COSMOS : composition de nœuds de contexte

Denis Conan* — **Romain Rouvoy**** — **Lionel Seinturier*****

* *Institut TELECOM, SudParis, CNRS Samovar, 9 rue Charles Fourier, F-91011 Évry*
Denis.Conan@int-evry.fr
** *Université d'Oslo, département d'informatique, P.O.Box 1080 Blindern, N-0316 Oslo*
rouvoy@ifi.uio.no
*** *INRIA-Futurs, Projet Adam, LIFL — Université des Sciences et Technologies de Lille*
Cité Scientifique, F-59655 Villeneuve d'Ascq cedex
Lionel.Seinturier@inria.fr

RÉSUMÉ. *Les applications ubiquitaires évoluent dans une grande diversité de contextes d'utilisation. Or, cette diversité requiert une adaptation continuelle afin de préserver le bon fonctionnement des applications. Dès lors, l'observation du contexte joue un rôle prépondérant. Si les approches actuelles « centrée utilisateur » et « système » ont prouvé leur pertinence dans ce domaine, elles souffrent néanmoins de certaines limitations liées à l'expressivité des compositions des inférences et au passage à l'échelle. Par conséquent, nous proposons de réorganiser les fonctionnalités traditionnelles d'un gestionnaire de contexte de manière systématique en cycles « collecte / interprétation / identification de situations ». Cette approche repose sur la définition du concept de nœuds de contexte composés dans un graphe (hiérarchie avec partage), et l'expression du concept en composant et architecture logicielle pour faciliter la définition et la gestion des politiques de gestion de contexte.*

ABSTRACT. *Ubiquitous applications are facing a large diversity of execution contexts. However, this diversity requires some continual adaptation to preserve the correct execution of these applications. Consequently, the observation of the context plays an important role. Even if "user-centered" and "system" approaches have proven their relevance in this domain, they suffer from some limitations when expressing the composition of inferences and when considering the scalability issue. Therefore, we propose to reorganise the classical functionalities of a context manager to systematically introduce cycles "collect / interpretation / identification of situations". This approach is based on the definition of the concept of context nodes composed in a graph (hierarchy with sharing), and the mapping of the concept into components and software architecture for easing the definition and the management of context management policies.*

MOTS-CLÉS : *Gestion de contexte, architecture, composants, resources système.*

KEYWORDS: *Context management, architecture, components, system resources.*

RSTI - TSI. Volume 27 - n° 9-10/2008, pages 1189 à 1224

1. Introduction

Les environnements ubiquitaires se caractérisent par une diversité de terminaux mobiles, de réseaux sans fil et de modes d'utilisation. Les applications réparties évoluant dans de tels environnements doivent continuellement gérer le contexte dans lequel elles s'exécutent afin de détecter les situations d'adaptation (Coutaz *et al.*, 2005). Le contexte est constitué de différentes catégories d'entités observables (ressources logicielles telles que les ressources système et les préférences des utilisateurs, et ressources matérielles telles que les capteurs), des rôles de ces entités dans le contexte et des relations entre ces entités. À titre d'exemple, la carte réseau WiFi d'un terminal mobile est une entité observable jouant le rôle de dispositif d'entrée pour la détection de situations de déconnexion. La gestion de toutes les informations qui décrivent le contexte d'exécution des applications est centralisée dans un gestionnaire de contexte. Sur un terminal, son rôle est de construire une vue cohérente et complète de l'environnement d'exécution et de proposer aux applications des mécanismes de description et d'identification de situations d'adaptation.

Historiquement, deux approches distinctes existent dans la littérature pour la gestion de contexte : une approche « centrée utilisateur » et une approche de type supervision « système ». Les travaux présentés dans cet article visent à concilier les deux approches et à appliquer l'orientation composant à la gestion de contexte.

Dans la première approche, « centrée utilisateur », le contexte inclut le terminal de l'utilisateur, celui des petits matériels accessibles à proximité, y compris des capteurs, et celui des autres utilisateurs ou terminaux atteignables par un réseau non personnel. Les travaux existants dans la littérature décomposent la gestion de contexte en quatre fonctionnalités : la collecte, l'interprétation, la détection de situations et l'utilisation pour l'adaptation (Yau *et al.*, 2002; Dey *et al.*, 2001; Coutaz *et al.*, 2005). L'élément discriminant des solutions existantes, c'est-à-dire des architectures effectivement réalisées sous la forme de canevas logiciels, est la qualité des informations abstraites obtenues par inférence et devant caractériser les situations de l'utilisateur. Les limites des solutions existantes sont *(1)* le manque de composition aisée des informations de contexte, *(2)* le passage à l'échelle, tant en termes de quantité d'informations de contexte que de nombre d'applications clientes, et *(3)* le manque de réutilisabilité comme cela a été identifié par (Mostefaoui *et al.*, 2004) et (Henricksen *et al.*, 2005).

La seconde approche, la supervision, et plus particulièrement l'observation des ressources système, est une problématique ancienne (Mansouri-Samani *et al.*, 1992; Schroeder, 1995) qui voit un regain d'intérêt avec l'émergence des grappes et des grilles de machines (Boutros Saab *et al.*, 2000; Cecchet *et al.*, 2005) et avec l'arrivée de l'informatique ubiquitaire (Rey *et al.*, 2004; David *et al.*, 2005). Les résultats de la littérature sont l'instrumentation des systèmes d'exploitation et la collecte des informations des machines distantes. Dans cet article, nous nous limitons à la collecte des informations locales sur les ressources des terminaux. De ce point de vue, l'élément le plus discriminant des solutions existantes est la performance des observations de-

vant inhiber le moins possible le fonctionnement du système (Schroeder, 1995)[1]. Une faiblesse importante de cette approche est le faible niveau d'abstraction des données collectées, qui restent souvent numériques et sont peu interprétées pour devenir symboliques et utilisables par l'application, voire l'utilisateur, pour faire de l'adaptation en situation de mobilité.

Dans les environnements ubiquitaires, nous ciblons les applications sensibles au contexte, c'est-à-dire dont l'exécution dépend du contexte. Des exemples de telles applications sont les guides touristiques avec navigation contextuelle, les applications d'annotations contextuelles ou encore les applications avec enrichissement contextuel et réalité augmentée telles que les jeux multijoueurs. Pour ces applications, la gestion de contexte doit être collaborative (Satyanarayanan, 2004). Les applications ne doivent pas gérer elles-mêmes leur contexte, sous peine d'incohérence et de mauvaise résolution des conflits entre elles. La gestion de contexte ne peut pas être transparente aux applications, au risque d'être contre-productive pour certaines d'entre elles en ne prenant pas en compte leur sémantique. Nous appliquons l'orientation composant pour la construction du gestionnaire de contexte. Les informations de contexte sont encapsulées dans des composants et peuvent alors être composées dans une architecture décrivant les politiques de gestion de contexte. Nous bénéficions donc des outils de spécification et de conception des architectures logicielles à base de composants.

Ainsi, un gestionnaire de contexte doit être *(1)* centré application/utilisateur pour fournir des informations de contexte aisément interprétables, *(2)* construit à partir d'éléments composés plutôt que programmés pour faciliter le développement, et *(3)* performant en contrôlant et minimisant l'utilisation des ressources pour son propre fonctionnement. Nous proposons un canevas logiciel pour la composition d'informations de contexte, appelé COSMOS pour *COntext entitieS coMpositiOn and Sharing*. L'originalité de COSMOS est l'expression de la composition de contexte dans un langage de définition d'architecture logicielle et la projection de cette architecture sur un graphe de composants. Ainsi, COSMOS facilite la conception, la composition, l'adaptation et la réutilisation des politiques de gestion de contexte. Pour démontrer l'intérêt de l'approche COSMOS, nous l'appliquons à la composition d'informations de contexte de ressources système (processeur, mémoire, réseau, etc.). COSMOS a pour objectif de montrer que la complexité inhérente au développement d'applications sensibles au contexte nécessite l'utilisation de techniques d'ingénierie logicielle avancées comme les composants et les architectures logicielles. Ce travail relève donc aussi bien des communautés académiques travaillant sur les applications contextuelles que des communautés des architectures logicielles et des composants.

La suite de l'article est organisée comme suit. La section 2 motive la conception d'un canevas logiciel dédié à la composition d'informations de contexte. Les sections 3 et 4 présentent la conception du canevas logiciel COSMOS, tout d'abord le concept de nœud de contexte, et ensuite les patrons de conception pour construire l'ar-

1. Voir pour cela par exemple les travaux pour les systèmes d'exploitation GNU/Linux (Smith *et al.*, 2002) et Microsoft Windows NT/2000 (Knop *et al.*, 2002).

chitecture. La section 5 donne quelques éléments d'implantation. La section 6 présente une évaluation du prototype pour la composition d'informations sur les ressources système. La section 7 positionne la proposition par rapport aux travaux de la littérature. Enfin, la section 8 conclut en résumant la contribution et identifie des perspectives à ce travail.

2. Motivations et objectifs de COSMOS

Dans cette section, nous commençons par proposer une architecture de gestion de contexte (cf. section 2.1). Ensuite, nous motivons le besoin de composition d'informations de contexte (cf. section 2.2). Nous donnons les objectifs du canevas logiciel COSMOS (cf. section 2.3) et nous proposons un scénario ubiquitaire (cf. section 2.4) que nous exploitons par la suite pour illustrer les concepts de COSMOS.

2.1. *Architecture de gestion de contexte*

L'architecture globale du gestionnaire de contexte présentée dans la figure 1 est inspirée de (Coutaz *et al.*, 2005; Dey *et al.*, 2001; Yau *et al.*, 2002). La première couche de l'architecture est constituée de la collecte des informations brutes sur les ressources système, les capteurs à proximité et directement accessibles à partir du terminal, les préférences données par l'utilisateur, et les informations de contexte en provenance des autres terminaux mobiles. Ces canevas logiciels fournissent les données de contexte à la base du traitement par ce que nous appelons un *processeur de contexte*. La sensibilité au contexte de l'application, quant à elle, est gérée par les *conteneurs* des composants applicatifs. Dans la boîte la plus haute, la figure 1 représente un des composants de l'application entouré de son conteneur sensible au contexte. En termes de fonctionnalités, la réification des ressources système permet de collecter des données brutes, souvent numériques, comme la qualité du lien réseau. Le processeur de contexte en déduit des informations de contexte de plus haut niveau, souvent des données symboliques comme le mode de connectivité (connecté, partiellement connecté ou déconnecté) et permet d'identifier des situations comme la perte prochaine de la connectivité réseau. Les politiques d'adaptation, quant à elles, sont généralement gérées en collaboration avec l'application au niveau du conteneur : par exemple, l'exploitation consiste à avertir l'utilisateur du mode de connectivité à l'aide d'un icone dédié.

Nous expliquons dans la suite de cette section pourquoi nous modifions l'architecture de gestion de contexte avec comme principe d'introduire des cycles indépendants « collecte/interprétation/identification de situations » dans chacune des couches, voire des boîtes, de l'architecture (cf. partie droite de la figure 1). De tels cycles complets sont présents dans les collecteurs de contexte, plusieurs autres dans le processeur de contexte, et de même pour le conteneur sensible au contexte.

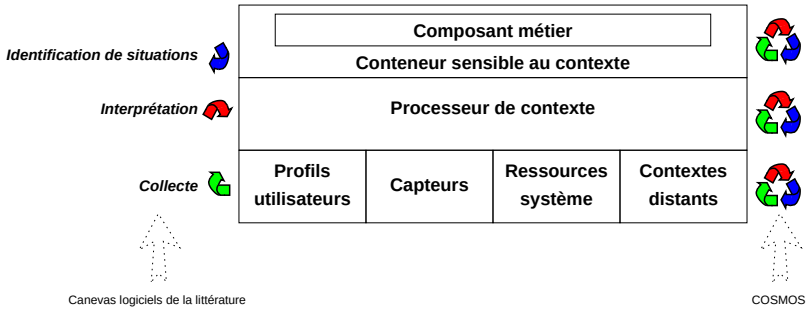

Figure 1. *Architecture de gestion de contexte*

Le principe de séparation des préoccupations implique que les fonctionnalités (collecte, interprétation, identification et adaptation) soient réalisées dans des « couches » différentes. Seulement, remarquons que ces couches logicielles, et même chacune des boîtes, correspondent aussi à des métiers différents, chacun ayant ses besoins particuliers. La collecte des données en provenance des capteurs met en œuvre des technologies de réseaux spontanés (aussi appelés *ad hoc*) telles que le protocole de découverte `UPnP`. La collecte d'informations de contextes distantes est une problématique des systèmes répartis avec par exemple des protocoles de cohérence de données ou de détection de défaillances. Pour la collecte des informations sur les ressources système, nous avons vu que c'est un sujet à part entière du domaine des systèmes d'exploitation. En ce qui concerne le processeur de contexte, le domaine de l'intelligence artificielle peut proposer des moteurs d'inférence complexes. Enfin, l'adaptation en liaison avec le conteneur sensible au contexte peut mettre en œuvre des approches très diverses telles que les approches à base de règles, à base d'optimisation de ressources, de processus biologiques (système nerveux, comportement d'espèces vivant en colonies).

