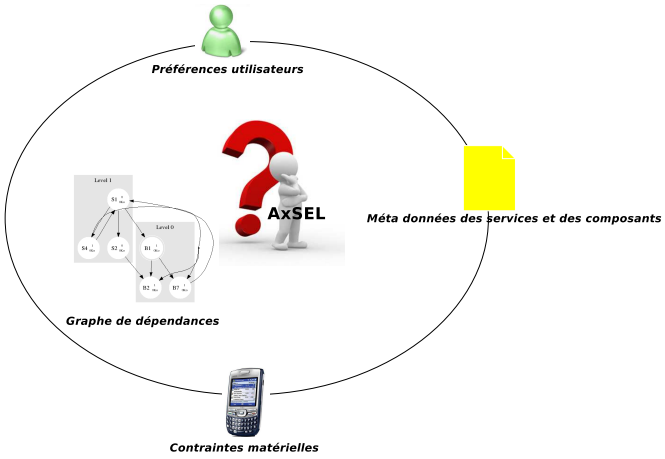

FIG. 3.30 – Enjeux de la décision du déploiement

Dans ce qui suit nous détaillons les stratégies de déploiement proposées ainsi que les mécanismes sous-jacents à la prise de décision. Le mécanisme de déploiement est composé par deux phases : la création de la stratégie de parcours du graphe à charger en définissant les contraintes à respecter et le graphe à parcourir, ensuite l'application de cette stratégie sur l'ensemble du graphe.

Stratégie de déploiement proactif et extensible AxSeL réalise une adaptation proactive en préconisant le comportement du processus de décision grâce aux stratégies. En effet, la stratégie de déploiement varie selon les contraintes spécifiées et choisies selon leur pertinence dans la politique adoptée. Dans un contexte contraint en ressources matérielles, des contraintes sur les dispositifs et leurs ressources sont importantes, alors que dans des contextes moins contraints, il est possible de considérer les préférences utilisateurs. Pour permettre à AxSeL d'avoir une politique de chargement extensible, nous adoptons le patron de conception *Strategy* [71] qui fournit un choix dans une panoplie d'algorithmes, les encapsule et les rend interchangeables. Grâce à cela, nous pouvons décliner les stratégies du déploiement selon une multitude de contraintes contextuelles.

Extensibilité des stratégies et des critères La figure 3.31 illustre la possibilité de choisir entre trois politiques distinctes selon la contrainte taille mémoire, la contrainte priorité de service ou les deux simultanément. Sur la base de ces contraintes, les nœuds du graphe à charger sont triés. Le chemin *CommonPathDecision* est un chemin générique à partir duquel il est possible de décliner d'autres chemins triés selon la taille *SizePathDecision* ou selon la priorité *PriorityPathDecision*. Le modèle supporte également l'aspect multi-critères et peut fournir une stratégie basée sur les deux critères en même temps *SizePriorityPathDecision*. Les critères de taille mémoire et de priorité d'un service sont cités à titre d'exemple. Ce modèle supporte l'extension par d'autres critères et d'autres stratégies choisis et implantés différemment. Les stratégies de déploiement peuvent être choisies par défaut ou spécifiées par l'utilisateur du périphérique.

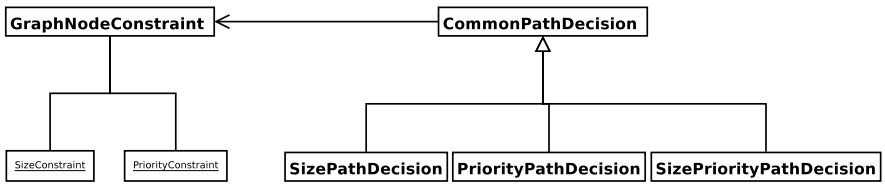

FIG. 3.31 – Stratégies de déploiement extensibles

Priorité des critères L'importance des critères non fonctionnels peut varier selon les contextes. AxSeL met en avant les critères prioritaires en triant les nœuds à traiter selon un ordre dicté par la contrainte contextuelle la plus pertinente. La stratégie de déploiement définie est appliquée sur l'ensemble des nœuds du graphe résultant de la phase d'extraction des dépendances. Pour effectuer cela nous proposons une heuristique qui a pour objectif de distinguer parmi les nœuds d'un graphe ceux obéissant aux contraintes de déploiement. Dans cette phase, AxSeL prend en entrée le graphe à charger et un ensemble de critères contextuels à partir desquels les contraintes de déploiement sont crées. Les nœuds du graphe sont disposés dans une liste de nœuds ensuite évalués selon leur respect des contraintes contextuelles. Les nœuds non chargés lors du premier passage sont mis dans une liste triée selon la pertinence du critère. Supposons par exemple que l'utilisateur ait défini ses préférences sur les services à charger, le tri des nœuds donnera une liste de services et de composants triée selon un ordre décroissant par rapport à la préférence de l'utilisateur. Ce premier tri permet de considérer en priorité les critères les plus pertinents dans la stratégie de déploiement.

Parcours du graphe optimisé L'heuristique distingue si le graphe a déjà été traité ou pas. S'il s'agit d'une première coloration du graphe le nœud d'entrée du graphe est ajouté à la liste des nœuds à traiter. Sinon, l'algorithme commence par traiter la liste des nœuds à traiter en premier *restartNodes*. Dans les deux cas, la coloration est propagée à l'ensemble des nœuds du graphe à travers l'appel à la fonction *recurseColourGraph()*. Nous distinguons entre les nœuds traités *taggedNodes*, les nœuds en attente de traitement *totagNodes* et enfin les nœuds à traiter en premier lors du prochain passage de l'algorithme *restartNodes*. Cette distinction nous évite de reparcourir des nœuds déjà visités et met en priorité ceux qui n'ont pas été couverts lors

des premiers passages. L'algorithme 4 illustre le traitement réalisé pour les nœuds d'un graphe donné. Le parcours est illustré d'une manière simplifiée par les étapes suivantes :

1. **Premier passage.** Traiter les nœuds en priorité *restartNodes*, les trier et lancer la coloration récursive.
2. **Seconds passages.** Ajouter le point d'entrée du graphe à la liste des nœuds à traiter et lancer la coloration récursive.

Algorithm 4 colourGraph()

Require: *graph*, *taggedNodes*, *totagNodes*, *restartNodes*

```
1: if not firstColouring and totagNodes is Empty then
2:    totagNodes.add(graph.getEntryNode())
3:    recurseColourGraph()
4: else
5:    while restartNodes.hasElements do
6:       totagNodes.remove(restartNodes.getElement())
7:    end while
8:    totagNodes ← restartNodes
9:    restartNodes ← sort(restartNodes)
10:   recurseColourGraph()
11: end if
```

Evaluation récursive de la faisabilité du déploiement L'évaluation de la faisabilité du déploiement est réalisée d'une manière récursive sur tous les nœuds du graphe de dépendances grâce à la fonction *recurseColourGraph()* qui propage l'application de la stratégie de chargement à l'ensemble des nœuds du graphe. Au niveau de chaque nœud les propriétés non fonctionnelles sont évaluées et comparées aux contraintes de déploiement. Pour réaliser cela nous proposons une fonction glouton qui prend une décision locale à chaque nœud du graphe de dépendances en vue d'obtenir un chemin global à coût optimal. Chaque élément de la liste des nœuds à traiter *totagNodes* est visité. L'installabilité de chaque nœud est évaluée à travers la fonction *isLoadable()* qui compare les critères contextuels des nœuds aux contraintes définies. Le résultat de cette fonction amène l'algorithme à appliquer un code de couleur pour différencier les nœuds installables des autres. Ainsi, lorsqu'un nœud obéit aux contraintes de chargement il est colorié en rouge et a un statut *Loadable*, sinon il est colorié en blanc et est *Unloadable*. Lorsqu'un nœud est blanc il est enlevé de la liste des nœuds à parcourir, ajouté à la liste des nœuds visités et à la liste des nœuds à traiter en premier lors des prochains passages. Grâce à cette technique nous le mettons en priorité par rapport aux nouveaux nœuds. Lorsqu'un nœud est rouge il est également enlevé de la liste des nœuds à traiter et ajouté à la liste des éléments parcourus. Ensuite, nous procédons au parcours de leurs dépendances. Ainsi, nous parcourons les nœuds destinations de chaque arc et l'ajoutons après tri à la liste des nœuds à traiter. Lorsqu'un arc portant l'opérateur logique OR est rencontré l'algorithme peut choisir aléatoirement son chemin parmi les options qui se présentent à lui ou opérer une évaluation sur les critères non fonctionnels des différents chemin. L'intégration de cet opérateur logique transforme le choix d'un nœud à charger en un problème NP-complet où une multitude de choix est offerte. L'algorithme 5

procède d'une manière récursive jusqu'au traitement de tous les nœuds de la liste *totagNodes*. Cette liste est obligatoirement finie car elle recense d'une manière unique les nœuds à traiter qui sont également puisés dans une liste finie.

1. **Premier passage.** La liste des nœuds visités est vide alors il s'agit d'un premier passage.
2. **Seconds passages.** Récupérer les éléments de la liste à traiter *totagNodes* et tester son installabilité. Si le nœud n'est pas installable alors le colorer en blanc, le retirer de la liste à traiter, et l'ajouter à liste prioritaire *restartNodes*. S'il est installable alors le colorer en rouge, le retirer de la liste à traiter et l'ajouter à la liste traitée *taggedNodes*. Ensuite, ajouter ses dépendances à la liste à traiter. S'il s'agit d'une dépendance OR trier les nœuds dans l'ordre pertinent avant de les ajouter à la liste à traiter.
3. lancer la coloration récursive.

Algorithm 5 recurseColourGraph()

Require: *graph*, *taggedNodes*, *totagNodes*, *restartNodes*, *totagEdgeNodes*

```
1: if taggedNodes is Empty then
2:   firstColouring ← true
3: else
4:   node ← totagNodes.getFirstElement()
5:   if node ∈ taggedNodes then
6:     taggedNodes.remove(node)
7:   else
8:     if !node.isLoadable then
9:       node.setColour(WHITE)
10:      totagNodes.remove(node)
11:      taggedNodes.add(node)
12:      restartNodes.add(node)
13:    else
14:      node.setColour(RED)
15:      totagNodes.remove(node)
16:      taggedNodes.add(node)
17:      edges ← node.getEdges()
18:      while edges.hasElements() do
19:        if edges.getElement().getTo() = 1 then
20:          totagNodes.add(edges.getElement().getTo())
21:        else
22:          totagEdgeNodes ← sort(totagNodes)
23:          while edges.getElement().getTo().hasElements() do
24:            totagEdgeNodes.add(edges.getElement().getTo().getElement())
25:          end while
26:          totagNodes.add(totagEdgeNodes.getFirstElement())
27:        end if
28:      end while
29:    end if
30:  end if
31:  recurseGraphColour()
32: end if
```

Exemple d'illustration Pour illustrer le processus de prise de décision réalisé par AxSeL, nous reprenons le graphe extrait dans la phase d'extraction des dépendances et y appliquons une politique orientée mémoire. Le tableau 3.3 liste les différents composants de l'application et leur taille mémoire respective. Les valeurs sont indiquées à titre d'exemple.

Noeud	Taille mémoire
S1	20Ko
S2	10Ko
S3	40Ko
S4	10Ko
S'4	5Ko
C1	10Ko
C2	10Ko
C3	40Ko
C4	10Ko
C5	10Ko

TAB. 3.3 – Tableau des valeurs d'usage mémoire

La figure 3.32 montre les différentes itérations du processus de décision. L'heuristique prend en entrée le graphe de dépendance global et charge le nombre maximal de services et de composants tant que la somme de leur taille mémoire ne dépasse pas **160**Ko.

1. *Itération 1 : installabilité du point d'entrée S1.* Il s'agit d'une première coloration du graphe de dépendances, l'ensemble des nœuds est à traiter. Le nœud S1 a une taille est de 10 Ko, il est chargé, ce qui laisse 170 Ko de libre sur le terminal. (taille disponible (170Ko) = taille maximale (180Ko) - taille du nœud S1 (10Ko)).
2. *Itération 2 et 3 : traitement des dépendances de S1.* L'heuristique propage l'évaluation de l'installabilité aux dépendances de S2 : C1 et S2. C1 a une taille de 10Ko et peut être chargé, il est donc coloré en rouge. La taille mémoire restante est de 160 Ko. De la même façon S2 et C1 sont coloriés en rouge et chargés localement.
3. *Itération 4 : traitement des dépendances de C1.* A cette étape les dépendances de C1 sont traités. C2 ayant une taille mémoire de 10Ko est coloré en rouge et chargé.
4. *Itération 5 : traitement des dépendances de S2.* S2 dépend de S3 qui a une taille mémoire de 40Ko et peut être chargé. Il est coloré et chargé.
5. *Itération 6 : traitement des dépendances de C2.* C2 dépend de C3, il a une taille mémoire de 40 Ko, il est chargé.
6. *Itération 7 : traitement des dépendances de S3.* S3 dépend en même temps de S4 (10Ko) et S'4 (5Ko), l'heuristique choisie le service ayant la plus petite taille mémoire. S'4 est alors coloré et chargé.
7. *Itération 8 : traitement des dépendances de S'4.* S'4 dépend de C4 (10Ko), la mémoire restante permet son chargement, il est alors coloré et chargé. A ce stade la mémoire disponible est de 5 Ko et est insuffisante pour accueillir le nœud C5 de taille 10Ko.

L'heuristique s'arrête d'itérer lorsque la limite de la mémoire est atteinte.

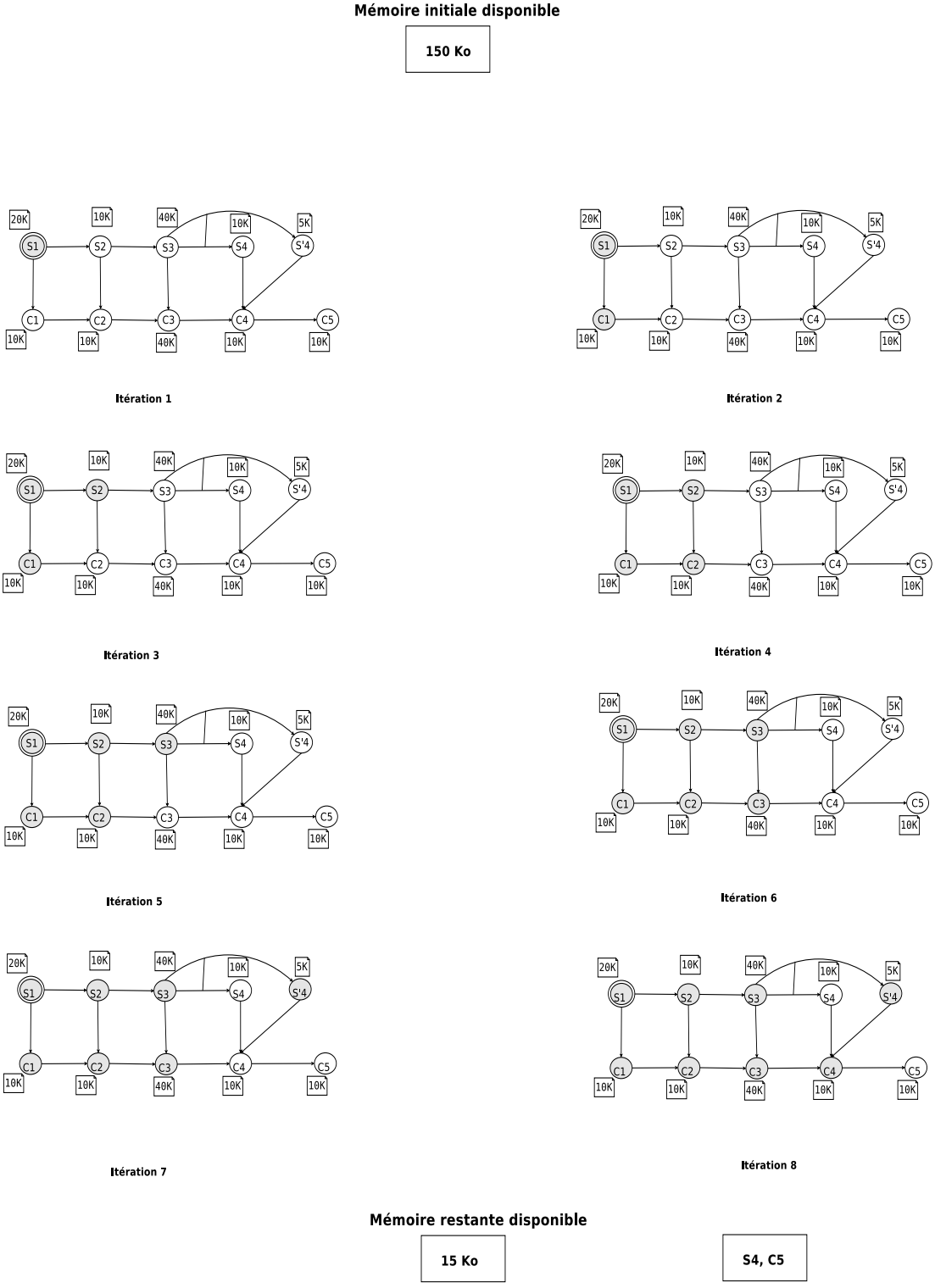

FIG. 3.32 – Illustration de la phase de décision du chargement

3.5.3 Déploiement adaptatif

AxSeL adapte le déploiement lors de l'exécution de l'application. Cette adaptation repose sur les mécanismes dynamiques présentés en section 3.4. Ainsi, lorsqu'un événement est capturé par les écouteurs du contexte, une action est entreprise selon la source et l'événement capturé. L'adaptation contextualise de nouveau le graphe de dépendances évalué à la phase de décision (figure 3.33).

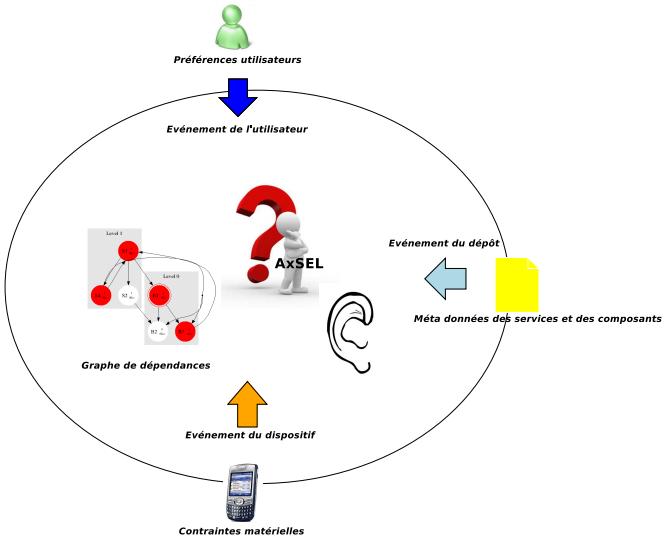

FIG. 3.33 – Adaptation contextuelle dynamique

Le tableau 3.4 récapitule les sources et les événements contextuels considérés par AxSeL.

Sources	Evénements
`Dépôt/Graphe`	Ajout/Suppression/Modification d'un nœud
`Dispositif`	Augmentation/Réduction de la mémoire
`Utilisateur`	Modification de ses préférences

TAB. 3.4 – Sources et événements contextuels considérés

Changements du dépôt Le dépôt de ressources peut subir des modifications dues par exemple à la livraison de nouvelles ressources, à la mise à jour des versions actuelles ou à la suppression de certaines ressources obsolètes. AxSeL prend en compte les changements observés au niveau du dépôt et y adapte le processus de déploiement.

- *Ajout d'un nœud* : lorsqu'un service est rajouté au dépôt et par conséquent au descripteur du dépôt, il est ajouté dynamiquement au graphe de dépendances de services tracé par AxSeL. Lorsque le nœud ajouté est un service il est placé au niveau service *ServiceLayer*, puis relié à ses dépendances. Il en est de même dans le cas où le nouveau nœud représente un composant, il est ajouté au niveau des composants *ComponentLayer*. Le nouveau nœud est ajouté à la liste des nœuds à colorier en premier *restartNodes*. La recoloration prend

en compte les contraintes contextuelles prédéfinies ainsi que les critères du nouveau nœud. L'algorithme 6 détaille ce processus.

