UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UNE APPROCHE POUR LA MODÉLISATION, LE RAISONNEMENT ET L'ADAPTATION D'APPLICATIONS DÉPENDANTES DU CONTEXTE

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

ANNE MARIE AMJA

OCTOBRE 2017

REMERCIEMENTS

J'exprime mes profonds remerciements à mes directeurs de recherche, les professeurs Abdellatif Obaid et Hafedh Mili, qui m'ont encadré tout au long de cette thèse. Je les remercie pour l'aide compétente qu'ils m'ont apportée, pour leurs patiences, pour le temps qu'ils m'ont consacré à répondre à mes questions et pour leurs suggestions lors de la réalisation de ce projet. Sans leurs encadrements minutieux, cette thèse n'aurait certainement pas pu aboutir.

Je souhaite remercier les membres du jury, les professeurs Hafedh Mili, Wessam Ajib, Naouel Moha et Azzedine Boukerche, pour l'intérêt qu'ils ont porté à mon travail de recherche en acceptant d'examiner ma thèse et de l'enrichir par leurs propositions.

Je tiens à remercier les professeurs Timothy Walsh et Louise Laforest de m'avoir fait profiter de leurs connaissances sur l'algorithmique et pour ce qu'ils m'ont appris et leur intérêt envers les étudiants.

Je remercie sincèrement le professeur Petko Valtchev pour ses remarques et ses conseils précieux sur le web sémantique, et pour sa coopération.

Je remercie également tous les professeurs associés aux laboratoires LATECE et TRIME qui m'ont conseillé sur divers aspects, particulièrement les professeurs Guy Tremblay et Wessam Ajib.

Je tiens à exprimer ma reconnaissance au Fond de recherche du Québec - Nature et Technologies (FRQNT) pour l'appui financier qu'il m'a été attribué.

J'adresse ma gratitude à mes chers amis pour leur soutien et leur disponibilité.

Je remercie mes parents et mes frères pour leurs support et encouragement dans ce projet de recherche.

DÉDICACE

À mes chers parents, Norma et Zouhair, pour tous leurs sacrifices, leur amour, leur tendresse, leur soutien et leurs prières tout au long de mes études.

À mes frères, Gabriel et Alexandre, pour leur appui et leur encouragement.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

Avec l'avènement des appareils mobiles et des réseaux mobiles et sans fil, divers paradigmes de l'informatique tels que l'informatique ubiquitaire, ambiante, sensible au contexte et autonome ont pris naissance. Ces paradigmes partagent tous la notion de contexte qui joue un rôle important et révèlent des comportements intelligents qui entourent les utilisateurs de ces environnements. La nature du contexte peut prendre plusieurs formes, à savoir, *spatial* (localisation, orientation, vitesse) *temporel* (date, temps, saison), *environnemental* (lumière, température ambiante, bruit) et *social* (utilisateurs, ses préférences et activités) (Brézillon et Gonzalez, 2014). En conséquence, l'implication du contexte entraine l'apparition d'applications sensibles au contexte.

Dans la présente thèse, nous proposons une méthode de modélisation de contexte qui se penche sur l'analyse relationnelle de concepts pour les systèmes dépendants du contexte. Nous nous sommes intéressés à proposer un modèle qui fonctionne main en main avec un moteur de raisonnement. Pour ce faire, nous avons utilisé la logique descriptive puisqu'il existe des règles de correspondance entre les entités de l'analyse relationnelle de concepts et la logique de description. Nous montrons notre approche grâce aux outils suivants : RCAExplore, Protégé, et le raisonneur Pellet.

Nous présentons également deux techniques d'adaptation. D'une part, nous proposons une adaptation par composition. Ainsi, nous présentons un modèle sémantique, basé sur des composants, qui permet de spécifier le comportement d'un système en fonction de ses composants. La composition décrit une architecture exprimée à l'aide d'opérateurs qui expriment le comportement des actions que chaque composant peut effectuer et leurs effets sur le comportement global de l'ensemble. Nous exploitons également l'utilisation de contraintes pour restreindre le comportement global d'un système selon le contexte. D'autre part, nous proposons une adaptation basée sur l'inclusion ou l'exclusion de *features* d'un modèle de variabilité, c'est-à-dire les fonctionnalités qu'un système peut offrir selon le contexte actuel. Nous modélisons le modèle de variabilité en ontologie et montrons l'ajout et la modification de configurations. Nous validons le modèle de variabilité et ses configurations avec le raisonneur Pellet. Nous avons construit des règles de contexte en utilisant un langage de règle pour le web sémantique, plus précisément SWRL, dans le but d'associer ces règles à leurs configurations correspondantes.

Finalement, nous proposons d'utiliser le modèle de référence de la boucle de contrôle MAPE-K pour effectuer soit l'adaptation par composition, soit l'adaptation par *features* d'un système. Nous avons implanté les différentes activités de cette boucle en Prolog, à savoir les activités d'observation, d'analyse, de planification et d'exécution, et leurs interactions avec la base de connaissances, pour les deux méthodes d'adaptation proposées.

Mots-clés : modélisation de contexte, raisonnement sur le contexte, composants, opérateurs de composition, modèle de variabilité, sémantique, auto-adaptation, boucle de contrôle.

CHAPITRE I

INTRODUCTION

Avec l'avènement des technologies d'appareils mobiles et des réseaux mobiles et sans fils, la nouvelle vague de l'ère de l'informatique se trouve en dehors du domaine traditionnel de bureau. Au cours des dernières années, nous avons vu naître divers paradigmes qui entourent, empreignent et servent intelligemment les utilisateurs.

D'abord, nous retrouvons le paradigme de l'*informatique mobile* qui sous-entend le concept d'utilisation d'appareils portables par des utilisateurs et consiste à offrir des services accessibles en tout temps et en tout lieu depuis toute sorte d'appareils mobiles (Nosrati *et al.*, 2012). Ensuite, nous avons le paradigme de l'*informatique ubiquitaire*, appelée aussi l'*informatique pervasive*, qui retrouve à sa disposition d'innombrables appareils informatiques, partout et sans cesse disponibles, dotés d'une intelligence et de capacités de communiquer et interagir entre eux, et avec l'environnement et les personnes, de façon transparente, et capables de se fondre dans l'arrière-plan (Weiser, 2002). Quant à l'*informatique ambiante*, cette dernière intègre de nouveaux dispositifs communicants et intelligents dans les objets de la vie quotidienne, et, en conséquence, permet la mise en oeuvre d'applications servant à interagir avec l'environnement physique et l'utilisateur pour offrir des services adaptés (Cook *et al.*, 2009). Nous avons également le paradigme de l'*informatique sensible au contexte* qui se caractérise par des logiciels qui découvrent et réagissent à des informations contextuelles telles que la localisation de l'utilisateur, le temps, les utilisateurs et les appareils voisins, et les activités de l'utilisateur (Musumba et

Nyongesa, 2013) alors que le paradigme de l'*informatique autonomique* se penche sur des systèmes capables de s'auto-gérer dont les objectifs sont de permettre aux systèmes d'évoluer de façon autonome en évitant l'intervention humaine, de réparer les comportements indésirables et de s'adapter aux changements imprévus de l'environnement (Lalanda *et al.*, 2013). Par ailleurs, l'*Internet des objets* est une composante majeure de l'Internet qui permet des applications qui s'étalent dans plusieurs domaines. Il permet l'identification électronique, et ceux des appareils mobiles sans fil, de récupérer, stocker, transférer et traiter des données entre les mondes physiques et virtuels (Benghozi *et al.*, 2011).

D'après ces descriptions brèves des paradigmes, nous constatons le chevauchement que ces technologies ciblent de créer. Les applications de l'informatique mobile, ubiquitaire, ambiante, sensible au contexte, autonomique et de l'Internet des objets manifestent des comportements intelligents et entourent les utilisateurs de ces environnements. Nous remarquons également que ces paradigmes ont divers aspects en commun portant sur le *contexte* et l'*adaptation* des applications dépendantes du contexte. D'une part, il existe plusieurs définitions de ce qui forme un contexte et celle qui est plus fréquemment citée est due à Dey et Abowd (1999): "toute information qui puisse être utilisée pour caractériser la situation d'entités (une personne, un endroit ou un objet) et qui sont considérées pertinentes à l'interaction entre un utilisateur et une application, incluant l'utilisateur et l'application même". D'autre part, les applications sensibles au contexte font face à des environnements hétérogènes et dynamiques, et, par conséquent, elles doivent s'adapter en temps réel, de manière autonome, au contexte.

Nous décrivons dans ce qui suit les enjeux rencontrés dans les applications dépendantes du contexte et présentons notre problématique ainsi que les objectifs établis. Nous expliquons la méthodologie utilisée dans le cadre de ce travail et indiquons la structure du document.

1.1 Enjeux

Bien que les applications sensibles au contexte aient été étudiées depuis un certain nombre d'années, les enjeux suivants se sont révélés.

1.1.1 Acquisition du contexte

Les informations contextuelles peuvent être acquises à partir de différents types de capteurs qui peuvent être :

- des *capteurs physiques* mesurant des données physiques comme la lumière, la température, le son, et la localisation. Ce type de capteurs peut fournir des données diverses telles que des données physiologiques et/ou environnementales (Boudra, Obaid et Amja, 2014).
- des *capteurs virtuels* fournissant des mesures indirectes qui sont par elles-mêmes non mesurables physiquement en combinant les données captées d'un groupe de capteurs physiques hétérogènes comme capter les entrées d'un calendrier, ou
- des *capteurs logiques* permettant d'agréger des informations de différentes sources, c'est-à-dire en combinant des capteurs physiques et virtuels dans le but de construire des bases de données de contextes.

Il existe plusieurs paramètres dans notre entourage pouvant faire partie du contexte. Pour qu'un système sensible au contexte puisse distinguer les informations de contexte du bruit, il est important d'introduire un certain niveau d'intelligence. Il faut également être en mesure d'identifier ce que représente le contexte et d'acquérir au bon moment et au bon endroit.

1.1.2 Interprétation du contexte

Les capteurs fournissent généralement des données qui ne sont pas adaptées à un usage immédiat par des systèmes adaptatifs. Les données contextuelles obtenues

depuis les capteurs doivent être interprétées, c'est-à-dire analysées et transformées, vers d'autres formats de haut niveau qui sont plus faciles à manipuler et à utiliser. Ces données brutes sont transformées par plusieurs opérations telles que l'extraction, le filtrage, l'abstraction et l'agrégation.

Toutefois, l'utilisation de plusieurs sources de contexte engendre des conflits (Bellavista *et al.*, 2012). Ces sources peuvent fournir des données contradictoires ou aboutir à des situations imprécises d'où le besoin d'une certaine intelligence pour interpréter et résoudre ces conflits. De plus, il faut agréger les données appropriées de contexte capté afin de fournir une réponse compréhensive à un contexte particulier. Les données captées peuvent donc se retrouver dans différents formats non compatibles; ce qui rend l'utilisation des informations contextuelles difficiles. Il faut donc être en mesure de détecter les données contradictoires, d'introduire un mécanisme de raisonnement dans l'interprétation des données et d'harmoniser leur présentation.

1.1.3 Stockage de contexte

Une fois les données capturées ou interprétées, elles peuvent être stockées pour une utilisation ultérieure dans des bases de données de contexte. Ceci pose plusieurs défis:

- Définir les techniques de stockage d'information utilisées de manière à ce que les requêtes et les mises à jour se gèrent efficacement sans perdre la sémantique des données.
- Déterminer la pertinence des informations pour les artéfacts qui interagissent avec les services.
- Déterminer la durée de vie de l'information volatile dans la base de données.
- Traiter les données de contexte au niveau des stratégies de modélisation utilisées.

Ce stockage peut se faire dans des bases de données réparties et mobiles dont les données pertinentes aux contextes de géolocalisation (Amja, Obaid et Seguin, 2011; Obaid, Amja, Mili et Seguin, 2012).

1.1.4 Architectures

Les systèmes sensibles au contexte peuvent être développés de plusieurs façons. L'approche dépend des critères et conditions exigées tels que la localisation des capteurs (locaux ou distants), le nombre d'utilisateurs potentiels, les ressources disponibles dans des appareils et la facilité d'étendre le système (Baldauf *et al.*, 2007).

La méthode choisie pour acquérir les données du contexte est importante au niveau de la conception d'un système sensible au contexte puisqu'elle définit l'approche architecturale du système (Baldauf *et al.*, 2007). Plusieurs approches ont été présentées sur la façon d'acquérir les informations contextuelles telles que l'utilisation d'un serveur de contexte ou d'un modèle d'accès direct d'une application aux données brutes des capteurs (Baldauf *et al.*, 2007).

Néanmoins, les approches traditionnelles utilisées pour la conception des systèmes sont souvent difficiles et complexes à adapter selon le contexte (Dargie, 2009). Les architectures sont souvent spécifiques à un domaine précis et pour un environnement donné (Mamo et Ejigu, 2014; Miraoui *et al.*, 2008). Toutefois, elles peuvent être généralisées (Loke, 2006; Mamo et Ejigu, 2014). Ces modèles s'articulent autour de la gestion du contexte, c'est-à-dire l'acquisition, la modélisation, le stockage et le raisonnement, et ce, sans considérer l'aspect d'adaptation. Par conséquent, il faudrait introduire un modèle architectural qui tient compte de cet aspect d'adaptation d'un système selon le contexte dynamique de l'utilisateur et de son environnement.

1.1.5 Modélisation du contexte

La modélisation de contexte réfère souvent à la représentation de contexte. Les approches de modélisation de contexte sont soit statiques, soit dynamiques. Les modèles statiques ont un ensemble d'informations contextuelles prédéfini qui est recueilli et stocké alors que les modèles dynamiques mettent à jour les informations contextuelles en temps réel.

Il n'existe pas de standard spécifiant quel type d'information nécessite d'être considérée pour modéliser le contexte. Par contre, certains critères ont été proposés tels que l'hétérogénéité, la mobilité, les liens et dépendances, l'imperfection, le raisonnement, l'historique du contexte, un formalisme de modélisation facile à utiliser et le prétraitement de contexte (Bettini *et al.*, 2010).

1.1.6 Raisonnement sur le contexte

Le raisonnement sur le contexte s'explique par le processus de déduction de contexte de haut niveau à partir d'un ensemble de contextes et de détection d'inconsistances possibles dans les informations contextuelles.

D'après Bikakis *et al.* (2008), le besoin d'effectuer le raisonnement dans les applications sensibles au contexte découle de l'imperfection et de l'incertitude des données contextuelles. Henrickson et Indulska (2004) caractérisent 4 types d'imperfections d'informations contextuelles :

- *Inconnue* : Les données contextuelles ne sont pas disponibles en tout temps lorsque des situations de problèmes de connectivité se produisent.
- *Ambiguïté* : Les données contextuelles sont ambigües lorsqu'elles proviennent de sources multiples.
- *Imprécision* : Les données contextuelles sont souvent imprécises lorsqu'elles proviennent des capteurs.

- *Erronée* : Les données contextuelles sont sujettes à des erreurs causées par l'humain ou le matériel.

Le rôle du raisonnement dans ces cas d'imperfection de données contextuelles consiste à détecter les erreurs possibles, prédire les données manquantes et décider sur la qualité et la validité des données captées (Bikakis *et al.*, 2008).

Étant donné que les informations contextuelles sont volatiles, il devient nécessaire de gérer des mises à jour de connaissances dynamiques (Bikakis *et al.*, 2008). Par conséquent, le raisonnement sur le contexte devient un défi. Le contexte devrait donc être représenté sous un format sémantique permettant de partager et réutiliser cette représentation pour faciliter le raisonnement sur le contexte.

1.1.7 Adaptation

L'adaptation est un processus de modification d'une entité suite à un changement dans le contexte. Elle se caractérise par la capacité de changer son comportement et de continuer son exécution dans des environnements où le contexte est différent. Nous retrouvons plusieurs types d'adaptation tels que l'adaptation statique ou dynamique, et l'adaptation centralisée ou distribuée.

Avec l'adaptation statique, plusieurs versions de la même ressource avant son exploitation sont préparées. Ce type d'adaptation est apparu suite à la sortie des appareils mobiles afin de leur permettre d'utiliser des ressources initialement destinées aux ordinateurs. L'avantage de cette technique se situe au niveau de sa simplicité. Toutefois, elle souffre de la difficulté à déduire de nouveaux contenus en fonction des contextes.

Quant à l'adaptation dynamique, elle établit les transformations des ressources au cours de l'utilisation de façon automatique. Toutefois, compte tenu de la difficulté à

transformer le contenu de manière totalement dynamique, elle risque de produire des résultats partiels, voire erronés.

L'adaptation peut également s'appliquer à une ressource locale (centralisée) ou distante (répartie). L'approche centralisée facilite la gestion des ressources alors que l'approche distribuée est plus complexe et nécessite des mécanismes de gestion et de synchronisation d'adaptation, ce qui alourdit une architecture de façon générale.

Dans le cas des applications dépendantes du contexte, l'adaptation se fait au niveau comportemental et/ou architectural du système.

L'adaptation comportementale implique des changements du comportement d'un système, ce qui nécessite de lui ajouter ou lui retrancher des fonctionnalités.

L'adaptation architecturale repose sur la modification de la structure du système lui-même. Ce type de changement peut impliquer dans le cas des systèmes à base de composants de le reconfigurer et de modifier son architecture.

1.2 Problématique

Au regard des enjeux présentés précédemment, il n'existe pas un modèle global de contexte qui soit utilisable par toute application sensible au contexte. Compte tenu du nombre et des types d'éléments impliqués, nous pensons que les applications dépendantes du contexte ont besoin d'une méthodologie qui leur est propre. Nous nous intéressons aux méthodes permettant de modéliser le contexte et de raisonner sur le contexte en permettant aux systèmes de s'adapter dynamiquement.

Nous désirons donc, par la présente recherche, résoudre les problèmes suivants en nous intéressant aux questions suivantes:

1) La modélisation du contexte:

   a. Comment modéliser les informations contextuelles en tenant compte des relations parmi ces informations?
   b. Quelles sont les informations contextuelles qui peuvent bien modéliser le contexte spécifique pour un domaine donné?

2) Le raisonnement sur le contexte :
   a. Comment raisonner sur les informations contextuelles?

3) L'adaptation d'applications dépendantes du contexte
   a. Comment un système peut-il s'adapter au niveau comportemental et architectural en tenant compte du contexte dynamique de l'utilisateur et de son environnement?
   b. Comment prendre en considération le changement d'information, et comment réagir à de tels changements, si nécessaire?

1.3 Objectifs

À partir des enjeux identifiés précédemment, nous visons les objectifs suivants dans le cadre de ce travail :

- Nous visons à introduire une méthodologie de modélisation du contexte qui permet d'établir des liens entre les informations contextuelles et les services. Les données contextuelles sont modélisées sans nécessairement prendre en considération leurs liens avec les services qui en dépendent.
- Nous voulons introduire une méthodologie de raisonnement sur le contexte qui permet de déduire des connaissances et d'inférer des situations contextuelles qui ne dépendent pas de la modélisation du contexte, mais qui fonctionne main en main avec cette dernière. La méthode de raisonnement choisi dans le passé dépend de la méthode de modélisation sur le contexte pour un domaine précis. Elle doit prendre en considération la volatilité des données contextuelles.
- Nous voulons introduire une méthodologie générale d'adaptation des services qui tiendrait compte des changements de contexte et de l'environnement de

l'utilisateur. Les méthodes d'adaptation des services des applications dépendantes du contexte sont souvent proposées pour des domaines précis.

1.4 Méthodologie

Dans le cadre de cette thèse, notre méthodologie s'articule autour des éléments suivants :

- *Modélisation du contexte*: Nous proposons une nouvelle méthode de modélisation du contexte en utilisant l'analyse relationnelle de concepts, une extension de l'analyse formelle de concept pour modéliser le contexte.
- *Raisonnement sur le contexte* : Nous utilisons la logique descriptive comme méthode de raisonnement sur le contexte vu qu'il existe des règles de correspondance entre les éléments de l'analyse relationnelle de concepts et la logique descriptive.
- *Modèle sémantique d'adaptation basé sur des composants* : Nous proposons un modèle sémantique d'adaptation qui permet aux architectures et aux comportements des systèmes de se modifier en fonction du contexte de leur utilisation. Nous utilisons des règles qui expriment la sémantique de certaines opérations que chaque composant peut avoir et leurs effets sur le comportement global de l'ensemble.
- *Modèle de variabilité* : Nous gérons la variabilité du contexte et les fonctionnalités d'un système. Pour ce faire, nous utilisons un modèle de variabilité dans le but de déterminer les fonctionnalités d'un système nécessitant d'être activés ou désactivées à partir d'une boucle de contrôle.

1.5 Structure du document

La thèse est organisée selon les chapitres suivants :

- Chapitre 2 présente une revue critique de la littérature sur les applications dépendantes du contexte et l'adaptation.
- Chapitre 3 expose une approche de modélisation du contexte basée sur l'analyse relationnelle de concepts, soit une extension de l'analyse formelle de concepts.
- Chapitre 4 montre le raisonnement sur le contexte avec la logique descriptive en utilisant des règles de correspondance entre les éléments de l'analyse relationnelle de concepts et la logique descriptive.
- Chapitre 5 présente un mécanisme de vérification de la cohérence d'un modèle de variabilité représenté sous forme d'ontologie.
- Chapitre 6 propose deux techniques d'adaptation des applications dépendantes du contexte : un modèle sémantique d'adaptation basé sur des composants et une adaptation basée sur l'activation des *features*.
- Chapitre 7 propose deux méthodes d'adaptation des applications dépendantes du contexte : adaptation par composition et adaptation par la variabilité. La première technique repose sur le modèle sémantique présenté dans le chapitre 6. Le second modèle se base sur l'activation et la désactivation de *features* présenté dans le chapitre 5.
- Chapitre 8 conclue le travail présenté dans cette thèse et discute des travaux futurs basés sur cette thèse.

CHAPITRE II

ÉTAT DE L'ART

2.6 Introduction

Les questions relatives à la notion de contexte, sa modélisation, son raisonnement ainsi que les techniques d'adaptation ont fait l'objet de plusieurs travaux de recherche.

Dans ce chapitre, nous présentons une revue de la littérature sur les travaux qui nous paraissent les plus pertinents à notre problématique en vue de les évaluer. Les sections 2.2 à 2.4 introduisent la notion de contexte, les types de contexte et la notion de sensibilité au contexte.

Nous considérons également quelques aspects qui ne sont pas directement reliés à notre problématique tels que l'acquisition (section 2.4) et le prétraitement de contexte (section 2.5).

Ensuite, nous présentons les divers modèles de contexte et raisonnement sur le contexte dans les sections 2.6 et 2.7, respectivement.

Dans les sections 2.9 à 2.11, nous élaborons l'utilisation des boucles de contrôle à des fins d'adaptation de systèmes dépendants du contexte, et nous présentons des exemples de modèles d'auto-adaptation inspirés de ces boucles.

En ce qui concerne les techniques d'adaptation, nous présentons 3 approches dans la section 2.12 : 1) une approche basée sur les lignes de produits logiciels, 2) une

approche basée sur l'algèbre des processus et 3) une approche basée sur les langages de coordination.

Nous terminons ce chapitre en présentant des exemples de modèles d'adaptation dans la section 2.13.

2.7 Notion de contexte

La notion de contexte a été définie dans plusieurs travaux et divers domaines tels que l'intelligence artificielle. Le terme *contexte* a été utilisé la première fois dans le travail de Schilit et Theimer (1994). Ils mentionnent 3 aspects importants reliés au contexte « where you are, who you are with, and what resources are nearby » (Schilit et Theimer, 1994). Dans un autre travail,Schilit *et al.* (1994) détaillent ces trois aspects par un changement continu de l'environnement. L'environnement fait référence à trois perspectives:

- *Environnement de l'ordinateur* : processeurs disponibles, appareils accessibles, interface, capacité du réseau, connectivité, etc.
- *Environnement de l'utilisateur* : localisation, personnes proches, situation sociale, etc.
- *Environnement physique* : bruit, lumière, etc.

Une autre définition similaire du contexte est celle de Brown *et al.* (1997) qui s'articule autour de la localisation d'un utilisateur, de l'identité des personnes autour de cet utilisateur, du moment de la journée, de la saison, de la température, etc. Pascoe *et al.* (1998) définissent le contexte également au niveau de la localisation de l'utilisateur, de son environnement, de son identité et du temps.

Certains travaux définissent le contexte comme étant soit l'environnement d'un utilisateur, soit l'environnement d'une application. Par exemple, Brown (1996) définit le contexte comme un ensemble d'éléments de l'environnement que le

système de l'utilisateur connaît. Franklin et Flachsbart (1998) expliquent le contexte comme étant la situation de l'utilisateur. Abowd et Mynatt (2000) identifient les 5 *W* (*Who*, *What*, *Where*, *When*, *Why*) comme étant l'information minime et requise pour comprendre le contexte.

Dey *et al.* (2001) donnent une définition générale du contexte qui s'applique à des entités qui sont pertinentes pour l'application et l'utilisateur.

> «Le contexte est toute information qui peut être utilisée pour caractériser la situation d'une entité. Une entité est une personne, un lieu ou d'un objet qui est considéré comme pertinent pour l'interaction entre un utilisateur et une application, y compris l'utilisateur et les applications elles-mêmes.» (Dey *et al.*, 2001)

2.8 Type de contexte

Schilit et Theimer (1994) ont groupé les types de contextes sous forme de trois sous-classes où chacune répond à l'une des questions : « Où se trouve l'utilisateur ? », « Avec qui est l'utilisateur? », « Quelles sont les ressources près de l'utilisateur? ».

Ryan *et al.* (Ryan *et al.*, 1998) ont catégorisé le contexte sous l'identité de l'utilisateur, les ressources de l'environnement proche, la localisation de l'utilisateur et la période temporelle d'exécution de l'interaction.

Dey *et al.* (2001) résument les catégories de contextes, soit : la localisation, l'identité, l'activité et le temps. Ces catégories permettent d'éclaircir les questions « Qui ?», « Quand ?», « Quoi? » et « Où? ». Le système utilisera les réponses à ces questions pour répondre à la question « Pourquoi? » une situation se produit.

Schmidt *et al.* (1999) ont proposé 6 catégories de contextes dont 3 reposent sur des facteurs humains et 3 reposent sur l'environnement physique : l'utilisateur, l'environnement social, la tâche, la condition, l'infrastructure et la localisation.

Le temps, l'identité, l'activité et la localisation sont des catégories de contextes primaires qui permettent d'expliquer une situation. Dans la littérature, nous remarquons ces types de contexte primaires ont évolué et des sous-contextes peuvent en découler dans le cadre d'applications sensibles au contexte. Nous donnons quelques exemples de type de contexte utilisé dans ces types d'application (voir figure 2.1).

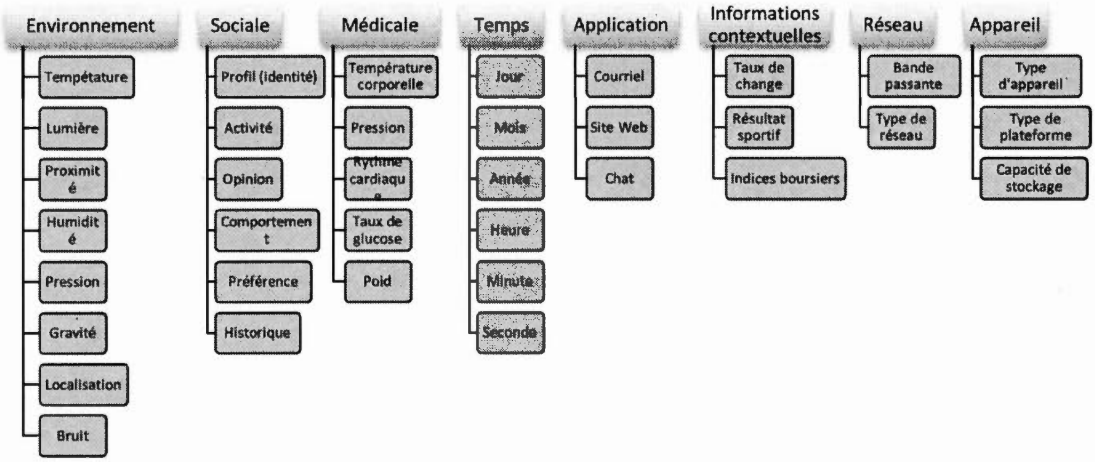


Figure 2.1 Exemples de type de contexte

2.9 Sensibilité au contexte

La sensibilité au contexte consiste à réagir proprement en considérant l'information contextuelle. Schilit *et al.* (1994) a introduit la première fois le terme « context-aware ». Ils considèrent que les applications sensibles au contexte s'adaptent en considérant l'utilisateur et son environnement (Schilit *et al.*, 1994).

Ryan *et al.* (1998) définissent ce terme comme la capacité d'un logiciel de détecter, de capter, d'interpréter et de répondre aux aspects de l'environnement de l'utilisateur et de l'appareil.

Dey et Abbowd (1999) considèrent qu'un système est sensible au contexte s'il utilise le contexte pour fournir des informations pertinentes et/ou des services à l'utilisateur dans lequel la pertinence dépend de la tâche de l'utilisateur (1999).

2.10 Acquisition

Les techniques d'acquisition de contexte sont de différents types:

1) Techniques basées sur la responsabilité : reposent sur l'envoi de données. Soit le logiciel envoie une requête au capteur (*pull*), soit le capteur envoie les données au logiciel (*push*) pour acquérir les données périodiquement ou instantanées (Perera *et al.*, 2014).
2) Techniques basées sur la fréquence : acquièrent les contextes lorsqu'un évènement se produit (instantané) ou à une certaine période du temps (intervalle) (Perera *et al.*, 2014).
3) Techniques basées sur la source : permettent d'acquérir les données depuis des capteurs, de middleware ou des serveurs de contextes (Perera *et al.*, 2014).
4) Techniques basées sur le type de capteurs : distinguent des *capteurs physiques* qui mesurent des données physiques comme la lumière, la température, le son et la localisation, des *capteurs virtuels* qui fournissent des mesures indirectes qui sont par elles-mêmes non mesurables physiquement en combinant les données captées d'un groupe de capteurs physiques hétérogènes (ex. capter les entrées d'un calendrier) ou des *capteurs logiques* qui permettent d'agréger des informations de différentes sources, c'est-à-dire en combinant des capteurs physiques et virtuels dans le but de construire des bases de données de contextes (Perera *et al.*, 2014).

2.11 Prétraitement de contextes

Les contextes acquis sont normalement incomplets, car ils sont sujets à des bruits et des erreurs qui peuvent être causés par l'environnement externe ou des mesures imprécises.

Le traitement des contextes doit gérer les informations contextuelles pour améliorer la qualité du contexte. Les étapes de traitement de contextes sont liées au filtrage et à l'agrégation.

1) Filtrage de contexte : consiste à réduire le nombre de données contextuelles transmises pour améliorer la mise à l'échelle d'un système. Les techniques de filtrage de contextes sont basées sur le temps ou le changement de valeurs. Avec les techniques basées sur le temps, les données à envoyer sont omises jusqu'à ce qu'une certaine condition reliée au temps devienne vraie. Quant aux techniques basées sur le changement, les données à envoyer sont omises tant et aussi longtemps que les données contextuelles sont égales ou similaires aux données transmises précédemment. Des techniques de filtrage peuvent également se baser sur une combinaison des deux à savoir le temps et le changement (Bellavista *et al.*, 2012; Zhang *et al.*, 2013).
2) Agrégation de contexte : consiste à gérer les différents contextes en utilisant des opérations. Les modèles sémantiques (logiques ou ontologies) et de raisonnement probabiliste sont les principales techniques utilisées pour agréger les données contextuelles dynamiques puisqu'elles sont plus faciles à intégrer à un engin de référence (Bellavista *et al.*, 2012; Zhang *et al.*, 2013).

Chen et Kotz (2002b) proposent la plateforme Solar qui offre des techniques de filtrage basées sur le temps et le changement dans un environnement ubiquitaire et qui permet d'agréger le contexte désiré depuis des sources hétérogènes d'applications sensibles au contexte en utilisant des opérateurs. Ils exploitent l'utilisation de graphe

d'opérateurs orientés pour récupérer, agréger et disséminer les informations contextuelles.

Un opérateur est un objet qui traite un ou plusieurs changements d'informations contextuelles, appelé des évènements, et en publie d'autres (Chen et Kotz, 2002). Ils proposent 4 opérateurs (voir figure 2.2):

a) *Filter* : L'opérateur *filter* donne en sortie un sous-ensemble des évènements en entrée. Par exemple, un capteur publie la température toutes les 10 secondes alors qu'une application a uniquement besoin de capter la donnée lorsque la température excède 32 degrés.
b) *Transformer* : L'opérateur *transformer* donne des évènements en entrée de type E1 et en sorties de type E2. E2 peut être le même qu'E1 si l'opérateur change uniquement les valeurs de certains attributs. Par exemple, un capteur de localisation signale les coordonnées, mais l'application a besoin d'une réponse symbolique comme « rez-de-chaussée ».
c) *Merger* : L'opérateur *merger* donne en sortie tous les évènements qu'il reçoit. Par exemple, une application qui affiche la localisation courante de tous les employés fusionne les données captées provenant de tous les capteurs.
d) *Aggregator* : L'opérateur *aggregator* donne en sortie un évènement arbitraire basé sur un ou plusieurs évènements d'entrée. Par exemple, un thermomètre à minimum et maximum donne en sortie un évènement lorsqu'il détecte un nouveau maximum ou minimum sur l'évènement d'entrée des données de température courante.

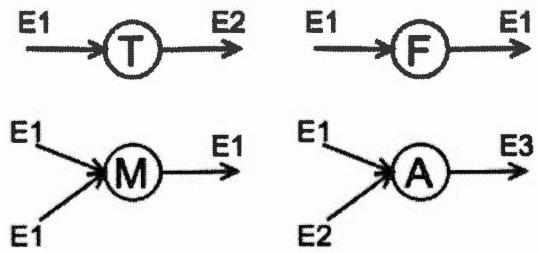


Figure 2.2 Les 4 types d'opérateurs : Transformer (T), Filter (F), Merger (M) et Aggregator (A) (Chen et Kotz, 2002)

Ces opérateurs sont connectés pour former un graphe d'opérateurs orientés qui se compose de 3 types de nœuds : sources, opérateurs et applications. Les sources représentent des *enveloppes* (*wrappers*) de capteurs de contexte. Les opérateurs représentent des fonctions qui s'appliquent sur les évènements d'entrée et publient des évènements lorsqu'ils reçoivent un évènement d'entrée. Les applications sont des « puits » du graphe. Dans un graphe, les *éditeurs* (*publishers*) sont les sources et les opérateurs, et les *membres* (*subscribers*) sont les opérateurs et les applications.

La figure 2.3 illustre un exemple de graphe d'opérateurs. Les cercles représentent des membres (*publishers)* et les carrés représentent les applications qui consomment les évènements. Supposons que nous ayons des capteurs de géo-localisation dans chaque salle qui établissent un suivi de l'emplacement des personnes qui portent un badge. À chaque fois que le capteur détecte un signal provenant du badge, il envoie un évènement contenant l'identificateur (ID) du badge et l'heurodate (*timestamp*). Dans la figure 2.3, ces sources sont indiquées par « Loc Sensor » avec le numéro de la salle et chacun a un opérateur *transformer* qui établit une correspondance entre l'ID du badge de la personne, et l'ID du capteur membre (*publisher*) à l'emplacement couvert par le capteur. L'opérateur *transformer* envoie les évènements contenant le nom traduit, l'emplacement et l'horodate (*timestamp*). Un exemple d'opérateur est « Building Locator » qui souscrit l'emplacement courant de chaque badge basé sur les

évènements de sortie des opérateurs *transformer* et *merge*. Il génère un évènement lorsqu'il s'aperçoit qu'un badge a changé d'emplacement. Cet évènement de sortie peut être utilisé par l'application « Active Map » pour afficher l'emplacement courant des badges en temps réel.

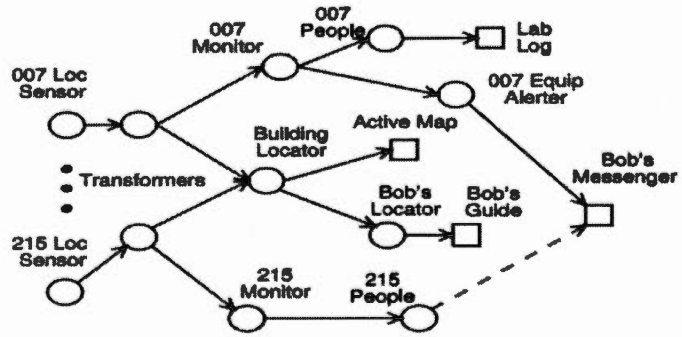


Figure 2.3 Exemple de graphe d'opérateurs (Chen et Kotz, 2002)

Solar facilite le routage des informations contextuelles en se basant sur le patron pipe-filtre et est indépendante d'un langage (Chen et Kotz, 2002; Chen et Kotz, 2002a, 2002b). L'avantage de Solar est qu'il permet le partage de données contextuelles prétraitées entre applications sensibles au contexte.

2.12 Modélisation du contexte

La modélisation ou la représentation du contexte consiste à définir les informations contextuelles en termes d'attributs, de caractéristiques et de liens. Elle consiste également à organiser le contexte selon un modèle qui nécessite une validation avant de pouvoir le stocker.

De nombreux modèles ont été utilisés dans la littérature pour modéliser le contexte. Nous présentons quelques uns de ces modèles.

Le modèle attribut-valeur présente le contexte sous forme de paires (attribut, valeur) où l'attribut indique le nom d'une information contextuelle et la valeur indique la

valeur de cette information (Schilit *et al.*, 1994). Il s'agit de la plus simple méthode de modélisation de contexte, et elle est facile à gérer pour une petite quantité de données. Par contre, cette méthode manque de capacité d'établir des liens entre les données, leurs unités et méthodes de traitement et ne peut être utilisée pour une structure de données complexes.

Le modèle basé sur un langage de marquage utilise généralement une structure de données hiérarchiques dont la profondeur dépend du contexte ainsi que des balises de marquage telles que XML (Knappmeyer *et al.*, 2010). Nous retrouvons souvent les langages à base de profils dans ces modèles (Held *et al.*, 2002). L'avantage de cette méthode est lié à l'acquisition de données. Néanmoins, cette méthode est normalement dépendante du type d'application puisqu'il n'existe pas de standard pour la structure et elle devient complexe à utiliser si un grand nombre de données est impliqué.

Le modèle graphique modélise le contexte avec des liens. UML (Bauer, 2003) et ORM (Henricksen *et al.*, 2003) sont des exemples de ce type de modélisation. Ces modèles sont faciles à apprendre et à utiliser. Un autre exemple est celui des bases de données telles que les bases de données SQL, les bases de données NoSQL et les bases de données XML (Perera et al., 2014). Ces dernières offrent des opérations de lecture et de lecture de façon simple et rapide, et elles peuvent servir de stockage de contexte. Par contre, les requêtes peuvent être complexes, des configurations peuvent être requises et les changements de la structure de données peuvent êtres difficiles.

Le modèle orienté-objet utilise des objets pour représenter différents types de contexte (Cheverst *et al.*, 1999). Il encapsule les détails du traitement et de représentation du contexte. L'accès à ces contextes se fait via des interfaces bien définies. Avec ce modèle, les objets servent à représenter les données captées alors que les méthodes permettent de traiter le contexte.

Le modèle basé sur la logique est une méthode de modélisation d'information contextuelle à un niveau abstrait (McCarthy et Buvac, 1997). Il gère les faits, les expressions et les règles pour définir le modèle contextuel. Ce type de modèles est plus expressif comparé aux modèles précédents, mais il est normalement couplé à l'application (Perera *et al.*, 2014).

Le modèle basé sur les ontologies organise le contexte sous forme d'ontologie en utilisant une sémantique telle que RDF, RDFS ou OWL (Wang *et al.*, 2004). Ce type de modèles offre également des capacités de raisonnement. Divers outils de développement et des moteurs de raisonnement sont disponibles. Toutefois, la représentation peut être complexe et la récupération d'informations peut être soumis à forte intensité de calcul si le nombre de données est élévé.

2.13 Raisonnement sur le contexte

Le raisonnement sur le contexte consiste à déduire de nouvelles connaissances en se basant sur des informations contextuelles courantes. Il permet également de générer des informations contextuelles abstraites à partir d'informations contextuelles de bas niveau.

Il existe un grand nombre de méthodes permettant de raisonner sur le contexte. Nous en présentons quelques-unes.

L'apprentissage automatique fait référence à l'analyse, la conception et l'implémentation de méthodes permettant à un système d'évoluer par un processus d'apprentissage tel que les réseaux bayésiens et les techniques de *clustering* (Perera *et al.*, 2014). Cette méthode de raisonnement ne donne pas toujours des données prévisibles, requiert des ressources intensives (e.g. traitement et stockage) et devient complexe si le nombre de données est élevé.

Le raisonnement à base de règles est souvent utilisé par les modélisations par la logique ou les ontologies (Gu, 2004). Une règle est vue comme un énoncé déclaratif qui spécifie comment les valeurs d'attributs peuvent être combinées pour dériver la valeur d'un autre attribut. Les règles peuvent exprimer des sémantiques implicites comprises dans un langage de modélisation ou défini par l'utilisateur.

Le raisonnement à base de logique s'avère suffisant pour des systèmes sensibles au contexte (Ranganathan et Campbell, 2003). Ces systèmes peuvent raisonner par rapport au contexte en utilisant des règles écrites en logique de premier ordre, temporelles ou descriptives.

Les ontologies sont également utilisées comme méthode de raisonnement sur le contexte (Turhan *et al.*, 2006). OWL, l'un des standards d'ontologie, est basé sur la logique descriptive et il permet donc le raisonnement. Une ontologie OWL est équivalente à une base de connaissance de logique descriptive. L'ontologie est également complémentée par des langages de requêtes (SWRL, RDQL, RQL, TRIPLE, etc.) ainsi que des moteurs d'inférences (FACT, RACER, PELLET, etc.). Ce type de raisonnements est limité car il ne peut trouver des valeurs ou des informations incertaines. En utilisant les langages de requêtes, des règles peuvent être écrites pour limiter cette contrainte, mais elles ne sont pas efficance dans un environnement dynamique et incertain (Perera *et al.*, 2014).

Les méthodes de raisonnement basé sur la logique probabiliste offrent l'avantage de gérer les situations incertaines en fournissant des données pertinentes (Lyu *et al.*, 2010). Des exemples de cette méthode sont les modèles de Markov et la classification naïve bayésienne. Par contre, cette méthode permet uniquement le raisonnement sur des valeurs numériques.

De façon générale, de nombreux enjeux se posent au niveau des stratégies de raisonnement par rapport au contexte tels que l'incertitude des informations, les

informations manquantes, l'adaptabilité du système au changement, l'efficacité du traitement ainsi que la nature des données car certaines méthodes ne fonctionnent que sur les données numériques. Ces approches ne gèrent pas tous les enjeux; certains sont complètement omis.

2.14 Adaptation et boucle de contrôle

Le terme adaptation des systèmes auto-adaptatifs a été défini de nombreuses fois dans la littérature. Nous énumérons quelques une des définitions que nous jugeons pertinentes :

> « L'auto-adaptation est la capacité du système d'ajuster son comportement en réponse à son environnement. Le préfixe *auto* indique que les systèmes décident autonomement comment s'adapter ou s'organiser pour qu'ils puissent accommoder les changements dans leurs contextes et environnements. » (Brun *et al.*, 2009)

> « Pour les systèmes multi-modèles, l'adaptation est une procédure ou une méthode permettant de changer de modèles. L'adaptabilité se définit comme la capacité d'un système d'atteindre ses buts dans un environnement changeant, en exécutant et changeant sélectivement entre modèles. » (Ravindranathan et Leitch, 1998)

> « Un système auto-adaptatif consiste en un système à boucle de contrôle fermée, c.-à-d. se modifie en temps d'exécution en utilisant la rétroaction suite au changement continu du système, ses exigences et tendances existantes en développement et déploiement de systèmes complexes. La conception de systèmes auto-adaptatifs dépend des exigences de l'utilisateur, des propriétés du système et des caractéristiques environnementales. » (Naqvi, 2012; Weyns *et al.*, 2012)

> « Un système auto-adaptatif évalue son propre comportement et change ses performances lorsque l'évaluation indique qu'il n'accomplit pas ce que le logiciel à l'intention de faire ou lorsque de meilleures fonctionnalités ou performances sont possibles. » (Salehie et Tahvildari, 2012)

De nombreux modèles de systèmes auto-adaptatifs ont été proposés. Ces modèles partagent tous un point en commun qui est de conceptualiser un système dont le contrôle de son comportement se fait dynamiquement lors de l'exécution et qu'il

puisse raisonner par lui-même sur son état et son environnement. Pour ce faire, les boucles de contrôle fournissent le mécanisme d'auto-adaptation de ces systèmes.

2.14.1 Boucle de contrôle *feedback*

La boucle de contrôle *feedback* a pour objectif de régulariser les caractéristiques d'un système afin qu'il puisse maintenir un état ciblé (Villegas *et al.*, 2011). Elle consiste à comprendre comment contrôler une entrée et comment les perturbations peuvent affecter les sorties ou le système ciblé (Villegas *et al.*, 2011). Les éléments essentiels d'une boucle de contrôle sont les suivants (voir figure 2.4):

- Point d'entrée : est la valeur ciblée comme sortie mesurée.
- Erreur de contrôle : indique la différence entre le point d'entrée et la sortie mesurée.
- Entrée de contrôle : est un paramètre qui affecte le comportement d'un système ciblé.
- Contrôleur : détermine l'ajustement requis de l'entrée de contrôle pour atteindre le point d'entrée.
- Entrée de perturbations : représente un changement qui affecte la façon dont l'entrée de contrôle impacte la sortie mesurée.
- Entrée de bruits : représente un changement qui altère la sortie mesurée générée par le système ciblé.
- Sortie mesurée : est une caractéristique mesurable d'un système ciblé.
- Système ciblé : représente le système à contrôler.
- Capteur : surveille les sorties d'un système ciblé.
- Actionneur : ajuste les entrées de contrôle au changement du comportement du système ciblé.

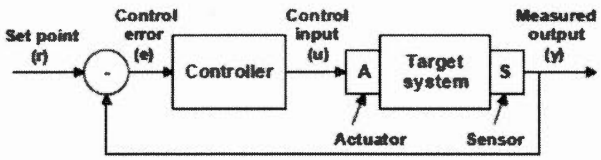


Figure 2.4 Boucle de contrôle *feedback* (Patikirikorala *et al.*, 2012)

Le système ciblé fournit un ensemble de sorties dont les valeurs sont surveillées par des capteurs. S'il existe une différence entre une donnée de sortie et un point d'entrée, causée par des perturbations, l'erreur de contrôle est envoyé au contrôleur. Le contrôleur a pour objectif de maintenir les sorties mesurées suffisamment près du point d'entrée en ajustant les entrées de contrôle. Le point d'entrée spécifie les valeurs de sortie qui doivent être maintenues en temps d'exécution. Ces entrées de contrôle sont ajustées par des actionneurs dont le but est de modifier le comportement du système.

La boucle de contrôle *feedback* surveille l'état courant du système ciblé, détermine l'écart entre l'état courant et l'état ciblé, calcule une valeur de contrôle et donne la différence (une entrée de contrôle) au système ciblé (Patikirikorala *et al.*, 2012). Cette boucle de contrôle se répète tant et aussi longtemps qu'il y a une perturbation affectant la valeur de sortie du système ciblé.

Ce type de boucle de contrôle n'élimine pas complètement les perturbations étant donné que le contrôleur n'intervient uniquement qu'une fois l'erreur a été causée par ces perturbations.

2.14.2 Contrôle adaptatif

Un contrôle adaptatif se compose d'une boucle de contrôle *feedback* et d'une boucle de contrôle adaptatif.

Un contrôle adaptatif est un ensemble de techniques d'ajustement automatique en temps réel des contrôleurs dont le but est d'accomplir ou maintenir un niveau désiré de la performance d'un système lorsque les paramètres sont inconnus ou s'alternent avec le temps.

Un contrôleur adaptatif est un contrôleur ayant des paramètres ajustables et un mécanisme d'ajustement de ses paramètres. Il peut modifier le comportement d'un système en fonction des perturbations et maintient la performance du système en présence de variations inconnues des paramètres du système.