Par ailleurs, nous ajoutons les constats suivants. Les cycles de vie des éléments physiques ou logiques observables sont très différents. En effet, la collecte des données brutes (découverte, filtrage, gestion de l'historique, précision, etc.) dépend du type de la source d'informations de contexte. En outre, les ontologies des collecteurs de contexte sont diverses et variées. Les transformations d'ontologies doivent être effectuées lorsqu'elles sont nécessaires afin de ne pas engorger le processeur de contexte avec ces transformations. Par ailleurs, les ressources système affectées à la collecte et aux manipulations des données brutes doivent être finement contrôlées. À titre d'exemple, dans le cas de nombreuses applications sensibles au contexte s'exécutant sur le même terminal, les activités créées pour la collecte des données brutes peuvent inhiber et rendre inutilisable le terminal de l'utilisateur si leur nombre est trop grand. La même remarque s'applique pour d'autres ressources système comme l'espace mémoire, par exemple, pour gérer les historiques des informations de contexte.

Une autre raison pour l'isolation des collecteurs de contexte du processeur de contexte est d'éviter, pour des raisons de performances, que les requêtes (synchrones) des applications se traduisent par la collecte (elle-aussi synchrone) des informations brutes avec appels (eux aussi synchrones) au système d'exploitation ou sur le réseau. En outre, les canevas logiciels de collecte des données brutes sont potentiellement nombreux et de styles architecturaux différents (par exemple, selon (Coutaz *et al.*, 2005), orientés processus ou orientés donnée).

En conclusion, nous ajoutons dans chaque couche ou boîte une notion de temps dans l'identification de situations, et par implication, une notion d'instant de collecte, avec dans l'intervalle de temps, les traitements d'inférence et d'identification de situations. Une autre conséquence est que les différentes couches sont indépendantes quant à la gestion des ressources système (mémoire, activités) qu'elles consomment pour leurs traitements. Nous obtenons ainsi un système faiblement couplé et facilement reconfigurable par les concepteurs de gestionnaires de contexte et d'applications sensibles au contexte.

2.2. *Architecture à base de composants*

Lors de la conception de COSMOS, nous appliquons les principes de base de la construction d'intergiciels : le canevas logiciel est construit à partir d'éléments génériques, spécialisables et modulaires afin de composer plutôt que de programmer. Nous avons choisi pour cela une approche à base de composants.

Cette approche apporte une vision unifiée dans laquelle les mêmes concepts (composant, liaison, interface) sont utilisés pour développer les applications et les différentes couches intergicielles et système sous-jacentes. Cette vision unifiée en facilite également la conception et l'évolution. Elle autorise en outre une vision hiérarchique dans laquelle l'ensemble « canevas et application » peut être vu à différents niveaux de granularité. Par ailleurs, la notion d'architecture logicielle associée à l'approche orientée composant permet d'exprimer la composition des entités logicielles indépendamment de leurs implantations, rendant ainsi plus aisée la compréhension de l'ensemble. La notion d'architecture logicielle favorise aussi la dynamicité en autorisant la redéfinition des liaisons de tout ou partie du canevas, voire de l'application à l'exécution. La reconfiguration et l'adaptation à des contextes nouveaux non prévus au départ en sont facilitées.

2.3. *COSMOS*

COSMOS est une proposition pour la composition d'informations de contexte intégrant la collecte, l'interprétation et l'identification de situations. COSMOS peut être utilisé dans toutes les couches d'une architecture de gestion de contexte. Les objectifs de COSMOS sont les suivants :

– composer des informations de contexte de manière déclarative, à opposer aux approches « par programmation » utilisées dans les canevas logiciels existants. Ceci doit faciliter la conception par composition, l'adaptation et la réutilisation des politiques de composition de contexte ;

– isoler chaque couche de l'architecture de gestion de contexte des autres couches afin de promouvoir la séparation des préoccupations et des cycles de vie des informations de contexte ;

– fournir les concepts « système » pour gérer finement les ressources consommées par les différents traitements d'inférence.

2.4. *Scénario cache/déchargement*

Afin d'illustrer, dans la section suivante, les concepts introduits par COSMOS nous nous basons sur le scénario suivant.

Nous supposons que l'utilisateur d'un terminal mobile exécute une application répartie lors d'un déplacement. D'une part, la connexion au réseau WiFi est sujette à des déconnexions. Pour tolérer ces déconnexions, l'intergiciel ou le système d'exploitation peuvent installer sur le terminal mobile et maintenir dans un cache logiciel certaines parties de l'application nécessaires au fonctionnement pendant les déconnexions. Un autre choix peut être d'exporter des traitements sur des nœuds du réseau fixe et de récupérer les résultats à la reconnexion. D'autre part, même lorsque la connectivité est bonne, il peut être intéressant d'exporter des traitements afin de disposer des ressources d'une machine du réseau fixe plus puissante. Pour choisir entre l'importation ou l'exportation, le système calcule la capacité mémoire comme la somme de la mémoire vive disponible et du *swap* disponible. Le système surveille aussi la connexion au réseau WiFi avec la qualité du lien réseau pour détecter les transitions de modes de connectivité (déconnecté, faiblement et fortement connecté) (Temal *et al.*, 2004). En détectant les déconnexions, le système établit le débit maximum possible pendant les périodes de bonne connectivité que nous appelons « débit ajusté ». Lorsque la mémoire est suffisante, mais le débit faible, la mise en cache est préférée. Lorsque la mémoire est insuffisante, mais le débit suffisant, l'exportation des traitements est préférée. Dans les deux autres cas, l'utilisateur ou le système donnent leur préférence (mise en cache ou exportation). Ensuite, la décision de mise en cache ou d'exportation des traitements est stabilisée par temporisation. Une fois que la décision est prise, les informations de connectivité sont utilisées pour décider de l'instant de mise en cache ou d'exportation des traitements lors des transitions de modes de connectivité (partiellement déconnecté vers déconnecté, partiellement déconnecté vers connecté).

3. Nœud de contexte

Dans cette section, nous présentons la composition d'informations de contexte avec COSMOS. Tout d'abord, la section 3.1 introduit le concept de nœud de contexte.

Puis, la section 3.2 en définit les propriétés de paramétrage. Ensuite, la section 3.3 développe l'architecture générique d'un nœud de contexte. Enfin, la section 3.4 présente l'implantation du scénario précédent à l'aide du concept de nœuds de contexte.

3.1. *Concept de nœud de contexte*

Le concept de base de COSMOS est le *nœud de contexte*. Un nœud de contexte est une information de contexte modélisée par un composant. Les nœuds de contexte sont organisés en une hiérarchie avec possibilité de partage. Tous les composants de la hiérarchie sont potentiellement accessibles par les clients de COSMOS. Ce sont tous des composants.

Les relations entre les nœuds sont donc des relations d'encapsulation. Le graphe représente l'ensemble des politiques de gestion de contexte utilisées par les applications clientes du gestionnaire de contexte. Le partage de nœuds de contexte correspond à la possibilité de partage ou d'utilisation d'une partie d'une politique de gestion de contexte par plusieurs politiques. Les nœuds de contexte feuilles de la hiérarchie encapsulent les informations de contexte élémentaires, par exemple les ressources système du terminal (mémoire vive, qualité du lien WiFi, etc.). Leur rôle est d'isoler les inférences de contexte de plus haut niveau, qui deviennent donc indépendantes du canevas logiciel utilisé pour la collecte des données brutes.

3.2. *Propriétés d'un nœud de contexte*

Passif ou actif

Chaque nœud peut être passif ou actif avec exécution périodique de tâches dans des activités. Un nœud passif est utilisé par des activités extérieures au nœud qui l'interrogent pour obtenir une information. Un nœud actif peut *a contrario* initier un parcours du graphe. Le cas d'utilisation le plus fréquent des nœuds actifs est la centralisation de plusieurs types d'informations et la mise à disposition de ces informations afin d'isoler une partie du graphe d'accès multiples trop fréquents.

Observation ou notification

Les rapports d'observation contenant les informations de contexte circulent du bas vers le haut de la hiérarchie dans des messages (dont les constituants élémentaires sont typés). Lorsque la circulation s'effectue à la demande d'un nœud parent ou d'un client, c'est une observation ; dans le cas contraire, c'est une notification.

Passant ou bloquant

Lors d'une observation ou d'une notification, le composant qui traite la requête peut être passant ou bloquant. Lors d'une observation, un nœud de contexte passant demande d'abord un nouveau rapport d'observation à ses enfants, puis calcule un rap-

port d'observation pour le transmettre (en retour) vers le haut de la hiérarchie. Lors d'une notification, un nœud de contexte passant calcule un nouveau rapport d'observation avec la nouvelle notification, puis passe vers le haut ce rapport en notifiant ses parents. Dans le cas bloquant, le nœud observé fournit l'information de contexte qu'il détient sans observer les nœuds enfants, et le nœud notifié modifie son état interne sans notifier les nœuds parents. Par ailleurs, les observations et notifications peuvent être uniques pour les informations de contexte ne changeant pas pendant l'exécution. Enfin, lorsque les informations de contexte des nœuds enfants ne peuvent pas être collectées, par exemple parce que la ressource système « interface réseau WiFi » n'existe pas, une exception est remontée vers les nœuds parents. La remontée de cette exception peut bien sûr être bloquée par un nœud de contexte donné pour masquer l'indisponibilité de l'information de contexte aux nœuds parents.

Opérateur

Un nœud de contexte récupère des informations de contexte de nœuds enfants de la hiérarchie et infère une information de plus haut niveau d'abstraction. Ce traitement est ce que nous appelons un opérateur. COSMOS propose des opérateurs génériques classés selon la typologie de (Rey *et al.*, 2004) : opérateurs élémentaires pour la collecte, opérateurs à mémoire comme le calcul de la moyenne, de traduction de format, de fusion de données avec différentes qualités, d'abstraction ou d'inférence comme « l'additionneur », et opérateurs à seuil comme un détecteur de connectivité (Temal *et al.*, 2004) ou un évaluateur de profil « énergie, durée prévisible de connexion, espace mémoire » (Boulkenafed *et al.*, 2003). Il est à noter que, dans une architecture de gestion de contexte classique, les premiers opérateurs élémentaires pour la collecte feraient partie de la couche « collecte » et la plupart des autres opérateurs de la couche « interprétation », tandis que les derniers opérateurs à seuil seraient dans la couche « identification de situations ». Dans COSMOS, ils peuvent être utilisés dans toutes les couches. La thèse défendu dans COSMOS est qu'il est possible de concevoir des politiques de gestion de contexte complexes en composant des opérateurs simples, tout en n'empêchant pas la conception d'opérateurs plus complexes. Nous verrons un exemple avec une vingtaine de nœuds, donc une vingtaine d'opérateurs, un peu plus loin dans la section 3.4.

Cycle de vie et gestion des ressources

Tous les nœuds de contexte de la hiérarchie sont gérés finement, tant au niveau de leur cycle de vie qu'au niveau de la gestion des ressources qu'ils consomment. Le cycle de vie des nœuds de contexte enfants est contrôlé par les nœuds de contexte parents. Pour la gestion des tâches, les nœuds de contexte actifs enregistrent leurs tâches auprès d'un gestionnaire d'activités. Ainsi, le gestionnaire d'activités, lui-même paramétrable, peut créer une activité par tâche (observation ou notification) ou bien une activité par nœud de contexte actif ou encore une activité pour tout ou partie de la hiérarchie. Pour la consommation d'espace mémoire, un gestionnaire de messages gère des réserves (en anglais, *pools*) de messages et autorise aussi bien les duplications « par référence » que « par valeur ».

Nommage

Pour faciliter les parcours dans le graphe et les reconfigurations, les nœuds de contexte possèdent un nom. Puisqu'un gestionnaire de contexte construit avec COSMOS est local à un terminal, les noms sont locaux. Les noms doivent être uniques, le respect de l'unicité relevant de l'architecte de l'application concevant les politiques d'adaptation.

3.3. *Architecture d'un nœud de contexte*

Avant de présenter l'architecture d'un nœud de contexte, nous introduisons quelques concepts généraux et les notations graphiques utilisées dans l'article pour l'approche composant.