Algorithm 6 adaptGraphNodeArrival(node)

Require: *node*, *graph*, *totagNodes*

```
1: if node.isService() then
2:    graph.add(node, serviceLayer)
3:    node.linkdependencies()
4: else if node.isComponent() then
5:    graph.add(node, componentLayer)
6:    node.linkdependencies()
7: end if
8: restartNodes.add(node)
9: totagNodes.add(node)
10: colourGraph()
```

- *Suppression d'un nœud* : correspond à la décoloration du nœud en question ainsi que ses dépendances immédiates pour sa non adéquation aux contraintes de chargement ou sa disparition du contexte. Pour ce, nous désignons le niveau auquel il appartient (service ou composant). Une fois positionné sur ce nœud l'algorithme lance une décoloration récursive des dépendances de ce nœud en prenant soin de ne pas décolorer ceux qui sont impliqués dans d'autres dépendances. La fonction *unColour(List)* vérifie qu'un nœud rouge n'a pas de prédécesseurs rouges, dans ce cas la coloration du nœud en question en blanc est possible. Un nœud traité n'est pas repris en compte. Lorsqu'un nœud est supprimé les ressources matérielles qu'il exploitait sont libérées. Le détail du mécanisme d'adaptation au retrait d'un nœud du graphe est présenté dans l'algorithme 7.

Algorithm 7 adaptNodeRemoval(node)

Require: *node*, *graph*

```
1: if node ∈ graph then
2:    node.Colour(WHITE)
3:    recurseUnColour(node.getDependency())
4: else if node ∈ graph then
5:    node not found
6: end if
```

- *Modification des propriétés d'un nœud* : les données contextuelles relatives aux nœuds peuvent subir des changements en raison d'une nouvelle livraison des services ou autre. Lorsqu'un nœud est modifié, dans le cas où il existe dans le graphe, nous modifions ses critères contextuels et vérifions sa couleur, si celle-ci est blanche, il est ajouté à la liste à colorier en premier et une recoloration du graphe est lancée. Sinon, s'il est rouge mais qu'il n'est plus installable nous lui attribuons la couleur blanche et décolorions récursivement ses dépendances. Le détail de ce traitement est présenté dans l'algorithme 8.

La fonction uncolor(List) présentée dans l'algorithme 9 vérifie qu'un nœud rouge n'a pas de prédécesseurs rouges. Dans ce cas, la coloration du nœud en question en blanc est possible. La taille du nœud est déduite de la mémoire disponible sizeNode, et la fonction propage ce

Algorithm 8 adaptNodeChange(node)

Require: *node*, *graph*, *totagNodes*

```
1: if node ∈ graph and node.Colour = WHITE then
2:    Change node properties by new ones
3:    restartNodes.add(node)
4:    colourGraph()
5: else if node ∈ graph and node.Colour = RED then
6:    Change nodei properties by new ones
7:    if !node.isLoadable() then
8:       nodei.Colour(WHITE)
9:       recurseUnColour(children(nodei))
10:      colourGraph()
11:   else if node.isLoadable() then
12:      Change nodei properties by new ones
13:   end if
14: end if
```

traitement pour les fils du nœud traité. Un nœud traité n'est pas repris en compte.

Algorithm 9 uncolor(List l)

Require: *list*

```
1: ∀ node ∈ l
2: if node.isColored(RED) and ∄ nodei \ father(node) and nodei.isColored(RED) then
3:    node.changeColor(WHITE)
4:    sizeNode ← sizeNode - node.getSize()
5:    uncolor(l - node+ children(node))
6: end if
```

Changement du dispositif et des préférences utilisateurs Les changements observés au niveau du dispositif ou des préférences utilisateurs entraînent des mécanismes similaires au sein d'AxSeL. En effet, dans les deux cas, cela entraîne une nouvelle prise de décision basée sur des paramètres contextuels mis à jour tels qu'une nouvelle quantité mémoire ou une liste de préférences modifiée.

- *l'augmentation ou la diminution de la mémoire dans le dispositif* : les événements provenant de la mémoire sont notifiés à AxSeL grâce aux écouteurs mis en place. Lors de la réception d'un événement mémoire AxSeL relance une recoloration du graphe de l'application en prenant en compte les nouvelles valeurs enregistrées. Dans le cas de la libération de la mémoire la recoloration entraîne l'installation de nouveaux services sur le dispositif. La diminution de la mémoire disponible peut engendrer la désinstallation de certains services.
- *le changement des préférences de l'utilisateur* influence le déploiement. Lorsque l'utilisateur change sa liste de services préférés, AxSeL prend en compte ces modifications et les intègre dans la stratégie de déploiement. Cela entraîne une nouvelle prise de décision et un nouveau parcours du graphe amenant au chargement ou déchargement éventuels de services.

Example d'illustration Nous reprenons le graphe de l'example précédent et illustrons les cas d'ajout et de suppression d'un nœud du graphe. Le tableau 3.5 liste les dépendances des nœuds.

Eléments de départs	Dépendances
S1	C1, S2
C1	C2
S2	C2, S3
C2	C3
C3	C4
S3	C3,S4—S'4
S4	C4
S'4	C4
C4	C5, C6

TAB. 3.5 – Tableau des dépendances entre services et composants

La figure 3.34 (A) présente le cas de l'ajout d'un nouveau composant *C6* de taille 5Ko. Dans ce cas l'algorithme d'adaptation suit les étapes suivantes :

1. *Itération 1 : ajout de C6 au graphe.* Le nœud représentant C6 est ajouté au graphe dans le niveau des composants. L'algorithme le lie à sa dépendance C4 dans le graphe et le met dans la liste des nœuds à traiter.
2. *Itération 2 : évaluation de l'installabilité de C6.* L'heuristique évalue la possibilité de charger localement C6. Ce dernier a besoin de 5 Ko de mémoire libre il est alors possible de le charger.

La nouvelle liste de nœuds non coloriés est composée de S4, et C5.

La figure 3.34 (B) illustre le cas du retrait du nœud S'4 du graphe. L'algorithme de décoloration procède comme suit :

1. *Itération 1 : décoloration de S'4.* S'4 est décoloré et déchargé de la plate-forme ce qui libère 5Ko de mémoire.
2. *Itération 2 : ré-évaluation de la décision.* L'heuristique réévalue l'installabilité des nœuds dans la liste non colorée. Avec 15Ko de mémoire disponible il est possible de charger le service S4 mais pas le composant C5.

La nouvelle liste de nœuds non coloriés est composée de C5 et S'4.

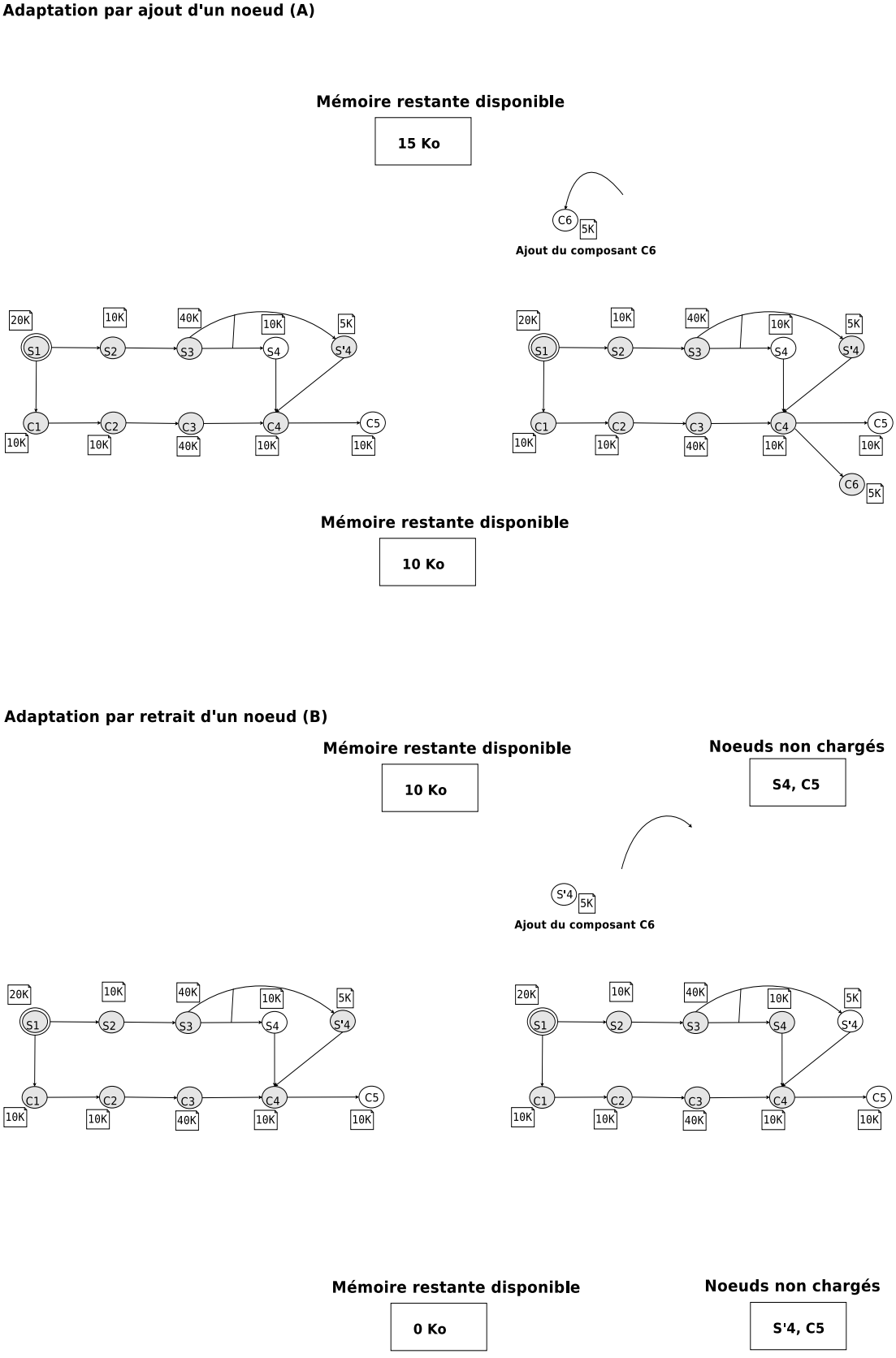

FIG. 3.34 – Illustration de la phase d'adaptation à l'ajout et au retrait d'un nœud

3.5.4 Evaluation théorique des heuristiques

La complexité d'un algorithme peut être évaluée en fonction du temps et en fonction de l'espace. La complexité temporelle évalue la vitesse d'exécution d'un algorithme en fonction d'un ou de plusieurs paramètres dépendant des données en entrée. Elle est proportionnelle au nombre d'opérations effectuées. La complexité spatiale évalue l'occupation mémoire en fonction de certains paramètres et représente la taille mémoire utilisée pour stocker les structures de données à l'exécution. Considérer ces paramètres nous permet d'évaluer la complexité des algorithmes indépendamment du contexte d'exécution, car souvent les performances d'un algorithme dépendent étroitement des performances matérielles de la machine exécutant l'algorithme, plus précisément de la vitesse de son processeur et du temps d'accès à la mémoire, de la taille du code et aussi de la complexité en temps des algorithmes "abstraits" sous-jacents. Dans le but de fournir une estimation indépendante du contexte, la complexité en temps de nos algorithmes est évaluée en fonction du nombre d'opérations élémentaires effectuées lors de son déroulement car nous supposons que le temps est proportionnel aux nombres d'opérations exécutées. Ainsi, nous spécifions les opérations que nous estimons coûteuses et évaluons leurs fréquences en fonction d'un ou de plusieurs paramètres dépendant des données en entrée. Pour évaluer la complexité de nos heuristiques nous reposons sur les hypothèses suivantes.

Hypothèses La complexité d'un algorithme est représentée par le temps de calcul d'une procédure. Pour évaluer la complexité d'un algorithme nous considérons les règles suivantes :

- pour une procédure p et des données de taille n, le temps de calcul de p(n) est le nombre d'instructions élémentaires exécutées pendant ce calcul.
- le temps d'exécution d'une affectation ou d'une lecture ou d'une écriture est considéré constant *T(test) = c*.
- le temps d'exécution d'une branche conditionnelle est la somme du temps d'évaluation de la condition avec le maximum des temps d'exécution des instructions : T(if C then I1 else I2) = T(C) + max(T(I1), T(I2)), C condition et Ii Instruction i.
- le temps d'exécution d'une séquence d'instructions est la somme des temps d'exécution des instructions la composant, $\sum_{i=0}^{i=n} T(E)$, E : élément de la séquence. Cela revient au temps d'exécution de l'instruction de plus forte complexité.
- le temps d'exécution d'une boucle T(boucle) = $\sum T(Ii)$, Ième itération de la boucle
- la complexité est évaluée dans le pire des cas : T_{max} (n) correspond au temps maximum pris par l'algorithme pour un problème de taille n.

Ordre de grandeur Nous évaluons la croissance de la fonction de complexité en fonction de la taille des données. Cela nous permet d'étudier le comportement de la complexité de l'algorithme lorsque la taille de données n tend vers l'infini. Nous représentons la complexité en ordre de O(n) où n est la taille des données. Nous considérons les règles suivantes :

- soit f et g deux fonctions de N dans R*+, f= O(g) ssi c $\in R^{*+}$, $\exists$ n_0 tq $\forall$ n $>n_0$, f(n) $\leq$ c g(n) Nous obtenons l'égalité T(n) = O(f(n)) lorsqu'il existe une constante c telle que, lorsque n tend vers l'infini le nombre d'opérations effectuées est bornée par cf(n)
- les constantes ne sont pas importantes.

- les termes d'ordre inférieur sont négligeables : Si T(n) = $a_k n^k$ + .. + a_1n + a_0 avec $a_k > 0$ alors T(n) est en (n^k)
- si T1(n) = O(f1) et T2(n) = O(f2) alors T1(n) + T2(n) = O(f1+f2) et T1(n) * T2(n) = O(f1*f2)
- si f(n) = O(g) et g(n) = O(h) alors f(n) = O(h)

Calcul Nous appliquons le principe précédemment présenté pour évaluer l'ordre de grandeur de la complexité de nos algorithmes. Nous détaillons le calcul des performances de l'heuristique générale d'extraction du graphe de dépendances 1, et donnons les complexités des autres heuristiques. L'extraction du graphe de dépendances fait appel aux algorithmes 2 et 3. Pour évaluer la complexité générale, nous calculons d'abord les complexités de chacune des deux fonctions séparément, ensuite nous l'intégrons dans le calcul global. Le détail du calcul est fourni uniquement pour l'opération d'extraction des dépendances.

- **Extraction des dépendances**. L'opération la plus coûteuse dans la fonction *extractResourceImport(import)* est la comparaison. Celle-ci est réalisée pour évaluer l'adéquation des éléments en *export* avec l'élément en *import* initial, autant de fois que possible jusqu'à obtention du résultat. Le nombre d'itérations maximal dépend de deux paramètres en entrée : la taille du dépôt *resources.length* autrement dit le nombre des ressources et le nombre d'exports au niveau de chaque ressource *exports.length*. Le détail du calcul de la complexité de l'algorithme 2 est présenté dans le tableau 3.6 et le listing 3.6.
- **Construction du graphe**. La construction du graphe repose la création du graphe, des nœuds et des arcs, mais aussi sur la fonction d'inclusion des graphes temporaires extraits dans un graphe global. De la même manière que précédemment nous avons évalué la complexité de la fonction d'ajout d'un arc au graphe, ainsi, lorsque l'arc ajouté relie deux nœuds du même niveau celle-ci est de O(n), et lorsque l'arc relie des nœuds appartenant à deux niveaux différents elle est de O(n^2). La complexité de l'inclusion des graphes présentée dans l'algorithme 3 dépend fortement du nombre de nœuds dans le graphe initial et dans le graphe final.

```
1  Pour évaluer la complexité de l'algorithme nous sommons les complexités de
       chacune des lignes du code :
2
3  Détail du calcul de complexité de extractDependency()
4
5  Soient T la complexité, n le taille des données, ci constante i, a—b :
       lignes du code du numéro a jusqu'à b.
6  T(n) = c1 + c2+ c3 + n*T(4--18)
7  T(n) = c1 + c2+ c3 + n*(c4+ n*T(6--17))
8  T(n) = c1 + c2+ c3 + n(c4 + n*(T(7--16) + c5+c6+ c7+ c8 + T(12--14)
9  T(n) = c1 + c2+ c3+ n*(c4+ n*(c5+ c6+ c7+ c8 + c9+ c10))
10
11 Après simplification par les constantes et les termes d'ordre inférieurs
       qui sont négligeables
12 T(n) = n*n = n^2 = O(n^2)
13
14 En appliquant la même procédure nous obtenons :
15
```

Ligne	**Opération**	**coût**
1	lecture	c1
2	initialisation	c2
3	lecture	c3
4	boucle	n*T(4–18)
5	lecture	c4
6	boucle2	n*T(6–17)
7	condition	T(7–16)
8	initialisation	c5
9	initialisation	c6
10	lecture	c7
11	lecture	c8
12	condition	T(12–14)
13	initialisation	c9
15	initialisation	c10
16, 17, 18, 19	fin	-

TAB. 3.6 – Détail du calcul de complexité théorique de extractDependency(import) 2

```
16 Complexité de l'extraction des dépendances : T(n) = O(n^2)
17 Complexité de la construction du graphe : T(n) = O(n^2)
18 Complexité de l'extraction du graphe de dépendances global : T(n) = O(n^5)
```

Listing 3.2 – Calcul de la complexité de l'algorithme *extractGraph()*

En appliquant le même principe pour les algorithmes présentés nous calculons leur complexité. Celles-ci sont présentées dans le tableau 3.7. Les algorithmes polynomiaux sont stables et efficaces. Nous avons également veillé à ce que les différentes heuristiques proposées se terminent et fournissent le résultat escompté. Dans les cas de parcours du graphe et de prise de décision, nous utilisons des structures spécifiques pour optimiser le parcours du graphe de dépendances. Ainsi, nous marquons nos passages et les nœuds déjà visités afin d'éviter leur reparcours.

Heuristiques	**Complexité**
Extraction du graphe de dépendances	O(n^5)
Décision du chargement	O(n^2)
Adaptation contextuelle	O(n^2) (Ajout de nœud) O(n^3) (retrait de nœud)

TAB. 3.7 – Complexité des heuristiques

Outre les critères de performances mesurés par la complexité, la qualité des heuristiques peut être évaluée à travers des critères qualitatifs tels que la simplicité de lecture des algorithmes, la facilité de maintenance, la fiabilité et la robustesse de ceux-ci. Nous proposons des heuristiques fiables qui reposent sur des standards de génie logiciel orientés objet et reconnus et dont l'efficacité est approuvée. Enfin, Les mécanismes décrits sont compréhensibles et faciles à maintenir.

3.6 Synthèse

Dans ce chapitre nous avons présenté notre contribution dans le domaine du déploiement d'application orientée services. Nous avons mis en évidence les problématiques à résoudre et proposé notre approche AxSeL pour le déploiement autonome et contextuel d'application orientée services composants. D'abord nous avons détaillé les services cœur d'AxSeL et les interactions prenant place entre elles. La figure 3.35 illustre les éléments principaux de notre architecture :

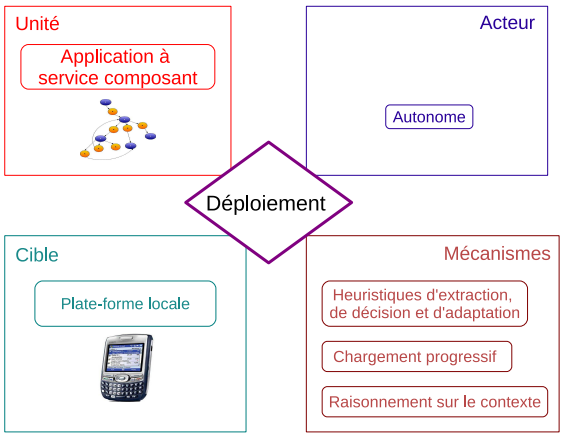

FIG. 3.35 – Synthèse du déploiement adaptatif

- *Un modèle d'application orientée services composants* basé sur une représentation flexible et dynamique sous la forme d'un graphe de dépendances. Le modèle de graphe proposé est expressif et contextuel, il intègre les connaissances non fonctionnelles des services et des composants dans ses nœuds. Il supporte également l'aspect multi-fournisseur inhérents aux approches à services par l'intégration de l'opérateur logique OU. Il est également flexible et permet la dynamique et l'extensibilité des éléments du graphe.
- *Un modèle de contexte dynamique* simple et basé sur la collecte à l'exécution des données contextuelles à partir du dispositif, des dépôts de services et de composants et de l'utilisateur et la notification d'AxSeL pour l'informer des nouvelles contraintes de déploiement.
- *Des mécanismes de déploiement autonome et contextuel* pour les applications orientées services composants. Les algorithmes et heuristiques proposés permettent d'extraire les dépendances fonctionnelles et non fonctionnelles dans un graphe de dépendances global, et d'opérer des décisions prenant en compte plusieurs critères du contexte. AxSeL propose un déploiement autonome et progressif des services et composants obéissant aux contraintes.