Model Identification Adaptive Control (MIAC) et *Model Reference Adaptive Control* (MRAC) sont deux modèles importants de contrôle adaptatif (Brun *et al.*, 2009). Les deux modèles utilisent un modèle de référence pour déterminer si le contrôleur adaptatif nécessite un ajustement. Le modèle MIAC (voir figure 2.5) utilise un modèle de référence dynamique en observant le système sans tenir compte des points d'entrée alors que le modèle MRAC (voir figure 2.5) utilise un modèle de référence prédéfini qui prend en considération des points d'entrée (Brun *et al.*, 2009).

MIAC détermine le modèle du système courant en utilisant des entrées et sorties mesurées ainsi que des méthodes d'identification qui permettent aux contrôleurs en temps réel de créer ou mettre à jour le modèle du système qu'il contrôle (Brun *et al.*, 2009). Ce modèle est créé soit à partir de données observées, soit à partir d'un modèle de base existant.

Le système d'identification de MIAC prend l'entrée de contrôle u et la sortie y_p pour déduire le modèle courant du système. Il fournit ensuite les caractéristiques du système qu'il a identifiées au mécanisme d'ajustement pour ajuster les paramètres du contrôleur. MIAC prend également en considération les perturbations d qui peuvent affecter le comportement du système.

Avec le modèle MIAC, la boucle interne est une boucle de contrôle *feedback* alors que la boucle externe identifie le modèle courant du système pendant qu'il s'exécute pour modifier les paramètres de contrôle (Brun *et al.*, 2009).

MIAC estime continuellement les paramètres du modèle et le modèle de référence est ajusté en temps réel (Brun *et al.*, 2009). Ce type de modèle est intéressant à utiliser s'il n'y a pas assez de connaissances sur les modèles de référence qui peuvent être utilisées par le système, mais qu'il en existe assez pour identifier ces caractéristiques.

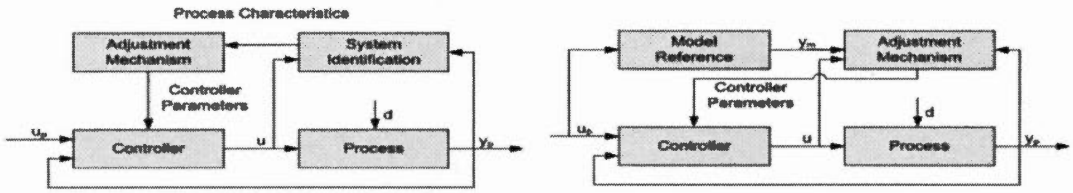


Figure 2.5 *Model Identification Adaptive Control* (MIAC) et *Model Reference Adaptive Control* (MRAC) (Brun *et al.*, 2009)

2.15 Modèles d'auto-adaptation inspirés des boucles de contrôle

Les systèmes auto-adaptatifs s'inspirent de la boucle de contrôle *feedback* pour effectuer leurs adaptations. Le modèle général d'une boucle d'adaptation pour les systèmes auto-adaptatifs implique 4 activités : *collect*, *analyze*, *decide* et *act* (Da *et al.*, 2012). Il s'agit d'un modèle inspiré de l'approche *sense-plan-act* qui permet de contrôler les robots mobiles autonomes, plus souvent utilisés dans le domaine de l'intelligence artificielle.

a) *Collection*: collecte les données pertinentes provenant de capteurs de l'environnement et d'autres sources qui reflètent l'état courant du système. Cette étape nécessite la répondre aux questions suivantes :
   - Quel est le taux d'échantillonnage requis?
   - Quelle est la fiabilité des données de capteurs ?

- Existe-t-il un format d'évènement commun à travers les capteurs?

b) *Analysis*: analyse les données en utilisant une approche permettant de structurer et de raisonner sur les données brutes. Ces données sont nettoyées, filtrées, élaguées et stockées pour déterminer quand un changement survient. Certaines questions qui nécessitent une réponse à cette étape sont :

- Quel est l'état courant du système inféré?
- Quelles données ont besoin d'être archivées pour validation et vérification?
- Dans quel mesure le modèle est-il fidèle au monde réel et existe-t-il un modèle adéquat qui puisse être obtenu à partir des données de capteurs disponibles?

c) *Decision*: détermine ce qui doit être modifié et comment le faire. Les questions qui reviennent dans cette étape sont :

- Comment est l'état futur du système inféré?
- Comment est atteinte une décision?

d) *Action*: implémente une décision via des actionneurs. Les questions qui reviennent le plus souvent dans cette étape sont :

- Est-ce que l'adaptation peut se faire de façon sécuritaire et quand doit-elle se faire?
- Comment les ajustements de différentes boucles de contrôle *feedback* s'interfèrent avec les autres?
- Est-ce que les actionneurs disponibles sont suffisants pour amener le système vers des états ciblés?

2.16 Boucle de contrôle MAPE-K

IBM a proposé la boucle de contrôle MAPE-K comme modèle de référence pour les systèmes auto-adaptatifs (Jacob *et al.*, 2004). Ce modèle se base sur l'approche *sense-plan-act.* Le terme *MAPE-K* réfère aux activités d'une entité autonome (*autonomic*

element), soit : *monitor*, *analyze*, *plan*, *execute* et *knowledge*. La boucle de contrôle MAPE-K décrit les éléments nécessaires pour qu'un système puisse s'autogérer, qui sont : 1) collecter les informations contextuelles pour identifier un besoin de changement, 2) planifier un ensemble d'actions pour effectuer le changement et 3) coordonner l'exécution des actions planifiées pour adresser le changement requis. La figure 2.6 illustre le modèle de référence MAPE-K. *Autonomic element* réfère à un logiciel ou matériel (*managed ressource*) qui a des propriétés d'auto-gestion et d'un contrôleur (*autonomic manager*) qui utilise les capteurs pour observer les changements du *managed ressource* et impose des actions sur le *managed ressource* à partir d'actionneurs. MAPE-K consiste en plusieurs éléments :

a) *Monitor* : consiste à capter les propriétés importantes du système à partir de capteurs. Les données collectées dépendent des objectifs. Cette phase fournit les informations aux autres phases et représente le contexte courant. Elle transforme l'information collectée sous un format qui peut être manipulé par les autres phases. La transformation peut impliquer des activités de filtrage, d'analyse et d'agrégation en prenant en considération le temps. Le modèle peut prendre différentes formes telles qu'une liste de faits, un graphe d'objets, un automate fini et une architecture de logiciel.
b) *Analyze* : consiste à analyser et comprendre l'état courant du contexte et de spécifier l'état ciblé si des problèmes surviennent. Diverses techniques telles que la régression, le modèle de Markov et *Event-Condition-Action* (ECA) peuvent être utilisés pour détecter les mauvais comportements et les lacunes, établir des corrélations, anticiper des situations, diagnostiquer des problèmes et définir des situations désirables.
c) *Plan* : consiste à prendre des décisions concernant les changements et les adaptations à assembler et implémenter sur le logiciel afin de passer de l'état courant à un état désiré. Elle dépend d'un ensemble d'actions qui peuvent être exécutées sur le logiciel. Un plan peut être statique ou dynamique. Par exemple,

un plan statique pourrait être un ensemble d'étapes qui doivent être réalisées lorsqu'une condition particulière survient alors qu'un plan dynamique pourrait être de modéliser le comportement des composants du logiciel et de choisir un plan parmi un nombre de plans prédéfinis. Cette phase doit anticiper le futur et prédire les conséquences des actions. Une façon d'établir la planification est d'exprimer l'état du système et de définir des opérateurs d'actions agissant sur les états. Les opérateurs définissent les préalables et les post-conditions sur l'état. La planification détermine une séquence d'opérateurs permettant de passer d'un état courant à un état ciblé et se fait généralement sous forme de graphe avec ou sans heuristiques (Lalanda *et al.*, 2013).

d) *Execute* : consiste à executer les étapes du plan. Elle planifie l'implémentation des plans et examine en temps réel les post-conditions de ses actions afin d'effectuer des ajustements, si nécessaire.

e) *Knowledge* : consiste à échanger des informations aux 4 phases de la boucle de contrôle. Elle fournit : les directives de surveillance à la phase *monitor*, l'état courant à la phase *analyze*, l'état courant et ciblé à la phase *plan* et les actions du *plan* à la phase *execute*.

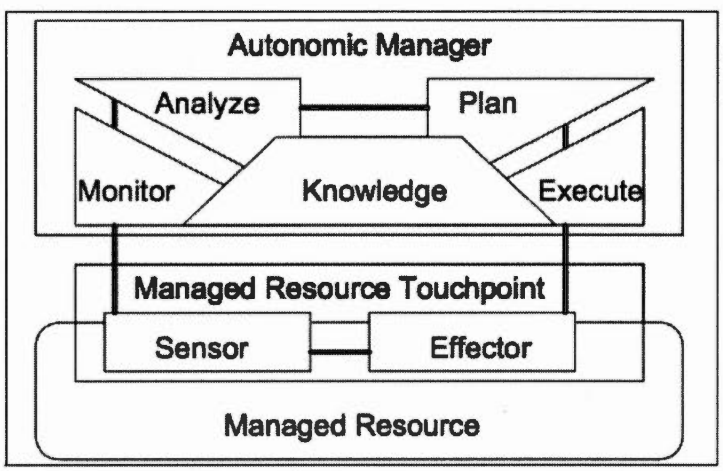


Figure 2.6 Boucle de contrôle MAPE-K (Jacob *et al.*, 2004)

2.17 Approches d'adaptation

2.17.1 Approches d'adaptation basées sur les lignes de produits logiciels

Une ligne de produits logiciels est un ensemble de systèmes partageant un ensemble de fonctionnalités communes, satisfaisant des besoins spécifiques pour un domaine particulier, et développés de manière contrôlée à partir d'un ensemble commun d'éléments réutilisables (Svahnberg *et al.*, 2005).

D'après Svahnberg *et al.* (2005), la variabilité dans les lignes de produits est la capacité d'un système logiciel à être prolongé, modifié, personnalisé ou configuré efficacement pour son utilisation dans un contexte particulier.

Par exemple, le travail de Cetina *et al.* (2009) porte à utiliser des modèles de variabilité, plus précisément les modèles de *features*, en temps réel pour reconfigurer un système. La figure 2.7 illustre l'approche MoRE (*Model-Based Reconfiguration Engine*). Dans cet exemple, le modèle de *features* est celui d'une maison intelligente. Elle se compose de certaines étapes. D'abord, le *context monitor* vérifie les conditions du contexte en temps réel. Chaque condition est associée à l'activation et/ou la désactiveation des *features* qui se définissent sous *resolution*. Ces *resolution* représentent les conditions activant un ensemble de changements. Par la suite, le moteur du modèle MoRE génère un plan d'action qui permet de modifier l'architecture du système. L'exécution du plan modifie donc l'architecture, à savoir l'activation et la désactivation de *features* spécifiées dans le modèle de *résolution*.

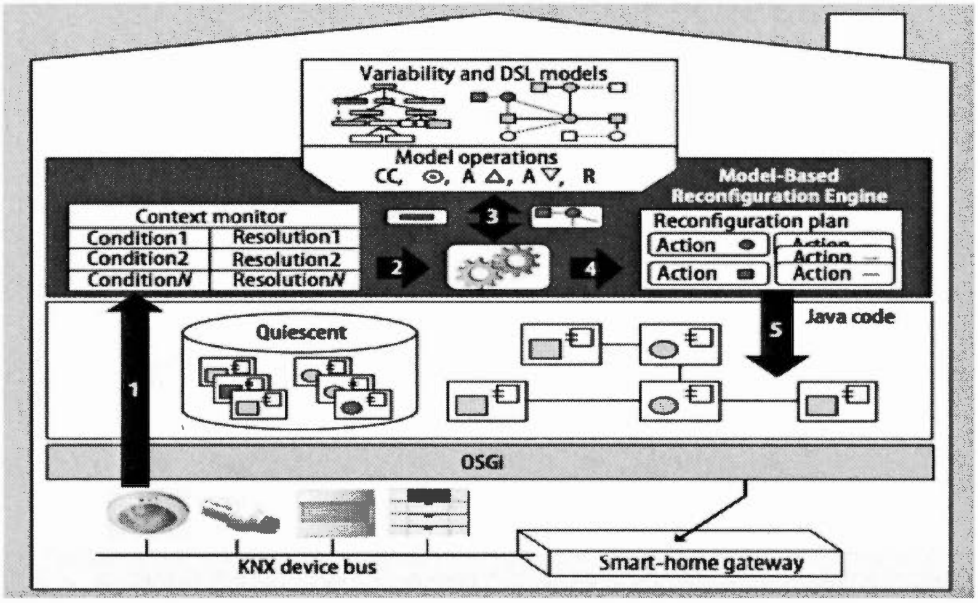


Figure 2.7 Exemple du modèle MoRE (Cetina *et al.*, 2009)

De nombreux modèles d'adaptation de système en temps réel ont été proposés en utilisant des mécanismes de variabilité dans les lignes de produits dynamiques (Capilla et Bosch, 2011). Ces types de modèles font face à des défis, dont le besoin de :

- Représenter la variabilité en temps réel, modifier les points de variations des entités existantes et nouvelles du système durant l'exécution et automatiser la reconfiguration du système.
- Automatiser la validation en temps réel et vérifier les modèles de variabilité pour maintenir un système stable.
- Automatiser le déploiement et relier les produits configurés en temps réel avec un minimum d'interruption.

2.17.2 Approches d'adaptation basées sur des langages de coordination

Arbab (2004) présente Reo, un modèle de coordination basé sur des canaux. Reo coordonne des entités. Une entité peut être des fragments ou des modules de code séquentiel, des objets passifs ou actifs, des processus, des agents ou des composants de logiciel (Arbab, 2004). Arbab réfère à ces entités comme des instances de composants (Arbab, 2004). Un système est composé d'un ensemble d'instances de composants qui sont réparties et qui communiquent via des opérations d'entrée et de sortie à partir de connecteurs qui coordonnent leurs activités (Arbab, 2004).

Les canaux sont utilisés dans Reo pour transférer ou échanger des données à partir des opérations d'entrée et de sortie. Il existe deux types de canaux : sources et puits. L'extrémité source met les données dans le canal alors que l'extrémité puits prend les données. Un canal qui est connu à une instance de composant peut être utilisé par des entités actives par cette instance dans les opérations de Reo. Il peut être connecté à au plus une instance de composant à un moment donné.

Par exemple, la figure 2.8 représente un connecteur composé de 3 canaux *ab*, *cd* et *ef* (Arbab, 2006). Les canaux *ab* et *cd* sont de type *Sync* et *ef* est de type *SyncDrain*. *Sync* a deux nœuds : un nœud source et un nœud puits. Les opérations d'entrée et de sortie de ces nœuds s'effectuent simultanément. *SyncDrain* a deux nœuds sources. Les opérations d'entrée et de sortie s'effectuent uniquement simultanément. Toutes les données écrites sur ce canal sont perdues. Ce connecteur a donc pour objectif de contrôler les opérations d'écriture. Il montre une coordination exogène, c'est-à-dire que le nombre de données qui passent de *a* à *d* est le même que le nombre d'opérations d'écriture qui réussissent sur *f*. Une instance d'un composant connecté à *f* peut contrôler le flux de données entre les nœuds *a* et *d* par le nombre d'opérations d'écriture qui se font sur *f*. L'entité qui contrôle les données qui passent par *f* n'a pas besoin de connaître les entités qui écrivent sur *a* ou qui consomment des données de *b*, ou que ces actions d'écritures contrôlent le flux. Les entités qui communiquent par

a et *d* n'ont pas besoin de connaître le fait qu'elle communiquent ensemble ou que la communication est contrôlée par l'entité *f*.

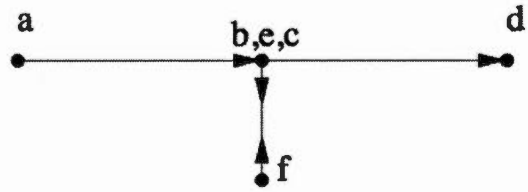


Figure 2.8 Exemple de connecteur du modèle Reo (Arbab, 2006)

Plusieurs formalismes de modèles sémantiques ont été proposés au cours des dernières années pour décrire le comportement des connecteurs Reo, incluant les modèles co-algèbres, les modèles opérationnels et les modèles colorimétriques (Jongmans et Arbab, 2012).

2.17.3 Approches d'adaptation basées sur les algèbres de processus

L'algèbre des processus est un formalisme mathématique pour la description de systèmes concurrents ou distribués (Bradbury *et al.*, 2004). Ce formalisme est utilisé dans les langages de description d'architecture pour modéliser les composants, les connecteurs et les configurations des systèmes dépendants du contexte (Bradbury *et al.*, 2004).

Les langages de description d'architecture tels que Darwin et LEDA permettent une reconfiguration architecturale d'un système en se basant sur π-calculus (Bradbury *et al.*, 2004). Wright ou PiLar se basent respectivement sur CCS (*Calculus of Communicating Systems*) pour modéliser l'interaction de système et CSP (*Communicating Sequential Processes*) pour décrire le comportement de composants. (Bradbury *et al.*, 2004).

Un exemple du modèle Pilar est présenté dans la figure 2.9 (Cuesta *et al.*, 2001). Cet exemple décrit un système 2-tiers (Client-Serveur) qui devient 3-tiers lorsqu'un serveur requiert la création d'un élément proxy pour gérer les interactions avec le client. Le serveur émet donc un signal spécial, appelé *H*. Les contraintes sont définies en CCS, et permettent de décrire les interactions du système. Les composants, leurs ports et leurs contraintes sont représentés par cette figure. Lorsque *inter* détecte que le serveur (par son port *B*) a émis un évènement, la réification précédente de *L* est détruite et un nouveau lien est créé avec le type *tier*s, c'est-à-dire un connecteur qui se compose d'un proxy *P* et deux liens de type *comm*. La première contrainte est de recevoir un message *x* qui provient du port *A* et de l'envoyer au port *B* ou vice versa, et ce de façon indéfinie. La seconde contrainte réifie le composant de type *comm* en le connectant avec les ports de sorties *A* et *B*. Il attend de recevoir le message *H* du serveur par le port *B*. Dès que le message arrive, la réification précédente est détruite et réifier avec *tier*s au lieu de *comm*.

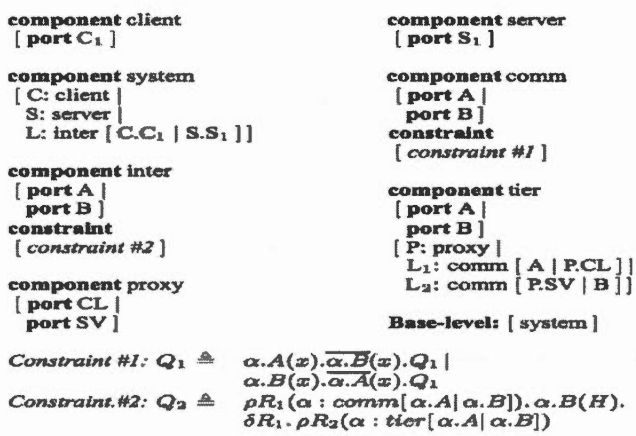


Figure 2.9 Exemple du modèle Pilar (Cuesta *et al.*, 2001)

L'adaptation des systèmes utilisant un langage de description d'architecture se limite aux opérations de reconfiguration qui sont disponibles comme l'ajout ou la suppression de composants et de connecteurs.

2.18 Exemples de modèles d'adaptation

2.18.1 Modèle Clariisa

Gardini *et al.* (2013) proposent un cadriciel *(framework)* pour les systèmes sensibles au contexte basé sur la géolocalisation pour un système médical : Clariisa. Il se compose de trois parties distinctes (voir figure 2.10) :

1) *Acquisition de données* : utilise les capteurs disponibles dans un appareil mobile pour capter les informations de l'utilisateur et de son environnement telles que la localisation, la date, la fréquence cardiaque et la pression.
2) *Traitement de données* : associe et organise les informations afin de fournir une structure compréhensible à stocker.
3) *Publication*: est responsable de recevoir les informations contextuelles provenant de différentes sources et de prendre des décisions.

Lorsqu'un utilisateur mobile lance son application médicale, le système capte et traite les données pour les enregistrer. Les données sont analysées en considérant les données de santé actuelle du patient et son profil historique. Le système effectue des inférences et des corrélations menant à poser des questions à l'utilisateur. Ce dernier répond aux questions et le système prend une décision.

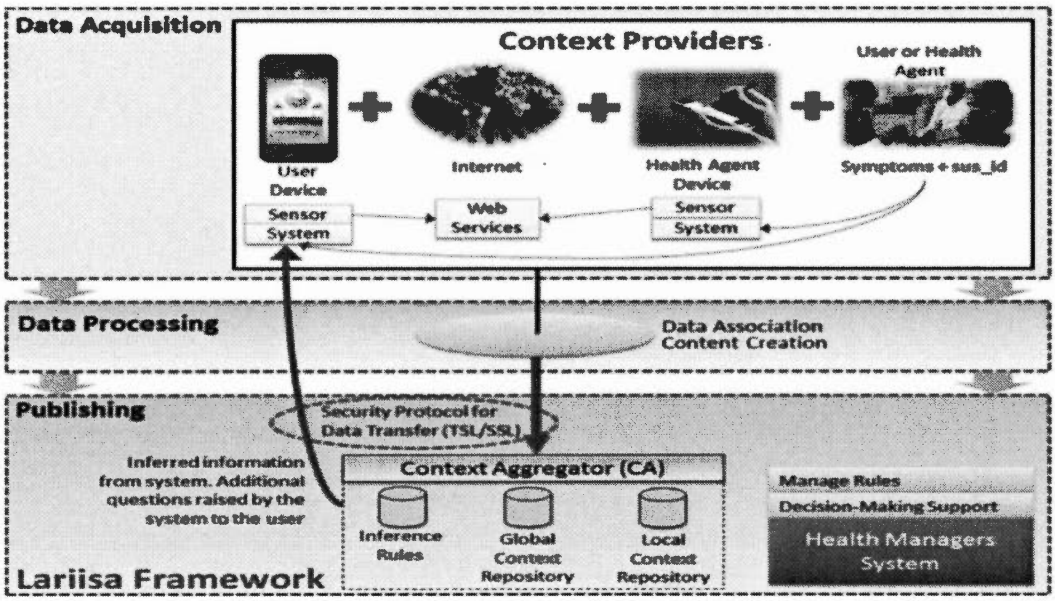


Figure 2.10 Modèle Clariisa (Gardini *et al.*, 2013)

2.18.2 Modèle CareDB

Levandoski et Mokbel (Levandoski, 2010; Mokbel et Levandoski, 2009) ont proposé une base de données adaptative au contexte de localisation et de préférences de l'utilisateur, appelée CareDB (voir figure 2.11).

L'architecture de CareDB prend en entrée la requête et les préférences de l'utilisateur. Les auteurs définissent trois types de contextes (Levandoski, 2010; Mokbel et Levandoski, 2009):

- *L'utilisateur* : indique les informations statiques ou dynamiques sur les utilisateurs telles que le salaire, la profession, l'âge et la localisation.
- *La base de données* : réfère aux données de la source telles que les restaurants, les hôtels et les taxis. Par exemple, les données de restaurants incluent les informations sur les prix, les heures d'ouverture, le temps d'attente, etc.

- *L'environnement* : représente l'information sur l'environnement telle que la température et le trafic.

Le module *Query Building* personnalise les requêtes pour chaque utilisateur en y ajoutant les contextes requis (Levandoski, 2010; Mokbel et Levandoski, 2009). Ces requêtes personnalisées sont envoyées au module *Query Processor* afin qu'il puisse les traiter (Levandoski, 2010; Mokbel et Levandoski, 2009). Les rôles principaux de ce module sont principalement d'intégrer des requêtes sensibles au contexte et d'évaluer celles qui impliquent les données incertaines.

Un inconvénient de cette approche est que tout son fonctionnement est axé sur la base de données, soit un besoin très spécifique et qui ne peut être généralisé. Toutefois, cela implique de fournir ses propres opérateurs ainsi que des stratégies spécifiques pour traiter les requêtes.

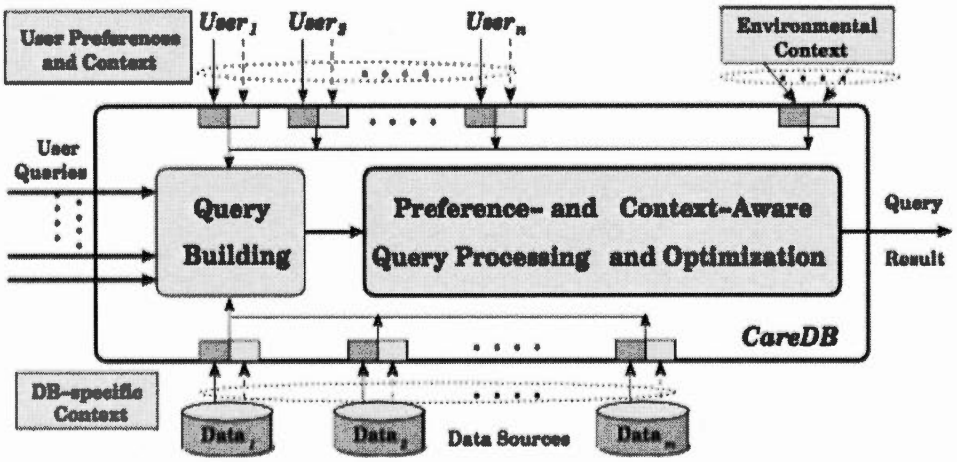


Figure 2.11 Modèle CareDB (Levandoski, 2010; Mokbel et Levandoski, 2009)

2.18.3 Modèle EUREMA (*Executable Runtime MegAmodels*)

Vogel et Giese (2013, 2014) ont proposé une approche basée sur l'ingénierie dirigée par les modèles, soit : *ExecUtable RuntimE MegAmodels* (EUREMA). EUREMA

(voir figure 2.12) permet la spécification et l'exécution d'un moteur d'adaptation pour les systèmes auto-adaptatifs avec de multiples boucles de contrôle *feedback* (Vogel et Giese, 2013, 2014).

EUREMA sépare le système auto-adaptatif en deux parties représentées sous une approche externe: le moteur d'adaptation et le système adaptable (Vogel et Giese, 2013, 2014). La séparation entre ces deux parties représente la boucle de contrôle *feedback*, c'est-à-dire entre le système adaptable et le moteur d'adaptation. Le moteur d'adaptation et le système adaptable sont connectés par des capteurs et des actionneurs.

Vogel et Giese (Vogel et Giese, 2013, 2014) ont proposé un langage pour modéliser un moteur d'adaptation basé sur le concept de *megamodel*. Ces *megamodels* réfèrent à un modèle qui contient d'autres modèles et des liens entre les modèles (Vogel et Giese, 2013, 2014). Ces liens constituent des transformations de modèles.

Les activités de la boucle de contrôle MAPE-K effectuent des opérations dans le but d'auto-adapter un système alors que la base de connaissances fournit les informations nécessaires à chacune de ces activités afin qu'elles puissent compléter leurs tâches. Selon Weyns *et al.* (2012), même s'il existe un partage d'informations entre les différentes activités de la boucle de contrôle MAPE-K, le rôle de la base de connaissances n'est pas précis. Dans le cas de Vogel et Giese, ils ont utilisé des *magamodels* pour justifier la partie de la base de connaissances (Vogel et Giese, 2013, 2014).

Le moteur d'adaptation s'inspire de la boucle de contrôle MAPE-K (IBM, 2006) que nous avons détaillé dans la section 2.10, et il y intègre des mégamodels, également appelés modèles d'introspection comme suit (Vogel et Giese, 2013, 2014):

1) *Modèles d'introspection* : reflète le système adaptable et son environnement.

2) *Activité de surveillance de MAPE-K et modèle de surveillance* : La phase surveillance observe l'adaptation du système et son environnement, et met à jour le modèle d'introspection. Les modèles de surveillance servent à spécifier la correspondance entre les observations du système et le modèle d'introspection dans un niveau abstrait.
3) *Activité d'analyse de MAPE-K et modèle d'évaluation* : La phase d'analyse est responsable d'analyser les modèles d'introspection pour identifier les besoins d'adaptation. Le modèle d'évaluation spécifie le raisonnement.
4) *Activité de planification de MAPE-K et modèle de changement* : La phase de planification détermine un plan décrivant l'adaptation sur le modèle d'introspection. Le modèle de changement décrit la variabilité de l'adaptation du système.
5) *Activité d'exécution de MAPE-K et modèle d'exécution* : La phase exécution met en place l'adaptation planifiée sur le système selon le modèle d'exécution. Ce modèle d'exécution guide l'exploration de l'état du système pour trouver une adaptation appropriée.

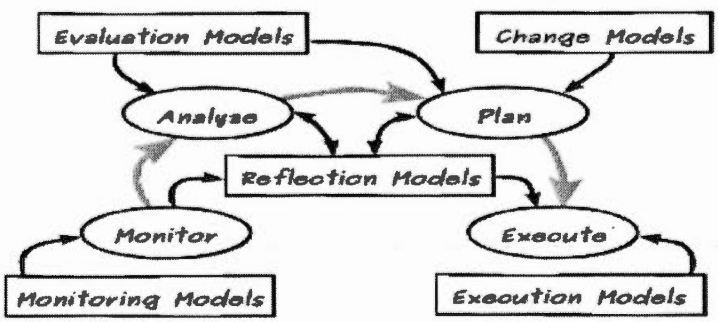


Figure 2.12 Modèle EUREMA (Vogel et Giese, 2013, 2014)

La figure 2.13 présente un exemple de *megamodel* pour une boucle de contrôle qui a pour objectif d'auto-réparer un système suite à des défaillances. Dans cet exemple,

toutes les activités d'adaptation s'y retrouvent. Par définition, ce *megamodel* contient d'autres modèles tels que *Failure Analysis Rules* et *Repair Strategies*.

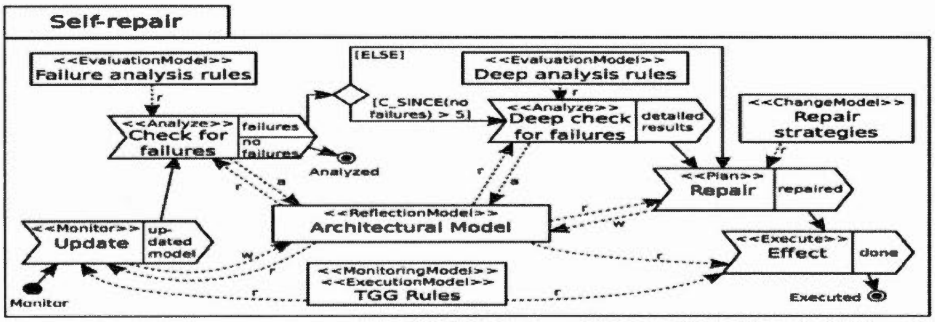


Figure 2.13 Exemple de *megamodel* (Vogel et Giese, 2013, 2014)

Un aspect intéressant du modèle EUREMA est que les boucles de contrôle sont spécifiées à un niveau abstrait en capturant l'interaction des activités d'adaptation et des modèles (*megamodels*) en temps réel. Ce modèle permet l'utilisation de boucles multiples à interagir alors que la plupart des modèles considèrent qu'une seule boucle de contrôle et garde ces boucles de contrôle actives en temps d'exécution.

2.18.4 Modèle de référence DYNAMICO

Tamura *et al.* (2013) ont présenté un modèle de référence pour les systèmes auto-adaptatifs, soit DYNAMICO (*Dynamic Adaptation Monitoring and Control Objectives*). Le modèle DYNAMICO (voir figure 2.14) fournit un guide de conceptualisation et d'implémentation de systèmes auto-adaptatifs. Il cible les systèmes nécessitant de faire face à des changements d'objectifs d'adaptation et d'exigences de surveillance de contexte en temps réel (Tamura *et al.*, 2013). DYNAMICO spécifie 3 boucles de contrôle feedback :

1) Boucle de contrôle *feedback* des objectifs (CO-FL) : effectue le suivi des changements des objectifs d'adaptation. Les points d'entrée des boucles A-FL et M-FL sont dérivés par la boucle de contrôle CO-FL. (Tamura *et al.*, 2013)

2) Boucle de contrôle *feedback* d'adaptation (A-FL) : modélise les mécanismes d'adaptation du système selon le comportement du système. Elle règlemente les exigences et s'assure de conserver les propriétés d'adaptation. Ces propriétés d'adaptation sont définies en tant que variables de systèmes qui requièrent un contrôle. (Tamura *et al.*, 2013)
3) Boucle de contrôle *feedback* de surveillance de contexte (M-FL) : est responsable de surveiller le contexte variable de l'environnement du système et les objectifs d'adaptation. (Tamura *et al.*, 2013)

Un aspect intéressant de ce modèle DYNAMICO est l'interaction entre les boucles de contrôle tout en caractérisant la séparation des préoccupations. Le modèle proposé permet également de surveiller le contexte dynamiquement, ce qui est un critère primordial dans le développement d'applications sensibles au contexte.

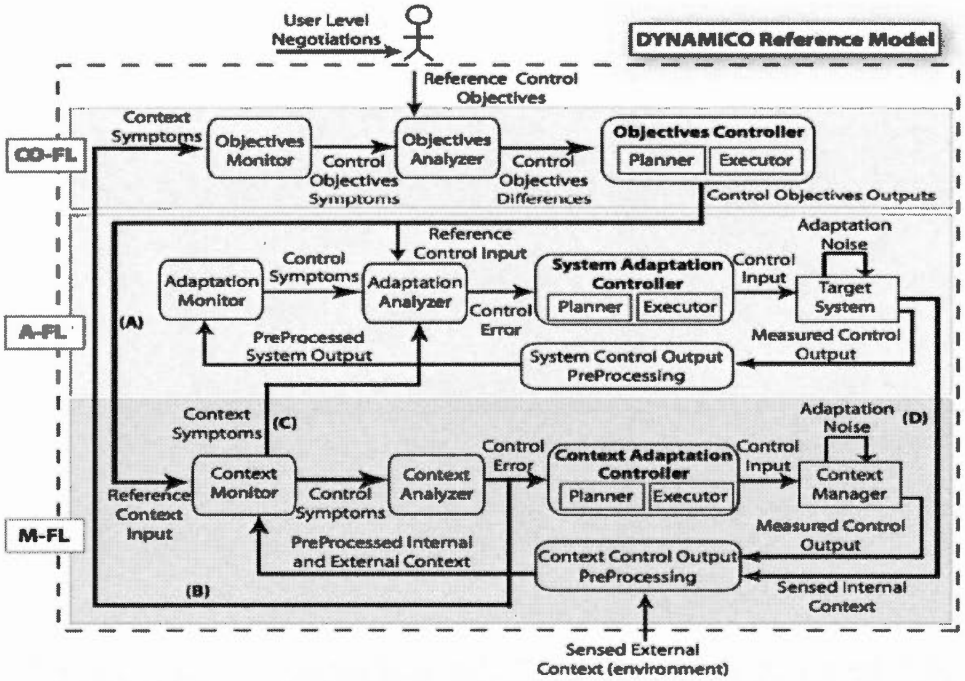


Figure 2.14 Modèle DYNAMICO (Tamura *et al.*, 2013)

2.18.5 Modèle en couches

Kramer et Magee (Kramer et Magee, 2007) ont présenté un modèle d'architecture à trois couches pour l'auto-gestion de systèmes. L'auto-gestion est une vision de systèmes capables de s'auto-configurer, s'auto-adapter, s'auto-réparer, s'auto-surveiller, s'auto-régler, etc. (Kramer et Magee, 2007). Les trois couches sont les suivantes (voir figure 2.15):

1) *Couche de contrôle de composants*: représente la couche de contrôle composée de capteurs, d'actionneurs et d'une boucle de contrôle. Elle détecte une situation que la configuration courante de composants n'est pas capable de gérer et, effectue le suivi auprès des couches supérieures. Son rôle est de signaler le statut courant du système aux couches supérieures et d'effectuer les changements requis tels que la création, la suppression et l'interconnexion de composants. (Kramer et Magee, 2007)
2) *Couche de gestion de changement*: réagit aux changements des statuts signalés ou aux nouveaux objectifs requis par le système. Elle exécute des plans qui nécessitent de nouveaux comportements tels qu'ajouter des composants, recréer des composants défectueux, et modifier l'interconnexion de composants. Cette couche permet également de modifier des paramètres de comportements existants. (Kramer et Magee, 2007)
3) *Couche de gestion des objectifs*: produit le plan requis en réponse aux demandes provenant de la couche de gestion de changement et introduit les nouveaux objectifs. (Kramer et Magee, 2007)

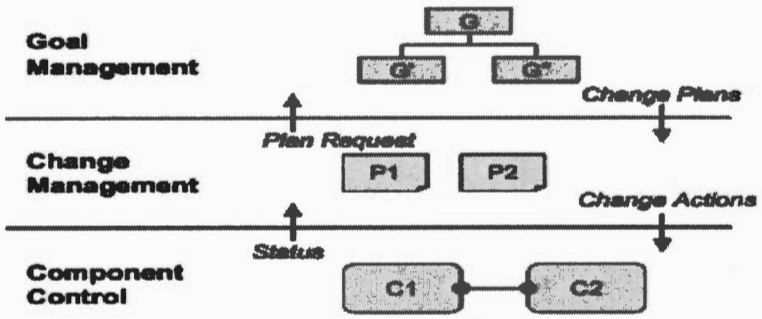


Figure 2.15 Modèle en couches (Kramer et Magee, 2007)

Le modèle de Kramer et Magee offre une généralité, c.-à-d. que l'approche proposée peut s'appliquer à un vaste domaine d'applications. Il offre un niveau d'abstraction pour décrire les changements dynamiques d'un système. Par contre, ce modèle engendre certains défis tels que s'assurer que l'information n'est pas perdue entre les différentes couches lors d'une configuration et spécifier des objectifs compréhensible par les utilisateurs et lisible par les systèmes.

2.19 Conclusion

Dans ce chapitre, nous avons présenté une revue de la littérature des principaux travaux de recherche en relation avec les contributions de cette thèse, soit la modélisation du contexte, le raisonnement sur le contexte et l'adaptation d'applications sensibles au contexte.

La notion de contexte est primordiale au développement d'applications dépendantes du contexte. Un contexte à l'état brut doit passer à travers diverses étapes afin d'être utilisé. Ces étapes représentent le cycle de vie du contexte telles que l'acquisition du contexte, la modélisation du contexte, le raisonnement sur le contexte et la dissémination du contexte (Li *et al.*, 2015; Perera *et al.*, 2014).

Bien que de nombreux travaux aient été proposés pour modéliser le contexte des applications sensibles au contexte, ces travaux ont été amenés à définir certains

critères qui doivent être pris en considération lors de la modélisation du contexte tel que le raisonnement, les liens et les dépendances (Bettini *et al.*, 2010). Dans la section 2.7 et 2.8, nous avons présenté quelques méthodes utilisées pour modéliser le contexte et raisonner sur ce contexte. Cependant, nous remarquons que certaines de ces méthodes de modélisation du contexte sont indépendantes de la méthode de raisonnement et certaines d'entre elles n'établissent pas de validation du modèle de contexte. Par conséquent, nous proposons d'utiliser une méthode qui permet de modéliser le contexte et de raisonner sur le contexte. Nous avons choisi d'appliquer l'analyse relationnelle de concepts (ARC) pour modéliser le contexte (Rouane-Hacene, Huchard, Napoli, et Valtchev, 2013b) et d'utiliser la logique descriptive pour raisonner sur le contexte pour les raisons suivantes :

1. ARC est une méthode qui permet d'analyser les données et de décrire formellement des concepts en les ordonnant hiérarchiquement dans une structure de treillis. Cette méthode permet d'établir des liens et des dépendances entre les contextes et les services des applications sensibles au contexte.
2. Des règles de correspondances entre les entités de l'analyse relationnelle de concepts et la logique descriptive existent, et par conséquent, il devient possible de raisonner sur le contexte modélisé.

Nous avons également présenté différentes approches d'adaptation utilisée par les applications sensibles au contexte qui tournent autour de l'utilisation d'une boucle de contrôle. Dans le cadre de ces applications, nous parlons d'une adaptation comportementale et/ou architecturale. Nous proposons deux approches d'adaptation, à savoir une approche basée une approche basée sur l'algèbre des processus pour effectuer une adaptation architecturale et sur les lignes de produits logiciels pour effectuer une adaptation comportementale.

Pour la première approche, nous proposons un modèle sémantique basé sur des composants. Nous utilisons des règles qui expriment la sémantique des opérations que chaque composant peut effectuer. À partir de ce modèle, nous restreignons le comportement du système en imposant des contraintes et/ou en transformant les opérateurs selon le contexte.

Quant au cas d'une adaptation basée sur les approches de lignes de produits, nous remarquons que l'utilisation des modèles de *features* ne permet de valider sémantiquement ni les modèles de *features* ni les configurations obtenues depuis ces modèles (Piash *et al.*, 2013; Wang *et al.*, 2007). Par conséquent, nous représentons le modèle de features d'une application sensible aux contextes sous forme d'ontologie pour valider la cohérence du modèle et de ses configurations, et nous établissons un lien entre le contexte modélisé et le modèle de *features*.

CHAPITRE III

MODÉLISATION DU CONTEXTE

3.1 Introduction

La plupart des informations contextuelles proviennent des capteurs et il y a habituellement un écart entre les données captées depuis un capteur et le niveau d'information qui est utile aux applications, et par conséquent, cet écart doit être comblé par divers traitements d'informations contextuelles avant qu'elles puissent être transmises à une application (Hoareau et Satoh, 2009). Les données contextuelles acquises sont structurées en format multiple et elles doivent être représentées sous un format unique afin d'être comprises et partagées (Li *et al.*, 2015).

Strang et Linnhoff-Popien (2004) ont défini des exigences qui permettent d'évaluer les modèles de contexte, soit la composition distribuée, la validation partielle, la richesse et la qualité de l'information, le niveau de formalisme, l'incomplétude et l'ambiguïté, et l'applicabilité aux environnements existants. Ces exigences ont pour objectif d'aborder les problèmes rencontrés lors de la modélisation du contexte. D'autres exigences ont été proposées pour modéliser le contexte tel que l'hétérogénéité, les relations et les dépendances, l'imperfection des données, le raisonnement et le temps.

Dans le chapitre 2, nous avons présenté quelques modèles de contexte, à savoir le modèle attribut-valeur, le modèle basé sur un langage de marquage, le modèle graphique, le modèle orienté-objet, le modèle basé sur la logique et le modèle basé

sur l'ontologie. De nombreux travaux portant sur la modélisation du contexte concluent que le modèle d'ontologie est celui qui semble le plus respecté certains des critères que nous avons mentionnés.

Nous avons décidé de modéliser le contexte des applications sensibles au contexte à partir de l'analyse relationnelle de concepts (Amja, Obaid, et Valtchev, 2014). Nous jugeons qu'il est important d'utiliser un modèle de contexte qui permet le raisonnement sur le contexte et nous jugeons qu'un modèle de contexte ne doit pas dépendre de la méthode de raisonnement choisie, et vice versa. Vu qu'il existe des règles de correspondance entre les entités de l'ARC et de la logique descriptive, cette méthode permet de modéliser le contexte tout en offrant le raisonnement. Nous avons également opté pour cette méthode, car elle permet d'établir des liens et des dépendances entre les contextes et les services offerts par une application sensible au contexte.

Dans ce chapitre, nous proposons de modéliser le contexte à partir de l'analyse relationnelle de concept. La section 3.2 présente un exemple de motivation qui sera utilisé tout au long de ce chapitre. Dans la section 3.3, nous faisons un rappel sur les notions de base de l'analyse formelle de concepts et nous présentons notre méthode d'échelonnage de contextes formels multi-valués en contextes formels mono-valués. Nous faisons un rappel sur les notions de base de l'analyse relationnelle de concepts dans la section 3.4 et nous expliquons en détail notre approche pour modéliser le contexte et les fonctionnalités contextuelles qui en dépendent à partir de l'analyse relationnelle de concept. Nous terminons par la section 3.5 en comparant notre modèle de contexte avec les modèles de contexte présentés dans le chapitre précédent selon certains critères et nous conluons le chapitre dans la section 3.6.

3.2 Exemple de motivation

Nous illustrons notre approche de modélisation du contexte pour un système de surveillance de l'état santé de patients. Un patient est entouré de différents types de contextes tels que l'humidité, l'intensité de la lumière, la température corporelle et la pression artérielle. Ces contextes ne sont pas tous pertienents pour l'application en question. La figure 3.1 illustre cet exemple.

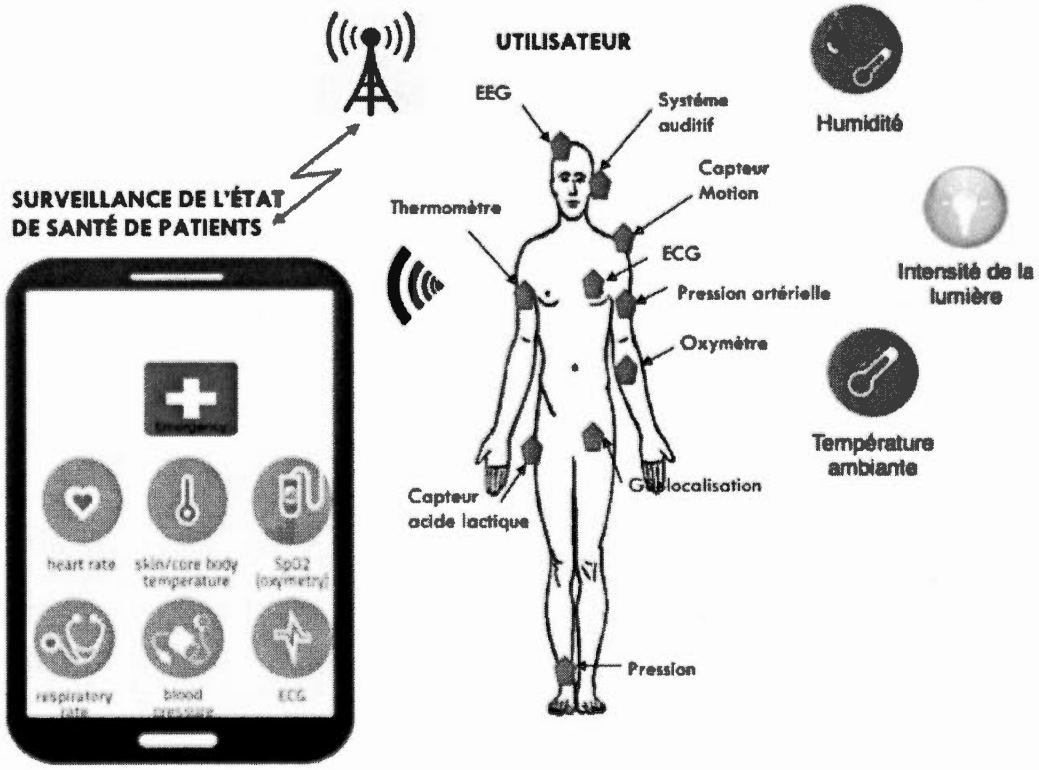


Figure 3.1 Utilisateur et ses diffèrents contextes

L'application offre des fonctionnalités dont chacune dépend d'un ou plusieurs contextes qui lui sont utiles et pertinents. Par exemple, les fonctionnalités de température corporelle, de pression artérielle et d'urgence dépendent des valeurs de contextes obtenus par les capteurs de température corporelle, de pression artérielle et

de géolocalisation, respectivement. Ainsi, les autres contextes sont inutiles pour ces fonctionnalités. La figure 3.2 illustre cet exemple.