Quelques concepts généraux et leur notation graphique

Comme représenté dans la figure 2, un composant est une entité logicielle qui fournit et requiert des services. Ces services sont regroupés au sein d'interfaces. Nous distinguons les interfaces dites « serveur » qui fournissent des services, des interfaces dites « client » qui sont la spécification des services requis. Un composant possède un contenu. Celui-ci peut être composé d'un ensemble de sous-composants. Dans ce cas, le composant est dit composite. Au dernier niveau, les composants sont dits primitifs. Il est ainsi possible de construire des hiérarchies de composants offrant une vision avec différents niveaux de granularité. Les hiérarchies de composants sont des graphes quelconques (ce ne sont pas nécessairement des arbres). Les composants inclus dans plusieurs composites sont dits partagés. Cette notion représente de façon naturelle le partage de ressources système (segments de mémoire, activités [en anglais, *threads*], etc.). Les composants sont assemblés à l'aide de liaisons. Une liaison représente un chemin de communication entre deux composants, plus précisément entre une interface requise (client) et une interface fournie (serveur) compatible. Enfin, les compositions de composants sont décrites avec un langage de description d'architecture (ADL pour *Architecture Description Language*). Dans la suite, comme montré dans la figure 2, nous utilisons la notation graphique du modèle de composants FRACTAL et le langage FRACTAL ADL (Bruneton *et al.*, 2006) basé sur XML. Cependant, la démarche de conception ne dépend pas de ce modèle de composants.

Architecture d'un nœud de contexte

Toutes les informations de contexte sont des composants étendant le composite abstrait ContextNode (cf. figure 3). Les interfaces Pull et Push sont les interfaces pour l'observation et la notification, respectivement. Les rapports d'observation sont des messages, constitués de sous-messages et de blocs (en anglais, *chunks*) typés. Dans COSMOS, toutes les informations élémentaires des rapports d'observation des entités observables liées aux collecteurs donnent lieu à un bloc typé : par exemple, la qualité du lien réseau WiFi est mémorisée dans un bloc de type LinkQualityChunk. Le composite abstrait ContextNode contient au moins un opérateur (composant primitif

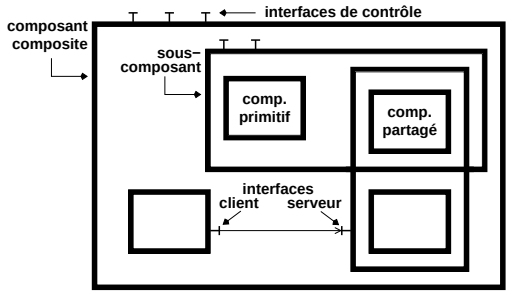

Figure 2. *Modèle de composant* FRACTAL

abstrait ContextOperator) et est connecté à un service de connexion en mode message. De manière générique, le composant ContextOperator prend en entrée les données obtenues par observation ou suite à une notification, effectue un traitement, et si besoin (observation ou notification passante), transfert les résultats du traitement vers la sortie par réponse à l'observation ou par notification. Le concepteur de nouveaux opérateurs doit simplement spécialiser le composant ContextOperator en définissant les traitements à effectuer.

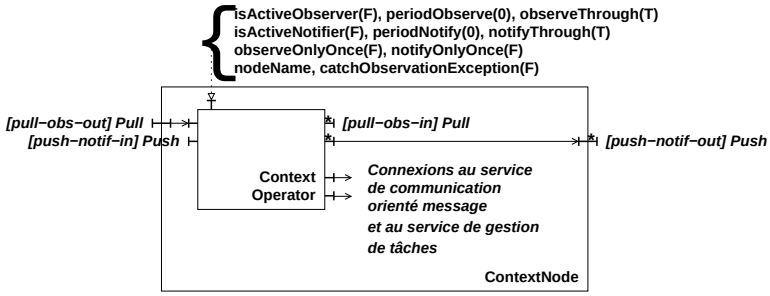

Figure 3. *Composite abstrait* ContextNode

Les possibilités de configuration des nœuds de contexte présentées dans la section précédente sont exprimées par des attributs au niveau du composant primitif ContextOperator. Ainsi, le composant ContextNode est spécialisable *via* des attributs pour la gestion des tâches d'observation et de notification, et pour la propriété passant/bloquant. Par défaut, les nœuds sont passifs (isActiveXxx $= false$) et passants (xxxThrough $= true$) pour l'observation et la notification, et les observations et notifications ne sont pas uniques (xxxOnlyOnce $= false$). Les attributs nodeName et catchObservationException permettent respectivement de nommer le nœud de contexte et de spécifier si les exceptions provoquées par l'indisponibilité d'une information de contexte « remontent » vers le haut de la hiérarchie ou sont masquées

aux nœuds parents. Contrairement aux autres attributs, nodeName ne possède pas de valeur par défaut et doit donc être renseigné comme argument dans les descriptions ADL.

Les nœuds de la hiérarchie se classent ensuite en deux catégories : les feuilles et les autres. Pour les feuilles, le ContextNode est étendu pour contenir, en plus de l'opérateur et des gestionnaires d'activités et de messages, un composant primitif qui encapsule l'accès à la couche du dessous dans l'architecture en couches du gestionnaire de contexte. Cette couche peut être le système d'exploitation ou un autre canevas logiciel construit ou non à l'aide de COSMOS. Par exemple, un nœud de contexte gestionnaire WiFi peut jouer le rôle d'interface pour encapsuler l'accès aux informations de contexte de l'interface WiFi du terminal, soit directement *via* l'interface du système d'exploitation soit *via* un canevas logiciel lui-même spécialisé dans l'accès aux informations des cartes réseaux. Pour les autres nœuds de la hiérarchie, le ContextNode est étendu pour y ajouter un ou plusieurs nœuds composites de type ContextNode. Par exemple, un nœud de contexte peut calculer la capacité mémoire du terminal en englobant deux nœuds de contexte de type ContextNode pour obtenir la moyenne de la mémoire vive disponible et la moyenne du *swap* disponible.

3.4. *Implantation du scénario cache/déchargement*

La figure 4 illustre la mise en œuvre du scénario cache/déchargement que nous avons présenté en section 2.4. Les nœuds du graphe sont repérés par leur nom, chacun indiquant intuitivement le type d'opérateur du nœud de contexte. Les arcs du graphe représentent les relations d'inclusion, y compris les partages de nœuds enfants par plusieurs nœuds parents. À côté des nœuds, figurent ses propriétés : actif/passif, bloquant/passant, etc. Dans la figure 4, la majorité des nœuds actifs observent ; seuls les nœuds détectant les changements d'états (« détecteur de changement des préférences utilisateur » et « détecteur de connectivité ») et de décision (« stabilisation de la décision ») notifient ces changements vers l'application. Les nœuds sans flèche sont passants pour l'observation et la notification. Dans le cas du nœud gestionnaire du réseau WiFi (en bas à droite de la figure), le nœud est bloquant pour l'observation car il gère lui-même une tâche d'observation. Le nœud gestionnaire du réseau WiFi permet de regrouper toutes les informations (qualité du lien, débit et « le débit est-il variable ? ») pour éviter que les trois nœuds parents ne multiplient les accès aux informations système et ne provoquent un grand nombre d'appels système.

D'un point de vue ingénierie logicielle, la plupart des informations de contexte du bas de la hiérarchie sont des informations générales potentiellement demandées par de nombreuses applications. Ensuite, viennent des nœuds comme celui qui calcule le débit WiFi ajusté avec des opérateurs/processeurs plus spécifiques à un domaine, ici l'informatique ubiquitaire. Enfin, les nœuds situés plus haut dans le graphe sont spécifiques à notre cas d'utilisation : la gestion d'importation/exportation de composants applicatifs. Par conséquent, afin de replacer les différents traitements dans l'architecture présentée dans la section 2.1, le premier ensemble de traitements pourrait être

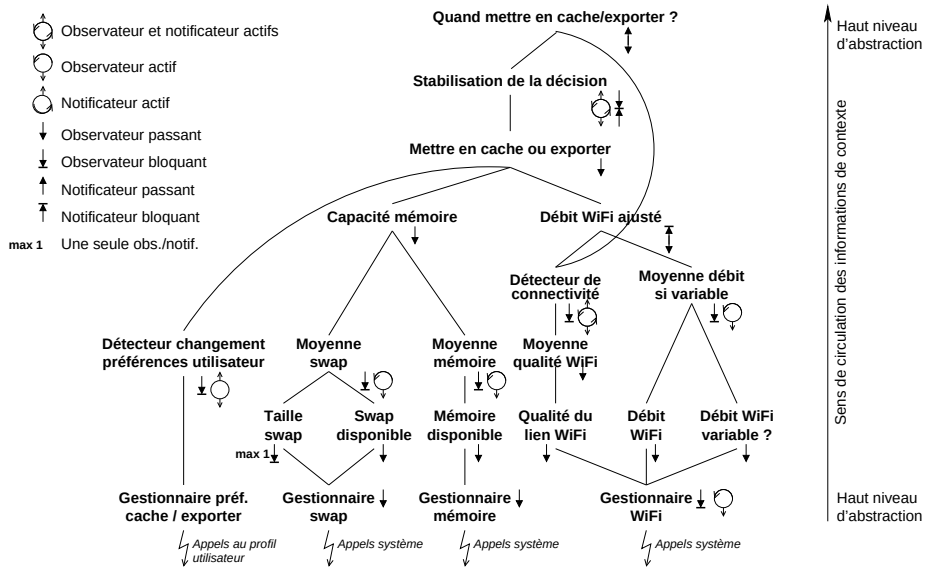

Figure 4. *Exemple de composition de nœuds de contexte*

placé dans un collecteur de contexte de ressources système, le second ensemble dans un processeur de contexte, et le troisième dans un conteneur sensible au contexte.

Dans cet exemple, la décision « quand mettre en cache/exporter ? » nécessite un graphe d'une vingtaine de nœuds de traitement. Dans les solutions existantes dans le domaine de la supervision de systèmes, le concepteur voulant réaliser cet exemple dispose des informations de base sous forme numérique : taille du *swap*, mémoire disponible, qualité du lien WiFi, débit du lien WiFi, etc. Mais, il doit composer « à la main », c'est-à-dire sans aide particulière, et par programmation, les informations de base ; la solution résultante est alors le plus souvent non réutilisable, même partiellement, pour d'autres applications. Le problème réside dans la non-séparation des préoccupations métiers (sélectionner les bonnes informations et calculer correctement les inférences) et extra-fonctionnels (gérer les tâches, les activités, etc.). Avec les solutions existantes dans le domaine de la gestion de contexte, comme indiqué dans la présentation de l'architecture du gestionnaire de contexte en section 2.2, le concepteur est aidé dans la conception des aspects métiers (opérateurs d'inférence), mais les canevas logiciels rendent très difficile la gestion des aspects extra-fonctionnels (gestion des activités, de la mémoire, etc). L'apport de COSMOS est de permettre la conception de telles politiques par composition de composants.

4. Conception de l'architecture du canevas logiciel COSMOS

Après avoir présenté l'architecture des nœuds de contexte, nous étudions les patrons de conception mis en œuvre dans COSMOS. L'originalité de l'approche est de faire en sorte que les patrons de conception se traduisent directement au niveau langage grâce aux notions d'architecture et de composant.

Dans la section 4.1, nous étudions le patron de conception « Composite » pour permettre la composition générique d'informations de contexte, c'est-à-dire pour composer indifféremment les nœuds simples et les nœuds complexes (représentant déjà une hiérarchie de nœuds). Puis, dans la section 4.2, nous présentons le patron de conception « Patron de méthodes » qui permet de définir de manière générique le comportement des nœuds de contexte dans les opérateurs d'inférence (composant abstrait ContextOperator de la figure 3). Ensuite, dans la section 4.3, nous nous appuyons sur le patron de conception « Poids mouche » qui exprime le partage d'un nœud par de nombreuses hiérarchies de nœuds de contexte de plus haut niveau. Enfin, dans la section 4.4, le patron de conception « Singleton » permet par exemple de centraliser la gestion des ressources système pour les traitements d'inférence dans un unique gestionnaire d'activités et un unique gestionnaire de messages.

4.1. *Patron de conception « Composite »*

L'organisation d'informations de contexte sous forme arborescente nécessite d'isoler les différents sous-arbres afin de faciliter leur composition. Cette isolation est réalisée en appliquant le patron de conception « Composite », inspiré de (Gamma *et al.*, 1994) pour les applications orientées objet. Le patron de conception « Composite » permet d'homogénéiser une architecture dans laquelle un élément est constitué de plusieurs objets eux aussi composites, excepté les feuilles de la récursion. Or, l'arborescence construite dans COSMOS exploite la composition de nœuds pour inférer des informations de contexte. Ce type de structure arborescente motive par conséquent l'utilisation du composite afin d'isoler au niveau de chaque nœud les branches qui le composent. Dans cette structure illustrée avec l'exemple de la figure 5, les composants les plus imbriqués dans l'arborescence représentent les ressources système réifiées tandis que les autres nœuds sont les informations de contexte de plus haut niveau d'abstraction.

Grâce au composite, la complexité et les dépendances du nœud sont automatiquement résolues et simplifient par conséquent la composition des nœuds à n'importe quel niveau de l'arborescence. Un exemple de composition de nœuds est décrit dans la portion d'ADL suivante. Cette composition construit le nœud de contexte CachingOrOffloading composé d'un nœud MemoryCapacity et d'un nœud AdjustedBitRate. Il est à noter que le patron de conception « Composite » n'exclut pas le partage de composants à plusieurs niveaux de la hiérarchie (non montré dans cet exemple).