Dans le chapitre suivant, nous détaillons la réalisation du prototype AxSeL4OSGi. Nous démontrons également la faisabilité et l'efficacité de notre système à travers un cas d'utilisation réel et des cas de simulation.

Chapitre 4

Prototype AxSeL4OSGi

Dans le chapitre 3 nous avons présenté AxSeL : un intergiciel pour le déploiement autonome et contextuel d'applications orientées services composants. AxSeL repose sur l'extraction d'un graphe expressif et flexible pour représenter les applications, une gestion dynamique du contexte et enfin, sur un ensemble d'heuristiques de déploiement contextuel.

Dans la section 4.1, nous décrivons l'architecture générale de notre prototype. Ensuite, nous illustrons le comportement de notre intergiciel grâce à un cas d'utilisation réel dans la section 4.2. Enfin, nous présentons les évaluations de performance réalisés en simulation en comparaison avec l'architecture OBR dans la section 4.3.

4.1 Réalisation d'AxSeL4OSGi

Dans cette section, nous présentons d'abord les aspects de déploiement du prototype réalisé ainsi que les services intergiciels nécessaires à son fonctionnement. Ensuite, nous abordons le contexte technique d'implémentation. Enfin, nous mettons en évidence la correspondance de nos modèles de service composant avec l'approche OSGi.

4.1.1 Architecture générale

Afin de valider les concepts proposés par AxSeL, nous implémentons le prototype AxSeL4OSGi. Il s'agit de la combinaison d'un ou de plusieurs services AxSeL4OSGi pouvant être déployés sur un ou plusieurs périphériques et accédés localement ou à distance. La figure 4.1 illustre trois types de périphériques électroniques avec des capacités matérielles différentes et sur lesquels AxSeL4OSGi peut être déployé. L'ensemble des services compilés de notre intergiciel a une taille mémoire globale de stockage de 65.5Ko, ce qui lui permet d'être déployé sur des machines contraintes.

Nous adoptons une approche intergicielle qui absorbe les traitements inhérents au déploiement contextuel et les masque à l'utilisateur et à l'application.

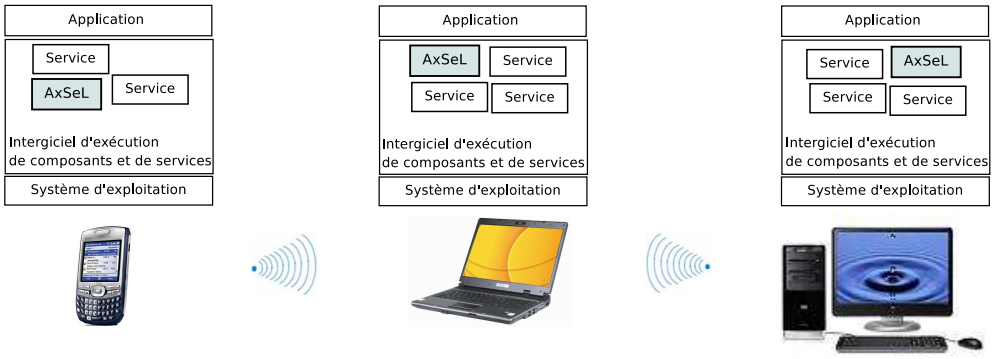

FIG. 4.1 – Déploiement distribué d'AxSeL4OSGi

Afin d'adhérer aux attentes des environnements intelligents, nous avons conçu AxSeL4OSGi selon l'approche de développement logiciel orienté services. Les services d'extraction des dépendances, de décision et de gestion du contexte présentés dans le chapitre précédent peuvent être déployés ensemble ou séparément sur un ou plusieurs périphériques. La figure 4.2 présente deux plates-formes chacune hébergeant un ou plusieurs services AxSeL4OSGi. L'approche décentralisée adoptée est adéquate avec les environnements intelligents.

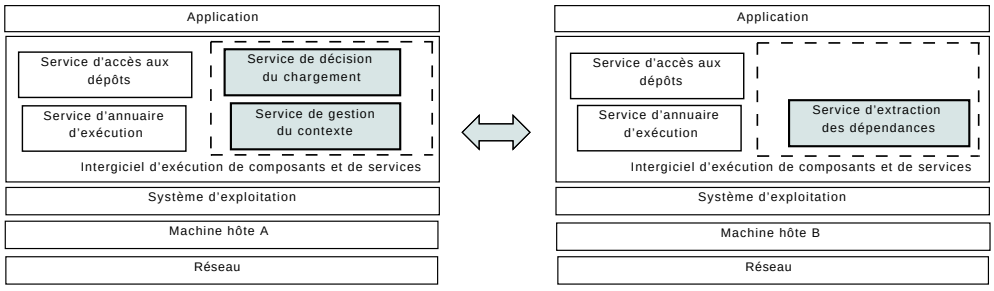

FIG. 4.2 – Architecture décentralisée

AxSeL4OSGi fait partie d'un intergiciel d'exécution de composants et de services et exploite ses services standards tels que le service d'annuaire d'exécution et le service d'accès aux dépôts :

Service d'annuaire d'exécution Le service d'annuaire d'exécution est censé exister sur tous les intergiciels orientés services ou composants. AxSeL4OSGi l'exploite pour rechercher et surveiller les services composants désirés. Il fournit l'accès aux méthodes permettant aux services composants d'interagir avec l'intergiciel. Il donne une vue sur tous les services et composants et offre notamment la possibilité de les enregistrer, de les récupérer, de les installer et de les surveiller à l'exécution. Dans les SOA la publication des services permet de les mettre à disposition pour d'éventuels clients. Le service contextuel d'exécution offre cette fonctionnalité et permet l'enregistrement d'un service composant avec des propriétés qui le renseignent. La spécification OSGi [6] fournit un contexte d'exécution pour les bundles : le *BundleContext* (section 4.5 de la spécification). Ce dernier offre un environnement d'exécution pour chacun des composants et des services, et une fonctionnalité d'enregistrement qui permet à un composant de publier et de renseigner plusieurs interfaces java. Un service enregistré possède une référence qui le distingue des autres de l'intergiciel. Afin d'utiliser ce dernier il suffit d'interroger le service contextuel en indiquant la référence d'enregistrement.

Service d'accès aux dépôts AxSeL4OSGi utilise ce service pour récupérer la liste des dépôts à partir desquels est réalisée la construction du graphe de dépendances. Il fournit l'accès à un ou plusieurs dépôts de services composants. Il permet également la découverte de ressources logicielles selon leurs propriétés contextuelles, la résolution de leurs dépendances et l'administration des différents dépôts. L'API *Bundle Repository* [5] est destinée à la gestion des dépôts OSGi. Elle fournit une abstraction du dépôt de *bundles* et le service *RepositoryAdmin* pour en administrer plusieurs. Ce dernier permet d'ajouter, de retirer un dépôt, mais aussi de lister l'ensemble des dépôts déclarés. La résolution des dépendances d'un ensemble de ressources est faite d'une manière itérative dans un ou plusieurs dépôts. Pour affiner la recherche des ressources dans un dépôt des filtres peuvent être considérés, cependant, la résolution des dépendances fournie effectue un chargement systématique des ressources nécessaires sans opérer de décision orientée utilisateur ou plate-forme matérielle.

Nous avons présenté le déploiement d'AxSeL4OSGi dans un environnement intelligent ainsi que les services avec lesquels il interagit. A présent, nous donnons le contexte d'implémentation de notre prototype.

4.1.2 Implémentation

Dans cette section nous présentons et motivons nos choix des langages et des plates-formes d'implémentation pour développer AxSeL4OSGi (tableau 4.1).

Langages et choix d'implémentation AxSeL4OSGi est développé selon la spécification OSGi. Il est réalisé avec le langage de programmation Java selon le paradigme orienté service. Le langage Java est portable et offre des mécanismes intéressants d'introspection, il possède également des bibliothèques déjà développés ce qui réduit considérablement le temps de développement. D'autant plus que de nos jours de plus en plus de périphériques mobiles sont fournis avec une machine virtuelle java. La technologie OSGi propose un intergiciel orienté

Objectifs	Langage	Outil technique
Développement de l'approche	Java	Jdk et JRE 1.6
Modélisation en services et composants	OSGi	Felix
Visualisation des graphes	Dot	Graphviz
Gestion des métadonnées	Java et XML	kxml2 et XML pull
Génération des descripteurs de dépôts	Java et XML	Bindex
Génération de l'interface graphique	Java et XML	SwiXML

TAB. 4.1 – Langages et outils d'implémentation d'AxSeL4OSGi

services et un environnement de développement basé sur les composants, elle offre également une manière standardisée pour gérer le cycle de vie des composants au dessus des plates-formes Java. Cette technologie est basée sur l'intégration de modules "pré-fabriqués" OSGi ce qui réduit les coûts de développement et de maintenance. En raison de ces nombreux avantages le marché de l'électronique s'est tourné ces dernières années vers cette technologie et en équipe de plus en plus les périphériques mobiles.

Nous avons utilisé le langage XML qui est communément employé dans le domaine pervasif pour sa facilité de manipulation et de compréhension, pour plusieurs objectifs : la description du profil utilisateur, des dépôts de services composants, des graphes de dépendances et enfin de l'interface graphique d'évaluation [20]. Enfin, pour fournir un support visuel pour les graphes de dépendances nous utilisons le langage dot [21].

Plate-forme de développement et d'exécution Nous utilisons la machine virtuelle Java et la plate-forme d'exécution Felix [69] qui est une implantation Apache de la spécification OSGi pour le déploiement des services. AxSeL4OSGi a été développé en utilisant le jdk et Java(TM) Standard Edition Runtime Environment 1.6. Afin de ne pas supporter un coût lié à l'interprétation et au traitement des fichiers XML nous utilisons un parseur léger kxml2 qui nécessite 9 Kb de mémoire de stockage et les API Xml Pull [24] et DOM. Les composants sont décrits grâce à l'outil Bindex [12] qui fournit un descripteur XML directement intégrable dans les dépôts. Nous visualisons les graphes de dépendances en exploitant les outils Graphviz et dotty [14]. Graphviz nécessite une machine virtuelle java et a été récemment développé pour des plates-formes mobiles. Enfin, nous utilisons la chaîne de compilation maven [126] et un packageur (*plugin*) OSGi pour générer automatiquement nos bundles OSGi. Nous avons développé AxSeL4OSGi et effectué les tests d'évaluation sur une machine Dell Lattitude D610 Intel Pentium Microsoft processor 2.13 Ghz sous le système d'exploitation Linux.

Après avoir défini les choix d'implémentation du prototype AxSeL4OSGi, nous présentons dans la section suivante une correspondance des modèles d'AxSeL4OSGi avec le modèle OSGi et *Bundle Repository*.

4.1.3 Adéquation avec OSGi et Bundle Repository

Cette section présente les principes de base de la spécification OSGi et du dépôt de composants *Bundle Repository* et la correspondance que nous faisons entre notre modèle de service composant présenté dans la section 3.3 du chapitre 3 et ces deux approches.

La spécification OSGi L'alliance OSGi [6] fournit un cadre de conception de développement et de déploiement de services pour les réseaux domotiques, et autres types d'environnements. Dans la spécification OSGi la notion d'application n'existe pas, les unités manipulés sont les *bundles*. Un *bundle* OSGi est une archive de classes java, de ressources et librairies java et d'un fichier métadonnées décrivant le bundle et ses dépendances. Il est déposé dans un dépôt de composants appelé *repository*. Un service OSGi est une interface Java fournie par un bundle et enregistrée dans un annuaire. Il peut être implanté de plusieurs manières par différents *bundles* et avoir plusieurs implantations pour une interface. Lorsqu'un bundle est requis, le repository retourne l'ensemble des implantations disponibles. Le modèle de service OSGi se base sur le mécanisme d'annuaire dans lequel les services sont publiés, recherchés et connectés par des clients. Un service est un objet java qui est enregistré sous un ou plusieurs interfaces dans l'annuaire de services. Il est définit par son interface de service et implanté comme un objet. Le bundle fournissant le service doit l'enregistrer pour en permettre l'usage par des tiers et leur offrir ses fonctionnalités. La couche service sous OSGi est définie par les éléments suivants :

- *les interfaces du services* spécifient ses méthodes publiques,
- *la référence des services*,
- *la propriété de service* est un ensemble de paires de clé/valeur associées aux interfaces enregistrées,
- *l'unité de déploiement* qui le bundle OSGi.

Au cours de son cycle de vie, le bundle OSGi passe par les états d'installation, de résolution, de démarrage, d'activation, d'arrêt et enfin de désinstallation. La figure 4.1.3 illustre le passage entre les états.

- *Installé*, cet état est atteint lorsque le bundle est correctement installé.
- *Résolu*, montre la disponibilité des dépendances du bundle et sa disposition à devenir actif.
- *En cours de démarrage*, le bundle enregistre ses services, et obtient les services desquels il a besoin.
- *Actif*, les services peuvent être appelés.
- *Arrêté*, le bundle s'arrête et désenregistre ses services.
- *Désinstallé*, le bundle est désinstallé.

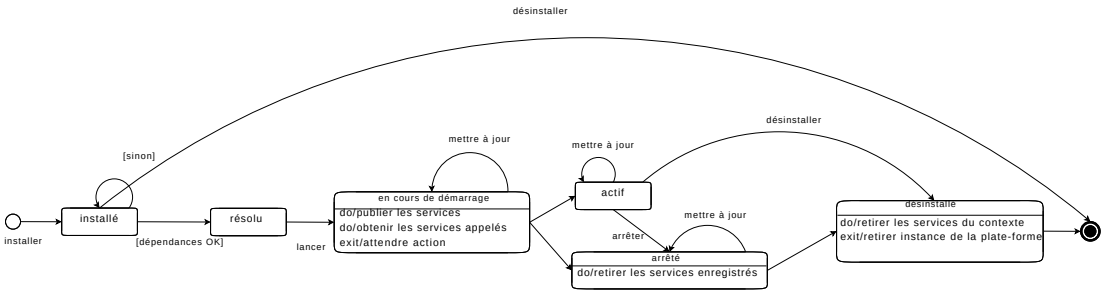

FIG. 4.3 – Cycle de vie d'un bundle OSGi

Les bundles OSGi fournissant les services sont déposés dans des dépôts de services composants de type *Bundle Repository* tels Paremus [18] ou Knopflerfish [15]. Dans ce qui suit, nous

présentons les concepts du dépôt de composants OSGi.

Bundle Repository pour OSGi L'API Bundle Repository est née de plusieurs besoins fonctionnels tels que la découverte et la recherche de composants OSGi dans un dépôt à travers un serveur web selon des propriétés non fonctionnelles (catégorie, mots clés, etc.), mais aussi l'automatisation de l'installation et de la résolution des dépendances, et enfin la possibilité de gérer une fédération de plusieurs dépôts. Et d'autres besoins non fonctionnels tels que la gestion d'un très grand nombre de bundles et la facilité d'utilisation.

OSGi Bundle Repository fournit une description générique d'une ressource et de ses dépendances. Une ressource est un bundle mais peut aussi être un certificat ou un fichier de configuration. La description de la ressource permet sa découverte et son déploiement sans causer d'erreur dues à l'absence des dépendances. Ainsi, chaque ressource possède une liste de requis d'autres ressources de l'environnement. L'API Repository fournit une abstraction pour décrire et gérer les dépendances d'un bundle, elle est constituée des éléments suivants extraits de la RFC [5] :

- *Repository Admin*, est un service qui fournit l'accès à une fédération de dépôts. Il doit s'assurer qu'une ressource ne figure pas plusieurs fois dans le dépôt et qu'il n'existe pas de références circulaires. L'administrateur du dépôt permet la gestion des dépôts par ajout, suppression ou consultation des différents dépôts. D'autre part, il assure la recherche de ressources dans les dépôts et la résolution automatique des dépendances. Des filtres peuvent être appliqués pour limiter les résultats cependant aucune décision contextuelle n'est prise en compte. La résolution des dépendances est assurée par l'API *Resolver*.
- *Repository*, permet l'accès à un ensemble de ressources définies dans un fichier de dépôts. Il inclut uniquement des références vers d'autres dépôts et un ensemble de ressources.
- *Resource*, une ressource fournit une description d'un bundle ou d'un artefact qui peut être installé sur la plate-forme. Elle possède un nom, un identifiant, une description, une taille mémoire, des informations sur la licence, le copyright, mais décrit essentiellement ce que le bundle fournit *Capability*, et ce qu'il requiert *Requirement*.
- *Capability*, représente tout ce qui peut être décrit par un ensemble de propriétés. Une *capability* peut être un export de package ou de service, un bundle, un certificat, un environnement d'exécution, etc. Une *Capability* est décrite par un type (service, bundle, package), nom, une version, etc. La spécification OSGi permet à chaque bundle de fournir des *capabilities* à l'environnement.
- *Requirement*, il s'agit d'une assertion sur les *capabilities* d'une ressource. Les *requirements* décrivent les imports réalisés par une ressource. Cela peut être des imports de packages, de bundles, de services, etc. Les *requirements* sont exprimés comme des filtres sur une ressource. Ils possèdent un nom, une description et des filtres contextuels. Un *requirement* correspond à une *capability* qui a le même nom que son filtre. Il peut être multiple ou optionnel. Les dépendances optionnelles peuvent être satisfaites par un ou plusieurs ressources.
- *Extend*, une ressource peut fournir une extension à une autre. Le mécanisme d'extension inverse celui des *requirements* dans le sens où la ressource choisit elle même les ressources

auxquelles elle peut être utile.

- *Resolver*, il s'agit d'un objet qui peut être exploité pour retrouver et installer les ressources dépendantes et leurs extensions. Le *resolver* prend en entrée un ensemble de bundles et calcule leurs dépendances.
- *Repository File*, est un fichier XML qui peut être référencé par une URL et qui contient la méta donnée des ressources et des références vers d'autres fichiers de repository. Il s'agit d'un fichier statique ou généré par le serveur. Le listing 4.1 illustre un fichier exemple qui comporte une ressource exportant un bundle et un service et important le package OBR.

```
<repository name='Untitled' time='20051210072623.031'>
   <resource version='3.0.0' name='org.osgi.test.cases.tracker'>
     <size>44405</size>
     <documentation>http://www.osgi.org/</documentation>
     <capability name='bundle'>
       <p v='1' n='manifestversion'/>
       <p v='org.osgi.test.cases.tracker' n='symbolicname'/>
       <p v='3.0.0' t='version' n='version'/>
     </capability>
      <capability name='service'>
       <p v='org.osgi.test.cases.tracker' n='symbolicname'/>
     </capability>
     <require optional='false' multiple='false' name='package'>
       Import package org.osgi.obr ;version=1.1.0
     </require>
 </resource>
</repository>
```

Listing 4.1 – Exemple de descripteur de dépôt

Adéquation avec OSGi et Bundle Repository La correspondance entre le modèle OSGi, le dépôt et notre modèle de service composant est illustrée figure 4.4. Les zones grisées représentent les concepts de la spécification OSGi et du dépôt *Bundle Repository*, ils sont décrits ci-après.

- *OSGi Service interface*, correspond dans notre modèle aux descriptions fonctionnelles des services, cela inclut la liste des méthodes fournies par celui-ci. Si le service propose plusieurs interfaces elles seront considérées comme des services indépendants.
- *OSGi References*, elles correspondent aux descriptions fonctionnelles et non fonctionnelles des services et des composants et fournissent leurs méta données.
- *OSGi Property*, correspond aux informations avec lesquelles sont enregistrés les services sur la plate-forme. Cela permet de renseigner les clients du service sur les implémentations de celui-ci.
- *OSGi Bundle*, correspond au composant AxSeL, ses implémentations et ses états.
- *Protocol*, il s'agit des techniques d'implantation et de communication entre les services composants, AxSeL4OSGi se base sur la technologie OSGi et Java.
- *OSGI Bundle Context*, correspond à l'annuaire d'exécution des services composants qui listent les interfaces enregistrées sur la plate-forme.
- *OSGi Service Model*, AxSeL4OSGi obéit à la spécification OSGi et à son modèle de service.

