Université d'Évry-Val-d'Essonne

Institut TELECOM, TELECOM & Management SudParis

Rapport de Stage
Master de Recherche en Informatique

Description de politiques de gestion de contexte

Léon Lim

Responsable du Master Recherche : Evripidis Bampis
Responsable de stage : Denis Conan

Juin 2008

Ce stage de Master 2 a été réalisé au sein du laboratoire CNRS UMR Samovar, Équipe ACMES du département **Informatique** de **TELECOM & Management SudParis**

Remerciements

Je tiens à remercier Monsieur Bruno Defude pour m'avoir accueilli dans le département Informatique et m'avoir permis d'effectuer ce stage dans de bonnes conditions.

Je remercie chaleureusement Monsieur Denis Conan, mon encadrant de stage, qui m'a aidé et soutenu pendant ces cinq mois de stage. Ses conseils avisés et sa patience m'ont donné la possibilité de dépasser les moments d'hésitation et d'embarras, et de rédiger le mémoire pour avoir les meilleurs résultats. Et c'est grâce à lui que j'ai retiré autant de satisfaction de ce stage. Je le remercie également pour sa constante disponibilité à mon égard et pour m'avoir donné la chance d'effectuer un stage aussi riche d'enseignements.

Je tiens à remercier les membres du département Informatique pour leur accueil chaleureux. Je remercie particulièrement Mesdames Sophie Chabridon et Chantal Taconet pour les conseils et les formations qu'elles m'ont données. Je remercie également Madame Brigitte Houassine, notre assistante de gestion toujours accueillante et chaleureuse. Mes vifs remerciements vont également à mes collègues, Mehdi Zaier, Zied Abid, Mahmoud Bouzid, Mounis Cherbira, Marie Buffat, Wided Ghardallou, Ines Ghargouri et tous mes autres chers collègues pour leur soutient et aide. Je remercie également les thésards pour leurs conseils et informations, en particulier je remercie Jérôme Sicard, Salah Salim Boutammine, Mohamed Sellami et Hamid Mukhtar.

Je me dois de remercier mes chers professeurs du master 2 à l'Université d'Évry ainsi que ceux de TELECOM & Management SudParis et ceux de l'IIE pour la qualité de leur enseignement.

Je tiens à remercier tous ceux qui m'ont apporté soutient et aide, et que je n'ai pas cité.

Je remercie enfin mon entourage, ma famille et mes amis pour m'avoir soutenu tout au long de ce stage.

Table des matières

Table des figures

Liste des tableaux

Introduction

Les applications ubiquitaires évoluent dans des contextes d'exécution toujours plus riches et changeants. Nous sommes de plus en plus entourés d'une grande quantité de dispositifs informatiques hétérogènes. La limitation des ressources, la distribution des applications, la mobilité des terminaux, la découverte des services et le déploiement automatique de logiciel sur ces terminaux rendent complexe et difficile la mise en place des applications ubiquitaires. Cette mise en place à grande échelle des applications ubiquitaires est un premier défi majeur que de nombreuses communautés de chercheurs essaient de relever en proposant des approches différentes.

Le second défi principal à relever consiste à rendre autonome l'adaptation des applications au contexte d'exécution avec garantie de cohérence et de performance. Les applications ubiquitaires doivent avoir conscience de l'environnement dans lequel elles s'exécutent, sur quel type de terminal elles s'exécutent, quel profil utilisateur est à prendre en compte pour la configuration, de quel type de réseau elles disposent pour la communication, à quel endroit se situe le terminal mobile, et quels dispositifs informatiques sont dans l'environnement. Toutes ces informations constituent le contexte d'exécution des applications et forment un nouveau type d'informations à traiter pour l'adaptation. Il n'existe actuellement pas de solution générale qui couvre complètement le processus de construction des applications ubiquitaires, de la conception à l'exploitation. Pour cela, parmi les services intergiciels (en anglais, *middleware*), l'observation du contexte joue un rôle prépondérant [13, 10, 5, 8]. En effet, sans cette observation, la description des changements de contexte est impossible, et sans cette description, les applications ne peuvent être ni conscientes ni sensibles à leur environnement. La description des changements de contexte peut être traitée en utilisant des politiques d'observation du contexte. La conception et l'implantation de ces politiques sont le fondement de toutes les applications ubiquitaires sensibles aux contextes.

Le travail de recherche présenté dans ce mémoire s'inscrit dans le cadre de l'intergiciel COSMOS [10]. L'originalité de COSMOS est l'expression de la composition d'informations de contexte dans une architecture logicielle à l'aide d'un langage de définition d'architecture logicielle (en anglais, *Architecture Description Language*, ADL) et la projection de cette architecture sur un graphe de composants logiciels. Cette approche facilite la définition et la gestion des politiques de gestion de contexte. Le modèle de composants et l'ADL qu'utilise COSMOS sont Fractal [6] et Fractal ADL [18]. L'inconvénient de l'approche utilisée par COSMOS est la complexité de l'écriture de la composition d'informations de contexte due à l'aspect très technique et prolixe de Fractal ADL et à la multiplicité des attributs de configuration. Avec Fractal ADL, les utilisateurs ont accès à des moyens de programmation très expressifs mais peu sûrs, du fait de la complexité et de la liberté d'écriture. Donc, l'expressivité du moyen de programmation doit être amoindri pour apporter la sûreté. C'est pourquoi mon travail de recherche pendant le stage se concentre sur l'étude d'une nouvelle approche consistant à proposer un langage dédié, du nom DSL COSMOS, au domaine de la composition d'informations de contexte. Ce nouveau langage a pour but d'offrir un formalisme plus accessible pour les utilisateurs non familiers à la syntaxe Fractal ADL. En effet, pour être utilisable, le gestionnaire de contexte doit être facile à

utiliser, il doit permettre d'atteindre le résultat prévu, et ce avec un effort moindre ou en un temps minimal. Ces exigences se traduisent par la nécessité de pouvoir composer les informations de contexte sans programmation de la part de l'utilisateur. Ces garanties doivent être assurées par des constructions de haut niveau qui abstraient la complexité des concepts sous-jacents de COSMOS.

Ce mémoire présente l'ensemble du travail et des résultats obtenus au cours du stage. Il est organisé comme suit. Le premier chapitre aborde les caractéristiques des applications ubiquitaires, les motivations et objectifs de COSMOS, la gestion des informations de contexte en particulier avec COSMOS, et les langages dédiés aux domaines. Le second chapitre présente l'état de l'art des langages de spécification de la gestion de contexte. Le troisième chapitre est consacré à la contribution au développement du langage COSMOS. Il présente le processus de développement que nous avons suivi pour le langage. Le dernier et quatrième chapitre conclut les résultats du travail de recherche effectué et présente nos perspectives.

Chapitre 1

Politiques de gestion de contexte dans COSMOS

Dans ce chapitre, nous commençons par présenter la gestion des informations de contexte avec COSMOS dans la section 1.1. Ensuite, la section 1.2 motive le développement d'un langage dédié à la composition d'informations de contexte.

1.1 Gestion des informations de contexte avec COSMOS

Les applications réparties évoluant dans des environnement ubiquitaires doivent continuellement gérer le contexte dans lequel elles s'exécutent afin de détecter les situations d'adaptation [11]. 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. Le canevas logiciel COSMOS (*COntext entitieS coMpositiOn and Sharing*) [10] permet de construire des gestionnaires de contexte. La gestion de contexte dans COSMOS est différente de celles présentes dans la littérature par son architecture et aussi par la manière dont la composition des informations de contexte s'effectue.

Nous présentons tout d'abord dans la section 1.1.1 les motivations et les objectifs d'un canevas logiciel dédié à la composition d'informations de contexte. Puis, la section 1.1.2 illustre l'architecture d'un gestionnaire de contexte. Ensuite, la section 1.1.3 présente l'architecture à base de composants qu'utilise COSMOS et la section 1.1.4 détaille le modèle de composant Fractal et le langage Fractal ADL. Enfin, la section 1.1.5 justifie le besoin de politiques de composition d'informations de contexte et la section 1.1.6 conclut sur la nécessité d'un langage dédié pour la composition d'informations de contexte.

1.1.1 Motivations et objectifs de COSMOS

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 [10]. Pour cela, COSMOS

est proposé pour la gestion des informations de contexte. COSMOS est un canevas logiciel orienté composant pour la gestion d'informations dans des applications sensibles au contexte. Il permet de construire des gestionnaires de contexte en environnements ubiquitaires. Les trois principes démontrés par COSMOS sont la séparation entre les activités de collecte et de synthèse des données de contexte, l'organisation des politiques de gestion de contexte en assemblages de composants logiciels et l'utilisation systématique de patrons de conception.

Dans notre étude, nous choisissons la définition suivante du terme contexte : « *un contexte est défini comme étant toute information caractérisant la situation d'une entité (une personne, une localisation, un objet, etc.) qui peut être considérée comme pertinente pour l'interaction de l'utilisateur avec son application* » [1]. Dans COSMOS, 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 entre ces entités dans le contexte et des relations entre ces entités. 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 unifiée, et si possible cohérente et complète, de l'environnement d'exécution, et de proposer aux applications des mécanismes de description et d'identification de situations d'adaptation.

Dans l'informatique mobile et ubiquitaire, les contextes d'exécution sont variés et dynamiques. En effet, le contexte n'est pas tant l'état d'un environnement prédéfini avec un ensemble fixe de ressources en interaction, mais une partie d'un processus en interaction avec un environnement dynamique composé de ressources configurables et mobiles [11]. Le traitement de tels types d'informations repose sur un gestionnaire ayant une architecture flexible et capable de définir de manière explicite des relations entre l'environnement ubiquitaire et l'adaptation.

1.1.2 Architecture de gestion de contexte

L'architecture globale du gestionnaire de contexte de COSMOS présentée dans la figure 1.1 est inspirée de [11, 14, 27]. C'est un système en couches dont chacune est responsable d'une fonctionnalité particulière. La couche la plus basse 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. Cette couche a pour rôle d'intégrer dans l'architecture globale du service de gestion de contexte les nombreux canevas logiciels utilisés pour les différentes sources d'informations de contexte. La couche de collecte fournit les données numériques brutes à la base du traitement par ce que les auteurs appellent les *processeurs de contexte*. Les données brutes sont des valeurs numériques observables, sont traitées et transformées par des processeurs de contexte en des valeurs symboliques. Des exemples de valeurs symboliques sont par exemple la qualité du lien WiFi et la localisation du terminal mobile. Les données symboliques constituent la perception du contexte. La couche la plus haute vers l'application identifie les situations d'adaptation. Cette couche d'identification de situations d'adaptation exprime la sensibilité au contexte de l'application. Cette sensibilité au contexte est gérée par les conteneurs ou membranes des composants métier. La dernière couche représentée verticalement à la droite du schéma illustre la gestion des ressources pour les traitements. Une notion de temps a été également ajoutée dans chaque couche, et par implication, une notion d'instant de collecte. Par conséquent, les différentes couches sont indépendantes quant à la gestion des ressources système (mémoire, activité) qu'elles consomment pour leur traitements. Ainsi, le système conçu à partir de cette architecture est faiblement couplé et plus facilement re-configurable.

En conclusion, le patron de conception « Architecture en couches » de COSMOS permet l'expression de la séparation des préoccupations, ici la séparation des fonctionnalités du gestionnaire

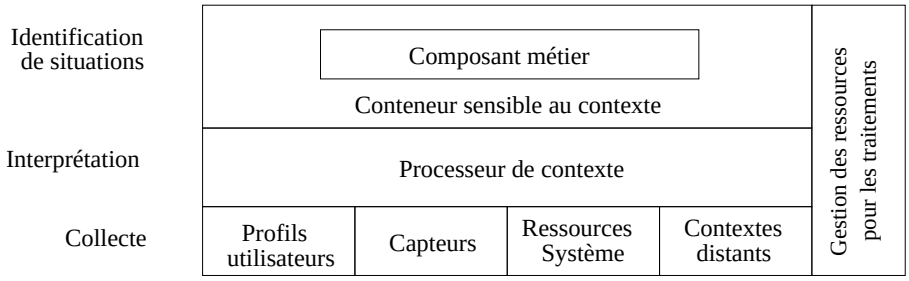

FIG. 1.1 – Architecture de gestion de contexte

de contexte. Cette flexibilité est possible grâce à une architecture à base de composants que nous présentons dans la section suivante.

1.1.3 Architecture à base de composants

COSMOS est conçu en appliquant les principes de base des intergiciels : le canevas logiciel est construit à partir d'éléments génériques, spécialisables et modulaires afin de composer plutôt que de programmer. Pour faire face à la diversité des informations de contexte, COSMOS repose sur les principes d'ingénierie logicielle à base de composants (en anglais, *Component-Based Software Engineering* [CBSE] *principles*) [24]. Szyperski définit un composant comme « *une unité de composition avec des interfaces contractuellement spécifiées et des dépendances explicites sur son contexte [terme pris dans son acception générale]. Un composant peut être déployé indépendamment et il est sujet à des compositions par des tiers* »[1] [24].

En particulier, le COSMOS combine les concepts de *composants*, d'*architecture logicielle* et de *patrons architecturaux*. L'approche orientée composant apporte une vision unifiée dans laquelle les mêmes concepts (composant, liaison, interface) sont utilisés pour développer les applications et les différents systèmes sous-jacents. Cette vision unifiée 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. 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. Ainsi, la reconfiguration et l'adaptation à des contextes d'utilisation nouveaux non prévus au départ sont facilitées.

Par conséquent, dans COSMOS, la composition d'informations de contexte est exprimée dans un langage de description d'architecture logicielle (ADL) et cette architecture est projetée sur un graphe de composants. Le modèle de composant et l'ADL utilisés sont FRACTAL [6, 7] et FRACTAL ADL [18] que nous présentons maintenant.

1.1.4 Modèle de composant logiciel Fractal et langage de description d'architecture logicielle Fractal ADL

Le modèle de composants FRACTAL est basé autour des concepts de composants, d'interfaces et de connecteurs (appelés liaisons, en anglais *bindings*). Ce modèle vise à mieux gérer (de ma-

[1] « *A component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* »

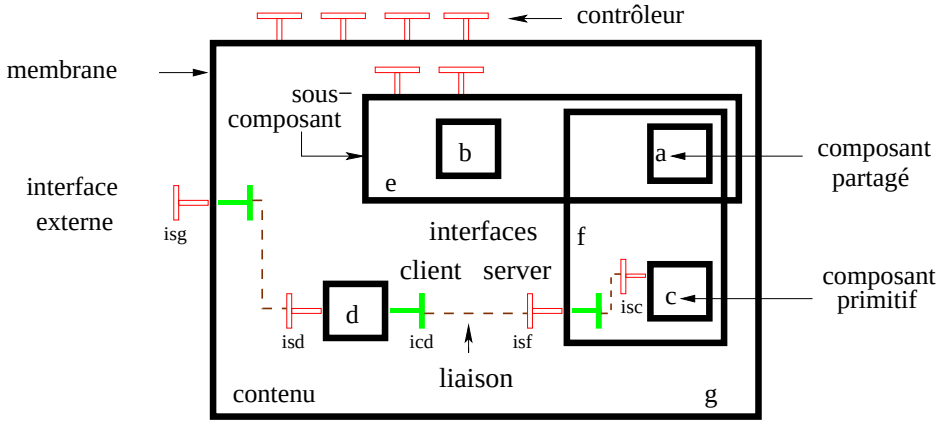

FIG. 1.2 – Exemple de composants Fractal

nière plus efficace) la composition des composants et la dynamicité de cette composition. En particulier, le modèle de composant FRACTAL permet la définition, la configuration et la reconfiguration dynamique de composants, et offre une séparation claire entre les parties fonctionnelles et extrafonctionnelles d'une application. C'est un modèle de haut niveau qui inclut la possibilité de partage dans une hiérarchie de composants. Les concepts important du modèle de composant FRACTAL sont les suivants :

- récursivité : un composant est de type primitif ou composite. Dans ce dernier cas, le composant correspond à un assemblage d'autres composants primitifs ou composites ;
- réflexivité : les composants possèdent des capacités d'introspection et d'intercession (c'est-à-dire, l'accès et la manipulation du contenu d'un composant et le contrôle de son cycle de vie) ;
- partage de composant : un composant peut être inclus (ou partagé) par plusieurs composants ;
- liaison entre composants : une liaison FRACTAL représente une connexion entre deux composants (liaison primitive). Les liaisons de type multiples (liaisons composites) sont autorisées. De ce fait, le type d'une interface serveur doit être un sous-type de l'interface cliente à laquelle elle est reliée. Les liaisons peuvent implanter n'importe quelle sémantique de communication ;
- ouverture : les services extrafonctionnels associés à un composant peuvent être personnalisés avec la notion de contrôle de la membrane. Une membrane est constituée d'un ensemble de contrôleurs ouverts et extensibles. Chaque contrôleur est associé à une interface et est responsable d'un service particulier.