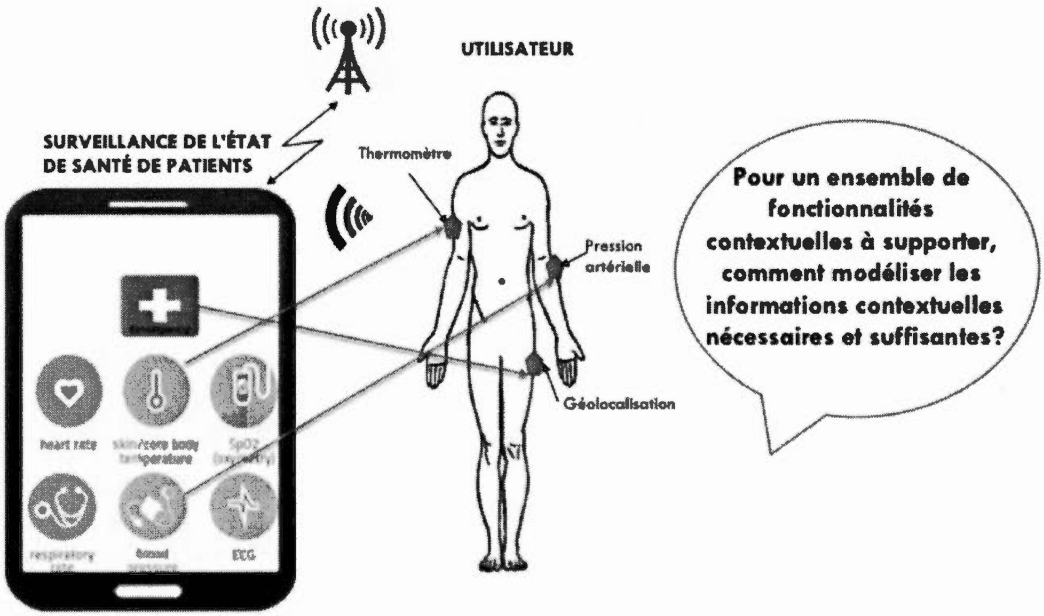


Figure 3.2 Fonctionnalités contextuelles et contextes

Le contexte est une information implicite captée par des sources hétérogènes et réparties pour être utilisées par des applications sensibles au contexte. Il reflète l'état dynamique d'un environnement et d'un utilisateur. Il doit donc capturer tous les aspects de l'information contextuelle: sa source, son identité, son type, sa date, etc. Par exemple, un appareil mobile Android fournit les informations contextuelles de geo-localisation d'un patient que nous décrivons en généralisant la méthode attribut-valeur (voir figure 3.3).

```
Context Name="Geographical Coordinates"
        (Attribut="Latitude" Value="45,530017")
        (Attribut="Longitude" Value="73.760434")
        (Attribut="Provider" Value="Network")
        (Attribut="Accuracy" Value="1046.0")
        (Attribut="Time to fix" Value="0")
        (Attribut="Enabled Providers" Value="Network")
        Timestamp (ns)="1458066944.828372
```

Figure 3.3 Méthode attribut-valeur

Dans ce chapitre, nous proposons une approche qui répond à la question suivante : « Pour un ensemble de fonctionnalités contextuelles à supporter, comment est-ce qu'on peut modéliser les informations contextuelles nécessaires et suffisantes ? »

Nous notons que nous montrons la modélisation du contexte pour cet exemple de motivation en considérant uniquement 3 types de contextes, à savoir : la température corporelle, la pression artérielle et la géo-localisation.

3.3 Analyse formelle de concepts

L'analyse formelle de concepts est une méthode mathématique de dérivation de structures conceptuelles fondée sur la théorie des treillis, les ordres partiels et les correspondances de Galois. Elle a été introduite par Rudolf Wille (1999) et repose sur les travaux antérieurs de Garrett Birkhoff (Birkhoff, 1940). Un des objectifs de Wille était de faciliter l'utilisation de cette approche dans des applications du monde réel et de permettre l'interprétation de ses notions.

L'analyse formelle de concepts a évolué en tant que méthode d'analyse de données spécialisée dans l'extraction d'un ensemble de concepts à partir d'un ensemble de données. Elle est devenue un cadre d'analyse qui s'applique à des applications de différents domaines tels que la recherche d'information, la fouille de données et la construction d'ontologies.

3.3.1 Contextes formels mono-valués

Définition 3.1 Contexte formel mono-valué. Un contexte formel mono-valué est un triplet $K = (G, M, I)$ où G est un ensemble d'objets, M est un ensemble d'attributs mono-valués et I est une relation binaire entre G et M appelée *Relation d'incidence* de K satisfaisant $I \subseteq GxM$. Un couple $(g, m) \in I$ (noté aussi gIm) signifie que l'objet $g \in G$ possède l'attribut $m \in M$.

Un contexte formel est représenté sous la forme d'un tableau où les lignes correspondent aux objets et les colonnes correspondent aux attributs. Si le $i^{ième}$ objet g est en relation d'incidence I avec le $j^{ème}$ attribut m, alors la case de ligne i et de colonne j contient un « x » sinon la case est vide (Carpineto et Romano, 2004). Nous présentons un exemple de contexte formel pour la température corporelle dans le tableau 3.1. Les instances de température corporelle et leurs caractéristiques représentent des objets et des attributs, respectivement.

Tableau 3.1 Contexte formel de la température corporelle

	PrecisionFaible	**PrecisionMoyenne**	**PrecisionElevee**	**ValeurTemperatureCluster1**	**ValeurTemperatureCluster2**	**ValeurTemperatureCluster3**	**Capteur1**	**Capteur2**
TemperatureCorporelle1	X			X			X	
TemperatureCorporelle2			X			X	X	
TemperatureCorporelle3		X			X			X

3.3.2 Méthode d'échelonnage de contexte formel multi-valué

Dans le monde réel, les objets sont décrits par des attributs qui peuvent prendre plusieurs valeurs plutôt que par la présence ou l'absence de certaines propriétés, appelée des *attributs mono-valués*. Ces types d'attributs sont appelés des *attributs multi-valués*.

Définition 3.2 Contexte formel multi-valué. Un contexte formel multi-valué est un quadruplet (G, M, W, I) où G est un ensemble d'objets, M est un ensemble d'attributs multi-valués, W est l'ensemble de valeurs prises par les attributs et $I \subseteq GxMxW$ est une relation ternaire entre G, M et W, telle que :

$$(g, m, w) \in I \text{ et } (g, m, v) \in I \text{ implique } w = v.$$

Les expressions $(g, m, w) \in I$ et $I(g, m) = w$ sont équivalentes et signifient que l'attribut m a la valeur w pour l'objet g.

Le contexte formel multi-valué doit être représenté sous un format de contexte formel mono-valué tel qu'expliqué à la définition 3.1 dans le but d'utiliser l'analyse formelle de concepts. À partir de la méthode de attribut-contexte, nous utilisons des techniques de *clustering* pour des données numériques et la méthode d'échelonnage conceptuel de Wille (Ganter et Wille, 1989) pour les données catégoriques pour obtenir un contexte formel mono-valué.

Nous illustrons cette utilisation à partir d'un exemple d'un contexte de température corporelle. La figure 3.4 illustre la correspondance de la méthode attribut-valeur et le contexte formel multi-valué pour le contexte de température corporelle. Nous utilisons certains attributs qui nous semblent pertinents pour cette illustration.

(**Context name**="TemperatureCorporelle1"
Type de capteur="Temperature Corporelle"
Max Range="165"
Min Delay="1000000"
Power="0.3"
Vendeur="Sensirion"
Version="1"
Precision="Sensor_Status_Medium"
Body Temperature Ambiente="37.578294"
Timestamp="2014-02-07"
)

	Type de capteur	Vendeur	Version	Précision	Valeur	Source
TemperatureCorporelle1	Température Corporelle	Sensirion	1	Low	37.578294	SHTC1 Body Temperature Sensor

Figure 3.4 Exemple de correspondance de la méthode attribut-valeur et le contexte formel multi-valué

3.3.2.1 Échelonnage des données numériques

Nous établissons des *clusters* basés sur la similarité des valeurs de chaque attribut pour établir les liens entre les données numériques. Celles-ci seront groupées selon la méthode de *clustering K-mean* basé sur la distance euclidienne (MacQueen, 1967). Nous pouvons, toutefois, utiliser la distance de Manhattan (Gan, Ma et Wu, 2007) ou de Minkowski (Gan *et al.*, 2007) puisque la distance euclidienne et Manhattan sont des cas particuliers de la distance Minkowski.

Le *clustering k-mean* consiste à grouper les objets sous *k clusters*. Pour chaque objet, il faut considérer la distance d'un point à la moyenne des points de son *cluster*. Comme il n'existe pas à priori un nombre optimal de *clusters*, nous devons donc le fournir.

Les étapes de l'algorithme *k-mean* sont les suivantes :

1) Regrouper les données en *k clusters* où *k* est prédéfini.

2) Sélectionner k points au hasard comme étant des centres de clusters.
3) Assigner les objets au cluster le plus proche selon la fonction de distance euclidienne.
4) Calculer le barycentre (*centroid*) de tous les objets dans chaque cluster.
5) Répéter les étapes 2, 3 et 4 jusqu'à ce que tous les points soient assignés à chaque cluster

Le *clustering k-mean* se définit mathématiquement comme suit (MacQueen, 1967):

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2$$

Supposons que nous voulons grouper les températures à partir des valeurs captées sous 2 *clusters* différents : 36, 37, 39, 40 et 41. Les *centroids* initiaux sont *Centroid* (C1)=0 et *Centroid* (C2)=0. À la première itération, nous obtenons C1=36 [36] et C2=39.25 [37 39 40 41]. La seconde itération donne C1=36.5 [36 37] et C2=40 [39 40 41]. Nous obtenons à la troisième itération C1=36.5 [36 37] et C2=40 [39 40 41]. Comme il n'y a pas eu de changement entre les itérations 2 et 3, l'algorithme s'arrête. Par cette méthode de *clustering*, 2 groupes ont été identifiés K1[36 37] et K2[39 40 41]. Le choix initial des *centroids* peut affecter le résultat des clusters, alors l'algorithme est exécuté à plusieurs reprises avec différentes initialisations de *centroids* afin d'obtenir des *clusters* raisonnables.

Une fois les données de chaque *cluster* obtenues, nous pouvons représenter les attributs multi-valués de données numériques sous un contexte formel (voir figure 3.5).

	Valeurs
TemperatureCorporelle1	36
TemperatureCorporelle2	37
TemperatureCorporelle3	39
TemperatureCorporelle4	40
TemperatureCorporelle5	41

	Cluster1: Valeurs	Cluster2: Valeurs
TemperatureCorporelle1	x	
TemperatureCorporelle2	x	
TemperatureCorporelle3		x
TemperatureCorporelle4		x
TemperatureCorporelle5		x

Figure 3.5 Exemple d'échelonnage des données numériques

Vu que cet algorithme utilise la distance euclidienne, cette fonction s'applique à d-dimensions. Nous pouvons donc appliquer le *clustering k-mean* pour tous les types de données numériques telles que les valeurs de température, de proximité, de pourcentage d'humidité relative et les coordonnées géographiques.

3.3.2.2 Échelonnage des données catégoriques

Les données catégoriques seront groupées en utilisant l'échelonnage conceptuel qui a été proposé par Wille (Ganter et Wille, 1989). Cette méthode consiste à transformer chaque attribut multi-valué en un ensemble d'attributs mono-valués afin de former un contexte formel mono-valué, appelé échelle conceptuelle de l'attribut mono-valué. Une échelle conceptuelle est un contexte formel dont les objets sont les valeurs et les attributs sont des attributs d'échelle.

Par exemple, nous représentons l'échelle conceptuelle de l'attribut *précision* pour les objets *température corporelle*, soit : précision faible, précision moyenne et précision élevée (voir figure 3.6).

	Précision
TemperatureCorporelle1	Faible
TemperatureCorporelle2	Faible
TemperatureCorporelle3	Moyenne
TemperatureCorporelle4	Élevé
TemperatureCorporelle5	Moyenne

	Précision: Faible	Précision: Moyenne	Précision: Élevée
TemperatureCorporelle1	x		
TemperatureCorporelle2	x		
TemperatureCorporelle3		x	
TemperatureCorporelle4			x
TemperatureCorporelle5		x	

Figure 3.6 Échelonnage des données catégoriques

3.3.2.1 Algorithme d'échelonnage

Notre approche d'échelonnage se résume par l'algorithme ci-dessous (voir figure 3.7). Cet algorithme permet de transformer les contextes formels multi-valués sous un format mono-valués. Il se compose de quatre fonctions principales, soit :

- *GenerateOneValuedContextProgram* : fait appel à d'autres fonctions selon le type de données nécessitant une transformation. Les types de données considérées sont les données numériques et catégoriques.
- *ComputeCollectionCategory*: est responsable de générer des listes pour chaque type de données contextuelles, c'est-à-dire des données de type catégorique, numérique et booléen.
- *ComputeCollectionSpatial* : fait appel à la fonction ComputeClustersUsingKMeans et génère les contextes formels dont le type de données est numérique.
- *GenerateOneValuedContextCategory* : génère les contextes formels dont le type de données est catégorique. Cette fonction se base sur la méthode d'échelonnage expliquée dans la section 3.3.2.2.
- *ComputeColumnList* : fournit des contextes formels mono-valués dont les types de données sont catégoriques, numériques et booléens.

- *ComputeClustersUsingKmeans* : détermine les clusters des données numériques selon le *clustering k-mean* que nous avons expliqué dans la section 3.3.2.1.

```
Function GenerateOneValuedContextProgram
        Inputfile <- Ouvrir pour lecture (Fichier input XML avec des contextes formels multi-valués)
        Outputfile <- Ouvrir pour écriture (Fichier output RCFT)
        For Each <Contexte> element E in Inputfile
                Trouver le type de contexte T pour l'élément E
                Ajouter E à la collection d'objets de type T
        Next
        For Each collection C
                If type de C est "Spatial" then
                        ComputeCollectionSpatial(C, Outputfile)
                Else If type de C est "Category" then
                        ComputeCollectionCategory(C, Outputfile)
                Else If type de C est "..." then
                        ComputeCollection...(C, Outputfile)
                End If
        Next
End Function
```

```
Function ComputeCollectionCategory(Contexts, Outputfile)
        StringCategoryCollection[] <- empty map
        NumericCollection[] <- empty map
        BooleanAttributesList[] <- empty map
        ClustersMap[] <- empty map
        For Each element E in collection Contexts
                Name <- Trouver attribut "name"  de E
                Type <- Trouver attribut "type" de E
                Value <- Trouver attribut "value" de E
                If Type = "string-category" Then
                        Ajouter valeur à StringCategoryCollection[Name]
                Else If Type = "numeric" Then
                        Ajouter valeur à NumericCollection[Name]
                Else If Type = "boolean" Then
                        Ajouter valeur à BooleanAttributesList
                End If
        Next
        For Each numeric collection C in Contexts
                Clusters <- ComputeClustersUsingKMeans( C )
                ClustersMap[C] <- Clusters
        Next
    ColumnList<-CompileColumnList(BooleanAttributesList,StringCategoryCollection, ClustersMap )
    GenerateOneValuedContextsCategory(C, ColumnList, ClustersMap, Outputfile )
End Function
```

```
Function ComputeCollectionSpatial( Contexts, Outputfile )
```

```
        LocationsList <- ComputeLocations(Contexts)
        Clusters <- ComputeClustersUsingKMeans(Locations)
        ColumnList <- Trouver les noms distincts de cluster de Clusters
        For Each context C in Contexts
                ObjectId <- Trouver le nom unique du contexte courant C
                Écrire ObjectId à Outputfile
                For Each column Col in ColumnList
                        If le nom du cluster associé avec la localisation du contexte (context)
                        courant C correspond au nom de la colonne (column) courant Then
                                Mettre un "x" dans la cellule
                        Else
                                Mettre un espace dans la cellule
                        End If
                        Ajouter un séparateur de cellule à Outputfile
                Next
                Ajouter une nouvelle ligne à Outputfile
        Next
End Function
```

```
Function GenerateOneValuedContextsCategory(Contexts, ColumnList, ClustersMap, Outputfile)
        Écrire les titres du contexte de type Catégorie à Outputfile
        For Each context C in Contexts
                ObjectId <- Trouver le nom unique du contexte (context) courant C
                Écrire ObjectId à Outputfile
                For Each column Col in ColumnList
                        If contexte (context) courant C correspond à la colonne courante (column)
                        Then
                                Mettre un "x" dans la cellule
                        Else
                                Mettre un espace dans la cellule
                        End If
                        Ajouter un séparateur de cellule à Outputfile
                Next
                Ajouter une nouvelle ligne à Outputfile
        Next
End Function
```

```
Function CompileColumnList(BooleanAttributesList, StringCategoryCollection, ClustersMap)
        ColumnList <- empty list
        Compiler les noms distincts des colonnes des noms d'attributs des valeurs booléennes, des
        valeurs catégoriques et numériques
End Function

Function ComputeClustersUsingKMeans(data)
        Grouper les données (data) en K groupes de cluster
        Repeat
                Sélectionner K points en tant que centroid du cluster
                Assigner les objets au centroid du cluster le plus proche basé sur une distance
                Euclidienne
```

Recalculer la moyenne du *centroid* pour tous les objets du cluster
Until (valeurs de cluster de l'itération courante != valeurs de cluster de l'itération précédente)
End Function

Figure 3.7 Algorithme d'échelonnage

3.3.3 Connexion de Galois dans un contexte formel

Définition 3.3 Connexion de Galois dans un contexte formel. Soit un contexte formel. Pour tout $A \in G$ et $B \in M$, nous définissons :

$$A' = \{m \in M | \forall g \in A, gIm\}$$

$$B' = \{g \in G | \forall m \in B, gIm\}$$

A' est l'ensemble des attributs communs à tous les objets de A et B' est l'ensemble des objets partageant tous les attributs de B. A' et B' sont appelés *Opérateurs de dérivation* entre l'ensemble des objets et l'ensemble des attributs dans un contexte formel (Carpineto et Romano, 2004).

Ces opérateurs de dérivation peuvent être combinés sous une paire d'opérateurs composés. Le premier opérateur A'' permet d'associer à un ensemble d'objets A l'ensemble maximal d'objets dans G ayant les attributs communs aux objets de A.

Le second opérateur B'' permet d'associer à un ensemble d'attributs B, l'ensemble maximal d'attributs dans M communs aux objets ayant les attributs dans B.

Les opérateurs de dérivation A' et B' induisent une connexion de Galois entre $\wp(G)$ et $\wp(M)$. Il en suit que les opérateurs A'' et B'' sont des opérateurs de fermeture qui induisent deux ensembles de fermés : $\wp(G)$ et $\wp(M)$.

Une fermeture dans un ensemble ordonné est une application, $\varphi: \wp(H) \rightarrow \wp(H)$, qui pour tout $A, B \subseteq H$ vérifie les propriétés suivantes :

1) Extensive : $A \subseteq \varphi(A)$. On dit que φ est extensive si et seulement si $\forall A \in \wp(H), A \subseteq \varphi(A)$
2) Monotone croissant : Si $A \subseteq B$, alors $\varphi(A) \subseteq \varphi(B)$. On dit que φ est monotone croissant si et seulement si $\forall A, B \in \varphi(H), A \subseteq B \Longrightarrow \varphi(A) \subseteq \varphi(B)$.
3) Idempotent : $\varphi\big(\varphi(A)\big) = \varphi(A)$. On dit que φ est idempotent si et seulement si $\forall A \in \wp(H), \varphi\big(\varphi(A)\big) = \varphi(A)$

Carpineto et Romano (2004) appellent *fermeture* de H une application de $\wp(H)$ dans $\wp(H)$ qui est extensive, monotone croissante et idempotente. Si φ est une *fermeture*, nous appelons *fermé* un ensemble $A \subseteq H$ tel que $\varphi(A) = A$. Soit φ une fermeture sur H, alors l'ensemble des fermés ordonné par inclusion forme un treillis.

3.3.4 Concept formel

Définition 3.4 Concept formel. Soit $K = (G, M, I)$ un contexte formel. Un concept formel est un couple (A, B) tel que $A \subseteq G$, $B \subseteq M$, $A' = B$ et $B' = A$. A et B sont respectivement appelées *extension* et *intension* du concept formel (A, B). L'ensemble des concepts formels associés au contexte formel $K = (G, M, I)$ est noté par $C(G, M, I)$.

Par exemple, pour le contexte formel de la température corporelle dans le tableau3.1, nous aurons au moins ces 3 concepts formels :

- C({TemperatureCorporelle1, TemperatureCorporelle2}, {Capteur1}
- C{TemperatureCorporelle3}, {ValeurTemperatureCluster2, PrecisionMoyenne, Capteur2})
- C(TemperatureCorporelle1}, {ValeurTemperatureCluster1, PrecisionFaible, Capteur1})

3.3.5 Construction des concepts formels

La construction d'un ensemble de concepts formels a fait l'objet de plusieurs travaux de recherche qui ont permis d'aboutir à différents algorithmes (Carpineto et Romano, 2004). Ces algorithmes se distinguent les uns des autres par certains critères tels que la méthode de génération des concepts, la recherche de l'ordre entre ces concepts et les structures de données utilisées pour le stockage des concepts formels intermédiaires et finaux (Carpineto et Romano, 2004).

Un des algorithmes utilisés est l'algorithme naïf. Il consiste à générer des concepts formels d'un contexte formel (G, M, I) en formant (A'', A') pour tout $A \subseteq G$, ou (B'', B') pour tout $B \subseteq M$ (Carpineto et Romano, 2004). Cette méthode est simple, mais elle n'est pas très efficace, car elle requiert de considérer tous les sous-ensembles de G ou M avec leur fermeture correspondant, et ce, peu importe la taille des concepts générés.

Un autre exemple d'algorithme est le « next closure » qui génère un ensemble de concepts formels en utilisant des fermetures basées sur l'ordre des sous-ensembles de G de telle sorte que seuls les sous-ensembles nécessitent d'être examinés (Carpineto et Romano, 2004). Cet algorithme impose un ordre lexicographique, dénoté $\leq$, sur les sous-ensembles de G. Un sous-ensemble A de G est plus petit que le sous-ensemble B de G si l'élément le plus petit qui distingue A et B appartient à B. L'algorithme considère un sous-ensemble A à la fois sous un ordre lexicographique, jusqu'à ce que le dernier élément du sous-ensemble G soit généré. Si A'' ne contient pas d'objets qui sont plus petits que ceux dans A, alors le concept formel avec l'extension A'' est ajouté à l'ensemble des concepts et le prochain sous-ensemble de G à être examiné est le successeur de A''. Si la condition n'est pas satisfaite, le concept avec l'extension A'' n'est pas ajouté à la liste de concepts formels, car il sera généré plus tard en examinant le sous-ensemble de A'', et le prochain sous-ensemble de l'ordre

lexicographique est généré. L'avantage de cette approche est que chaque concept est créé une seule fois.

Une autre approche pour trouver les concepts formels se base sur l'observation que chaque extension d'un concept est l'intersection des attributs de l'extension et chaque intension d'un concept est l'intersection des objets de l'intension (Carpineto et Romano, 2004). Pour générer l'ensemble des concepts, il faut donc former toutes les intersections possibles entre les ensembles d'objets associés avec chaque attribut et utiliser le contexte pour trouver l'intension correspondant pour chaque extension de concept généré. Il faut considérer chaque attribut et calculer l'intersection entre ces extensions, et chaque extension d'un concept de l'ensemble de concepts courant contient les concepts générés par les attributs qui ont déjà été examinés. L'inconvénient de cette approche est que les mêmes concepts peuvent être générés plusieurs fois.

3.3.6 Relation de subsomption

Définition 3.5 Relation de subsomption. Soient (A_1, B_1) et (A_2, B_2) deux concepts formels de $C(G, M, I)$. $(A_1, B_1) \sqsubseteq (A_2, B_2)$ ssi $A_1 \subseteq A_2$ (ou de façon duale $B_1 \subseteq B_2$). (A_2, B_2) est dit super-concept de (A_1, B_1) et (A_1, B_1) est dit sous-concept de (A_2, B_2). La relation « $\sqsubseteq$ » est dite relation de subsomption.

Par exemple, les concepts formels C({TemperatureCorporelle1}, {ValeurTemperatureCluster1, PrecisionFaible, Capteur1}) et C({TemperatureCorporelle1, TemperatureCorporelle2},{Capteur1}) ont une relation de subsumption comme suit : C({TemperatureCorporelle1, TemperatureCorporelle2},{Capteur1}) $\sqsubseteq$ C({TemperatureCorporelle1}, {ValeurTemperatureCluster1, PrecisionFaible, Capteur1}).

Les concepts formels extraits du contexte formel et organisés par la relation de subsomption $\sqsubseteq$. La relation de subsomption s'appuie entre l'ensemble d'objets et

entre l'ensemble d'attributs et s'interprète comme une relation de généralisation entre les concepts formels.

3.3.7 Treillis de concepts

Définition 3.6 Treillis de concepts. La relation « $\sqsubseteq$ » permet d'organiser les concepts formels en un treillis complet. La relation $(C(G,M,I),\sqsubseteq)$, appelé treillis de concepts, est notée par $C(G,M,I)$.

Par exemple, la figure 3.8 illustre le treillis de concepts pour le contexte formel de la température corporelle (voir tableau 3.1).

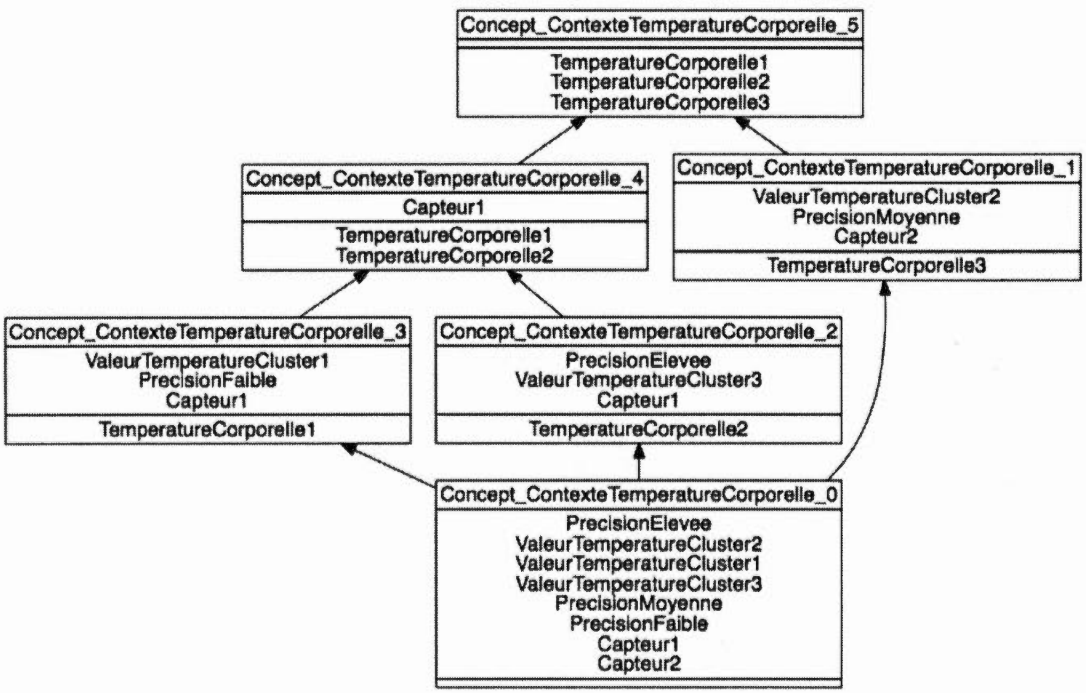


Figure 3.8 Treillis de concepts du contexte de *température corporelle*

3.3.8 Construction du treillis de concepts

La représentation graphique du treillis se fait par la construction du diagramme de Hasse en générant itérativement les voisins selon la relation $\prec$. La construction se fait

un niveau à la fois en débutant par le sommet du treillis (« top »). Le niveau qui suit contient les enfants des concepts formels du niveau précédent. Pour chaque concept du niveau courant, les voisins d'un concept formel des enfants (niveau plus bas) sont calculés. Si un enfant n'a pas déjà été généré alors il est ajouté au treillis et ce concept est lié à son parent. Il s'agit donc une approche descendante et de parcours en largeur (*top-down breadth-first*), mais un parcours en profondeur fonctionnerait également.

3.4 Analyse relationnelle de concepts

Comme présenté précédemment, l'analyse formelle de concepts est un formalisme de représentation et d'extraction de connaissances fondé sur les concepts. C'est une méthode mathématique qui permet d'organiser des objets en fonction de leurs propriétés (attributs) en dérivant un treillis de concepts à partir d'un contexte formel. Nous avons facilement pu représenter le contexte de la température corporelle à partir de cette technique. Nous pouvons également utiliser cette méthode pour représenter les fonctionnalités contextuelles de ce type d'application. Par contre, l'analyse formelle de concepts ne nous permet pas d'établir un lien entre les fonctionnalités contextuelles d'une application et ses contextes, alors nous avons choisi d'utiliser l'analyse relationnelle de concepts, soit une extension de l'AFC.

L'analyse relationnelle de concepts permet l'extraction de concepts formels en tenant compte des relations entre les objets de différents contextes formels (Huchard *et al.*, 2007). L'ARC génère une famille de treillis de concepts, précisément un treillis par catégories d'objets, et connecte les concepts formels de ces treillis par des attributs relationnels.

Nous notons que les objets des contextes formels des contextes qui entourent un patient sont des instances de contextes alors que les attributs de ces contextes formels représentent des propriétés fournies par les capteurs des appareils mobiles Android tels que la précision et la valeur captée. Nous notons également que pour représenter les fonctionnalités contextuelles d'une application sensible au contexte, nous avons

utilisé la technique de *Feature Binding Analysis* (Lee et Kang, 2004; Lee et Muthig, 2008, 2009) qui consiste à grouper des fonctionnalités communes en composant. Dans ce chapitre, nous appelons ces composants *Services*. Dans ce qui suit, nous montrons la modélisation du contexte et des fonctionnalités contextuelles qui en dépendent à partir de l'analyse relationnelle de concepts. Dans le cadre de notre exemple de surveillance de l'état de santé de patients, notre modélisation est faite en utilisant l'outil *RCAExplore* (Dolques, 2014). Nous rappelons que nous avons considéré 3 contextes, à savoir : la température corporelle, la pression artérielle et la géolocalisation.

3.4.1 Famille de contextes relationnels

La structure de ARC se centre sur un ensemble de contextes formels et un ensemble de relations (Rouane-Hacene *et al.*, 2013a).

Définition 3.7 Famille de contextes relationnels. Une famille de contextes relationnels (FCR) est un couple (K, R) où :

- K est un ensemble de contextes formels $K_i = (G_i, M_i, I_i)$
- R est un ensemble de relations $R_j = (G_k, G_i, I_j)$
    - (G_k, G_i) sont des ensembles d'objets de contextes formels $(K_k, K_i) \in K^2$
    - $I_j \subseteq G_k x\, G_i$
    - K_k est le contexte source (domaine), K_i est le contexte ciblé (co-domaine). Nous pouvons avoir $K_k = K_i$

La FCR se compose de 4 contextes formels et 3 relations binaires : *localisation*, *température corporelle*, *pression artérielle*, *services*, *dependLocalisation*, *dependTemperatureCorporelle* et *dependPressionArterielle*. À partir de ces relations binaires, nous serons en mesure d'établir un lien entre les contextes et les fonctionnalités contextuelles de l'application qui en dépendent.

Dans le tableau 3.2, nous montrons le contexte formel de la *localisation* de patients, où les objets sont des patients et les attributs sont des propriétés décrivant l'emplacement des patients. Une case *(i, j)* dans le tableau indique que l'objet *i* est décrit par l'attribut *j*, ou de façon équivalente, l'attribut *j* décrit l'objet *i*. Par exemple, le patient 1 (LocalisationUtilisateur1) a un emplacement dans le cluster 1, alors qu'appartenir à l'emplacement dans le cluster 2 est partagé par les patients 2 (LocalisationUtilisateur2) et 3 (LocalisationUtilisateur3).

Tableau 3.2 Contexte formel de la localisation de patients

	ValeurLocalisationCluster1	**ValeurLocalisationCluster2**
LocalisationUtilisateur1	X	
LocalisationUtilisateur2		X
LocalisationUtilisateur3		X

Dans le tableau 3.3, nous montrons le contexte formel de la *température corporelle*, où les objets sont des instances de température corporelle et les attributs sont des propriétés décrivant les valeurs de température corporelle captées. Par exemple, la TemperatureCorporelle1 a une valeur de précision faible appartenant au cluster 1 et captée par le capteur1, alors qu'une valeur de température corporelle captée par le capteur1 et partagée par TempératureCorporelle1 et TempératureCorporelle2.

Tableau 3.3 Contexte formel de la température corporelle

	PrecisionFaible	PrecisionMoyenne	PrecisionElevee	ValeurTemperatureCluster1	ValeurTemperatureCluster2	ValeurTemperatureCluster3	Capteur1	Capteur2
TemperatureCorporelle1	X			X			X	
TemperatureCorporelle2			X			X	X	
TemperatureCorporelle3		X			X			X

Dans le tableau 3.4, nous montrons le contexte formel de *pression artérielle*, où les objets sont des instances de pression artérielle et les attributs sont lss propriétés décrivant les valeurs de pression artérielle captées. Par exemple, la PressionArterielle2 a une valeur de précision moyenne appartenant au cluster 2 et captée par le capteur2, alors qu'une valeur de pression corporelle captée par le capteur2 et partagée par PressionArterielle2 et PressionArterielle3.

Tableau 3.4 Contexte formel de la pression artérielle

	PrecisionFaible	PrecisionMoyenne	PrecisionElevee	ValeurTemperatureCluster1	ValeurTemperatureCluster2	ValeurTemperatureCluster3	Capteur1	Capteur2
PressionArterielle1	X			X			X	
PressionArterielle2		X			X			X
PressionArterielle3			X			X		X

Dans le tableau 3.5, nous montrons le contexte formel des *services*, où les objets sont des services et les attributs sont des propriétés décrivant les fonctionnalités contextuelles de ces services. Par exemple, le service *GeoLocalisation* offre l'emplacement des patients. Dans cet exemple, nous n'avons pas de propriétés partagées par les services, car chaque service offre une ou plusieurs fonctionnalités spécifiques dont ces derniers ne peuvent être offerts par d'autres services.

Tableau 3.5 Contexte formel des services

	F1Communication	F11Wifi	F12Zigbee	F2Authentification	F3GeoLocalisation	F31GPS	F32Wifi	F4NotificationPatient	F41PrioriteAlerte	F411Fifo	F412Priorite	F42Alertes	F421AvertisseurEcran	F422AvertisseurSonore	F5Alarme	F51DetectionChute	F52BoutonPanique	F6Capteur	F61CapteursPhysiologi	F611PressionArterielle	F612TempCorporelle	F62CapteursEnvironne	F621TempAmbiante	F622Lumiere	F7Soins
ServiceCommunication	X	X	X																						
ServiceAuthentification				X																					
ServiceGeoLocalisation					X	X	X																		
ServiceNotificationPatient								X	X			X													
ServiceAlarme															X										
ServiceCapteur																		X	X			X			
ServiceSoins																									X
ServiceOptionPrioriteAlerte										X	X														
ServiceAvertisseur													X	X											
ServiceOptionAlarme																X	X								
ServiceSignesVitaux																				X	X				
ServiceEnvironnement																							X	X	

Dans le tableau 3.6, la relation binaire *dependLocalisation* établit un lien entre les *Services* et le contexte *Localisation.*

Tableau 3.6 Relation binaire entre les objets services et localisation des utilisateurs

	LocalisationUtilisateur1	LocalisationUtilisateur2	LocalisationUtilisateur3
ServiceCommunication			
ServiceAuthentification			
ServiceGeoLocalisation	X	X	X
ServiceNotificationPatient			
ServiceAlarme	X	X	X
ServiceCapteur	X	X	X
ServiceSoins	X	X	X
ServiceOptionPrioriteAlerte			
ServiceAvertisseur			
ServiceOptionAlarme	X	X	X
ServiceSignesVitaux			
ServiceEnvironnement			

Dans le tableau 3.7, la relation binaire *dependTemperature* établit un lien entre les *Services* et le contexte *TemperatureCorporelle*.

Tableau 3.7 Relation binaire entre les services et les températures

	TemperatureCorporelle1	TemperatureCorporelle2	TemperatureCorporelle3
ServiceCommunication			
ServiceAuthentification			
ServiceGeoLocalisation	X	X	X
ServiceNotificationPatient			
ServiceAlarme	X	X	X
ServiceCapteur	X	X	X
ServiceSoins	X	X	X
ServiceOptionPrioriteAlerte			
ServiceAvertisseur			
ServiceOptionAlarme	X	X	X
ServiceSignesVitaux	X	X	X
ServiceEnvironnement			

Dans le tableau 3.8, la relation binaire *dependPressionArterielle* établit un lien entre les *Services* et le contexte *PressionArterielle*.

Tableau 3.8 Relation binaire entre les services et les pressions artérielles

	PressionArterielle1	PressionArterielle2	PressionArterielle3
ServiceCommunication			
ServiceAuthentification			
ServiceGeoLocalisation	X	X	X
ServiceNotificationPatient			
ServiceAlarme	X	X	X
ServiceCapteur	X	X	X
ServiceSoins	X	X	X
ServiceOptionPrioriteAlerte			
ServiceAvertisseur			
ServiceOptionAlarme	X	X	X
ServiceSignesVitaux	X	X	X
ServiceEnvironnement			

3.4.2 Fonctions de domaine et codomaine

Définition 3.8 Les fonctions de relation dom(r) et cod(r). Soit FCR (K, R), une paire de fonctions mappe les relations dans R aux ensembles d'objets domaine et codomaine à partir de la famille d'ensemble d'objets $G = \{G|K(G, M, I) \in K\}$.

- *Fonction de domaine* est dom : $R \rightarrow G$, noté $dom(r) = G_k$
- *Fonction de codomaine* est cod : $R \rightarrow G$, noté $cod(r) = G_i$

Les relations sont définies sur un ensemble d'objets, c'est-à-dire que pour chaque relation, r est $r \subseteq G_k \, x \, G_i$, où G_k et G_i sont des ensembles d'objets prédéfinis. L'ensemble G_k est l'ensemble d'objets du contexte K_k, appelé *domaine* de la relation r_k, et l'ensemble G_i est l'ensemble d'objets du contexte K_i, appelé *codomaine* de la relation r_k (Rouane-Hacene *et al.*, 2013b).

3.4.3 Échelonnage relationnel

L'échelonnage relationnel permet de définir les attributs relationnels. Un attribut relationnel consiste à établir une relation entre deux concepts. L'approche consiste à étendre le contexte *domaine* d'une relation r par de nouveaux attributs relationnels, notés $q\, r : c$ où q est un opérateur, r est la relation et c est un concept du contexte d'un ensemble d'objets. (Rouane-Hacene *et al.*, 2013a). Un opérateur q parmi $Q = \{\forall, \exists, \forall\exists, \geq, \geq_q, \leq, \leq_q\}$ peut être choisi. Le concept formel dans lequel apparaît l'attribut relationnel est le *domaine* d'une relation et le concept formel vers lequel pointe l'attribut relationnel est le *codomaine* d'une relation (Rouane-Hacene *et al.*, 2013a).

Il existe plusieurs opérateurs d'échelonnages relationnels dans ARC (voir tableau 3.9). La corrélation dans le tableau définit l'attribut relationnel pour un opérateur donné.

Tableau 3.9 Échelonnage relationnel dans ARC (Rouane-Hacene *et al.*, 2013b)

Nom de l'opérateur	Notation	Syntaxe	Corrélation $k(q, r, Ext(c))$
Universel	$S_{(r,\forall),\mathcal{L}}$	$\forall\, r: c$	$r(o) \subseteq Ext(c)$
Existentiel	$S_{(r,\exists),\mathcal{L}}$	$\exists\, r: c$	$r(o) \cap Ext(c) \neq \emptyset$
Universel stricte	$S_{(r,\forall\exists),\mathcal{L}}$	$\forall\exists\, r: c$	$r(o) \subseteq Ext(c), r(o) \neq \emptyset$
Restriction de cardinalité (max)	$S_{(r,\geq),\mathcal{L}}$	$\geq n\, r: \top\mathcal{L}$	$\lvert r(o) \geq n \rvert$
Restriction de cardinalité (min)	$S_{(r,\leq),\mathcal{L}}$	$\leq n\, r: \top\mathcal{L}$	$\lvert r(o) \leq n \rvert$
Restriction de cardinalité qualifiée (max)	$S_{(r,\geq q),\mathcal{L}}$	$\geq n\, r: c$	$r(o) \subseteq Ext(c), \lvert r(o) \rvert \geq n$
Restriction de cardinalité qualifiée (min)	$S_{(r,\leq q),\mathcal{L}}$	$\leq n\, r: c$	$r(o) \subseteq Ext(c), \lvert r(o) \rvert \leq n$

3.4.4 Algorithme de construction des treillis de concepts avec l'analyse relationnelle de concepts

Le processus de construction du treillis de concepts associé avec une FCR est un processus itératif (Rouane-Hacene *et al.*, 2013a, 2013b). Ce processus débute par la construction préliminaire des treillis de concepts. Cette première partie omet les attributs relationnels et se concentre sur les attributs binaires. Elle fournit la base nécessaire pour effectuer l'échelonnage relationnel qui s'applique sur toutes les relations de FCR lors de la prochaine itération. Ensuite, un nouvel ensemble de treillis de concepts est construit. Cette itération se poursuit et se termine lorsque la construction des treillis n'introduit pas de nouveaux concepts formels sur l'ensemble des treillis de concepts.

Chaque contexte $K_i \in K$ d'un FCR produit une séquence $K_i^0 = (G_i^0, M_i^0, I_i^0)$, où le membre zéro est le contexte formel K_i lui-même (Rouane-Hacene *et al.*, 2013a, 2013b). À partir de là, chaque membre subséquent est l'extension relationnelle d'un concept formel précédent. L'extension relationnelle d'un contexte formel consiste à fusionner deux concepts formels qui ont les mêmes objets, mais des attributs différents selon un opérateur d'échelonnage et un ensemble de treillis.

3.4.4.1 Treillis de concepts à l'état initial

Avec notre exemple, nous construisons les treillis de concepts préliminaires pour chaque contexte formel en nous concentrant sur les attributs binaires et en omettant les relations.

Le treillis de concepts pour le contexte de *Localisation*, illustré par la figure 3.9, contient les concepts formels *Concept_ContexteLocalisation_0* (CL0), *Concept_ContexteLocalisation_1* (CL1), *Concept_ContexteLocalisation_2* (CL2) et *Concept_ContexteLocalisation_3* (CL3) dont leurs définitions sont données comme suit (voir Annexe A.1 pour plus de détails):

a) Concept_ContexteLocalisation_0 (CL0) : Les localisations ayant toutes les propriétés.
b) Concept_ContexteLocalisation_1 (CL1) : Les valeurs de localisation appartenant au cluster 1.
c) Concept_ContexteLocalisation_2 (CL2) : Les valeurs de localisation appartenant au cluster 2.
d) Concept_ContexteLocalisation_3 (CL3) : Localisation.

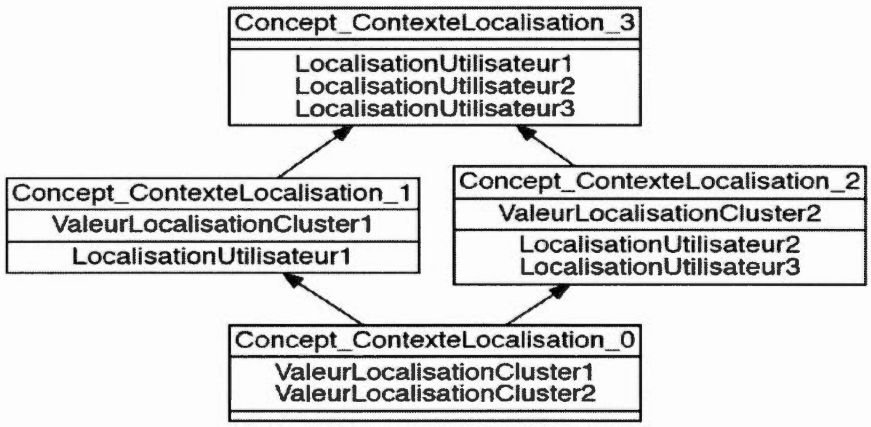


Figure 3.9 Treillis de concepts du contexte de *localisation*

Le treillis de concepts pour le contexte de *température corporelle,* illustré par la figure 3.10, contient les concepts formels *Concept_ContexteTemperatureCorporelle_0* (CT0), *Concept_ContexteTemperatureCorporelle_1* (CT1), *Concept_Concept_ContexteTemperatureCorporelle_2* (CT2), *Concept_ContexteTemperatureCorporelle_3* (CT3), *Concept_ContexteTemperatureCorporelle_4* (CT4) et *Concept_ContexteTemperatureCorporelle_5* (CT5) dont leurs définitions sont données comme suit (voir Annexe A.1 pour plus de détails) :

a) Concept_ContexteTemperatureCorporelle_0 (CT0) : Les températures corporelles ayant toutes les propriétés.
b) Concept_ContexteTemperatureCorporelle_1 (CT1) : Les valeurs de température corporelle ayant une précision moyenne, appartenant au cluster 2 et ayant été captée par le capteur source 2.
c) Concept_ContexteTemperatureCorporelle_2 (CT2) : Les valeurs de température corporelle ayant une précision élevée, appartenant au cluster 3 et ayant été captés par le capteur source 1.

d) Concept_ContexteTemperatureCorporelle_3 (CT3) : Les valeurs de température ayant une précision faible, appartenant au cluster 1 et ayant été captée par le capteur source 1.

e) Concept_ContexteTemperatureCorporelle_4 (CT4) : Les valeurs de température corporelle ayant été captée par le capteur source 1.

f) Concept_ContexteTemperatureCorporelle_5 (CT5) : Température corporelle.

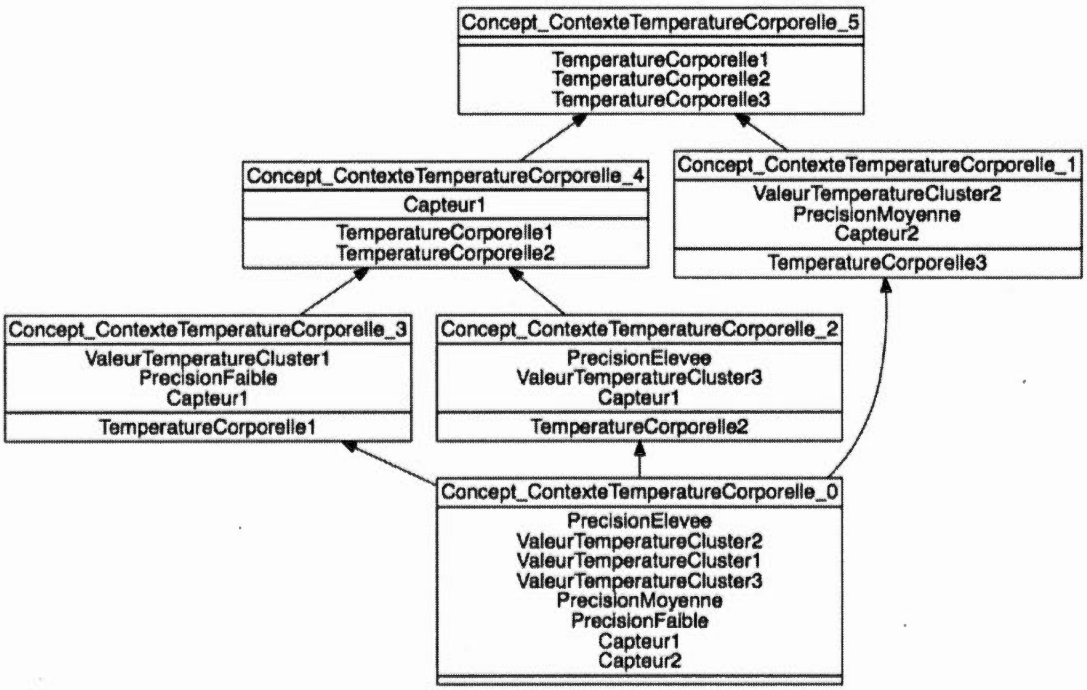


Figure 3.10 Treillis de concepts du contexte de *température corporelle*

Le treillis de concepts pour le contexte de *pression artérielle,* illustré par la figure 3.11, contient les concepts formels *Concept_ContextePressionArterielle_0* (CP0), *Concept_ContextePressionArterielle_1* (CP1), *Concept_ ContextePressionArterielle_2* (CP2), *Concept_ContextePressionArterielle_3* (CP3), *Concept_ContextePressionArterielle_4* (CP4) et *Concept_ContextePressionArterielle _5* (CP5) dont leurs définitions sont données comme suit (voir Annexe A.1 pour plus de détails):

a) Concept_ContextePressionArterielle_0 (CT0) : Les pressions artérielles ayant toutes les propriétés.
b) Concept_ContextePressionArterielle_1 (CT1) : Les valeurs de pression artérielle ayant une précision élevée, appartenant au cluster 3 et ayant été captés par le capteur source 2.
c) Concept_ContextePressionArterielle_2 (CT2) : Les valeurs de pression artérielle ayant une précision moyenne, appartenant au cluster 2 et ayant été captée par le capteur source 2.
d) Concept_ContextePressionArterielle_3 (CT3) : Les valeurs de pression artérielle ayant une précision faible, appartenant au cluster 1 et ayant été captée par le capteur source 1.
e) Concept_ContextePressionArterielle_4 (CT4) : Les valeurs de pression artérielle ayant été captée par le capteur source 2.
f) Concept_ContextePressionArterielle_5 (CT5) : Pression artérielle.