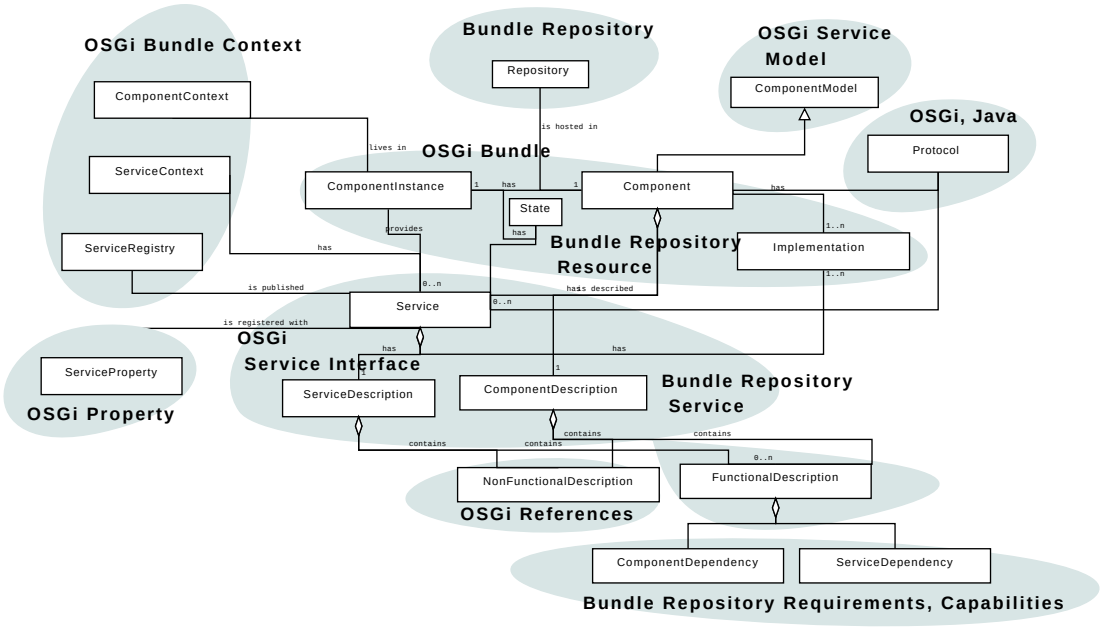

FIG. 4.4 – Adéquation avec OSGi

- *Bundle Repository*, correspond dans notre modèle au *Repository* dans lequel sont déposés les services composants. Les dépôts de services composants OSGi respectent le modèle présenté précédemment.
- *Bundle Repository Resource*, les ressources déposées dans le *Bundle Repository* correspondent dans notre modèle aux unités de déploiement des services composants.
- *Bundle Repository Requirements, Capabilities*, les *Requirements* et *Capabilities* dans le *Bundle Repository* correspondent aux dépendances de services et de composants dans notre modèle. Les premiers renseignent les imports et les seconds les exports.

Dans cette section, nous avons présenté chacun des modèles OSGi et *Bundle Repository* et réalisé une correspondance entre ceux-ci et notre modèle de service composant. Dans ce qui suit, nous illustrons le comportement de notre prototype sur un cas d'utilisation réel : une application de visualisation de documents PDF.

4.2 Scénario d'une application de services documents

L'objectif de notre prototype est de prouver la faisabilité de notre intergiciel AxSeL et des concepts proposés, en montrant sa capacité à réaliser un déploiement contextuel et autonome basé sur une vue flexible et une capture dynamique du contexte d'exécution. Les étapes d'AxSeL4OSGi à illustrer sont l'extraction de la vue globale et flexible, la prise de décision contextuelle et l'adaptation aux fluctuations de l'environnement.

L'application considérée propose des services documents qui réalisent la visualisation de documents PDF. Nous l'avons créé et décidé de la granularité de ses services et de ses composants. L'application est composée par les éléments suivants et présentée dans le tableau 4.2 :

- Un service PDF qui fournit le document en format PDF,
- Un service de navigation dans le document pdf, la navigation peut se faire :
  - à travers le clavier (keyNavigator), ou bien
  - à travers une interface utilisateur qui s'affiche sur l'écran avec des boutons de navigation et qui facilite la navigation (widgetNavigator).

Bundles	Services	Mémoire en o	Désignation
`pdfkeynavigator-0.0.1.jar`	-	5135	A
`pdfnavigatorservice-0.0.1.jar`	-	3093	B
`pdfserver-0.0.1.jar`	PDFServiceIfc	236837	C-S1
`pdfservice-0.0.1.jar`	-	2923	D
`pdfviewer-0.0.1.jar`	-	10250	E
`pdfwidgetnavigator-0.0.1.jar`	DocumentNavigatorService	14102	F-S2

TAB. 4.2 – Composants de l'application

Pour des raisons de lisibilité sur les schémas nous avons assigné à chacun des composants et services de l'application des désignations simplifiées. Les bundles C et F exportent respectivement les services S1 et S2. Il est à noter que la taille du service S1 dépend de la taille du fichier à visionner.

Dans ce scénario nous prenons en compte les propriétés non fonctionnelles concernant l'utilisateur et le terminal : les préférences de services et l'usage mémoire. Afin de réaliser l'évaluation d'AxSeL4OSGi nous fournissons une interface graphique simple présentée dans la figure 4.5 pour que l'utilisateur renseigne ses paramètres.

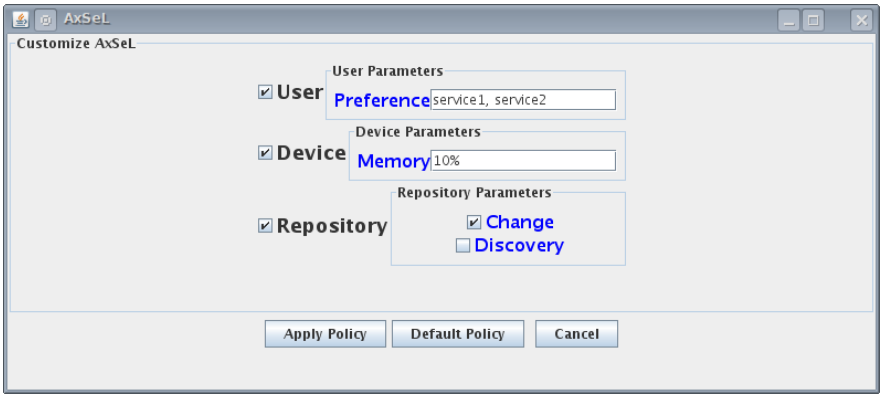

FIG. 4.5 – Configuration d'AxSeL4OSGi

La configuration peut être qualitative orientée préférences utilisateurs et quantitative orientée optimisation de la mémoire. Ainsi, l'utilisateur indique la liste des services qu'il souhaite avoir en priorité et s'il le désire la limite de la valeur mémoire à laquelle il voudrait être notifié. Lorsque la contrainte mémoire n'est pas choisie par l'utilisateur il est quand même alerté automatiquement lorsqu'elle est critique. Ensuite, afin de ne pas supporter un coût de capture d'un contexte non pertinent à l'utilisateur nous laissons le choix d'activer les paramètres qu'il désire prendre en considération. Par conséquent, s'il décide de ne pas prendre en compte les

découvertes de nouveaux dépôts ou les modifications de ceux hébergeant les services composants il doit décocher le bouton adéquat. Il est aussi tout à fait possible d'utiliser AxSeL4OSGi sans interface graphique.

Une fois les contraintes non fonctionnelles renseignées par l'utilisateur, AxSeL4OSGi applique la politique désirée et procède aux étapes d'extraction du graphe de dépendances, de décision contextuelle et d'adaptation dynamique en cas de besoin. Dans ce qui suit, nous illustrons chacune de ces étapes en déroulant le scénario de l'application de visualisation du document PDF.

4.2.1 Extraction du graphe de dépendances

L'extraction des dépendances est le processus par lequel AxSeL4OSGi construit l'ensemble des liens des services et composants à charger dans une unique vue. Ce processus part d'une description statique des dépendances et fournit un graphe globale et flexible. Nous présentons le descripteur de dépôt de services composants de notre application et la génération de la vue globale. Enfin, nous illustrons le graphe obtenu en résultat à l'étape d'extraction d'AxSeL4OSGi.

Le listing 4.3 présente un extrait du code de la fonction qui lance l'extraction des dépendances, cela se fait par appel à la fonction *extractGraph(resource)* qui prend comme paramètre d'entrée la ressource à résoudre (E dans notre cas). Le processus d'extraction est ensuite réalisé par les fonctions d'extraction qui parcourent récursivement le dépôt (repositoryPDFViewer dans notre cas) à la recherche des dépendances de la ressource.

```
1  public void load(Repository repositoryPDFViewer, Resource E) {
2    if ((repository != null) && (resource != null)) {
3      if (this.m_repository != repository) {
4        this.m_repository = repositoryPDFViewer;
5      //Construction de la classe d'extraction
6      this.m_repositoryExtractor = new CommonRepositoryExtractor(
           repositoryPDFViewer);
7      //Extraction des dépendances de la ressource E
8      this.m_resourceGraph = this.m_repositoryExtractor.extractGraph(E);
```

Listing 4.2 – Extrait de la fonction d'extraction du graphe de dépendances

Descripteur OSGi du dépôt Le descripteur standards de bundles OSGi représente principalement les dépendances entre bundles et packages et ne mentionne le niveau de dépendances entre services que d'une manière facultative. Pour nos besoins de conception et d'expressivité, nous mettons en évidence cette mention d'imports et d'exports de services dans les descripteurs de dépôts considérés. Nous exploitons l'outil bindex pour générer le descripteur de dépôt pour notre application.

Exemple 1 *Le listing 4.3 décrit le service PDF Viewer de notre cas d'utilisation. Nous ne présentons qu'un extrait du descripteur de dépôt.*

```
1 <repository>
2 <resource id='pdfviewer/0.0.1' presentationname='PDF viewer'>
```

```
3  <description>PDF document viewer</description>
4  <size>10250</size>
5  <capability name='bundle'>
6      <p n='manifestversion' v='2'/>
7      <p n='presentationname' v='PDF viewer'/>
8      <p n='symbolicname' v='pdfviewer'/>
9      <p n='version' t='version' v='0.0.1'/>
10 </capability>
11 <require>Import Service pdfnavigatorservice.DocumentNavigatorService</require>
12 <require>Import Service fr.inria.amazones.pdfservice.PDFServiceIfc</require>
13 <require>Import package fr.inria.amazones.pdfservice</require>
14 <require>Import package pdfnavigatorservice</require>
15 </resource>
16 </repository>
```

Listing 4.3 – Descripteur du dépôt de PDFViewer

- *La ressource PDFViewer fournit à l'environnement le bundle PDFViewer. La notion d'export est exprimée par le terme capability.*
- *Les besoins en services et composants sont exprimés par le terme require. Le PDFViwer importe les services DocumentNavigator et PDFServiceIfc.*

Les autres services composants de l'application sont décrits de la même manière que le PDFViewer. A partir du descripteur BundleRepository, nous déduisons le tableau des dépendances des services et composants de l'application de visualisation de documents PDF.

Eléments de départs	**Dépendances**
A	B
B	-
C-S1	-
D	-
E	C-S1,F-S2,D,B
F-S2	A,B—F,B

TAB. 4.3 – Tableau des dépendances entre les services et composants du scénario

Génération de la vue globale La génération de la vue globale se fait sur la base du descripteur du dépôt de services composants. Nous faisons la correspondance du modèle de dépôt de services composants *BundleRepository* avec le modèle du graphe de dépendances proposé par AxSeL4OSGi et présenté dans le chapitre 3, section 3.3. Les parties grisées de la figure 4.2.1 montrent la correspondance entre les deux modèles. Les éléments suivants y sont cités :

- *Resource.* Une ressource référencée dans le dépôt correspond à un nœud de *AxSeL4OSGiGraph*. Les nœuds du graphe peuvent représenter indépendemment les services et les composants (*AxSeL4OSGiGraphNode*).
- *Requirement, Capability.* Ils représentent les références d'import et d'export de services et de composants. Elles sont représentées dans notre modèle par l'arc du graphe *AxSeL4OSGiGraphEdge* qui lie deux nœuds.

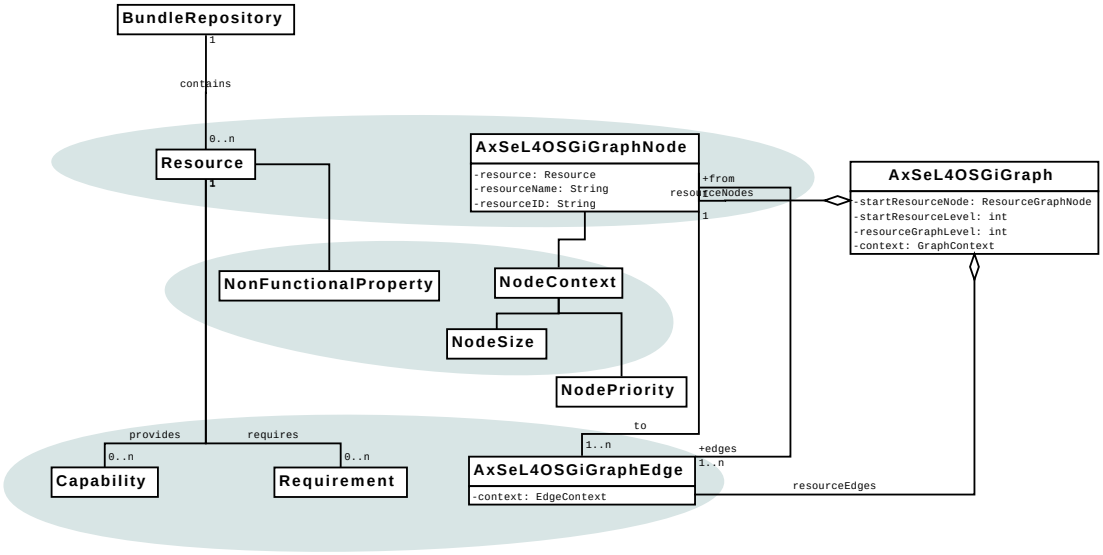

FIG. 4.6 – Adéquation entre le modèle du repository OSGi et le graphe de dépendances AxSeL4OSGi

- *resourceGraphLevel.* Les éléments importés et exportés possèdent sont de type service ou bundle. Chacune des ressources et des services sont assignés au niveau correspondant à leur type. AxSeL4OSGi prévoit deux niveaux : 0 pour les composants et 1 pour les services.
- *NonFunctionalProperty.* Les ressources dans le dépôt sont décrites par des données non fonctionnelles renseignant la taille mémoire des bundles. Nous exploitons cette information l'assignons aux nœuds du graphe extrait *NodeSize*. La priorité (*NodePriority*) est renseignée par l'utilisateur au départ. Elle est indiquée au niveau de chaque nœud par un entier. Nous supposons que la priorité indiquée par l'utilisateur sur un nœud inonde ses dépendances. Les propriétés non fonctionnelles des nœuds sont exprimées dans notre modèle par *NodeContext*.

Exemple 2 *Nous exécutons la fonction d'extraction des dépendances dans le cas de notre scénario d'illustration. Le graphe obtenu compte huit nœuds dont deux correspondent aux services et les six autres aux bundles de l'application. Le résultat est illustré dans la figure 4.7. Nous y distinguons les éléments suivants :*

- *Niveaux d'exécution et de déploiement. Les environnements d'exécution et de déploiement sont représentés par le niveau 1 (level 1) pour les services et 0 (level 0) pour les bundles.*
- *Propriétés du nœud. Au niveau des nœuds les propriétés non fonctionnelles sont indiquées et renseignent la taille mémoire et la priorité.*
- *Noeud d'entrée du graphe. Le nœud d'entrée du graphe est distingué par une bordure double, dans notre cas, il s'agit du nœud E correspondant au PDFViewer.*
- *Opérateur logique OU L'opérateur logique OU apparaît au niveau du choix entre les deux éléments de navigation par clavier ou par widget (A et F). En effet, le service S2 peut être fourni par le bundle A ou le bundle F, ce qui explique le point de bifurcation au niveau de la dépendance entre S2 et ses bundles.*

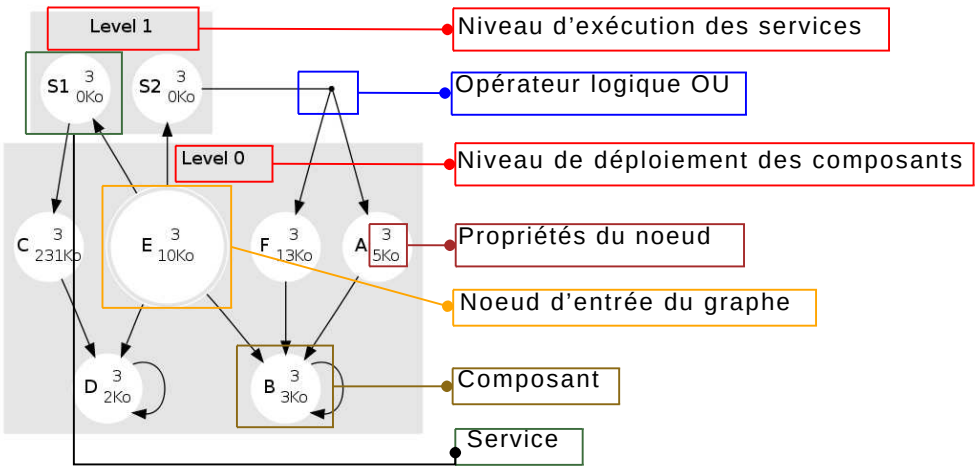

FIG. 4.7 – Graphe de dépendances de l'application PDFViewer politique mémoire

Conclusion Dans cette section, nous avons présenté le fonctionnement du service d'extraction des dépendances d'AxSeL4OSGi. Ensuite, nous avons fourni le descripteur de OSGi des composants et services formant notre application d'illustration, et généré notre vue globale et flexible de l'application. Enfin, nous avons obtenu le graphe des dépendances de l'application de visualisation de documents PDF résultant de cette étape. Dans ce qui suit, nous présentons le processus de prise de décision de déploiement exploitant ce graphe.

4.2.2 Décision contextuelle

Le processus de décision consiste en la confrontation des contextes requis avec les contraintes exprimées par l'utilisateur et le dispositif matériel. A cette étape, AxSeL4OSGi réalise un parcours du graphe de dépendances extraits dans l'étape précédente et évalue l'installabilité de chacun des nœuds. Cette opération est assurée par le service de décision du chargement qui applique différentes heuristiques multi-critère prenant en compte les paramètres de haut niveau portant sur les services et/ou de bas niveau d'ordre matériel. Ces heuristiques se basent sur l'algorithme glouton présenté dans le chapitre 3, section 3.5.2.

Décision du chargement L'heuristique de coloration contextuelle exploite le graphe de dépendances extrait à l'étape précédente. Lorsque les contraintes contextuelles sont fixées le processus de décision est initié. D'abord, les nœuds du graphe sont récupérés et triés selon l'ordre voulu des propriétés. Ce chemin trié est retraité par l'heuristique pour une évaluation locale des contraintes de chaque nœud. Les nœuds respectant les contraintes contextuelles sont colorés en rouge et chargés sur la plate-forme, les autres en blanc. Les nœuds non traités lors du premier passage de l'heuristique sont mis dans une liste d'attente pour être pris en compte lors du prochain passage. Ainsi, AxSeL4OSGi charge progressivement tous les nœuds de l'application.