Les interfaces jouent un rôle central dans le modèle de composant FRACTAL. Il en existe deux catégories : les interfaces fonctionnelles et les interfaces de contrôle. Les interfaces fonctionnelles sont les points d'accès externes d'un composant. Fractal fournit des interfaces cliente et serveur. Une interface cliente reçoit des opérations d'invocation et une interface serveur en émet. Les interfaces de contôle sont responsables de la gestion des besoins extrafonctionnels du composant : la gestion du cycle de vie ou des liaisons du composant. Un composant primitif Fractal est implanté en tant que classe Java dans Julia [6]. Julia est l'implantation de référence Java du modèle de composant FRACTAL.

La figure 1.2 représente un exemple de composant Fractal. Les composants sont représentés par des rectangles. Nous pouvons distinguer ici :

- un composant composite « racine » f et son contenu,
- des sous-composants e, f, etc.,
- des composants primitifs a, b, c et d,
- un composant primitif a partagé par plusieurs composants composites (à plusieurs niveaux)
- des membranes (les contours en graisse forte),
- des interfaces externes (les T creux) qui sont des interfaces de contrôle,
- des interfaces internes (les T pleins),
- des interfaces clientes se trouvant à la droite des composants,
- des interfaces serveurs se trouvant à la gauche des composants,
- des liaisons (les lignes pointillées) entre les interfaces clientes et serveurs.

Le langage de description d'architecture Fractal ADL est le langage de base, ouvert et extensible de composition de composants Fractal. À l'aide de XML, le langage permet de définir des assemblages de composants Fractal. Le DTD (en anglais, *Document Type Definition*) de ce langage peut être étendu pour prendre en compte des propriétés extrafonctionnelles. Fractal ADL est conçu à partir d'un ensemble de modules ADL ouvert et extensible : chaque module définit une syntaxe abstraite pour un « aspect » architectural donné (tel que les interfaces, les liaisons, les attributs, ou les relations de partage). Les utilisateurs sont libres de définir leurs propres modules pour leurs propres aspects.

La figure 1.3 est une description d'architecture en Fractal ADL du composant Fractal illustré dans la figure 1.2. Dans cette description, nous notons que le composant composite « racine » est une définition (definition name="f"), et que ce composite contient trois sous-composants directs qui sont d, e et f. Les composants a, b, c, et d sont des composants primitifs (<controller desc="membranePrimitive">). Les composants e et f sont des composants composites car ils contiennent d'autres composants. Dans cet exemple, trois liaisons sont définies : la liaison entre l'interface cliente de g et l'interface serveur de d, la liaison entre l'interface cliente de d et l'interface serveur de f, et la liaison entre l'interface cliente de f et l'interface serveur de c. Nous pouvons également remarquer qu'une liaison entre l'interface interne du composant englobant et l'interface serveur du sous-composant se fait de manière implicite dans l'implantation Julia.

En conclusion, la base du développement Fractal réside dans l'écriture de composants et de liaisons permettant aux composants de communiquer. Le langage Fractal ADL constitue le vecteur privilégié pour la composition de composants.

1.1.5 Nœud de contexte et politique de composition d'informations de contexte

Dans cette section, nous présentons la composition d'informations de contexte avec COSMOS. Tout d'abord, nous abordons le concept de nœud de contexte. Ensuite, nous étudions quelles sont les propriétés d'un nœud de contexte. Enfin, nous concluons en justifiant le besoin d'une nouvelle approche pour spécifier les politiques de composition d'informations de contexte.

Le *nœud de contexte* est un concept de base de COSMOS. Un nœud de contexte est une information de contexte modélisée par un composant. En effet, COSMOS applique l'approche composant pour concevoir les nœuds de contexte. Ainsi, la définition d'un nœud de contexte est directement inspirée de celle de composant logiciel donnée par Szyperski [24]. Un nœud de contexte est une unité de composition avec des interfaces serveurs contractuellement spécifiées et des dépendances explicites vers d'autres nœuds de contexte. Un nœud de contexte peut être déployé indépendamment et il est sujet à des compositions par des tiers.

```
<definition name="f">
   <interface name="isg" role="server" signature="java.lang.Runnable" />
   <component name ="e" >
      <component name="b">
      ...
      </component>
      <component name="a">
      ...
      </component>
   </component>
   <compoent name="f">
      <interface name="isf" role="server" signature="java.lang.Runnable" />
      <component name="a" definition="./e/a"/>
      <component name="c">
         <interface name="isc" role="server" signaure="java.langa.Runnalbe" />
         <content class="org.objectweb.julia.Exemple.MyclassImp" />
         <cotroller desc="membranePrimitive">
      </component>
   </component>
   <component name"d">
      <interface name="isd" role="server" signaure="java.langa.Runnalbe" />
      <interface name="icd" role="client" signaure="Service" />
      <content class="org.objectweb.julia.Exemple.MyclassImp" />
      <cotroller desc="membranePrimitive">
   </component>
   <biding client="this.isg" server="d.isd" />
   <biding client="d.icd" server="f.isf" />
   <biding client="f.isf" server="c.isc" />
   <controller desc="membraneComposite" />
</definition>
```

FIG. 1.3 – FRACTAL ADL : description en FRACTAL ADL du composant illustré dans la figure 1.2

Les nœuds de contexte sont organisés en une hiérarchie avec possibilité de partage pour former les politiques de gestion de contexte. La politique de gestion de contexte est définie à partir de la définition d'architecture logicielle[2] donnée par Bass, Clements et Kazman [2]. Une politique de gestion de contexte est une partie de la structure ou la structure d'un graphe de nœuds contexte qui inclut les nœuds de contexte, les propriétés de ces nœuds et les relations entre ces nœuds. Une forêt représente l'ensemble des politiques de gestion de contexte utilisées par les applications clientes du gestionnaire de contexte. Les relations entre les nœuds sont des relations d'encapsulation et de partage. 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 comme la mémoire vive, la qualité du lien WiFi. Leur rôle est d'isoler les inférences de contexte de plus haut niveau, qui deviennent donc indépendants du canevas logiciel utilisé pour la collecte des données brutes.

Architecture de nœud de contexte Chaque nœud de contexte étend le nœud abstrait ContextNode (le composant composite de base) illustré dans la figure 1.4 [10, 23]. En plus des attributs avec des valeurs par défaut, le nœud de contexte abstrait possède un opérateur (un composant primitif abstrait appelé ContextOperator), un gestionnaire de messages (Message Manager) et un gestionnaire d'activités (Activity Manager). L'opérateur ContextOperator est responsable des opérations d'inférence. Il prend en entrée les données obtenues par observation ou suite à une notification, effectue un traitement et transfert les résultats du traitement vers la sortie par réponse

[2] *« A software architecture of a program or computingsystem is the structure or structure of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. »*

à l'observation ou à la notification. Le gestionnaire d'activités fournit le mécanisme nécessaire pour gérer des composant actifs. Le gestionnaire de messages est en charge du traitement des rapports d'observation et de notification qui sont envoyés ou reçus par le composant via les interfaces Pull et Push.

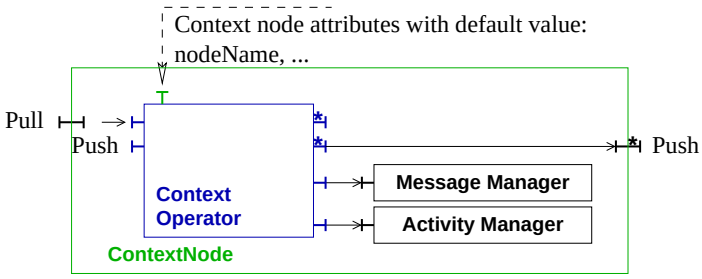

FIG. 1.4 – Architecture de nœud de contexte abstrait

Les nœuds de contexte possèdent des propriétés qui définissent leur comportement :

- *passif ou actif* : un nœud passif est utilisé par des activités extérieures au nœud qui l'interrogent pour obtenir une information. Un nœud actif est équipé d'une activité pour exécuter une tâche donnée ;
- *observation ou notification* : les communications peuvent s'opérer du bas vers le haut ou du haut vers le bas de l'arborescence de nœuds de contexte. Les notifications correspondent aux messages envoyés par les nœuds à leurs parents (du bas vers le haut), tandis que les observations sont déclenchées par un nœud parent. Un nœud de l'arborescence peut être à la fois observateur et notificateur actifs ;
- *passant ou bloquant* : un nœud passant propage les observations et les notifications. 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 ;
- *nommage* : les noms des nœuds de contexte sont uniques pour permettre les parcours dans le graphe et les configurations.

Un nœud récupère des informations de contexte de nœuds enfants de la hiérarchie et infère une information de plus haut niveau d'abstraction. Le traitement correspondant est effectué dans l'opérateur de contexte. Le cycle de vie des nœuds de contexte enfants est contrôlé par les nœuds de contexte parents. Pour la gestion des activités, les nœuds de contexte actifs enregistrent leurs activités auprès d'un gestionnaire d'activités. Ainsi, le gestionnaire d'activités, lui-même paramétrable, peut créer une activité par traitement (observation ou notification) ou bien une activité par nœud de contexte ou encore une activité pour tout ou partie de la hiérarchie. Pour la consommation mémoire, un gestionnaire de messages gère des réserves de messages et autorise des duplications *par référence* ou *par valeur*.

1.1.6 Conclusion

Dans COSMOS, les politiques sont décomposées en unités à grain fin appelées *nœuds de contexte* qui sont des composants logiciels. Les politiques de gestion de contexte ont été introduites dans COSMOS afin d'identifier les situations d'adaptation pour lesquelles une réaction de l'application est attendue. L'utilisation de FRACTAL et FRACTAL ADL pour la réalisation de

COSMOS autorise une conception avec des composants à granularité fine et par conséquent une configuration très fine. La limite de l'approche est la complexité de l'écriture des politiques de gestion de contexte correspondant à des hiérarchies de composants avec partage. Cette difficulté est due, d'une part, à l'aspect très technique et prolixe de FRACTAL ADL, et d'autre part, à la multiplicité des méta-données disponibles pour configurer très finement les traitements d'inférence. Ceci est un frein important à la spécification pour les concepteurs métier, non spécialistes du domaine de la gestion de contexte. Nous proposons donc d'étudier une nouvelle approche permettant de rendre déclarative la définition de l'observation par le composant. Le langage employé doit être simple, donc dédié au domaine, afin d'envisager par la suite d'ajouter une analyse de cohérence des politiques de gestion de contexte obtenues.

1.2 Langages dédiés aux domaines

Van Deursen, Klint et Visser donnent la définition suivante d'un langage dédié à un domaine (en anglais, *Domain Specific Language* ou DSL) : « *un langage dédié est un langage de programmation ou un langage de spécification exécutable qui offre, grâce à des notations et abstractions appropriées, un pouvoir d'expression concentré sur, et généralement limité à, un domaine d'application particulier* »[3] [25].

Les langages dédiés sont des langages adaptés à un domaine ou à une famille d'applications. Contrairement aux langages de programmation généralistes, ils offrent un haut niveau d'abstraction sur le domaine considéré tel que définis dans [20]. En effet, un langage de programmation généraliste permet de résoudre un vaste ensemble de problèmes, tandis qu'une solution dédiée se focalise sur un domaine restreint. Du point de vue l'utilisateur, le principal intérêt d'un langage dédié est de fournir des abstractions et des notations de plus haut niveau concernant le domaine étudié. Le DSL est un *petit langage* ou *micro langage*, souvent plus déclaratif qu'impératif. Grâce à ses constructions de haut niveau, le DSL permet aux experts du domaine ou aux utilisateurs concernés par le domaine (et non aux experts en programmation) de développer les programmes qui les intéressent assez rapidement et efficacement. Les langages dédiés sont moins expressifs que les langages généralistes, mais la réduction de l'expressivité ne nuit pas à la résolution des problèmes inhérents au domaine. À titre d'exemples, LaTeX est un langage de description et de préparation de document. HTML est un langage dit de « marquage » pour écrire un document avec des balises de formatage. VHDL (*VHSIC Hardware Description Language*) est un langage formel pour la spécification de structures électroniques, aussi bien au niveau comportemental que structurel.

1.2.1 Motivations pour le développement d'un langage dédié

Les avantages des langages dédiés par rapport aux langages généralistes sont identifiés dans [12, 20]. Ces avantages sont les suivants :

- analyses statiques : le pouvoir d'expression généralement limité du langage dédié permet d'offrir des garanties et des analyses statiques plus poussées. Par exemple, le langage d'interrogation de base de données SQL n'est pas récursif, ce qui garantit que les requêtes SQL terminent toujours ;
- simplicité d'utilisation et extensibilité : l'utilisation d'un niveau d'abstraction et de notations spécifiques au problème rend le code beaucoup plus lisible et concis. La définition

[3] « *A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.* »

d'une grammaire basée sur la syntaxe BNF (en anglais, *Backus-Naur Form*), ou son extension EBNF, est plus simple à écrire, à comprendre et surtout à faire évoluer ;
- développement rapide : le développement de programmes à l'aide de DSL est généralement beaucoup plus rapide qu'avec un langage généraliste et la correction des programmes est plus facilement vérifiée. En effet, ces langages sont souvent plus déclaratifs qu'impératifs et leur niveau d'abstraction élimine un grand nombre d'erreurs courantes dans les langages de plus bas niveau ;
- réutilisabilité : les concepts manipulés par un langage dédié correspondent aux connaissances des experts dans ce domaine et les rendent plus facilement réutilisables par les programmeurs ou simples utilisateurs ;
- performance : les programmes écrits dans un langage dédié peuvent souvent avoir de meilleures performances que ceux écrits dans des langages généralistes, car les implantations de ces langages peuvent utiliser les meilleurs algorithmes connus et tirer partie d'optimisations spécifiques au domaine.

Ainsi, l'utilisation des langages dédiés pour résoudre des problèmes dans un domaine particulier présentent de nombreux avantages. Cependant, utiliser des DSL comporte aussi un certain nombre de désavantages :

- les DSL sont difficiles à concevoir et à implanter car ils requièrent une double expertise, l'une du domaine concerné et l'autre des langages de programmation ;
- la conception d'un DSL peut être très difficile car il n'est pas simple de trouver le bon niveau d'abstraction et de généralité. Un langage trop généraliste risque de ne pas tirer partie de tous les avantages potentiels des DSL. En revanche, un langage trop restrictif risque de ne pas être utilisé s'il n'inclut pas toutes les constructions nécessaires, c'est-à-dire ne reflètant pas assez bien les concepts du domaine ;
- les futurs utilisateurs doivent être formés à l'utilisation du DSL. L'apprentissage d'un nouveau langage, surtout dédié, peut être plus difficile que celui d'un langage généraliste déjà connu. En effet, la difficulté ne réside pas dans le langage mais plutôt dans les moyens d'apprentissage mis à disposition des utilisateurs potentiels comme la documentation.

Ces inconvénients rendent difficile la décision de développer un nouveau langage dédié pour un domaine. Le processus de développement avec une méthodologie bien déterminée permet d'atténuer ces désavantages et d'arriver finalement au résultat attendu. Pour cela, le développement du langage dédié doit suivre un processus de développement rigoureux. Nous décrivons maintenant le processus général de développement d'un langage dédié.

1.2.2 Processus de développement d'un langage dédié

Le processus de développement d'un DSL peut s'avérer très complexe. Il requiert une expertise du domaine concerné, mais aussi une connaissance dans le domaine de la conception des langages de programmation. Pour aider les concepteurs/développeurs, [20] identifie les différentes phases de développement : décision, analyse, conception, implantation et déploiement. La décision répond à la question « pourquoi et quand » développer et les autres phases correspondent au « comment développer ». En pratique, le développement d'un DSL n'est un pas un simple processus séquentiel mais un processus itératif. Nous suivons maintenant ces phases de développement.

Décision L'objectif principal de nos travaux est de fournir un formalisme plus accessible pour la gestion des politiques de gestion de contexte. Étant donnés les avantages des DSL décrits dans la section 1.2.1, nous pensons que le développement d'un langage dédié à la compostion d'informations de contexte est en adéquation avec cet objectif.