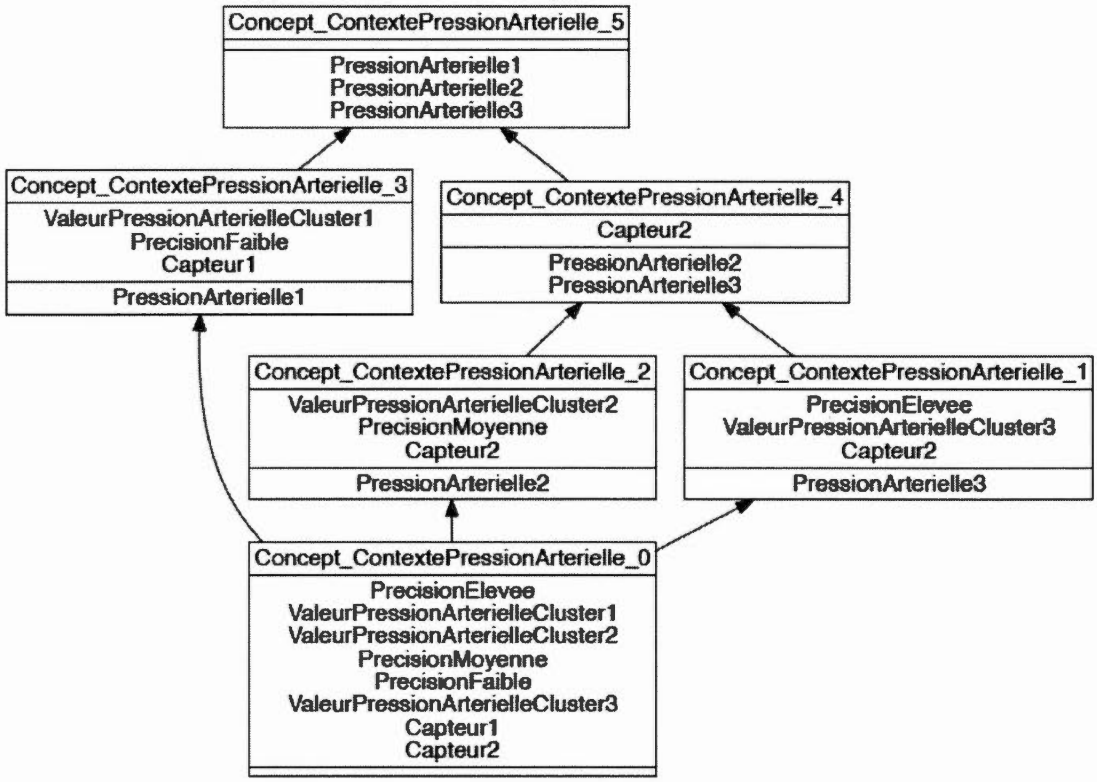


Figure 3.11 Treillis de concepts du contexte de *pression artérielle*

Le treillis de concepts pour le contexte de *Services,* illustré par la figure 3.12, contient les concepts formels *Concept_Services_0* (CS0), *Concept_Services_1* (CS1), *Concept_Services_2* (CS2), *Concept_Services_3* (CS3), *Concept_Services_4* (CS4), *Concept_Services_5* (CS5), *Concept_Services_6* (CS6), *Concept_Services_7* (CS7), *Concept_Services_8* (CS8), *Concept_Services_9* (CS9), *Concept_Services_10* (CS10), *Concept_Services_11* (CS11), *Concept_Services_12* (CS12) et *Concept_Services_13* (CS13) dont leurs définitions sont données comme suit (voir Annexe A.1 pour plus de détails):

a) Concept_Services_0 (CS0) : Les services ayant toutes les propriétés.
b) Concept_Services_1 (CS1) : Les services offrant les fonctionnalités F622Lumières et F621TempAmbiante.

c) Concept_Services_2 (CS2) : Le service offrant les fonctionnalités F611PressionArterielle et F612TempCorporelle.
d) Concept_Services_3 (CS3) : Le service offrant les fonctionnalités F51DetectionChute et F52BoutonPanique.
e) Concept_Services_4 (CS4) : Le service offrant les fonctionnalités F421AvertisseurEcran et F422AvertisseurSonore.
f) Concept_Services_5 (CS5) : Le service offrant les fonctionnalités F412Priorite et F411Fifo.
g) Concept_Services_6 (CS6) : Le service offrant la fonctionnalité F7PlanSoins.
h) Concept_Services_7 (CS7) : Le service offrant les fonctionnalités F6Capteur, F61CapteurPhysiologique et F62CapteurEnvironnement.
i) Concept_Services_8 (CS8) : Le service offrant la fonctionnalité F5Alarme.
j) Concept_Services_9 (CS9) : Le service offrant les fonctionnalités F41PrioriteAlerte, F42Alertes et F4NotificationPatient.
k) Concept_Services_10 (CS10) : Le service offrant les fonctionnalités F3GeoLocalisation, F31GPS et F32Wifi.
l) Concept_Services_11 (CS11) : Le service offrant la fonctionnalité F2Authentification.
m) Concept_Services_12 (CS12) : Le service offrant les fonctionnalités F12Zigbee, F11Wifi et F1Communication.
n) Concept_Services_13 (CS13) : Services.

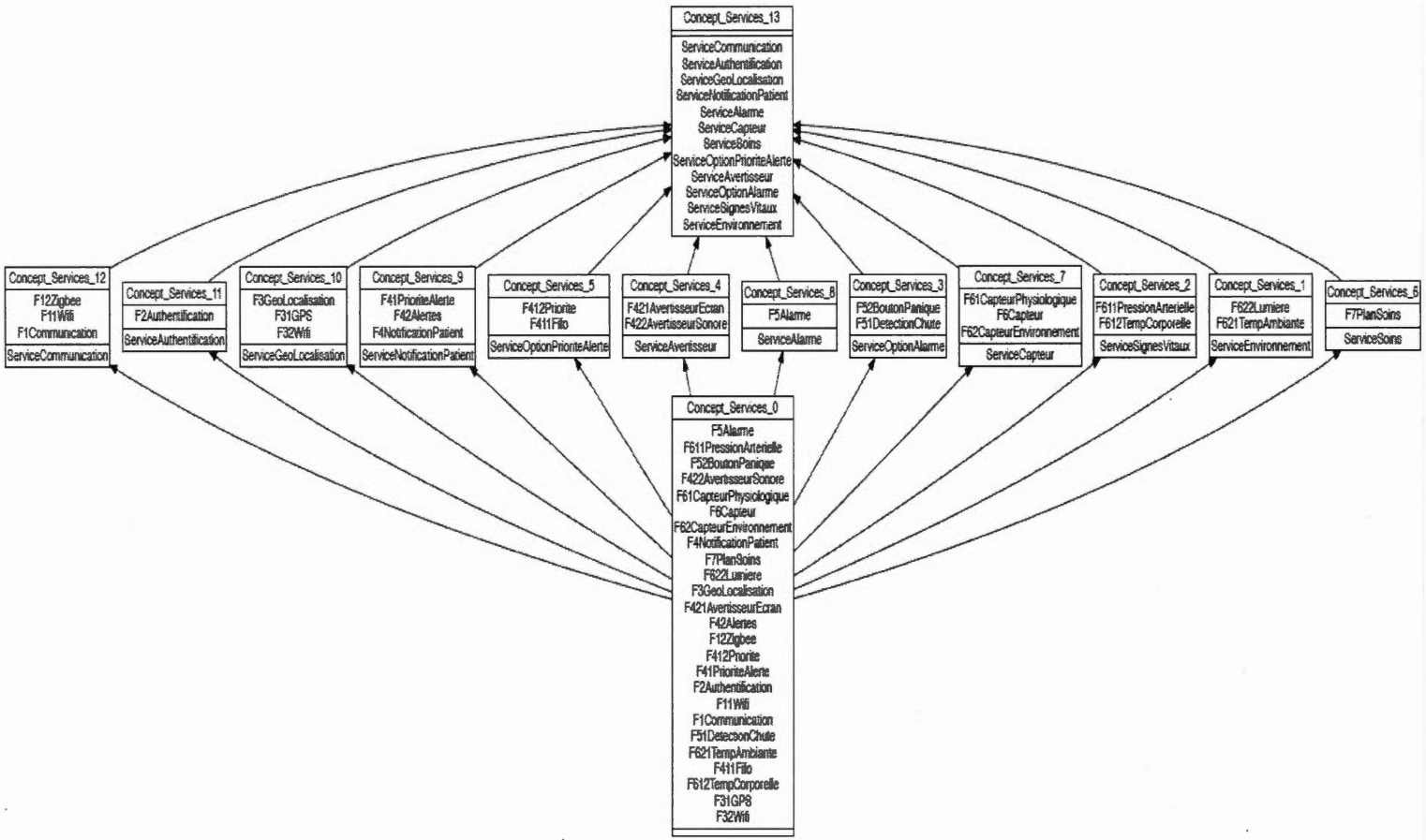


Figure 3.12 Treillis de concepts des *Services*

3.4.4.2 Échelonnage relationnel *existentiel*

Parmi les différents niveaux d'échelonnages illustrés dans le tableau 3.4, nous avons utilisé l'*échelonnage existentiel* où $r(o) \cap Ext(c) \neq \emptyset$ afin de déterminer les attributs relationnels des contextes formels de *localisation*, *température corporelle* et *pression artérielle*.

Les tableaux 3.10 montrent les résultats de l'*échelonnage existentiel* du contexte formel de *service* relatif à *dependLocalisation*, *dependTemperature* et *dependPressionArterielle*.

Tableau 3.10 Échelonnage existentiel sur le contexte formel des *Services*

	F1Communication	F11Wifi	F12Zigbee	F2Authentification	F3GeoLocalisation	F31GPS	F32Wifi	F4NotificationPatient	F41PrioriteAlerte	F411Fifo	F412Priorite	F42Alertes	F421AvertisseurEcran	F422AvertisseurSonore	F5Alarme	F51DetectionChute	F52BoutonPanique	F6Capteur	F61CapteursPhysiologique	F611PressionArterielle	F612TempCorporelle	F62CapteursEnvironnement	F621TempAmbiante	F622Lumiere	F7Soins	dependLocalisation(CL0)	dependLocalisation(CL1)	dependLocalisation(CL2)	dependLocalisation(CL3)	dependTemperature(CT0)	dependTemperature(CT1)	dependTemperature(CT2)	dependTemperature(CT3)	dependTemperature(CT4)	dependTemperature(CT5)	dependPressionArterielle(CP0)	dependPressionArterielle(CP1)	dependPressionArterielle(CP2)	dependPressionArterielle(CP3)	dependPressionArterielle(CP4)	dependPressionArterielle(CP5)
ServiceCommunication	X	X	X																																					
ServiceAuthentification				X																																				
ServiceGeoLocalisation					X	X	X																			X	X	X		X	X	X	X	X		X	X	X	X	X
ServiceNotificationPatient								X	X			X																												
ServiceAlarme															X											X	X	X		X	X	X	X	X		X	X	X	X	X
ServiceCapteur																		X	X			X				X	X	X		X	X	X	X	X		X	X	X	X	X
ServiceSoins																									X	X	X	X		X	X	X	X	X		X	X	X	X	X
ServiceOptionPrioriteAlerte										X	X																													
ServiceAvertisseur													X	X																										
ServiceOptionAlarme																X	X									X	X	X		X	X	X	X	X		X	X	X	X	X
ServiceSignesVitaux																				X	X									X	X	X	X	X		X	X	X	X	X
ServiceEnvironnement																							X	X																

L'*échelonnage existentiel* du contexte formel des *services* sur *dependLocalisation* et *dependTemperature* permet d'ajouter et de modifier des concepts formels (voir figure 3.13), plus précisément *Concept_Services_0* (CS0), *Concept_Services_2* (CS2), *Concept_Services_3* (CS3), *Concept_Services_6* (CS6), *Concept_Services_7* (CS7), *Concept_Services_8* (CS8), *Concept_Services_10* (CS10), *Concept_Services_14* (CS14) et *Concept_Services_15* (CS15). Nous notons que la définition des concepts formels qui n'ont pas eu de modification pour le treillis de concepts des services est décrite dans la section 3.4.4.1 et nous présentons ces concepts formels comme suit (voir Annexe A.2 pour plus de détails):

a) Concept_Service_0 (CS0) : Les services offrant toutes les fonctionnalités.
b) Concept_Service_2 (CS2) : Le service offrant les fonctionnalités F611PressionArterielle et F612TempCorporelle. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.
c) Concept_Service_3 (CS3) : Le service offrant les fonctionnalités F52BoutonPanique et F51DetectionChute. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la

pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

d) Concept_Service_6 (CS6) : Le service offrant la fonctionnalité de F7PlanSoins. Ce service dépend de la pression artérielle selon 4 scénarios: 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

e) Concept_Service_7 (CS7) : Le service offrant les fonctionnalités F6Capteurm F62CapteurEnvironnement et F51DetectionChute. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle

appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

f) Concept_Service_8 (CS8) : Le service offrant la fonctionnalité F5Alarme. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

g) Concept_Service_10 (CS10) : Le service offrant les fonctionnalités F3GeoLocalisation, F31GPS et F32Wifi. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

h) Concept_Service_14 (CS14) : Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

i) Concept_Service_15 (CS15) : Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

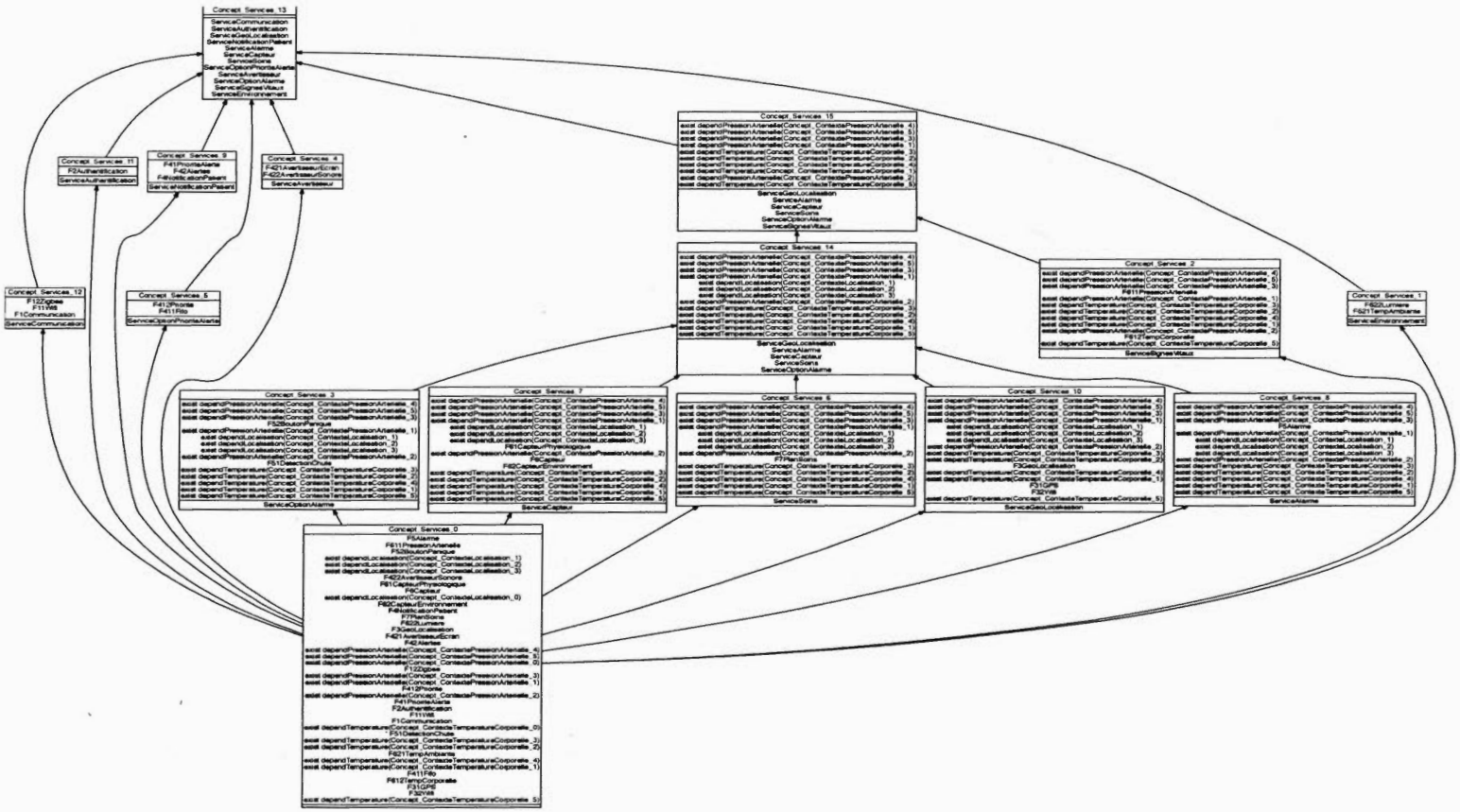


Figure 3.13 Treillis de concepts des services après échelonnage existentiel

Vu que l'échelonnage relationnel du contexte formel des services relatifs a *dependLocalisation, dependTemperatureCorporelle* et *dependPressionArterielle* n'ajoute rien de nouveau au treillis de concepts, l'algorithme se termine.

Ces nouveaux treillis de concepts de services sont donc plus riches et élaborés que le treillis initial. Ce processus nous permet entre autres d'établir une base de connaissances en logique descriptive. Nous présenterons la correspondance entre les entités de l'analyse relationnelle de concepts et la logique descriptive dans le chapitre 4.

3.5 Comparaison des approches

Par rapport aux techniques de modélisation de contexte existant dans la littérature, notre modèle présente des avantages tels que résumés dans le tableau 3.11.

Elle se base sur les aspects suivants :

- *Regroupement conceptuel*: la capacité de la méthode de regrouper et expliquer les données.
- *Découverte de règles*: la découverte de dépendances entre les services et les contextes.
- *Adaptabilité*: la facilité de s'adapter aux nouvelles situations.
- *Facilité d'utilisation*: la facilité à apprendre et d'utiliser la méthode.
- *Composition distribuée*: la possibilité de modéliser le contexte dans un environnement distribué (Strang et Linnhoff-Popien, 2004).
- *Validation partielle*: le potentiel de valider - partiellement - les connaissances contextuelles selon l'approche de modélisation de contexte choisie. (Strang et Linnhoff-Popien, 2004).
- *Richesse et qualité de l'information*: la qualité et la richesse de l'information contextuelle fournie par les capteurs (Strang et Linnhoff-Popien, 2004).

- *Niveau de formalisme*: partager la compréhension ou l'interprétation des données échangées (Strang et Linnhoff-Popien, 2004).
- *Applicabilité aux environnements existants*: la capacité d'appliquer le modèle dans des environnements informatiques ubiquitaires (Strang et Linnhoff-Popien, 2004).

Tableau 3.11 Comparaison des approches de modélisation de contexte

Caractéristiques	Approches existantes de modélisation de contexte						Approche proposée
	Attribut-valeur	***Langage de marquage***	***Graphique***	***Objet***	***Logique***	***Ontologie***	***Analyse relationnelle de concepts***
Regroupement conceptuel	Non applicable	Non applicable	Non applicable	Capacité de grouper un ensemble d'objets.	Peut se baser sur des associations de patrons dans les données.	Groupe des concepts similaires ou reliés sous forme de cluster	Application d'approches de *clustering* telles que les mesures de similarité des contextes formels.
Découverte de règle	Non applicable	Non applicable	Non applicable	Non applicable	Possible en utilisant des règles d'association.	Possible en utilisant des règles d'association.	Possible par inférence d'implication, de dépendance fonctionnelle, extraction de règles d'association et d'induction de règles de classification.
Adaptabilité	Facile à gérer si la taille est petite.	Complexe lorsque plusieurs niveaux d'information sont impliqués.	Complexe lorsque plusieurs niveaux d'information sont impliqués.	Permet la manipulation de contexte en temps réel.	Adapte à des situations changeantes.	Permet de dériver des situations.	Permet de dériver des situations.
Rentabilité	Non mise à l'échelle.	Difficile de récupérer de l'information.	Configuration peu être requise. Interopérabilité parmi différentes implémentations est difficile.	Difficile de récupérer de l'information.	Difficile à maintenir.	Problème de performance. Récupérer l'information est complexe et requiert des ressources intensives.	Implémentation efficace de différents algorithmes de constructions de treillis de concepts.
Facilité à utiliser	Simple et flexible. Facile	Flexible et plus structuré.	Plus facile de récupérer des informations.	Réduis la maintenance. Améliore la	Simple à modéliser et utiliser.	Représentation peut être complexe.	Facile à utiliser.

Caractéristiques	Approches existantes de modélisation de contexte						Approche proposée
	Attribut-valeur	*Langage de marquage*	*Graphique*	*Objet*	*Logique*	*Ontologie*	*Analyse relationnelle de concepts*
	à gérer. Modèle limité pour des données de petite taille.			fiabilité et la réutilisation.			
Composition distribuée	Limité	Fonctionnel	Non disponible	Flexible.	Flexible.	Flexible.	Flexible.
Validation partielle	Aucune validation supportée.	Réalisable avec des schémas de validation.	Réalisable avec des contraintes.	Manque de validation.	Manque de validation et de standards.	Validation forte et supportée par des standards.	Validation forte et supportée par des standards.
Richesse et qualité de l'information	Aucune façon de représenter les liens.	Schéma permet de gérer la richesse de l'information. Difficile de récupérer des informations.	Possible de modéliser des liens.	Contextes complexes peuvent être modélisé. Difficile de récupérer des informations.	Contextes complexes peuvent être modélisé. Supporte le raisonnement logique.	Permets la représentation expressive et supporte le raisonnement sur le contexte.	Permets la représentation expressive et supporte le raisonnement sur le contexte.
Incomplétude et ambigüité	Incomplétude gérée un niveau d'instance. Aucune considération pour des ambigüités.	Géré par l'application, mais devient complexe avec différent niveau d'information.	Aucun mécanisme pour gérer l'incomplétude et l'ambigüité.	Qualité de l'information aide à gérer l'incomplétude et l'ambigüité.	Incomplétude gérée par des règles. Ambigüité n'est pas adressée.	Possible de gérer l'incomplétude et l'ambigüité avec des mécanismes d'inférence.	Possible de gérer l'incomplétude et l'ambigüité.
Niveau d'expressivité	Aucune formalité ou structure.	Plus structuré et flexible.	Différents standards et implémentations disponibles qui sont gérés par des principes.	Aucun standard et géré par des principes. Invisibilité par l'encapsulation.	Langage de format abstrait.	Langage de format abstrait.	Langage de format abstrait.
Applicabilité aux environnements existants	Aucun outil standard.	Dépendant de l'application. Aucun standard.	Peut être utilisé pour archiver des volumes larges et à long terme.	Langage de programmation bien intégré.	Outils disponibles, mais couplés avec l'application.	Différents outils disponibles et indépendants de l'application.	Un seul outil disponible.

En comparant notre approche avec les modèles de contexte existants, nous dégageons certains avantages. D'abord, le modèle de contexte que nous proposons permet d'établir des liens et des dépendances entre les contextes et les fonctionnalités contextuelles d'une application sensible au contexte. Non seulement les liens et les dépendances sont pris en considération, mais les concepts formels des treillis

permettent de grouper conceptuellement les informations contextuelles. Nous supportons également le raisonnement sur le contexte à partir de ce modèle et elle ne dépend pas de l'approche choisie pour modéliser le contexte. Les règles de correspondance existante entre les entités de l'analyse relationnelle de concepts et la logique descriptive permettent de valider le modèle de contexte. Cette partie de la thèse est détaillée dans le chapitre 4. De plus, nous avons choisi une méthode formelle pour modéliser le contexte. Le modèle d'ontologie est le modèle le plus populaire dans la littérature et nous notons qu'il n'y a pas eu de nouvelles approches proposées qui respectent un ou plusieurs critères présentés dans ce tableau. Notre modèle de contexte offre des possibilités de découvrir des règles de dépendance alors que le modèle d'ontologie est limité sur cet aspect. Les autres critères sont assez similaires entre ces deux méthodes. Toutefois, la construction du modèle de contexte à partir de l'analyse relationnelle de concepts est beaucoup plus simple à visualiser et à analyser que le modèle d'ontologie.

3.6 Conclusion

Dans ce chapitre, nous avons présenté notre méthode de modélisation du contexte en utilisant l'analyse relationnelle de concepts. Vu que notre modèle de contexte se base sur une extension de l'analyse formelle de concepts, nous avons fait un rappel des notions de base de cette méthode d'analyse de données. De plus, nous avons présenté une méthode d'échelonnage permettant la transformation des données multi-valués en mono-valués dans le but d'obtenir des contextes formels et de procéder à l'étape de construction de treillis. Nous avons également fait un rappel sur l'analyse relationnelle de concepts et nous avons expliqué comment nous avons proposé de modéliser le contexte depuis cette méthode. Nous avons terminé ce chapitre par une comparaison de notre modèle de contexte avec les modèles de contexte que nous avons présenté dans le chapitre précédent.

CHAPITRE IV

RAISONNEMENT SUR LE CONTEXTE

4.1 Introduction

Les applications sensibles au contexte nécessitent d'avoir des capacités de raisonnement afin de déterminer si l'adaptation d'un système est requise en réponse aux changements du contexte. Par conséquent, la vérification de la cohérence d'un modèle de contexte et des techniques de raisonnement deviennent importantes et doivent être soutenues par les techniques de modélisation de contexte.

Dans le chapitre précédent, nous avons proposé de modéliser le contexte depuis l'analyse relationnelle de contexte. Dans ce chapitre, nous voulons effectuer le raisonnement sur le contexte. À partir des treillis de concepts que nous avons présentés dans le chapitre précédent, nous produisons une ontologie à laquelle nous faisons correspondre une logique descriptive dans le but de :

1. Valider formellement le modèle de contexte produit précédemment.
2. Établir une relation entre les contextes et les services qui en dépendent.
3. Établir les bases pour déterminer les dépendances entre les contextes.

Étant donné qu'un treillis de concepts peut être considéré comme une ontologie, nous avons construit une ontologie du contexte à partir des règles de correspondance entre les entités de l'analyse relationnelle de concept et la logique descriptive.

Ce chapitre est divisé comme suit. Nous présentons notre approche dans la section 4.2. Dans la section 4.3, nous faisons un rappel de la logique descriptive. Nous présentons les règles de correspondance entre l'analyse relationnelle de concepts et la logique descriptive dans la section 4.4. Ensuite, nous montrons les opérations de raisonnement que nous pouvons effectuer à partir du modèle de contexte dans la section 4.5. Enfin, la section 4.6 présente une comparaison des diverses méthodes de raisonnement sur le contexte et nous concluons le chapitre dans la section 4.7.

4.2 Approche choisie

Nous proposons d'utiliser une approche de raisonnement sur le contexte qui fonctionne conjointement avec notre méthode de modélisation du contexte, c'est-à-dire l'analyse relationnelle de concepts (Amja, Obaid et Valtchev, 2014). Pour ce faire, nous utilisons les règles de correspondance entre les entités de l'analyse relationnelle de concepts et la logique descriptive pour obtenir une base de connaissances. Ces expressions produites par l'analyse relationnelle de concepts sont représentées en expressions d'une logique descriptive puis implémentées en ontologie OWL.

Les ontologies sont des spécifications explicites et formelles d'une conceptualisation partagée (Gruber, 1993). « Conceptualisation » réfère à une abstraction d'un phénomène du monde obtenue en identifiant les concepts de ce phénomène. «Explicite » signifie que le modèle est décrit dans un langage formel et défini. « Formelle » indique que l'ontologie est interprétable par une machine. « Partagée » reflète la notion qu'une ontologie capture des connaissances consensuelles. OWL est un langage du web sémantique qui permet d'implémenter les ontologies. Ce langage se base sur la logique descriptive. Par conséquent, une ontologie OWL équivaut à une base de connaissances de la logique descriptive.

À partir de la logique descriptive (ou d'une ontologie OWL de contexte), nous raisonnons sur le contexte afin de :

- *Identifier l'incohérence potentielle de l'information contextuelle* : L'objectif est de vérifier la cohérence du modèle de contexte en vérifiant l'ensemble des concepts et leurs liens.
- *Déterminer la présence de contexte de haut niveau d'abstraction* : L'objectif est de définir les différents contextes d'une application sensible au contexte basé sur un ensemble particulier d'instances de données contextuelles et de leurs liens.

Nous pouvons également effectuer des opérations de raisonnement associées à cette ontologie de contexte, à savoir :

- *La subsomption* : Le but est de déterminer si un contexte contextuel est un sous-concept d'un autre concept. En d'autres termes, nous vérifions les instances de contexte en fonction des relations hiérarchiques des concepts dans le modèle de contexte.
- *L'instanciation de concepts* : L'objectif est de retrouver un concept (ou un contexte abstrait de haut niveau) dont un objet est une instance.
- *La comparaison de concepts* : Le but est de déterminer si deux instances appartiennent au même concept contextuel.
- *La détection du domaine ou du co-domaine d'une relation*: L'objectif est de déterminer le concept contextuel d'un objet (une instance de contexte) sachant le domaine ou le co-domaine d'une relation binaire (une dépendance entre service et contexte).

Dans la section 4.4.3, nous montrons un exemple pour chacune de ces opérations de raisonnement. Nous jugeons qu'il est plus facile de comprendre le raisonnement sur le contexte une fois que nous avons montré la transformation du modèle de contexte en logique descriptive (ou ontologie OWL). La figure 4.1 illustre l'approche choisie.

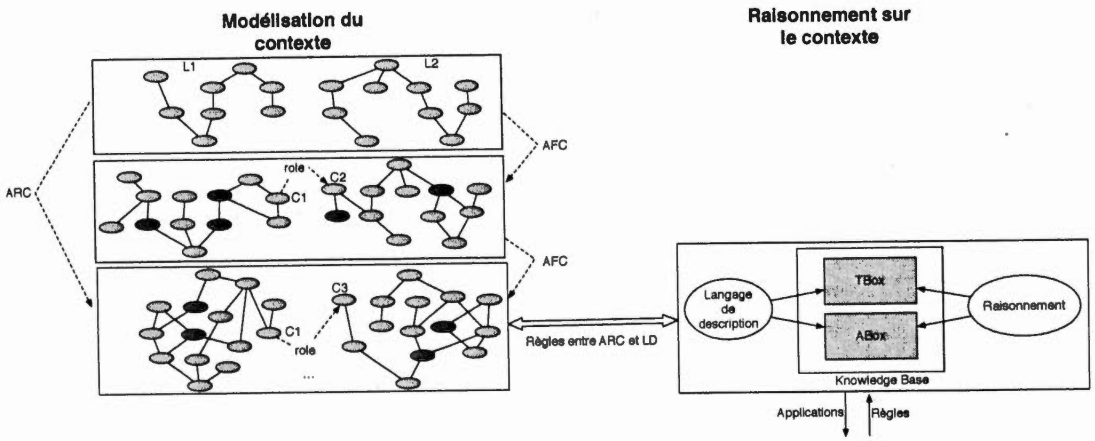


Figure 4.1 L'approche proposée : modélisation du contexte et de raisonnement sur le contexte

4.3 Rappel sur la logique descriptive

La logique descriptive est une famille de langages de représentation des connaissances d'un domaine d'application de façon structurée et formelle qui se base sur la logique (Baader *et al.*, 2003). Elle met l'accent sur un raisonnement efficace pour la prise de décision.

La logique descriptive répartit la connaissance en deux parties :

- *les informations terminologiques* (*TBox*) : définissent les notions de base, plus précisément les *concepts* et les *rôles*, et établissent des liens entre eux (Baader *et al.*, 2003).
- *les informations sur les individus* (*ABox*) : définissent les *assertions* où interviennent les concepts et les instances de concepts (Baader *et al.*, 2003).

Une base de connaissance K est une paire (T, A) où T est un ensemble d'axiomes *terminologiques* (TBox) et A est un ensemble d'axiomes *assertionnels* (ABox).

4.3.1 TBox

Le TBox décrit les connaissances générales d'un domaine. Les éléments du monde réel sont représentés par des *concepts*, des *rôles* et des *individus*. Un *concept* représente un ensemble d'individus alors que les *rôles* représentent une relation binaire entre *individus*. Un *concept* correspond à une entité générique d'un domaine et un *individu* à une instance d'un *concept*.

Les concepts et les rôles peuvent être *primitifs* ou *définis (composés)*. Les concepts et rôles primitifs sont comparables à des axiomes et constituent les entités élémentaires d'un TBox. Par exemple, *Femelle*, *Male*, *Homme* et *Femme* sont des concepts primitifs et *aEnfant* est un rôle primitif. Les concepts et rôles primitifs voient à ce que la construction de concepts et rôles composés se définisse au moyen de *constructeur*. Les *constructeurs* permettent la combinaison de concepts et rôles primitifs pour former des entités composées. Par exemple, le concept composé $Male \sqcap Femelle$ résulte de l'utilisation du constructeur $\sqcap$ aux concepts primitifs $Male$ et $Femelle$.

La logique descriptive prédéfinit 4 concepts et rôles primitifs minimaux: le concept $\top$ et le rôle $\top_R$, les plus généraux de leur catégorie respective, et le concept $\bot$ et le rôle $\bot_R$, les plus spécifiques (Baader *et al.*, 2003).

Un TBox contient des axiomes terminologiques de la forme $C \sqsubseteq D$ $(R \sqsubseteq S)$ ou $C \equiv D$ $(R \equiv S)$, où C, D sont des concepts et R, S sont des rôles. La première forme énonce des *relations d'inclusion* et la seconde forme exprime des *relations d'équivalence*.

La description de concepts et de rôles suit les règles de syntaxe comme suit (voir tableau 4.1) :

Tableau 4.1 Syntaxe d'une logique descriptive

Syntaxe	Constructeur de concepts
$C \sqsubseteq A$	Subsomption de concepts
$\bot$	Concept vide
$\top$	Concept universel
$C \sqcap D$	Conjonction de concepts
$C \sqcup D$	Disjonction de concepts
$\neg C$	Négation de concepts
$\forall r.C$	Restriction universelle
$\exists r.C$	Restriction existentielle
$(\geq n\, r)$	Restriction de cardinalité supérieure
$(\leq n\, r)$	Restriction de cardinalité inférieure
$r \sqsubseteq a$	Subsomption de rôles
$r \wedge s$	Composition de rôles

Une sémantique est associée aux descriptions de concepts et de rôles en se penchant sur la notion *d'interprétation.*

Définition 4.1 Interprétation. Une interprétation $I = (\Delta_I, .^I)$ est la donnée d'un ensemble Δ_I appelé *domaine de l'interprétation,* c'est-à-dire un ensemble non vide représentant les entités du monde décrit et composé d'individus, et d'une *fonction d'interprétation* $.^I$ qui fait correspondre à un concept un sous-ensemble de Δ_I et à un rôle un sous-ensemble de $\Delta_I \times \Delta_I$, de telle sorte que les définitions suivantes soient satisfaites (Baader *et al.*, 2003) (voir tableau 4.2):

Tableau 4.2 Sémantique des expressions en logique descriptive

Constructeur	Sémantique
$\top$	Δ^I
$\bot$	$\emptyset$
$\neg A$	$\Delta^I \backslash A^I$
$C \sqcap D$	$C^I \cap D^I$
$C \sqcup D$	$C^I \cup D^I$
$\forall R.C$	$\{a \in \Delta^I \mid \forall b.(a,b) \in R^I \rightarrow b \in C^I\}$
$\exists R.C$	$\{a \in \Delta^I \mid \forall b.(a,b) \in R^I \wedge b \in C^I\}$
$\geq n\, R$	$\{a \in \Delta^I \mid \lvert\{b \mid (a,b) \in R^I\}\rvert \geq n\}$
$\leq n\, R$	$\{a \in \Delta^I \mid \lvert\{b \mid (a,b) \in R^I\}\rvert \leq n\}$

L'interprétation I satisfait un *axiome d'équivalence* $C \equiv D$ si et seulement si $C^I = D^I$ (Baader *et al.*, 2003). L'interprétation I satisfait un *axiome d'inclusion* $C \sqsubseteq D$ si et seulement si $C^I \subseteq D^I$ (Baader *et al.*, 2003). L'interprétation I satisfait un *TBox* si et seulement si I satisfait tous les axiomes du TBox (Baader *et al.*, 2003). Le tableau 4.3 décrit un exemple de TBox.

Tableau 4.3 Exemple TBox (Baader *et al.*, 2003)

TBox :

$$Femme \equiv Personne \sqcap Femelle$$
$$Homme \equiv Personne \sqcap \neg Femmelle$$
$$Mere \equiv Femme \sqcap \exists aEnfant.Personne$$
$$Pere \equiv Homme \sqcap \exists aEnfant.Personne$$
$$Parent \equiv Pere \sqcup Mere$$
$$GrandMere \equiv Mere \sqcap \exists aEnfant.Parent$$
$$MereAvecPlusieursEnfants \equiv Mere \sqcap \geq 3\ aEnfant$$
$$MereSansFille \equiv Mere \sqcap \forall aEnfant.\neg Femme$$
$$Epouse \equiv Femme \sqcap \exists aCommeMari.Homme$$

4.3.2 ABox

Le ABox décrit un état spécifique du monde réel en introduisant les individus ainsi que les assertions de propriétés de ces individus (Baader *et al.*, 2003). Nous dénotons les individus par les noms a, b et c. Soient le concept C et le rôle R, les assertions sont représentées par les deux formes suivantes : $C(a)$ et $R(b, c)$. La première forme, appelée assertion de concept, indique que a appartient à C alors que la seconde forme, appelée assertion de rôle, indique que les individus b et c sont en relation par l'intermédiaire de r.

Une sémantique est associée non seulement aux concepts et rôles, mais également aux individus. Cette sémantique repose sur *l'hypothèse d'un monde ouvert*, c'est-à-dire que l'absence d'information représente l'ignorance plutôt qu'une information négative (Baader *et al.*, 2003).

Une fonction d'interprétation associe à chaque nom d'individu nommé a, un individu a^I tel que $a^I \in \Delta^I$. La logique descriptive fait souvent référence à l'*hypothèse de noms uniques* : pour tout individu nommé a et b, $a^I \neq b^I$ (Baader *et al.*, 2003).

L'interprétation d'une assertion de concept s'explique comme suit : Soit une assertion de concept, noté $C(a)$, déclarant qu'il existe un individu a membre du concept C, une interprétation I satisfait $C(a)$ si et seulement si $a^I \in C^I$ (Baader *et al.*, 2003).

L'interprétation d'une assertion de rôle s'explique comme suit : Soit une assertion de rôle $R(a, b)$ déclarant qu'il existe un individu a en relation avec un individu b par le rôle R, tel qu'a fait partie du domaine de R et b fait partie de l'image de R, une interprétation I satisfait $R(a, b)$ si et seulement si $(a^I, b^I) \in R^I$ (Baader *et al.*, 2003).

L'interprétation I satisfait un *ABox* si et seulement si I satisfait toutes les assertions du ABox (Baader *et al.*, 2003).

Vu que l'interprétation des connaissances respecte *l'hypothèse du monde ouvert*, un système conçu sur la logique descriptive doit considérer plusieurs interprétations possibles.

Le tableau 4.4 décrit un exemple d'ABox.

Tableau 4.4 Exemple d'ABox (Baader *et al.*, 2003)

ABox :
$Femme(Alice)$
$MereSansFille(Marie)$
$Pere(Pierre)$
$aEnfant(Marie, Pierre)$
$aEnfant(Marie, Paul)$
$aEnfant(Pierre, Alice)$

4.3.3 Inférence

En logique descriptive, l'inférence s'effectue au niveau terminologique (TBox) ou assertionnel (ABox).

Baader *et al.* (2003) présentent quatre principaux problèmes d'inférence au niveau terminologique (TBox) T:

- *Satisfiabilité* : Un concept C d'une terminologie T est *satisfiable* s'il existe un modèle d'interprétation I de T tel que $C^I \neq \emptyset$. Un concept est donc satisfiable s'il existe au moins une entité du monde décrit qui peut appartenir à l'ensemble décrit par ce concept.
- *Subsomption* : Un concept C est *subsumé* par un concept D d'une terminologie T si $C^I \sqsubseteq D^I$ pour tout modèle d'interprétation I de T.
- *Équivalence* : Un concept C est *équivalent* à un concept D d'une terminologie T si $C^I \equiv D^I$ pour tout modèle d'interprétation I de T.

- *Disjonction* : Deux concepts C et D d'une terminologie T sont *disjoints* si et seulement si $C^I \cap D^I = \emptyset$ pour tout modèle d'interprétation I de T.

En principe, résoudre des problèmes d'inférence au niveau terminologique (TBox) consiste à prouver une de ces 4 propriétés. Par contre, il n'est pas nécessaire de prouver chacune de ces priorités, car réduire des problèmes d'inférence d'un TBox se ramène à des problèmes de subsomption et de satisfiabilité (Baader *et al.*, 2003). .

a) *Réduction de problème d'inférence à des problèmes de subsomption*
   - C est insatisfiable $\Leftrightarrow$ C est subsumé par $\bot$.
   - C et D sont équivalents $\Leftrightarrow$ C est subsumé par D, et D par C.
   - C et D sont disjoints $\Leftrightarrow$ $C \sqcap D$ est subsumé par $\bot$.

b) *Réduction de problème d'inférence à des problèmes de satisfiabilité (satisfaisabilté)*
   - C est subsumé par D $\Leftrightarrow$ $C \sqcap \neg D$ est insatisfiable.
   - C et D sont équivalents $\Leftrightarrow$ $C \sqcap \neg D$ et $\neg C \sqcap D$ sont non satisfaisables.
   - C et D sont disjoints $\Leftrightarrow$ $C \sqcap D$ est non satisfaisables.

Baader *et al.* (2003) présentent les principales tâches de raisonnement associé au niveau assertionnel (ABox) A:

- *Cohérence* : Un ABox A est cohérent par rapport à un TBox T si et seulement s'il existe un modèle d'interprétation I de A et T.
- *Vérification d'instances* : consiste à vérifier par inférence si une assertion de concept $C(a)$ est vraie pour tout modèle d'interprétation I d'un ABox A et d'un TBox T.
- *Vérification de rôle* : consiste à vérifier par inférence si une assertion de rôle $R(a, b)$ est vraie pour tout modèle d'interprétation I d'un ABox A et d'un TBox T.

- *Récupération d'individus*: Soit un ABox A, un concept C d'une terminologie T, inférer les individus $a_1^I, \ldots, a_n^I \in C^I$ pour tout modèle d'interprétation I de T.

4.4 Règles de correspondances entre l'analyse relationnelle de concepts et la logique descriptive

Parmi les langages de la famille de logique descriptive, le langage de la logique descriptive $\mathcal{FL} - \mathcal{E}$ inclut les constructeurs T, $\bot$, $C \sqcap D$ (conjonction de concepts), $\forall r.C$ (restriction universelle) et $\exists r.C$ (restriction existentielle). Cet ensemble de constructeurs permet de représenter les entités de l'analyse relationnelle de concepts en une base de connaissances.

Dans les travaux de Bendaoud et Toussaint (2010), Bendaoud *et al.* (2007, 2008) et Rouane *et al.* (2007), ces derniers proposent des règles de correspondance entre les éléments de l'analyse relationnelle de concepts et la logique descriptive tout en conservant la sémantique des concepts formels des treillis.

Le TBox tient la traduction des symboles de contextes, d'attributs, d'attributs relationnels et de concepts. Les règles respectives du TBox sont les suivantes:

- Les attributs et les contextes deviennent des concepts primitifs.
- Les attributs relationnels deviennent des rôles.
- Les concepts formels deviennent des concepts définis qui reflètent l'intension des concepts formels.
- Les liens de sous-concepts des treillis finaux sont représentés par des inclusions entre les concepts définis.

Le tableau 4.5 montre les règles de correspondance entre les éléments de l'analyse relationnelle de concepts et de la logique descriptive du niveau terminologique (TBox) de la base de connaissances.

Tableau 4.5 TBox: Règles de correspondance entre ARC et LD (Hacene Rouane *et al.*, 2007)

Entité source de l'ARC	Notation	Élément ciblé de la LD	Règle
Contexte	$K \in \mathbf{K}$	Concept primitif	$\alpha[K] \in T_C$
Attribut atomique	$a \in A_i$	Concept primitif	$\alpha[a] \in T_C$
Relation	$r \in \boldsymbol{R}$	Rôle primitif	$\alpha[r] \in T_R$
Attribut relationnel (universel)	$(r:c) \in A_i^{\infty}$	Restriction universel	$\forall\alpha[r].\alpha[c] \in T$
Attribut relationnel (existentiel)	$(r:c) \in A_i^{\infty}$	Restriction existentielle	$\exists\alpha[r].\alpha[c] \in T$
Concept	$c \in \mathcal{L}_i^{\infty}, c \neq \top_{\mathcal{L}_i^{\infty}}$	Concept défini	$\alpha[a] \in definitions(T)$
		Concept défini	$(\alpha[c] \equiv \sqcap_{a_j \in int(c)} \alpha[a_j]) \in T$
		Axiome d'inclusion	$(\alpha[c] \sqsubseteq \alpha[K_i]) \in T$
Lien sous-concept	$c_1 \leq_{\mathcal{L}_i^{\infty}} c_2$	Axiome d'inclusion	$(\alpha[c_1] \sqsubseteq \alpha[c_2]) \in T$

Le ABox implique la traduction des objets. Les règles respectives du ABox sont les suivantes :

- Les objets deviennent des individus.
- Les liens relationnels entre les objets deviennent des instances de rôles.
- Les extensions de concepts deviennent des instances de concepts définis.

Le tableau 4.6 montre les règles de correspondance entre les éléments de l'analyse relationnelle de concepts et de la logique descriptive du niveau assertionnel (ABox) de la base de connaissances.

Tableau 4.6 ABox: Règles de correspondance entre ARC et LD (Hacene Rouane *et al.*, 2007)

Entité source de l'ARC	Notation	Élément ciblé de la LD	Règle
objets	$o \in O_i$	Individu	$\alpha[o] \in individu(A)$
		Instanciation de concept primitif	$\alpha[K_i](\alpha[o]) \in A$
Incidence objet à attribut	$(o, a) \in I_j^0$	Instanciation de concept primitif	$\alpha[a](\alpha[o]) \in A$
Lien relationnel	$r(o_1, o_2), r_i \in \boldsymbol{R}$	Instanciation de rôle	$\alpha[r](\alpha[o_1], \alpha[o_2]) \in A$
Incidence objet à concept	$c \in \mathcal{L}_i^{\infty}, o \in ext(c)$	Instanciation de concept défini	$\alpha[c](\alpha[o]) \in A$

4.5 Raisonnement sur le contexte avec la logique descriptive

4.5.4 Base de connaissances

À partir de la correspondance entre les éléments de l'analyse relationnelle de concepts et la logique descriptive décrite dans la section 4.4, nous transformons les concepts formels des treillis finaux de l'exemple présenté dans la section 3.4.4 en base de connaissances dans le but de valider la consistance du modèle de contexte et de déterminer des contextes d'abstraction de haut niveau.