La décision appliquée par AxSeL4OSGi prend en entrée le graphe de dépendances extrait dans l'étape précédente, la priorité minimale tolérée pour les nœuds à charger et la taille

mémoire maximale à ne pas dépasser. Le listing 4.4 présente des extraits du code de la fonction de coloration du graphe.

```
1  public PSConstraintPPathColouringDecision(ResourceGraph pdfViwerGrah, int
       priority, int sizeMax) {
2    super(resourceGraph);
3    //Initialisation du parcours
4    this.m_totagNodes = new ResourceGraphPriorityColouringPath();
5    //Ajout des noeuds du graphe au chemin à traiter
6    this.m_totagNodes.add(this.m_resourceGraph.getResourceGraphEntryNode());
7    //Tri les noeuds à colorier en premier selon la priorité
8    this.m_restartNodes = new ResourceGraphPriorityColouringPath();
9    //Création de la contrainte de parcours
10   this.m_resourceGraphNodeConstraint = new
         ResourceGraphNodePrioritySizeConstraint(prioMin, sizeMax);
11   this.m_resourceLoading = new BundleRepositoryResourceLoading();
12 }
```

Listing 4.4 – Extraits du code de la fonction de coloration du graphe

L'heuristique de chargement contextuel suit les étapes suivantes :

1. **Initialisation du chemin.** Le chemin de coloration des nœuds est initialisé (ligne 4). Ensuite, le nœud d'entrée du graphe est ajouté à cette liste, cela permet également l'ajout de tous les nœuds du graphe à la liste (ligne 6).
2. **Tri des nœuds blancs.** Les nœuds non chargés au premier passage sont triés dans l'ordre pertinent. *ResourceGraphPriorityColouringPath* désigne la liste des nœuds à colorier rangés selon l'ordre souhaité. Dans notre cas, les nœuds du chemin de parcours sont rangés selon la priorité et la taille. (ligne 8)
3. **Création de la contrainte de parcours.** *ResourceGraphNodePrioritySizeConstraint* représente les contraintes de parcours prises en compte par l'algorithme lors de son parcours de graphes. Dans le cas de notre scénario il s'agit de la priorité et la taille mémoire. Nous renseignons la priorité par un entier appartenant à un intervalle allant de 1 à 5 du moins important au plus important. La mémoire est indiquée par un entier renseignant les kilo octets. (ligne 10)
4. **Parcours et évaluation des nœuds.** L'évaluation de l'installabilité de chaque nœud est réalisé en fonction de l'adéquation de ses propriétés avec les contraintes de chargement priorité et taille mémoire. Elle est propagée sur toutes les dépendances dans le graphe.

Stratégies de déploiement autonome et contextuel En variant les contraintes contextuelles et leurs prises en compte, il est possible de décliner plusieurs stratégies de déploiement. Par exemple dans un contexte contraint les stratégies orientées ressources matérielles sont à prendre en compte en priorité par rapport à d'autres. Dans d'autres cas l'utilisateur peut choisir de ne considérer qu'une propriété par rapport à une autre. Dans ce qui suit nous illustrons différentes stratégies appliquées sur l'application de visualisation de documents PDF. Nous montrons que selon les contraintes considérés le comportement du déploiement n'est pas le

même. Les exemples suivants traitent dans l'ordre une stratégie de déploiement multi-critère couplant la taille mémoire et la priorité des nœuds (exemple 3), une stratégie qui considère uniquement la priorité (exemple 4) et enfin une stratégie qui ne prend en compte que la mémoire (exemple 5).

Exemple 3 ***Stratégie Priorité et Taille mémoire*** *Le processus de décision contextuelle est appliqué à notre cas d'utilisation. Nous supposons que l'utilisateur a désigné les deux services S1 et S2 en tant que prioritaires. Par conséquent, les mêmes valeurs de priorité leur sont assignés ainsi que leurs composants (3 dans notre cas). Dans cet exemple, AxSeL4OSGi applique une stratégie couplant les deux contraintes priorité et mémoire, le graphe obtenu est illustré figure 4.8.*

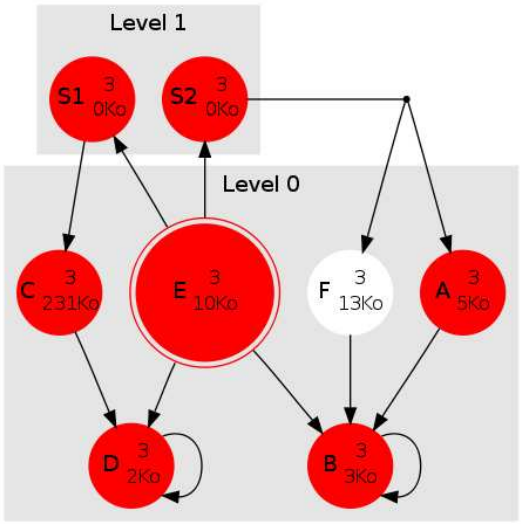

FIG. 4.8 – Graphe de dépendances de l'application après décision

Les nœuds respectant les contraintes sont coloriés en rouge sinon en blanc. En raison des limites de la mémoire disponible sur le périphérique, la décision du chargement a abouti à une navigation par clavier plutôt que par widget. En effet, les nœuds F et A ont la même priorité et sont séparés par l'opérateur logique OU. Dans ce cas, le processus de décision opte pour le nœud le plus petit qui ne consomme pas beaucoup de ressources matérielles. Le déploiement de l'application lance le visionneur de documents PDF illustré figure 4.9. A défaut de mémoire disponible, les services chargés permettent une navigation minimale à travers les flèches du clavier.

FIG. 4.9 – Visionneur de documents PDF adapté sans l'assistant de navigation

Exemple 4 ***Stratégie Priorité*** *La politique de décision adoptée est extensible et peut considérer un attribut non fonctionnel plutôt qu'un autre. L'utilisateur peut choisir de ne tenir compte que de la priorité des nœuds et écarter l'attribut taille mémoire. Dans ce cas, les nœuds seront évalués par rapport à leur adéquation avec la liste des services renseignés auparavant par l'utilisateur. Cette stratégie est appliquée lorsque la mémoire est abondante, évidemment lorsque la mémoire du dispositif est insuffisante le chargement est entravé.*

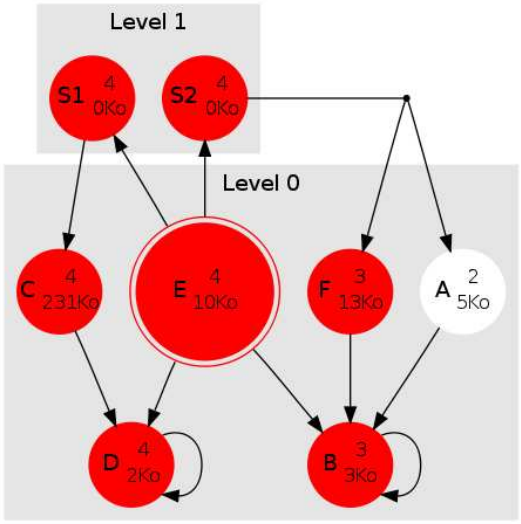

FIG. 4.10 – Graphe de dépendances de l'application PDFViewer en appliquant la politique de priorité

Nous illustrons ce cas en considérant des nœuds de priorité différentes, ainsi A, B et F sont d'ordre inférieur aux autres. Dans ce cas AxSeL4OSGi ne choisit pas le nœud A car il est moins prioritaire que le nœud F. La figure 4.10 illustre le graphe de dépendances de l'application

PDF Viewer. La décision prise par AxSeL4OSGi lance le visionneur de PDF avec l'assistant de navigation présenté dans la figure 4.11.

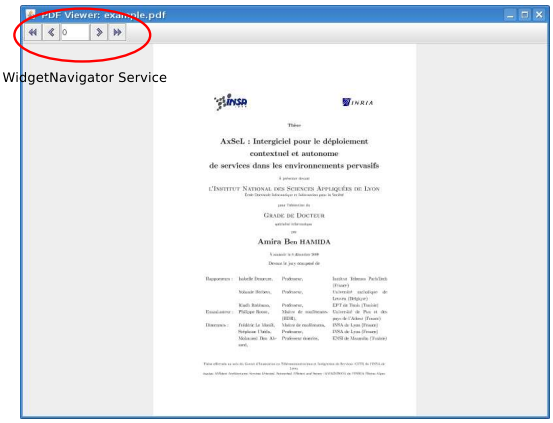

FIG. 4.11 – Visionneur de documents PDF avec un assistant de navigation

Exemple 5 ***Stratégie Taille mémoire*** *Dans cet exemple nous illustrons un scénario qui prend en compte uniquement la taille mémoire des nœuds par rapport à une taille maximale disponible de 70Ko. Dans ce cas, le processus réalisé par AxSeL4OSGi ne procède pas à un tri des nœuds selon leur priorité mais évalue uniquement la somme des tailles mémoires des nœuds à charger par rapport à la mémoire disponible sur le périphérique.*

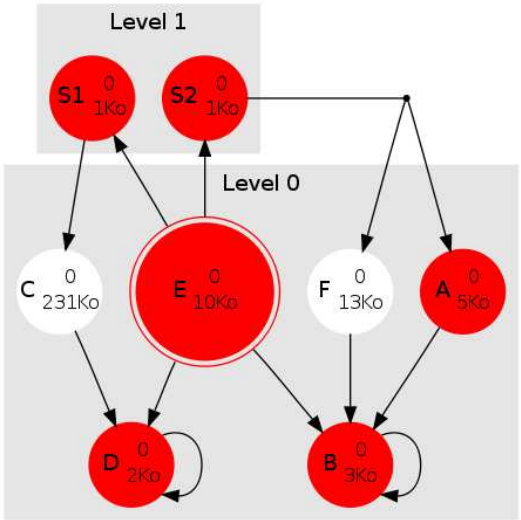

FIG. 4.12 – Graphe de dépendances de l'application PDFViewer politique mémoire

La figure 4.12 illustre le graphe de dépendances de l'application PDF Viewer après décision. Les priorités ont des valeurs nulles car elles ne sont pas prises en considération. A cause de la limite de la mémoire AxSeL4OSGi ne charge pas les nœuds C et F car trop volumineux. L'absence de la navigation par widget fournie par le nœud F n'entrave pas le fonctionnement de

l'application car elle est remplacée par la navigation par clavier fourni par le nœud A, cependant, celle de C ne permet pas au visionneur de se lancer. Ultérieurement et si cela est possible, notre intergiciel tentera de charger en priorité le service C ou son équivalent moins volumineux dès que la mémoire sera disponible ou que l'équivalent apparaîtra dans l'environnement.

Conclusion Dans cette section, nous avons montré la faisabilité du processus de décision réalisé par notre prototype AxSeL4OSGi à travers des exemples concrets appliqués sur le cas du visionneur de documents PDF. Notre intergiciel a pris en considération plusieurs propriétés relevant des préférences utilisateurs et du matériel et a su fournir un déploiement autonome et contextuel de l'application. Un mode progressif de décision a été mis en évidence de manière à déployer en priorité les nœuds importants et peu gourmands en mémoire, ensuite de faire suivre les autres. Nous avons également démontré la possibilité d'usage de plusieurs politiques de déploiement et l'impact que cela avait sur le graphe de dépendances et le fonctionnement de l'application. Dans ce qui suit, nous présentons l'étape d'adaptation dynamique au contexte.

4.2.3 Adaptation dynamique

Dans cette section nous illustrons l'adaptation réalisée par AxSeL4OSGi face aux événements du contexte. Ceux-ci proviennent du dépôt de services et de composants, du dispositif ou de l'utilisateur. Prendre en compte ces trois types d'événements nous permet d'abord, d'être conscients des nouvelles mises à jour de services et de composants et donc d'obtenir une vue actualisée de l'application déployée. Ensuite, de fournir un déploiement qui est sensible aux ressources matérielles et enfin d'adhérer aux besoins de l'utilisateur. Ces adaptations sont effectuées d'une manière autonome sans aucune intervention de l'utilisateur.

Changement du dépôt Le dépôt de services et de composants évolue par ajout, retrait ou modification des éléments. Pour chacun de ces cas de figure, AxSeL4OSGi préconise des fonctions d'adaptation contextuelle et dynamique du déploiement selon la nature de l'événement provenant du dépôt. Dans ce qui suit, nous présentons et illustrons à travers des exemples concrets ces cas.

Ajout d'un nœud Lorsqu'un composant ou un service est ajouté au dépôt, il est nécessaire de créer le nœud correspondant et de le connecter au graphe de dépendances de l'application concernée. Ensuite, d'adapter le déploiement en prenant en compte cette nouvelle entrée. Le nœud est ajouté dans le niveau correspondant. Enfin, nous relançons le coloriage du graphe pour que les changements soient pris en compte. Dans le cas contraire, nous testons s'il appartient aux nœuds visités, changeons ses propriétés et évaluons s'il obéit toujours aux contraintes. S'il obéit aux contraintes il est coloré en rouge et chargé. Sinon, il est coloré en blanc et déchargé. Nous lançons ensuite la fonction de décoloration récursive sur l'ensemble des nœuds dépendants cela, les décolore et les décharge de la plate-forme.

Exemple 6 *Dans cet exemple, nous illustrons le cas d'ajout d'un composant fournissant la fonctionnalité d'impression à l'application de visualisation de documents PDF, pour des raisons de lisibilité nous l'appelons le nœud X. X est ajouté au niveau du dépôt tel que présenté dans*

le listing 4.5 en tant que dépendances du composant D PDF service. Il est également nécessaire de mettre à jour le nœud D.

```
1  <resource id='X' presentationname='print' symbolicname='X'>
2      <description>Document Printer </description>
3      <size>5400</size>
4      <capability name='bundle'>
5       <p n='presentationname' v='print'/>
6      </capability>
7  </resource>
8  <resource id='D' presentationname='PDF service' symbolicname='D'>
9      <description>PDF document service</description>
10     <size>2923</size>
11     <capability name='bundle'>
12       <p n='manifestversion' v='2'/>
13       <p n='presentationname' v='PDF service'/>
14       <p n='symbolicname' v='D'/>
15     </capability>
16     <require extend='false' name='package'>Import package fr.inria.amazones.D
           </require>
17     <require extend='false' name='bundle'>Import package X</require>
```

Listing 4.5 – Extraits du descripteur du respoitory OSGi du PDF Viewer

Le changement du dépôt entraîne l'adaptation de la décision en intégrant les paramètres du nouveau nœud. La recoloration du graphe débute des nœuds non coloriés lors du premier passage de l'algorithme et ce pour aboutir à la couverture de l'ensemble du graphe. Les figures 4.13 et 4.14 illustrent l'adaptation réalisée au niveau du graphe de dépendances de l'applications de visualisation de document PDF. Le nouveau nœud n'a pas été coloré en rouge et donc pas chargé

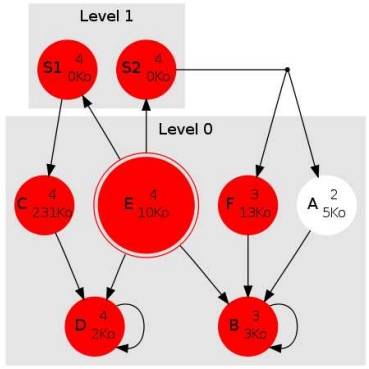

FIG. 4.13 – Avant l'ajout d'un nœud

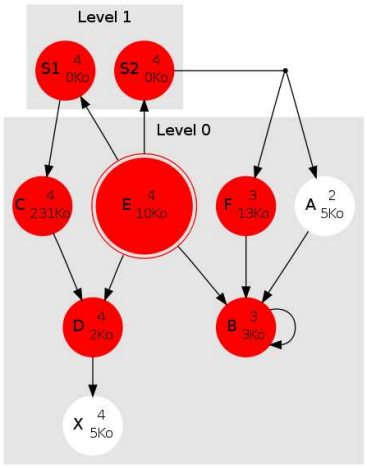

FIG. 4.14 – Ajout d'un nœud à la volée

car la limite de la mémoire est atteinte. Cependant, il est ajouté à la liste prioritaire lors du prochain parcours du graphe.

Retrait d'un nœud Lors d'une mise à jour au sein du dépôt par livraison de nouvelles versions par exemple, il est possible que les anciens composants aient à disparaître. Lorsqu'un composant ou un service est retiré du dépôt de services et de composants, cela est répercuté sur le graphe de dépendances de l'application. Le retrait d'un nœud implique sa décoloration ainsi que celle de ses dépendances excepté celles qui sont déjà requises par des nœuds chargés (rouge).

Exemple 7 *Nous présentons le cas du retrait du nœud F. Les figures 4.15 et 4.16 illustrent le passage entre les deux étapes, nous reprenons l'état initial tel que considéré précédemment. AxSeL4OSGi le décolore et le décharge de la plate-forme et itère sur ses dépendances pour propager ce traitement. Cependant, le nœud F dépend de B qui lui est chargé sur la plate-forme et requis par le nœud E. Pour protéger les dépendances requises par E, notre intergiciel ne décharge pas le nœud B. Dans cet état l'application de visualisation se lancera mais sans permettre aucune navigation dans le document.*

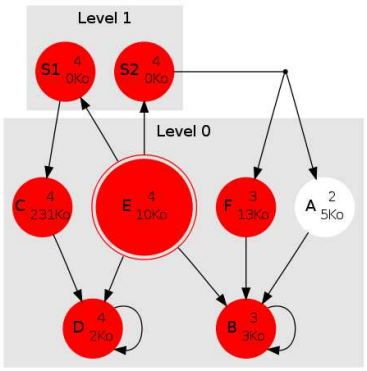

FIG. 4.15 – Avant le retrait d'un nœud

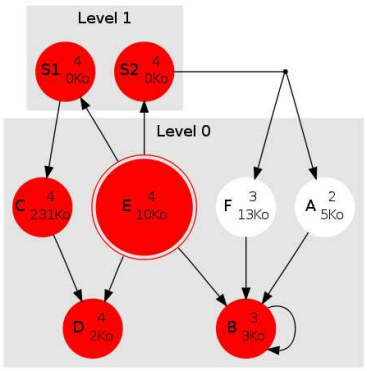

FIG. 4.16 – Retrait d'un nœud à la volée

Modification d'un nœud Les propriétés des nœuds peuvent changer au sein du dépôt des services et des composants. Ainsi, une nouvelle mise à jour du nœud peut être livrée et celle-ci peut nécessiter un espace mémoire différent. Dans tous les cas, la décision du déploiement est reconsidérée. Si le nœud mis à jour est moins volumineux que l'ancien, cela entraîne la libération de la mémoire et donc le chargement probable d'autres nœuds. Sinon, si le nœud mis à jour est plus volumineux cela peut entraver son chargement dans le cas où la mémoire est limitée. Si le nœud ne peut être chargé AxSeL4OSGi procède à une décoloration récursive de ses dépendances tant qu'elles ne sont pas requises par d'autres nœuds.

Exemple 8 *Dans cet exemple, nous illustrons le cas du changement de la taille mémoire d'un nœud. Nous évaluons le cas où la taille du composant diminue par mise à jour. Le nœud C a une taille mémoire de 231Ko et n'a pas pu être chargé au départ sur la plate-forme à cause de son volume inadéquat. Si nous supposons qu'une nouvelle mise à jour du même composant a été livrée et qu'elle ne compte que 91Ko de mémoire, ceci va permettre au nœud C d'être déployé sur la plate-forme.*

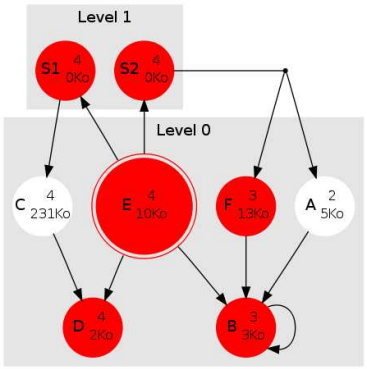

FIG. 4.17 – Avant la modification d'un nœud

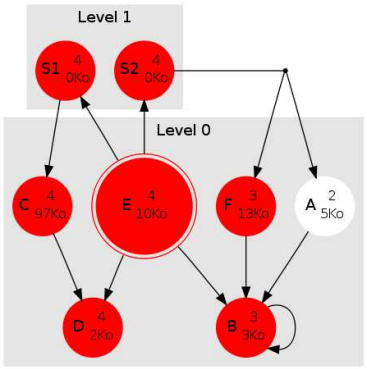

FIG. 4.18 – Modification du nœud

Les figures 4.17 et 4.18 illustrent les graphes de dépendances avant et après l'adaptation réalisée par AxSeL4OSGi. En effet, si le nœud C dans le premier cas est trop volumineux pour être chargé, il est désormais installable dans le deuxième cas et coïncide avec les contraintes de taille mémoire du dispositif.