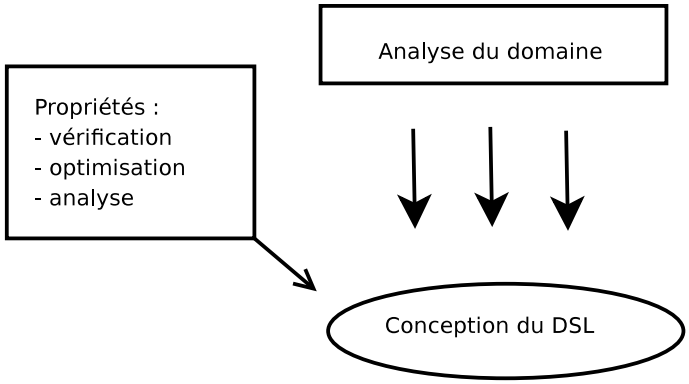

FIG. 1.5 – Conception d'un DSL

Analyse de domaine L'analyse de domaine permet de mieux capturer les différents concepts sous-jacents, et cela grâce la connaissance des experts du domaine. Plusieurs méthodologies peuvent être utilisées pour faire l'analyse de domaine [20]. Elles sont classées en deux catégories : formelle et informelle. La plupart des DSL sont conçus en suivant une méthodologie informelle, par exemple en écrivant une spécification à partir d'exemples de codes. La conception formelle consiste à écrire explicitement la spécification en utilisant des méthodes de définition sémantique. Nous pouvons citer par exemple FODA [17]. La conception formelle de la syntaxe et de la sémantique permet de faire apparaître des problèmes sous-jacents (par exemple de cohérence) et de résoudre ces problèmes avant la conception du langage. L'importance de la syntaxe du langage dédié ne doit pas être sous-estimée et elle doit être conforme le plus possible à des notations du domaine.

Conception La figure 1.5 représente un cadre de conception du DSL à partir de données d'entrée fournies par l'analyse de domaine. Des données d'entrée sont des informations qui sont considérées comme pertinentes au domaine : idiomes, propriétés, terminologies, objets et relations entre ces objets. La conception du DSL doit prendre en compte certaines propriétés du domaine telles que la vérification, l'optimisation et l'analyse.

Plusieurs approches de conception d'un DSL peuvent être comparées [20]. La première est de se baser sur une partie d'un langage existant. Les avantages sont alors une implantation facilité et une plus grande familiarité de programmation. Une autre approche consiste à prendre un langage existant et à l'étendre avec les nouveaux concepts du domaine. La difficulté réside alors au niveau de l'intégration de ces nouvelles caractéristiques spécifiques au domaine dans le langage. De plus, trouver un langage existant adéquat n'est pas une tâche aisée. La dernière approche consiste à développer un nouveau langage dédié en partant de rien indépendamment des langages existants.

Implantation L'implantation d'un langage dédié est techniquement similaire à celui d'un langage généraliste. Les DSL sont implantés en tant qu'interprètes, compilateurs ou techniques de génération de code. L'approche la plus appropriée doit être choisie pour implanter le DSL spécifié. Par exemple, pour un DSL possédant les propriétés telles que l'analyse, la vérification, l'optimisation, la transformation et l'implantation en tant que compilateur ou interprète est conseillé [20].

1.2.3 Conclusion

Un DSL présente de nombreux avantages mais comporte également un certain nombre de désavantages. Pour tirer profit des avantages d'un DSL et en même temps minimiser le plus possible les désavantages de ce dernier, le développement doit suivre un processus rigoureux : de la décision à l'implantation en passant par l'analyse et la conception. Les méthodes et techniques dans chaque phase du processus de développement doivent être concises et bien déterminées afin de créer un DSL performant et adapté au domaine.

Chapitre 2

État de l'art sur les langages de composition d'informations de contexte

Au chapitre précédent, la réutilisation et la composition d'informations de contexte par l'approche composant ont été exposées. Cette étude commence par l'étude de l'état de l'art des langages de composition d'informations de contexte.

Dans ce chapitre et pour concevoir un DSL efficace, nous étudions en premier lieu les différents critères d'évaluation des solutions de la littérature. Nous présentons dans la section 2.1 les critères généraux et ensuite les critères spécifiques au domaine de la composition d'informations de contexte. Après l'étude de ces différents critères, nous établissons dans la section 2.2 une classification des langages dédiés au domaine de la gestion des informations de contexte. Ensuite, nous détaillons dans la section 2.3 certains langages de description des informations de contexte et analysons dans la section 2.4 les langages d'expressions d'observation. Nous comparons dans la section 2.5 différents langages de spécification des informations de contexte selon l'angle de la composition. Enfin, avant de conclure, nous dressons en section 2.6 un tableau récapitulatif des solutions étudiées dans ce chapitre.

2.1 Critères d'évaluation des solutions de la littérature

Adopter l'approche DSL pour résoudre des problèmes intrinsèques à un domaine particulier implique des risques et des avantages. Les avantages d'un DSL découlent directement de sa conception et de son développement. Comme nous avons vu précédemment, les risques sont dus à la difficulté de conception, au manque d'utilisateurs experts, etc. Un DSL bien conçu est le résultat d'un compromis entre ces risques et ces avantages. Nous essayons d'optimiser ce compromis avantage-risque. Pour cela, nous définissons les exigences d'un DSL pour COSMOS qui correspondent à des critères généraux et spécifiques.

2.1.1 Critères généraux

Les critères généraux identifiés pour évaluer un DSL sont les suivants, certains d'entre-eux étant repris de [20] :

– facilité d'utilisation : les DSL ne doivent pas être ciblés vers des experts en programmation mais vers des experts du domaine ou au moins des utilisateurs concernés par le domaine. Par ses constructions, le DSL doit améliorer la lisibilité et la concision des programmes mais surtout être facilement accessible. L'accessibilité se traduit, d'une part, par une structure simple du langage, et d'autre part, par les moyens mis à la disposition des utilisateurs potentiels (une documentation complète du langage). Le DSL doit permettre aux utilisateurs de savoir rapidement ce que fait le programme, savoir comment ne les intéressent pas forcément. Le DSL doit utiliser les idiomes et le niveau d'abstraction du domaine ciblé ;
– extensibilité : en plus d'être simple, la structure du DSL doit être modulaire afin de favoriser l'extensibilité ;
– analysabilité : la sémantique du DSL doit être restreinte, donc facilement analysable. Cette restriction doit permettre de rendre certaines propriétés décidables. La décidabilité permet la vérification et l'analyse de cohérence ;
– réutilisation systématique : au delà de constructions et notations appropriées qui amènent à certains idiomes, le DSL peut être considéré comme un modèle tant au niveau des concepts qu'au niveau de la programmation. Dans ce sens, le DSL est un métamodèle du domaine. Ainsi, il doit permettre de manipuler les concepts du domaine et de décomposer les problèmes en sous-problèmes. Par définition, les idiomes du DSL doivent pouvoir être réutilisés : réutilisation du code source s'il est implanté en tant que bibliothèque, réutilisation de concepts du domaine et enfin réutilisation d'utilisation par extension pour un autre langage du domaine. Dans notre cadre d'étude, l'ordre de propriété de réutilisation est le suivant : réutilisation de concepts, réutilisation d'utilisation par extension et puis réutilisation en tant que bibliothèque ;
– sûreté : un programme écrit dans un DSL est potentiellement exposé aux erreurs. Néanmoins, le fait qu'il soit restreint à un domaine doit permettre d'assurer une certaine sûreté. La restriction à un domaine permet de vérifier les propriétés connues que doit satisfaire le programme. Cette vérification est facilitée car le DSL possède des abstractions appropriées. Nous pouvons considérer que la réduction des possibilités de programmation permet de réduire les erreurs sous-jacentes de type syntaxique ainsi que sémantique ;
– testabilité : les DSL sont restreints aux domaines, possèdent une structure légère, et doivent donc favoriser la testabilité des programmes ;
– optimisation : les algorithmes utilisés pour implanter le compilateur du langage dédié doivent permettent au compilateur d'être performant.

2.1.2 Critères spécifiques au domaine

Des politiques de gestion de contexte dans COSMOS correspondent à des hiérarchies de composants avec partage. Les domaines qui nous intéressent dans la gestion de politiques de contexte sont la composition d'informations de contexte et le partage de ces informations. Ainsi, les critères spécifiques de notre domaine sont la composabilité et le partage.

– composabilité : le DSL doit faciliter la définition de composants logiciels pour permettre la définition des politiques de gestion de contexte. La structure du langage doit être modulaire (par exemple, une expression est composée d'autres expressions plus élémentaires). Chaque expression correspond soit à une définition d'un concept « élémentaire » du domaine soit à une expression composée. La composition doit se retrouver au niveau de la structure du langage. Ainsi, plutôt que de programmer, nous composons les informations de contexte de façon déclarative ;
– partage : les expressions du DSL doivent permettre d'exprimer le partage des informations de contexte. En particulier, le partage de nœuds d contexte qui correspond à la possibilité

de partage ou d'utilisation d'une partie de politique de gestion de contexte par plusieurs politiques. Pour cela, le DSL doit permettre les définitions par référence et l'accès aux références doit être explicite en définissant par exemple des espaces de nommage.

2.2 Classification des langages de gestion des informations de contexte

Nous avons vu dans la section 1.1.1 la définition du terme *contexte* sous l'angle conceptuel. Winograd donne une acception en terme de langage. Un contexte est un modèle de communication, peut être exprimé par une expression d'un langage et n'est effectif que s'il est partagé par tous les acteurs dans l'environnement [26].

Il existe dans la littérature deux grandes familles de styles architecturaux d'un gestionnaire de contexte pour gérer le traitement et le partage des informations qui sont l'orientation donnée et l'orientation processus. Dans l'orientation donnée, la gestion des informations de contexte est centralisée dans une base de données accessible comme un tableau noir ou dans une mémoire virtuelle partagée gérant l'espace comme un ensemble d'enregistrements. Les sources de contexte ou les collecteurs publient leurs informations sans considération de destination (comme le mode de diffusion *broadcast* dans le réseau IP). Les processeurs de contexte et les applications accèdent aux informations de contexte et opèrent un traitement sur ces informations grâce à des requêtes. Contraiment à l'orientation donnée, l'orientation processus encapsule toutes les acquisitions et transformations d'informations de contexte dans des entités de traitement séparées appelées selon le paradigme de conception « objet », « processus », « composant » ou « service ». La gestion de contexte correspond alors à la manipulation de graphes, par exemple, de composants.

Nous classons les langages dédiés au domaine de la gestion des informations de contexte en deux catégories telles qu'illustrées dans la figure 2.1 : les langages sans composition et les langages avec composition. Dans la première catégorie (sans composition), nous avons des langages pour la description des informations brutes et des langages de description de la collecte. Dans la seconde catégorie (avec composition), nous trouvons des langages de requête utilisés dans l'orientation donnée et des langages de composition utilisés dans l'orientation processus.

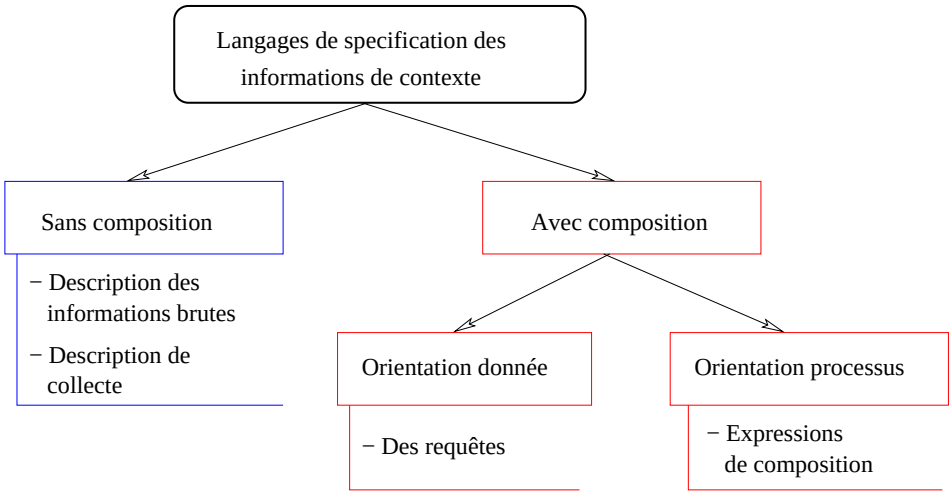

FIG. 2.1 – Classification des langages dédiés à la gestion de contexte

2.3 Langages de description des informations de contexte

Dans un environnement ubiquitaire, les informations peuvent venir de diverses sources. Dès lors, il peut y avoir différents modèles possibles pour ces informations. Nous étudions divers langages de description d'informations de contexte afin de mieux comprendre quelles sont les représentations possibles. Nous présentons tout d'abord un langage de modélisation de capteurs SensorML dans la section 2.3.1. Ensuite, nous caractérisons la famille de langages basés sur RDF dans la section 2.3.2.

2.3.1 SensorML

SensorML (*Sensor Model Language*) [4] est défini par le consortium OCG (*Open Geospatial Consortium*) pour l'exploitation de capteurs Web (en anglais, *Sensor Web Enablement* ou SWE). SensorML est un langage de modélisation avec un encodage XML pour les capteurs. L'architecture de SensorML permet de supporter un nombre important de capteurs aussi bien dans des plates-formes stationnaires que dans des plates-formes mobiles. SensorML est utilisé pour définir les capacités, la géolocalisation et les interfaces d'un capteur.

SensorML modélise les composants dans les systèmes de capteurs comme une collection de processus physiques ou logiques. Les composants sont des composants matériels et logiques : les composants matériels comme les transmetteurs, les actionneurs, les capteurs et les composants logiques comme les unités de calcul. Comme le montre la figure 2.2, une instance d'un processus dans SensorML permet de décrire les entrées et les sorties attendues ainsi que les paramètres et les méthodes requis permettant de fournir ces valeurs de sortie à partir des valeurs en entrée.

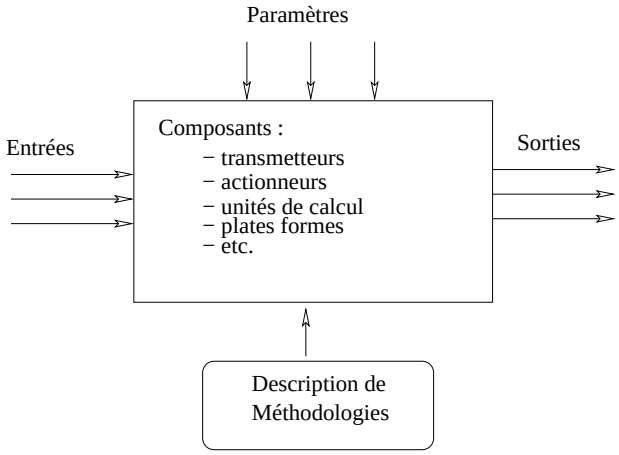

FIG. 2.2 – SensorML : instance de processus

Nommage SensorML utilise la syntaxe Xlink[1] pour gérer les espaces de nommage. XLink est une spécification du W3C qui permet de définir les liens entre les fichiers XML ou portions de fichiers XML grâce à XPointer. La figure 2.3[2] illustre un modèle de processus de thermomètre utilisé dans une station météo. Un modèle de processus définit un processus atomique pouvant

[1] http://www.w3.org/TR/xlink/ ou http://www.yoyodesign.org/doc/w3c/xlink/
[2] Exemple extrait de http://www.s-ten.eu/deliverables/D2.1/document_files/sensorml.htm

être exécuté par une méthode. Comme dans tout modèle de processus, des entrée et des sorties sont spécifiées qui sont ici respectivement la température et la résistance. Les deux paramètres sont steadyStateResponse (réponse d'état de suivi) et accuracy (précision) dont la description est volontairement omise. En particulier, dans cet exemple, la méthode transducer du modèle de processus est définie ailleurs et la référence utilise la syntaxe d'association `xlink:href`. Cette syntaxe est similaire celle des URI (*Uniform Resource Identifier*) utilisée pour relier des liens entre fichiers HTML. L'une des particularités intéressantes de Xlink par rapport à HTML est que les liens Xlink sont mutidirectionnels. Dans HTML, les liens sont mono-directionnels et la cible n'a aucune connaissance sur la source[3]. Ce qui nous intéresse dans Xlink est la définition explicite des relations entre des ressources. En effet, elle permet, d'une part, de réutiliser une méthode ou un paramètre en mettant la référence (*typage faible*) au lieu de définir explicitement (*typage fort*) la méthode ou le paramètre en question, d'autre part, de simplifier l'analyse de cohérence (par exemple, existence de définition ou de document) et le partage des informations grâce à des relations explicites entres des ressources.

```
<ProcessModel ... >
 <referenceFrame>
 </referenceFrame>
 <inputs>
  <InputList>
   <input name="temperature">
    <swe:Quantity definition="urn:ogc:def:phenomenon:ogc:1.0.3:temperature"/>
   </input>
  </InputList>
 </inputs>
 <outputs>
  <OutputList>
   <output name="measuredTemperature">
    <swe:Quantity definition="urn:ogc:def:phenomenon:ogc:1.0.3:resistance"/>
   </output>
  </OutputList>
 </outputs>
 <parameters>
  <ParameterList>
   <steadyStateResponse>
   </steadyStateResponse>
   <accuracy>
   </accuracy>
  </ParameterList>
 </parameters>
 <method xlink:href="urn:ogc:def:process:ogc:1.0.3:transducer"/>
</ProcessModel>
 <inputs>
  <InputList>
   <input name="physicalPhenomena">
    <swe:DataGroup>
     <swe:component name="atmosphericTemperature">
      <swe:Quantity definition="urn:ogc:def:phenomenon:ogc:1.0.3:temperature"/>
     </swe:component>
     <swe:component name="wind">
      <swe:Quantity definition="urn:ogc:def:phenomenon:ogc:1.0.3:wind"/>
     </swe:component>
    </swe:DataGroup>
   </input>
  </InputList>
 </inputs>
```

FIG. 2.3 – SensorML : modèle de processus

[3] « Un lien XLink est une relation explicite entre des ressources ou portions de ces ressources. Cette relation est rendue explicite par un élément de liaison Xlink, qui est un élément XHTML conforme à XLink, confirmant l'existence du lien. »

Regroupement de données Dans SensorML, un processus composite est un regroupement de processus de base. Le regroupement se retrouve également au niveau des données. Ainsi, comme le montre la figure 2.3, la température et le vent sont définis dans un groupe `<swe:DataGroup>`. Le regroupement de données présente plusieurs avantages dont le principal est une meilleur lisibilité des données disponibles.