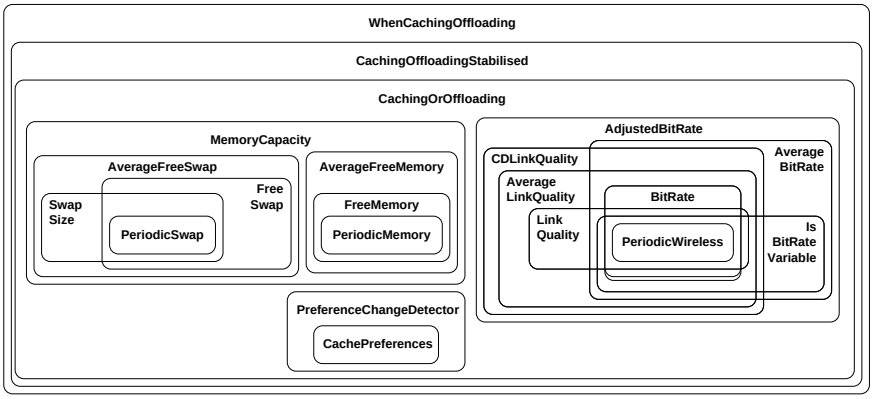

Figure 5. *Illustration du patron de conception « Composite »*

```
<definition name="CachingOrOffloading" extends="ContextNode">
    <component name="cn-1" definition="MemoryCapacity"/>
    <component name="cn-2" definition="AdjustedBitRate"/>
    [...]
</definition>
```

4.2. *Patron de conception « Patron de méthodes »*

Chaque nœud de l'arborescence réalise un traitement particulier sur les informations qui lui sont fournies soit par ses nœuds enfants, soit par le composant primitif encapsulé dans le cas d'une feuille. Les nœuds de contexte réalisent une version orientée composant du patron de conception « Patron de méthodes » (Gamma *et al.*, 1994). Le squelette du nœud est construit par l'assemblage de l'opérateur (extension du composant primitif de type ContextOperator) avec, d'une part, les composants de services techniques toujours présents (gestionnaires d'activités ActivityManager et de messages MessageManager), et d'autre part, les composants des nœuds fils directs dans la hiérarchie. Grâce à cette approche, la définition d'un nœud est simplifiée. Dans l'exemple de la figure 4, les nœuds feuilles de la hiérarchie sont tous dérivés du type PeriodicResourceManager (cf. figure 6). Ce composite générique observe périodiquement une ressource. Il est paramétrable pour que l'utilisateur fournisse, *via* l'attribut resourceName, le nom *système* de la ressource : par exemple, une interface réseau WiFi de nom `eth1`. L'opérateur ForwarderCO (dérivé de l'opérateur abstrait générique ContextOperator) est lui-aussi générique et mémorise n'importe quel type de messages. Les composants génériques et spécifiques à un canevas logiciel collecteur, qui réifie les ressources système, étant fournis dans le canevas logiciel COSMOS, les seules lignes de description ADL que l'utilisateur doit écrire pour obtenir un composite concret, sont les lignes suivantes :

```
<definition name="PeriodicWireless"
            extends="PeriodicResourceManager(PeriodicWireless)">
    <component name="rm" definition="WirelessInterfaceRM(eth1)"/>
</definition>
```

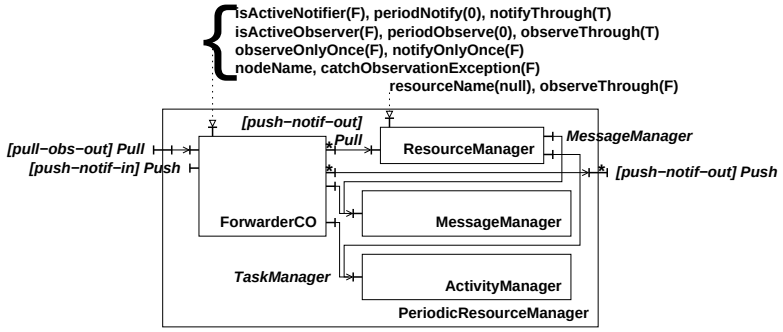

Figure 6. *Composite gestionnaire de ressource PeriodicResourceManager*

Par ailleurs, pour la conception interne (orientée objet) des composants primitifs ContextOperator, nous appliquons aussi le patron de conception « Patron de méthodes » (version orientée objet). Au travers de ses interfaces, ce composant définit des méthodes génériques (respectivement abstraites) à surcharger (respectivement définir) dans les opérateurs concrets. En effet, les interfaces fonctionnelles des nœuds de contexte étant toujours les mêmes, les algorithmes d'observation et de notification sont génériques. Les squelettes de ces algorithmes délèguent la définition de certaines étapes des traitements aux sous-classes. Le morceau de code qui suit présente à titre d'exemple, une version simplifiée de la méthode pull réalisant l'algorithme générique d'observation. Dans cet algorithme, les méthodes dont le nom commence par « `do` » sont à définir dans les sous-classes de ContextOperator.

```
public abstract class ContextOperator
    protected Message message = null;
    [...]
    public Message pull() {
        if (firstObservation) {
            firstObservation = false;
            // create local data containers to send/receive messages
            doInitialiseMessageAndMessages();
            if (doObserve()) doComputeNewMessageContent();
        } else if (!observeOnlyOnce) {
            if (doObserve()) doComputeNewMessageContent();
        }
        return message;
    }
}
```

4.3. *Patron de conception « Poids mouche »*

Les ressources réifiées dans les feuilles de l'arborescence peuvent être utilisées par de multiples nœuds de plus haut niveau afin d'en extraire différents types d'informations. Les feuilles nécessitent donc d'être partagées entre plusieurs nœuds de l'arborescence. Dans l'exemple de la figure 5, le composant PeriodicWireless, qui réifie une ressource système de type réseau sans fil, fournit plus de 30 informations élémentaires. Donc, potentiellement, le nœud de contexte gérant l'observation de cette ressource peut être partagé par une trentaine de composants primitifs responsables de l'isolement, de l'extraction et du traitement d'une de ces informations élémentaires. Le patron de conception orienté objet « Poids mouche » (Gamma *et al.*, 1994) a pour objectif de partager de manière performante de nombreux petits (granularité) objets. En appliquant ce patron de conception à l'orientation composant, les nœuds de contexte dans COSMOS peuvent partager un nœud enfant quelconque de la hiérarchie.

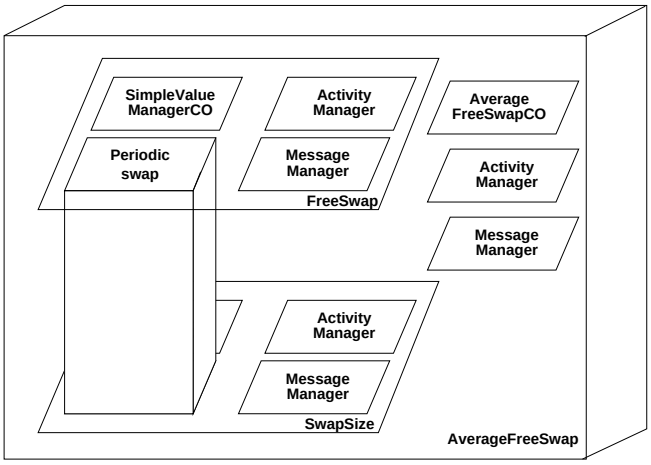

Figure 7. *Illustration du patron de conception « poids mouche »*

Ce type de construction est réalisé en définissant la portion d'ADL ci-après, illustrée dans la figure 7. La définition montre le partage du composant PeriodicSwap par les nœuds FreeSwap et SwapSize. Ces nœuds sont déjà définis dans une autre description ADL. Ici, le partage est effectué en ré-ouvrant récursivement le composant SwapSize et en nommant le composant « cible » de FreeSwap par un chemin (`./cn-1/cn/rm`). Cette description indique que l'instance du composant PeriodicSwap contenue dans le composant SwapSize est la même que celle contenue dans le composant FreeSwap.

```
<definition name="AverageFreeSwap" extends="ContextNode">
    <component name="co" definition="AverageFreeSwapCO(AverageFreeSwap)"/>
    <component name="cn-1" definition="FreeSwap"/>
```

```
    <component name="cn-2" definition="SwapSize">
        <component name="cn" definition="PeriodicSwap">
            <component name="rm" definition="./cn-1/cn/rm"/>
        </component>
    </component>
    [...]
</definition>
```

4.4. *Patron de conception « Singleton »*

Une autre situation de partage de composants est le cas du partage des composants techniques de gestion des activités et des messages. L'objectif est ici de minimiser l'utilisation des ressources pour le fonctionnement du gestionnaire de contexte en centralisant la gestion des ressources système pour leur contrôle à grain fin. Par exemple, l'utilisateur de COSMOS peut désirer n'instancier qu'une activité pour gérer tous les événements d'observation et de notification de la hiérarchie, ou au contraire, partitionner le graphe en zones chacune contrôlée par une ou deux activités[2]. La conséquence est la nécessité de partage entre tous les nœuds de la hiérarchie d'un composant unique pour la gestion des activités. Par analogie, le contrôle des réserves de messages peut être effectué par un seul gestionnaire de messages partagé par toute la hiérarchie. Pour ce faire, nous utilisons le patron de conception « Singleton » (Gamma *et al.*, 1994). La portion d'ADL suivante montre l'utilisation de ce patron de conception. Cette définition ADL organise le partage du composant ActivityManager en précisant que le nœud enfant PeriodicWireless utilise le composant contenu dans le nœud de cette définition (LinkQuality) : `<component name="am" definition="./am"/>`. La ligne suivante opère la même opération pour le composant gestionnaire de messages MessageManager `mm`. Non montrée dans les exemples précédents pour les simplifier, l'application du même patron de conception à tous les niveaux de la hiérarchie permet de n'avoir qu'un seul gestionnaire d'activités et qu'un seul gestionnaire de messages.

```
<definition name="LinkQuality" extends="...">
    <component name="mm" definition="MessageManager"/>
    <component name="am" definition="ActivityManager"/>
    <component name="cn" definition="PeriodicWireless">
        <component name="mm" definition="./mm"/>
        <component name="am" definition="./am"/>
    </component>
</definition>
```

2. Dans le cas d'une seule activité, elle gère les observations et les notifications. Dans le cas de deux activités, l'une gère les observations et l'autre les notifications.

5. Mise en œuvre de COSMOS

COSMOS est un canevas logiciel qui peut être utilisé dans les différentes couches d'une architecture de gestion de contexte telle que présentée dans la figure 1. À titre expérimental, et pour validation et évaluation des principes architecturaux, nous avons appliqué COSMOS à la composition d'informations de ressources système.

Dans un premier temps, la section 5.1 présente les technologies intergicielles que nous avons intégrées pour mettre en œuvre COSMOS. Ainsi, notre choix s'est porté sur les intergiciels FRACTAL, Dream et SAJE. Néanmoins, COSMOS ne se limite pas à ces seules technologies et peut être également implanté avec d'autres modèles de composants comme par exemple WComp (Lavirotte *et al.*, 2005). Dans un second temps, la section 5.2 présente la réalisation de nœuds de contexte COSMOS en utilisant les technologies précédemment citées.

5.1. *Intégration des technologies existantes*

FRACTAL (Bruneton *et al.*, 2006) est un modèle de composant du consortium ObjectWeb[3] pour le domaine des intergiciels en logiciel libre. C'est un modèle de composant léger, hiérarchique, extensible et indépendant des langages de programmation. COSMOS est notamment compatible avec les implantations Julia (Bruneton *et al.*, 2006) et AOKell (Seinturier *et al.*, 2006) disponibles pour le langage Java. Le modèle de composant FRACTAL est associé au langage de description d'architecture appelé FRACTAL ADL. Basé sur une syntaxe XML, ce langage permet d'exprimer des assemblages de composants FRACTAL. La définition du type du document (DTD pour *Document Type Definition*) et la chaîne de traitement de ce langage peuvent être étendues.

Nous utilisons également la bibliothèque de composants FRACTAL DREAM (Leclercq *et al.*, 2005). Celle-ci permet de construire des systèmes orientés message et de gérer de façon fine des activités concurrentes organisées en réserves (*pool*). Dans le cadre de COSMOS, nous tirons parti de cette possibilité de gestion fine des activités. Les composants de contexte ContextNode actifs enregistrent leurs tâches auprès d'un gestionnaire d'activités DREAM (ActivityManager) *via* un contrôleur[4], un second contrôleur servant alors à appeler périodiquement les tâches (observation ou notification). Le gestionnaire d'activités est chargé de faire la correspondance entre les tâches définies par les composants et les unités concrètes d'exécution fournies par le système (typiquement des activités). Cette correspondance peut être mise en œuvre de différentes façons, par exemple en attribuant des priorités différentes aux tâches ou en exécutant les tâches à l'aide d'une réserve d'activités. En outre, nous utilisons aussi le gestionnaire de messages DREAM (MessageManager).