Changement des ressources matérielles du dispositif La variation positive ou négative de la mémoire disponible sur le dispositif entraîne des changements dans le graphe de dépendances et sur les composants et services déployés. Lorsqu'une diminution de la mémoire est observée, AxSeL4OSGi tente de libérer l'espace mémoire utilisée en déchargeant certains composants et services.

Exemple 9 *Nous présentons le cas de la diminution de la mémoire disponible sur le dispositif. Les figures 4.19 et 4.20 illustrent le déchargement des nœuds à cause de la diminution de la mémoire.*

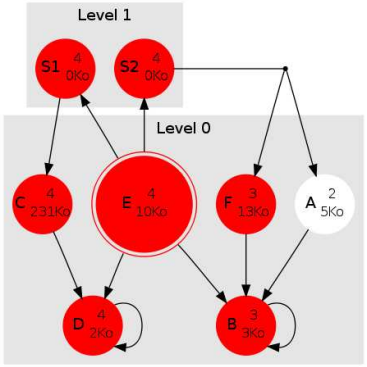

FIG. 4.19 – Avant la diminution de la mémoire

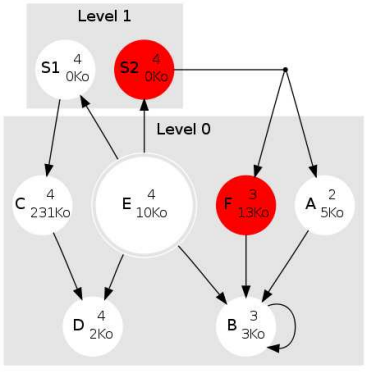

FIG. 4.20 – Après diminution de la mémoire

Changement des préférences utilisateur L'utilisateur peut décider de modifier ses préférences sur les services et composants à déployer en fonction de ses besoins, par exemple,

s'il estime qu'il n'a pas besoin du service d'impression, il mettra ce service avec une priorité minime. De tels changements modifient le processus de décision de manière à ce que les services tolérés avant ne le soient plus et vice versa.

Exemple 10 *Dans cet exemple, nous présentons deux cas de figure où l'utilisateur a d'abord mis en priorité le service pdfserver(S1) et ses dépendances, par rapport au service de navigation dans le document (S2) et ses dépendances. Dans ce cas, AxSeL4OSGi réalise un déploiement qui charge en premier les nœuds prioritaires, soient S1 et ses dépendances. Le graphe du déploiement de l'application est illustré figure 4.21.*

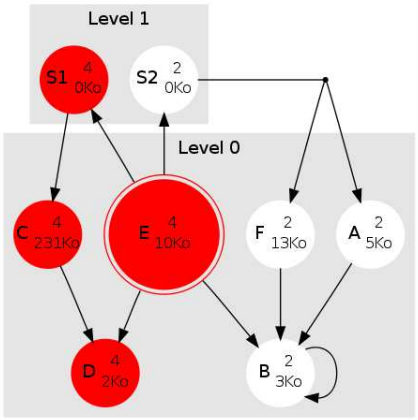

FIG. 4.21 – Variation des priorités

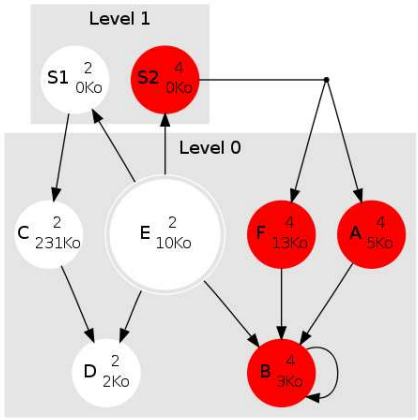

FIG. 4.22 – Variation des priorités

Dans la figure 4.22 nous avons évalué le comportement de la décision d'AxSeL4OSGi si l'utilisateur décidait de mettre en priorité le service de navigation plutôt que celui de serveur de documents PDF. Notons que dans les deux cas de figure seuls les composants et services prioritaires ont été colorés en rouge et chargés.

Conclusion Dans cette section, nous avons démontré la faisabilité de l'adaptation dynamique fournie par AxSeL4OSGi. Ensuite, nous avons énuméré et illustré les différents cas de variation du processus de décision en fonction de la nature des événements collectés à travers les sondes contextuelles. Ainsi, nous avons mis en évidence les comportements de notre intergiciel en fonction des évènements provenant du dépôt de services composants, du dispositif et enfin de l'utilisateur. Dans la section suivante, nous évaluons les performances de notre intergiciel lors du passage à l'échelle.

4.3 Évaluation des performances d'AxSeL4OSGi

Dans cette section, nous évaluons les performances de notre intergiciel de deux manières. D'abord, nous observons ses performances avec un graphe d'application standard, et ce en comparaison avec la plate-forme OBR (*BundleRepository*) [98] sur les différentes phases d'extraction, de décision et d'adaptation. Ensuite, nous évaluons le comportement des deux architectures lors du passage à l'échelle en variant le nombre d'exécution et le nombre des noeuds

dans le graphe. La plate-forme OBR puise les bundles à installer localement sur une machine dans un dépôt distant. Cependant, elle opère uniquement au niveau composant et non service, n'applique pas de décision respectant des contraintes contextuelles matérielles mais essaie de charger tous les bundles, et ne permet pas d'adaptation.

AxSeL4OSGi fournit une fonctionnalité de contextualisation du déploiement qui implique un coût à supporter. Ce coût est évalué en termes d'utilisation de la mémoire du dispositif électronique et de temps d'attente d'exécution correspondant à l'attente de l'utilisateur avant d'avoir une réponse de l'intergiciel.

4.3.1 Performances avec un graphe de dépendances standard

Génération des descripteurs de dépôts de simulation L'évaluation des performances d'AxSeL4OSGi ont été réalisées en se basant sur des descripteurs de dépôts générés automatiquement. Après observation de plusieurs applications OSGi, nous avons établi un modèle de graphe qui représentent les propriétés de ces applications. Ces dernières sont composées de *bundles* qui exportent des services et dépendent d'autres *bundles*, et ne possèdent pas de boucles sauf si le renseignement des dépendances est erroné. Le service de génération de graphe fourni prend en paramètre le nombre maximal de bundles et de services souhaités, et de dépendances entre eux. La génération aboutit à un fichier XML ayant le même modèle que le Repository OSGi. Dans le listing ci-dessous 4.6 nous présentons des extraits du pseudo code de la génération.

```
1  public ResourceGraphGenerator(int bundleNumMax, int serviceNumMaxBundle, int
       bundleCRNumMaxBundle, int serviceCRNumMaxBundle, String filename) throws
       IOException {
2  //le nombre maximal de bundles
3      this.m_bundleNumMax = bundleNumMax;
4  //le nombre maximal de services
5      this.m_serviceNumMaxBundle = serviceNumMaxBundle;
6  //le nombre maximal de dépendances par bundle
7      this.m_bundleCRNumMaxBundle = bundleCRNumMaxBundle;
8  //le nombre maximal de dépendances par service
9      this.m_serviceCRNumMaxBundle = serviceCRNumMaxBundle;
10 //le fichier XML de sortie
11     this.m_filename = filename;
12     new SimpleDateFormat("yyyyMMddhhmmss.SSS");
13     this.generate();
14   }
```

Listing 4.6 – Pseudo Code de la fonction de génération du Repository OSGi

En observant les applications standards fournies dans le projet Felix, nous avons réalisé les tests avec des graphes représentant une application moyenne de vingt noeuds services/composants, chaque noeud composant ayant entre un et cinq dépendances et trois dépendances pour les noeuds services. Bien qu'AxSeL4OSGi supporte les graphes cycliques (la détection n'entrave pas son fonctionnement et ses performances), nous avons testé avec des graphes sans boucles comme OBR ne les supporte pas. Il est à noter que l'objectif de la génération est d'obtenir une description d'une application orientée service composant, nous ne nous sommes pas intéressés aux aspects de propriétés intrinsèques du graphe.

Performances temps/mémoire et comparaison avec OBR Nous comparons le temps d'exécution d'AxSeL4OSGi et d'OBR, lors du déroulement des différentes phases.

Performances temps Lors de l'extraction (figure 4.23), notre intergiciel construit le graphe en moyenne deux fois plus rapidement qu'OBR. La prise de décision, pour un même coloriage total du graphe, s'effectue en moyenne dix fois plus rapidement. Notons également

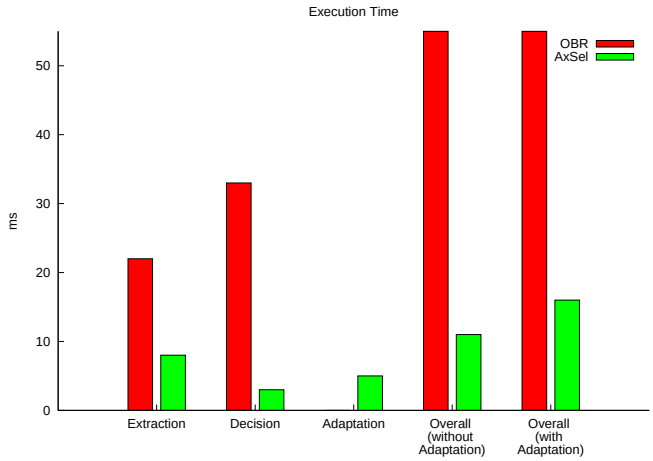

FIG. 4.23 – Évolution de la performance temps en fonction des phases

qu'AxSeL4OSGi réalise une phase d'adaptation absente dans l'architecture OBR, et qui reste très performante. Il est a noter que l'architecture OBR est destinée à la gestion d'une fédération de dépôts et à la résolution des dépendances des ressources à déployer. Cette architecture n'est pas pensée pour appliquer une quelconque heuristique de chargement, toutefois elle permet le tri des ressources selon des filtres les renseignant.

Performances temps La figure 4.24 illustre l'allocation de la mémoire. AxSeL4OSGi exploite plus de ressources mémoire à l'extraction du graphe (sur-coût du graphe bidimensionnel), mais est cependant, deux fois moins gourmand en mémoire sur l'ensemble des phases restantes sans l'adaptation, et une fois et demi avec l'adaptation.

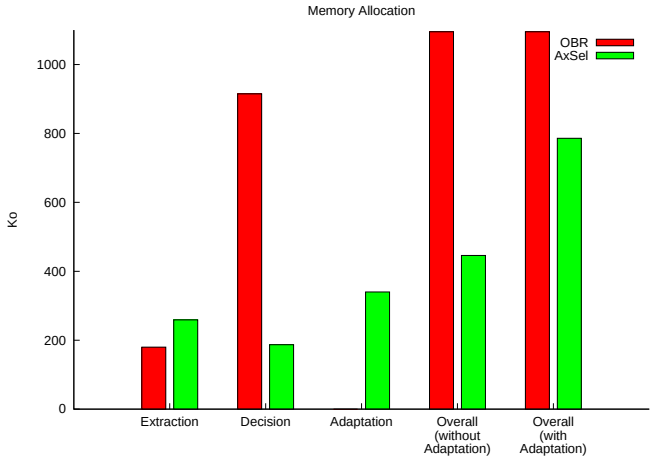

FIG. 4.24 – Évolution de la performance mémoire en fonction des phases

4.3.2 Performances lors du passage à l'échelle

A présent, nous soumettons notre intergiciel à des tests de variation des nombres d'exécution et du nombre de noeuds dans le graphe. Ces tests de performances tendent à observer le comportement d'AxSeL4OSGi lors du passage à l'échelle en comparaison avec le *Bundle Repository*.

Variation des résultats en fonction du nombre d'exécution Nous évaluons les performances temps d'exécution et mémoire des deux plates-formes en fonction du nombre d'exécution des algorithmes.

Performances en temps d'exécution Les courbes de la figure 4.25 montrent une perte de performance au premier lancement de tous les algorithmes. Cette perte s'explique par le coût utilisé par le chargeur de classes Java. Le sur-coût est le même pour les deux plates-formes qui ont ensuite une performance constante.

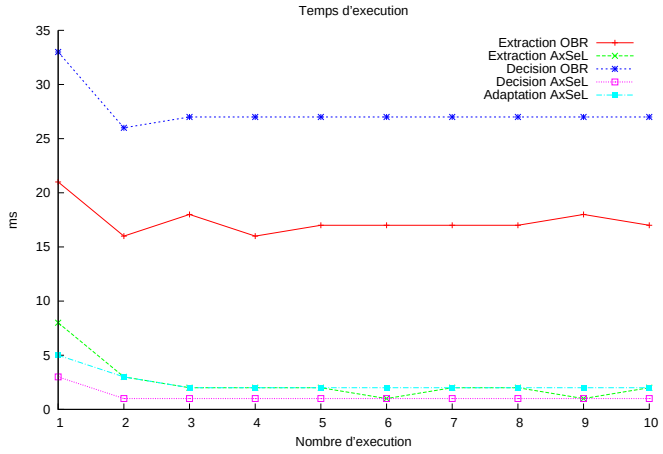

FIG. 4.25 – Évolution de la performance temps en fonction du nombre d'exécution

Performances mémoire L'utilisation de la mémoire a été également évaluée en fonction du nombre d'exécution des algorithmes. Les courbes de la figure 4.26 montrent l'évolution de cette ressource en fonction de la phase observée. OBR utilise une mémoire constante à la phase

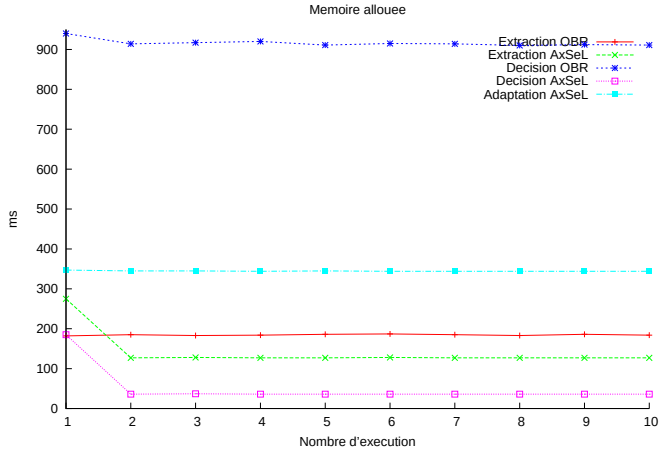

FIG. 4.26 – Évolution de la performance mémoire en fonction du nombre d'exécution

d'extraction et de décision. AxSeL4OSGi réduit par contre considérablement l'utilisation de la mémoire dès la deuxième utilisation pour les phases d'extraction et de décision. Cette réduction s'explique par l'utilisation du design pattern Singleton pour ne pas réallouer des objets déjà créés. La phase d'adaptation est constante en allocation mémoire, et est cependant testée dans le pire des cas (recoloriage total), et utilise moins de mémoire pour un recoloriage partiel.

Variation des résultats en fonction du nombre de noeuds du graphe Nous avons fait varier le nombre de noeuds des graphes testés et observé l'utilisation de la mémoire et le temps d'exécution des deux plates-formes.

Performances en temps d'exécution Les courbes de la figure 4.27 montrent que la phase d'extraction d'AxSeL4OSGi est en moyenne deux fois plus rapide que celle d'OBR selon un comportement linéaire. La phase de décision d'OBR suit un comportement linéaire mais celle d'AxSeL4OSGi est plus proche d'un temps constant considérablement plus performant. Enfin, seul AxSeL4OSGi réalise l'adaptation qui observe un temps polynomial.

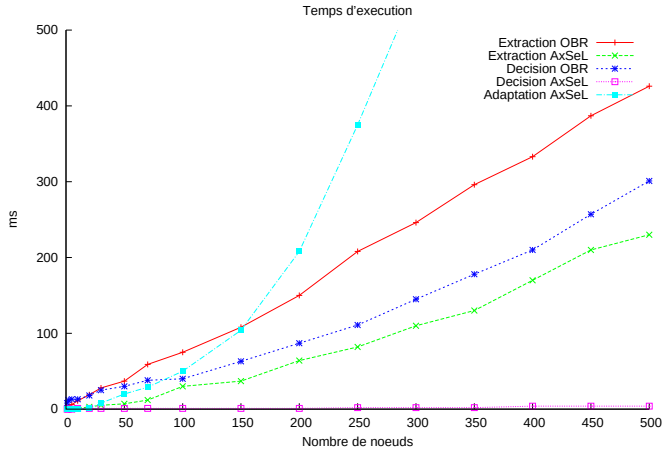

FIG. 4.27 – Évaluation de la performance temps en fonction du nombre de noeuds du graphe

Performances mémoire Les courbes de la figure 4.28 illustrent l'évaluation de l'utilisation mémoire selon la variation du nombre de noeuds. Pendant l'extraction OBR et AxSeL4OSGi suivent un comportement asymptotique où AxSeL4OSGi alloue en moyenne deux fois moins de mémoire. L'allocation mémoire observée dans la figure 4.24 correspond donc à un comportement inversé à l'origine. A la phase de décision, OBR et AxSeL4OSGi ont une allocation mémoire constante. Enfin, la courbe observée à la phase d'adaptation a un comportement oscillatoire, en raison d'un parcours total du graphe lors de l'adaptation. Ces valeurs dépendent fortement de la structure du graphe et peuvent être améliorées ultérieurement avec des changements dans la conception de notre système (recherche des noeuds parents plus performante).

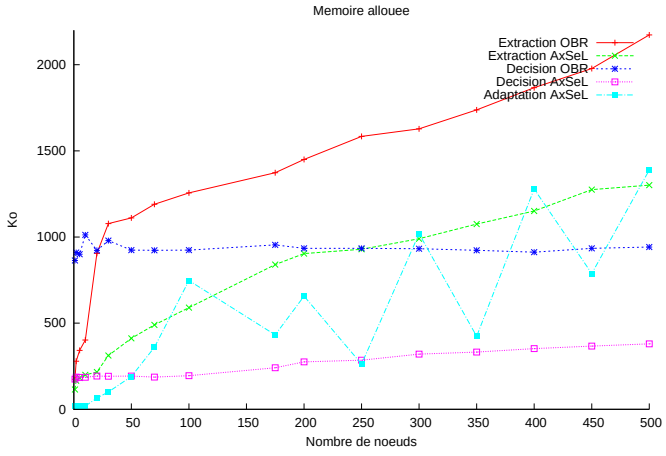

FIG. 4.28 – Évaluation de la performance mémoire en fonction du nombre de noeuds du graphe

Dans la section 3.5.4 du chapitre 3, nous avons évalué d'une manière théorique les différentes complexités des algorithmes réalisant l'extraction des dépendances, la prise de décision contextuelle, et l'adaptation dynamique. La complexité de la phase d'extraction des dépendances a été évaluée à $O(n^5)$, et celle de la phase de décision à $O(n^2)$. Cependant, les courbes correspondant

à l'utilisation temps et mémoire dans les deux figures précédentes observent un comportement presque linéaire sur la plage de tests de 0 à 500 noeuds du graphe. Un comportement polynomial adéquat à la complexité théorique sera probablement observé si nous élargissons encore la plage de tests. Quant à la phase d'adaptation, elle possède des courbes plus prononcées qui correspondent bien au comportement polynomial prévu de $O(n^3)$. Pour prouver cela, sur la base des données de tests recueillies nous avons procédé à des dérivées successives de la courbe jusqu'à son annulation pour déterminer l'ordre de complexité. L'ordre est retrouvé lorsque la dérivée s'annule : il s'agit du nombre de dérivation jusqu'à l'annulation de la courbe auquel nous retranchons un. La dérivée correspondant à l'adaptation contextuelle s'est annulée à la quatrième dérivation, nous pouvons alors en déduire une complexité de l'ordre 3, ce qui confirme la valeur déterminée théoriquement.

Coût de l'adaptation lors du passage à l'échelle Nous observons les performances temps et mémoire relatives à l'exécution d'AxSeL4OSGi avec et sans adaptation. Nous distinguons entre les phases car l'adaptation est une opération ponctuelle qui n'est pas constamment supportée et l'intégrer au sein de notre intergiciel a un coût supplémentaire.