2.3.2 Famille de langages basés sur RDF

RDF (en anglais, *Resource Description Framework*)[4] est un modèle de graphe destiné à décrire les ressources Web et leurs métadonnées de façon à permettre le traitement automatique des descriptions. Standardisé par le W3C, RDF est le langage de base du Web sémantique. L'un des objectifs principaux de RDF est de proposer un cadre formel de définition de métadonnées sans donner trop l'importance sur les vocabulaires et syntaxes utilisés pour décrire ces métadonnées. RDF est un métalangage qui permet de définir des langages de description de données grâce à des vocabulaires RDF. Une des syntaxes de ce langage est RDF/XML.

RDF permet d'organiser des ressources Web dans un graphe orienté décrivant des triplets (sujet, prédicat, objet). Chaque triplet correspond à un arc orienté dont l'étiquette est le prédicat, le sujet est le nœud source et l'objet est la cible. Le sujet est ce dont on parle, souvent une URI, le prédicat représente un type de propriété applicable à la source et l'objet une donnée qui peut être simple ou composée.

CC/PP (*Composite Capabilities/Preferences Profiles*)[5] est une extension du vocabulaire de RDF. Un profil CC/CP définit les capacités des dispotifs et les préférences des utilisateurs à l'aide d'une structure de profils composés en une hiérarchie à deux niveaux. Un fichier RDF dont le vocabulaire est enrichi peut être décrit comme le montre la figure 2.4 : chaque profil possède un certain nombre de composants (<ccpp:component>) et chaque composant peut avoir plusieurs caractéristiques (<ex:xxx>). Le vocabulaire CP/PP peut être étendu. Ainsi, certains travaux de la littérature l'étendent pour décrire d'autres types d'informations de contexte tels que la localisation, les caractéristiques du réseau et les dépendances d'une application.

UAprof (*User Agent Profile*)[6] est construit au dessus de CC/CP. Il définit les caractéristiques des dispositifs et les préférences des utilisateurs, en particulier dans le protocole d'application sans fil WAP. Contrairement à CC/PP dont la spécification des caractéristiques des appareils est universelle et indépendante, Uaprof définit une variété de dictionnaires décrivant les caractéristiques des dispositifs.

WURFL (*Wireless Universal Resource File*)[7] est un fichier XML de description de ressources des terminaux mobiles. WURFL n'est pas une extension du vocabulaire RDF. Néanmoins sa syntaxe basée sur XML justifie sa présence ici. Les caractéristiques des appareils sont spécifiées dans un fichier avec un nom et une valeur sous forme de chaînes de caractères. Elles sont regroupées par catégorie, mais l'espace de nommage des caractéristiques est plat. Une description du terminal mobile Nokia N93 est illustrée dans la figure 2.5. Dans cette description, nous pouvons distinguer le nom du dispositif (`user_agent`) et un identifiant unique (`id`). Les groupes sont des descriptions des caractéristiques de l'appareil permettant de rendre WURFL plus lisible. Les noms des groupes sont explicites. Par exemple, l'affichage (`display`) désigne la résolution et la taille de l'écran. Deux capacités ne peuvent avoir le même nom (nom d'attribut). Les capacités ont toujours une valeur qui peut être un booléen, un nombre ou une chaîne de caractères.

La force principale des langages de la famille RDF est de permettre une extensibilité aisée. En

[4] http://www.w3.org/RDF/
[5] http://www.w3.org/Mobile/CCPP/
[6] http://www.w3.org/2005/MWI/BPWG/techs/UAProf
[7] http://wurfl.sourceforge.net/

```
<?xml version="1.0"?>
 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:ccpp="http://www.w3.org/2006/09/20-ccpp-schema#"
          xmlns:ex="http://www.example.com/schema#">
   <rdf:Description
       rdf:about="http://www.example.com/profile#MyProfile">
     <rdf:type
      rdf:resource=
            "http://www.w3.org/2006/09/20-ccpp-schema#Client-profile"/>
     <ccpp:component>
      <rdf:Description
           rdf:about="http://www.example.com/profile#TerminalHardware">
        <rdf:type
             rdf:resource="http://www.example.com/schema#HardwarePlatform" />
        <ex:displayWidth>320</ex:displayWidth>
        <ex:displayHeight>200</ex:displayHeight>
      </rdf:Description>
     </ccpp:component>
     <ccpp:component>
     </ccpp:component>
   </rdf:Description>
</rdf:RDF>
```

Fig. 2.4 – CC/PP : exemple de profil

effet, les utilisateurs peuvent définir leurs métadonnées avec des vocabulaires de leur choix. Cet aspect extensible du langage est important et plus particulièrement dans le cas d'un langage dédié. D'autre aspects importants à prendre en compte sont la déclaration explicite des identifiants. Enfin, l'unicité des identifiants dans RDF favorise l'analysabilité de ces descriptions.

2.4 Langages d'expression d'observation

Il existe des langages permettant l'observation des informations de contexte. L'observation est exprimée grâce des expressions avec des opérations spécifiques. La plupart des langages de requêtes permettent non seulement l'observation, mais aussi le traitement des informations observées. La composition est faite lors des requêtes par des opérations relativement élémentaires : sélection, jointure, comptage, comparaison, etc. Nous étudions différents langages de requêtes de contexte dans la section 2.4.1. La section 2.4.2 présente le langage de contexte défini dans le cadre de l'intergiciel Phœnix.

2.4.1 Langages de requête de contexte

Dans l'orientation donnée, les sources de contexte ou les collecteurs publient leurs informations sans considération de destination. La diversité de l'information nécessite des modes d'accès et de traitement différents. En effet, l'information peut être envoyée, dérivée, mesurée, détectée, raisonnée, traitée, calculée ou enregistrée par l'utilisateur [16]. Elle peut également être statique ou dynamique, temporelle, imprécise, erronée, ambiguë, indisponible ou incomplète. Les requêtes doivent permettre aux applications d'accéder aux informations qu'elles désirent de manière efficace pour pouvoir ensuite faire la composition. Cela suppose qu'il y ait un ajustement des informations pour qu'elles soient adaptées avant le traitement.

Il existe plusieurs catégories de langages de requêtes d'informations de contexte [16] : les langages orientés SQL, les langages de requêtes graphiques, les langages de requêtes orientés RDF et les langages de requêtes interactifs.

```
<device user_agent="NokiaN93"
        actual_device_root="true"
        fall_back="nokia_generic_series60_dp30"
        id="nokia_n93_ver1">
   <group id="product_info">
      <capability name="model_name" value="N93"/>
   </group>
   <group id="display">
      <capability name="resolution_width" value="240"/>
      <capability name="resolution_height" value="320"/>
      <capability name="max_image_width" value="240"/>
      <capability name="max_image_height" value="320"/>
   </group>
   <group id="image_format">
      <capability name="colors" value="262144"/>
   </group>
   <group id="object_download">
      <capability name="ringtone_aac" value="true"/>
      <capability name="ringtone_voices" value="64"/>
      <capability name="video_qcif" value="true"/>
      <capability name="video_sqcif" value="true"/>
      <capability name="video_vcodec_h263_0" value="true"/>
      <capability name="video_vcodec_mpeg4" value="true"/>
      <capability name="video_acodec_aac" value="true"/>
   </group>
   <group id="sound_format">
      <capability name="aac" value="true"/>
   </group>
</device>
```

FIG. 2.5 – WURFL : exemple de description d'un appareil Nokia N93

Langages orientés SQL SQL (*Structured Query Language*) est sans doute le langage le plus utilisé pour faire des requêtes sur des bases de données. C'est un langage déclaratif qui fournit une base solide pour créer et exécuter un grand nombre de requêtes. SLQ permet l'accès aux bases de données et en même temps de faire le traitement sur ces données. Le traitement consiste à modifier la structure de la base de données, modifier le contenu de la base de données, etc. SQL permet une simple manipulation de données. Le langage COSMOS peut s'inspirer de SQL concernant la simplicité d'accès aux informations.

Langages de requêtes graphiques Ce sont des alternatives aux langages orientés SQL pour faire des requêtes sur des modèles relationnels. L'un des langages les plus utilisés dans ce domaine est QBE (*Query-By-Examples*) [28]. QBE est basé sur le calcul relationnel. QBE est facile à utiliser et permet aux utilisateurs (non programmeurs) de faire des requêtes en créant des tables d'exemples. En effet, QBE permet aux utilisateurs de définir une image de la réponse qu'ils désirent et ils peuvent par la suite voir figurer les données répondant à l'interrogation demandée. La force principale de QBE est la simplicité d'utilisation : les utilisateurs n'ont pas besoin d'écrire des requêtes mais plutôt de définir le besoin qu'ils attendent, aussi appelé image de réponse. L'utilisation via l'interface graphique apporte encore plus de simplicité d'utilisation.

Un autre langage très étudié est CML (*Context Modeling Language*). CML permet de modéliser de manière formelle le contexte auquel une application est sensible. C'est une modélisation à la UML. CML peut ainsi bénéficier des avantages d'un langage de modélisation.

Langages de requêtes orientés RDF Dans les systèmes d'applications sensibles au contexte, le fichier XML sert souvent comme support de configuration. Concernant le domaine de requête, AWQL (*Augmented World Query Language*) [21] raisonne en terme d'objets. Ainsi, il représente

le contexte comme un ensemble d'objets de données avec des attributs. Tous les objets qui sont produits par un fournisseur de contexte appartiennent à un environnement (décrit comme étant un *espace actif augmenté*).

RDF est un cas particulier de XML. Il existe des langages d'interrogation de graphes dans RDF. L'un des langages les plus innovants est CQP (*Collaborative Query Processing*). Le modèle CQP supporte des requêtes explicites demandées par l'utilisateur et des requêtes implicites mises en jeu par les profils utilisateurs.

Avec une approche formelle de représentation de contextes, les langages orientés RDF semblent être les mieux placés pour répondre à l'accès et au traitement des données. La faiblesse des langages RDF réside toutefois dans leur utilisation car il nécessite une certaine connaissance dans la modélisation.

Langages de requêtes interactifs iQL (*Interactive Query Language*) [9] est un langage de requêtes de données interactif. Le but de ce langage est de faciliter la gestion des sources de contexte dynamique. Il permet la création de nouvelles liaisons (*rebinding*) à l'exécution. Les langages iQL sont composés d'entités appelées composeurs. Les composeurs prennent en entrée une valeur déterminée par une composition de composeurs. L'expression en entrée permet la reconfiguration des liaisons pour les sources de contexte qui satisfont la requête. Cette expression décrit également quelle information a besoin d'être ajustée avant le traitement effectif ayant un impact dans le contexte.

Parmis les langages de requêtes de contexte présentés, certains présentent une utilisation relativement simple tels que les langages de requêtes orientés SQL ou les langages de requêtes graphiques. Cette simplicité est souvent associée à une structure simple du langage permettant de faire une analyse statique de programme. D'autres langages sont moins simples d'utilisation (iQL), ce qui nous intéresse dans ces langages est alors la gestion dynamique de contexte. Entre ces deux extrémités, il y a des langages orientés RDF. La définition explicite des requêtes dans ces langages permet de favoriser une analyse statique. La description explicite des contextes est possible car les requêtes peuvent aussi être définies de manière dynamique. Cela suppose que le langage ait une structure simple.

2.4.2 Phœnix

Phœnix est un canevas logiciel dédié à l'observation système d'applications réparties s'exécutant sur une grappe de machines [5]. L'architecture de Phœnix est décomposée en quatre modules : agent d'observation, sondes (dans l'application), diffusion sur le réseau local et librairies pour l'outillage. Les agents d'observation permettent de configurer la fréquence des observations et de multiplexer les observations. Le langage dédié Phœnix permet d'exprimer l'observation des informations de contexte. La structure du langage permet de spécifier l'identifiant des ressources observables, des opérations logiques de premier ordre, des opérations de comparaison et l'opérateur **DELTA** pour l'amplitude des variations. Le langage ne possède pas d'opérateur à mémoire, à seuil, de traduction de format, de fusion de données, etc. À titre d'exemple, l'expression suivante décrivant la sonde `io` « `probe io ((IO-FREQ > 500) AND (IO-CALL_NBR > 5)) OR (DELTA(IO-VOLUME) > 500.000 )) ;` » se traduit par « les informations sont délivrées au moins 500 ms après la dernière trace et plus de 5 entrées/sorties ou lorsque le volume des entrées/sorties depuis la dernière trace excède 500 Kb ». La structure du langage est simple, un autre point essentiel de ce langage est que toute expression peut être vérifiée très facilement. En revanche, ce langage ne permet la compostion d'informations qu'à un seul niveau et ne permet pas le partage.

2.5 Langages de composition d'informations de contexte

Les langages de cette section sont utilisés généralement dans l'orientation processus. Dans cette orientation, la composition d'informations de contexte est une construction de graphe dont chaque nœud représente la compostion d'informations de contexte. La recherche d'informations de contexte correspond à la recherche des entités « objet », « processus », « composant » ou « service ».

Nous allons étudier différents langages dédiés à la compostions d'informations de contexte définis dans des services logiciels. Tout d'abord, nous présentons dans la section 2.5.1 le langage de contexte défini dans le canevas logiciel Gaia. Ensuite, la section 2.5.2 étudie le langage dédié WildCAT. La section 2.5.3 montre que la composition d'informations est aussi présente dans des langages de spécification de qualité de service tel que QML.

2.5.1 Gaia

Gaia [22] est un canevas logiciel pour la gestion des services contextuels, c'est-à-dire de services sensibles au contexte. Il est construit sur un service de gestion d'événements pour la distribution d'événements dans les espaces actifs. Le service de contexte est composé de fournisseurs de contexte, de synthétiseurs, de consommateurs et d'ontologies. Les fournisseurs de contexte sont des capteurs ou autres sources d'informations de contexte. Ils offrent des interfaces de requêtes aux autres entités et des canaux d'événements pour les notifications. Les fournisseurs peuvent associer des informations de contexte aux mesures prises. Les informations sur la sémantique et la structure des informations de contexte qu'ils doivent fournir sont obtenues à partir de la base de données des ontologies. Les synthétiseurs d'informations de contexte agrègent les informations de contexte depuis plusieurs fournisseurs et en déduisent des informations de plus haut niveau d'abstraction. La spécification de ces informations repose sur un langage dédié qui est basé sur la logique des prédicats. Enfin, les services de Gaia sont développés en utilisant un langage de script de haut niveau (LuaOrb).