3. Le projet ObjectWeb FRACTAL : `http://fractal.objectweb.org`.
4. La membrane d'un composant FRACTAL est composée d'un nombre arbitraire de contrôleurs et d'intercepteurs réalisant les propriétés extra-fonctionnelles du composant.

Pour la réalisation de l'exemple illustratif utilisé dans cet article, COSMOS s'appuie sur le canevas logiciel SAJE (Courtrai *et al.*, 2003). SAJE réifie sous forme d'objets les ressources système, aussi bien physiques (batterie, processeur, mémoire, réseau, etc.) que logiques (*socket*, fichier, activité, etc.) ; nous n'utilisons dans notre cas que la partie concernant les ressources physiques. Nous avons choisi SAJE parce qu'il continue à évoluer, est maintenu et fonctionne pour les systèmes d'exploitation GNU/Linux, Windows XP, Windows 2000 et Windows Mobile 2003 (pour PDA). SAJE réifie des objets ressources implantant l'interface `ObservableResource`, qui étend l'interface `Observable`, laquelle comprend la méthode `observe` sans argument et renvoie un rapport d'observation typé : `ObservationReport`. SAJE définit ensuite dans une hiérarchie les différents types de ressources et dans une autre les types de rapports d'observation. Par conséquent, les composants primitifs qui sont les nœuds feuilles de la hiérarchie COSMOS encapsulent une ressource observable. Par ailleurs, comme les informations élémentaires du canevas logiciel SAJE ne sont pas fournies avec des informations de qualité de service (par exemple, la précision), les blocs de messages n'en possèdent pas. Mais, rien n'empêche que ce type d'information soit ajouté dans les blocs de messages, ou comme blocs de messages, voire comme sous-messages.

5.2. *Implantation des nœuds de contexte COSMOS*

L'implantation de politiques de gestion de contexte avec COSMOS comprend deux tâches principales : la conception des composants primitifs (les gestionnaires de ressources liés au canevas logiciel de collecte SAJE et les opérateurs génériques ou spécifiques), et la description en FRACTAL ADL de l'architecture (réification architecturale des composants primitifs et compositions). Concernant la réification architecturale et la gestion des attributs des composants primitifs, nous utilisons FRACLET (Rouvoy *et al.*, 2006). FRACLET permet de réduire la taille du code écrit par le développeur en remplaçant le code technique des composants FRACTAL par des annotations. Ces annotations expriment la sémantique des concepts FRACTAL (interface, liaison, attribut, cycle de vie, etc.) sans imposer un modèle de programmation particulier. Le code annoté est ensuite analysé pour générer non seulement le code technique des composants FRACTAL mais aussi les fichiers FRACTAL ADL qui leurs sont associés. Ainsi, sur les 139 fichiers Java de COSMOS (comprenant aussi l'exemple de cet article) et les 207 fichiers FRACTAL ADL, 36 fichiers Java et 56 fichiers ADL sont générés par FRACLET.

Les nœuds de contexte sont conçus pour être hautement configurables *via* de nombreux attributs : au moins 8 par nœud. L'inconvénient est la complexité de la configuration. Pour faciliter la reconfiguration d'une application à base de composants FRACTAL, nous utilisons FPath (David, 2005). FPath propose un langage inspiré de XPath et dédié à la navigation dans les architectures construites à base de composants FRACTAL. Ce langage permet en particulier de capturer un ensemble d'attributs de composants répondant à un critère pour

effectuer une reconfiguration globale de leurs valeurs. Par exemple, l'expression `$root/descendant::*[starts-with("co",name(current()))]` retourne, sous la forme d'un ensemble, un nœud FPath référençant les composants dont le nom commence par `co`[5] défini pour les sous-composants contenus dans le composant racine `root`. Cet ensemble correspond à tous les composants qui contiennent les paramètres de configuration des nœuds de contexte de l'arborescence. COSMOS propose ensuite des méthodes pour parcourir cet ensemble, et par exemple, fixer la valeur d'un attribut d'un nœud sélectionné par son nom de contexte, lui-même donné par l'attribut nodeName.

Une première version de COSMOS est disponible sous licence GNU LGPL à l'URL `http://picolibre.int-evry.fr/projects/cosmos`.

6. Évaluation du prototype

L'objectif de l'évaluation du prototype est de mesurer le coût des compositions de nœuds de contexte avec COSMOS. Nous cherchons à valider expérimentalement la possibilité de construire un gestionnaire de contexte à base de composants à fine granularité. Les coûts de composition des nœuds de contexte excluent par conséquent les coûts de collecte des informations système *via* les appels système. Aussi, pour chaque type de mesures effectuées, nous présentons les durées avec le canevas logiciel SAJE (appels système plus réification objet) et avec COSMOS.

Nous avons mené une série de tests de performance sur un ordinateur portable ayant les caractéristiques suivantes : processeur Intel Pentium cadencé à 1.30 GHz, mémoire vive de 1 Go, carte WiFi Compaq WL110, système d'exploitation GNU/Linux Debian Sarge avec le noyau 2.6.15, la machine virtuelle Java Sun JDK 1.5 *Update* 6 et Julia 2.1.3. Nous avons effectué quelques mesures pour évaluer le surcoût à l'exécution de COSMOS par rapport à SAJE. Les mesures sont rassemblées dans le tableau 1. Les deux durées mesurées sont l'instanciation de l'objet (SAJE) ou du composant feuille (COSMOS) correspondant, puis l'observation. Les ressources observées sont la mémoire (disponible) et l'interface réseau WiFi (qualité du lien). La dernière ligne du tableau indique les durées obtenues pour l'exemple de la figure 4. L'unité de mesure est la milliseconde, les résultats sont obtenus en calculant la moyenne de $1\,000$ instanciations et $10\,000$ observations, respectivement. Pour les valeurs inférieures à la milliseconde, le nombre d'itérations est passé à $1\,000\,000$. La configuration des nœuds de contexte est celle par défaut : observation passante, sans activité, laissant remonter les exceptions.

La première série de mesures (cf. tableau 1-a) concerne la réification et l'extraction des informations sur la mémoire vive. Avec SAJE, l'instanciation de l'objet Memory et l'observation (lecture des attributs de l'objet) correspondent à un accès au système de fichiers Unix `/proc` (qui est présent en mémoire vive) et à l'initialisation des struc-

5. `co` pour ContextOperator. Rappelons que les paramètres de configuration des nœuds de contexte sont contenus dans les opérateurs (cf. figure 1).

				Instanciation (ms)	**Observation** (ms)
a	SAJE	Mémoire disponible	Memory	$0,003$	$0,038$
	COSMOS	Gestionnaire mémoire	PeriodicMemory	$46,4$	$0,045$
	COSMOS	Mémoire disponible	FreeMemory	$156,2$	$0,049$
b	SAJE	Qualité du lien WiFi	WirelessInterface	$16,1$	$14,0$
	SAJE	Toutes les valeurs WiFi	WirelessInterface	$16,1$	$32,7$
	COSMOS	Gestionnaire WiFi	PeriodicWireless	$59,3$	$33,8$
	COSMOS	Qualité du lien WiFi	LinkQuality	$169,2$	$36,2$
c	COSMOS	Configuration par défaut	WhenCachingOffloading	4 019	$163,7$
	COSMOS	Exemple de la figure 4	WhenCachingOffloading	4 019	$4,7$

Tableau 1. *Performances de SAJE et de COSMOS*

tures de données mémorisant les informations élémentaires, soit moins de 1 ms. COSMOS ajoute l'encapsulation dans un gestionnaire de ressources (PeriodicMemory), qui est un composite FRACTAL englobant 4 composants primitifs[6]. Ce surcoût est de 46 ms environ ($46,4-0,003$) ; remarquons qu'un surcoût similaire est observé pour le gestionnaire qui encapsule l'objet SAJE réifiant l'interface réseau WiFi ($59,3-16,1$). La différence entre les observations *via* SAJE et *via* le gestionnaire de mémoire PeriodicMemory, évaluée à environ 7 μs, est la somme (1) du coût de l'appel des composants FRACTAL (traversée des membranes et interception par les contrôleurs), (2) de l'extraction de toutes les informations contextuelles de l'objet SAJE, et (3) du remplissage des blocs de messages *via* le gestionnaire de messages DREAM.

La première série de mesures est complétée avec l'instanciation et l'extraction d'une des informations contextuelles, c'est-à-dire d'un des blocs de messages, en l'occurrence la mémoire disponible. Le nœud de contexte correspondant (FreeMemory) comprend 7 composants, soit un facteur 2 sur le nombre de composants, et pourtant, c'est environ un facteur 3 qui est observé sur les durées d'instanciation. Ce surcoût est explicable par le chargement d'un nombre (proportionnellement) plus important de fichiers FRACTAL ADL[7], ce qui impacte l'instanciation du composant FreeMemory. Pour ce qui concerne l'observation, le surcoût par rapport à l'observation de PeriodicMemory est moins important que la différence entre PeriodicMemory et l'objet SAJE. Ce surcoût correspond à la traversée du composite FreeMemory plus l'extraction du bloc de message FreeMemoryChunk, sachant que la copie du bloc effectuée par le gestionnaire de message DREAM est par référence (opposé à par valeur).

La deuxième série de mesures (cf. tableau 1-b) concerne la réification et l'extraction de la qualité du lien de l'interface réseau WiFi. L'instanciation de l'objet SAJE réifiant l'interface réseau WiFi est plus longue car l'objet SAJE réifiant va chercher des informations présentes sur la carte réseau. Il est d'ailleurs plus rapide d'extraire une unique information contextuelle (la qualité du lien réseau) que d'extraire toutes les informations disponibles (plus de 30 informations élémentaires). Dans COSMOS,

6. Dans un souci de simplification, nous assimilons les gestionnaires de messages et d'activités de DREAM à des composants primitifs, ce qui n'est bien sûr pas le cas.
7. Chaque composant est décrit dans un fichier ADL séparé.

le gestionnaire PeriodicWireless extrait toutes les valeurs disponibles. Aussi, nous n'observons pas de surcoût significatif entre l'observation de la qualité du lien (composant LinkQuality) et l'observation du composant PeriodicWireless.

La dernière série de mesures (cf. tableau 1-c) considère l'exemple de la figure 4 (composant WhenCachingOffloading). Dans le pire des cas, c'est-à-dire tous les composants sont passants en observation, l'observation nécessite 163 ms. Si les composants sont configurés comme présenté dans la figure 4, c'est-à-dire les composants « décision stabilisation » et « détecteur de connectivité » sont bloquants, la durée de l'observation devient négligeable : $4,7$ ms, sans que la précision des informations de contexte ne soit remise en cause. Cela nous permet de conclure que la composition de nœuds de contexte est pertinente et performante. Enfin, la durée d'instanciation est très élevée. Le prix payé est celui de la dynamicité et de la configurabilité. Cependant, à cause d'un bogue dans FRACTAL ADL, nous n'avons pas pu réaliser le patron de conception « Singleton » comme présenté dans la section 4.4, mais avons dû instancier dans chaque nœud de contexte un composant primitif dont le corps (la classe Java) implante le patron de conception « Singleton ». Ainsi, au lieu d'une trentaine de composants à instancier, nous nous retrouvons à instancier une soixantaine de composants. Nous avons réalisé un essai en re-spécifiant les composants FreeMemory et LinkQuality sans utiliser les mécanismes d'extension de FRACTAL ADL ; nous obtenons des résultats intéressants : l'instanciation dure dans ce cas environ 75 ms, soit un facteur 2 par rapport aux résultats du tableau 1. Afin de quantifier le surcoût engendré par FRACTAL ADL, nous avons mesuré le coût d'instanciation d'un composant WhenCachingOffloading à partir d'un *template*[8] : environ 45 ms. Des optimisations, à rechercher dans FRACTAL, sont en cours de test par les développeurs du projet FRACTAL.

Pour conclure, notons que les instanciations de composants de nœuds de contexte ne sont *a priori* pas aussi fréquentes que les instanciations de composants métiers : le gestionnaire de contexte est un service technique qui peut être démarré automatiquement à l'initialisation du terminal mobile. Sur ce point des instanciations de composants FRACTAL avec FRACTAL ADL, les premières pistes de recherche nous font espérer une amélioration d'au moins un facteur 2 après la résolution du bogue FRACTAL ADL précité. L'instanciation de l'exemple prendrait alors environ 2 s, et n'intervient qu'une seule fois dans l'exécution. Quant à l'observation, remarquons que nous montrons le pire des cas. Dans une architecture correctement configurée, les gestionnaires de ressources sont bloquants et actifs pour l'observation. Les observations des objets SAJE sont alors effectuées en tâche de fond et ne sont plus dans le chemin critique d'une observation de plus haut niveau. Par exemple, dans le cas de l'observation de la qualité du lien réseau WiFi, l'observation avec SAJE prend 14 ms et celle avec COSMOS $36,2 - 33,8 = 2,4$ ms. Par conséquent, le coût des compositions de contexte

8. Un *template* FRACTAL réalise le patron de conception « Fabrique » (Gamma *et al.*, 1994) et permet d'instancier des composants qui sont quasi « isomorphes » au *template*. Le surcoût dû à FRACTAL ADL est alors supporté par la création du *template* et n'existe plus lors de la création des composants à partir du *template*.

est négligeable au regard du coût des appels au système d'exploitation. Il est donc réaliste de construire un gestionnaire de contexte à base de composants FRACTAL à granularité fine.