Performances temps La courbe 4.29 illustre le coût en temps observé avec et sans adaptation. La courbe montre que notre prototype est plus lent avec l'adaptation que sans. Cela dit, le processus d'adaptation est un processus ponctuel qui n'est pas réalisée constamment. Ce qui signifie que la plate-forme n'aura à supporter ces coûts que d'une manière passagère. Le temps total d'exécution ne dépasse jamais les 500 ms dans le pire des cas (graphe de 500

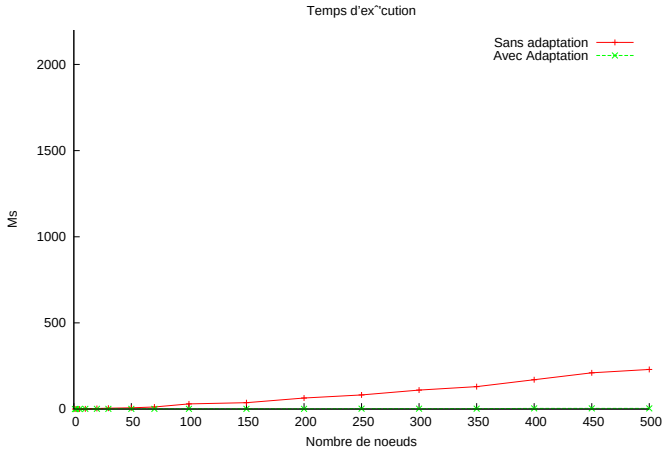

FIG. 4.29 – évaluation du coût du temps d'exécution

noeuds). Ce qui ne pénalise pas l'utilisateur avec un temps d'attente très long.

Performances mémoire Nous observons ces coûts pour évaluer le passage à l'échelle d'AxSeL4OSGi. Les tests ont été réalisés en variant le nombre de nœuds des graphes. La courbe 4.30 représente le coût en termes de mémoire en fonction du nombre de nœuds. La courbe d'usage mémoire sans adaptation a un comportement linéaire allant de 400Ko de RAM

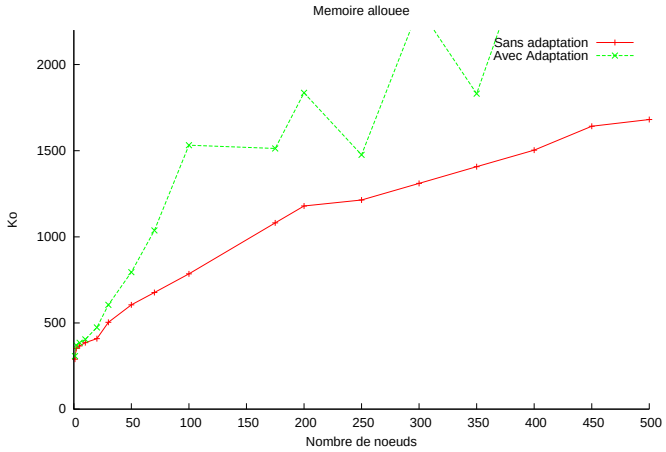

FIG. 4.30 – évaluation de l'utilisation mémoire

et n'excédant pas 1500Ko dans le pire des cas (graphe à 500 noeuds). Quant à la courbe d'adaptation nous notons des oscillations correspondant au processus de recherche des dépendances pour chaque nœud. Les simulations ont été réalisées en considérant le pire des cas qui est la recoloration totale du graphe. Dans des cas plus communs nous ne procédons pas à la recoloration du graphe en entier mais à une partie uniquement.

Si nous nous plaçons dans un cas contraint, par exemple un téléphone mobile Nokia 6630, après usage intensif par chargement des applications de calendriers, de la liste des contacts, de l'appareil photo et caméra et d'un navigateur web il lui reste 2000Ko de mémoire RAM libre. En sachant que le nombre de noeuds moyen d'une application usuelle ne dépasse pas une centaine de noeud, nous observons qu'à cette valeur l'usage mémoire de notre intergiciel n'atteint jamais la valeur maximale de 2000Ko avec et sans adaptation.

4.4 Conclusion

Dans ce chapitre, nous avons présenté l'état actuel de notre prototype. AxSeL4OSGi est un projet privé gforge inria. Nous l'avons réalisé et l'avons soumis à un ensemble de tests, et comparé à une plate-forme similaire : OBR. AxSeL4OSGi offre de bonnes performances en terme de temps d'exécution et de mémoire allouée. Nous avons réalisé l'évaluation des performances temps et mémoire, quant aux étapes d'extraction de graphes de dépendances, de prise de décision des services à charger et enfin d'adaptation dynamique. Cependant, plusieurs améliorations sont possibles au niveau de chacune des étapes réalisées. Notre intergiciel repose sur la théorie des graphes afin de représenter les services/composants à charger. Pour l'instant, la concordance entre les dépendances entre services et composants est réalisée par une adéquation exacte entre leurs noms d'interface, il est possible d'améliorer cette étape en intégrant une adéquation sémantique plus élaborée basée sur la description des services et des composants. Cela s'apparente aux travaux de composition de services présentés dans les travaux de [82]. Ensuite, l'étape de décision est faite par coloration du graphe de dépendances en prenant en compte les contraintes des services, dispositifs et préférences utilisateurs. Cette étape peut être

améliorée pour intégrer d'autres stratégies de décision basées sur des contraintes hiérarchisées par exemple tels que dans les travaux de [46]. Enfin, seuls les noeuds rouges sont chargés localement sur la plate-forme, les noeuds blancs restent en attente de chargement lors des prochains parcours. A ce niveau, nous comptons remplacer les noeuds blancs par des noeuds vides et effectuer une recherche dans l'environnement en vue de trouver un service distant offrant la même fonctionnalité.

Chapitre 5

Conclusions et travaux futurs

Dans ce chapitre, nous récapitulons les apports de cette thèse qui s'articulent autour de trois contributions : une vue globale et flexible pour le déploiement contextuel, une gestion dynamique du contexte et enfin des heuristiques pour réaliser le déploiement dirigé par les contraintes, nous y consacrons la section 5.1. Ensuite, dans la section 5.2 nous esquissons les perspectives à court et à long terme de notre travail.

5.1 Contributions

Les environnements intelligents promettent de fournir des fonctionnalités variées disséminées dans les objets du quotidien. Le déploiement de celles-ci d'une manière personnalisée et adaptée au contexte est un défi inhérent à ces environnements. En effet, les terminaux électroniques utilisés ont des capacités matérielles hétérogènes auxquelles il faut s'adapter. D'autre part, il est essentiel de fournir à chacun des utilisateurs des fonctionnalités personnalisées et adaptées à ses attentes d'une manière non intrusive. Enfin, dans un contexte en perpétuel changement il est important d'avoir une vue actualisée et non obsolète des fonctionnalités fournies.

Dans cette thèse nous avons investigué le développement de nouvelles abstractions et de nouveaux mécanismes pour le déploiement contextuel d'applications. Nous nous sommes principalement intéressés à l'adaptation structurelle de l'application et à l'étude de ses dépendances. Ainsi, nous avons attiré l'attention sur la nécessité de fournir une représentation contextuelle

et flexible qui intègre plusieurs aspects inhérents aux environnements intelligents et permet l'évolution de l'application. Conjointement à cela il est nécessaire de fournir des mécanismes de raisonnement dirigés par les contraintes collectées dynamiquement. Pour réaliser cela nous adoptons une approche située à la frontière des trois mondes d'architectures orientées services, d'architectures orientées composants et d'environnements pervasifs. Nous proposons une solution intergicielle qui absorbe les mécanismes de représentation et de déploiement contextuel. Les fonctionnalités sont conçues comme des applications de services composants à déployer. Le déploiement contextuel d'applications orientées services passe par une étape de représentation globale de l'application, ensuite par des mécanismes heuristiques pour réaliser le déploiement dirigé par les contraintes d'une manière autonome. Les informations pertinentes au déploiement sont fréquemment remontées à l'exécution et pris en compte.

Sur la base de l'étude d'approches similaires de déploiement contextuel d'application. Nous proposons notre approche AxSeL qui est basée sur trois contributions : une vue globale orienté graphe pour représenter les applications orientées services composants, une couche contextuelle de gestion dynamique du contexte et un ensemble d'heuristiques qui réalisent le déploiement sur la base du modèle d'application et du modèle de contexte recueilli. Nous avons également valider notre approche et évalué ses performances pour le déploiement contextuel d'application par l'implantation et le test du prototype AxSeL4OSGi.

5.1.1 Une vue globale et flexible pour le déploiement contextuel

Nous avons proposé une vue globale indépendante de la technologie pour représenter les applications orientées services composants. Cette vue repose sur une intégration simultanée de plusieurs aspects inhérents aux environnements intelligents tels que les aspects multi-fournisseur et contextuels du déploiement et de l'exécution. Nous modélisons les dépendances entre services et composants dans un graphe orienté bidimensionnel où les noeuds représentent les services et les composants et les arcs leurs dépendances. Pour fournir une vue contextuelle les informations sont puisées à partir des métadonnées des services et des composants et représentées dans les noeuds et arcs du graphe. Les environnements d'exécution et de déploiement y sont modélisés grâce à des niveaux distincts dans le graphe, cette distinction donne la possibilité d'affecter à chacun des niveaux les propriétés qui lui sont relatives. La multiplicité des fournisseurs permet d'élargir les possibilités de sélection des services et des composants, elle est renseignée par des opérateurs logiques entre les dépendances et introduits dans le graphe. Nous modélisons également les modifications de la structure d'application générées par les fluctuations de l'environnement à travers la flexibilité du graphe. Ainsi, nous lui insufflons la dynamique et l'extensibilité à travers des fonctions qui permettent la modification, l'ajout et le retrait des noeuds et des arcs. L'ensemble de ces paradigmes nous a permis de définir une vue globale contextuelle et flexible en adéquation avec les défis des environnements intelligents et permettant l'adaptation structurelle. Cette vue est actualisée selon les modifications de l'environnement immédiat.

Nous avons formalisé les services, les composants et leur dépendances et proposé un modèle orienté objet pour le graphe de l'application et ses mécanismes de manipulation. Nous avons également implémenté ce modèle et l'avons fait correspondre au modèle OSGi. Enfin, nous avons prouvé sa faisabilité et son adéquation aux besoins des environnements intelligents.

5.1.2 Une gestion dynamique du contexte

Afin de fournir une approche non intrusive qui ne nécessite pas l'intervention constante de l'utilisateur, nous avons délégué les opérations de capture et de représentation du contexte à la couche contextuelle proposée. Celle-ci modélise un contexte pertinent au déploiement et constitué par l'utilisateur, le terminal électronique et le dépôt de services et de composants à partir duquel il sont puisés. Les données considérées intègrent la mémoire virtuelle disponible sur le dispositif, les préférences en services de l'utilisateur, et les méta données renseignées dans les dépôts. Le contexte fourni est extensible car il est important de pouvoir intégrer d'autres sources de données estimées pertinentes pour le déploiement. Par ailleurs, grâce à des mécanismes d'écoute du contexte que nous déployons sur les sources de données, nous capturons les événements recueillis et adaptons l'application à déployer selon les nouvelles contraintes. Nous préconisons des API spécifiques chargées d'introspecter les différentes sources de données et de générer les événements de notification. La considération de l'aspect dynamique du contexte nous permet d'actualiser fréquemment l'application déployée en vue de fournir une meilleure réponse à l'utilisateur ou une meilleure optimisation des ressources matérielles.

Nous avons conçu le contexte à travers un modèle orienté objet et extensible. Les mécanismes de représentation et de gestion du contexte sont fournis dans un service d'AxSeL pouvant interagir avec les autres services de l'intergiciel. Nous avons également fourni une implémentation de cette couche contextuelle et prouvé sa faisabilité.

5.1.3 Des heuristiques pour le déploiement autonome et contextuel d'applications

Notre intergiciel allège la charge de l'utilisateur et de l'application de la capture du contexte et de la gestion du déploiement en absorbant les opérations de résolution de dépendances et de raisonnement contextuel. AxSeL fournit un support automatique pour la résolution des dépendances grâce à leur extraction à partir des dépôts de services et de composants et à leur modélisation dans un graphe contextuel bidimensionnel. Le graphe est ensuite déployé grâce à un algorithme de décision qui le parcourt et évalue la faisabilité du déploiement compte tenu des contraintes matérielles et des préférences utilisateurs. Le parcours du graphe réalise un marquage par coloration des noeuds installables en attribuant la couleur rouge lorsque le noeud respecte les contraintes et la couleur blanche sinon. Le déploiement se fait d'une manière progressive, ainsi lorsque les services et les composants n'ont pas été chargés lors d'un premier passage de l'algorithme ils sont mis en priorité pour qu'ils soient déployés lors d'un prochain passage. La décision du déploiement obéit à des stratégies multi-critère et extensibles capables de considérer à la fois plusieurs paramètres contextuels. L'intégration des opérateurs logiques dénotant de la multiplicité des fournisseurs augmentent la complexité de la tâche en augmentant les possibilités de sélection des services et composants à déployer. Nous prenons en compte cette contrainte et appliquons des traitements du graphe optimisés pour préserver les ressources matérielles. D'autre part, nous fournissons des mécanismes d'adaptation à l'exécution sur la base de l'écoute des événements générés par le contexte. Ainsi, lorsque des éléments sont modifiés, ajoutés ou retirés du dépôt des services et des composants ces événements sont dynamiquement répercutés

sur la structure de l'application et pris en compte dans le processus de déploiement en intégrant les nouvelles mises à jour. Il en est de même lorsque le terminal ne possède pas assez de ressources matérielles disponibles le déploiement est réévalué avec les nouvelles contraintes. Enfin lorsque l'utilisateur change ses préférences des services AxSeL intègre ces modifications et attribue une autre vue contextuelle à l'application.

Nous avons conçu les mécanismes de déploiement sous la forme d'une heuristique de parcours et d'évaluation du graphe de dépendances. Nous avons également fourni des implémentations pour chacun de ces graphes et les avons encapsulés dans des services interagissant pour fournir le déploiement contextuel. Enfin, nous avons évalué les performances de ces heuristiques et prouvé leur efficacité dans un contexte intelligent.

5.2 Perspectives

Dans cette section, nous présentons les perspectives de notre travail. Nous distinguons entre perspectives à court terme propres à AxSeL et d'autres à long terme en relation avec le domaine de déploiement contextuel.

5.2.1 Amélioration de l'architecture AxSeL

Notre intergiciel peut être amélioré sur plusieurs points, ci-après nous listons les points qu'il serait intéressant d'étudier :

- *Ontologies contextuelles.* Nous avons fourni un modèle générique et simple pour éviter un coût important de capture et de traitement du contexte. Cet aspect d'AxSeL pourrait être amélioré pour intégrer un modèle de contexte plus expressif mais toujours adéquat avec les contraintes matérielles des dispositifs mobiles. Les approches ontologiques bien que gourmandes en ressources matérielles sont de plus en plus tolérées.
- *Aspect multi-fournisseur et équivalence de services.* Actuellement AxSeL fournit un graphe réalisé à partir de l'agrégation de plusieurs dépôts de services et composants provenant de divers fournisseurs. L'évaluation des services équivalents est réalisée à travers une correspondance simple entre les noms des services. Nous pouvons adopter une politique d'agrégation basée sur les descriptions sémantiques des services pour mettre en avant les qualités de services et fournir une solution plus adaptée.
- *Applications multi-entrée*, le graphe de dépendances proposé dans AxSeL supporte l'aspect multi-entrée et ce grâce à la possibilité d'avoir plusieurs points d'entrée dans un même graphe applicatif. Il serait intéressant de concevoir et de tester de telles d'applications et de préconiser par exemple pour chaque entrée une stratégie de déploiement différente et un comportement différent.
- *Déploiement multi-stratégie.* AxSeL propose un graphe bidimensionnel réunissant les aspects d'exécution et de déploiement dans une unique vue. Actuellement, la décision du chargement est réalisée à travers l'application d'une stratégie globale sur l'ensemble des services et des composants de l'application. Une approche intéressante peut être d'appliquer une stratégie par niveau dans le graphe. Ainsi, il sera possible d'appliquer une stratégie orienté déploiement et basée sur ses critères sur le niveau des composants, et

d'une manière analogique une stratégie orientée exécution et qualité de services sur le niveau des services.

- *Auto-contextualisation d'AxSeL*, de par sa conception orientée services, nous pouvons AxSeL peut s'auto appliquer la contextualisation. Ainsi, en environnement contraint il peut charger uniquement les services nécessaires à son fonctionnement. Cependant, cela implique une assignation de priorités aux différents services d'AxSeL pour qu'il ne se décharge pas ses services noyaux en cas de limite de ressources matérielles.
- *Cohérence de l'application.* Nous réalisons une adaptation structurelle de l'application selon des propriétés non fonctionnelles de ses services et de ses composants. Cependant, le choix d'un service par rapport à un autre peut entraîner une incohérence dans le fonctionnement global de l'application. Il est intéressant d'étudier les aspects au niveau des services et de proposer une formalisation comportementale permettant de vérifier la cohérence de l'application après contextualisation.

5.2.2 Une adaptation intergicielle horizontale et verticale

Dans cette thèse, nous avons abordé les aspects de contextualisation structurelle des applications orientées services. Cette adaptation peut être considéré comme horizontale positionnée dans le niveau intergiciel. Il est possible d'extrapoler ce raisonnement sur les couches en dessous et en dessus de l'intergiciel pour réaliser une contextualisation horizontale.

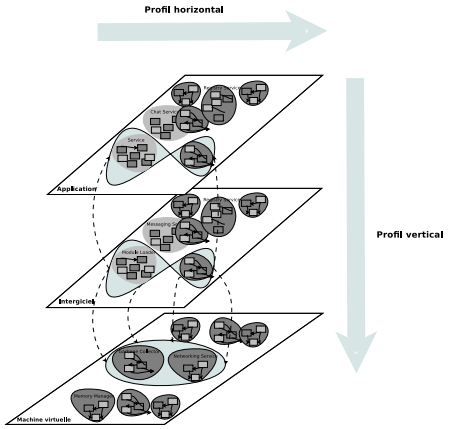

FIG. 5.1 – Adaptation verticale et horizontale

Si les couches considérées sont pensées en fonctionnalités et avec des structures modulaires et flexibles il est possible de réaliser une contextualisation verticale transversale à ces couches. Cela évite de supporter le coût du stockage et de la gestion de la totalité de l'intergiciel sur un dispositif contraint et permet une personnalisation à tous les niveaux. La figure 5.1 illustre cet aspect en considérant les couches de la machine virtuelle, de l'intergiciel d'exécution des composants et des services et de l'application.

L'adaptation au niveau intergiciel consisterait à choisir uniquement les composants et services intrinsèques nécessaires. Par exemple, si l'utilisateur estime qu'il n'a pas besoin du service de messagerie ou de l'annuaire de services, il peut ne pas le charger sur sa machine. Au niveau

de la machine virtuelle il sera possible de choisir d'utiliser uniquement les services qui nous intéressent tels que le garbarge collector ou le service de connexion réseau.

5.2.3 Agrégation intelligente des dépôts

Dans les environnements intelligents les périphériques et les objets du quotidien constituent un *Internet of Things* (IoT). Chacun des éléments de cet IoT peut jouer le rôle de fournisseur de services et de composants. Actuellement, AxSeL puise ses dépendances d'applications dans les dépôts découverts en réalisant une agrégation simple de leurs contenus. Il est intéressant d'étudier les aspects d'une agrégation intelligente inter-dépôts et qui serait guidée par leur temps de disponibilité ou par des critères de confiance.

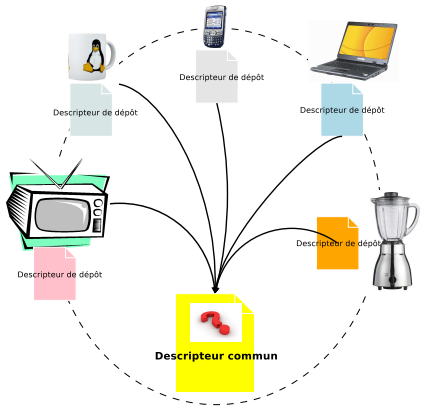

FIG. 5.2 – Agrégation intelligente dans un *Internet of Things*

La figure 5.2 illustre la possible agrégation des services et composants fournis par un IoT où tous les périphériques et tous les objets du quotidien peuvent renfermer des fonctionnalités. Cependant, il est à noter que la prolifération des dispositifs électroniques est en croissance exponentielle continue sur les prochaines années. Le traitement d'une agrégation dans un tel environnement peut avoir un coût croissant et inadapté aux dispositifs mobiles, pour cela il est essentiel d'étudier les aspects de passage à l'échelle dans ces environnements.