Langage de spécification d'informations de Gaia Le langage est basé sur des règles de logique de prédicat. Il s'inspire directement du langage naturel avec des expressions de la forme <sujet><verbe><objet>. Ainsi, un prédicat de contexte élémentaire est défini de la façon suivante : Context(<ContextType>,<Subject>,<Relater>,<Object>). Le ContextType représente le type du contexte, le sujet Subject est la personne, la place ou l'élément ayant un rôle dans le contexte. L'objet Object est une valeur associée au sujet. La description d'événements (représentée par Relater) explicite les relations entre le sujet et l'objet grâce aux opérateurs de comparaison (=, >, or, <), à un verbe, ou à une préposition. En pratique, un type de contexte est un canal d'événements. Des exemples de prédicats de contexte sont définis comme suit [22]. Les exemples de prédicats de la figure 2.6 sont aisément interprétables. Le premier prédicat explicite le type de contexte qui est un emplacement, le sujet est une personne (chris), l'objet est une chambre numéro 3231 et la relation entre le sujet et l'objet est entrant (entering), ce qui se traduit par « dans le contexte emplacement, chris entre dans la chambre numéro 3231 ». Les prédicats de contexte des lignes 2 et 3 sont définis de la même manière. La ligne 2 définit un contexte de type social tandis que la ligne 3 est un contexte de type temporel.

Il est possible de construire des contextes plus complexes en composant des prédicats de contexte grâce aux opérations de logique de premier ordre telles que la quantification, l'implication, la conjonction, la disjonction et la négation de prédicats de contexte. La figure 2.7 illustre un exemple de spécification d'un contexte composé de plusieurs contextes. Ainsi, la figure 2.7 décrit une composition de contexte à partir de trois prédicats de contexte. La composition est la

conjonction des deux premiers prédicats suivie par une implication.

La description de contexte dans Gaia est une description de très haut niveau. Ce haut niveau d'abstraction permet une compréhension et une utilisation simples de la notion de contexte. En revanche, l'analyse de contexte peut être ambigue car chaque prédicat peut être interprété différemment selon les utilisateurs. Néanmoins, la définition des relations (*relater*) doit être explicite le plus possible afin d'atténuer l'ambiguïté. La composition de contexte est très élémentaire et peut être fastidieuse à mettre en œuvre surtout quand il s'agit de la composition de plusieurs prédicats de contexte. Cela peut entraîner finalement une perte au niveau de l'analyse et de la lisibilité qui sont pourtant des facteurs clés. Pour résumé, le langage Gaia permet la composition d'informations élémentaire de contexte et les modes d'observation et de notification.

```
1 Context(location, chris, entering, room 3231);
2 Context(social, relationship, sister, serena);
3 Context(time,~, Is, 12:00 01/01/01)};
```

FIG. 2.6 – Gaia : exemples de prédicats de contexte

```
1 Context(Number of people, Room 2401, >, 4)
2 AND
3 Context(Application, PowerPoint, is, Running)
4 =>
5 Context(Social Activity,Room 2401, Is, Presentation).
```

FIG. 2.7 – Gaia : exemple de composition de prédicats de contexte

2.5.2 WildCAT

SAFRAN est une extension du modèle de composant Fractal permettant d'associer dynamiquement des politiques d'adaptation aux composants d'une application. Ces politiques sont décrites dans un langage dédié sous la forme de règles réactives. Leur exécution repose d'une part sur WildCAT [13], un système permettant de détecter les évolutions du contexte d'exécution, et d'autre part sur FScript, un langage dédié pour la reconfiguration dynamique cohérente de composants Fractal.

WildCAT est utilisé par SAFRAN pour implanter la partie « événements » des règles d'adaptation qui nécessite d'observer le contexte d'exécution et de détecter les événements mentionnés. Le contexte d'exécution est organisé en un ensemble de *domaines contextuels*, identifiés par un nom unique, qui représentent chacun un aspect spécifique du contexte (par exemples, les ressources matérielles, le réseau, le profil utilisateur). Chacun de ces domaines contextuels est modélisé sous la forme d'une arborescence de ressources nommées, chacune d'entre elles étant décrite par un ensemble d'attributs écrits sous la forme de paires (nom,valeur). Le langage dédié de WilCAT fournit une syntaxe simple inspirée des URI pour désigner les ressources et attributs. 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. Bien que cette utilisation facilite la description des évolutions du contexte, ce langage ne permet pas de compostion de haut niveau.

Un autre langage intéressant à étudier dans WildCAT est FPath. De notre point de vue, l'intérêt de FPath est la possibilité de naviguer dans les architectures FRACTAL en sélectionnant

les interfaces d'un composant, ses attributs de configuration, ses sous-composants directs ou parents, et en suivant une connexion à partir d'une interface. FPath permet aussi de sélectionner en une seule expression tous les sous-composants directs et indirects ainsi que les composants parents. Par exemple, l'expression `child::server/attribute::cacheEnabled` sélectionne dans un premier temps les sous-composants nommés server du composant courant pour retourner son attribut de configuration nommé cacheEnabled. FPath permet aussi de tester la configuration de l'architecture, par exemple `count(interface::*[required(.) and not(bound(.))]) > 0` retourne vrai si et seulement si les interfaces requises du composant courant (.) ne sont pas connectées.

2.5.3 QML

Bien que QML [15] ne soit pas à la base un langage dédié à la composition d'informations mais plutôt un langage dédié à la spécification de la qualité de service, nous y trouvons aussi de la composition d'informations. Il est donc intéressant d'étudier cette forme de composition afin de la comparer aux autres déjà étudiées précédemment. Parallèlement, nous pouvons nous inspirer des propriétés intéressantes de ce langage.

Le but de QML est de spécifier de manière systématique et déclarative la qualité de service (QdS) des composants logiciels. Le langage QML est implanté comme une extension de UML centrée sur l'expression des besoins de QdS. QML possède trois mécanismes d'abstraction principaux pour la spécification de QdS : le type de contrat, le contrat, et le profil.

QML permet la définition de types de contrat qui représentent des aspects de QdS spécifiques comme la performance ou la fiabilité. Un type de contrat définit des dimensions qui peuvent être utilisées pour caractériser un aspect de QdS particulier. Une dimension possède un domaine de valeurs qui peuvent être ordonnées. Les domaines peuvent être de trois types : ensemble, énuméré et numérique. Un contrat est une instance d'un type de contrat et représente une spécification particulière de la QdS. Enfin, les profils QML associent les contrats à des interfaces, des opérations, des arguments et des résultats d'opérations.

QML permet aussi le raffinement de profil et de contrat. La raison principale du raffinement de profil est de permettre la définition par dérivation des profils et d'éviter de définir des nouveaux profils à partir de rien. Le raffinement fournit un profil qui possède une propriété sémantique plus forte. Le raffinement de profil est défini dans les termes du raffinement de contrat. Le raffinement de contrat s'exprime par un simple lien d'héritage. L'autre motivation pour le raffinement de contrat est de supporter la notion de contrat par défaut à l'intérieur d'un profil.

QML permet une spécification des informations de contexte par encapsulation. La composition d'informations dans QML correspond au regroupement de contraintes à l'intérieur d'un contrat et également au regroupement de contrats à l'intérieur d'un service. Ce sont des compositions très élémentaires. D'autres éléments importants à noter dans QML sont la définition modulaire de types de contrat, de contrat et d'interface. Cela donne une vision concise de ce que fait chaque programme et permet donc une utilisation simple et concise du langage.

2.6 Synthèse

Dans cette section, nous réalisons une synthèse de l'ensemble des langages étudiés dans ce chapitre. Le tableau 2.1 présente l'étude des langages de composition d'informations de contexte étudié dans ce chapitre. Les lignes sont les critères généraux et spécifiques au domaine de la composition d'informations de contexte et les colonnes sont les langages étudiés. Nous utilisons les notations suivantes pour représenter les informations :

– un signe « + » indique que des éléments notables sont proposés dans le langage ;

	SensorML	RDF	Orientés SQL	Langages graphiques (QBE)	Orientés RDF (CQP)	Langage interactif (iQL)	Phœnix	Gaia	SAFRAN	QML
Simplicité d'utilisation	−	#	+	+	#	#	+	+	+	#
Extensibilité		+			+			−		
Analysabilité		#		+	+	#	#	#		
Sûreté			#				−	−	#	#
Réutilisation	#	#			#	#		#		
Optimisation					#					
Testabilité			#					#		
Composabilité			#	#			#	#	#	#
Partage	#									

TAB. 2.1 – Tableau récapitulatif des langages dédiés à la gestion de contexte

- un signe « # » indique que certains éléments sont présents mais de manière insuffisante ;
- un signe « − » indique qu'aucun élément n'est rempli, ou que des éléments ne sont pas pris en compte dans le langage ;
- une case est vide lorsque le langage ne considère pas le critère.

Dans cet état de l'art, nous avons tout d'abord classé les langages dédiés à la gestion de contexte en deux catégories. Ensuite, nous avons présenté différents langages de la littérature dans les deux catégories. Cette analyse nous montre qu'il existe deux types de composition d'informations de contexte : le premier est une composition élémentaire (un seul niveau) tandis que le deuxième est une composition complexe (plusieurs niveaux). C'est le deuxième type de composition qui nous intéresse. En effet, du fait de l'utilisation du modèle de composants Fractal (hiérarchique avec partage), les compositions d'informations de contexte dans COSMOS sont fondamentalement faites à plusieurs niveaux. Ces politiques de gestion de contexte nécessitent, d'une part, une structure de composition sophistiquée et, d'autre part, des opérateurs de composition. Cela repose sur un langage ayant des constructions de haut niveau pour permettre la composition aisée des informations de contexte. Parallèlement, la structure du langage doit faciliter la conversion vers FRACTAL ADL. Dans la littérature, les solutions existantes ne sont pas suffisantes concernant la composition d'informations de contexte et le partage de ces informations. Premièrement, la compostion est souvent associée au regroupement de données ou est une relation avec des opérations relativement élémentaires telles que les opérations de la logique de prédicat du premier ordre. Deuxièment, le modèle de composants et l'ADL utilisés dans les canevas logiciels existants ne sont pas FRACTAL et FRACTAL ADL. Enfin, troisièment, même si certains canevas adoptent FRACTAL et FRACTAL ADL, les compositions d'informations de contexte sont relativement élémentaires. Les objectifs de notre travail de recherche consistent à définir un nouveau langage dédié permettant la composition aisée (avec partage) d'information

de contexte. De plus, grâce à ses idiomes, le langage doit permettre d'exprimer les patrons de conception de COSMOS qui sont le patron conception « Composite », le patron de conception « Patron de méthode », le patron de conception « Poids-mouche » et le patron de conception « Singleton ».

Chapitre 3

Développement du langage dédié COSMOS

Ce chapitre aborde le développement du langage dédié COSMOS. Tout d'abord, la section 3.1 présente le processus de développement de DSL COSMOS de façon générale. La section 3.2 analyse le domaine de la composition d'informations de contexte. La sections 3.3 présente la conception de DSL COSMOS. La section 3.4 met en œuvre le langage. Enfin, la section 3.5 évalue DSL COSMOS par rapport aux critères d'exigence donnés dans le chapitre 2.

3.1 Processus de développement de DSL COSMOS

La figure 3.1 illustre le processus de développement de DSL COSMOS. Le processus de développement de DSL COSMOS commence par une analyse de domaine à l'aide de la méthodologie FODA basée sur des notations de très haut niveau. Cette analyse fournit des informations nécessaires à la conception du langage. La grammaire peut alors être spécifiée à partir de l'analyse. À partir de cette spécification de la grammaire, le compilateur se charge de construire un arbre de syntaxe. Cet arbre n'est pas commode pour être manipulé directement afin de faire l'analyse sémantique. L'idée est donc de construire un métamodèle en UML de la grammaire et de construire un arbre d'interprétation ou arbre de syntaxe abstraite à partir de l'arbre de syntaxe. Le métamodèle UML construit est le métamodèle de COSMOS.

3.2 Analyse de domaine

L'étude de la gestion des informations de contexte avec COSMOS dans le chapitre 1 est une première ébauche de l'analyse de domaine nécessaire à la création de DSL COSMOS. La section 1.2.2 a présenté une méthodologie générale pour l'analyse de domaine. Nous présentons maintenant comment nous effectuons cette analyse afin de capturer les concepts du domaine pour avoir les données nécessaires à la spécification du langage. Nous utilisons FODA [17] comme méthodologie pour analyser le domaine. Il existe plusieurs concepts à plusieurs niveaux d'abstraction dans la description des politiques de gestion de contexte dans COSMOS. Les concepts de base sont les nœuds de contexte et les concepts de haut niveau sont les politiques de gestion de contexte. Un programme DSL COSMOS permet de décrire une politique de gestion de contexte, c'est-à-dire une composition hiérarchique avec partage d'informations de contexte.

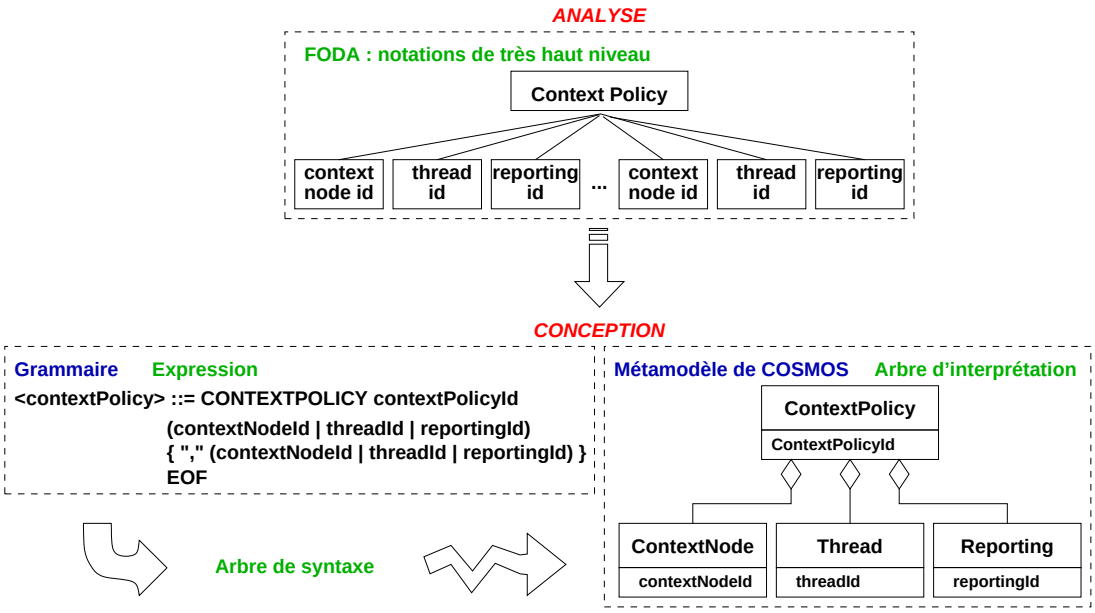

FIG. 3.1 – Présentation du processus du développement de DSL COSMOS

Cette section est organisée comme suit. La section 3.2.1 présente la méthodologie d'analyse FODA. Puis, la section 3.2.2 analyse le domaine avec la méthodologie FODA. Enfin, la section 3.2.3 justifie les notations utilisées qui deviennent dans la suite des constructions de base du langage.

3.2.1 Présentation de la méthodologie d'analyse FODA

FODA (en anglais, *Feature Oriented Domaine Analysis*) [17] est une méthodologie d'analyse préconisée par [20]. C'est un diagramme de concepts dans lequel un concept correspond à une terminologie, à un élément, à un objet, à une propriété, etc. FODA permet de représenter les instances du domaine. Dans notre cadre d'étude, une instance est une structure d'un programme. Avec FODA, nous construisons un modèle du domaine. La spécification du DSL correspond alors à la traduction du modèle du domaine en des expressions avec des règles de grammaire spécifiques. La figure 3.2 décrit la notation FODA que nous utilisons. Un diagramme FODA est composé de nœuds de concepts :

- un nœud racine fait référence au concept de plus haut niveau d'abstraction ;
- un nœud peut être obligatoire ou optionnel ;
- des relations entre ces nœuds sont exprimées par des opérations de décomposition telles que la décomposition « et » (*and-decompostion*) et la décomposition « ou exclusif » (*xor-decompostion*).

3.2.2 Analyse du domaine avec FODA

La politique de gestion de contexte est le premier concept rencontré donc en quelque sorte le concept de plus haut niveau dans la gestion de contexte dans COSMOS. En effet, les utilisateurs composent les informations de contexte en décrivant des politiques de gestion de contexte.