7. Travaux connexes

Dans cette section, nous commençons par positionner COSMOS par rapport aux canevas logiciels d'observation des ressources système en étudiant les canevas logiciels Phoenix, SAJE et LeWYS. Ensuite, nous positionnons plus précisément COSMOS par rapport à quatre canevas logiciels de gestion de contexte : Context Toolkit, le *Context Service* de MoCA, MoCoA, Draco, Le Contexteur et WildCAT. Nous terminons l'étude de la littérature avec d'autres canevas logiciels de gestion de contexte.

Phoenix est un canevas logiciel d'observation système d'applications réparties s'exécutant sur une grappe de machines (en anglais, *cluster* (Boutros Saab *et al.*, 2002)). L'architecture de Phoenix est décomposée en quatre modules : agent d'observation, sondes (dans l'application), diffusion sur le réseau local et librairie pour outillage. Les agents d'observation permettent de configurer la fréquence des observations et de multiplexer les observations (ajustement de la fréquence à la plus petite des valeurs demandées). Phoenix fournit un langage dédié pour exprimer l'observation demandée : identifiant des ressources observables, opérateurs de comparaison, opérateurs logiques de premier ordre et opérateur `DELTA` pour l'amplitude des variations. Le langage fournit des opérateurs relativement élémentaires : pas d'opérateur à mémoire, à seuil, de traduction de format, de fusion de données, etc. Cependant, l'approche langage dédié pour l'expression des besoins d'observation est à utiliser pour les futurs développements de COSMOS. En outre, Phoenix ne fournit pas d'aide pour l'ajout de nouveaux opérateurs, contrairement à COSMOS qui propose un modèle d'architecture de nœuds de contexte orienté composant dans lequel les différents composants (par exemple les opérateurs sous-types de ContextOperator) peuvent être étendus ou remplacés.

SAJE est un canevas logiciel orienté objet qui réifie les ressources système (Courtrai *et al.*, 2003). L'apport de l'orientation composant de COSMOS utilisant SAJE est la composition des informations élémentaires fournies par SAJE. SAJE propose des classes gérant des activités pour l'observation périodique des ressources. Mais, l'orientation objet impliquant une approche programmation, rend difficile le contrôle du nombre de ces activités et leur paramétrage (par exemple, périodicité). En outre, les composants ajoutent la notion d'interface ou service requis, permettant ainsi une modélisation architecturale et une implantation plus explicite et plus élégante de la notification (en plus de l'observation). En orienté objet, le concepteur doit avoir recours à des patrons de conception, par exemple « Observateur » (Gamma *et al.*, 1994), et ceci se complique lorsqu'il faut coupler ce patron de conception avec d'autres gérant les activités par exemple. C'est, selon nous, ce type de constatations

qui a conduit aux travaux sur Le Contexteur (Coutaz *et al.*, 2002; Rey *et al.*, 2004) par exemple (cf. ci-après page 1216). Dans COSMOS, nous allons plus loin en quittant l'orientation objet et en explicitant la notion de nœud de contexte avec un modèle de composant.

LeWYS est un canevas logiciel pour la supervision de grappes de machines (Cecchet *et al.*, 2005). L'architecture comprend des sondes sur chaque machine de la grappe et un système réparti de notification d'événements. Les sondes système sont réalisées et optimisées pour Linux et Windows XP. Elles récupèrent les informations de performance : processeur, mémoire, disque, réseau, noyau (dans le cas de Linux). LeWYS cible les applications J2EE sur grappe et fournit donc des sondes spécifiques JMX (*Java Management Extensions*)[9], et plus spécifiquement pour l'intergiciel C-JDBC (Cecchet *et al.*, 2004). Même si LeWYS est construit à base de composants FRACTAL, il ne propose pas de fonctionnalité de composition des informations de contexte. Par exemple, toutes les informations des sondes système sont transmises pour analyse, sans filtrage. Dans COSMOS, le filtrage est naturellement opéré par les nœuds de contexte situés juste au-dessus des collecteurs PeriodicResourceManager et qui sélectionnent les informations de contexte pertinentes : mémoire disponible, qualité du lien WiFi, etc. Par conséquent, LeWYS, plus particulièrement la partie de l'architecture comprenant les sondes, peut potentiellement être vu comme un canevas logiciel producteur de données en entrée pour COSMOS, donc en plus ou à la place de SAJE.

Context Toolkit Parmi les premiers travaux sur la gestion de contexte, Context Toolkit (Dey *et al.*, 2001) est un canevas logiciel orienté objet empruntant aux IHM les concepts de programmation événementielle et de *widget* pour la collecte du contexte des ressources. Dans le même canevas logiciel, sont rassemblées les autres fonctionnalités : l'interpréteur pour composer et abstraire les informations de contexte, l'agrégateur pour la médiation avec l'application, le *service* pour contrôler les actions de l'application sur le contexte et le *discoverer* qui est un serveur de noms ou registre. Dans la philosophie de ce canevas logiciel, les fonctions d'interprétation et d'agrégation sont à programmer dans des blocs monolithiques : un agrégateur et un interpréteur par application, et ce quels que soient le nombre de *widget*s et le niveau d'abstraction demandés par l'application. Enfin, la gestion des ressources système consommées par les traitements de gestion de contexte, et plus particulièrement, la gestion des activités, est explicitement laissée de côté.

***Context Service* de MoCA** Les auteurs du *Context Service* de MoCA (da Rocha *et al.*, 2006; da Rocha *et al.*, 2005) mettent en exergue la nécessité de spécifier les offres et les besoins d'informations de contexte, et suggèrent l'approche orientée composant pour ces spécifications. L'architecture définit une interface d'accès par les

9. `http://java.sun.com/products/JavaManagement`.

clients, un service de gestion des événements, un gestionnaire de types d'information de contexte et un dépôt d'informations de contexte. L'architecture du gestionnaire de contexte met en évidence la nécessité d'adjoindre des services extra-fonctionnels, que les auteurs appellent sous-services orthogonaux, pour la performance de gestion de contexte. En outre, les informations de contexte sont typées et décrites dans un modèle basé sur XML : informations structurelles (attributs, dépendances, partages), comportementales (fraîcheur, à la demande) et spécifiques (événements de notification, requêtes d'observation). Ces descriptions construisent un système de type ensuite traduit en objets Java.

De manière similaire à nos travaux, les auteurs notent la nécessité d'utiliser des méta-informations pour gérer la performance et le passage à l'échelle des inférences, un point que les auteurs signalent comme beaucoup plus difficile à réaliser avec des moteurs d'inférence d'ontologies. En revanche, l'architecture du *Context Service* de MoCA ne sépare pas les fonctionnalités du gestionnaire de contexte : le gestionnaire de type et le dépôt sont les seuls points d'accès aux informations de contexte, et ce quel que soit leur niveau d'abstraction et quelle que soit l'utilisation qu'il en est faite. En fait, les auteurs transposent l'approche à base d'ontologies en une approche à base d'objets : par exemple, la provenance d'une information de contexte (locale ou distante) est donnée par un attribut au lieu d'être explicitée dans l'architecture. En outre, dans COSMOS, nous appliquons l'approche orientée composant tant au niveau de la définition de l'architecture du gestionnaire de contexte, qu'au niveau de la définition du concept de nœud de contexte. Le modèle basé XML de MoCA correspond plus ou moins à un descripteur de composant avec ses attributs. Dans le cas d'un ADL, la spécification est explicite et les manipulations sont plus puissantes et facilitées par les outils « classiques » fournis avec l'ADL. Enfin, pour des raisons de performance, les auteurs envisagent de partitionner l'espace des informations de contexte en vues. Avec un modèle de composant tel que FRACTAL qui autorise la construction de hiérarchies de composants avec partage, cette possibilité est automatiquement fournie.

MoCoA (Senart *et al.*, 2006) fournit un environnement de construction d'applications sensibles sur réseaux *ad hoc* à base d'objets sensibles (en anglais, *sentient objects*) (Biegel *et al.*, 361–365). Les objets sensibles possèdent la plupart des caractéristiques des modèles de composant de la littérature. La gestion de contexte dans l'architecture de MoCoA est divisée en deux parties. Tout d'abord, les traitements d'inférence de bas niveau organisent des tubes de fusion de données composés de composants[10] partageables. Le seul mode de communication des informations de contexte est la notification. Ces traitements dans les tubes sont complétés par des traitements d'inférence dont l'esprit est emprunté au domaine de l'intelligence artificielle (à base de faits et de règles). Ces derniers traitements constituent la partie métier de l'application sensible au contexte. Les tubes sont logiquement inclus dans, et donc gérés par, les objets sensibles, donc y compris pour la gestion des ressources système consom-

10. Les auteurs n'indiquent pas le modèle de composant utilisé pour réaliser cette partie de l'architecture de MoCoA.

mées par les traitements de contexte. Les données de contexte sont enregistrées dans une base de données. Enfin, aucun élément ne décrit la gestion des ressources système (mémoire, activité, etc.) consommées pour les traitements.

Contrairement à MoCoA, COSMOS permet l'observation d'information de contexte, en plus de la notification. En outre, les auteurs de MoCoA notent l'absence et la nécessité d'utiliser un ADL pour exprimer la composition des tubes et des objets sensibles, possibilité fournie dans COSMOS. Conceptuellement, moyennant l'encapsulation dans un composant du modèle de ceux utilisés dans les tubes, COSMOS pourrait être utilisé pour la réalisation des tubes dans les objets sensibles.

Draco Le gestionnaire de contexte de Draco (Preuveneers *et al.*, 2005) est basé sur les orientations ontologies pour la modélisation des informations de contexte et composants pour la conception de l'architecture. L'architecture de Draco est donc organisée autour d'une base de données et d'un courtier d'ontologies. Le *Context Control Block* est composé des composants *Context Retrieval*, *Context Storing* et *Context Manipulation*. Le premier composant peut gérer la qualité des informations collectées (précision, confiance et âge) et contient les fonctionnalités spécifiques d'un collecteur d'informations de contexte (temporisation, pertinence, filtrage, comparaison de précision). Le composant *Context Storing* gère les données et les métadonnées de contexte respectivement dans les composants *Fact Container* et *Ontology Container*. Ce composant délivre les données sur demande au dernier composant, le composant *Context Manipulation*. Ce dernier composant fournit des opérateurs de transformation de données, de raisonnement en utilisant le moteur d'inférence Jena 2, et de dissémination des informations de contexte entre machines.

L'orientation composant est choisie pour sa capacité d'adaptation dynamique du gestionnaire de contexte suite aux changements aussi bien des exigences des applications que des entités du contexte lui-même. L'objectif est de déployer/replier ces composants à la demande. L'inconvénient de cette utilisation du patron de conception « Singleton » pour les services fonctionnels est le non-passage à l'échelle. Au contraire, dans COSMOS, les fonctionnalités telles que le filtrage, la gestion de l'historique, la transformation de données, sont répliquées et intégrées dans les nœuds de contexte lorsque nécessaire. Concernant l'orientation ontologie, l'évaluation de Draco dans (Preuveneers *et al.*, 2005) conclut *(1)* à la difficulté de définir un déploiement optimal à cause de la difficulté d'évaluer précisément les durées de calcul de toutes les activités de gestion de contexte, et *(2)* à la difficulté d'exécuter un moteur d'inférence d'ontologies sur les petits matériels. Enfin, Draco fournit des fonctionnalités non disponibles dans COSMOS, les plus importantes étant la dissémination des informations de contexte et l'inférence d'information de contexte de plus haut niveau. La première fonctionnalité peut aisément être ajouté à COSMOS *via* le bus à messages Dream. Comme suggéré par les auteurs de Draco, pour les petits matériels, on peut concevoir l'utilisation de moteurs d'inférence d'ontologies déportés sur des hôtes plus puissants.

Le Contexteur Les travaux les plus proches des nôtres sont les contexteurs (Coutaz *et al.*, 2002; Rey *et al.*, 2004), qui sont des entités logicielles ressemblant à des composants avec des données et leurs métadonnées (décrivant la qualité des données) en entrée et en sortie, et des contrôles (modifications de la configuration) en entrée et en sortie. Le gestionnaire de contexte est alors construit comme un réseau de contexteurs. Le mode d'échange des données est paramétrable : envoi unique (une seule fois) des données, envoi à chaque fois que la valeur change, envoi sur demande (observation), envoi à chaque calcul (notification). COSMOS possède plus de possibilités en rendant les observations et les notifications symétriquement paramétrables.