Publications

Journaux

Amira Ben Hamida, F. Le Mouël, S. Frénot, and M. Ben Ahmed *Une approche pour un chargement contextuel de services dans les environnements pervasifs*, In Networking and Information Systems / Ingénierie des Systèmes d'Information (ISI), 13(3) :59 ?82, June 2008. Special edition.

Conférences internationales

Stéphane Frénot, Noha Ibrahim, Frédéric Le Mouël, Amira Ben Hamida, Julien Ponge, Mathieu Chantrel, Denis Beras *ROCS : a Remotely Provisioned OSGi Framework for Ambient Devices*, In the 12th IEEE/IFIP Operations and Management Symposium (NOMS), april 2010, Osaka, Japan.

Amira Ben Hamida, F. Le Mouël, S. Frénot, and M. Ben Ahmed *A Graph-based Approach for Contextual Service Loading in Pervasive Environments*, In Proceedings of the 10th International Symposium on Distributed Objects and Applications (DOA'2008), Lecture Notes in Computer Science, vol. , pp. -, Springer Verlag, Monterrey, Mexico, November 2008.

Amira Ben Hamida, F. Le Mouël, S. Frénot, and M. Ben Ahmed *Contextual Service Loading by Dependency Graph Colouring*, In Proceedings of the 8th International Conference on New Technologies in Distributed Systems (NOTERE'2008), pp. 182 ?187, Lyon, France, June 2008.

Amira Ben Hamida and F. Le Mouël *Resurrection : A Platform for Spontaneously Generating and Managing Proximity Documents*, In Proceedings of the IEEE International Conference on Pervasive Services (ICPS'2006), Lyon, France, June 2006.

Conférences et ateliers nationaux

Amira Ben Hamida and F. Le Mouël *Middleware minimal et auto-extensible pour le déploiement de services en environnement pervasif*, In Actes des 3ème Journées Francophones de la Mobilité et Ubiquité (UbiMob'2006), CNAM, Paris, France, September 2006.

Amira Ben Hamida, F. Le Mouël, S. Frénot, and M. Ben Ahmed *Approche pour un chargement contextuel de services sur des dispositifs contraints*, In Actes du 6ème atelier sur les Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information (OCM-SI'2007) organisé conjointement avec INFORSID'2007, Perros-Guirec, France, May 2007.

Bibliographie

[1] Smart Install Maker. http ://www.sminstall.com/.

[2] *The Jini Specification, 2nd Edition*. Arnold, Ken, 2000.

[3] Oscar Bundle Repository. http ://oscar-osgi.sourceforge.net/, 2005.

[4] Deployment and Configuration of Component-based Distributed Applications v4.0. Technical report, Object Management Group, 2006.

[5] Rfc-0112 bundle repository, confidential, draft. Technical report, OSGi Alliance, 2006.

[6] OSGi Service Platform Service Compendium Release 4, Version 4.1. Technical report, OSGi Alliance, 2007.

[7] Sca service component architecture assembly model specification v 1.0.0. Technical report, OSOA, 2007.

[8] .net framework developer center, November 2008.

[9] Actual Installer. http ://www.actualinstaller.com/, 2009.

[10] Advanced Installer. http ://www.advancedinstaller.com/, 2009.

[11] Android. http ://www.android.com/, 2009.

[12] Bindex. http ://www.osgi.org/Repository/BIndex, 2009.

[13] CORBA/e Resource Page. http ://www.omg.org/corba-e/index.htm, 2009.

[14] Graphviz. http ://www.graphviz.org/, 2009.

[15] Knopflerfish Bundle Repository. http ://www.knopflerfish.org/repo/repository.xml, 2009.

[16] kXML 2. http ://kxml.sourceforge.net/, 2009.

[17] Mac os x mobile. http ://www.apple.com/fr/macosx/, 2009.

[18] Paremus Bundle Repository. http ://sigil.codecauldron.org/spring-external.obr, 2009.

[19] Spring Dynamic Modules for OSGi(tm) Service Platforms, 2009.

[20] SwiXML 1.5. http ://www.swixml.org/, 2009.

[21] The DOT Language. http ://www.graphviz.org/doc/info/lang.html, 2009.

[22] The Fractal Project. http ://fractal.ow2.org/, 2009.

[23] Windows Mobile. http ://www.microsoft.com/windowsmobile/fr-fr/default.mspx, 2009.

[24] XML Pull Parsing. http ://www.xmlpull.org/, 2009.

[25] IsPack package once. Deploy everywhere. http ://izpack.org/, 2010.

[26] E. Aarts and J. L. Encarnaço. *Into Ambient Intelligence*. True Visions The Emergence of Ambient Intelligence, 2006.

[27] E. Aarts, R. Harwig, and M. Schuurmans. *Ambient intelligence*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[28] G.D. Abowd, A. Dey, R. Orr, and J. Brotherton. Context-awareness in wearable and ubiquitous computing. In *Journal of Virtual Reality*, pages 179–180. Springer-Verlag, 1997.

[29] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC '99 : Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[30] A. Agostini, C. Bettini, and D. Riboni. Hybrid reasoning in the care middleware for context awareness. *International journal of Web engineering and technology*, 5(1) :3–23, 2009.

[31] N. Aguirre and T. Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. *International Conference on Automated Software Engineering*, 0 :271, 2002.

[32] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services : Concepts, Architecture and Applications*. Springer Verlag, 2004.

[33] T.S.C. Alvin and S-N. Chuang. MobiPADS : A Reflective Middleware for Context-Aware Mobile Computing. *IEEE Transactions on Software Engineering*, 29(12) :1072–1085, 2003.

[34] E. Austvold and K. Carter. Service-oriented architectures : survey findings on deployment and plans for future. Technical report, AMR Research Market Analytics, 2005.

[35] M. Autili, M. Caporuscio, and V. Issarny. A Reference Model for Service Oriented Middleware. Research Report inria-00326479, version 1, ARLES - INRIA Rocquencourt - INRIA, 2008.

[36] D. Ayed, C. Taconet, G. Bernard, and Y. Berbers. Cadecomp : Context-aware deployment of component-based applications. *Journal of Network and Computer Applications*, 31(3) :224 – 257, 2008.

[37] M. Baldauf and S. Dustdar. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2004.

[38] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, June 2009.

[39] G. Bieber and J. Carpenter. Introduction to service-oriented programming (rev 2.1). Technical report, OpenWings Whitepaper, 2001.

[40] G. Bieber and J. Carpenter. *A service-oriented Component Architecture for self-Forming, Self-Healing, Network-Centric systems*. OpenWings, 2001.

[41] G. Blair, T. Coupaye, and J. B. Stefani. Component-based architecture : the Fractal initiative. *Annals of Telecommunications*, 64(1) :1–4, February 2009.

[42] B. Blunden. *Virtual Machine Design and Implementation in C/C++ with Cdrom*. Wordware Publishing Inc., Plano, TX, USA, 2002.

[43] C. Bolchini, C. Curino, F. A. Schreiber, and L. Tanca. Context Integration for Mobile Data Tailoring. *IEEE International Conference on Mobile Data Management*, 0 :5, 2006.

[44] C. Bolchini, C. A. Curino, E. Quintarelli, F.A. Schreiber, and L. Tanca. A data-oriented survey of context models. *SIGMOD Rec.*, 36(4) :19–26, December 2007.

[45] G. Booch, I. Jacobson, and J. Rumbaugh. *UML 2.0, Guide de référence*. Pearson Campuspress, 2004.

[46] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *LISP and Symbolic Computation*, 5(3) :223–270, 1992.

[47] M. Bouzeghoub, G. Gardarin, and P. Valduriez. *"Les Objets"*. Eyrolles, 1997.

[48] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *2nd Workshop on Self-Healing Systems*, 2004.

[49] P. J. Brown, J. D. Bovey, and Xian Chen. Context-aware applications : from the laboratory to the marketplace. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 4(5) :58–64, 1997.

[50] L. Capra, W. Emmerich, and C. Mascolo. Carisma : context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10) :929–945, 2003.

[51] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A.V.D. Hoek, and A. L. Wolf. "a characterization framework for software deployment technologies". Technical report, University of Colorado Department of Computer Science, 1998.

[52] A. Casimiro, J. Kaiser, P. Veríssimo, P. Veríssimo, U. Lisboa, K. Cheverst, K. Cheverst, V. Cahill, V. Cahill, A. Friday, and A. Friday. Cortex : Towards supporting autonomous and cooperating sentient entities, 2002.

[53] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dartmouth College, Hanover, NH, USA, 2000.

[54] S-N. Chuang. Mobipads : A reflective middleware for context-aware mobile computing. *IEEE Trans. Softw. Eng.*, 29(12) :1072–1085, 2003. Member-Chan, Alvin T. S.

[55] P. C. Clements. A survey of architecture description languages. In *Software Specification and Design, 1996., Proceedings of the 8th International Workshop on*, pages 16–25, 1996.

[56] J. Coutaz, J.L. Crowley, S. Dobson, and D. Garlan. Context is Key. *Commun. ACM Volume 48*, 48, 2005.

[57] Brad J Cox. *Object oriented programming : an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[58] B. de Ruyter. 365 days ambient intelligence in homelab. Technical report, Philips Research, 2003.

[59] A. Dearle, G. Kirby, and A. McCarthy. A Framework for Constraint-based Deployment and Autonomic Management of Distributed Applications. In *International Conference on Autonomic Computing*, 2004.

[60] H.S. Delugach, L.C. Cox, and D.J. Skipper. Dependency Language Representation Using Conceptual Graphs. Autonomic Information Systems. Technical Report A405993, BEVILACQUA RESEARCH CORP HUNTSVILLE AL, 2001.

[61] A. K. Dey and G. D. Abowd. Towards a Better Understanding of Context and Context-Awareness. *CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness*, 2000.

[62] A.K. Dey, D. Salber, and G.D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. In *HCI conference*, 2001.

[63] Anind K. Dey. Context-aware computing : The cyberdesk project. In *AAAI 1998 Spring Symposium on Intelligent Environments*, pages 51–54, Palo Alto, 1998. AAAI Press.

[64] A. Diaconescu, J. Bourcier, and C. Escoffier. Autonomic ipojo : Towards self-managing middleware for ubiquitous systems. In *IEEE International Conference on Wireless and Mobile Computing, Networking and Communication*, pages 472–477. IEEE Computer Society, 2008.

[65] P. Dourish. *Where the action is : the foundations of embodied interaction.* MIT Press, Cambridge, MA, USA, 2001.

[66] George Edwards, Chiyoung Seo, Daniel Popescu, Sam Malek, and Nenad Medvidovic. Self-* software architectures and component middleware in pervasive environments. In *MPAC '07 : Proceedings of the 5th international workshop on Middleware for pervasive and ad-hoc computing*, pages 25–30, New York, NY, USA, 2007. ACM.

[67] W. Emmerich. Software engineering and middleware : A roadmap. *In the Future of software Engineering - International Conference on Software Engineering*, 22 :117–129, May 2000.

[68] C. Escoffier, R. S. Hall, and P. Lalanda. ipojo : an extensible service-oriented component framework. In *IEEE International Conference on Services Computing*, pages 474–481. IEEE Computer Society, 2007.

[69] Felix. The Apache Felix Project. In *http ://cwiki.apache.org/FELIX/index.html*, 2008.

[70] A. R. Frei. *Jadabs - An adaptive pervasive middleware architecture.* PhD thesis, Swiss Federal Institute of Technology Zurich, 2009.

[71] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Boston, MA, January 1995.

[72] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura : Towards distraction-free pervasive computing. *IEEE Pervasive Computing, special issue on "Integrated Pervasive Computing Environments"*, 21(2) :22–31, 2002.

[73] P. Grace, G. S. Blair, and S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1) :2–14, 2005.

[74] T. Gu, H. Pung, and D. Zhang. A Middleware for Building Context-Aware Mobile Services. In *IEEE VT conference*, 2004.

[75] Eric Gunnerson. *A Programmer's Introduction to C Sharp (Second Edition)*. APRESS, Bonn, 2002.

[76] R. S. Hall and H. Cervantes. Automating Service Dependency Management in a Service-Oriented Component Model. In *Sixth Component-Based Software Engineering Workshop*, 2003.

[77] George T. Heineman and William T. Council. *Component-Based Software Engineering, Putting the Pieces Together*. Addison Weysley, 2001.

[78] K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling context information in pervasive computing systems. In *Proceedings of the First International Conference on Pervasive Computing*, pages 167–180, London, UK, 2002. Springer-Verlag.

[79] D. Hoareau and Y. Mahéo. Middleware support for the deployment of ubiquitous software components. *Personal Ubiquitous Comput.*, 12(2) :167–178, 2008.

[80] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, and J. Altmann. Context-awareness on mobile devices- the hydrogen approach. In *36th Hawaii International Conference on System Sciences*, 2002.

[81] Michael N. Huhns and Munindar P. Singh. Service-oriented computing : Key concepts and principles. *IEEE Internet Computing*, 9(1) :75–81, 2005.

[82] N. Ibrahim. *Spontaneous Integration of Services for Pervasive environments*. PhD thesis, Insa de Lyon, September 2008.

[83] David Franklin Joshua, David Franklin, and Joshua Flachsbart. All gadget and no representation makes jack a dull environment. In *In AAAI Spring Symposium on Intelligent Environments. AAAI TR*, pages 155–160, 1998.

[84] A. Keller, E. Keller, U. Blumenthal, and G. Kar. Classification and computation of dependencies for distributed management. In *Proceedings of the Fifth International Conference on Computers and Communications (ISCC)*, 2000.

[85] T. Kichkaylo and V. Karamcheti. Optimal Resource-Aware Deployment Planning for Component-based Distributed Applications. In *13th IEEE International Symposium on High Performance Distributed Computing*, 2004.

[86] K. Kui and Z. Wang. Software component models. *IEEE Transactions on Software Engineering conference*, 2007.

[87] M. MacKenzie, K. Laskey, F. McCabe, P. Brown, and R. Metz. Reference model for service oriented architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, February 2006.

[88] D. Mcilroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[89] P. Merle. OpenCCM : The Open CORBA Components Platform, 2003.

[90] D. Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions Software Engineering*, 24(7) :521–533, 1998.

[91] Daniel Le Metayer. Software architecture styles as graph grammars. In *In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23. ACM Press, 1996.

[92] Bertrand Meyer. The grand challenge of trusted components. *Software Engineering, International Conference on Software Engineering*, 0 :660, 2003.

[93] Microsoft Corporation. "Understanding UPnP : A white paper". Technical report, UPnP Forum, 2000.

[94] Microsoft TechNet. Fonctionnement du gestionnaire de packages.

[95] Microsoft TechNet. What Is a Cabinet (.cab) File ?

[96] M. Miraoui, C. Tadj, and C. ben Amar. Modeling and simulation of a multiagent service oriented architecture for pervasive computing systems. In *Proceedings of the 2009 Workshop on Middleware for Ubiquitous and Pervasive Systems*, pages 1–6, New York, NY, USA, 2009. ACM.

[97] J. M. Myerson. *The Complete Book of Middleware*. AUERBACH Publications, 2002.

[98] OBR. Obr Bundle Repository. In *http ://www.osgi.org/Repository/HomePage*, 2008.

[99] M. Offermans. Automatically managing service dependencies in OSGi. Technical report, Luminis, 2005.

[100] OSGi Alliance. OSGi-The Dynamic Module System for Java. In *http ://www.osgi.org/*, 2009.

[101] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing : State of the art and research challenges. *Computer*, 40(11) :38–45, 2007.

[102] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann, and Bernd J. Krämer. 05462 service-oriented computing : A research roadmap. In Francisco Cubera, Bernd J. Krämer, and Michael P. Papazoglou, editors, *Service Oriented Computing (SOC)*, number 05462 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[103] N. Park, K.W. Lee, and H. Kim. A Middleware for Supporting Context-Aware Services in Mobile and Ubiquitous Environment. In *IEEE ICMB conference*, 2005.

[104] D.E. Petrelli, E. Not, O. Stock, and M. Zancanaro. Modeling context is like taking pictures, 2000.

[105] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup : architecture for component trading and dynamic updating. In *Fourth International Conference on Configurable Distributed Systems, 1998. Proceedings*, pages 43–51, 1998.

[106] V. Poladian, J. Sousa, D. Garlan, and M. Shaw. Dynamic configuration of resource-aware services. In *26th ICSE conference*, 2004.

[107] S. Prashant. *JavaBeans : developer's resource*. Prentice Hall, 1997.

[108] Davy Preuveneers and Yolande Berbers. Towards context-aware and resource-driven self-adaption for mobile handheld applications. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1165–1170, New York, NY, USA, March 2007. ACM Press.

[109] Davy Preuveneers and Yolande Berbers. Encoding Semantic Awareness in Resource-Constrained Devices. *IEEE Intelligent Systems*, 23(2) :26–33, 2008.

[110] Quest Software Smart System Management. Service Level Management.

[111] C. E. Cuesta Quintero, P. de la Fuente, and M. Barrio-Solórzano. Dynamic coordination architecture through the use of reflection. In *SAC*, pages 134–140, 2001.

[112] K. Raatikainen. Functionality needed in middleware for future mobile computing platforms. In *IFIP/ACM Middleware Conference*. Advanced topic workshop : Middleware for mobile computing, Heidelberg, Germany, november 2001.

[113] A. Ranganathan, Al J. Muhtadi, S. Chetan, R.H.Campbell, and M.D. Mickunas. Middlewhere : a middleware for location awareness in ubiquitous computing applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (Middleware '04)*, pages 397–416. Springer-Verlag, 2004.

[114] Red Hat The Gnome Project. Vala - Compiler for the GObject type system. http ://-live.gnome.org/Vala, 2007.

[115] T. Rodden, K. Chervest, N. Davies, and A. Dix. Exploiting Context in HCI Design for Mobile Systems. In *in Workshop on Human Computer Interaction with Mobile Devices*, 1998.

[116] G. Rossi, S. Gordillo, and F. Lyardet. Design patterns for context-aware adaptation. *Applications and the Internet Workshops, IEEE/IPSJ International Symposium on*, 0 :170–173, 2005.

[117] M. Satyanarayanan. Pervasive Computing : Vision and Challenges. *IEEE Personal Communication*, August 2001.

[118] Mahadev Satyanarayanan. Fundamental challenges in mobile computing. *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1996.

[119] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *In Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, 1994.

[120] B. Schilit and M. Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8 :22–32, 1994.

[121] Maria A. Strimpakou, Ioanna G. Roussaki, and Miltiades E. Anagnostou. A context ontology for pervasive service provision. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 775–779, Washington, DC, USA, 2006. IEEE Computer Society.

[122] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[123] C. Szyperski. Component technology : what, where, and how ? In *Proceedings of the 25th International Conference on Software Engineering*, pages 684–693, Washington, DC, USA, 2003. IEEE Computer Society.

[124] C. Taconet, E. Putycz, and G. Bernard. Context-Aware Deployment for Mobile Users. In *27th IEEE International Computer Software and Applications Conference*, 2003.

[125] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic Change Management by Distributed Graph Transformation : Towards Configurable Distributed Systems. In *TAGT*, pages 179–193, 1998.

[126] The Apache Maven Project. Apache Maven. http ://maven.apache.org/, 2009.

[127] The Eclipse Foundation. Eclipse. http ://www.eclipse.org/, 2009.

[128] W3C. Install Shield. http ://www.acresso.com/products/is/installshield-overview.htm, 2009.

[129] J.-B. Waldner. *Nano-informatique et intelligence ambiante*. Hermès - Lavoisier, 2007.

[130] R. Want, A. Hopper, ao V. Falc and J. Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1) :91–102, 1992.

[131] A. Ward and A. Jones. A new location technique for the active office, 1997.

[132] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, september 1991.

[133] M. Wermelinger. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings - Software*, 145(5) :130–136, 1998.

[134] E. Wolfgang and K. Nima. Component technologies : Java beans, com, corba, rmi, ejb and the corba component model. In *Proceedings of the 24th International Conference on Software Engineering*, pages 691–692, New York, NY, USA, 2002. ACM Press.

[135] S. J. H. Yang, A. F. M. Huang, R. Chen, S-S. Tseng, and Y-S. Shen. Context Model and Context Acquisition for Ubiquitous Content Access in ULearning Environments. In *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, 2006.

[136] U. Zdun, C. Hentrich, and W M. V.D. Aalst. A survey of patterns for service oriented architectures. *Int. J. Internet Protoc. Technol.*, 1(3) :132–143, 2006.