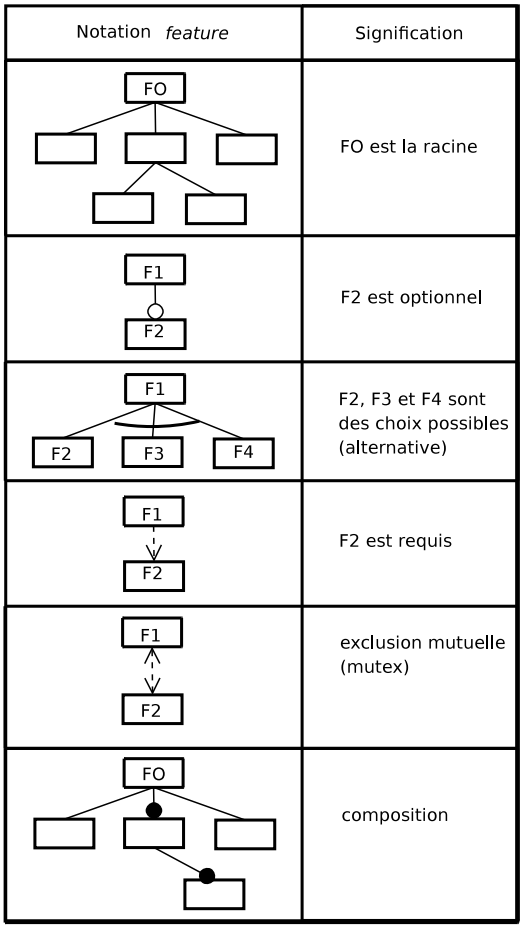

FIG. 3.2 – FODA : description des notations utilisées dans la suite

Une politique de gestion de contexte (`context policy`) est un regroupement d'informations de contexte qui sont des nœuds de contexte. Elle contient un ensemble de fils d'exécution (thread) et un ensemble de gestionnaires de rapports de contexte (reporting). La figure 3.3 montre qu'un context policy contient soit un contextnode id soit un thread id soit un reporting id, et ce choix peut être répété un nombre quelconque de fois.

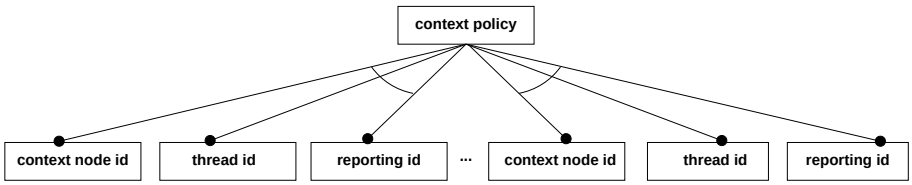

FIG. 3.3 – Modélisation FODA de politique de gestion de contexte de COSMOS

Partie fonctionnelle La racine de la partie fonctionnelle, le nœud de contexte context node, se décompose en deux constructions : le capteur (sensor) et le processeur (processor). Dans la figure 3.4, la racine du diagramme représente un nœud de contexte qui est un capteur ou un processeur. Un capteur (sensor) possède un identifiant unique, des attributs qui sont optionnels et la référence vers un gestionnaire de ressource. Les attributs peuvent être génériques/prédéfinis ou définis par l'utilisateur. Les attributs prédéfinis sont « observateur actif » (active observer), « observateur bloquant » (blocking observer), « notificateur actif » (active notifier) et « notificateur bloquant » (blocking notifier), et l'attribut défini par l'utilisateur attibute assignment est une paire (attribute name, attribute value). Un capteur possède également un opérateur qui est toujours le même (extract operator), donc il n'est pas nécessaire de le nommer.

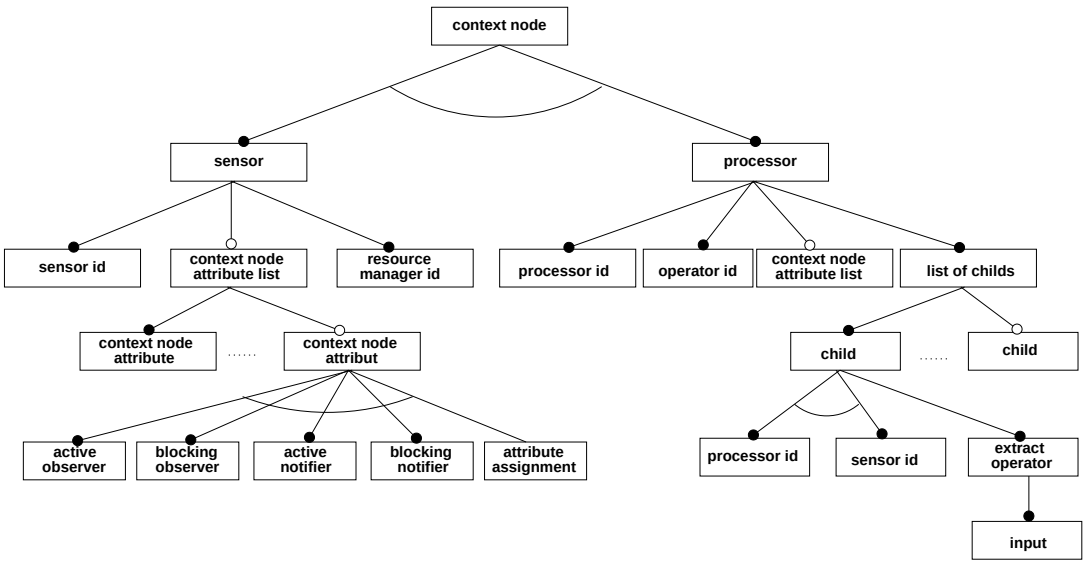

FIG. 3.4 – Modélisation FODA de nœud de contexte de COSMOS

Le gestionnaire de ressource est illustré dans la figure 3.5. Un gestionnaire de ressource est

constitué d'un identifiant unique (resource manager id), d'un rapport de contexte (un message) en sortie (output), d'un type (resource manager type), p.ex. le nom d'une classe Java, et d'attributs (resource mnger attribute list). Un message, tel qu'illustré dans la figure 3.6, est une hiérarchie de sous-messages (chunk or sub-message list). Les morceaux de message sont typés, p.ex. avec les noms de classes Java. Les attributs (attribute assignment) d'un gestionnaire de ressource sont tous des paires (nom d'attribut, valeur).

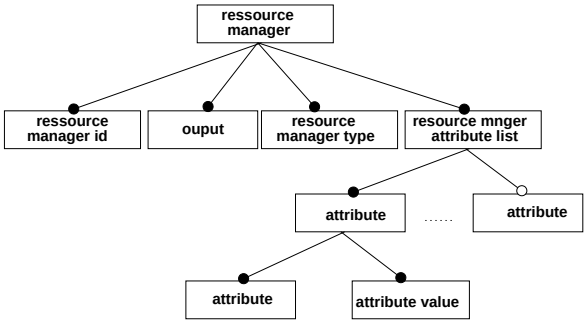

FIG. 3.5 – Modélisation FODA de gestionnaire de ressource

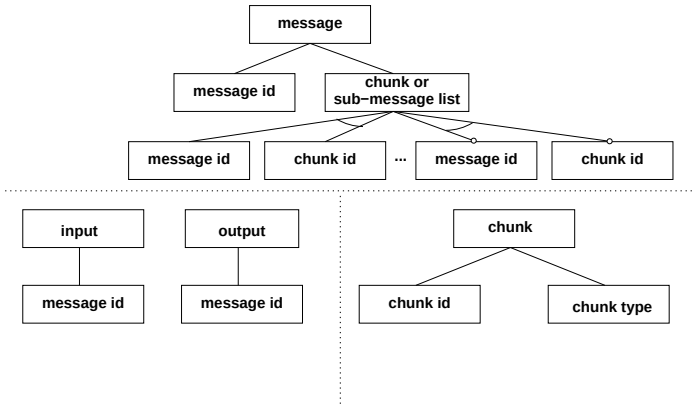

FIG. 3.6 – Modélisation FODA de message et de morceaux de messages

Étant à la base un nœud de contexte tout comme le capteur, un processeur possède un identifiant unique, un opérateur, des attributs optionnels et des nœuds enfants. Chaque enfant est un nœud de contexte, c'est-à-dire un capteur ou un processeur. L'opérateur illustré dans la figure 3.7 possède un identifiant, un type (le nom d'une classe Java), prend en entrée des rapports de contexte et fournit en sortie un rapport de contexte. Un processeur peut extraire un morceau de messages désiré à partir d'un ensemble de messages. L'opération d'extraction de morceaux de messages est un cas particulier et d'utilisations très fréquente, donc une construction spéciale est identifiée dans le langage (extract operator de la figure 3.4).

Parties extra-fonctionnelles Les parties extra-fonctionnelles correspondent à la gestion des activités et à la gestion des réserves de rapports de messages. Les activités (activity) sont regrou-

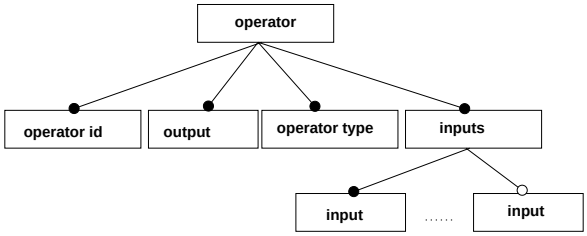

FIG. 3.7 – Modélisation FODA d'opérateur du processeur

pées dans des tâches. Plus précisément, une tâche est composée d'un identifiant et d'une liste d'activités. Les tâches (task to thread) sont regroupées dans des fils d'exécution (thread) tels que les représente la figure 3.8.

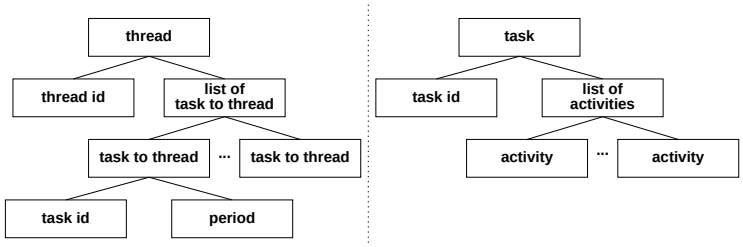

FIG. 3.8 – Modélisation FODA d'ctivités, de tâches et de fils d'exécution

3.2.3 Abstractions

L'analyse de domaine en adoptant FODA comme méthodologie nous a permis d'identifier les principaux concepts du domaine de la composition d'informations de contexte. Ces concepts sont des constructions de base du langage : contextpolicy, contextnode (donc sensor et processor), resource manager, operator, task, thread, etc. Ces terminologies utilisées sont des abstractions appropriées au domaine de la gestion de contexte avec COSMOS car elles correspondent aux termes utilisés dans COSMOS et peuvent être facilement interprétables par des experts du domaine. À partir de ces abstractions et de leurs propriétés, nous pouvons spécifier la grammaire du langage.

3.3 Conception de DSL COSMOS

Dans cette section, nous concevons DSL COSMOS. Nous commençons par présenter une vue générale du langage dans la section 3.3.1. Ensuite, la section 3.3.2 détaille les types et opérateurs primitifs du langage. Enfin, nous donnons la spécification du langage dans la section 3.3.3.

3.3.1 Vue générale de DSL COSMOS

DSL COSMOS est un langage déclaratif. Contrairement à un langage impératif, un langage déclaratif permet la déclaration des fonctions ou expressions au lieu de définir des procédures. Le langage COSMOS est aussi un langage descriptif, c'est-à-dire un langage à l'expressivité réduite

qui permet de décrire des informations ou fonctions, en particulier la composition d'informations de contexte. D'autres caractéristiques importants de DSL COSMOS sont la définition de façon modulaire des expressions. Les politiques peuvent être définies indépendamment des expressions élémentaires. L'accès à des références peut s'inspirer de FPath utilisé dans WildCAT. Le langage COSMOS est typé. En effet, toutes les constructions de base de COSMOS sont typées, de même pour les opérateurs. En revanche, les types de construction de base sont de même nature et exprimée par une chaîne de caractères. Le typage est essentiel pour faire l'analyse de cohérence en appliquant des règles d'inférence. Le langage étant construit sans instruction, les règles d'inférence dans notre cadre d'étude se limitent à des vérifications de type des expressions, des sorties et des entrées d'un opérateur. La restriction des règles d'inférence permet de rendre ces règles compactes et concises pour la spécification de la sémantique.

3.3.2 Types, opérateurs primitifs et sémantique

Comme tout langage de programmation, le langage COSMOS est basé sur l'utilisation de types primitifs de données et d'opérateurs. Le type d'un langage détermine les moyens d'analyser les mots reconnus de ce langage.

Pour les types primitifs, nous avons besoin principalement des entiers (int) et des chaînes de caractères (string). Les entiers nous servent pour spécifier la périodicité d'exécution des activités dans les fils d'exécution. Les chaînes de caractères permettent de définir des identifiants. La définition de nouveaux types est interdite. À titre d'exemple, en Caml, il y a possibilité de définir de nouveaux type (p.ex. le type énuméré, `type cours = informatique | mathématiques | physique ;;` ou le type enregistrement en C : `struct point {float x ; float y ;}`). Cette interdiction de définition de nouveaux types est importante car le langage doit être restreint au domaine. Dans ce langage dédié, nous avons également besoin d'avoir des opérateurs d'affectation ("=" et "=>"). Il est important de noter qu'un DSL n'est pas nécessairement purement déclaratif. Des procédure peuvent être intégrées dans la structure du langage permettant de traiter les données du domaine. Dans notre cadre d'étude, nous pensons que ces fonctions procédurales ne sont pas nécessaires. Dans le langage, il n'y a donc pas de construction d'instruction de contrôle (p.ex. `for ( ; ; ;) { instructions }`).

3.3.3 Spécification

La spécification de DSL COSMOS repose principalement sur l'analyse de domaine effectuée en utilisant la méthodologie FODA dans la section 3.2.1. Nous détaillons dans cette section la spécification des constructions principales de DSL COSMOS. La spécification complète se trouve dans l'annexe A.

Caractéristiques du langage La grammaire de DSL COSMOS est de type LL(k) que nous spécifions en respectant la syntaxe EBNF. En effet, la syntaxe des expressions du langage dédié au domaine de la composition d'informations de contexte est définie par un ensemble de formules constituant une grammaire EBNF. C'est aussi une grammaire non contextuelle, hors contexte ou algébrique.

Extensions de la forme de Backus-Naur La forme de Backus-Naur est un métalangage, c'est-à-dire qu'elle permet de décrire les langages étudiés. La forme BNF possède sa propre syntaxe de base. Elle est caractérisée par la distinction entre les méta-symboles, les terminaux et les non-terminaux, et bien sûr par la manière dont nous pouvons écrire ou « placer » ces symboles côte à côte. Pour faciliter la rédaction d'une grammaire particulière, la forme BNF

peut être étendue. À ce jour, plusieurs extensions ont été proposées. Cependant, il existe un point commun entre ces extensions qui est l'utilisation des opérateurs de répétition tels que l'étoile de Kleen « * » et «+».

La syntaxe de la grammaire de DSL COSMOS utilise une forme étendue de Backus-Naur. Nous utilisons le métasymbole « ::= » pour définir le rôle du symbole (les définitions de la production). Les symboles non terminaux sont écrits à l'intérieur des caractères inférieur et supérieur (< >), p.ex `< symbole non terminal >`. Nous utilisons les crochets ([ ]) pour entourer les éléments optionnels (p.ex. `[élément optionnel]`). Les accolades ({ }) sont utilisées pour entourer les éléments à répéter un nombre quelconque de fois (c'est-à-dire 0 fois ou n fois où n peut être infini).