Nous présentons d'abord les éléments reliés au niveau terminologique (TBox). Le tableau 4.7 montre un extrait des concepts et des rôles primitifs (voir Annexe B.1

pour plus de détails). Un attribut d'un contexte formel est traduit en concept primitif dans le TBox. Par exemple, les attributs *ValeurLocalisationCluster1* et *ValeurLocalisationCluster2* du contexte formel de *Localisation* sont des concepts primitifs. Les rôles *dependLocalisation*, *dependTempCorporelle* et *dependPressionArterielle* sont traduits en rôles primitifs

Tableau 4.7 TBox : Concepts primitifs et rôles

Concepts primitifs et rôles
ValeurLocalisationCluster1, ValeurLocalisationCluster2, TempPrecisionFaible, TempPrecisionMoyenne, TempPrecisionElevee, ValeurTemperatureCluster1, ValeurTemperatureCluster2, TempCapteur1, TempCapteur2, F1Communication, F11Wifi, F12Zigbee, F2Authentification, F3GeoLocalisation, F31GPS, F32Wifi, dependLocalisation, dependTempCorporelle, dependPressionArterielle

Le tableau 4.8 montre quelques concepts définis du TBox (voir Annexe B.2 pour plus de détails). Les concepts formels des treillis de concepts équivalent à des concepts définis. Par exemple, le concept formel *Concept_Services_2* (CS2) du treillis de concepts *Services* est représenté comme un concept défini :

$$CS2 \equiv dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap \exists F611PressionArterielle.\top \sqcap dependPressionArterielle.CP1 \sqcap \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap dependPressionArterielle.CP2 \sqcap \exists F612TempCorporelle.\top \sqcap \exists dependTemperatureCorporelle.CT5$$

Tableau 4.8 TBox : Concept défini

ID du treillis	Nom du concept	Concept défini
Concept_Services_2 (CS2)	CS2	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap \exists F611PressionArterielle.\top \sqcap dependPressionArterielle.CP1 \sqcap \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap dependPressionArterielle.CP2 \sqcap \exists F612TempCorporelle.\top \sqcap \exists dependTemperatureCorporelle.CT5$
Concept_Services_3 (CS3)	CS3	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap \exists F52BoutonPanique.\top \sqcap dependPressionArterielle.CP1 \sqcap dependLocalisation.CL1 \sqcap dependLocalisation.CL2 \sqcap dependLocalisation.CL3 \sqcap dependPressionArterielle.CP2 \sqcap \exists F51DetectionChute.\top \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap \exists dependTemperatureCorporelle.CT5$
Concept_Services_4 (CS4)	CS4	$\exists F421AvertisseurEcran.\top \sqcap \exists F422AvertisseurSonore.\top$
Concept_Services_5 (CS5)	CS5	$\exists F412Priorite.\top \sqcap \exists F411Fifo.\top$
Concept_ContexteLocalisation_0 (CL0)	CL0	$\bot$
Concept_ContexteLocalisation_1 (CL1)	CL1	$\exists ValeurTemperatureCluster1.\top$
Concept_ContexteLocalisation_2 (CL2)	CL2	$\exists ValeurTemperatureCluster2.\top$
Concept_ContexteLocalisation_3 (CL3)	CL3	$Localisation$

Le tableau 4.9 présente quelques inclusions du TBox (voir Annexe B.3 pour plus de détails). La relation de subsomption entre concepts des treillis est transformée en une inclusion de concepts. Par exemple, les liens entre les concepts du treillis du contexte de *Temperature Corporelle* sont représentés comme suit : $CT4 \sqsubseteq CT5, CT1 \sqsubseteq CT5, CT3 \sqsubseteq CT4, CT2 \sqsubseteq CT4, CT0 \sqsubseteq CT3, CT0 \sqsubseteq CT2, CT0 \sqsubseteq CT1$.

Tableau 4.9 TBox : Axiomes d'inclusion

Axiomes d'inclusion
$CS12 \sqsubseteq CS13, CS11 \sqsubseteq CS13, CS9 \sqsubseteq CS13, CS4 \sqsubseteq CS13, CS15 \sqsubseteq CS13, CS1 \sqsubseteq CS13, CS5 \sqsubseteq CS13,$ $CS14 \sqsubseteq CS15, CS2 \sqsubseteq CS15, CS3 \sqsubseteq CS14, CS7 \sqsubseteq CS14, CS6 \sqsubseteq CS14, CS10 \sqsubseteq CS14, CS8 \sqsubseteq CS14,$ $CS0 \sqsubseteq CS12, CS0 \sqsubseteq CS11, CS0 \sqsubseteq CS9, CS0 \sqsubseteq CS5, CS0 \sqsubseteq CS4, CS0 \sqsubseteq CS3, CS0 \sqsubseteq CS7, CS0 \sqsubseteq$ $CS6, CS0 \sqsubseteq CS10, CS0 \sqsubseteq CS8, CS0 \sqsubseteq CS2, CS0 \sqsubseteq CS1, CT4 \sqsubseteq CT5, CT1 \sqsubseteq CT5, CT3 \sqsubseteq CT4, CT2 \sqsubseteq$ $CT4, CT0 \sqsubseteq CT3, CT0 \sqsubseteq CT2, CT0 \sqsubseteq CT1$

Nous présentons les éléments reliés au niveau assertionnel (ABox). Le tableau 4.10 montre certains individus créés (voir Annexe B.4 pour plus de détails). Les objets des treillis de concepts sont tous représentés en tant qu'individus. Par exemple, les objets du treillis de concepts Services sont des individus : *ServiceCommunication*, *ServiceAuthentification*, *ServiceGeoLocalisation*, *ServiceNotificationPatient*, *ServiceAlarme*, *ServiceCapteur*, *ServiceSoins*, *ServiceOptionPrioriteAlerte*, *ServiceAvertisseur*, *ServiceOptionAlarme*, *ServiceOptionAlarme*, *ServiceSignesVitaux* et *ServiceEnvironnement*.

Tableau 4.10 ABox : Individus

Individus
LocalisationUtilisateur1, LocalisationUtilisateur2, LocalisationUtilisateur3, ServiceCommunication, ServiceAuthentification, ServiceGeoLocalisation, ServiceNotificationPatient, ServiceAlarme, ServiceCapteur, ServiceSoins, ServiceOptionPrioriteAlerte, ServiceAvertisseur, ServiceOptionAlarme, ServiceSignesVitaux, ServiceEnvironnement

Le tableau 4.11 montre un extrait d'instances de rôle, c'est-à-dire les relations entre individus, au niveau assertionnel (ABox) (voir Annexe B.5 pour plus de détails). Les liens relationnels entre les objets des treillis de concepts deviennent des instances de rôle. Par exemple, la relation *dependTempCorporelle* établit un lien entre les individus suivants :

dependTempCorporelle(ServiceGeoLocalisation, TemperatureCorporelle1),
dependTempCorporelle(ServiceGeoLocalisation, TemperatureCorporelle2),
dependTempCorporelle(ServiceGeoLocalisation, TemperatureCorporelle3),

dependTempCorporelle(ServiceAlarme, TemperatureCorporelle1),
dependTempCorporelle(ServiceAlarme, TemperatureCorporelle2),
dependTempCorporelle(ServiceAlarme, TemperatureCorporelle3),
dependTempCorporelle(ServiceCapteur, TemperatureCorporelle1),
dependTempCorporelle(ServiceCapteur, TemperatureCorporelle2),
dependTempCorporelle(ServiceCapteur, TemperatureCorporelle3),
dependTempCorporelle(ServiceSoins, TemperatureCorporelle1),
dependTempCorporelle(ServiceSoins, TemperatureCorporelle2),
dependTempCorporelle(ServiceSoins, TemperatureCorporelle3)

Tableau 4.11 ABox : Instance de rôle

Instance de rôle
dependLocalisation(ServiceGeoLocalisation, LocalisationUtilisateur1), dependLocalisation(ServiceGeoLocalisation, LocalisationUtilisateur2), dependLocalisation(ServiceGeoLocalisation, LocalisationUtilisateur3), dependLocalisation(ServiceAlarme, LocalisationUtilisateur1), dependLocalisation(ServiceAlarme, LocalisationUtilisateur2), dependLocalisation(ServiceAlarme, LocalisationUtilisateur3), dependLocalisation(ServiceCapteur, LocalisationUtilisateur1)

Le tableau 4.12 présente quelques instanciations de concept défini précédemment dans le TBox au niveau assertionnel (ABox) (voir Annexe B.6 pour plus de détails). Les extensions de concepts formels des treillis deviennent des instances de concepts qui ont été définis dans le TBox. Par exemple, nous retrouvons dans le ABox les instances de concepts définis tels que *CS1(ServiceEnvironnement), CS2(ServiceSignesVitaux)* et *CS3(ServiceOptionAlarme),*

Tableau 4.12 ABox : Instanciation de concept défini

Instanciation de concept défini
CS1(ServiceEnvironnement), CS2(ServiceSignesVitaux), CS3(ServiceOptionAlarme), CS4(ServiceAvertisseur), CS5(ServiceOptionPrioriteAlerte), CS6(ServiceSoins), CS7(ServiceCapteur), CS8(ServiceAlarme), CL3(LocalisationUtilisateur1), CL3(LocalisationUtilisateur2), CL3(LocalisationUtilisateur3), CL1(LocalisationUtilisateur1)

Le tableau 4.13 montre quelques instances de concept primitif au niveau assertionnel (ABox) (voir Annexe B.7 pour plus de détails). Nous déterminons les incidences

entre objet et attribut d'un contexte formel s'il existe une relation binaire. Par exemple, l'attribut F42Alertes du contexte formel des Services est instancié par les concepts primitifs *F42Alertes(ServiceNotificationPatient)*.

Tableau 4.13 ABox : Assertion de concept primitif

Instanciation/assertion de concept primitif
F2Authentification(ServiceAuthentification), F41PrioriteAlerte(ServiceNotificationPatient), F42Alertes(ServiceNotificationPatient), F4NotificationPatient(ServiceNotificationPatient), F421AvertisseurEcran(ServiceAvertisseur), F422AvertisseurSonore(ServiceAvertisseur), ValeurLocalisationCluster1(LocalisationUtilisateur1), ValeurLocalisationCluster2(LocalisationUtilisateur2)

4.5.5 Ontologie

Nous pouvons obtenir la conception de l'ontologie de contexte et de service en respectant les règles de correspondance de l'analyse relationnelle de concepts à la logique descriptive (voir figure 4.2), éditée avec Protégé ("Protégé Ontology Editor," 2015) et basée sur le raisonneur Pellet ("Pellet," 2015).

Les principales opérations de raisonnement qui peuvent être tirées sont l'instanciation de concepts et la subsomption, la comparaison de concepts et l'analyse du domaine ou du co-domaine d'une relation (Bendaoud et Toussaint, 2010).

- *L'instanciation de concepts et la subsomption* : consiste à trouver le concept d'un objet, c'est-à-dire l'extension d'un concept d'un treillis ou une classe d'un individu en termes de logique descriptive (Bendaoud et Toussaint, 2010). Il permet de réponde à des questions comme « Quel est le concept de l'instance *ServiceSignesVitaux* dont les attributs sont {F611PressionArterielle, F612TempCorporelle} et les attributs relationnels sont {dependPressionArterielle : CP1, CP2, CP3, CP4, CP5,

dependTemperatureCorporelle: CT1, CT2, CT3, CT4, CT5}. La réponse est le concept défini comme suit :

$$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap \exists F611PressionArterielle.\top \sqcap dependPressionArterielle.CP1 \sqcap \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap dependPressionArterielle.CP2 \sqcap \exists F612TempCorporelle.\top \sqcap \exists dependTemperatureCorporelle.CT5.$$

Ce concept est le Concept_Service_2 (CS2) dans l'ontologie de la figure 4.2.

- *La comparaison de concepts* : consiste à déterminer si deux objets ont le même concept (Bendaoud *et al.*, 2007). Par exemple, considérons les objets *ServiceNotificationPatient* et *ServiceAvertisseur*. *ServiceNotificationPatient* est une instance du concept CS9 et *ServiceAvertisseur* est une instance du CS4. Dans un tel cas, $CS9 \sqcap CS4 = \bot$ implique que les deux objets n'appartiennent pas au même concept (voir figure 4.2).
- *L'analyse du co-domaine d'une relation*: consiste à déterminer le concept d'un objet en sachant le domaine ou le co-domaine d'une relation (Bendaoud et Toussaint, 2010) . Par exemple, « Quel est le concept de l'objet *ServiceSoins* et de quels contextes dépend-il ?». L'objet *ServiceSoins* est instancié par le concept Concept_Service_6 (CS6). D'après la définition de CS6 dans l'ontologie, il existe plusieurs relations, soit :
    - *dependLocalisation* entre les concepts CS6 et (CL1, CL2, CL3) ce qui signifie que *ServiceSoins* dépend du contexte de localisation par les instances de (CL1, CL2, CL3) : *LocalisationUtilisateur1, LocalisationUtilisateur2* et *LocalisationUtilisateur3* (voir figure 4.2).
    - *dependTemperatureCorporelle* entre les concepts CS6 et (CT1, CT2, CT3, CT4, CT5) ce qui signifie que *ServiceSoins* dépend du contexte de température corporelle par les instances de (CT1, CT2, CT3, CT4 et

CT5) : *TemperatureCorporelle1*, *TemperatureCorporelle2* et *TemperatureCorporelle3* (voir figure 4.2).

- *dependPressionArterielle* entre les concepts CS6 et (CP1, CP2, CP3, CP4, CP5) ce qui signifie que *ServiceSoins* dépend du contexte de pression artérielle par les instances de (CP1, CP2, CP3, CP4 et CP5) : *PressionArterielle1*, *PressionArterielle2* et *PressionArterielle3* (voir figure 4.2).

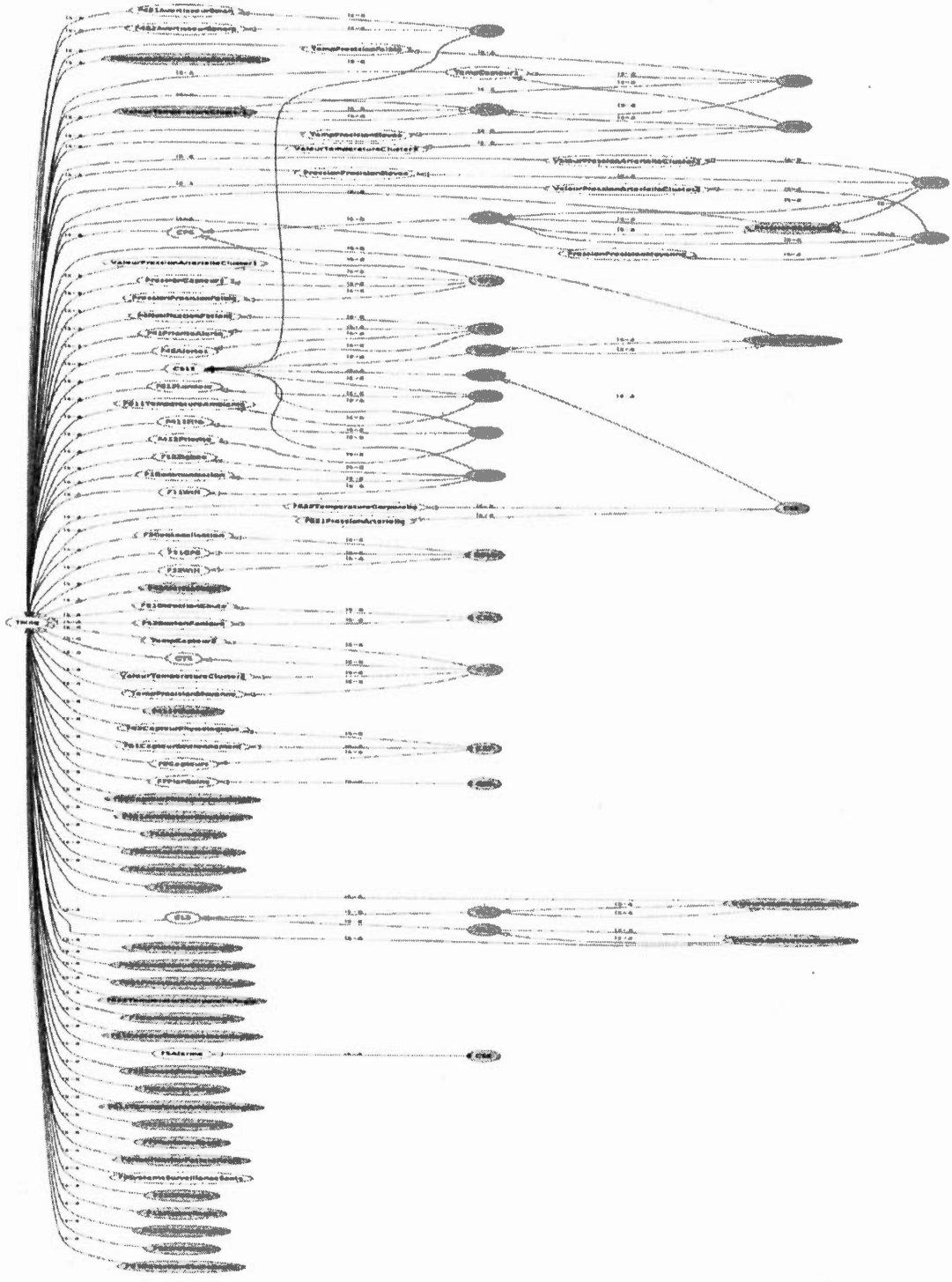


Figure 4.2 Modèle d'ontologie

4.6 Comparaison des approches

La comparaison des techniques de raisonnement sur le contexte est basée sur les critères suivants (voir tableau 4.14):

- *Efficacité, validité et complétude*: la capacité de traiter avec des mises à jour des bases de connaissances de manière dynamique est nécessaire, et, en conséquence, il devient essentiel d'utiliser une méthode de raisonnement qui est solide et complète (Perttunen *et al.*, 2009).
- *Interopérabilité*: le contexte doit être modélisé sous un format syntaxiquement et sémantiquement interopérable pour permettre le partage et la réutilisation des représentations. Les implémentations de méthodes de raisonnement devraient être normalisées pour s'assurer que les différentes méthodes produisent les mêmes résultats (Perttunen *et al.*, 2009).
- *Évaluation de changement et dérivation des informations de haut niveau* : le contexte est acquis à l'état brut et la méthode de raisonnement doit permettre d'acquérir un niveau d'abstraction de l'information contextuelle (Bettini *et al.*, 2010).
- *Habileté à soutenir des données incertaines*: la capacité d'améliorer la qualité des informations de contexte et d'inférer de nouveaux types d'informations de contexte (Bettini *et al.*, 2010).

Tableau 4.14 Comparaison des approches de raisonnement sur le contexte

Caractéristiques	Approches existantes de raisonnement sur le contexte						Approche utilisée
	Apprentissage supervisé	***Apprentissage non supervisé***	***Règles***	***Logique floue***	***Logique de premier ordre***	***Logique probabiliste***	***Logique descriptive***
Efficacité, validité et complétude	Fondé sur des approches mathématiques et statistiques. Approche n'est pas très sémantique donc moins expressive.	Approche n'est pas très sémantique donc moins expressive.	Sujettes à l'erreur due au travail manuel.	Raisonnement approximatif (n'est pas fixe ou exacte)	Raisonnement numérique limité.	Décisions basées sur des probabilités qui fournissent des résultats modérément significatifs.	Algorithmes de raisonnement valide et complet.
Interopérabilité	Validation possible pour évaluer l'exactitude des modèles de prédictions.	Difficile à valider	Aucune validation ou vérification de qualité.	Aucune validation ou vérification de qualité.	Validation possible et vérification de qualité.	Validation possible.	Validation possible et vérification de qualité.
Évaluer les changements et dériver des informations abstraites	Modèles peuvent être complexes. Difficile à saisir les connaissances existantes.	Génère des contextes abstraits, mais peu ne pas découvrir des connaissances intéressantes.	Aucun support n'explicite de dériver des informations abstraites.	Convertir les informations contextuelles brutes en informations contextuelles abstraites.	Génère les informations contextuelles abstraites à partir de contextes bruts.	Utilisé pour raisonner sur un niveau abstrait.	Génère les informations contextuelles abstraites à partir de contextes bruts.
Aptitude à soutenir des informations incertaines.	Aucun support explicite, mais diverses approches sont disponibles pour gérer l'incertitude.	Aucun support explicite, mais différentes approches sont disponibles pour gérer l'incertitude.	Gère l'incertitude.	Gère l'incertitude.	Aucun support pour l'incertitude.	Gérer des situations invisibles et d'incertitudes.	Moins approprié pour gérer l'incertitude.

En comparant notre approche avec les modèles de raisonnement existants, nous dégageons certains avantages. Notre approche de raisonnement nous permet d'acquérir un niveau abstrait de l'information contextuelle et d'inférer de nouveaux types d'informations contextuelles. Elle permet également de valider le modèle de contexte. Un aspect pertinent de cette approche est qu'elle ne dépend pas du type de modèle de contexte utilisé.

4.7 Conclusion

Dans ce chapitre, nous avons exploré le raisonnement sur le contexte dans le but de valider la cohérence du modèle de contexte et de déterminer des contextes d'abstraction de haut niveau. Pour ce faire, nous avons utilisé la logique descriptive vu qu'il existe des règles de correspondance entre les entités de cette logique et de l'analyse relationnelle de concepts. Nous avons également expliqué dans ce chapitre qu'une ontologie OWL équivaut à une base de connaissances de la logique descriptive et nous avons montré des opérations de raisonnement que nous pouvons associer au modèle d'ontologie de contextes et de services, soit : la subsomption, l'instanciation, la comparaison de concepts et la détection du domaine ou du co-domaine d'une relation. Nous avons terminé ce chapitre par une comparaison des diverses approches de raisonnement du contexte présenté dans le chapitre 2 avec celle que nous avons choisi d'utiliser.

Toutefois, la logique descriptive est limitée sur certains points, à savoir : la difficulté de représenter des relations entre les rôles, les relations n-aires, la représentation de quantifications sur les relations et les traitements numériques (Napoli, 1997). Ainsi, dans le prochain chapitre, nous utiliserons un raisonneur à base de règles pour représenter des situations contextuelles.

CHAPITRE V

MODÈLE MIXTE BASÉ SUR LA VARIABILITÉ ET L'ANALYSE RELATIONNELLE DE CONCEPTS

5.1 Introduction

Dans le chapitre 3, nous avons proposé de modéliser le contexte et de montrer les dépendances (liens) établies entre les services et les contextes à partir de l'analyse relationnelle de concepts. Par la suite, dans le chapitre 4, nous avons raisonné sur le contexte en fonction des règles de correspondance qui existent entre les éléments de l'analyse relationnelle de concepts et la logique descriptive. Bien que la logique descriptive nous ait permis de valider notre modèle de contexte et d'effectuer certaines opérations de raisonnement, elle est limitée sur certains aspects, à savoir : le manque de support pour exprimer des conditions, des contraintes numériques, des relations et des types de données personnalisés.

À partir d'un modèle de contexte validé, nous voulons exprimer des conditions (ou des situations contextuelles) qui indiquent les fonctionnalités à activer ou désactiver pour un ou plusieurs contextes donnés d'une application sensible au contexte. La logique descriptive ne nous permet pas d'effectuer ce type de raisonnement. Par conséquent, nous proposons d'utiliser une logique à base de règles afin de créer des règles de contexte qui représentent des états ou des situations contextuelles et qui détermineront les fonctionnalités à activer ou désactiver selon les valeurs du contexte. Chaque règle de contexte fait référence à un ensemble de fonctionnalités qui représentent une configuration.

Nous supposons qu'un service peut offrir une ou plusieurs fonctionnalités d'une application dépendante du contexte. Nous considérons également le fait que certaines fonctionnalités ne dépendent pas du contexte, c'est-à-dire qu'un service est sensible au contexte uniquement dans le cas où la fonctionnalité offerte à un utilisateur en dépend.

Pour ce faire, nous nous inspirons des modèles de variabilité pour exprimer les fonctionnalités d'une application sensible au contexte. Bien que les modèles de variabilité manquent de support sémantique (Bidarra et Bronsvoort, 2000), nous avons modélisé le modèle de variabilité d'une application sensible au contexte sous forme d'ontologie OWL et nous le validons avec le raisonneur Pellet. Nous établissons un lien sémantique entre l'analyse relationnelle de concepts et les *features* d'un modèle de variabilité (Amja, Obaid, et Mili, 2016; Amja, Obaid, Mili, et Valtchev, 2016). Ainsi, nous exprimons des règles de contexte à partir d'un langage du web sémantique, soit *Semantic Web Rule Language* (SWRL).

Ce chapitre est organisé comme suit. Nous présentons dans la section 5.2 le problème spécifique que nous voulons résoudre et la solution proposée. La section 5.3 présente les notions de base des modèles de variabilité. La section 5.4 explique les règles utilisées pour traduire le modèle de variabilité sous forme d'ontologie OWL. Dans la section 5.5, nous montrons la validation des configurations avec le raisonneur Pellet. Par la suite, nous expliquons en détail dans la section 5.6 le lien entre l'analyse relationnelle de concepts et le modèle de variabilité. Dans la section 5.7, nous présentons les règles de contexte qui définissent des configurations valides pour des valeurs de contexte précises ou des intervalles de valeurs de contexte. Nous terminons ce chapitre par la section 5.8 qui explique notre programme écrit en Groovy. Ce programme permet d'ajouter de nouvelles configurations ou de modifier des configurations existantes, et de les valider avec un raisonneur. Dans la section 5.9, nous concluons ce chapitre.

5.2 Approche proposée

À partir de notre exemple de surveillance de l'état de santé de patients, nous supposons que les informations contextuelles captées sont les suivantes :

- Contexte de température corporelle : une valeur de 40°C dont la valeur a une précision faible et elle a été captée le capteur ayant comme ID TempCapteur1.
- Contexte de pression artérielle: une valeur de 120/80 dont la valeur a une précision élevée et elle a été captée le capteur ayant comme ID PressionCapteur1.
- Contexte de géolocalisation : les coordonnées géographiques sont (45.53, -73.76).

Selon les informations contextuelles captées, nous nous posons la question suivante « Comment est-ce qu'on peut vérifier si les fonctionnalités installées ou actives sur l'appareil mobile permettent de répondre à l'état du système ?» La figure 5.1 illustre la problématique.

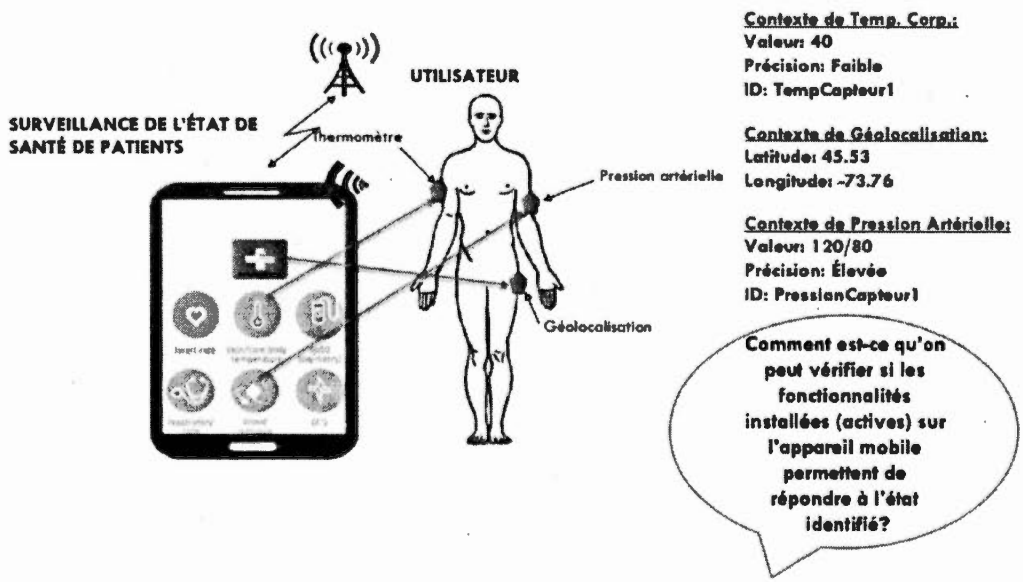


Figure 5.1 Fonctionnalités contextuelles actives selon le contexte

Notre approche consiste à construire un modèle d'ontologie qui comprend les informations contextuelles modélisées et le modèle de *features* de l'application sensible au contexte. Autrement dit, nous ramenons d'abord le modèle de contexte que nous avons modélisé à partir de l'analyse relationnelle de concepts sous forme d'ontologie. Dans le chapitre précédent, nous avons présenté les règles de correspondance entre les entités de l'analyse relationnelle de concepts et la logique descriptive. Par la suite, nous représentons le modèle de features de l'application sensible au contexte sous forme d'ontologie afin d'ajouter une sémantique au modèle, et de valider le modèle et ses configurations. Ainsi, nous nous retrouvons avec un seul modèle d'ontologie qui comprend les informations contextuelles, les services, le modèle de *features* et ses configurations. À partir de ce modèle d'ontologie, nous construisons des règles contextuelles qui se composent d'un ensemble de contextes. Ces règles sont associées à des configurations valides du modèle de *features*. Par configuration, nous sous-entendons un ensemble de fonctionnalités qui nécessitent d'être activées selon le contexte courant de l'utilisateur. Nous avons utilisé le logiciel Protégé pour modéliser le modèle d'ontologie et le raisonneur Pellet. Nos règles de contextes sont construites à partir du langage SWRL. Nous notons qu'ici nous utilisons deux types de logique : la logique descriptive et la logique à base de règles. La figure 4.2 illustre notre approche.

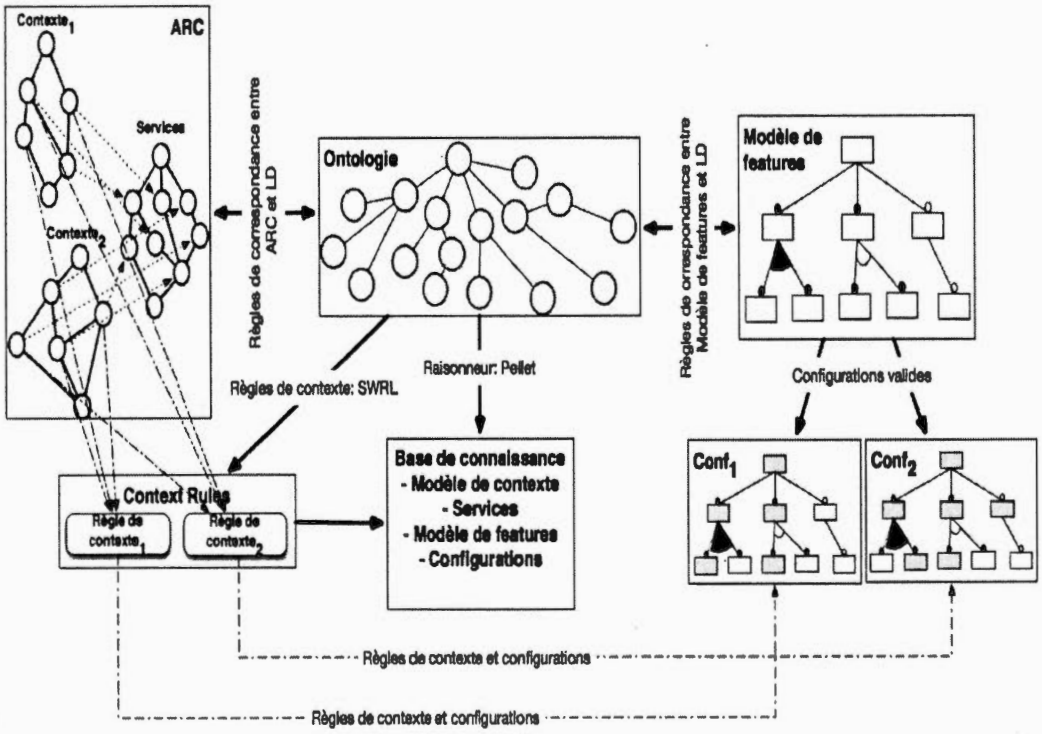


Figure 5.2 Approche proposée

5.3 Modèle de variabilité

Une ligne de produits est un ensemble de systèmes partageant un ensemble de fonctionnalités communes, satisfaisant des besoins spécifiques pour un domaine particulier, et développés de manière contrôlée à partir d'un ensemble commun d'éléments réutilisables (Clements et Northrop, 2002).

La variabilité dans les lignes de produits logiciels se définit comme le regroupement des caractéristiques qui différencient les produits logiciels appartenant à la même ligne de produits (Royer et Arboleda, 2012). Il s'agit d'un concept relié au *point de variation*. Un point de variation identifie la partie du système où une variation sera introduite dans la famille de produits pour générer des élèments de la ligne (Royer et Arboleda, 2012). Ces points de variation déterminent où, comment et pourquoi ces variabilités peuvent apparaître.

Le modèle des *features* est une notation standard pour décrire la variabilité dans les lignes de produits (Appel *et al.*, 2013). Un modèle de *features* est représenté sous la forme d'un arbre composé de nœuds reliés par des arêtes. Les nœuds correspondent à des *features* alors que les arêtes correspondent à une relation entre deux *features*.

Les types de *features* sont :

1) Racine (*root feature*) : Ce *feature* détermine le point d'entrée du modèle et il représente l'unique nœud ne possédant pas de parent.

2) Optionnel (*optional features*) : Un *feature* optionnel n'est pas nécessairement sélectionné si un de ses parents est sélectionné.

3) Obligatoire (*mandatory features*) : Un *feature* obligatoire est sélectionné si un de ses parents est également sélectionné.

Les *features* sont reliés par différents types de relations :

1) Décompositions: consistent à décomposer les *features* en *sous-features* suivant une décomposition et définissent des contraintes entre des *features* partageant le même parent.

    a. *AND* : Dans le cas où le parent est sélectionné, alors ses enfants le sont également.

    b. *XOR* : Dans le cas où le parent est sélectionné, alors un et un seul de ses enfants peut être sélectionné.

    c. *IOR* : Dans le cas où le parent est sélectionné, alors au moins un de ses enfants peut être sélectionné.

2) Contraintes : permet d'exprimer des contraintes entre des *features* qui ne partagent pas le même parent.

a. *Requires* : si un *feature f1* est sélectionné alors *f2* doit nécessairement l'être, mais pas inversement (f1 *requires* f2).

b. *Excludes* : si un *feature f1* est sélectionné alors l'autre *feature f2* ne peut pas l'être en même temps pour le même produit (*f1* excludes *f2*).

La figure 5.3 est un exemple de modèle de *features* pour un système de surveillance de l'état de santé de patients.

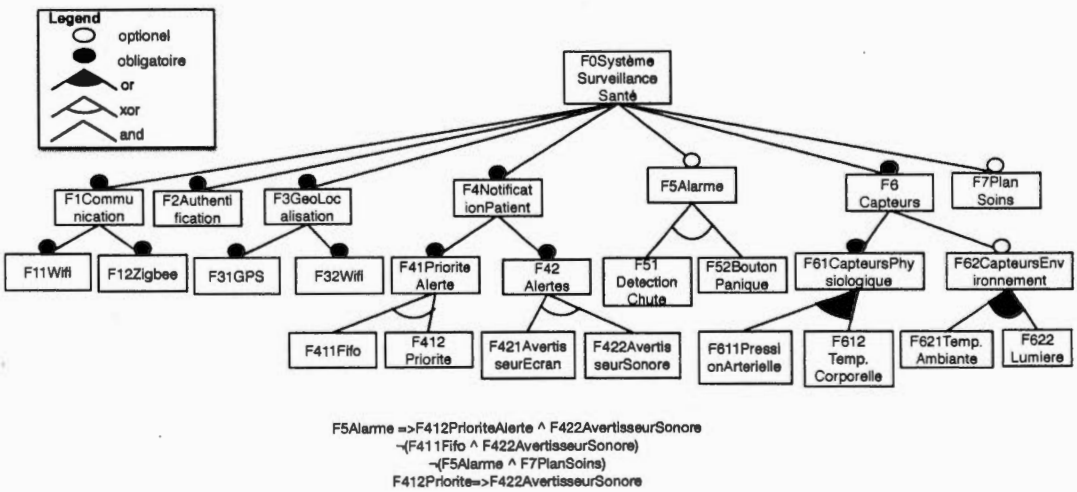


Figure 5.3 Modèle de *features* d'un système de surveillance de l'état de santé de patients

5.4 Modélisation du modèle de *features* en OWL

Le modèle de *features* est la notation la plus souvent utilisée pour modéliser la variabilité des lignes de produits logiciels. Toutefois, les modèles de *features* n'offrent pas de sémantique formelle et il n'existe pas d'outil automatisé permettant de vérifier la consistance des configurations basée sur les contraintes spécifiées dans le modèle de *features*.

Nous nous inspirons des travaux de Piash *et al.* (2013), Wang *et al.* (2007; 2005), et Noorian *et al.* (2011) pour modéliser un modèle de *features* en ontologie, plus précisément OWL-DL.

En nous basant sur l'exemple le modèle de *features*, illustré par la figure 5.3, nous construisons l'ontologie du modèle de *features* en 3 étapes :

a) Identifier les nœuds du modèle de *features* : La première étape consiste à identifier les nœuds présents dans le modèle de *features*. Ces nœuds (*features*) sont représentés sous forme de concepts OWL, c'est-à-dire des sous-concepts du concept universel (*top concept*). Ainsi, tous les *features* du modèle de *features* sont représentés par des concepts en ontologie comme suit :

- $F0SystemeSurveillanceSante \sqsubseteq T,$
- $F1Communication \sqsubseteq T,$
- $F11Wifi \sqsubseteq T,$
- $F12Zigbee \sqsubseteq T,$
- $F2Authentification \sqsubseteq T,$
- $F3GeoLocalisation \sqsubseteq T,$
- $F31GPS \sqsubseteq T,$
- $F32Wifi \sqsubseteq T,$
- $F4NotificationPatient \sqsubseteq T,$
- $F41PrioriteAlerte \sqsubseteq T,$
- $F411Fifo \sqsubseteq T,$
- $F412Priorite \sqsubseteq T,$
- $F42Alertes \sqsubseteq T,$
- $F421AvertisseurEcran \sqsubseteq T,$
- $F422AvertisseurSonore \sqsubseteq T,$
- $F5Alarme \sqsubseteq T,$
- $F51DetectionChute \sqsubseteq T,$
- $F52BoutonAlarme \sqsubseteq T,$
- $F6Capteurs \sqsubseteq T,$
- $F61CapteursPhysiologique \sqsubseteq T,$
- $F611PressionArterielle \sqsubseteq T,$
- $F612TempCorporelle \sqsubseteq T,$
- $F62CapteursEnviroonnement \sqsubseteq T,$
- $F621TempAmbiante \sqsubseteq T,$

- $F622Lumiere \sqsubseteq T$,
- $F7PlanSoins \sqsubseteq T$

b) Créer des propriétés d'objets : La racine et les features d'un modèle de features sont interreliés par différentes relations, représentées par des arêtes. Une propriété d'objet est créée pour chaque arête et le co-domaine de cette propriété est le concept du feature respectif. Les propriétés d'objets sont représentées comme suit :

- $aF0SystemeSurveillanceSante \sqsubseteq ObjectProperty$
- $aF1Communication \sqsubseteq ObjectProperty$
- $aF11Wifi \sqsubseteq ObjectProperty$
- $aF12Zigbee \sqsubseteq ObjectProperty$
- $aF2Authentification \sqsubseteq ObjectProperty$
- $aF3GeoLocalisation \sqsubseteq ObjectProperty$
- $aF31GPS \sqsubseteq ObjectProperty$
- $aF32Wifi \sqsubseteq ObjectProperty$
- $aF4NotificationPatient \sqsubseteq ObjectProperty$
- $aF41PrioriteAlerte \sqsubseteq ObjectProperty$
- $aF411Fifo \sqsubseteq ObjectProperty$.
- $aF412Priorite \sqsubseteq ObjectProperty$
- $aF42Alertes \sqsubseteq ObjectProperty$
- $aF421AvertisseurEcran \sqsubseteq ObjectProperty$
- $aF422AvertisseurSonore \sqsubseteq ObjectProperty$
- $aF5Alarme \sqsubseteq ObjectProperty$
- $aF51DetectionChute \sqsubseteq ObjectProperty$
- $aF52BoutonAlarme \sqsubseteq ObjectProperty$
- $aF6Capteurs \sqsubseteq ObjectProperty$
- $aF61CapteursPhysiologique \sqsubseteq ObjectProperty$
- $aF611PressionArterielle \sqsubseteq ObjectProperty$
- $aF612TempCorporelle \sqsubseteq ObjectProperty$
- $aF62CapteursEnviroonnement \sqsubseteq ObjectProperty$
- $aF621TempAmbiante \sqsubseteq ObjectProperty$
- $aF622Lumiere \sqsubseteq ObjectProperty$
- $aF7PlanSoins \sqsubseteq ObjectProperty$

De plus, le co-domaine d'une propriété d'objet qui est le feature respectif est représenté comme suit :

- $T \sqsubseteq \forall aF0SystemeSurveillanceSante.F0SystemeSurveillanceSante$
- $T \sqsubseteq \forall aF1Communication.F1Communication$
- $T \sqsubseteq \forall aF11Wifi.F11Wifi$
- $T \sqsubseteq \forall aF12Zigbee.F12Zigbee$
- $T \sqsubseteq \forall aF2Authentification.F2Authentification$
- $T \sqsubseteq \forall aF3GeoLocalisation.F3GeoLocalisation$
- $T \sqsubseteq \forall aF31GPS.F31GPS$
- $T \sqsubseteq \forall aF32Wifi.F32Wifi$
- $T \sqsubseteq \forall aF4NotificationPatient.F4NotificationPatient$
- $T \sqsubseteq \forall aF41PrioriteAlerte.F41PrioriteAlerte$
- $T \sqsubseteq \forall aF411Fifo.F411Fifo$
- $T \sqsubseteq \forall aF412Priorite.F412Priorite$
- $T \sqsubseteq \forall aF42Alertes.F42Alertes$
- $T \sqsubseteq \forall aF421AvertisseurEcran.F421AvertisseurEcran$
- $T \sqsubseteq \forall aF422AvertisseurSonore.F422AvertisseurSonore$
- $T \sqsubseteq \forall aF5Alarme.F5Alarme$
- $T \sqsubseteq \forall aF51DetectionChute.F51DetectionChute$
- $T \sqsubseteq \forall aF52BoutonAlarme.F52BoutonAlarme$
- $T \sqsubseteq \forall aF6Capteurs.F6Capteurs$
- $T \sqsubseteq \forall aF61CapteursPhysiologique.F61CapteursPhysiologique$
- $T \sqsubseteq \forall aF611PressionArterielle.F611PressionArterielle$
- $T \sqsubseteq \forall aF612TempCorporelle.F612TempCorporelle$
- $T \sqsubseteq \forall aF62CapteursEnviroonnement.F62CapteursEnvironnement$
- $T \sqsubseteq \forall aF621TempAmbiante.F621TempAmbiante$
- $T \sqsubseteq \forall aF622Lumiere.F622Lumiere$
- $T \sqsubseteq \forall aF7PlanSoins.F7PlanSoins$

c) Créer les concepts Règle : Cette étape consiste à créer un concept Règle pour chaque nœud du modèle de *features*. Un concept *Règle* a 2 conditions :

   a. Une *condition nécessaire et suffisante* en utilisant une restriction existentielle pour lier le noeud *règle* au nœud du *feature* correspondant dans le modèle de *features*.

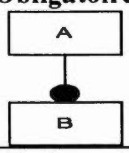
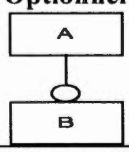
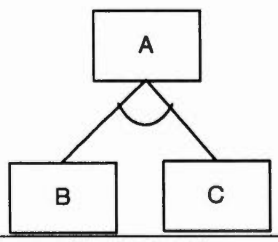
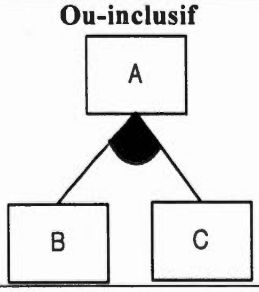
b. Des *contraintes nécessaires* spécifiant comment chaque *feature* enfant est relié à son parent, c'est-à-dire capturer les liens entre *features*, et comment un *feature* est contraint par d'autres *features*, c'est-à-dire les contraintes *requires* et *excludes* du modèle de *features*.

Les concepts Règles sont décrits comme suit :

- $F0SystemeSurveillanceSanteRegle \sqsubseteq T$
- $F1CommunicationRegle \sqsubseteq T$
- $F11WifiRegle \sqsubseteq T$
- $F12ZigbeeRegle \sqsubseteq T$
- $F2AuthentificationRegle \sqsubseteq T$
- $F3GeoLocalisationRegle \sqsubseteq T$
- $F31GPSRegle \sqsubseteq T$
- $F32WifiRegle \sqsubseteq T$
- $F4NotificationPatientRegle \sqsubseteq T$
- $F41PrioriteAlerteRegle \sqsubseteq T$
- $F411FifoRegle \sqsubseteq T$
- $F412PrioriteRegleRegle \sqsubseteq T$
- $F42AlertesRegle \sqsubseteq T$
- $F421AvertisseurEcranRegle \sqsubseteq T$
- $F422AvertisseurSonoreRegle \sqsubseteq T$
- $F5AlarmeRegle \sqsubseteq T$
- $F51DetectionChuteRegle \sqsubseteq T$
- $F52BoutonAlarmeRegle \sqsubseteq T$
- $F6CapteursRegle \sqsubseteq T$
- $F61CapteursPhysiologiqueRegle \sqsubseteq T$
- $F611PressionArterielleRegle \sqsubseteq T$
- $F612TempCorporelleRegle \sqsubseteq T$
- $F62CapteursEnviroonnementRegle \sqsubseteq T$
- $F621TempAmbianteRegle \sqsubseteq T$
- $F622LumiereRegle \sqsubseteq T$
- $F7PlanSoinsRegle \sqsubseteq T$

Le tableau 5.1 présente la représentation en OWL-DL des différents types de liens entre *features*.

Tableau 5.1 Représentation OWL-DL des types de liens entre *features*

Type de *features*	Représentation en OWL-DL
Obligatoire 	$ARegle \sqsubseteq \exists\, aB_i \cdot B_i \; pour \; 1 \leq i \leq n$
Optionnel 	$B_iRegle \sqsubseteq \exists\, aB_i \cdot B_i \; pour \; 1 \leq i \leq n$
Alternatif 	$ARegle \sqsubseteq ((\exists\, aB \cdot B) \sqcup (\exists\, hasC \cdot C))$ $ARegle \sqsubseteq \neg((\exists\, aB \cdot B) \sqcap (\exists\, hasC \cdot C))$
Ou-inclusif 	$ARegle \sqsubseteq ((\exists\, aB \cdot B) \sqcup (\exists\, hasC \cdot C))$
Requires	$ARegle \sqsubseteq \exists\, aB_i \cdot B_i \; pour \; 1 \leq i \leq n$
Excludes	$ARegle \sqsubseteq \neg(\exists\, aB_i \cdot B_i) \; pour \; 1 \leq i \leq n$

Le tableau 5.2 présente des exemples de définition des règles qui définissent les liens entre les *features* en OWL.

Tableau 5.2 Exemple des types de lien entre *features*

***Feature* et contrainte**	**Exemple en OWL**
***Feature* obligatoire**	$F0SystemeSurveillanceSanteRegle \sqsubseteq ((\exists\ aF1Communication.F1Communication) \sqcap (\exists\ aF2Authentification.F2Authentification) \sqcap (\exists\ aF3GeoLocalisation.F3GeoLocalisation) \sqcap (\exists\ aF4NotificationPatient.F4NotificationPatient) \sqcap (\exists\ aF6Capteurs.F6Capteurs))$
	$F1CommunicationRegle \sqsubseteq ((\exists\ aF11Wifi.F11Wifi) \sqcap (\exists\ aF12Zigbee.F12Zigbee))$
	$F3GeoLocalisationRegle \sqsubseteq ((\exists\ aF31GPS.F31GPS) \sqcap (\exists\ aF32Wifi.F32Wifi))$
	$F4NotificationPatientRegle \sqsubseteq ((\exists\ aF41PrioriteAlerte.F41PrioriteAlerte) \sqcap (\exists\ aF42ZAlertes.F42Alertes))$
***Feature* optionnel**	$F7PlanSoinsRegle \sqsubseteq (\exists\ aF7PlanSoins.F7PlanSoins)$ $F5AlarmeRegle \sqsubseteq (\exists\ aF5Alarme.F5Alarme)$ $F62CapteursEnvironnementRegle \sqsubseteq (\exists\ aF62CapteursEnvironnement.F62CapteursEnvironnement)$
***Feature* alternative (XOR)**	$F41AlertePrioriteRegle \sqsubseteq ((\exists\ aF411Fifo.F411Fifo) \sqcup (\exists\ aF412Priorite.F412Priorite))$ $F41AlertePrioriteRegle \sqsubseteq \neg((\exists\ aF411Fifo.F411Fifo) \sqcap (\exists\ aF412Priorite.F412Priorite))$
	$F42AlertesRegle \sqsubseteq ((\exists\ aF421AvertisseurEcran.F411AvertisseurEcran) \sqcup (\exists\ aF422AvertisseurSonore.F422AvertiseurSonore))$ $F42AlertePrioriteRegle \sqsubseteq \neg((\exists\ aF421AvertisseurEcran.F421AvertisseurEcran) \sqcap (\exists\ aF422AvertisseurSonore.F422AvertisseurSonore))$
	$F5AlarmeRegle \sqsubseteq ((\exists\ aF51DetectionChute.F51DetectionChute) \sqcup (\exists\ aF52BoutonPanique.F52BoutonPanique))$ $F5AlarmeRegle \sqsubseteq \neg((\exists\ aF51DetectionChute.F51DetectionChute) \sqcap (\exists\ aF52BoutonPanique.F52BoutonPanique))$
***Feature* ou (IOR)**	$F6CapteursRegle \sqsubseteq (\exists\ aF62CapteursPhysiologique.F62CapteursPhysiologique)$
	$F61CapteursPhysiologiqueRegle \sqsubseteq ((\exists\ aF611PressionArterielle.F611PressionArterielle) \sqcup (\exists\ aF612TempCorporelle.F612TempCorporelle))$
	$F62CapteursEnvironnementRegle \sqsubseteq ((\exists\ aF621TempAmbiante.F621TempAmbiante) \sqcup (\exists\ aF622Lumiere.F622Lumiere))$
Contraintes *requires*	$F5AlarmeRegle \sqsubseteq ((\exists\ aF412PrioriteAlerte.F412Priorite) \sqcap (\exists\ aF422AvertisseurSonore.F422AvertisseurSonore))$
	$F412PrioriteRegle \sqsubseteq (\exists\ aF422AvertisseurSonore.F422AvertisseurSonore)$
Contraintes *excludes*	$F411FifoRegle \sqsubseteq \neg\ (\exists\ aF422AvertisseurSonore.F422AvertisseurSonore)$
	$F5AlarmeRegle \sqsubseteq \neg\ (\exists\ aF7PlanSoins.F7PlanSoins)$

5.5 Vérification de la configuration des lignes de produits logiciels avec le raisonneur Pellet

Nous construisons le modèle de *features* de l'exemple de la figure 5.1 en ontologie en utilisant l'outil Protégé et le raisonneur Pellet afin de vérifier toute inconstance de configuration. Pour chaque configuration, nous créons une instance (un individu) de

F0SystemeSurveillanceSanteRegle. Par exemple, nous ajoutons l'individu $C1$ et nous lui donnons la configuration comme suit :

$$C1 \equiv (aF1Communication\ some\ F1Communication) \sqcap (aF11Wifi\ some\ F11Wifi)$$
$$\sqcap (aF12Zigbee\ some\ F12Zigbee)$$
$$\sqcap (aF2Authentification\ some\ F2Authentification)$$
$$\sqcap (aF3GeoLocalisation\ some\ F3GeoLocalisation)$$
$$\sqcap (aF31GPS\ some\ F31\ GPS)$$
$$\sqcap (aF4NotificationPatient\ some\ F4NotificationPatient)$$
$$\sqcap (aF41PrioriteAlerte\ some\ F41PrioriteAlerte)$$
$$\sqcap (aF42Alertes\ some\ F42Alertes)$$
$$\sqcap (\neg(aF422AvertisseurSonore\ some\ F422AvertisseurSonore))$$
$$\sqcap (aF5Alarme\ some\ F5Alarme) \sqcap (aF6Capteurs\ some\ F6Capteurs)$$
$$\sqcap (aF7PlanSoins\ some\ F7PlanSoins)$$

Le raisonneur Pellet se plaindra que C1 est incohérent (voir figure 5.4). En examinant de près la configuration, nous constatons qu'une des contraintes du modèle de *features* n'est pas respectée, plus particulièrement *F5Alarme excludes F7PlanSoins*.