Un contexteur est une classe Java accompagnée d'un fichier de description XML. Ainsi, le canevas logiciel construit de manière *ad hoc* un conteneur autour de ce que l'on peut appeler un composant logiciel. Le modèle de composant *ad hoc* construit par la plate-forme n'est pas explicite et n'est donc pas configurable, par exemple, en ce qui concerne la gestion des ressources système. La version orientée objet du patron de conception « Patron de méthode » peut être appliquée à la conception des hiérarchies de classes Java des contexteurs, mais pas la version orientée composant du patron de conception. En effet, le canevas logiciel des contexteurs est dépourvu de mécanismes d'extension des définitions qui se trouvent dans FRACTAL ADL (cf. section 4.2) et qui permettent la mise en œuvre du patron de conception « Patron de méthodes ». Ces mécanismes ont pour rôle d'exprimer en XML les extensions et de générer ou fabriquer le conteneur adéquat.

Chaque contexteur possédant au moins une activité, la consommation de ressources locales pose question pour les exemples tels que celui que nous ciblons. Pour limiter le nombre d'activités, les auteurs proposent d'appliquer le patron de conception « Composite » en encapsulant des contexteurs dans des contexteurs. Théoriquement, c'est-à-dire dans la spécification XML, le canevas logiciel permet donc une configuration avec une activité (ou un ensemble d'activités) pour toute une hiérarchie. Cependant, l'implantation n'autorise pas l'expression du patron de conception « Composite », contrairement à COSMOS. En outre, se poserait la question du partage de sous-contexteurs encapsulés dans des contexteurs différents, ce qu'autorise FRACTAL avec les composants partagés.

Les données échangées entre les contexteurs sont accompagnées de métadonnées. Les contexteurs échangent aussi des informations de contrôle. Les flots de contrôle circulent uniquement dans le sens contraire de ceux des données. Les auteurs donnent des exemples de requêtes de contrôle : demande de fermeture, d'envoi de données, d'arrêt pendant un certain temps, de changement du mode d'échange. Cependant, le fait que le modèle de composant ne soit pas explicite et donc non « manipulable » empêche d'étendre les possibilités de configuration : nouveaux attributs, nouveaux modes de contrôle. Dans COSMOS, le cycle de vie et la structure des composants sont finement gérés grâce aux contrôleurs FRACTAL, permettant ainsi des flots de contrôle de haut en bas selon le patron de conception « Composite ». Par ailleurs, le découpage des messages en morceaux typés permet d'ajouter des morceaux de message ou des

sous-messages pour ces informations de contrôle, et par conséquent, autorise que les flots de contrôle utilisent les mêmes chemins de communication, de bas en haut de la hiérarchie.

En conclusion, le canevas logiciel des contexteurs couvre aussi bien la collecte locale que la distribution des informations de contexte sur réseau mobile. Les contexteurs sont classés en catégories, mais le même concept de contexteur est appliqué à tous les niveaux. Nous défendons l'idée que la gestion des ressources doit être adaptée à chaque niveau. La gestion d'activités du contexteur nous semble adaptée à la distribution d'informations de contexte sur réseau, mais pas à la gestion d'informations de contexte de ressources système avec de très nombreuses informations élémentaires à composer et en contrôlant très finement les ressources consommées pour cela.

WildCAT Les travaux autour de WildCAT (David *et al.*, 2005) visent à mettre en relation la gestion de contexte de ressources système avec le modèle de composant FRACTAL. En particulier, le canevas logiciel WildCAT propose de faciliter la création d'applications sensibles au contexte en définissant une architecture logicielle extensible. Cette architecture est structurée comme un ensemble de *domaines contextuels* qui isolent les différents aspects du système (ressources matérielles, ressources logicielles, topologie du réseau, etc.). Les ressources système regroupées par domaines contextuels sont accessibles *via* une syntaxe simple d'adressage, inspirée des URI (*Uniform Resource Identifier*). Par exemple, l'expression `sys://storage/drives/hdc#removable` permet à toute application d'accéder à la ressource système `hdc` afin de déterminer si celle-ci est amovible ou non. Certes, cette simplicité d'utilisation facilite la conception des applications sensibles au contexte mais WildCAT ne propose pas d'opérateur de composition pour construire des abstractions de plus haut niveau tel qu'il est proposé dans COSMOS. En effet, COSMOS propose aussi un ensemble d'opérateurs génériques (ou à défaut configurables) pour concevoir des politiques d'observation de contexte de haut niveau. En outre, WildCAT est une librairie orientée objet pour la réification des ressources système. COSMOS bénéficie en plus de toutes les propriétés du modèle de composant FRACTAL pour fournir une vision architecturale de l'arbre des ressources système observées.

Un contexte de ressources système WildCAT supporte aussi bien l'exécution de requêtes de découverte (mode *Pull*) que la notification des changements des informations (mode *Push*). Ces modes de fonctionnement sont également disponibles dans COSMOS tout en offrant des politiques d'observation et de notification plus élaborées. En particulier, la granularité fine de conception de COSMOS permet de configurer la gestion des activités utilisées pour la réification et la composition des informations contextuelles. COSMOS propose également un mode de notification périodique et un mécanisme configurable d'historique des informations contextuelles non disponible dans WildCAT.

Le canevas logiciel WildCAT prend en compte l'évolution du système en supportant l'apparition (respectivement la disparition) des ressources système ou des informations associées à une ressource système. Dans COSMOS, toute ressource système

potentiellement observable est décrite par un composant présent à l'exécution même si la ressource système associée n'est pas disponible. Tant que la ressource n'est pas disponible, une exception lors de la réification est levée. L'attribut du nœud de contexte catchReificationException permet de gérer cette exception pour la faire remonter ou non à l'application. De plus, la dynamicité du modèle de composant FRACTAL autorise la reconfiguration de l'arborescence des informations de contexte pendant l'exécution pour intégrer de nouvelles informations de contexte. Cette dernière fonctionnalité est en cours d'étude dans COSMOS.

La configuration de WildCAT est réalisée *via* un fichier XML spécifique décrivant la structure de chaque domaine contextuel. Dans COSMOS, cette configuration est réalisée *via* un descripteur FRACTAL ADL standard. Ainsi, dans le cadre d'un développement d'applications avec le modèle de composant FRACTAL, l'utilisation de COSMOS permet d'obtenir une vision architecturale complète et homogène depuis la logique métier jusqu'à la réification des ressources systèmes. Ce canevas logiciel peut être intégré dans COSMOS afin de compléter les domaines contextuels disponibles dans SAJE avec ceux de WildCAT.

Autres canevas logiciels de gestion de contexte *Context Information System* (Pascoe, 1998) est un canevas logiciel orienté objet permettant de mémoriser les informations de contexte dans différents formats, et ce, de la collecte à l'exploitation finale par l'application. Les *Context Handling Components* (Gray *et al.*, 2001) s'organisent en une architecture en couches à trois niveaux : l'acquisition, la transformation et le contrôle par dialogue des deux couches basses. Les composants ressemblent beaucoup aux contexteurs avec des données et des métadonnées en entrée et en sortie. Les auteurs listent des critères de qualité (métadonnées) que pourrait utiliser COSMOS : résolution, précision, répétabilité, fraîcheur, fréquence, etc. RCSM (Yau *et al.*, 2002) est un canevas logiciel orienté objet proposant une architecture proche de celle proposée dans la section 2.1. Chaque source d'information de contexte (utilisateur, capteur, système d'exploitation, machine distante) est clairement séparée. Mais, les auteurs n'abordent pas la question de la synchronie des traitements ni celle de la gestion des ressources système. PACE (Henricksen *et al.*, 2005) organise différemment l'architecture du gestionnaire de contexte en introduisant un dépôt d'information de contexte géré par une base de données. Les méta-informations (temporalité, qualité, etc.) sont ajoutées soit aux informations de contexte, soit aux relations entre elles. Les auteurs de PACE indiquent clairement que les questions de passage à l'échelle et de performances non pas été étudiées.

En conclusion, dans les solutions de la littérature, les fonctionnalités de gestion de contexte ne sont pas séparées et le contrôle des ressources système consommées par les traitements (de collecte, d'interprétation, d'identification de situations d'adaptation) n'est pas explicitement exprimé. En outre, nous ne pensons pas qu'un seul canevas logiciel puisse traiter la diversité des dispositifs matériels et systèmes. Pour ce faire, nous disposons de canevas logiciels dédiés à la collecte comme LeWYS (Cecchet *et al.*, 2005), SAJE (Courtrai *et al.*, 2003) et WildCAT (David *et*

al., 2005)). Nous ne pensons pas non plus qu'un seul canevas logiciel puisse exprimer la diversité des interprétations et des identifications de situations d'adaptation. Dans la littérature, nous trouvons des processeurs de contexte qui expérimentent de nouveaux opérateurs comme NWS (Wolski, 1998) et NWS-Lite (Gurun *et al.*, 2004), CARISMA (Capra *et al.*, 2003), WComp (Lavirotte *et al.*, 2005), et des conteneurs sensibles au contexte qui étudient plus précisément les mécanismes de collaboration entre l'application et l'intergiciel pour l'adaptation comme CAMidO (Belhanafi Behlouli *et al.*, 2006), Enactor (Newberger *et al.*, 2003), JADE-LEAP (Bucur *et al.*, 2005), RCSM (Yau *et al.*, 2002) et PACE (Henricksen *et al.*, 2005)). Par exemple, NWS et NWS-Lite construisent des prédictions sur le fonctionnement futur d'un système (bande passante réseau, consommation d'énergie) comme des prévisions météorologiques ; CARISMA met en œuvre une approche micro-économique à base d'enchères ; ou encore WComp étudie la notion de distance. CAMidO complète les conteneurs des composants CORBA pour les rendre sensibles au contexte ; et Enactor fait le lien entre Context Toolkit et Macromedia Flash pour faciliter l'observation du contexte et ensuite mieux le contrôler.

8. Conclusion

Les environnements ubiquitaires imposent des contraintes fortes sur la conception et le développement des applications. Au-delà de l'action d'adaptation elle-même, la prise de décision pour le déclenchement de cette adaptation est un problème complexe pour lequel peu de solutions existent. Cette décision repose sur une collecte, une analyse et une synthèse des nombreux paramètres physiques et logiques fournis par le contexte d'exécution. Pour cela, nous proposons, dans cet article, le canevas COSMOS pour la gestion des informations de contexte.

Le canevas COSMOS permet de construire des gestionnaires de contexte en environnements ubiquitaires. Par rapport à l'existant, l'originalité de ce canevas, destiné à être utilisé pour adapter des applications, est d'être conçu de façon à être lui-même adaptable. Pour cela, nous avons conçu et développé COSMOS selon trois grands principes : séparation entre les activités de collecte et de synthèse des données de contexte, organisation des politiques de gestion de contexte en assemblages de composants logiciels et utilisation systématique de patrons de conception. Le premier principe fondateur de COSMOS est de séparer la collecte des données de contexte des traitements qui peuvent leur être appliqués. Cela permet de proposer une nouvelle architecture de gestionnaire de contexte avec plusieurs niveaux ou successions de cycles « collecte/interprétation/identification de situations ». COSMOS étant conçu à base de composants, un gestionnaire de contexte peut être construit à l'aide d'une seule instance de COSMOS, contenant alors plusieurs de ces cycles et effectuant tous les traitements d'inférence, ou de plusieurs instances de COSMOS composées entre elles, pouvant avoir dans ce cas un double rôle : glu entre des canevas logiciels hétérogènes et réalisation d'opérateurs d'inférence spécifiques.

Le second principe fondateur de COSMOS est celui de l'utilisation systématique de composants logiciels pour la définition des politiques de gestion de contexte. Une politique est donc un assemblage de composants. Chaque composant, appelé nœud de contexte, récupère des informations des composants des niveaux inférieurs et les répercute aux niveaux supérieurs. Dans l'exemple développé dans cet article, les feuilles de la hiérarchie encapsulent les ressources systèmes réifiées sous forme de composants (dans notre cas, avec SAJE). Chaque nœud peut être passif ou actif, avec exécution périodique de tâches dans des activités. La circulation d'information dans la hiérarchie peut se faire soit par notification, soit par observation. Les approches existantes dans la littérature suivent presque toutes une approche à base d'objets et d'ontologies (base de données). L'approche orientée composant suivie par COSMOS permet de systématiser et de rendre plus souple la conception et le développement de nouvelles politiques de gestion de contexte.

Enfin, le troisième principe fondateur de COSMOS est l'utilisation de patrons de conception pour l'organisation et l'assemblage des composants nœuds de contexte. Ces patrons sont des schémas architecturaux qui ont un caractère répétitif dans l'organisation des politiques de gestion de contexte. Nous en avons ainsi identifiés quatre : ce sont les patrons Composite, Patron de méthodes, Poids mouche et Singleton. L'apport de ces patrons est de guider les concepteurs dans la définition de leurs politiques et de rendre la conception plus systématique.