Syntaxe de la grammaire du langage dédié COSMOS Les mots réservés de la grammaire sont les suivants :

```
contextpolicy     ressourcemgr     task
sensor            message          reporting
processor         operator         thread
chunk             extract
```

Une politique de de contexte est défini par un ensemble de nœuds de contexte, un ensemble de fils d'exécutions et un ensemble de rapports de messages.

```
< contextPolicy >   ::=   CONTEXTPOLICY contextPolicyId "="
                             (contextNodeId | threadId | reportingId)
                               { "," (contextNodeId | threadId reportingId)}
```

DSL COSMOS isole la description des parties fonctionnelles des parties extra-fonctionnelles. Les parties fonctionnelles sont réifiées par les deux constructions sensor et processor. Ainsi, nous spécifions les parties fonctionnelles de la manière suivante. Un capteur peut être configuré avec les attributs prédéfinis de nœuds de contexte et son gestionnaire de ressources peut être défini par référence (typage faible) ou en donnant la définition exacte de manière explicite (typage fort). Il y a la possibilité de déclarer de nouveaux gestionnaire de ressources. Un gestionnaire de ressources fournit en sortie un message ou un morceau de message et aucune donnée en entrée. Un gestionnaire de ressource référence un nom de classe Java et peut être configuré avec des attributs. Ces attributs sont regroupés dans une liste. Ce sont des paires (attribut, valeur) qui sont définies par des utilisateurs.

```
< sensor >           ::=   SENSOR sensorMgrID "="
                             (resourceMgrId | "(" < resourceMgrDef> ")")
                                "[" < attributeCNList > "]" EOF
< resourceMgr >      ::=   < ressourceMgrDef> EOF
< resourceMgrDef >   ::=   ("{" < chunkList > "}" | messageId)
                              resourceMgrClass "[" < attributeRMList > "]"
< attributeCNList >  ::=   attributeCN { "," attributeCN }
< attributeCN >      ::=   "BO" | "AO" | "AN" | "BN" | < attributeASSIGN >
< attributeASSIGN >  ::=   attributeID "=>" attributeValue
```

Un processeur est configuré avec des attributs par défaut et son opérateur est défini par référence ou en donnant explicitement la définition correspondante. L'opérateur d'un processeur prend en entrée un ensemble de morceaux de message et fournit en sortie un morceau de messages.

```
< processor >        ::=   PROCESSOR processorID "="
```

```
                              (operatorId |  "("< operatorDef >")")
                                 [< attributeCNList >] "("< childList >")" EOF
< operator >          ::=   OPERATOR operatorID "=" < operatorDef > EOF
< operatorDef >       ::=   (< chunkList > | messageId)
                              operatorClass "("< chunkListList >")"
```

Les parties extrafonctionnelles sont réifiées par trois constructions task, thread et reporting. Les activités sont regroupées dans des hiérarchies de tâches. Les tâches sont organisées dans des fils d'exécution avec une périodicité d'exécution pour chaque activité. Les rapports de contexte sont regroupés dans un gestionnaire de rapports de contexte.

```
< task >              ::=   TASK taskName "=" < activityMgrList >;
< activityMgrList >   ::=   activityMgr { "," activityMgr }
activityMgr           ::=   contextNodeId;

< thread >            ::=   THRED threadId "=" < taskToThreadList > EOF
< taskToThreadList >  ::=   < taskToThread > { "," < taskToThread > }
< taskToThread >      ::=   taskToThreadID "[" period "]"
< period  >            ::=   intValue

< reporting >         ::=   REPORTING reportingID "=" < reportNodeList >
< reportNodeList >    ::=   reportNode { "," reportNode }
< reportNode >        ::=   contextNodeID [Path FPathOp]
```

3.3.4 Modèle sémantique

À partir de la spécification de la grammaire, l'analyseur lexical et syntaxique nous permet de construire l'arbre de syntaxe. L'arbre représente la structure de la grammaire, mais cette représentation ne permet pas de faire une analyse sémantique directement. Il est donc essentiel de modifier l'arbre de syntaxe en un arbre d'interprétation (ou arbre de syntaxe abstraite) en vue de l'analyse sémantique. Cette version optimisée de l'arbre syntaxique profite de plusieurs avantages, notamment la perte d'informations est évitée et il est plus facile d'ajouter des informations au fur et à mesure des analyses. Pour cela, nous construisons un modèle sémantique de DSL COSMOS. C'est un métamodèle en UML du langage construit à partir de l'arbre de syntaxe. Ainsi, l'arbre d'interprétation doit être conforme au métamodèle. Ce métamodèle est dessiné dans la figure 3.9. Nous retrouvons dans ce diagramme les constructions de la grammaire. Nous avons un modèle de la politique de gestion de contexte. Nous notons en particulier qu'une politique n'est pas une composition, mais une agrégation de fils d'exécution, de nœuds de contexte et de rapport de messages. Cela met en évidence la notion de partage de nœuds de contextes dans COSMOS. À gauche, nous avons le modèle d'un fils d'exécution, et au milieu, le modèle du nœud de contexte. La notion d'extension est mise évidence par les classes Sensor et Processor étendant la classe ContextNode. Un opérateur prend en entrée des messages et fournit en sortie un message. À droite, sont représentés les attributs de nœuds de contexte. Enfin, nous avons un message qui est composé de morceaux de message et de sous-messages.

3.3.5 Analyse sémantique

La section précédente présente un métamodèle du langage dédié COSMOS. Nous construisons un arbre d'interprétation à partir de l'arbre de syntaxe et respectant ce métamodèle. L'arbre d'interprétation est présenté dans la figure 3.10. Dans cet arbre, nous n'avons pas besoin de garder trace des éléments syntaxiques qui explicitent le parenthésage, les accolades, etc. Les symboles

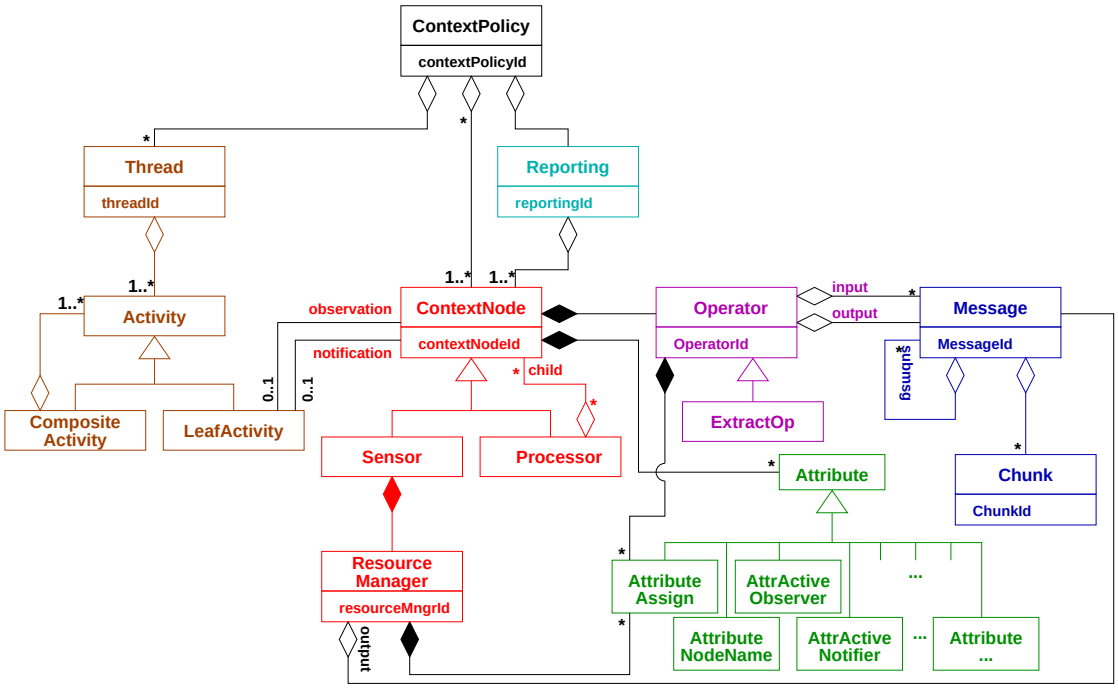

FIG. 3.9 – Métamodèle en UML de COSMOS

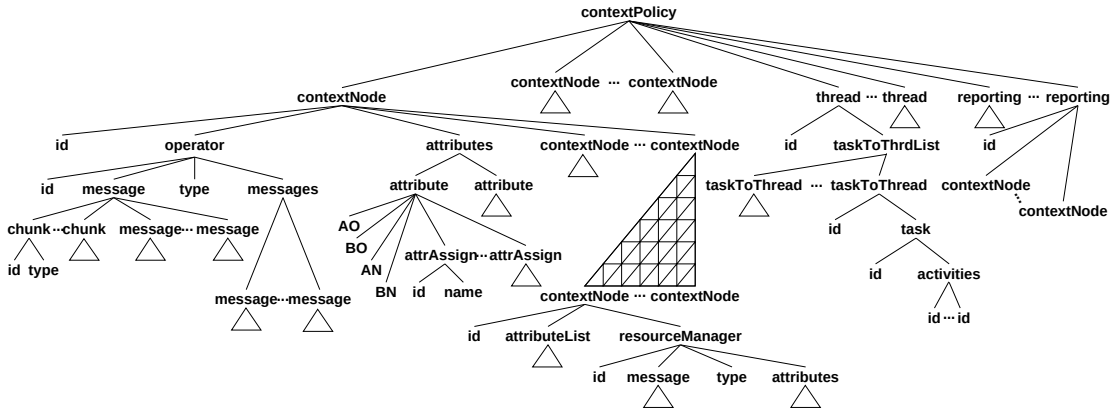

FIG. 3.10 – Arbre d'interprétation

terminaux de la syntaxe concrète (les virgules, les guillemets, etc.) ne sont plus représentés. Nous ne gardons que les symboles ou constructions pertinents permettant de faire l'analyse sémantique.

Au delà de la traduction vers FRACTAL ADL à partir des expressions du langage de la composition d'informations de contexte, l'analyse sémantique doit nous permettre également de vérifier certaines propriétés intéressantes, en particulier la détection d'interblocage. En effet, dans le graphe de nœuds de contexte, les communications peuvent s'opérer du bas vers le haut (notification) ou du haut vers le bas (observation). Un nœud de contexte peut être observateur bloquant ou notificateur bloquant. Les flots d'appels descendants à la recherche des informations de contexte et ceux ascendants peuvent se rencontrer, ce qui peut engendrer des risques d'interblocage. En effet, deux nœuds bloquants peuvent être accédés mutuellement. Il y a alors deux possibilités permettant de résoudre ce problème d'interblocage. La première possibilité est de mettre en place des algorithmes de détection d'interblocage. La deuxième possibilité est la traduction vers un langage synchrone. Nous laissons ce point en perspective de nos travaux de stage.

Une autre propriété intéressante à étudier dans l'analyse sémantique est le système de typage. En effet, tous les composants FRACTAL nœuds de contexte possèdent les mêmes interfaces. C'est donc un typage générique qui limite la vérification de type dans FRACTAL ADL. En particulier, il n'y pas de possibilité pour vérifier qu'un morceau de messages de tel type est fourni par un nœud enfant. COSMOS utilise en effet Dream [19] qui est une bibliothèque de composants FRACTAL pour la construction d'intergiciels orientés messages dynamiquement configurables. Les messages peuvent être distribués à des systèmes par publication/abonnement complexes. Dans Dream, un message est un ensemble fini de morceaux de messages (chunks) typés. Un chunk peut être ajouté, supprimé ou mis à jour via des opérations sur le gestionnaire de message. Donc, un chunk peut être absent lors d'une lecture, d'une suppression ou d'une mise à jour. Il peut être déjà présent lors d'un ajout. Enfin, il peut ne pas posséder le bon type lors d'une lecture. Dream Types [3] est un travail de recherche en cours dont le but est de concevoir un système de typage sur les messages pour Dream permettant de vérifier ces problèmes de cohérence sur les types de message. Le système de typage de DSL COSMOS pour les entrées/sorties des nœud de contexte pourrait ainsi s'inspirer de celui de Dream Types. C'est une seconde perspective de nos travaux de recherche.

3.4 Mise en œuvre

Après la spécification dans la section précédente, nous décrivons la mise en œuvre du langage. Tout d'abord, nous présentons une vue globale de la mise en œuvre du langage dans la section 3.4.1. Ensuite, la section 3.4.2 décrit des outils utilisés et ainsi que l'architecture du compilateur.

3.4.1 Vue globale

La structure du langage DSL COSMOS est relativement simple, donc facile à manipuler. La simplicité d'utilisation est bien sûr l'un des objectifs fixés dés la conception du langage. Cependant, cette simplicité d'utilisation est associée à une complexité grandissante du côté de la mise en œuvre. La difficulté réside notamment dans la partie analyse sémantique du langage.

Pour cela, il faut d'abord construire un arbre de syntaxe correspondant à la grammaire spécifier grâce à l'analyse lexicale et à l'analyse syntaxique (les étapes du compilateur sont illustrées dans la figure 3.11). Ce sont les deux premières étapes du compilateur. À partir de l'arbre de syntaxe, l'analyse sémantique peut être effectuée ainsi que la génération du code cible. Cela correspond à un processus de compilation classique. Dans notre cadre d'étude, l'arbre de

syntaxe est remanié et transformé afin de mieux capturer tous les concepts du domaine. Ainsi, au lieu d'effectuer des analyses directement sur l'arbre de syntaxe, nous effectuons des analyses sur l'arbre de syntaxe transformé pour pouvoir ensuite faire la vérification, la traduction et la transformation.

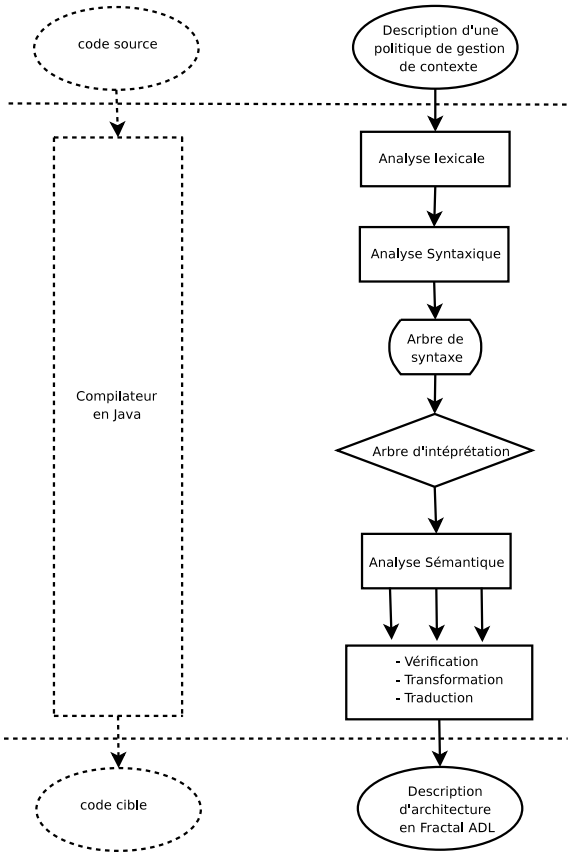

Fig. 3.11 – Étapes du compilateur

3.4.2 Outils utilisés

Étant donné que l'intergiciel COSMOS est programmé en Java, nous voulons réaliser un compilateur en Java pour une meilleure compatibilité et en même temps pour bénéficier de la portabilité de ce langage. Il existe plusieurs outils pour construire des compilateurs en Java. Dans l'étape de compilation, nous avons besoin tout d'abord de réaliser un analyseur lexical et syntaxique. La construction d'un analyseur syntaxique peut être automatisée grâce à des outils disponibles. Parmi les générateurs d'analyseurs syntaxiques[1] existants, nous avons choisi JavaCC[2] pour son efficacité et sa simplicité d'utilisation.

[1] http://catalog.compilertools.net/java.html
[2] https://javacc.dev.java.net/

JavaCC JavaCC est un outil *open source*, un générateur d'analyseur lexical et syntaxique de type LL(k) à la manière de LEX/YACC en C. C'est un outil qui lit les spécifications d'une grammaire écrites en BNF et génère un programme en Java. Ce programme, un analyseur descendant, lit et analyse les expressions du langage. Les productions écrites en JavaCC sont de la forme : `void Affectation() : {} {Identifiant() "=" Expressions() ";" }`. La production est `Affectation()`, une méthode, et la définition de la production composée se trouve entre les deux dernières accolades. Les symboles `Affectation()`, `Identifiant()` et `Expressions()` sont des non-terminaux. Les symboles terminaux sont « = » et « ; ». JavaCC fournit via d'autres outils le moyen de construction d'arbre de syntaxe et le débogage. En effet, JavaCC ne construit pas l'arbre de syntaxe automatiquement. Néanmoins, une fois que la grammaire est spécifiée dans JavaCC, il est facile de construire par la suite l'arbre de syntaxe correspondant. Pour cela, il y a deux possibilités : construire l'arbre en définissant des méthodes et algorithmes nécessaires à partir de la spécification ou utiliser des outils de génération d'arbre de syntaxe tel que JTB. La construction de l'arbre de façon manuelle n'est pas complexe, mais est très fastidieuse. Cela peut engendrer des erreurs potentielles. Notre choix est donc porté sur l'utilisation d'un outil déjà développé, ici JTB.

JTB[3] (*Java Tree Builder*) est un outil *open source* conçu pour être utilisé avec JavaCC. JTB permet de construire l'arbre de syntaxe. Il prend une grammaire spécifiée en JavaCC comme entrée et génère automatiquement :

- un ensemble de classes basées sur les productions de la grammaire. Chaque classe contient une méthode principale qui correspond à une production de la grammaire ;
- deux interfaces : Visitor et GjVisitor, un patron visiteur possèdant des méthodes par défaut permettant de faciliter le parcours dans l'arbre ;
- une grammaire `jtb.out.jj` contenant la grammaire JavaCC avec le code Java embarqué avec les annotations spécifique à JTB permettant de construire l'arbre de syntaxe durant la décomposition analytique.