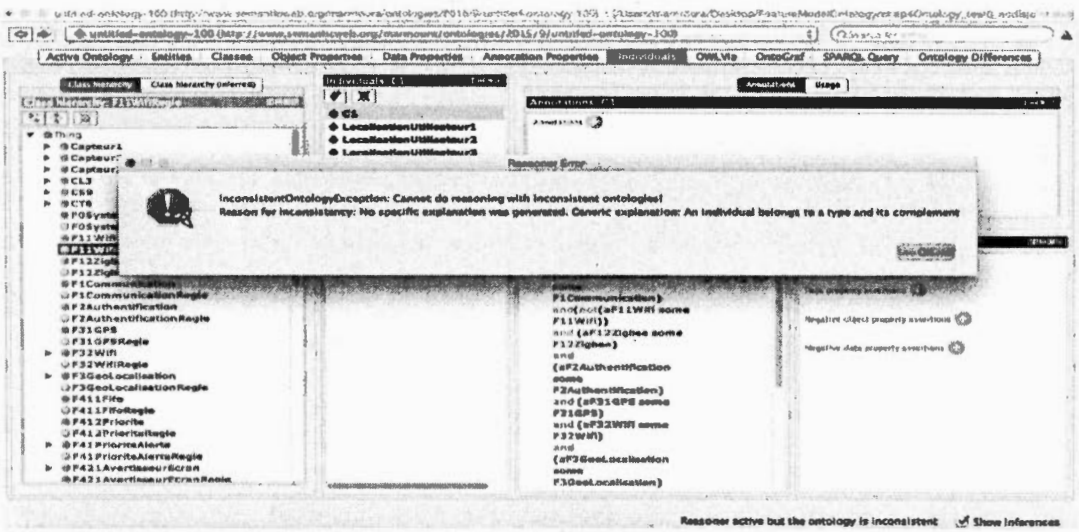


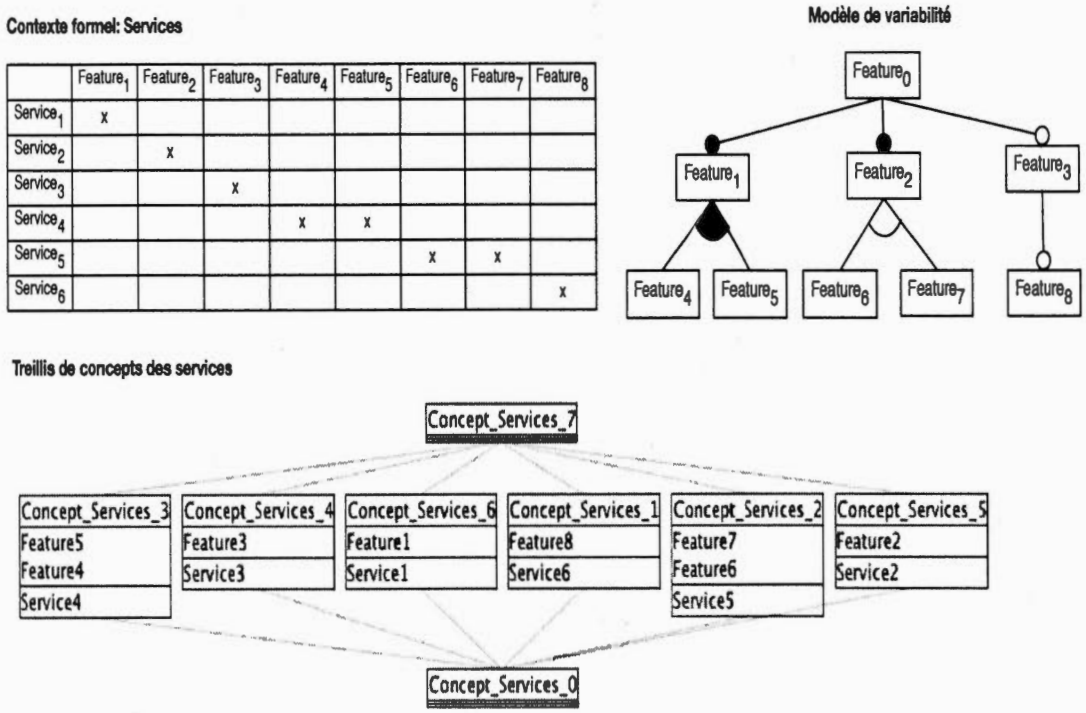
Figure 5.4 Exemple d'une configuration inconsistante en OWL

Nous corrigeons la configuration de $C1$ en ajoutant la négation à $F7PlanSoins$ mais le raisonneur Pellet continue de signaler une incohérence. Cette incohérence provient de $F422AvertisseurSonore$. D'une part, nous avons la contrainte $F5Alarme\ requires\ (F412Priorite \wedge F422AvertisseurSonore)$ et, d'autre part, nous avons la contrainte $F412Priorite\ requires\ F422AvertisseurSonore$. Ainsi, nous enlevons la négation de $F422AvertisseurSonore$ et nous ajoutons le *feature* $F412Priorite$ pour obtenir une configuration sans incohérence. La configuration $C1$ se définit donc comme suit :

$$
\begin{aligned}
C1 \equiv\ & (aF1Communication\ some\ F1Communication) \sqcap\ (aF11Wifi\ some\ F11Wifi) \\
& \sqcap (aF12Zigbee\ some\ F12Zigbee) \\
& \sqcap (aF2Authentification\ some\ F2Authentification) \\
& \sqcap (aF3GeoLocalisation\ some\ F3GeoLocalisation) \\
& \sqcap (aF31GPS\ some\ F31\ GPS) \\
& \sqcap (aF4NotificationPatient\ some\ F4NotificationPatient) \\
& \sqcap (aF41PrioriteAlerte\ some\ F41PrioriteAlerte) \\
& \sqcap (aF412Priorite\ some\ F412Priorite) \sqcap\ (aF42Alertes\ some\ F42Alertes) \\
& \sqcap (aF422AvertisseurSonore\ some\ F422AvertisseurSonore) \\
& \sqcap (aF5Alarme\ some\ F5Alarme) \sqcap (aF6Capteurs\ some\ F6Capteurs) \\
& \sqcap (\neg(aF7PlanSoins\ some\ F7PlanSoins))
\end{aligned}
$$

5.6 Lien entre l'analyse relationnelle de concepts et le modèle de *features*

Dans le chapitre 3, nous avons représenté des services sous forme d'un treillis de concepts. Les objets de ce treillis représentent des services, c.-à-d. des composants offrant un service spécifique. Les attributs représentent des *features* que nous retrouvons dans un modèle de variabilité. La figure 5.5 illustre ce concept. Nous omettons le *feature* racine, car il représente un système, soit un *feature* abstrait.



	$Feature_1$	$Feature_2$	$Feature_3$	$Feature_4$	$Feature_5$	$Feature_6$	$Feature_7$	$Feature_8$
$Service_1$	x							
$Service_2$		x						
$Service_3$			x					
$Service_4$				x	x			
$Service_5$						x	x	
$Service_6$								x

Figure 5.5 Treillis de concept des services

Chaque contexte est également représenté sous forme d'un treillis de concepts. La figure 5.6 illustre un exemple de treillis de concepts pour le contexte de température corporelle.

Contexte formel:Contexte de température corporelle

	Precision Faible	Precision Moyenne	Precision Elevee	Valeur Cluster1	Valeur Cluster2
$TempCorp_1$	x			x	
$TempCorp_2$		x			x
$TempCorp_3$			x		x

Treillis de concepts du contexte de température corporelle

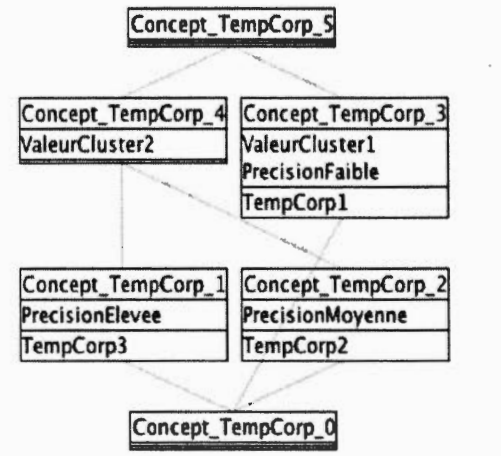


Figure 5.6 Treillis de concepts du contexte de température corporelle

En appliquant l'analyse relationnelle de concepts, nous établissons une dépendance entre les services et leurs contextes. La figure 5.7 illustre un exemple.

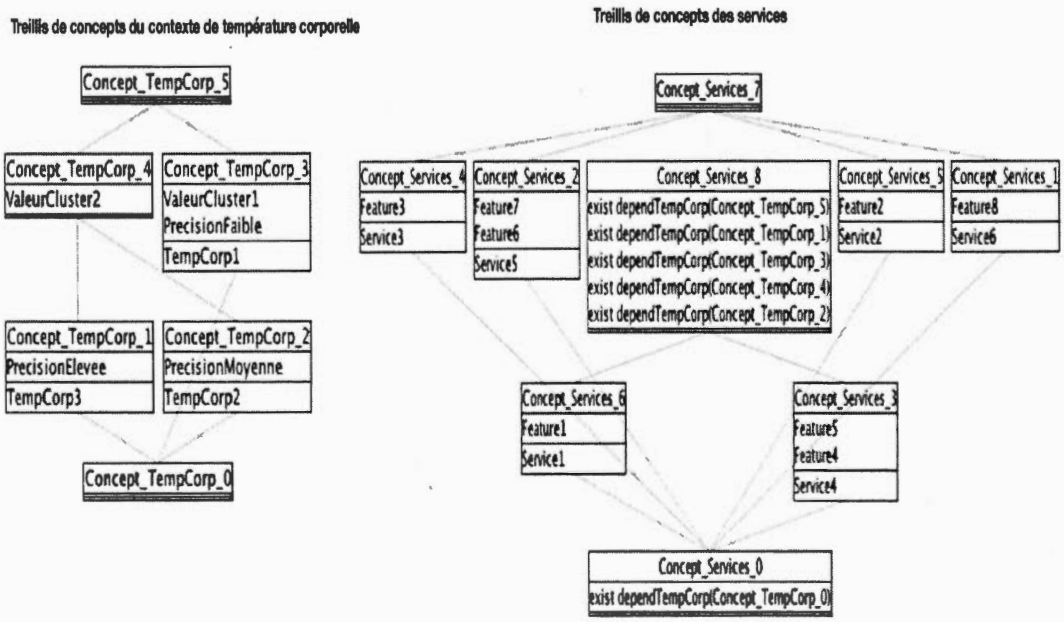


Figure 5.7 Treillis de concepts

Nous modélisons le contexte en utilisant cette approche dans le but de :

- produire une description formelle des informations contextuelles qui sont présentes pour un système dépendant du contexte.
- établir des liens selon les données contextuelles structurées.

À partir du contexte modélisé, nous établissons des règles de contexte qui sont détaillées dans la section 5.7.

5.7 Configuration d'exécution contextuelle

Dans un modèle de variabilité, la configuration d'un système s'établit par la sélection ou la désélection de *features* en respectant les contraintes associées sur la combinaison de *features* imposé par le modèle. Par exemple, le modèle de *features* du système de surveillance de l'état de santé de patients (voir figure 5.1) comprend 72 configurations possibles.

Nous associons une règle de contextes à chacune des configurations valides du modèle de variabilité. Une règle de contexte est composée d'une ou plusieurs valeurs de contextes et correspond à une configuration spécifique. La figure 5.8 illustre ce concept.

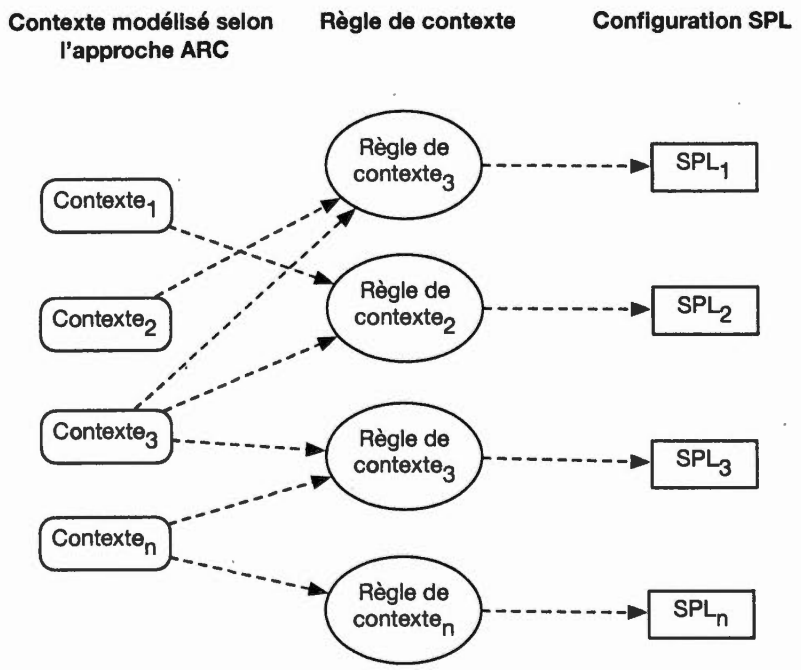


Figure 5.8 Règle de contexte et configuration contextuelle

5.7.1 Règles de contextes

À partir de la modélisation du contexte, nous définissons des règles de contextes. Chaque règle de contexte est représentée sous forme d'expression SWRL (W3C, 2004).

SWRL (*Semantic Web Rule Language*) est un langage de règles pour le web sémantique, combinant les langages d'ontologies OWL-DL et de règles RuleML (*Rule Markup Language*).

Les règles sont construites suivant le format *antécédent -> conséquent*. L'antécédent est une conjonction d'atomes et le conséquent et un seul atome. L'atome étant soit :

- Une instance de concept (prédicat unaire) : C(z)
- Une relation OWL (prédicat binaire) : R(x,y)

- Des relations SWRL telles que *same-as*(?x, ?y) et *different-from*(?x, ?y).

Une règle fonctionne selon le principe de satisfiabilité de l'antécédent ou du conséquent, soit :

- L'antécédent et le conséquent sont définis : si l'antécédent est satisfait alors le conséquent est aussi satisfait.
- L'antécédent est vide et cela revient à un antécédent satisfait : permet de définir des faits, c.-à-d. de décrire des individus.
- Le conséquent est vide et cela revient à un conséquent insatisfait : l'antécédent ne doit pas être satisfiable.

SWRL permet de manipuler des instances par des variables et d'ajouter des relations suivant les valeurs des variables, c.-à-d. des individus, et la satisfaction de la règle.

SWRL repose sur l'hypothèse du monde ouvert : le manque d'information est considéré comme une ignorance. Il n'est pas possible de vérifier si quelque chose est faux à moins qu'il soit explicitement mentionné.

SWRL comporte certaines limitations. Elle ne permet pas de règles avec des *disjonctions*. Par contre, nous pouvons tout de même contourner cette restriction. Par exemple, la règle suivante n'est pas possible : A(?x) v B(?x) –>C(?x). Nous pouvons remédier à ce cas en utilisant deux règles qui produisent : A(?x) –>C(?x) et B(?x) –>C(?x). De plus, SWRL ne permet pas la *négation par l'échec*. Par exemple, nous ne pouvons pas exprimer Etudiant(?e) ^ ¬aVoiture(?e, ?v)->SansVoiture(?e). Toutefois, la *négation classique* est permise dans les règles. Nous pouvons avoir, par exemple (not Etudiant)(?x)->NonPersonneInscrite(?x). Cette conclusion ne peut qu'être atteinte que pour des individus dont nous pouvons conclure qu'ils ne peuvent être membres de la classe Etudiant.

Nous représentons les critères de contexte dans l'antécédent et nous faisons référence au modèle de résolution de la variabilité dans le conséquent. Le modèle de résolution de la variabilité est expliqué dans la section 5.7.2. En d'autres termes, le conséquent représente une configuration valide du modèle de variabilité. Des exemples sont détaillés dans la section 5.7.3.

Nous avons utilisé le moteur d'inférence Pellet qui supporte SWRL. Pellet intègre les règles SWRL dans le moteur d'inférence OWL-DL fondé sur les algorithmes des tableaux sémantiques.

5.7.2 Modèle de résolution de la variabilité

Vu que les règles de contextes peuvent déclencher l'activation et la désactivation des *features*, nous utilisons le concept de *résolution de la variabilité* (RV) qui consiste à déterminer une configuration souhaitée en spécifiant la valeur de chaque *feature* du modèle de variabilité (Cetina *et al.*, 2009).

La *résolution de la variabilité* est une paire *(F, E)* où *F* indique un *feature* et *E* indique l'état de ce *feature*, c.-à-d. actif ou inactif. Chaque *résolution de la variabilité* est associée à une ou plusieurs règles de contextes et représente le changement d'état du *feature* (activation-désactivation) lorsque ces règles de contextes sont accomplies. Nous exprimons une résolution de la variabilité comme suit (Cetina *et al.*, 2009) :

$$RV = \{(F, E)\} |\ F \in [MF]\ E \in \{Activé, Désactivé\}$$

5.7.3 Exemples de règles de contexte et configuration

En nous basant sur le modèle de *features* représenté par la figure 5.3, nous obtenons 72 modèles de *résolution de la variabilité*. Chacun d'entre eux est associé à une règle de contexte. Dans cette section, nous détaillons 3 exemples.

Il faut noter que l'association pour les règles de contexte et leurs configurations correspondantes n'apparaît pas pour les *features* alternatifs, optionnels ou ou-

inclusifs. Cela s'explique par l'hypothèse du monde ouvert et le fait que SWRL ne supporte pas les négations par l'échec.

Dans le premier exemple, nous définissons la *résolution de la variabilité* RV_1 comme suit :

RV_1={(F1Communication, Activé), (F11Wifi, Activé), (F12Zigbee, Activé), (F2Authentification, Activé), (F3GeoLocalisation, Activé), (F31GPS, Activé), (F32Wifi, Activé), (F4NotificationPatient, Activé), (F41PrioriteAlerte, Activé), (F411Fifo, Activé), (F412Priorite, Désactivé), (F42Alertes, Activé), (F421AvertisseurEcran, Activé), (F422AvertisseurSonore, Désactivé), (F5Alarme, Désactivé), (F51DétectionChute, Désactivé), (F52BoutonPanique, Désactivé), (F6Capteurs, Activé), (F61CapteursPhysiologique, Activé), (F611PressionArtérielle, Activé), (F612TempCorporelle, Activé), (F62CapteursEnvironnement, Désactivé), (F621TempAmbiante, Désactivé), (F622Lumiere, Désactivé), (F7PlanSoins, Désactivé)}

Nous illustrons RV_1 par la figure 5.9. Les *features* en vert représentent les *features* actifs alors que ceux en blancs sont inactifs.

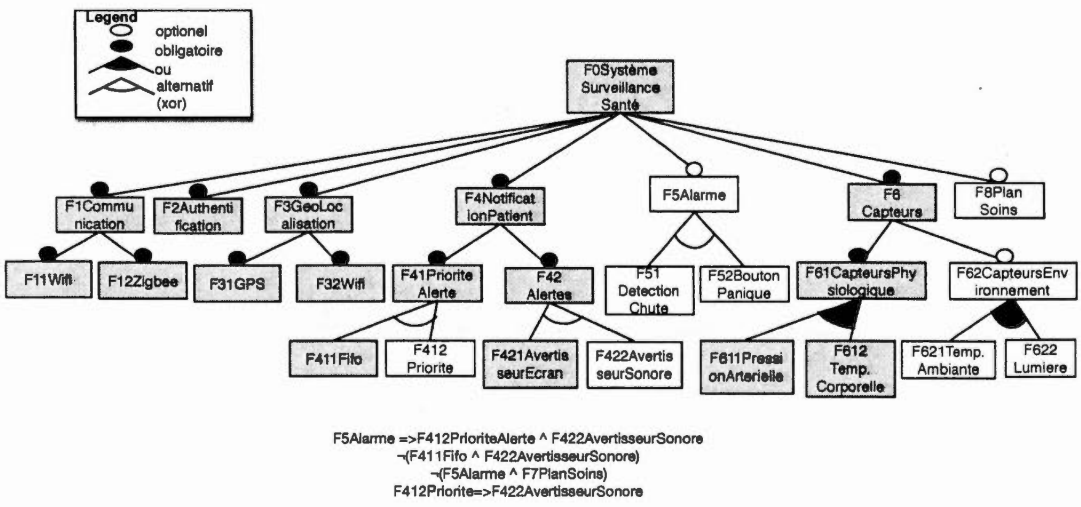


Figure 5.9 Résolution de la variabilité RV_1

Cette configuration prend en considération deux contextes : la température corporelle et la pression artérielle. Il s'agit d'une configuration (RV_1) où la santé d'un patient est à l'état normal.

Nous définissons une règle de contexte sous format SWRL qui est associé à RV_1 comme suit :

CP1(?contextePression), CT2(?contexteTempCorp), aContextePressionArterielle(?config, ?contextePression), aContexteTemperatureCorporelle(?config, ?contexteTempCorp), aPressionDiastoliqueValeur(?config, ?x), aPressionSystoliqueValeur(?config, ?y), aTempCorporelleValeur(?config, ?z), greaterThanOrEqual(?x,75), lessThanOrEqual(?x,80), greaterThanOrEqual(?y, 115), lessThanOrEqual(?y,120), greaterThanOrEqual(?z, 36.5), lessThanOrEqual(?z,38) -> F0SystemeSurveillanceSanteRegle(?config)

CP1(?contextePression) contraint les individus qui s'appliquent à la classe CP1 et assigne la variable *contextePression* à ces individus. *aContextePressionArterielle(?config, ?contextePression)* contraint les individus (*config*) correspondants à ceux qui possèdent un contexte de pression artérielle (*contextePression*). *aPressionDiastoliqueValeur(?config, ?x)* et *aPressionSystoliqueValeur(?config, ?y)* contraignent les individus (*config*) qui ont des valeurs de pression diastolique supérieure ou égale à 75 *(greaterThanOrEqual(?x,75))* et inférieure ou égale à 80 *(lessThanOrEqual(?x,80))* ainsi que des valeurs de pression systolique supérieure ou égale à 115 *(greaterThanOrEqual(?y, 115))* et inférieure ou égale à 120 *(lessThanOrEqual(?y,120))* respectivement.

CT2(?contexteTempCorp) contraint les individus qui s'appliquent à la classe CT2 et assigne la variable *contexteTempCorp* à ces individus.

aContexteTemperatureCorporelle(?config, ?contexteTempCorp) contraint les individus (*config*) correspondants à ceux qui possèdent un contexte de température corporelle (*contexteTempCorp*). *aTempCorporelleValeur(?config, ?z)* exprime que les individus *(config)* ont une valeur de température corporelle *(x)*, dont cette valeur est supérieure ou égale à 36.5 *(greaterThanOrEqual(?z, 36.5))* mais inférieure ou égale à 38 *(lessThanOrEqual(?z, 38))*. Finalement, *F0SystemeSurveillanceSanteRegle(?config)*, le conséquent, est le modèle de résolution de la variabilité répondant à ces critères de contexte.

Nous montrons un exemple d'explication de raisonnement obtenu par le moteur d'inférence Pellet pour le *feature F62CapteurPhysiologique* du modèle de résolution de variabilité RV_1 (voir figure 5.10). Nous notons que RV_1 est appelé *Conf1* dans l'ontologie et nous expliquons ce raisonnement comme suit :

- *Conf1* a une valeur de température ambiante de 37C.
- *Conf1* a une valeur de pression systolique de 120.
- *Conf1* a une valeur de pression diastolique de 80.
- *Conf1* dépend du contexte de pression artérielle PressionArterielle3.
- *Conf1* dépend du contexte de température corporelle TemperatureCorporelle2.
- TemperatureCorporelle2 est de type CT2. Les valeurs de contexte de température corporelle de CT2 sont captées par le capteur1 avec une précision élevée et appartiennent au cluster3.
- Les *features F1Communication*, *F2Authentification*, *F3GeoLocalisation*, *F4NotificationPatient* et *F6Capteurs* sont des sous-classes de *F0SystèmeSurveillanceSante*.
- La règle de contexte s'applique au modèle de résolution de variabilité RV_1, c.-à-d. la configuration *Conf1*.
- Le *feature F62CapteurPhysiologique* est une sous-classe du *feature F6Capteur*.

- PressionArterielle3 est de type CP1. Les valeurs de contexte de pression artérielle de CP1 sont captées par le capteur2 avec une précision élevée et appartiennent au cluster3.

Figure 5.10 Explication de raisonnement

Dans le second exemple, nous définissons la *résolution de la variabilité* RV_2 comme suit :

RV_2=*{(F1Communication, Activé), (F11Wifi, Activé), (F12Zigbee, Activé), (F2Authentification, Activé), (F3GeoLocalisation, Activé), (F31GPS, Activé), (F32Wifi, Activé), (F4NotificationPatient, Activé), (F41PrioriteAlerte, Activé), (F411Fifo, Désactivé), (F412Priorite, Activé), (F42Alertes, Activé), (F421AvertisseurEcran, Désactivé), (F422AvertisseurSonore, Activé), (F5Alarme, Activé), (F51DétectionChute, Activé), (F52BoutonPanique, Désactivé), (F6Capteurs, Activé), (F61CapteursPhysiologique, Activé), (F611PressionArtérielle, Activé), (F612TempCorporelle, Désactivé), (F62CapteursEnvironnement, Désactivé), (F621TempAmbiante, Désactivé), (F622Lumiere, Désactivé), (F7PlanSoins, Désactivé)}*

Nous illustrons RV_2 par la figure 5.11. Les *features* en verts représentent les *features* actifs alors que ceux en blancs sont inactifs.

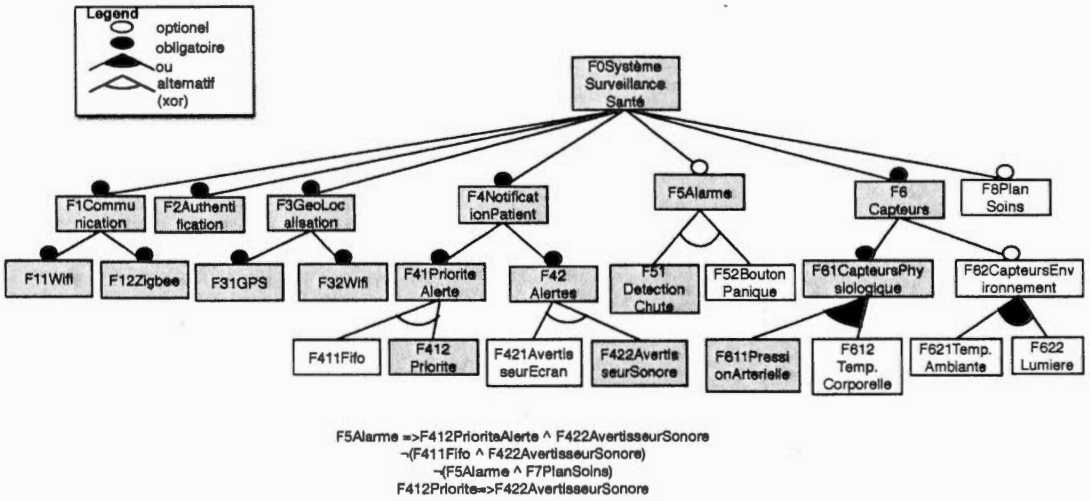


Figure 5.11 Résolution de la variabilité RV_2

Cette configuration prend en considération un seul contexte, soit la pression artérielle. Il s'agit d'une configuration (RV_2) où la pression artérielle d'un patient est basse.

Nous définissons une règle de contexte sous format SWRL qui est associé à RV_2 comme suit :

CP1(?contextePression), aContextePressionArterielle(?config, ?contextePression), aPressionDiastoliqueValeur(?config, ?x), aPressionSystoliqueValeur(?config, ?y), lessThanOrEqual(?x, 60), lessThanOrEqual(?y, 90) -> F0SystemeSurveillanceSanteRegle(?config)

CP1(?contextePression) contraint les individus qui s'appliquent à la classe CP1 et assigne la variable *contextePression* à ces individus. *aContextePressionArterielle(?config, ?contextePression)* contraint les individus (*config*) correspondants à ceux qui possèdent un contexte de pression artérielle (*contextePression*). *aPressionDiastoliqueValeur(?config, 60)* et *aPressionSystoliqueValeur(?config, 90)* contraignent les individus (*config*) qui ont les valeurs de pression diastolique de 60 et de pression systolique de 90 respectivement.

Finalement, *F0SystemeSurveillanceSanteRegle(?config)*, le conséquent, est le modèle de résolution de la variabilité répondant à ces critères de contexte.

Nous montrons un exemple d'explication de raisonnement obtenu par le moteur d'inférence Pellet pour le *feature F62CapteurPhysiologique* du modèle de résolution de variabilité RV_2 (voir figure 5.12). Nous notons que RV_2 est appelé *Conf2* dans l'ontologie et nous expliquons ce raisonnement comme suit :

- *Conf2* a une valeur de pression diastolique de 60.
- *Conf2* a une valeur de pression systolique de 90.
- *Conf2* dépend du contexte de pression artérielle PressionArterielle3.
- Les *features F1Communication*, *F2Authentification*, *F3GeoLocalisation*, *F4NotificationPatient* et *F6Capteurs* sont des sous-classes de *F0SystèmeSurveillanceSante*.
- PressionArterielle3 est de type CP1. Les valeurs de contexte de pression artérielle de CP1 sont captées par le capteur2 avec une précision élevée et appartiennent au cluster3.
- La règle de contexte s'applique au modèle de résolution de variabilité RV_2, c.-à-d. la configuration *Conf2*.

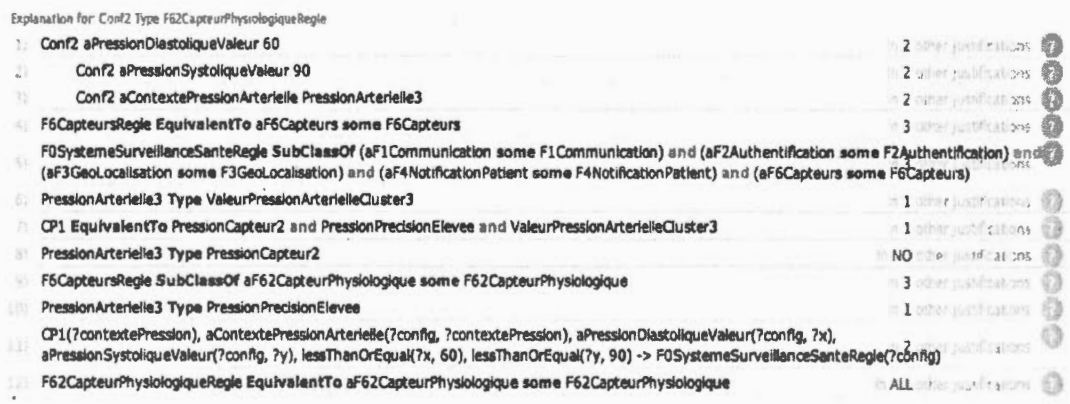


Figure 5.12 Explication de raisonnement

Dans le troisième exemple, nous définissons la *résolution de la variabilité* RV_3 comme suit :

RV_3={(F1Communication, Activé), (F11Wifi, Activé), (F12Zigbee, Activé), (F2Authentification, Activé), (F3GeoLocalisation, Activé), (F31GPS, Activé), (F32Wifi, Activé), (F4NotificationPatient, Activé), (F41PrioriteAlerte, Activé), (F411Fifo, Désactivé), (F412Priorite, Activé), (F42Alertes, Activé), (F421AvertisseurEcran, Désactivé), (F422AvertisseurSonore, Activé), (F5Alarme, Activé), (F51DétectionChute, Désactivé), (F52BoutonPanique, Activé), (F6Capteurs, Activé), (F61CapteursPhysiologique, Activé), (F611PressionArtérielle, Activé), (F612TempCorporelle, Activé), (F62CapteursEnvironnement, Désactivé), (F621TempAmbiante, Désactivé), (F622Lumiere, Désactivé), (F7PlanSoins, Désactivé)}

Nous illustrons RV_3 par la figure 5.13. Les *features* en vert représentent les *features* actifs alors que ceux en blancs sont inactifs.

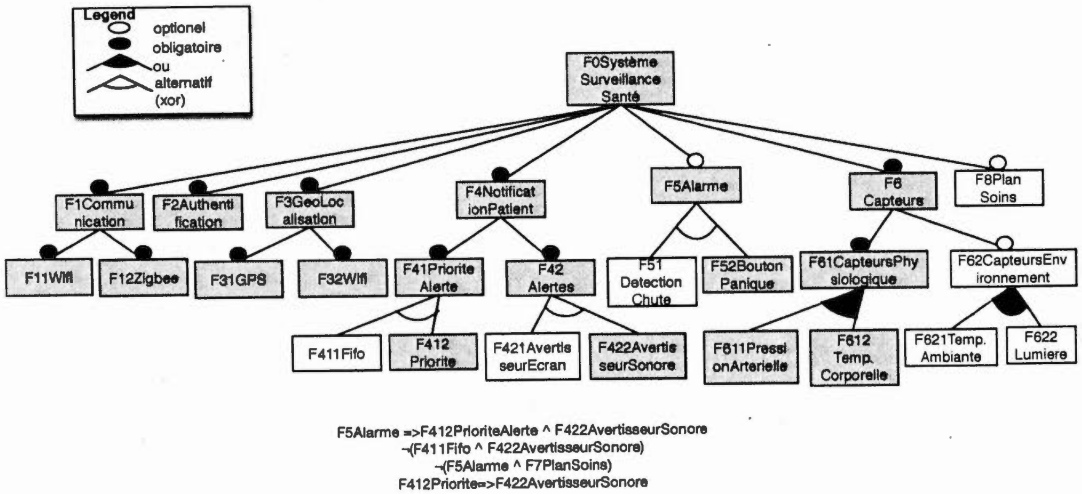


Figure 5.13 Résolution de la variabilité RV_3

Cette configuration prend en considération deux contextes : la température corporelle et la pression artérielle. Il s'agit d'une configuration (RV_3) où le patient à une fièvre et une pression artérielle élevée.

Nous définissons une règle de contexte sous format SWRL qui est associé à RV_3 comme suit :

CP1(?contextePression), CT2(?contexteTempCorp), aContextePressionArterielle(?config, ?contextePression), aContexteTemperatureCorporelle(?config, ?contexteTempCorp), aPressionDiastoliqueValeur(?config, ?x), aPressionSystoliqueValeur(?config, ?y), aTempCorporelleValeur(?config, ?z), greaterThanOrEqual(?x, 90), greaterThanOrEqual(?y, 140), greaterThanOrEqual(?z, 38) -> F0SystemeSurveillanceSanteRegle(?config)

CP1(?contextePression) contraint les individus qui s'appliquent à la classe CP1 et assigne la variable *contextePression* à ces individus. *aContextePressionArterielle(?config, ?contextePression)* contraint les individus (*config*) correspondants à ceux qui possèdent un contexte de pression artérielle (*contextePression*). *aPressionDiastoliqueValeur(?config, ?x)* et *aPressionSystoliqueValeur(?config, ?y)* contraignent les individus (*config*) qui ont des valeurs de pression diastolique supérieure ou égale à 90 *(greaterThanOrEqual(?x,90))* et une valeur de pression systolique supérieure ou égale à 140 *(greaterThanOrEqual(?y, 140))* respectivement.

CT2(?contexteTempCorp) contraint les individus qui s'appliquent à la classe CT2 et assigne la variable *contexteTempCorp* à ces individus. *aContexteTemperatureCorporelle(?config, ?contexteTempCorp)* contraint les individus (*config*) correspondants à ceux qui possèdent un contexte de température corporelle (*contexteTempCorp*). *aTempCorporelleValeur(?config, ?z)* exprime que les individus *(config)* ont une valeur de température corporelle *(x)*, dont cette valeur est supérieure ou égale à 38 *(greaterThanOrEqual(?z, 38))*. Finalement, *F0SystemeSurveillanceSanteRegle(?config)*, le conséquent, est le modèle de résolution de la variabilité répondant à ces critères de contexte.

Nous montrons un exemple d'explication de raisonnement obtenu par le moteur d'inférence Pellet pour le *feature F62CapteurPhysiologique* du modèle de résolution de variabilité RV_3 (voir figure 5.14). Nous notons que RV_3 est appelé *Conf3* dans l'ontologie et nous expliquons ce raisonnement comme suit :

- *Conf3* a une valeur de température corporelle de 40C.
- *Conf3* a une valeur de pression systolique de 140.
- *Conf3* a une valeur de pression diastolique de 90.
- *Conf3* dépend du contexte de pression artérielle PressionArterielle3.
- *Conf3* dépend du contexte de température corporelle TemperatureCorporelle2.
- La règle de contexte s'applique au modèle de résolution de variabilité RV_3, c.-à-d. la configuration *Conf3*.
- Le *features F62CapteurPhysiologique* est une sous-classe du *feature F6Capteurs*.
- Les *features F1Communication*, *F2Authentification*, *F3GeoLocalisation*, *F4NotificationPatient* et *F6Capteurs* sont des sous-classes de *F0SystèmeSurveillanceSante*.
- PressionArterielle3 est de type CP1. Les valeurs de contexte de pression artérielle de CP1 sont captées par le capteur2 avec une précision élevée et appartiennent au cluster3.
- TemperatureCorporelle2 est de type CT2. Les valeurs de contexte de température corporelle de CT2 sont captées par le capteur1 avec une précision élevée et appartiennent au cluster3.

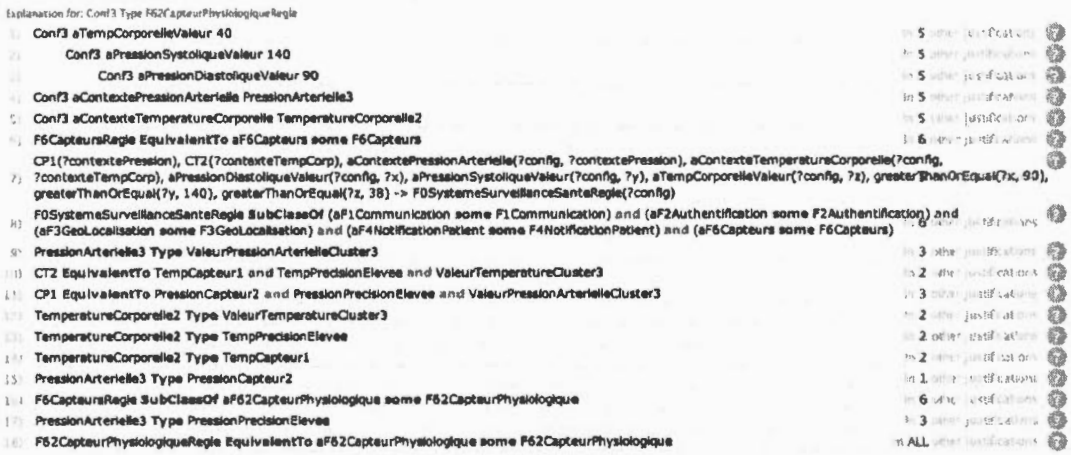


Figure 5.14 Explication de raisonnement

5.8 Manipulation du fichier ontology

Nous implémentons un programme qui vise deux objectifs distincts. D'une part, notre programme permet d'ajouter une nouvelle configuration ainsi que de modifier ou de supprimer une configuration existante d'un système dont le modèle de variabilité est modélisé en ontologie. D'autre part, le raisonneur Pellet permet de vérifier et valider la configuration ajoutée ou altérée selon les combinaisons de *features* et les contraintes du modèle de variabilité. Pour ce faire, le programme est écrit en Java et en Groovy.

Dans cette section, nous expliquons certaines parties du programme.

La figure 5.15 illustre la classe *OntologyProcessor* qui se compose de 3 méthodes. La méthode *main* spécifie où commence le programme; elle appelle les méthodes *loadProcessorScript* et *createGroovyShellWithParams,* et elle évalue le script *processor.groovy* par rapport au fichier *instruction.xml*. Le fichier *instruction.xml* est un fichier xml indiquant les *features* actifs et inactifs pour une configuration spécifique. La méthode *loadProcessorScript* initie le programme groovy que nous appelons *processor.groovy*. La méthode *createGroovyShellWithParams* permet de

créer un groovy shell capable d'exécuter arbitrairement des scripts groovy. Groovy shell est une application de ligne de commande qui permet d'évaluer les expressions Groovy, de définir les classes et d'exécuter du code.

```
package processor;

import java.io.File;
import java.nio.file.Files;
import java.nio.file.Paths;

import groovy.lang.Binding;
import groovy.lang.GroovyShell;

public class OntologyProcessor {

   static final String INSTRUCTIONS_FILE = "instructions.xml";

   public static void main(String[] args) throws Exception {
      GroovyShell scriptRunner =  createGroovyShellWithParams(INSTRUCTIONS_FILE);
      String processorScript = loadProcessorScript();
      scriptRunner.evaluate(processorScript);
   }

   private static String loadProcessorScript() throws Exception {
      return new String(Files.readAllBytes(Paths.get("processor.groovy")));
   }

   private static GroovyShell createGroovyShellWithParams(String instructionsFile) {
      Binding binding = new Binding();
      binding.setVariable("instructionsFile", new File(instructionsFile));
      return new GroovyShell(binding);
   }
}
```

Figure 5.15 Classe *OntologyProcessor*

Le programme *processor.groovy* permet entre autres d'ajouter de nouvelles configurations ou d'altérer des configurations existantes. La figure 5.16 illustre un exemple de code qui permet d'effectuer cette tâche.

```
config = readConfiguration()

owlInputFile = readInputFile()

individualNode = owlInputFile.children().find { node ->
   node.name() == 'NamedIndividual' && node.'@rdf:about' == config.individual
```

```
}

println "Found individual : ${individualNode.'@rdf:about'}"
println "Applying transformations ..."

config.instructions.each {
   println "  -> ${it.operation} ${it.value}"
   "process_${it.operation}"(it.value)
}

println "Writing output file ..."
writeOutputFile()
```

Figure 5.16 Extrait du code *processor.groovy*

Dans le programme *processor.groovy*, nous pouvons ajouter ou enlever des classes d'ontologie et la négation de classes d'ontologie, c-.à-d des *features* du modèle de variabilité, pour une configuration existante. La figure 5.17 illustre le code permettant de faire cela.

```
def process_ADD(val) {
   individualNode.type[1].'Class'.intersectionOf.appendNode {
      'owl:Restriction'() {
         'owl:onProperty'('rdf:resource':config.namespace+val)
         'owl:someValuesFrom'('rdf:resource':config.namespace+val.minus("has"))
      }
   }
}

def process_ADD_NOT(val) {
   individualNode.type[1].'Class'.intersectionOf.appendNode {
      'owl:Class' {
         'owl:complementOf' {
            'owl:Restriction'() {
               'owl:onProperty'('rdf:resource':config.namespace+val)
               'owl:someValuesFrom'('rdf:resource':config.namespace+val.minus("has"))
            }
         }
      }
   }
}

def process_REMOVE(val) {
   def nodeToRemove = individualNode.type[1].'Class'.intersectionOf.'Restriction'.find {
      it.'onProperty'.'@rdf:resource' == config.namespace + val
   }
   println nodeToRemove.'onProperty'.'@rdf:resource'
   nodeToRemove.replaceNode{}
}

def process_REMOVE_NOT(val) {
   def nodeToRemove = individualNode.type[1].'Class'.intersectionOf.'Class'.find {
      it.'complementOf'.'Restriction'.'onProperty'.'@rdf:resource' == config.namespace + val
   }
   println nodeToRemove.'onProperty'.'@rdf:resource'
   nodeToRemove.replaceNode{}
}
```

Figure 5.17 Ajout ou suppression (avec ou sans négation) de *features* pour une configuration

Les méthodes *process_ADD* et *process_REMOVE* permettent d'ajouter ou d'enlever, respectivement, des classes d'ontologie, c.-à-d. des *features* du modèle de variabilité pour une configuration existante qui est un individu en ontologie.

Par exemple, nous voulons créer une nouvelle configuration *C21* comme suit :

RV_{20}={(F1Communication, Activé), (F11Wifi, Activé), (F12Zigbee, Activé), (F2Authentification, Activé), (F3GeoLocalisation, Activé), (F31GPS, Actif),(F32Wifi, Activé), (F4NotificationPatient, Activé), (PrioriteAlerte, Activé), (F411Fifo, Activé), (F412Priorite, Désactivé), (F42Alertes, Activé), (F421AvertisseurEcran, Activé), (F422AvertisseurSonore, Désactivé), (F5Alarme, Désactivé), (F51DétectionChute, Désactivé), (F52BoutonPanique, Désactivé), (F6Capteurs, Activé), (F61CapteursPhysiologique, Activé), (F611PressionArtérielle, Activé), (F612TempCorporelle, Désactivé), (F62CapteursEnvironnement, Activé), (F621TempAmbiante, Activé), (F622Lumiere, Activé), (F7PlanSoins, Désactivé)}

Dans la figure 5.18, nous montrons la création de la configuration *C21*. Vu que la configuration C21 n'existe pas dans le modèle d'ontologie, elle est créée dans l'ontologie comme une nouvelle configuration.

```
************************************************************
Processor started using instructions file 'instructions.xml'
************************************************************
Reading test9_modified.owl ...
**** Processing individual C21
Individual not found ... creating new individual
Applying transformations ...
 -> ADD aF1Communication
 -> ADD aF11Wifi
 -> ADD aF12Zigbee
 -> ADD aF2Authentification
 -> ADD aF3GeoLocalisation
 -> ADD aF31GPS
 -> ADD aF32Wifi
 -> ADD aF4NotificationPatient
 -> ADD aF41PrioriteAlerte
 -> ADD aF411Fifo
 -> ADD aF42Alertes
 -> ADD aF42AvertisseurEcran
 -> ADD aF6Capteurs
 -> ADD aF62CapteurPhysiologique
 -> ADD aF621PressionArterielle
 -> ADD aF611CapteurEnvironnement
 -> ADD aF611TemperatureAmbiante
 -> ADD_NOT aF612Lumiere
 -> ADD_NOT aF412Priorite
 -> ADD_NOT aF5Alarme
 -> ADD_NOT aF51DetectionChute
 -> ADD_NOT aF52BoutonPanique
Writing output file ...
Process finished with exit code 0
```

Figure 5.18 Confirmation de la création de la configuration C21

Les méthodes *process_ADD_NOT* et *process_REMOVE_NOT* ajouter ou d'enlever, respectivement, la négation à une classe d'ontologie (*feature*), c.-à-d. des *features* du modèle de variabilité pour une configuration existante qui est un individu en ontologie. Par exemple, nous avons la configuration *C20* et nous voulons ajouter le *feature F612TemperatureCorporelle* et enlever le *feature F622Lumiere*.