En guise de perspectives, COSMOS constitue une brique logicielle de base pour le développement d'une plate-forme d'adaptation pour les applications réparties en environnement mobile. Pour l'instant, les politiques de gestion de contexte sont définies statiquement dans COSMOS. Nous étudions donc l'adaptation dynamique de la gestion de contexte. Ainsi, pour rendre COSMOS dynamiquement adaptable par des ajouts et des retraits de politiques de compositions avec des intersections non vides entre elles, nous prévoyons d'utiliser FScript (David, 2005). L'intérêt serait alors, d'un côté, de permettre à des applications clientes de modifier leurs politiques, et de l'autre, d'auto-adapter la gestion de contexte lors de la découverte de ressources système, par exemple lors de l'apparition d'une nouvelle interface réseau à observer. Par ailleurs, afin d'offrir un formalisme plus accessible pour les utilisateurs non familiers à la syntaxe de FRACTAL ADL, nous envisageons la définition d'un langage dédié à la déclaration des compositions d'informations de contexte. Ce langage pourrait s'inspirer du langage disponible dans le canevas logiciel WildCAT (David *et al.*, 2005) et il disposerait d'un outil de conversion vers FRACTAL ADL afin de conserver les avantages de notre approche. Par ailleurs, la puissance d'expression de la composition de nœuds de contexte est permise grâce à la définition d'interfaces très génériques, nommément push et pull. Il en résulte l'inconvénient que peu de contraintes sont exprimées sur la cohérence des assemblages de composants. En fait, dans un intergiciel orienté message tel que DREAM, les contraintes sont plutôt présentes dans le contenu (donc le type) des messages que dans le type des interfaces. Par conséquent, nous étudions la possibilité d'utiliser les travaux réalisés dans DREAM sur la vérification par système de types des messages (Bidinger *et al.*, 2005). Enfin, nous étudions la prise en compte des environnements contraints dans COSMOS. Notre objectif est d'expé-

rimenter COSMOS sur les petits équipements en utilisant la version J2ME CLDC de Julia ou de AOKell. Finalement, nous envisageons de mettre en œuvre d'autres études de cas afin d'éprouver la réutilisabilité des politiques de gestion de contexte apportée par COSMOS.

Remerciements

Les auteurs tiennent à remercier, par ordre alphabétique, Djamel Belaïd, Sophie Chabridon, Bertil Folliot, Pierre Sens et Chantal Taconet pour leur relecture en détail et leurs nombreux commentaires qui ont permis d'améliorer de manière significative la présentation des travaux de recherche exposés dans cet article. Que les évaluateurs anonymes, par leurs nombreux commentaires sur la forme et sur le fond, soient aussi remerciés.

9. Bibliographie

Belhanafi Behlouli N., Taconet C., Bernard G., « An architecture for supporting Development and Execution of Context-Aware Component applications », *Proc. IEEE International Conference on Pervasive Services*, June, 2006.

Bidinger P., Leclercq M., Quéma V., Schmitt A., Stefani J.-B., « Dream Types : A Domain Specific Type System for Component-Based Message-Oriented Middleware », *Proc. 4th ESEC/FSE Workshop on Specification and Verification of Component-Based Systems*, Lisbon (Portugal), September, 2005.

Biegel G., Cahill V., « A framework for developing mobile, contex-aware applications », *Proc. 2nd IEEE Conference on Pervasive Computing and Communications*, Orlando, Florida (USA), p. 2004, March, 361–365.

Boulkenafed M., Issarny V., « A middleware Service for Mobile Ad Hoc Data Sharing, Enhancing Data Availability », *in* M. Endler, D. Schmidt (eds), *Proc. IFIP/ACM/USENIX International Middleware Conference*, vol. 2672 of *Lecture Notes in Computer Science*, Springer-Verlag, Rio de Janeiro, Brazil, June, 2003.

Boutros Saab C., Bonnaire X., Folliot B., « A flexible monitoring platform to build cluster management services », *Proc. IEEE International Conference on Cluster Computing*, Chemnitz, Germany, p. 258-265, November, 2000.

Boutros Saab C., Bonnaire X., Folliot B., « PHOENIX : A Self Adaptable Monitoring Platform for Cluster Management », *Cluster Computing*, vol. 5, n° 1, p. 75-85, January, 2002.

Bruneton É., Coupaye T., Leclercq M., Quéma V., Stefani J.-B., « The FRACTAL Component Model and Its Support in Java », *Software—Practice and Experience, Special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, vol. 36, n° 11, p. 1257-1284, September, 2006. `http://fractal.objectweb.org`.

Bucur O., Beaune P., Boissier O., « Définition et Représentation du Contexte pour des Agents Sensibles au Contexte », *Actes de la 2ème Conférence Francophone Mobilité et Ubiquité*, vol. 120 of *ACM International Conference Proceeding Series*, Grenoble (France), p. 13-16, June, 2005.

Capra L., Emmerich W., Mascolo C., « CARISMA : Context-Aware Reflective mIddleware System for Mobile Applications », *IEEE Transactions on Software Engineering*, vol. 29, n° 10, p. 929-945, October, 2003.

Cecchet E., Elmeleegy H., Layaïda O., Quéma V., « Implementing Probes for J2EE Cluster Monitoring », *Studia Informatica*, 2005.

Cecchet E., Marguerite J., Zwaenepoel W., « C-JDBC : Flexible Database Clustering Middleware », *Proc. of Freenix*, 2004.

Courtrai L., Guidec F., Le Sommer N., Mahéo Y., « Resource Management for Parallel Adaptive Components », *Proc. IEEE IPDPS Workshop on Java for Parallel and Distributed Computing*, Nice, France, p. 134-141, April, 2003.

Coutaz J., Crowley J., Dobson S., Garlan D., « The disappearing computer : Context is Key », *Communications of the ACM*, vol. 48, n° 3, p. 49-53, March, 2005.

Coutaz J., Rey G., « Foundations for a Theory of Contextors », *in* C. Kolski, J. Vanderdonckt (eds), *Proc. 4th International Conference on Computer-Aided Design of User Interfaces*, Kluwer, Valenciennes (France), p. 13-34, May, 2002.

da Rocha R., Endler M., « Evolutionary and Efficient Context Management in Heterogeneous Environments », *Proc. 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, Grenoble (France), November, 2005.

da Rocha R., Endler M., « Context Management in Heterogeneous, Evolving Ubiquitous Envrionments », *IEEE Distributed Systems Online*, April, 2006.

David P.-C., Développement de composants FRACTAL adaptatifs : un langage dédié à l'aspect d'adaptation, PhD thesis, Université de Nantes, École des Mines de Nantes (France), July, 2005.

David P., Ledoux T., « WildCAT : a generic framework for context-aware applications », *Proc. 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, Grenoble (France), p. 1-7, November, 2005. `http://wildcat.objectweb.org`.

Dey A., Salber D., Abowd G., « A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications », *Special issue on context-aware computing in the Human-Computer Interaction Journal*, vol. 16, n° 2–4, p. 97-166, 2001.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Gray P., Salber D., « Modelling and Using Sensed Context Information in the Design of Interactive Applications », *Proc. 8th IFIP International Conference on Engineering for Human Computer Interaction*, vol. 2254 of *Lecture Notes in Computer Science*, Toronto (Canada), p. 317-336, May, 2001.

Gurun S., Krintz C., Wolski R., « NWSLite : A Light-Weight Prediction Utility for Mobile Devices », *Proc. 2nd ACM/USENIX International Conference on Mobile Systems, Applications and Services*, Boston, USA, p. 2-11, June, 2004.

Henricksen K., Indulska J., McFadden T., Balasubramaniam S., « Middleware for Distributed Context-Aware Systems », *Proc. 7th International Symposium on Distributed Objects and Applications*, Lecture Notes in Computer Science, Springer-Verlag, Agia Napa (Cyprus), November, 2005.

Knop M., Schopf J., Dinda P., « Windows Performance Monitoring and Data Reduction Using WatchTower », *Proc. Workshop on Self-Healing, Adaptive and Self-Managed Systems*, New York, USA, June, 2002.

Lavirotte S., Lingrand D., Tigli J.-Y., « Définition du contexte : fonctions de coût et méthodes de sélection », *Actes de la 2ème Conférence Francophone Mobilité et Ubiquité*, vol. 120 of *ACM International Conference Proceeding Series*, Grenoble (France), p. 9-12, June, 2005.

Leclercq M., Quéma V., Stefani J.-B., « DREAM : a Component Framework for the Construction of Resource-Aware, Configurable MOMs », *IEEE Distributed Systems Online*, September, 2005. `http://dream.objectweb.org`.

Mansouri-Samani M., Sloman M., Monitoring Distributed Systems (A Survey), Research report no. doc92/23, University of London, Imperial College, London, England, September, 1992.

Mostefaoui G., Pasquier-Rocha J., Brézillon P., « Context-Aware Computing : A Guide for the Pervasive Computing Community », *IEEE/ACS International Conference on Pervasive Services*, Novi Sad (Serbia/Montenegro), p. 39-48, July, 2004.

Newberger A., Dey A., Designer Support for Context Monitoring and Control, Technical Report n° IRB-TR-03-017, Intel Research, Berkeley, USA, June, 2003.

Pascoe J., « Adding Generic Contextual Capabilities to Wearable Computers », *Proc. 2nd IEEE International Symposium on Wearable Computers*, Pittsburgh, Pennsylvania (USA), p. 92-99, October, 1998.

Preuveneers D., Berbers Y., « Adaptive Context Management Using a Component-Based Approach », *in* L. Kutvonen, N. Alonistioti (eds), *Proc. 5th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, vol. 3543 of *Lecture Notes in Computer Science*, Springer-Verlag, Athens (Greece), p. 14-26, June, 2005.

Rey G., Coutaz J., « Le Contexteur : Capture et distribution dynamique d'information contextuelle », *Actes de la 1ère Conférence Francophone Mobilité et Ubiquité*, vol. 64 of *ACM International Conference Proceeding Series*, Nice (France), p. 131-138, June, 2004.

Rouvoy R., Pessemier N., Pawlak R., Merle P., « Using Attribute-Oriented Programming to Leverage FRACTAL-based Developments », *Proc. 5th International ECOOP Workshop on* FRACTAL *Component Model*, Nantes (France), July, 2006.

Satyanarayanan M., « The Many Faces of Adaptation », *IEEE Pervasive Computing*p. 4-5, July, 2004.

Schroeder B., « On-Line Monitoring : A Tutorial », *IEEE Computer*p. 72-78, June, 1995.

Seinturier L., Pessemier N., Duchien L., Coupaye T., « A Component Model Engineered with Components and Aspects », *Proc. 9th International SIGSOFT Symposium on Component-Based Software Engineering*, vol. 4063 of *Lecture Notes in Computer Science*, Västeras, Sweden, p. 139-153, June, 2006.

Senart A., Cunningham R., Bouroche M., O'Connor N., Reynolds V., Cahill V., « MoCoA : Customisable Middleware for Context-Aware Mobile Applications », *Proc. 8th International Symposium on Distributed Objects and Applications*, vol. 4275 of *Lecture Notes in Computer Science*, Springer-Verlag, Montpellier (France), p. 1722-1738, November, 2006.

Smith C., Henry D., « High-performance linux cluster monitoring using Java », *Proc. 3rd Linux Cluster International Conference*, St. Petersburg, Florida, USA, October, 2002.

Temal L., Conan D., « Détections de défaillances, de connectivité et de déconnexions », *Actes de la 1ère Conférence Francophone Mobilité et Ubiquité*, vol. 64 of *ACM International Conference Proceeding Series*, Nice (France), p. 90-97, June, 2004.

Wolski R., « Dynamically forecasting network performance using the Network Weather Service », *Cluster Computing*, vol. 1, n° 1, p. 119-132, January, 1998.

Yau S., Karim F., Wang Y., Wang B., Gupta S., « Reconfigurable Context-Sensitive Middleware for Pervasive Computing », *IEEE Pervasive Computing*, vol. 1, n° 3, p. 33-40, July, 2002.

Article reçu le 21 novembre 2006
Accepté après révisions le 31 mai 2007

Denis Conan *est maître de conférences à l'Institut TELECOM, TELECOM & Management Sudparis. Ses domaines de recherche sont l'informatique mobile et les systèmes répartis. Ses travaux portent sur les intergiciels et l'algorithmique répartie, et plus particulièrement, la détection d'entraves pour la tolérance aux fautes, et les architectures et les canevas logiciels à base de composants, par exemple, pour la gestion de contexte en environnement ubiquitaire.*

Romain Rouvoy *est chercheur associé à l'Université d'Oslo. Ses travaux portent sur l'ingénierie des intergiciels et plus particulèrement sur les méthodes et techniques pour le développement d'intergiciels autonomes fiables.*

Lionel Seinturier *est professeur d'informatique à l'Université des Sciences et Technologies de Lille (USTL). Ses travaux portent sur l'ingénierie logicielle pour les intergiciels. Il s'intéresse plus particulièrement aux techniques à base de composants logiciels et d'aspects.*