Autres outils JUnit[4] est une bibliothèque de construction de tests unitaires pour le langage de programmation Java. Il peut être intégré dans Eclipse et Maven. Il permet de faire des test unitaires des expressions du langage.

Maven[5] est un logiciel *open source* pour la gestion et l'automatisation de la production des logiciels Java. Il permet de produire un logiciel à partir de ses sources, en optimisant les tâches réalisées à cette fin et en garantissant le bon ordre d'exécution des outils (JavaCC, Javac, Junit, etc.). Maven gère aussi les dépendances entre les modules constituant notre projet ainsi qu'avec les bibliothèques dont dépend COSMOS.

Subversion[6] est un logiciel *open source* de gestion version. Il permet à des utilisateurs distants de travailler ensemble sur les mêmes fichiers. Il s'appuie sur le principe d'un dépôt centralisé et unique dans lequel les utilisateurs peuvent déposer des données, les récupérer et publier leurs modifications. Subversion garde un historique des différentes versions des fichiers d'un projet, permettant ainsi le retour à une version antérieure en cas de besoin. Cet outil nous offre une meilleure gestion du travail en équipe.

[3] http://compilers.cs.ucla.edu/jtb/jtb-2003/
[4] http://junit.org/
[5] http://maven.apache.org/
[6] http://subversion.tigris.org/

3.5 Évaluation

Dans cette section, nous évaluons le DSL COSMOS par rapport aux critères définis dans la section 2.1 :

- facilité d'utilisation : le DSL cosmos a une structure relativement simple. Les constructions du langage sont appropriées au domaine, par exemple contextPolicy désigne effectivement une politique de gestion de contexte, ainsi que contextNode pour spécifier un nœud de contexte. Les notations appropriées de DSL COSMOS augmentent la lisibilité et la concision des programmes. Grâce aux idiomes avec l'opérateur de nœud de contexte, les utilisateurs peuvent aisément composer les informations de contexte ;
- facilité d'extension : le principe de séparation est utilisée dans la spécification de la grammaire de DSL COSMOS, notamment grâce à une analyse et une modélisation du domaine. Les constructions principales de la grammaire sont définies de façon indépendante en respectant une grammaire de la forme EBNF, ce qui facilite donc l'ajout d'autres constructions. Ces deux caractéristiques, séparations de construction et grammaire de la forme EBNF, permettent au langage d'être facilement extensible ;
- réutilisation systématique : il existe plusieurs types de réutilisation telles que définies dans la section 2.1. Nous ciblons en priorité la réutilisation des idiomes du domaine. Cette réutilisation est réalisable grâce à un modèle du domaine ;
- sûreté améliorée : DSL COSMOS met à disposition des utilisateurs des moyens de programmation restrictifs, ce qui limite les risques d'erreur de type syntaxique et également sémantique. Ainsi, la composition d'informations de contexte dans COSMOS est beaucoup plus sûre que celle décrite directement via le langage de description d'architecture Fractal ADL ;
- composabilité : DSL COSMOS permet de décrire des politiques de gestion de contexte qui sont des compositions d'informations de contextes de haut niveau. Les expressions de DSL COSMOS sont définies en composant les unes avec les autres. À titre d'exemple, une politique de gestion de contexte est composée d'un ensemble de rapports de messages, d'un ensemble de fils d'exécution et d'un ensemble de nœuds de contextes. Un nœud de contexte à son tour est composé d'autres nœuds de contexte ;
- partage : dans DSL COSMOS, l'appel d'une construction peut être fait par référence ou en donnant explicitement la définition correspondante. L'appel par référence permet de partager une définition, une information de contexte.

Les deux critères testabilité et optimisation sont à discuter tant sur la spécification que sur l'implantation. Dans le premier critère, la spécification, la grammaire peut être encore améliorée et complétée, notamment la spécification des espaces de nommage, de l'accès aux références et du système de typage. L'optimisation se focalise en particulier sur la mise en œuvre des algorithmes adaptés permettant de concevoir un compilateur efficace. Ces éléments manquants de DSL COSMOS constituent des perspectives de ce mémoire.

Chapitre 4

Conclusion et perspectives

Dans COSMOS, la composition d'informations de contexte est exprimée dans une architecture logicielle et cette architecture est ensuite projetée sur un graphe de composants logiciels. La composition d'informations de contexte exprimée à travers le langage de description d'architecture Fractal ADL est assez technique et prolixe rendant ainsi difficile cette composition pour les utilisateurs métiers, non spécialistes de la syntaxe Fractal ADL.

L'étude de l'état de l'art sur des langages de composition d'informations de contexte nous permet de classer ces langages en deux catégories. Dans la première catégorie, nous trouvons les langages orientés donnée dont la composition s'effectue lors des requêtes. Dans la deuxième catégorie, nous trouvons les langages orientés processus dont les expressions permettent la composition d'informations de contexte. Dans ces deux catégories, aucune solution ne fournit de manière suffisante la composition d'informations de contexte avec la notion de partage d'informations de contexte. Or, ces deux propriétés, la composition et le partage d'informations de contexte, sont fondamentales pour la description de politiques de gestion de contexte dans COSMOS.

Dans ce mémoire, nous proposons une approche nouvelle permettant la composition aisée des informations de contexte grâce à un nouveau langage dédié au domaine de la composition d'informations de contexte. Avec ce langage dédié, les utilisateurs déclarent la composition d'informations de contexte en manipulant les concepts du domaine COSMOS. En effet, DSL COSMOS permet la composition d'informations de contexte pour COSMOS et intègre la notion de partage grâce à sa structure et ses constructions de haut niveau. Nous avons réalisé une grammaire spécifiant la politique de gestion de contexte, ainsi qu'une première ébauche d'un analyseur syntaxique du compilateur de DSL COSMOS.

Notre première perspective est de compléter l'implantation pour avoir un compilateur capable de convertir la description de politiques de gestions de contexte via le DSL COSMOS vers Fractal ADL. Dans une politique de gestion de contexte, deux nœuds bloquants peuvent être accédés mutuellement. La mise en place d'un système de détection d'interblocage constitue notre deuxième perspective. Enfin, le système de typage du composant Fractal pour les messages n'est pas assez efficace à cause son aspect générique. Des problèmes de cohérence de typage de message existent : l'absence d'un morceaux dans un message, le message ne possède pas le bon type, etc. La mise en place d'un système de typage tel que Dream Type constitue notre troisième perspective.

Annexe A

A.1 Syntaxe de la grammaire du langage dédié COSMOS

Les mots réservés de la grammaire sont les suivants :

```
contextpolicy     ressourcemgr     task
sensor            message          reporting
processor         operator         thread
chunk             extract
```

```
Définition de politique de contexte :
< contextPolicy >        ::=   CONTEXTPOLICY contextPolicyId "="
                                  (contextNodeId | threadId | reportingId)
                                     { "," (contextNodeId | threadId |
reportingId) }
CONTEXTPOLICY            ::=   "contextpolicy"
contextPolicyId          ::=   IDENT

Définition de contexte node (noeud de contexte):
< contextNode >          ::=   < sensor > | < processor >

Parties fonctionnelles:
Définition de sensor (capteur):
< sensor >               ::=   SENSOR sensorMgrID "="
                                  (resourceMgrId | "(" < resourceMgrDef > ")")
                                     "[" < attributeCNList > "]" EOF
SENSOR                   ::=   "sensor"
sensorceMgrID            ::=   IDENT
resourceMgrID            ::=   IDENT
< resourceMgr >          ::=   RESOURCEMGR resourceMgrId "="
                                  < resourceMgrDef > EOF
< resourceMgrDef >       ::=   ("{" < chunkList > "}" | messageId)
                                     resourceMgrClass "[" < attributeRMList"]"
RESOURCEMGR              ::=   "resourcemgr"
< chunkList >            ::=   (< chunkDef > | chunkId) { "," (< chunkDef > |
chunkId) }
< chunk >                ::=   CHUNK chunkID "=" < chunkDef > EOF
< chunkDef >             ::=   chunkClass
CHUNK                    ::=   "chunk"
chunkID                  ::=   IDENT
< attributeCNList >      ::=   attributeCN { "," attributeCN }
```

```
attributeCN              ::=   "BO" | "AO" | "AN" | "BN" | < attributeASSIGN >
chunkClass               ::=   IDENT
message                  ::=   MESSAGE messageId "=" "{" <chunkSubMessageList>
"}"
MESSAGE                  ::=   "message"
< chunkSubMessageList >  ::=   (chunkId | messageId) { "," (chunkId |
messageId) }
messagesId               ::=   IDENT
resourceMgrClass         ::=   IDENT
< attributeRMList >      ::=   attributeASSIGN { "," attributeASSIGN }
< attributeASSIGN >      ::=   attributeID "=>" attributeValue
attributeID              ::=   IDENT
attributeValue           ::=   IDENT
IDENT                    ::=   CHAR { CHAR | chiffre_0 | "0" }
CHAR                     ::=   "a"-"z" | "A"-"Z" | "."
EOF                      ::=   ";"

P.ex.,
sensor Wifi = Wifi [BO,AO];
sensor Battery = ({TimeleftChunk,ChargeLevelChunk,StatusChunk,IsChargingChunk}
                    cosmos.saje.BatteryRM) [BO, AO, AN];
resourcemgr WiFiRM = {LinkQualityChunk,ESSIDChunk,accessPointAddressChunk...}
                          cosmos.saje.WiFiRM [resource => eth1];

< operator >             ::=   OPERATOR operatorID "=" < operatorDef > EOF
< operatorDef >          ::=   (< chunkList > | messageId)
                                 operatorClass ť(ť< chunkListList >ť)ť
OPERATOR                 ::=   "operator"
operatorID               ::=   IDEN
operatorClass            ::=   IDENT
< chunkListList >        ::=   < chunkList > { ";" chunkList }

P.ex.,
operator BatteryLifeSpanCO = { BatteryLifeSpanChunk }
                               cosmos.saje.battery.operator.BatteryLifeSpanCO
                                  ({ StatusChunk },{ TimeLeftChunk });


Définition de processor (processeur):
processor                ::=   PROCESSOR processorID "="
                                  (operatorId |  "("< operatorDef >")")
                                     [< attributeCNList >] "("< childList >")"
                                        EOF

PROCESSOR                ::=   "processor"
proccessorId             ::=   IDENT
< childList >            ::=   < child > { "," < child > }
< child >                ::=   (processorId | sensorId)
                                  ["." EXTRAOP < chunkList > ]
EXTRACTOP                ::=   "extract"


P.ex.,
```

```
processor BatteryLifespan = BatteryLifeSpanCO(BatteryStatus,BatterytimeLeft);
processor AverageLinkQuality = AverageCP[nbSamples => 5]
                                (WiFI.extract{LinkQualityChunk});
precessor Proc = ({chunk1} identOpClass({chunk2,chunk},{chunk4}))
                    [attr=>value](child1,child2.extract{chunk4});
```

```
Parties extra fonctionnelles :
Définition de task (tâche):
task                      ::=   TASK taskName "=" < activityMgrList >;
< activityMgrList >       ::=   activityMgr { "," activityMgr }
activityMgr               ::=   IDENT

P.ex.,
task ressource = BluetoothConnectivity,WifiMgr,BatteryMgr ;
Définition de thread :
thread                    ::=   THRED threadId "=" < taskToThreadList >
threadId                  ::=   IDENT
< taskToThreadList >      ::=   < taskToThread > { "," < taskToThread > }
< taskToThread >          ::=   taskToThreadID "[" period "]"
period                    ::=   intValue
intValue                  ::=   chiffre_0 { chiffre_0 | "0" }
chiffre_0                 ::=   "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
"9"

P.ex.,
thread RFIDThread = RFIDTasks[5000];
thread GroupThread = GroupMembershipService[10000],FailureDetector[3000] ;


Définition de reporting (report de messages):
reporting                 ::=   reportingID "=" < reportNodeList >
reportingID               ::=   IDENT
< reportNodeList >        ::=   reportNode { "," reportNode }
< reportNode >            ::=   contextNodeID [FPath FPahtOp]
contextNodeName           ::=   IDENT
CPathOP                   ::=   "/" | "//" | "." | "@" | "*"

P.ex.,
reporting GroupReport = GroupMembershipService,DisconnectionDetector ,
                                FailaureDetector/descendant-or-self::* ;
```

Bibliographie

[1] G.D. Abowd, A.K. Dey, P.J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a Better Understanding of Context and Context-Awareness. In *Proc. of the 1st International Symposium on Handheld and Ubiquitous Computing*, volume 1707 of *Lecture Notes in Computer Science*, pages 304–307, London, UK, 1999. Springer-Verlag.

[2] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley, 1998.

[3] Philippe Bidinger, Matthieu Leclercq, Vivien Quéma, Alan Schmitt, and Jean-Bernard Stefani. Dream types : a domain specific type system for component-based message-oriented middleware. *SIGSOFT Softw. Eng. Notes*, 31(2) :2, 2006.

[4] M. Botts and A. Robin. Sensor Model Language (SensorMl). Technical report, University of Alabama in Huntsville, 17-07-2007.

[5] C. Boutros Saab, X. Bonnaire, and B. Folliot. PHOENIX : A Self Adaptable Monitoring Platform for Cluster Management. *Cluster Computing*, 5(1) :75–85, January 2002.

[6] É. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and its Support in Java. In *Proc. 7th International Symposium on Component-Based Software Engineering*, pages 7–22, May 2004.

[7] É. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proc. 7th ECOOP Workshop on Component Oriented Programming*, Malaga, Spain, June 2002.

[8] L. Capra, W. Emmerich, and C. Mascolo. CARISMA : Contex-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10) :929–945, October 2003.

[9] N.H. Cohen, H. Lei, P. Castro, J.S. Davis, and A. Purakayastha. Composing Pervasive Data Using iQL. In *Proc. of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, pages 94–105, Washington, DC, USA, 2002. IEEE Computer Society.

[10] D. Conan, R. Rouvoy, and L. Seinturier. COSMOS : composition de nœuds de contexte. *Technique et Science Informatiques*, 2008. À paraître.

[11] J. Coutaz, J.L. Crowley, S. Dobson, and D. Garlan. The disappearing computer : Context is Key. *Communications of the ACM*, 48(3) :49–53, 2005.

[12] P.-C. David. *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.

[13] P.-C. David and T. Ledoux. WildCAT : a generic framework for context-aware applications. In *Proc. of the 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 1–7, Grenoble, France, 2005. ACM.

[14] A.K. Dey, D. Salber, and G.D. Abowd. 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*, 16(2–4) :97–166, 2001.

[15] S. Frølund and J. Koistinen. QML : A Language for Quality of Service Specifiation. Technical report, Hewlett-Packard Laboratories, Palo Alto, USA, 1998.

[16] P.D. Haghighi, A. Zaslavsky, and S. Krishnaswamy. An Evaluation of Query Languages for Context-Aware Computing. In *Proc. of the 17th IEEE International Conference on Database and Expert Systems Applications*, pages 455–462, Washington, DC, USA, 2006.

[17] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania (USA), November 1990.

[18] M. Leclercq, A.E. Özcan, V. Quéma, and J.-B. Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *Proc. 29th ACM International Conference on Software Engineering*, (USA), May 2007.

[19] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. Dream : a component framework for the construction of resource-aware, reconfigurable moms. In *ARM '04 : Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 250–255, New York, NY, USA, 2004. ACM.

[20] M. Mernik, J. Heering, and A.M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Survey*, 37(4) :316–344, 2005.

[21] D. Nicklas, M. Grossmann, and T. Schawarz. NexusScout : An Andwanced Location-Based Application on a Distributed, Open Mediation Plateform.

[22] M. Román, H. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt. Gaia : A Middleware Platform for Active Spaces. *SIGMOBILE Mobile Computing Communication Review*, 6(4) :65–67, 2002.

[23] R. Rouvoy, D. Conan, and L. Seinturier. Software Architecture Patterns for a Context Processing Middleware Framework. *IEEE Distributed Systems Online*, 9(6), June 2008.

[24] C. Szyperski. *Component Software : Beyond Object-Oriented Programming, 2nd edition.* Addison-Wesley, 2002.

[25] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : an annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, 2000.

[26] T. Winograd. Architectures for Context. *Special issue on context-aware computing in the Human-Computer Interaction Journal*, 16(2–4) :401–419, 2001.

[27] S.S. Yau, F. Karim, Y. Wang, B. Wang, and S.K.S. Gupta. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing*, 1(3) :33–40, 2002.

[28] M.M Zloof. Query-By-Example : A Data Base Language. *IBM Systems Journal*, 16(4) :324–343, 1977.