Dans le fichier *instruction.xml*, nous indiquons le code suivant (voir figure 5.19):

```
<?xml version="1.0"?>
<configuration>
  <inputFile>test9.owl</inputFile>
  <outputFile>test9_modified.owl</outputFile>
  <namespace>http://www.semanticweb.org/marmoura/ontologies/2015/9/untitled-ontology-
100#</namespace>
  <identifier>http://www.semanticweb.org/marmoura/ontologies/2015/9/untitled-ontology-
100#C20</identifier>

<instructions>

    <instruction>
      <operation>remove_not</operation>
      <value>aF612TemperatureCorporelle</value>
    </instruction>

    <instruction>
      <operation>add_not</operation>
      <value>aF622Lumiere</value>
    </instruction>

</instructions>
</configuration>
```

Figure 5.19 Fichier *instruction.xml*

Dans le cas où une nouvelle configuration est ajoutée ou une configuration existante est modifiée, nous effectuons la vérification et validation de cette configuration en lançant le raisonneur Pellet.

5.9 Conclusion

Dans ce chapitre, nous avons proposé une approche permettant d'exprimer des situations contextuelles à partir d'un modèle d'ontologie. Nous nous sommes inspirés du modèle de variabilité pour représenter les fonctionnalités d'un système dépendant du contexte. Comme ces modèles n'offrent pas de support sémantique, nous avons représenté le modèle de variabilité et ses configurations sous forme d'ontologie OWL et nous les validons par le raisonneur Pellet. Par la suite, nous avons expliqué le lien établi entre le modèle de contexte et le modèle de variabilité. Nous avons jugé que les services peuvent offrir une ou plusieurs fonctionnalités sans nécessairement être obligés de dépendre d'un contexte particulier. De plus, nous avons présenté un

programme, écrit en Groovy, qui sert principalement à ajouter ou à modifier des configurations. Ces configurations représentent des *features* qui doivent être activés ou désactivés, et ce tout en respectant les contraintes et les dépendances définies par le modèle de variabilité. Nous avons associé ces configurations à des valeurs de contexte ou des intervalles de valeurs de contexte. Ces associations sont des règles de contexte qui représentent des situations ou des états de contexte. Pour ce faire, nous avons utilisé un langage de règles pour le Web sémantique, plus précisément SWRL. Dans ce cas-ci, nous avons également utilisé le raisonneur Pellet, car il supporte le langage SWRL. Ces règles de contexte déclenchent le besoin d'adapter une application sensible au contexte et elles détermineront la configuration requise pour un contexte donné.

Par conséquent, nous proposons deux techniques d'adaptation : une adaptation par *features* et une adaptation par composition. Étant donné qu'une adaptation d'une application dépendante du contexte nécessite l'utilisation d'une boucle de contrôle, nous jugeons qu'il est préférable d'expliquer dans le prochain chapitre notre modèle sémantique basé sur les composants avant d'entamer l'adaptation des deux techniques proposées.

CHAPITRE VI

MODÈLE SÉMANTIQUE D'ADAPTATION

6.1 Introduction

Dans ce chapitre, nous présentons une technique d'adaptation basée sur un modèle permettant non seulement d'exécuter un système sensible au contexte, mais également un modèle qui est capable d'effectuer des transformations aussi bien sur le comportement du système que sur son architecture. Une revue de la littérature n'a pas identifié de système qui intègre les deux aspects. Par exemple, les modèles basés sur l'algèbre des processus permettent, grâce à leur sémantique opérationnelle, de modéliser les architectures, mais ne permettent pas de modéliser le comportement.

Pour ces raisons, nous l'avons intégré avec une implantation faite dans un langage de programmation logique, en l'occurrence Prolog. Il faut noter cependant que l'utilisation de ce modèle là n'a pas de lien direct avec notre modélisation de contexte basée sur l'ARC. Nous aurions pu établir un lien entre la logique descriptive et Prolog, mais nous n'avons pas pu le faire par manque de temps. Le sujet d'établissement des liens entre les deux pourrait faire l'objet de recherche plus approfondie que nous n'avons pas fait.

Le modèle que nous présentons dans ce chapitre montre cependant l'intérêt qu'il offre et ouvre des perspectives, notamment lorsqu'il est combiné avec la boucle de contrôle. C'est ce que nous avons fait dans cette partie du travail.

Notre approche se base sur deux concepts importants que nous expliquons en détail : 1) les contraintes permettent d'imposer un ordre particulier dans l'exécution d'un système et 2) les transformations qui définissent quand et comment modifier une architecture. Ces deux concepts ont été intégrés dans un système de gestion des contraintes que nous présentons plus loin.

Ce système doit faire le lien entre un changement dans les valeurs du contexte et les contraintes dans le but de sélectionner les plus appropriées. Ces changements de valeurs sont observés au fur et à mesure de l'exécution du système.

La description de la structure globale de ces systèmes est importante dans le processus d'adaptation. Pour cela, nous devons être en mesure de donner une abstraction de l'organisation de ces systèmes en termes d'interconnexions de ses composants et de la description de leurs comportements.

Dans la plupart des cas, ces architectures sont décrites de manière informelle en utilisant des notations graphiques qui illustrent ces systèmes et l'interconnexion entre ses composants. Il existe par ailleurs des modèles qui décrivent les comportements des systèmes sans considération de leurs architectures. Chaque composant ayant son comportement propre et nous exprimons ses interactions avec son environnement par le biais des données échangées localement.

Notre approche consiste à utiliser une méthode qui décrit les deux aspects en même temps, à savoir l'architecture et le comportement dans un même cadre descriptif. Ces méthodes sont d'autant plus utiles qu'elles sont basées sur un modèle formel. Ce qui permet de manipuler ces descriptions et les valider.

Pour décrire le comportement de ces architectures et de ces systèmes, nous avons utilisé des règles qui expriment la sémantique des opérations que chaque composant peut effectuer et leurs effets sur le comportement global de l'ensemble.

L'utilisation de ces techniques nous a permis de faire de la validation des descriptions, d'exécuter des transformations sur ces descriptions, et de montrer l'équivalence entre descriptions et leurs transformations.

Ce chapitre est réparti comme suit. La section 6.2 décrit les aspects d'une architecture basée sur des composants, à savoir : les composants, les ports et les mécanismes de composition. Dans la section 6.3, nous montrons un exemple général d'un système composé de composants. Nous décrivons l'utilisation des contraintes dans la section 6.4. La section 6.5 explique notre système de transitions étiqueté. Nous montrons des exemples de composition dans la section 6.6. Par la suite, nous expliquons respectivement l'exécution de séquences et la substitution d'opérateurs dans les sections 6.7 et 6.8. Enfin, la section 6.9 présente l'implantation du modèle sémantique en Prolog.

6.2 Description des architectures

Pour décrire une architecture, nous pouvons recourir à plusieurs types de formalismes. Nous avons opté pour un modèle basé sur l'utilisation de l'algèbre des processus (Fokkink, 2007; Milner, 1980). La sémantique ce cette algèbre est décrite à l'aide de règles sémantiques qui définissent l'effet que produit chaque action d'un composant sur le comportement de l'ensemble.

Le contexte décrit alors des changements dans l'environnement du système qui ont une influence sur la façon dont le système se comporte.

6.2.1 Les composants

Un composant est décrit comme une entité qui contient un comportement interne et deux ensembles de ports : les ports d'entrées *inP* et les ports de sorties *outP*. Le composant est décrit par une boîte noire qui n'est accessible qu'à travers ces ports.

L'exemple de la figure 6.1 présente un composant qui offre un service de transport pour un patient qui a besoin d'assistance. Ce composant offre quatre ports d'entrées et trois ports de sorties : *inP(Transportation)={adresse, temps, destination, condition}* et *outP(Transportation)={itinéraire, coût, véhicule}*

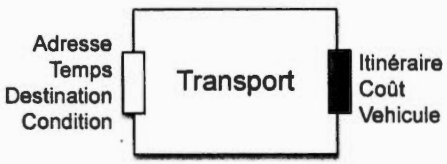


Figure 6.1 Exemple de composant

6.2.2 Les ports

Les composants du système sont reliés par les ensembles de ports d'entrées (*inP*) ou de sorties (*outP*). Chaque port est identifié grâce à son étiquette et des types de données qui le traversent. Les types décrivent les valeurs ou les paramètres qui transitent à travers le port. Certains ports peuvent contenir une horodate ou une condition sur les valeurs échangées.

6.2.2.1 Échanges de données et synchronisation

Définition 6.1 Évènement. Un *évènement* d'un composant correspond à une activité d'échange de données qui a lieu sur ses ports. Il est constitué d'un port d'étiquette p et accompagné des variables ou de valeurs qui transitent en même temps à travers ce port:

- L'envoi d'une valeur v à travers un port d'étiquette p est exprimé par la fonction $send(p, v, TS)$. TS correspond à l'horodate de cet envoi. Il représente l'heure à laquelle cet envoi a eu lieu.

- La réception d'une donnée sur le port d'étiquette p qui doit être affectée à la variable x de type t est exprimée par la fonction $recv(p, x: t, C)$. C est la condition qui s'applique à la valeur reçue. Il faut qu'elle soit vraie pour que la donnée soit acceptée par le composant et reçue à travers ce port.

Définition 6.2 Synchronisation. La *synchronisation* de deux évènements λ_1 et λ_2 de deux composants donne un évènement λ qui obéit aux règles suivantes:

- Si $\lambda_1 = send(p, v, TS)$ et $\lambda_2 = recv(p, x: t, C)$

- Alors $sync(\lambda_1, \lambda_2) = send(p, v, TS)$ et $x := v;$ si $t = type(v)$ et C est vraie.

- Pour simplification, nous écrivons $sync(\lambda_1, \lambda_2) = \lambda$

6.2.3 Les mécanismes de composition

Pour exprimer le fait qu'un composant exécute une action λ et se transforme en C', nous utilisons la notation:

$$C \xrightarrow{\lambda} C' \tag{6.1}$$

Cela veut dire que si le contexte du composant C fait en sorte que l'évènement λ se produise, alors ce composant se transforme en composant C'.

Les composants peuvent être interconnectés entre eux de diverses manières pour exprimer diverses configurations. Ils peuvent être mis en parallèle, en séquence, etc.

Ils peuvent également être invoqués grâce à des opérations d'entrée-sortie à travers leurs ports.

Dans ce qui suit, nous décrivons chacune de ces interconnexions exprimées au moyen de règles sémantiques qui décrivent ces opérateurs.

6.2.3.1 Parallèle

Ce type de composition exprime le fait que deux composants s'exécutent en parallèle et peuvent à un moment donné communiquer entre eux de manière synchrone. Après une synchronisation, chaque composante peut continuer de manière autonome. Cette communication s'accompagne d'un échange de données à travers des ports que ces composants ont en commun. Ce type de composition s'appelle *parallèle*.

Dans l'exemple de la figure 6.2, nous montrons deux composants connectés à travers un port qui fait partie de l'ensemble des ports *outP* du premier composant et de l'ensemble des ports *inP* du deuxième composant. Ce port est utilisé pour faire échanger des données représentées par les valeurs de x et y lues par le composant *Urgence* et qui correspondent aux valeurs de la latitude et la longitude fournies par le composant *Géolocalisation*.

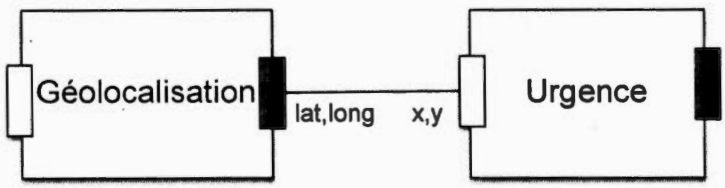


Figure 6.2 Exemple de composition parallèle

Nous utilisons l'opérateur || pour indiquer cette composition. Cet opérateur exprime le fait qu'une action peut être prise de l'un des composants qui forment cette composition. Si λ est une action qui a été exécutée par un des composants, alors l'autre composant reste entier. Également, il est possible que des actions se synchronisent auquel cas, chacun des composants progresse vers son état suivant.

Ce comportement est exprimé par les trois règles suivantes:

$$C1 \,||\, C2 \xrightarrow{\lambda} C1' \,||\, C2 \text{ si } C1 \xrightarrow{\lambda} C1' \tag{6.2}$$

$$C1 \,||\, C2 \xrightarrow{\lambda} C1 \,||\, C2' \text{ si } C2 \xrightarrow{\lambda} C2' \tag{6.3}$$

$$C1 \,||\, C2 \xrightarrow{\lambda} C1' \,||\, C2' \text{ si } C1 \xrightarrow{\lambda_1} C1' \text{ et } C2 \xrightarrow{\lambda_2} C2' \text{ et } \lambda = synch(\lambda_1, \lambda_2) \tag{6.4}$$

Une version de cet opérateur notée |||, n'utilise que les deux premières règles. Elle exprime un comportement sans synchronisation.

$$C1 \,|||\, C2 \xrightarrow{\lambda} C1' ||| \, C2 \text{ si } C1 \xrightarrow{\lambda} C1' \tag{6.5}$$

$$C1 \,|||\, C2 \xrightarrow{\lambda} C1 \,||| \, C2' \text{ si } C2 \xrightarrow{\lambda} C2' \tag{6.6}$$

6.2.3.2 Alternative

Pour décrire une situation qui correspond à des composants qui coexistent à la manière d'un choix (ou alternative), nous utilisons un autre type de connexion appelée *alternative*.

Dans l'exemple de la figure 6.3, nous présentons deux composants dont l'un ou l'autre peut s'exécuter. Soit nous offrons une assistance médicale locale à un patient, soit nous le transportons (ex. vers un centre de soins).

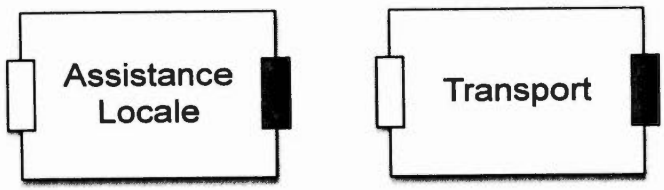


Figure 6.3 Exemple de composition alternative

Nous utilisons l'opérateur + pour indiquer une composition alternative. Cet opérateur exprime le fait qu'une action peut être prise de l'un des composants qui forment l'alternative. Si λ est une action qui a été exécutée par un des composants alors l'autre est simplement ignoré.

Les règles qui définissent cet opérateur sont les suivantes:

$$C1 + C2 \xrightarrow{\lambda} C1' \text{ si } C1 \xrightarrow{\lambda} C1' \tag{6.7}$$

$$C1 + C2 \xrightarrow{\lambda} C2' \text{ si } C2 \xrightarrow{\lambda} C2' \tag{6.8}$$

6.2.3.3 Pipeline (Séquence)

Un autre type de composition exprime une séquence entre composants. Ce type de comportement s'apparente à celui d'un *pipeline*.

Un deuxième composant ne peut s'exécuter que si l'exécution du composant qui le précède est complétée. Dans cette opération, il faut que le premier composant exécute une action sur un port de sortie qui fait partie des ports d'entrées du deuxième composant.

Cette composition importe un ordre strict sur l'ordre de déroulement des activités des composants du système.

Nous utilisons l'opérateur * pour indiquer une composition séquentielle.

$$C1 * C2 \xrightarrow{\lambda} C1' * C2 \; si \; C1 \xrightarrow{\lambda} C1' \; et \; \lambda \in inP(C1) \tag{6.9}$$

$$C1 * C2 \xrightarrow{\lambda} C2' \; if \; C1 \xrightarrow{\lambda} C1' \; et \; C2 \xrightarrow{\lambda} C2' \; et \; \lambda \in outP(C1) \cap inP(C2) \tag{6.10}$$

Dans l'exemple de la figure 6.4, nous composons trois composants dans un pipeline. Le premier est le service de localisation, le deuxième est le service de transport et le dernier est un service d'assistance médicale.

Figure 6.4 Exemple de composition séquentielle

6.2.3.4 Conditionnelle

Nous pouvons effectuer des activations conditionnelles de composants. La condition peut porter sur des variables et des valeurs échangées entre composants. Ces valeurs peuvent provenir de divers types de sources de données (ex. des capteurs).

Dans l'exemple de la figure 6.5, nous déclenchons un service de transport dans des bonnes conditions de trafic routier.

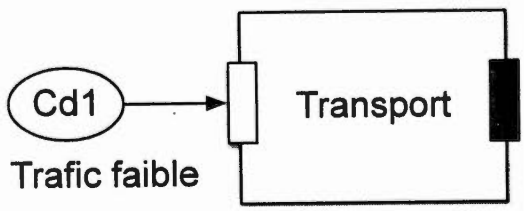


Figure 6.5 Exemple de composition conditionnelle

Cet opérateur spécifie que le composant n'est activé que si la condition est évaluée à *true*. Il est exprimé par la règle ci-dessous.

$$E -> C \xrightarrow{\lambda} C1' \text{ si } C1 \xrightarrow{\lambda} C1' \text{ et } E = true \quad (6.11)$$

6.2.3.5 Invocation

Cet opérateur désigne l'invocation d'un composant. Dans cette invocation, nous ne nous intéressons pas à l'exécution du composant, mais seulement à l'activation de ses ports d'entrées et de sorties. Le résultat de cette invocation est une séquence d'opérations d'entrées/sorties qui débute par des opérations sur les ports d'entrées et se termine par des opérations sur les ports de sorties.

Cette opération est illustrée par la figure 6.6.

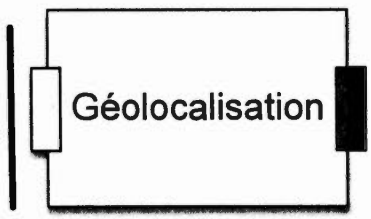


Figure 6.6 Exemple d'invocation

Pour invoquer un composant, nous devons l'appeler avec l'opérateur *Call()*. Cet opérateur fournit une sorte d'enveloppe pour le composant.

Il est défini grâce aux règles suivantes:

$$Call(C(inP, outP)) \xrightarrow{\lambda} C(inP \backslash \lambda, outP) \; si \; \lambda \in inP \tag{6.12}$$

$$Call(C(\emptyset, outP)) \xrightarrow{\lambda} C(\emptyset, outP \backslash \lambda) \; si \; \lambda \in outP \tag{6.13}$$

Notons que pour abréger l'écriture, parfois, nous écrivons P(inP, outP) au lieu de Call(P(inP, outP)).

6.2.3.6 Encapsulation

Dans certaines circonstances, nous voulons encapsuler un ou plusieurs composants afin de les cacher du monde extérieur. Ils peuvent communiquer à travers certains de leurs ports. Cette encapsulation cache certains ports. Ce comportement est exprimé avec l'opération *hide*.

Dans l'exemple de la figure 6.7, nous montrons certains ports d'entrées et nous cachons tous les ports de sorties.

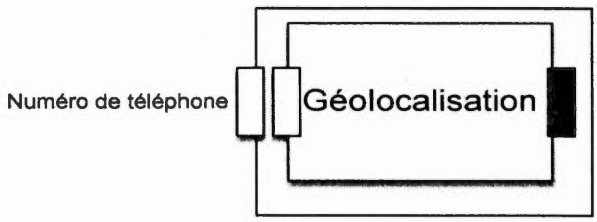


Figure 6.7 Exemple d'encapsulation

L'opérateur *hide* est exprimé par la règle ci-dessous.

$$C \; hide \; L \xrightarrow{\lambda} C' \; hide \; L \text{ si } C \xrightarrow{\lambda} C' \text{ et } \lambda \notin L. \qquad (6.14)$$

L est un ensemble de ports (d'entrée ou de sortie).

Notons que dans cet opérateur les actions de synchronisation sont toujours cachées. Ce qui veut dire que si une action est de la forme *synch(...)*, elle ne fait pas partie de l'ensemble L par défaut.

6.3 Exemple général

Un exemple plus général est donné ci-dessous (voir figure 6.8). Le système est composé de deux composants P1 et P2.

a) P1 contient deux éléments composés sous forme d'alternatives: Le premier est un service qui effectue la géolocalisation du patient. Ce composant est contraint par une condition qui pourrait correspondre à l'existence de mécanismes pour effectuer cette localisation (exemple existence d'un réseau de téléphonie mobile, d'un système GPS ou d'un réseau Wifi). Si aucune de ces méthodes n'est disponible, nous utilisons un composant appelé *Neigh* (pour *Neighbour*) qui donne l'emplacement le plus proche de l'endroit ou se trouve le malade dans lequel il peut se faire traiter.

b) P2 est composé de deux composants qui sont en parallèle:

- P21 offre le choix entre deux composants: a) un appelé *Pressure* qui effectue le traitement de la pression artérielle du patient. Ce composant est contraint par une condition appelée C3 qui pourrait vérifier si cette pression est supérieure à une valeur (exemple 160 pour la valeur de pression systolique), b) un deuxième qui s'appelle *Temperature* qui traite de la température corporelle. Il est déclenché si la condition C4 est vraie. Cette condition pourrait correspondre à une température corporelle supérieure à 38°C par exemple.

- Un composant, appelé *MedProfile,* traite le profil médical du patient. Ce composant pourrait consulter le dossier du patient et également le mettre à jour si nécessaire.

Les deux composants P1 et P2 sont composés sous forme de Pipeline. Ce qui signifie que pour effectuer le traitement il faut que les mécanismes de la localisation soient activés afin de décider de l'endroit où nous devons traiter le patient.

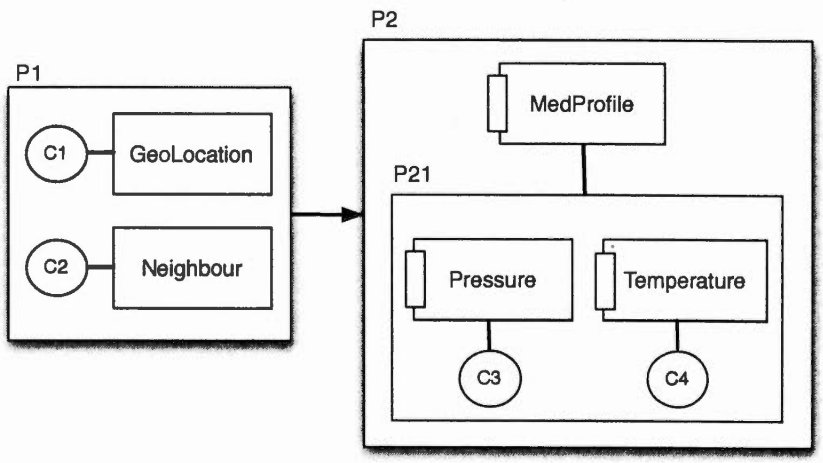


Figure 6.8 Exemple

L'expression correspondant à l'exemple est la suivante: P = P1 * P2 avec

$$P1 = call(MedProfile) \,||\, (C1 \rightarrow call(GeoLocation) + C2 \rightarrow call(Neighbour)$$

$$P21 = call(MedProfile) \,||\, (C3 \rightarrow call(Pressure) + C4 \rightarrow call(Temperature)$$

Les ports de ces composants sont donnés dans la figure 6.9.

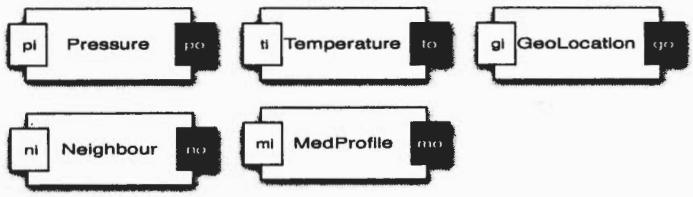


Figure 6.9 Ports des composants

6.4 Les contraintes

Une contrainte est décrite à l'aide d'une expression qui correspond à un comportement spécifique. Elle est considérée comme un composant dont le rôle est de restreindre le comportement d'autres composants.

Ce mécanisme nous permettra d'appliquer une approche qui consiste à définir une architecture globale et, en fonction du contexte, lui ajouter des contraintes pour l'adapter en le restreignant à la situation correspondant au contexte courant.

Nous utilisons l'opérateur < pour exprimer une contrainte. Il est défini par la règle suivante :

$$C1 < C2 \xrightarrow{\lambda} C1' < C2' \; si \; C1 \xrightarrow{\lambda} C1' \; et \; C1 \xrightarrow{\lambda} C2' \qquad (6.15)$$

Cet opérateur signifie que les deux composants doivent absolument se synchroniser pour avancer à l'étape suivante.

Pour représenter graphiquement une contrainte, nous utilisons une représentation telle qu'illustrée dans la figure 6.8. L'exemple, illustré à la figure 6.10, contraint le comportement du composant *Géolocalisation* à celui du composant *Sécurité*. Ce dernier est considéré comme une directive à suivre par le composant *Géolocalisation*. Ce qui veut dire que le comportement de Géolocalisation est contraint par le comportement du composant *Sécurité*.

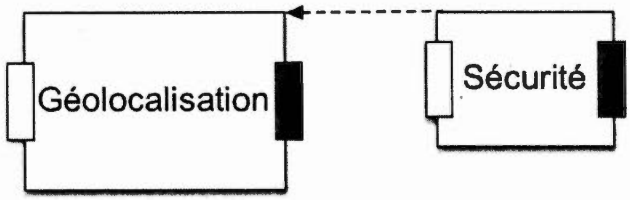


Figure 6.10 Exemple de contrainte

Nous montrons un autre exemple, illustré dans la figure 6.11, qui impose des contraintes qui consistent à exiger des éléments de sécurité et protéger des informations personnelles comme le montre l'exemple de la figure 6.11.

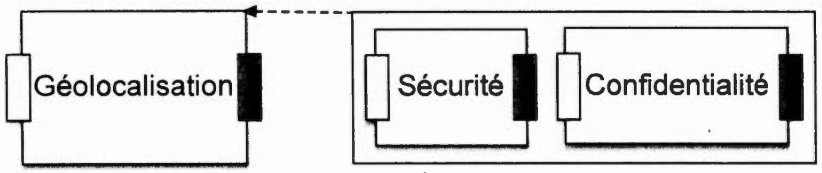


Figure 6.11 Exemple de contraintes

Pour exprimer des contraintes, nous pouvons utiliser n'importe quel opérateur décrit plus haut. Cependant, nous avons besoin parfois d'exprimer des contraintes qui sont construites à l'aide de séquence d'actions. Pour cela, nous devons définir un autre opérateur de séquence noté ; qui permet d'exprimer une séquence entre une action et un autre composant, selon le format:

$$\lambda; P \tag{6.16}$$

P étant une expression quelconque qui peut être le composant nul appelé *null*. Ceci nous permet d'exprimer des contraintes telles que:

1. *a;(b;null + c;d; call(P{e,f},{g})*
2. *a;b.null + e;f;null*
3. *a;(b;c;null ||| d;(e;null + f;null))*

À titre d'exemple, dans le premier cas, le composant exécuter l'action *a* et seulement après il peut exécuter *b* ou *c*. S'il exécute *b*, il a terminé. Sil exécute *c*, il doit exécuter *d* et ensuite invoquer la composante P sur ces ports d'entrée et de sortie.

Pour illustrer nos exemples, nous avons utilisé des arbres d'exécution qui seront présentés dans la section 6.6.

La règle sémantique décrivant cet opérateur est:

$$\lambda; P \xrightarrow{\lambda} P \tag{6.17}$$

Le composant *null* n'a pas de règle. Il forme un élément terminal.

6.4.1 Gestion des contraintes

Les contraintes peuvent s'appliquer en cascades. Nous partons d'une architecture globale que nous raffinons avec l'utilisation de contraintes.

Les contraintes peuvent impliquer des conditions sur les données échangées grâce à l'opérateur < (Équation 6.15).

Un exemple est illustré à la figure 6.12. La condition *cd1* s'applique au composant *C1* et la contrainte *co1* s'applique à l'ensemble du système. La contrainte *co2* s'applique sur l'ensemble global du système, et ce après avoir imposé la contrainte *co1*.

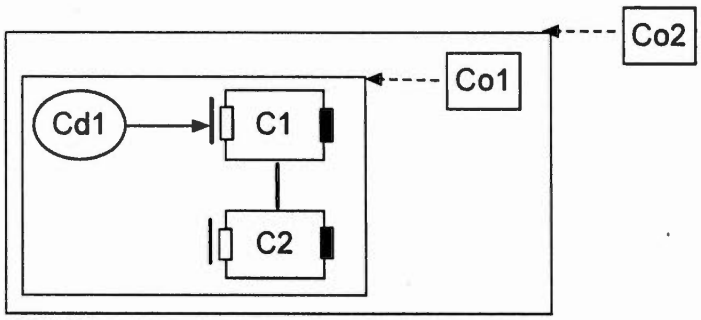


Figure 6.12 Application de contraintes

L'exemple précédent est équivalent à la figure 6.13.

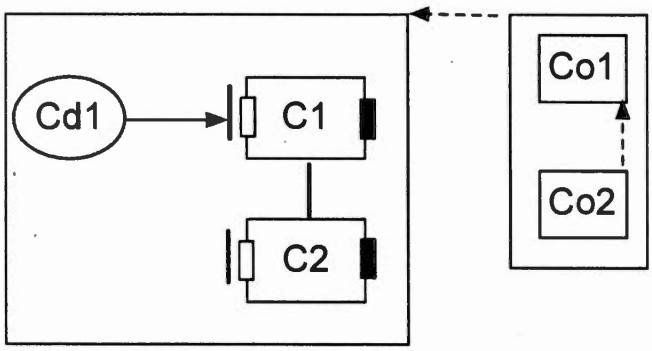


Figure 6.13 Contraintes en cascade

Nous pouvons également utiliser un système de *Gestion du contexte* qui permet de gérer des contraintes. Ce système consisterait à répertorier les contraintes, les découvrir et les appliquer.

Le *Système de gestion du contexte* est composé des modules principaux suivants (Figure 6.14):

1. Le module de découverte détermine (grâce à des règles) les contraintes relatives à un contexte et un état. Elle détermine les contraintes à appliquer en fonction de la valeur courante du contexte.

2. Le module d'application permet d'appliquer les contraintes sélectionnées et les intégrer dans une architecture de contraintes dans le cadre d'un processus d'exécution.

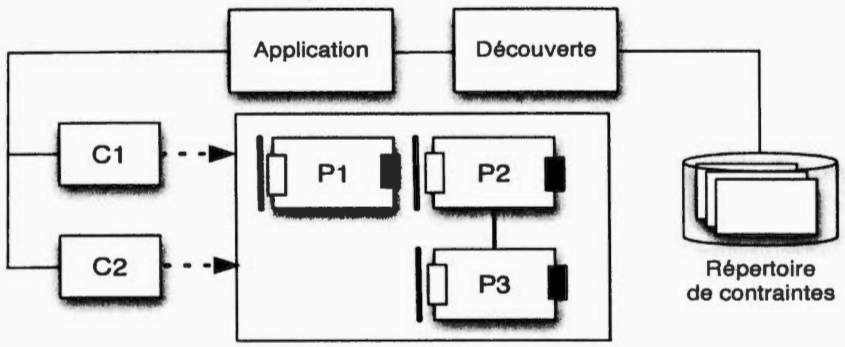


Figure 6.14 Adaptation basée sur des contraintes

6.5 Système de transitions étiqueté

Définition 6.3 Système de transitions ST. Une architecture exprimée sous forme de graphe représente ce que nous appelons *un système de transitions ST* défini comme un tuple

$$ST = (N, \lambda, s0, tr) \tag{6.18}$$

N est un ensemble de nœuds (états). λ est un ensemble d'arcs (ou d'actions). $s0$ est un nœud racine (état initial). tr est une fonction de transition exprimée comme suit:

$$tr\colon N \; x \; \lambda \; -> \; N \tag{6.19}$$

Par exemple, $tr(n_1, \lambda) = n_2$ indique qu'à partir du nœud n_1, nous transitons vers le nœud si l'action λ est exécutée.

Définition 6.4 Expression. Soit un nœud n_1 dans ST, nous définissons $expr(n_1)$ qui décrit le comportement qui lui est associé. Nous construisons une transition à partir

de n_1, en utilisant les règles d'inférence exprimées dans la section précédente comme suit :

$$tr(n_1, \lambda) = n_2 \text{ si } expr(n_1) \xrightarrow{\lambda} expr(n_2) \quad (6.20)$$

$expr(x)$ est l'expression représentant par le nœud n.

Par exemple, considérons trois nœuds n_1, n_2 et n_3 et leurs expressions comme suit:

$expr(n_1) = call(P1(\{a\}, \{b\})) \,||\, call(P2(\{b\}, \{d\}))$,

$expr(n_2) = call(P1(\{\}, \{c\})) \,||\, call(P2(\{b\}, \{d\}))$,

$expr(n_3) = call(P1\{a\}, \{b\})) \,||\, call(P2(\{\}, \{d\}))$

Nous obtenons $tr(n_1, a) = n_2, tr(n_1, b) = n_3$ en appliquent les règles suivantes :

- Règles de composition:
  - $C1 \,||\, C2 \xrightarrow{\lambda} C1' || C2$ si $C1 \xrightarrow{\lambda} C1'$
  - $C1 \,|| C2 \xrightarrow{\lambda} C1 \,|| C2'$ si $C2 \xrightarrow{\lambda} C2'$
- Règles d'invocation:
  - $Call(C(inP, outP)) \xrightarrow{\lambda} Call(C(inP \backslash \lambda, outP)) si\ \lambda \in inP$
  - $Call(C(\emptyset, outP)) \xrightarrow{\lambda} Call(C(\emptyset, outP \backslash \lambda))\ si\ \lambda \in outP$

Le ST avec ces trois nœuds est représenté dans la figure 6.15.

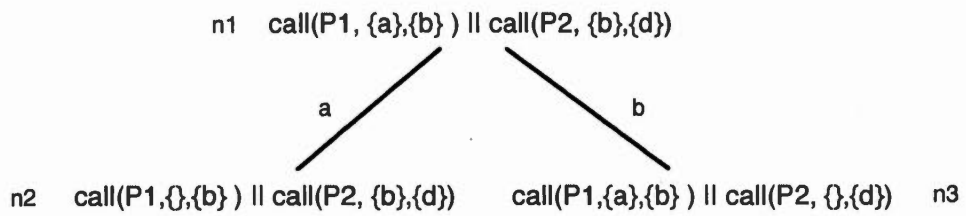


Figure 6.15 Exemple de transitions

6.6 Exemples de compositions

Dans cette section, nous montrons quelques exemples de mécanismes de composants définis précédemment ainsi que leurs comportements. En utilisant les règles des opérateurs, nous parvenons à transformer ce comportement sous forme d'alternatives et de branchement. Cela donne un arbre dont les liens sont étiquetés par les actions exécutées.

L'exemple représenté par la figure 6.16 montre une composition d'invocation, soit P1({a,b}, {c,d}) où P1 représente un composant avec ses ports d'entrées [a,b] et de sorties [c,d] respectifs.

Nous appliquons d'abord le mécanisme de composition d'*invocation* grâce à la règle 6.12, et ce qui nous donne initialement 2 choix possibles : l'activation du port d'entrée *a* ou l'activation du port d'entrée *b*.

a) En choisissant le port d'entrée *a*, la prochaine expression est P1({b}, {c,d}). Nous réappliquons la règle 6.12 en choisissant le port d'entrée b, et en conséquence, nous obtenons l'expression P1({}, {c,d}). Ensuite, nous appliquons la règle 6.13 soit sur le port de sortie *c* qui nous donnera l'expression P1({}, {d}), soit sur le port de sortie *d* qui nous donnera l'expression P1({}, {c}). Peu importe le port de sortie choisi, nous réappliquons la règle 6.13 sur les ports de sortie *d* et *c* et nous obtenons les dernières expressions possibles, soit P1({}, {}) et P1({}, {}) respectivement.

b) En choisissant le port d'entrée *b*, la prochaine expression est P1({a}, {c,d}). Nous réappliquons la règle 6.12 en choisissant le port d'entrée a, et en conséquence, nous obtenons l'expression P1({}, {c,d}). Ensuite, nous appliquons la règle 6.13 soit sur le port de sortie *c* qui nous donnera l'expression P1({}, {d}), soit sur le port de sortie *d* qui nous donnera l'expression P1({}, {c}). Peu importe le port de sortie choisi, nous réappliquons la règle 6.13 sur les ports de sortie *d* et *c* et nous obtenons les dernières expressions possibles, soit P1({}, {}) et P1({}, {}) respectivement.

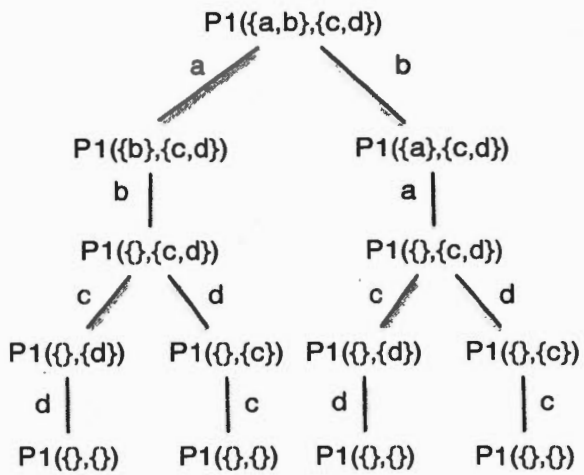


Figure 6.16 Exemple *call*

L'exemple représenté par la figure 6.17 montre une composition alternative, soit P1({a,b}, {c,d}) + P2({e,f}, {g}) où P1 et P2 représentent des composants avec leurs ports respectifs. Les ports {a,b} et {e,f} sont des ensembles de ports d'entrée et les ports {c,d} et {g} sont des ensembles de ports de sortie.

Nous appliquons d'abord le mécanisme de composition d'*alternatif* (voir règle 6.7 et 6.8) et d'*invocation* (voir la règle 6.12 et 6.13), et ce qui nous donne initialement 4 choix possibles : l'activation du port d'entrée *a* de P1, l'activation du port d'entrée *b* de P1, l'activation du port d'entrée *e* de P2 et l'activation du port d'entrée *f* de P2.

a) Nous choisissons la règle 6.7 du mécanisme de composition *alternatif*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *a* de P1, la prochaine expression est P1({b}, {c,d}). Nous réappliquons la règle 6.12 en choisissant le port d'entrée *b*, et en conséquence, nous obtenons l'expression P1({}, {c,d}). Ensuite, nous appliquons la règle 6.13 sur le port de sortie *c* qui nous donnera l'expression P1({}, {d}), soit sur le port de sortie *d* qui nous donnera l'expression P1({}, {c}). Peu importe le port de sortie choisi, nous réappliquons la règle 6.13 sur les ports de sortie *d* et *c* et nous obtenons les dernières expressions possibles, soit P1({}, {}) et P1({}, {}) respectivement.

b) Nous choisissons la règle 6.7 du mécanisme de composition *alternatif*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *b* de P1, la prochaine expression est P1({a}, {c,d}). Nous réappliquons la règle 6.12 en choisissant le port d'entrée *a*, et en conséquence, nous obtenons l'expression P1({}, {c,d}). Ensuite, nous appliquons la règle 6.13 sur le port de sortie *c* qui nous donnera l'expression P1({}, {d}), soit sur le port de sortie *d* qui nous donnera l'expression P1({}, {c}). Peu importe le port de sortie choisi, nous réappliquons la règle 6.13 sur les ports de sortie *d* et *c* et nous obtenons les dernières expressions possibles, soit P1({}, {}) et P1({}, {}) respectivement.

c) Nous choisissons la règle 6.8 du mécanisme de composition *alternatif*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *e* de P2, la prochaine expression est P2({f}, {g}). Nous réappliquons la règle 6.12 en choisissant le port d'entrée *f*, et en conséquence, nous obtenons l'expression P1({}, {g}). Ensuite, nous appliquons la règle 6.13 sur le port de sortie *g* qui nous donnera l'expression P1({}, { }).

d) Nous choisissons la règle 6.8 du mécanisme de composition *alternatif*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *f* de P2, la prochaine expression est P2({e}, {g}). Nous

réappliquons la règle 6.12 en choisissant le port d'entrée *e*, et en conséquence, nous obtenons l'expression P1({}, {g}). Ensuite, nous appliquons la règle 6.13 sur le port de sortie *g* qui nous donnera l'expression P1({}, { }).

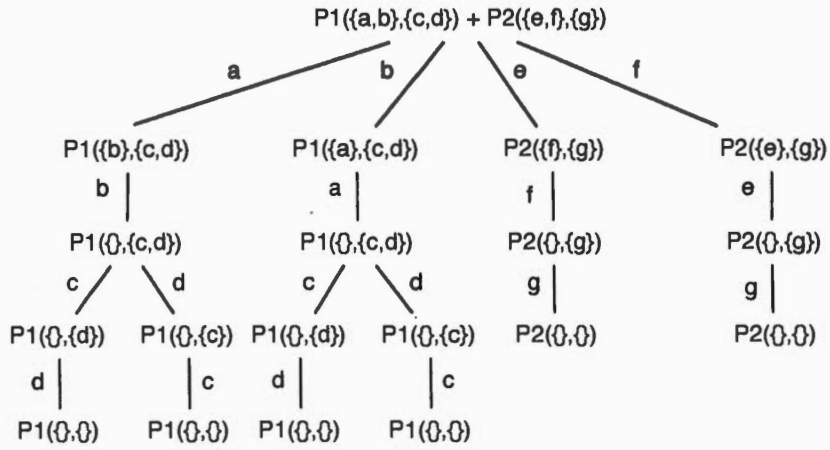


Figure 6.17 Exemple *alternatif*

L'exemple représenté par la figure 6.18 montre une composition séquentielle, soit P1({a,b}, {c,d}) * P2({c,d}, {g}) où P1 et P2 représentent des composants avec leurs ports respectifs. Les ports {a,b} et {c,d} sont des ensembles de ports d'entrée et les ports {c,d} et {g} sont des ensembles de ports de sortie.

Nous appliquons d'abord le mécanisme de composition en pipeline (voir règle 6.9 et 6.10) et d'*invocation* (voir la règle 6.12 et 6.13), et ce qui nous donne initialement 2 choix possibles : l'activation du port d'entrée *a* de P1 et l'activation du port d'entrée *b* de P2.

a) Nous choisissons la règle 6.9 du mécanisme de composition en pipeline. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *a* de P1, la prochaine expression est P1({b}, {c}) * P2({c}, {g}). Nous réappliquons la règle 6.12 en choisissant le port d'entrée *b*, et en conséquence, nous obtenons l'expression P1({}, {c}) * P2({c}, {g}). Ensuite,

nous appliquons la règle 6.10 sur le port de sortie *c*. Cette règle nécessite la règle qui nous donnera l'expression P2({}, {g}). Nous réappliquons la règle 6.13 sur le port de sortie *g* et nous obtenons P2({}, {}).

b) Nous choisissons la règle 6.9 du mécanisme de composition en pipeline. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *b* de P1, la prochaine expression est P1({a}, {c}) * P2({c}, {g}). Nous réappliquons la règle 6.12 en choisissant le port d'entrée *a*, et en conséquence, nous obtenons l'expression P1({}, {c}) * P2({c}, {g}). Ensuite, nous appliquons la règle 6.10 sur le port de sortie *c*. Cette règle nécessite la règle 6.13 qui nous donnera l'expression P2({}, {g}). Nous réappliquons la règle 6.13 sur le port de sortie *g* et nous obtenons P2({}, {}).

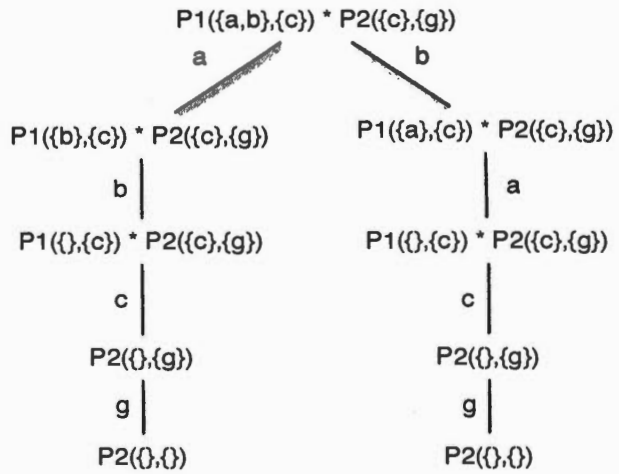


Figure 6.18 Exemple *séquence*

La figure 6.19 montre la transformation d'une expression utilisant la composition parallèle, soit call(P1({a,b},{}) || P2({c,d},{}) où P1 et P2 représentent des composants avec leurs ensembles de ports d'entrée {a,b} et {c,d} respectivement.

Nous appliquons d'abord le mécanisme de composition *parallèle* (voir règle 6.2 et 6.3) et d'*invocation* (voir la règle 6.12 et 6.13), et ce qui nous donne initialement 3

choix possibles : l'activation du port d'entrée *a* de P1, l'activation du port d'entrée *b* de P1 et l'activation du port d'entrée *c* de P2.

a) Nous choisissons la règle 6.2 du mécanisme de composition *parallèle*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *a* de P1, la prochaine expression est P1({b}, {}) || P2({c}, {}). Nous réappliquons la règle 6.12 en choisissant soit le port d'entrée *b* (voir règle 6.2), soit le port d'entrée *c* (voir règle 6.3), et en conséquence, nous obtenons respectivement les expressions P1({}, {}) || P2({c}, {}) et P1({b}, {}) || P2({}, {}). Dans le cas où nous choisissons le port d'entrée *b*, nous réappliquons les règles 6.2 et 6.12 à partir du port d'entrée *c* pour obtenir l'expression P1({}, {}) || P2({}, {}). Dans le cas contraire où nous choisissons le port d'entrée *c*, nous réappliquons les règles 6.3 et 6.12 à partir du port d'entrée *b* pour obtenir l'expression P1({}, {}) || P2({}, {}).
b) Nous choisissons la règle 6.2 du mécanisme de composition *parallèle*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *b* de P1, la prochaine expression est P1({a}, {}) || P2({c}, {}). Nous réappliquons la règle 6.12 en choisissant soit le port d'entrée *a* (voir règle 6.2), soit le port d'entrée *c* (voir règle 6.3), et en conséquence, nous obtenons respectivement les expressions P1({}, {}) || P2({c}, {}) et P1({a}, {}) || P2({}, {}). Dans le cas où nous choisissons le port d'entrée *a*, nous réappliquons les règles 6.2 et 6.12 à partir du port d'entrée *c* pour obtenir l'expression P1({}, {}) || P2({}, {}). Dans le cas contraire où nous choisissons le port d'entrée *c*, nous réappliquons les règles 6.3 et 6.12 à partir du port d'entrée *a* pour obtenir l'expression P1({}, {}) || P2({}, {}).
c) Nous choisissons la règle 6.3 du mécanisme de composition *parallèle*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *c* de P2, la prochaine expression est P1({a,b}, {}) || P2({}, {}). Nous appliquons les règles 6.2 et 6.12, et nous obtenons respectivement à

partir du port d'entrée *a* l'expression P1({b}, {}) || P2({}, {}) et à partir du port d'entrée *b* l'expression P1({a}, {}) || P2({}, {}). Dans le cas où nous choisissons le port d'entrée *a*, nous réappliquons les règles 6.2 et 6.12 à partir du port d'entrée *b* pour obtenir l'expression P1({}, {}) || P2({}, {}). Dans le cas contraire où nous choisissons le port d'entrée *b*, nous réappliquons les règles 6.3 et 6.12 à partir du port d'entrée *a* pour obtenir l'expression P1({}, {}) || P2({}, {}).

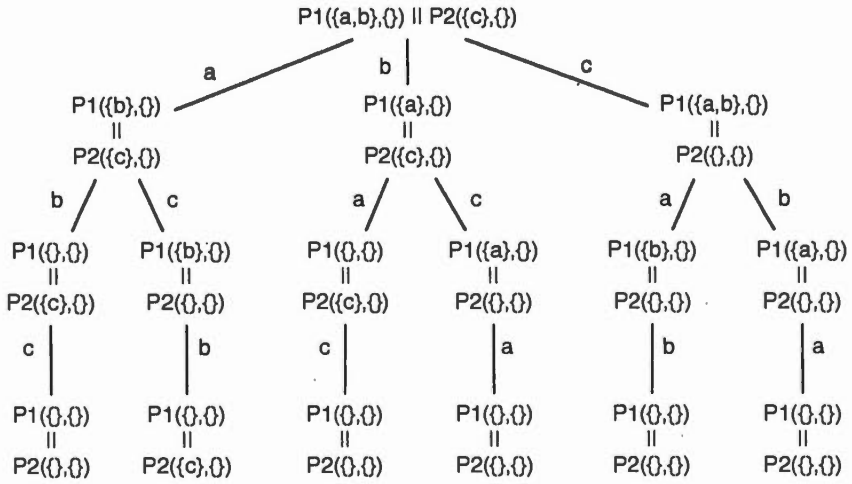


Figure 6.19 Exemple *parallèle*

Dans le dernier exemple, nous supposons que deux composants ont un port en commun et que sur ce port le premier composant émet une valeur et la deuxième la reçoit (voir figure 6.20). L'action d'émission est appelée *b_* qui veut dire qu'une valeur a été émise sur le port *b*. Pour l'autre composant, cette action s'appelle *b* qui reçoit la valeur émise sur le port. Dans ce cas, et selon la règle 6.4 les deux composants se synchronisent sur ce port et produisent une action appelée (selon la définition 6.3) *sync(b_,b)*. Cette composition donne le comportement que nous présentons dans la figure 6.22.

Nous appliquons d'abord le mécanisme de composition *parallèle* (voir règle 6.2, 6.3 et 6.4) et d'*invocation* (voir la règle 6.12 et 6.13), et ce qui nous donne initialement 2 choix possibles : l'activation du port d'entrée *a* de P1 et l'activation du port d'entrée *b* de P2.

a) Nous choisissons la règle 6.2 du mécanisme de composition *parallèle*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *a* de P1, la prochaine expression est P1({}, {b}) || P2({b}, {}). Nous réappliquons la règle 6.12 en choisissant parmi les ports suivants : le port d'entrée *b* (voir règle 6.2), le port d'entrée *b* (voir règle 6.3) ou la synchronisation *synch(b,b)* des ports de sortie *b* et d'entrée *b* (voir règle 6.4). En choisissant ces ports, nous obtenons respectivement les expressions P1({}, {}) || P2({b}, {}), P1({}, {b}) || P2({}, {}) et P1({}, {b}) || P2({b}, {}). Pour le premier cas, nous appliquons les règles 6.3 et 6.12 sur le port d'entrée *b* de P2 et nous obtenons P1({}, {}) || P2({}, {}). Dans le second cas, nous appliquons les règles 6.2 et 6.13 sur le port de sortie *b* de P1 et nous obtenons P1({}, {}) || P2({}, {}). Quant au dernier cas, la *synch(b_,b)* nous permet déjà d'obtenir P1({}, {}) || P2({}, {}) à partir de la règle 6.4.
b) Nous choisissons la règle 6.3 du mécanisme de composition *parallèle*. Cette règle exige que nous utilisions la règle 6.12 (*invocation*). En choisissant le port d'entrée *b* de P2, la prochaine expression est P1({a}, {b_}) || P2({}, {}). Nous appliquons les règles 6.2 et 6.12 sur le port d'entrée *a* de P1 et nous obtenons l'expression P1({}, {b_}) || P2({}, {}). Par la suite, nous appliquons la règle 6.2 et 6.13 sur le port de sortie *b_* de P1 et nous obtenons l'expression P1({}, {}) || P2({}, {}).

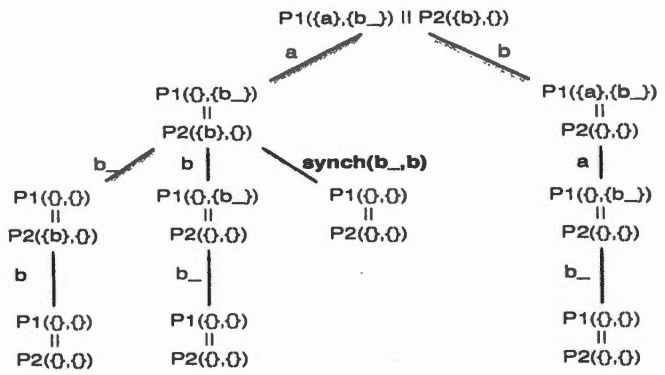


Figure 6.20 Exemple de *synchronisation* pour une composition *parallèle*

6.7 Exécution de séquences

À partir d'un nœud donné n_0, nous pouvons exécuter une séquence d'actions et arriver à un autre nœud m. Soit $s = \lambda_1, \lambda_2, \ldots, \lambda_m$ une séquence d'actions. Nous exécutons cette séquence à partir du nœud initial n_0, comme suit:

$n_0 \xrightarrow{\{s\}} n_m$ s'il existe des nœuds $n_1, n_2, \ldots, n_{n-1}$ tels que $tr(n_i, \lambda_1) = n_{i+1}, i = 0, \ldots, m-1$

Nous notons cette opération: $\alpha(n_0, s) = n_m$ (6.21)

Nous pouvons représenter graphiquement ces transformations comme dans les exemples qui suivent. Le *ST* de l'exemple précédent est donné dans la figure 6.21.

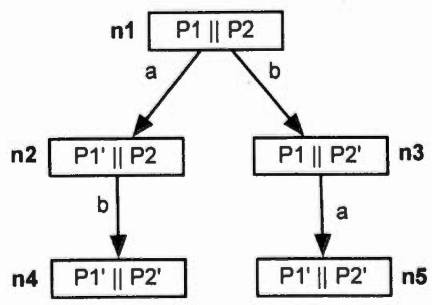
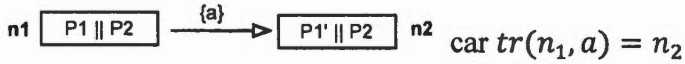
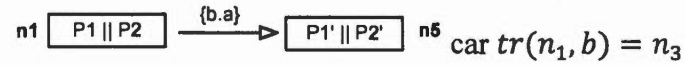


Figure 6.21 Représentation graphique des transformations

À partir du nœud n_1, nous pouvons avoir les exécutions suivantes:

- 

car $tr(n_1, a) = n_2$

- 

car $tr(n_1, b) = n_3$ et $tr(n_3, a) = n_5$

Donc: $\alpha(n_1, \{b.a\}) = n_5$.

6.8 Substitution d'opérateurs

Afin de faire des changements d'architecture, nous utilisons l'opération de substitution d'opérateur. Par exemple, nous pouvons décider qu'en fonction de la valeur du contexte, nous pouvons modifier la composition parallèle entre deux composants en alternatif. Ce qui aurait pour effet de permettre le choix exclusif entre deux comportements et leurs exécutions parallèles.

Pour une expression $expr(n)$ correspondant à un nœud n, nous définissons la substitution $[[op1/op2]]$ d'opérateur dans une expression P comme suit:

$P\,[[op1/op2]]$ est l'expression obtenue en remplaçant dans P la $n^{ième}$ toute occurrence de l'opérateur $op1$ par $op2$.

La sémantique de la transformation des opérateurs ||, + et * se définit par la règle suivante:

$$(C1\ op1\ C2)[op1/op2] = C1[op1/op2]\ op2\ C2[op1/op2] \tag{6.22}$$

Cet opérateur est défini sur d'autres types d'expressions comme suit :

$$Call(inP, outP)[op1/op2] = Call(inP, outP) \tag{6.23}$$

$$(C\ hide\ \lambda)[op1/op2] = C[op1/op2]\ hide\ \lambda \tag{6.24}$$

$$(E -> C)[op1/op2] = E -> C\ [op1/op2] \quad (6.25)$$

Nous pouvons également spécifier l'étape à laquelle nous voulons commencer une transformation en utilisant un paramètre général (n est le niveau) :

$$(C1\ op1\ C2)[op1/op2]_n = C1[op1/op2]_{n-1} op2\ C2[op1/op2]_{n-1} \text{si } n > 1 \quad (6.26)$$

$$(C1\ op1\ C2)[op1/op2]_1 = C1\ op2\ C2 \quad (6.27)$$

Nous pouvons aussi spécifier la profondeur à laquelle nous voulons terminer une transformation en utilisant un paramètre général (n est le niveau) :

$$(C1\ op1\ C2)[op1/op2]^n = C1[op1/op2]^{n-1} op1\ C2[op1/op2]^{n-1} \text{ si } n > 1 \quad (6.28)$$

$$(C1\ op1\ C2)[op1/op2]^1 = (C1\ op1\ C2)\ [op1/op2] \quad (6.29)$$

Si n=*all* alors toutes les occurrences de *op1* sont remplacées par *op2*.

Voici quelques exemples de transfomation :

- $(P1 \,||\, P2)\ [[\,||\,/+\,]^1] = P1 + P2$
- $(P1 \,||\, P2 \,||\, P3)\ [[\,||\,/+\,]^1] = P1 + (P2||P3)$
- $(P1 \,||\, P2 \,||\, P3)\ [[\,||\,/+\,]_2] = P1 \,||\, (P2 + P3)$
- $(P1 \,||\, P2 \,||\, P3)\ [[\,||\,/+\,]all] = P1 + P2 + P3$

Nous définissons la sémantique des transformations d'opérateurs avec la notion de profondeur :

1. $(P1\ op1\ P2)\ [[op1\ /\ op2]_1\ ] = P1\ op2\ P2$
2. $(P1\ op1\ P2)\ [[op1\ /\ op2]_n\ ] = P1\ op1\ P2\ [[op1\ /\ op2]_{n-1}]\ si\ n > 1$
3. $(P1\ op1\ P2)\ [[op1\ /\ op2]all] = P1\ op2\ P2\ [[op1\ /\ op2]\ all]$

Définition 6.5 Transformation différée. Soit une séquence d'actions $s = \lambda_1, \lambda_2, \ldots, \lambda_m$ nous l'appliquons à partir d'un nœud racine n_0 et après l'exécution de la séquence s. q représente soit l'étape à laquelle nous voulons commencer une transformation (indice), soit la profondeur à laquelle nous voulons terminer une transformation (exposant) (voir figure 6.22):

$$< n_0, s, [op1/op2]^q > \; -> \; n_0 \xrightarrow{\{s\}} n_m \, [op1 \,/\, op2]^q \tag{6.30}$$

$$< n_0, s, [op1/op2]_q > \; -> \; n_0 \xrightarrow{\{s\}} n_m \, [op1/op2]_q \tag{6.31}$$

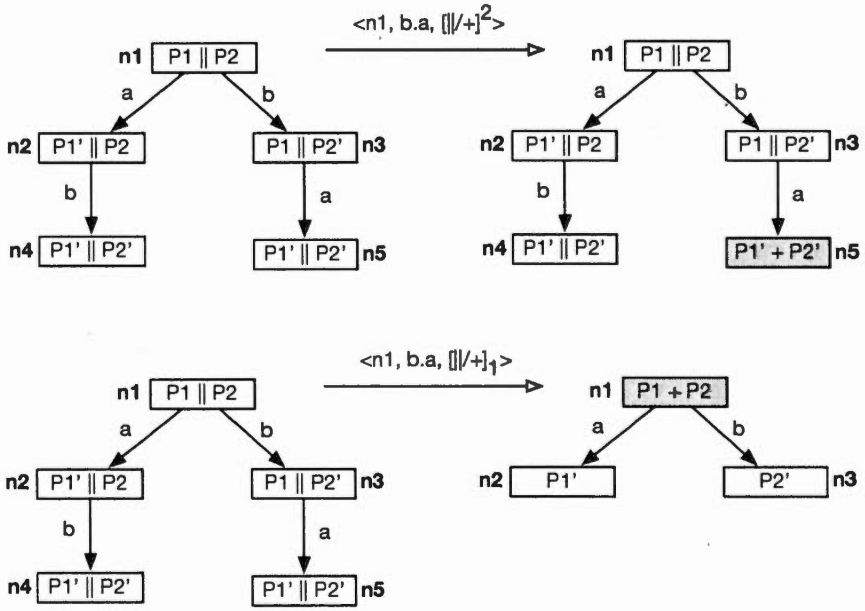


Figure 6.22 Exemple de transformation

Définition 6.6 Applications des contraintes. Soit un système P et une contrainte C. L'expression P < C indique que la contrainte C s'applique à P dès le début de son comportement. Si C exécute une séquence d'actions $s = a_1, \ldots, a_n$, nous devons exécuter la même séquence sur P:

Si $C \xrightarrow{a_1.a_2....a_n} C'$, alors il existe P' tel que $P \xrightarrow{a_1.a_2....a_n} P'$. Donc: $P < C \xrightarrow{a_1.a_2...a_n} P' < C'$.

Notons cette opération, $\alpha(P < C, s) = P' < C'$. (6.31)

6.9 Implantation du modèle sémantique en Prolog

Pour implanter les règles sémantiques définies dans les sections 6.2, 6.4 et 6.8, nous avons recours à la programmation à base de règles en utilisant le langage Prolog (SWI-Prolog, 2016). Nous avons choisi Prolog en tant que langage de programmation logique car il se prête bien à cette implémentation.

6.9.1 Implantation des opérateurs

Nous implantons les règles d'inférence présenté dans la section 6.2 en utilisant la clause *infer/3* qui a trois arguments: l'expression courante, l'action à exécuter et l'expression qui résulte de l'exécution de cette action.

Les opérateurs sont représentés par une notation postfix grâce à une expression de type *op(X1,X2)* où *op* est le nom de l'opérateur (exemple: *par* pour || , *alt* pour +, etc.). Par exemple, l'expression *call(p1,{a,b},{c,d}) || call(p2,{c,e},{g,h})* est représentée par:

par(call(p1,[a,b],[c,d]), call(p2,[c,e],[g,h])).

Pour définir ces règles, nous avons besoin de clauses qui déterminent les ports d'entrée et de sortie de composants. Nous utilisons les clauses *inPorts/2* et *outPorts/2*:

```
inPorts(call(P,InP,OutP),InP).
outPorts(call(P,InP,OutP),OutP).
inPorts(par(P1,P2),I) :- inPorts(P1,I1),inPorts(P2,I2),union(I1,I2,I).
outPorts(par(P1,P2),I) :- outPorts(P1,I1),outPorts(P2,I2),union(I1,I2,I).
```

Avec ces notations, nous donnons les règles qui implantent ces opérateurs:

- Séquence :

```
infer(seq(A,P1, P1).
```

- Alternative

```
infer(alt(P1,P2),A,P11):-infer(P1,A,P11).
infer(alt(P1,P2),A,P21):-infer(P2,A,P21).
```

- Parallèle

```
infer(par(P1,P2),A,par(P11,P2)):- infer(P1,A,P11).
infer(par(P1,P2),A,par(P1,P21)):- infer(P2,A,P21).
infer(par(P1,P2),A,par(P11,P21)):-
                infer(P1,A1,P11), infer(P2,A2,P21), synch(A1,A2,A).
```

- Pipeline

```
infer(pipe(P1,P2),A, pipe(P11,P2)):-
                infer(P1,A,P11),  inPorts(P2,P1),
                not(member(A,IP)).
infer(pipe(P1,P2),A, P22):-
                infer(P1,A,P11), infer(P2,A,P21),
                outPorts(P1,P11),member(A,P11),
                inPorts(P2,P12), member(A,P12).
```

Notons que dans le cas d'une composition en pipeline, il n'y a pas de ports d'entrée du composant P2 qui fasse partie du port d'entrée du composant P1.

- Invocation:

Le cas de l'invocation nécessite un traitement particulier, car il utilise l'opérateur universel ∀ qui n'existe pas en prolog. Pour l'implanter, nous devons utiliser le mécanisme d'expansion. Si nous avons une expression avec deux composants reliés par un opérateur, nous essayons de la transformer en une somme (alternative) de séquences. Cette opération est souvent utilisée pour transformer des expressions avec des opérateurs de parallélisme en une expression qui ne contient que des préfixes et des alternatives.

Nous devons donc transformer les expressions de type *call(P, InP, OutP)* en définissant des règles d'inférence plus simples à manipuler.

Nous définissons d'abord formellement l'expansion de cette expression comme suit:

- $call(P, InP, OutP) = \sum_{x \in InP} x.call(P, InP \backslash x, OutP)$
- $call(P, [], OutP) = \sum_{y \in OutP} y.call(P, [], OutP \backslash y)$

Nous ne définissons pas de règle pour les expressions de type *call(P,[],* [redacted]. Ce qui en fait des éléments terminaux.

Nous implantons les règles d'expansion en utilisant la clause *bagof/3* de Prolog. *bagof(X, P, C)* produit tous les éléments *X* qui satisfont le prédicat *P*. Le résultat est mis dans la variable *C*.

Par exemple, si la clause enfant(X,Y) asserte que *Y* est un enfant de *X* :

enfant(jean, marie).
enfant(alain, pierre).
enfant(jean, michel).
enfant(jean, claude).

alors *bagof(X, enfant(jean,X), C)* met dans *C* tous les enfants de *jean*, soit : *marie*, *michel* et *claude*.

Avec cette définition, nous pouvons procéder à l'expansion des expressions d'invocation:

```
expansion(call(P, InP, OutP), Res):-
        bagof(seq(A, call(P, InPm, OutP)), (member(A, InP), minus(A, InP, InPm)),
        C), alternatives(C, Res).
expansion(call(P, [], OutP), Res):-
        bagof(seq(B, call(P, [], OutPm)), (member(B, OutP), minus(A, OutP,
        OutPm)), C), alternatives(C, Res).
```

Nous appliquons ces règles à l'expression call(foo,[a,b,c],[d,e]):

```
process(p1, call(foo,[a,b,c],[d,e])).
go9:-  process(p1, P),
       expansion(call(foo,[a,b,c],[d,e]) ,X),
       write('Expression :  '), write(P),nl,
       write('Son expansion :  '), write(X).
?- go9.
Expression : call(foo,[a,b,c],[d,e])
```

Son expansion :

```
alt(seq(a,call(foo,[b,c],[d,e])),alt(seq(b,call(foo,[a,c],[d,e])),seq(c,call(foo,[a,b],[d,e
])))
```

6.9.2 Exécution des actions

Pour montrer l'exécution des actions à partir des règles du modèle sémantique (i.e. des règles d'inférence), nous avons défini des clauses pour l'exécution du comportement d'un composant. Ces clauses donnent également le résultat de l'exécution de cette exécution.

```
process(p2,par(call(p1,[a,b],[c,d]), call(p2,[c,e],[g,h]))).
go1 :- process(p2,Process),
       infer(Process, Action, Next),
       write(Action), write("-> "),  write(Next), nl, fail.
```

En l'exécutant, nous obtenons pour chaque action exécutée, l'étape suivante dans le comportement du composant.

```
?-go1.
a-> par(call(p1,[b],[c,d]),call(p2,[c,e],[g,h]))
b-> par(call(p1,[a],[c,d]),call(p2,[c,e],[g,h]))
c-> par(call(p1,[a,b],[c,d]),call(p2,[e],[g,h]))
e-> par(call(p1,[a,b],[c,d]),call(p2,[c],[g,h]))
```

Pour connaitre le résultat d'une action en particulier sur un processus, nous pouvons appeler la règle *go0/1*:

```
go0(Action) :-
    process(p2, Process),
    infer(Process, Action, X),
    write("-> "),  write(X),  nl, fail.
?- go0(a).
-> par(call(p1,[b],[c,d]),call(p2,[c,e],[g,h]))

?- go0(b).
-> par(call(p1,[a],[c,d]),call(p2,[c,e],[g,h])).
```

6.9.3 Exécution de séquences d'actions

Pour obtenir l'ensemble des résultats suite à l'exécution d'une séquence d'actions, nous utilisons la clause *trace/3* ci-dessous qui détermine si une liste d'action est acceptée par un composant:

```
trace(P,[],P):-!.
trace(P, [S1|S], Res):-
        infer(P, S1, Next),
        trace(Next, S, Res).
```

La clause *go3/2* permet d'effectuer cette vérification.

```
process(p2, Process).
go3(S, Process):-
        trace(Process, S, Res),
        write('Sequence: '), write(S),nl,
        write(' -> '), write(Res).

?- go3([b,a]).
Sequence: [b,a]
-> par(call(p1,[],[c,d]),call(p2,[c,e],[g,h]))
```

6.9.4 Comportement global d'un processus

Nous pouvons déterminer l'ensemble des actions et leurs résultats correspondants dans le comportement d'un composant. Cela est implanté grâce à la clause *bagof/3* de Prolog.

Dans la clause ci-dessous, nous déterminons l'ensemble des solutions résultant de l'exécution d'un processus. La clause *go2/1* permet d'obtenir le résultat et le lister avec la clause *do/1*.

```
go2 :- process(P), bagof([A, N], infer(P, A, N), Res), do(Res).

do([A]):- write("-"),write(A).
do([A|X]):-write("-"),write(A),nl,do(X).
```

L'exécution de *go2* donne:

```
?- go2.
-[a, par(call(p1,[b],[c,d]),call(p2,[c,e],[g,h]))]
-[b, par(call(p1,[a],[c,d]),call(p2,[c,e],[g,h]))]
-[c, par(call(p1,[a,b],[c,d]),call(p2,[e],[g,h]))]
-[e, par(call(p1,[a,b],[c,d]),call(p2,[c],[g,h]))]
```

6.9.5 Implantation des contraintes

Pour implanter les contraintes, nous avons ajouté les règles suivantes:

```
infer(seq(A,P), A, P).
infer(const(P1,P2), A, const(P11,P21)):-
        infer(P1,A,P11), infer(P2,A,P21).
```

Pour montrer un exemple, nous définissons une contrainte *c1* que nous imposons au processus *p1* en utilisant l'expression suivante. Nous montrons uniquement les premières actions de l'exécution:

```
process(p11, call(pressure,[a,b,c],[])).
constraint(c1, alt( seq(a,seq(b,seq(c,null))), seq(b,seq(a,seq(c,null))))).
```

```
go4:-
        process(p11,Pexp),
        constraint(c1, Cexp), !,
        infer(const(Pexp,Cexp), Action, Res),
        write('Action: '), write(Action), nl,
        write('Result: '), write(Res), nl.

?- go4.
a ->const(call(pressure,[b,c],[]),seq(b,null))
b ->const(call(pressure,[a,c],[]),seq(a,null))
```

6.9.6 Transformation d'opérateurs

Nous avons également implanté les opérations de transformation des opérateurs. La clause *transformOperator/4* prend une expression et deux opérateurs et produit une expression dans laquelle le premier opérateur est remplacé par le deuxième. Elle implante l'opérateur [[op1/op2]]. Elle consiste à remplacer *Op1* par *Op2* dans le composant *Exp*. Le résultat est retourné dans *Res*.

```
transformOperator( call(P, InP, OutP), Op1,Op2, call(P,InP, OutP)) :- !.

transformOperator(Exp, Op1, Op2, Res):-
        Exp =.. [Op1, A1, A2],
        transformOperator(A1,Op1,Op2,Res1),
        transformOperator(A2,Op1,Op2,Res2),
        Res =..[Op2, Res1, Res2].

transformOperator(Exp, Op1, Op2, Res):-
        Exp =.. [Op3, A1, A2],
        transformOperator(A1,Op1,Op2,Res1),
        transformOperator(A2,Op1,Op2,Res2),
        Res =.. [Op3, Res1, Res2].
```

Pour appeler cette clause, nous avons défini la clause *go/06* suivante:

```
process(p12, par(call(p1,[a,b],[c,d]),call(p2,[c,e],[g,h]))).
go6:-  process(p12,P),
        write('Initial expression:   '), write(P),nl,
```

```
transformOperator(P,par,pipe,S),
write('Resulting expression: '), write(S),nl.
```

En l'exécutant, nous obtenons:

```
?-go6.
Initial expression: par(call(p1,[a,b],[c,d]),call(p2,[c,e],[g,h]))
Resulting expression: pipe(call(p1,[a,b],[c,d]),call(p2,[c,e],[g,h]))
```

Nous avons également implanté l'opération qui consiste à faire une opération de substitution des opérateurs après avoir exécuté une séquence en particulier. L'opération est implantée grâce à la clause *transformExecute/5* donnée ci-dessous.

```
transformExecute(Exp, [], _, _,[]):- !.
transformExecute(Exp, Seq, Op1, Op2, Res1):-
        trace(Exp,Seq,Res),
        transformOperator(Res, Op1, Op2, Res1).
```

Voici un exemple d'exécution de cette clause.

```
process(p4, par(call(p1,[a,b],[c,d]), call(p2,[c,e],[g,h]))).
sequence([a,b,e]).

go7:- process(p4,P), sequence(S),
      write('Initial expression:  '), write(P),nl,
      write('Sequence: '), write(S),nl,
      transformExecute(P,S, par, seq, Res),
      write('Resulting expression: '), write(Res).
```

L'exécution de cette clause donne:

```
?- go7.
Initial expression: par(call(p1,[a,b],[c,d]),call(p2,[c,e],[g,h]))
Sequence: [a,b,e]
Resulting expression: seq(call(p1,[],[c,d]),call(p2,[c],[g,h]))
```

Les opérateurs [[op1/op2] ^{n}] et [[op1/op2] $_{n}$] sont implantés respectivement par les clauses *transformFrom/5* et *transformTo/5* suivantes.

```
transformFrom(Exp, Op1, Op2, Res,1):-
        transformOperator(Exp, Op1, Op2, Res), !.
transformFrom(Exp, Op1, Op2, Res,N):-
        N > 1, N1 is N-1,
        transformFrom(Exp, Op1, Op2, Res,N1).

transformTo(Exp, Op1, Op2, Res1,N):-
        N > 1, N1 is N-1,
        transformOperator(Exp, Op1, Op2, Res),
        transformTo(Res, Op1, Op2, Res1,N1).
transformTo(Exp, Op1, Op2, Res,1):-
        transformOperator(Exp, Op1, Op2, Res).
```

Nous avons utilisé ces opérations dans notre mise en oeuvre des techniques d'adaptation que nous présentons dans le chapitre 7.

6.10 Conclusion

Dans ce chapitre, nous nous sommes inspirés de l'algèbre des processus pour proposer un modèle sémantique d'adaptation basé sur des composants. Nous avons décrit les ports et les mécanismes de composition de composants, à savoir : le parallèle, l'alternative, le pipeline, l'invocation, la conditionnelle et l'encapsulation. Notre modèle se base sur deux concepts importants qui permettant l'adaptation comportementale: les contraintes qui imposent un ordre particulier dans l'exécution du système et les transformations d'opérateurs qui définissent quand et comment modifier l'architecture. Nous avons également montré l'implantation des divers opérateurs en Prolog.

Dans le prochain chapitre, nous montrons la mise en œuvre de deux techniques d'adaptation. La première repose sur l'adaptation par features qui est une suite du chapitre 5 et la seconde repose sur l'adaptation par composition que nous avons présentée dans ce chapitre. Dans les deux cas, nous utilisons le modèle de référence

de la boucle de contrôle MAPE-K et nous l'implantons en Prolog pour chacune des techniques d'adaptation.

CHAPITRE VII

TECHNIQUES D'ADAPTATION

7.1 Introduction

Comme mentionné dans le chapitre 1, différents niveaux peuvent être considérés lors d'une adaptation, à savoir une adaptation statique ou dynamique, l'adaptation centralisée ou distribuée, et l'adaptation comportementale ou architecturale.

Dans le cas des applications sensibles au contexte, l'adaptation se fait au niveau comportemental et/ou architectural du système. Nous rappelons que l'adaptation comportementale implique des changements du comportement d'un système alors que l'adaptation architecturale repose sur la modification de la structure du système. Quel que soit le type d'adaptation choisie, l'adaptation est un processus qui est effectué suite à un changement dans le contexte.

Nous avons adopté ces deux méthodes d'adaptation que nous avons appelées *Adaptation par composition* (Amja, Obaid et Mili, 2017) et *Adaptation par features* (Amja, Obaid, Mili, et Jarir, 2016):

- *L'adaptation par composition* : Dans le chapitre précédent, nous avons proposé un modèle sémantique permettant de spécifier le comportement d'un système en termes de comportement de ses composants. Cette composition décrit une architecture exprimée à l'aide d'opérateurs. À partir de ce modèle sémantique, nous proposons d'effectuer une adaptation architecturale par l'imposition de contraintes et par transformation d'opérateurs.

- *L'adaptation par features* : Dans le chapitre 5, nous avons proposé un modèle à base de *features* en utilisant une ontologie qui établit une relation entre les *features* et le contexte modélisé à partir de l'analyse relationnelle de concepts. Grâce au modèle de *variabilité* dans la définition d'un système que nous avons déjà introduit, nous pouvons modifier son comportement par l'inclusion ou l'exclusion de certains *features* permettant ainsi une adaptation basée sur les fonctionnalités que le système peut offrir.

Nous allons décrire notre méthodologie et sa simulation en Prolog.

Ce chapitre est organisé comme suit. La section 7.2 présente la boucle de contrôle que nous utilisons pour effectuer l'adaptation comportementale et architecturale, à savoir la boucle de contrôle MAPE-K. Dans les sections 7.3 et 7.4, nous présentons par la suite notre méthodologie d'adaptation par *features* et par recomposition ainsi que son implantation en Prolog, respectivement.

7.2 Boucle de contrôle

Comme expliqué dans la section 2.8 du chapitre 2, les boucles de contrôles fournissent un mécanisme général pour l'auto-adaptation de système. Elles représentent un aspect important des systèmes auto-adaptatifs et ses activités affectent le comportement et l'architecture de ces systèmes. Nous avons opté pour la boucle MAPE-K proposée par IBM (Jacob *et al.*, 2004). Cette boucle comprend 4 activités: l'observation, l'analyse, le planification et l'exécution. Chacune de ces activités interagit avec la base de connaissances. La figure 7.1 illustre la boucle de contrôle MAPE-K et les interactions entre ses activités.

Le modèle de la boucle de contrôle est très utilisé dans l'implantation des systèmes autonomes. C'est également un système qui est utilisé dans le contrôle des systèmes dynamiques intelligents tels que les processus de fabrication intelligents ainsi que dans les systèmes réactifs.

Comme nous l'avons expliqué auparavant, il existe plusieurs types de modèles de boucles de contrôle. Nous avons adopté le plus simple d'entre eux et qui répond à nos besoins tout en offrant une simplicité dans son implantation.

Nous utilisons et implantons ce modèle en Prolog. Le choix de ce langage s'est imposé naturellement à cause de ses propriétés. C'est un langage permettant de représenter des connaissances. Il est extensible grâce à l'ajout de clauses dynamiques et il possède des propriétés de langage fonctionnel.

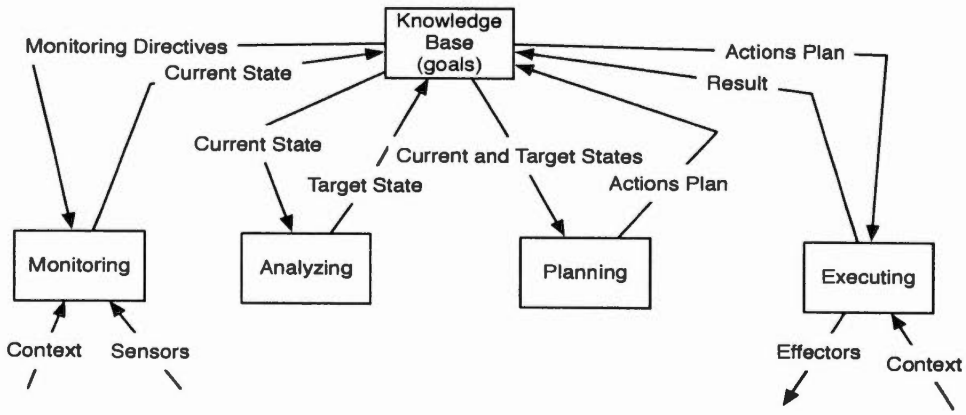


Figure 7.1 Boucle de contrôle MAPE-K (Jacob *et al.*, 2004)

7.2.1 Le moniteur

L'interaction entre le moniteur et la base de connaissances est illustrée par la figure 7.2:

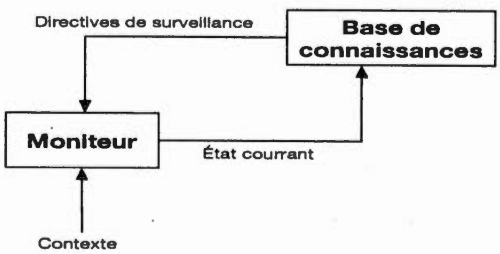


Figure 7.2 Le rôle du moniteur

À la demande de la base de connaissances, le moniteur collecte des données des capteurs, et informe celle-ci des changements d'état qui ont eu lieu. Il détermine également les évènements qui doivent être analysés.

D'un autre côté, la base de connaissances envoie au moniteur des directives lui permettant de déterminer les données à considérer, quand et comment les traiter. En retour, le moniteur informe cette base de l'état courant du système.

Le moniteur transforme l'information collectée sous un format qui peut être manipulée par les autres phases. La transformation implique des activités de filtrage, d'analyse et d'agrégation.

7.2.2 L'analyseur

L'interaction entre l'analyseur et la base de connaissances est illustrée par la figure 7.3.

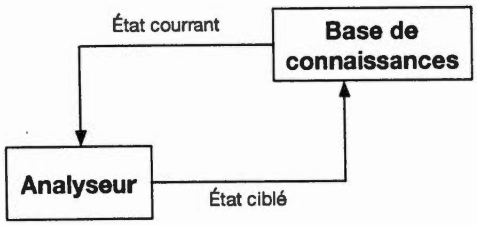


Figure 7.3 Le rôle de l'analyseur

L'analyseur évalue les données reçues du moniteur et décide s'il y a eu un changement d'état. Dans ce cas, il apprend l'identité du nouvel état et en informe la base de connaissances.

Il consiste donc à analyser et comprendre l'état courant du contexte et du système. Son rôle implique également de fournir à la phase de planification un état ciblé qui doit être atteint à partir de l'état courant.

7.2.3 Le planificateur

L'interaction entre le planificateur et la base de connaissances est illustrée par la figure 7.4.

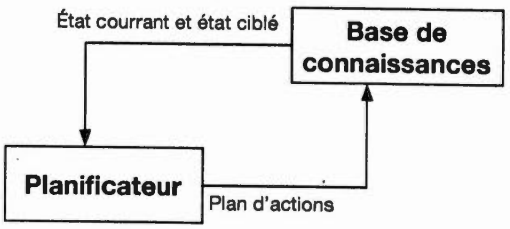


Figure 7.4 Le rôle du planificateur

Cette phase consiste à prendre des décisions concernant les changements et les adaptations à implémenter sur le système afin de passer de l'état courant à un état désiré. Elle dépend d'un ensemble d'actions qui peuvent être exécutées sur le système.

Un plan peut être statique ou dynamique. Par exemple, un plan statique pourrait être un ensemble d'étapes qui doit être réalisé lorsqu'une condition particulière survient alors qu'un plan dynamique pourrait être de modéliser le comportement des composants du logiciel et de choisir un plan parmi un nombre de plans prédéfinis.

Une façon d'établir la planification est d'exprimer l'état du système et de définir les actions agissant sur ces états. Ils définissent les pré-conditions et les post-conditions sur les états. Le planificateur détermine une séquence d'opérateurs permettant de passer d'un état courant à un état ciblé (c.-à-d. généralement sous forme de graphe avec ou sans heuristiques) (Lalanda *et al.*, 2013).

À partir de l'état courant et sachant l'état ciblé, le planificateur prend une décision concernant les changements et l'adaptation à implémenter afin de passer de l'état courant à l'état ciblé.

7.2.4 L'exécuteur

L'interaction entre l'exécuteur et la base de connaissances est illustrée par la figure 7.5.

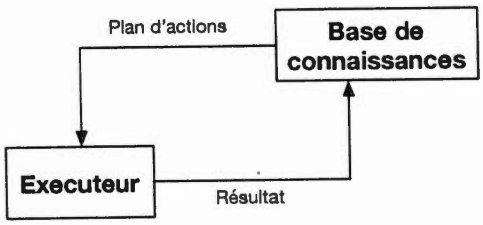


Figure 7.5 Le rôle de l'exécuteur

Cette phase implémente les détails des actions déterminées par la phase de planification. Elle exécute des plans et examine en temps réel les post-conditions de ces actions afin d'exécuter des ajustements. De plus, elle effectue des requêtes pour établir des modifications sur les entités du système. Normalement, les solutions pour cette phase sont spécifiques au domaine en question.

7.3 Adaptation par *features*

Grâce au modèle sémantique de la variabilité décrit dans le chapitre 5 (section 5.1), nous pouvons déterminer des configurations possibles respectant ce modèle et les exécuter.

Pour effectuer cette adaptation, nous utiliserons les éléments suivants:

- Conditions sur les valeurs du contexte : les conditions sur les valeurs du contexte permettent de guider le système dans le choix de *features* qu'il doit inclure ou exclure.
- Conditions sur des intervalles de valeurs du contexte : Les valeurs captées ne sont pas toutes considérées par le système. Certains de ces intervalles forceront le choix de certaines configurations du système. Chaque

configuration consiste en un ensemble de *features* que nous devons inclure ou exclure.

7.3.1 Le moniteur

Pour effectuer ses tâches, le moniteur utilise une structure contenant des directives pour un contexte donné, appelé Structure de directives. Cette structure lui est fournie par la base de connaissances.

Définition 7.1 Structure de directives. Les directives contiennent les informations suivantes concernant la manipulation des valeurs du contexte:

{id, type, delta, svalues : categorie$<e_1:i_1, e_2:i_2, ..., e_n:i_n>$*, Intervals : {Categorie :* $<e_1:i_1, e_2:i_2, ..., e_n:i_n>$*, ...} }*

- *id*: un identifiant associé au contexte.
- *type*: un type de données associé aux valeurs de ce contexte. Ce type peut être réel, entier ou chaîne (*string*).
- *delta:* une valeur de précision qui désigne un taux d'erreur toléré dans les valeurs lues pour ce contexte.
- *svalues: categorie* :$<e_1:i_1, e_2:i_2, ..., e_n:i_n>$ qui indique l'intervalle des valeurs possibles d'une catégorie. Catégorie désigne la catégorie des données. Par exemple, dans le cas des données sur les températures corporelles, il existe une seule catégorie que nous avons appelée *oneset*. Par contre, dans le cas des données sur la pression artérielle, nous avons deux catégories : *systolic* qui correspond aux valeurs systoliques de la pression et *distolic* qui correspond aux valeurs diastoliques de cette pression.

- *Intervals : {Categorie : <e_1:i_1, e_2:i_2,..., e_n:i_n>, ...} :* un ensemble d'intervalles de valeurs possibles pour ce contexte. À chaque intervalle est associé un état qui l'identifie et auquel nous devons associer un traitement spécifique. Ce dernier pourra être implémenté par un composant ou un *feature*.

Pour chaque contexte, nous créerons des règles en Prolog qui déterminent les directives.

Par exemple, pour le contexte de température corporelle appelée *bodytemperature*, nous définissons les directives suivantes en Prolog:

```
profile(bodytemperature, [
                        id:id_bt,
                        type:float,
                        detla:1,
                        svalues:[temp:[30,42]],
                        intervals:[   hypo: [temp:[30,35.5]],
                                      normal:[temp:[35.5,37.5]],
                                      fever: [temp:[37.5,42]]
                                  ]
                    ]
    ).
```

Ce contexte peut prendre des valeurs réelles entre 30 et 42 degrés. Ces valeurs sont divisées en trois intervalles: un qui corresponde à l'état normal (températures entre 35.5°C et 37.5°C), un à l'état hypothermique (températures entre 30°C et 35.5°C) et un à l'état fiévreux (températures entre 37.5°C et 42°C). Ces valeurs peuvent être déterminées avec une précision de 1°C. Ce qui veut dire que si deux températures diffèrent d'une valeur inférieure à 1, nous ne détectons pas de changement d'état. Dans ce cas, le moniteur n'informera pas le système d'un quelconque changement d'état.

Après avoir reçu des directives de la base de connaissances pour un contexte, le moniteur créé dynamiquement des règles qui les représentent.

La création de ces règles par la base de connaissances se fait avec la clause *kb/2*. Pour le contexte *bodytemperature*, les règles suivantes sont créées:

?- kb([bodytemperature]).
Context: bodytemperature
Regle: id(bodytemperature,id_bt)
Regle: type(bodytemperature,float)
Regle: detla(bodytemperature,2)
Regle: svalues(bodytemperature,[temp:[30,42]])
Regle:intervals(bodytemperature,[hypo:[temp:[30,35.5]],normal:[temp:[35.5,37.5]], fever:[temp:[37.5,42]]])

Le moniteur lit des données à partir d'un capteur (dans notre cas, celui-ci est représenté par un simple fichier texte). Il ajoute l'horodate de cette lecture et vérifie les données lues en fonction des directives reçues dans les propriétés du contexte à partir du filtre créé par la base de connaissances.

En utilisant ces règles de filtrage, il détermine si les données sont valides et détermine l'intervalle de valeurs auquel elles appartiennent ainsi que l'état correspondant à chaque valeur lue.

Dans l'exemple du contexte *bodytemperature*, le moniteur produit (après avoir reçu les directives et lu les données) le résultat suivant:

Context Value Timestamp
Context : bodytemperature - 35.4 - Time : 1458066944.828372
Context : bodytemperature - 35.5 - Time : 1458066944.840909
Context : bodytemperature - 35.6 - Time : 1458066944.841064
Context : bodytemperature - 35.6 - Time : 1458066944.841174
Context : bodytemperature - 35.7 - Time : 1458066944.841279
Context : bodytemperature - 35.8 - Time : 1458066944.841386
Context : bodytemperature - 35.7 - Time : 1458066944.841484
Context : bodytemperature - 35.9 - Time : 1458066944.841598
Context : bodytemperature - 36 - Time : 1458066944.841705

Context : bodytemperature - 36.2 - Time : 1458066944.841809
Context : bodytemperature - 36.1 - Time : 1458066944.841921
Context : bodytemperature - 37.6 - Time : 1458066944.842032
Context : bodytemperature - 37.5 - Time : 1458066944.842118
Context : bodytemperature - 37.9 - Time : 1458066944.842179
Context : bodytemperature - 38.1 - Time : 1458066944.842239
Context : bodytemperature - 38.1 - Time : 1458066944.842371
Context : bodytemperature - 38.3 - Time : 1458066944.8425
Context : bodytemperature - 38.9 - Time : 1458066944.842633
Context : bodytemperature - 39.2 - Time : 1458066944.842769

7.3.2 L'analyseur

Pour mettre à jour l'état actuel pour une valeur de contexte lue, le moniteur passe son résultat à l'analyseur après avoir appelé la clause *analyzer/2* ci-dessous :

```
analyzer(C,S2) :-   cstate(C,S1), S2 \== S1,
                    write('...'), write(S1), write(' changed to '), write(S2), nl,
                    retract(cstate(C,S1)),
                    Fact =.. {cstate, C, S2}, assert(Fact),
                    nextTargetState(C,S2).
```

Notons que les fonctions de l'analyseur peuvent être intégrées dans le moniteur lui-même. Dans notre cas, nous l'avons implanté séparément avec la clause *analyzer/2*.

Pour représenter l'état d'un contexte, l'analyseur utilise un prédicat dynamique appelé *cstate/2*. Par exemple, pour le contexte *bodytemperature*, nous représentons l'état normal par la règle suivante :

cstate(bodytemperature, normal)

À chaque fois qu'un changement d'état a été détecté, cette règle contiendra le nouvel état. L'analyseur signale ce nouvel état à la base de connaissances qui effectue cette mise à jour.

Par exemple, si le capteur fournit une donnée qui indique que le nouvel état est *fièvre*, le prédicat *cstate* sera modifié en conséquence:

cstate(bodytemperature, fever)

Si la température fournie par le capteur ne modifie pas l'état courant, l'analyseur restera dans l'état silencieux. Si cette valeur ne fait pas partie des valeurs spécifiées dans les directives fournies au moniteur, un message indiquera que cette valeur n'en fait pas partie. Dans les deux cas, aucun changement d'état n'est notifié.

Notons que dans la plupart des cas un système peut traiter plusieurs contextes en même temps. Par exemple, un système de surveillance de l'état de santé de patients (*Health Monitoring System*) peut traiter de la température corporelle et de la pression artérielle. Dans ce cas nous aurons affaire à un état composite que nous présenterons avec une règle de type:

compositeState(bodytemperature, fever, bloodpressure, normal).

Nous traitons de ce cas plus tard dans ce chapitre.

Cette étape de détermination des changements d'état est illustrée dans l'exemple ci-dessus:

Context	**State**	**State Change**
Context : bodytemperature	state : unspecified !	
Context : bodytemperature	state : unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: unspecified !	
Context : bodytemperature	state: fever	X
Context : bodytemperature	state: normal	X
Context : bodytemperature	state: fever	X
Context : bodytemperature	state: unspecified !	

Context : bodytemperature state: unspecified !
Context : bodytemperature state: unspecified !
Context : bodytemperature state: unspecified !
Context : bodytemperature state: unspecified !

7.3.3 Le planificateur

Chaque état du système déterminé par une valeur du contexte correspond à une configuration donnée. Le rôle du planificateur est d'indiquer les étapes à franchir pour arriver à chacune de ces configurations.

Notons que la configuration peut correspondre soit à des fonctionnalités qui doivent être offertes dans un nouvel état, soit à une architecture que le système doit avoir pour répondre aux exigences de cet état (comme nous le verrons dans la section 7.4).

Comme nous utilisons actuellement le modèle orienté *features* basé sur la variabilité, le planificateur propose une configuration en choisissant les *features* qui la constituent. Cette configuration est exprimée sous forme d'un ensemble d'ensembles de *features*. Chaque sous-ensemble forme une sous-configuration. Dans l'exemple ci-dessous, nous avons une configuration qui a trois sous-configurations:

configuration = {{f1}, {f2}, {f1,f2}}

Cette configuration décrit un système qui peut être construit avec le *feature* f1 seule, le *feature* f2 seul, ou les deux.

Dans l'exemple illustré par la figure 7.6, nous avons deux états *state1* et *state2* d'un contexte. Chaque état est associé à des ensembles de *features*.

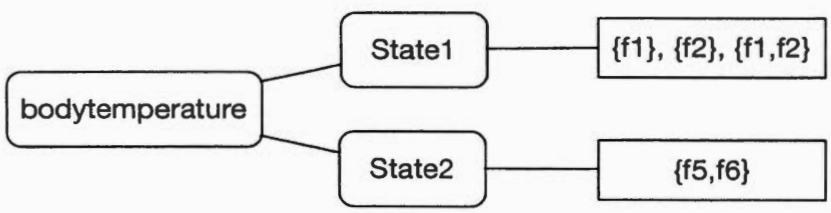


Figure 7.6 Exemple d'associations contexte - configuration.

Ces associations sont exprimées à l'aide de la clause *configuration/3*.

```
configuration(bodytemperature,   normal,   f0_BodyTemperatureNormal).
configuration(bodytemperature,   fever,    f0_BodyTemperatureFever).
```

La base de connaissances retourne le contrôle au planificateur en utilisant la clause suivante:

send_kb(Context, State):- planner(Context, State, Config, Result).

Elle lui fournit le nom du contexte et l'état dans lequel se trouve le système par rapport à ce contexte.

Les clauses du planificateur sont :

```
planner(Context,State1,State2,Config,Result):-
        getConfig(Context, State1, Config1, Result1),
        getConfig(Context, State2, Config2, Result2),
        execute(Result1, Result2).

getConfig(Context,State,Config):-
        configuration(Context,State,Config).
```

Le prédicat *planner/4* prend l'état courant du contexte et détermine sa configuration actuelle grâce à *getConfig/3*. De cette configuration, il extrait les ensembles de *features* qu'elle contient et les présente à l'utilisateur. Celui-ci trouve la configuration courante associée au contexte en utilisant le prédicat *configuration/3*.

La clause *configuration/3* permet de décrire les configurations. Un exemple de configurations pour le contexte *bodytemperature* qui a deux états (normal et fièvre) est:

configuration(bodytemperature, normal, f0_BodyTemperatureNormal).

configuration(bodytemperature, fever, f0_BodyTemperatureFever).

7.3.3.1 Expression de configurations et features

Le modèle de variabilité est basé sur des opérateurs de combinaison de *features* tel que décrit dans la section 5.1 du chapitre 5.

Une *expression* de variabilité d'une configuration permet de décrire les relations entre les *features* ainsi que leurs contraintes d'option et de précédence entre *features*. Ces contraintes pour un *feature* sont:

- *mandatory*: le *feature* doit faire partie de la configuration.
- *optional*: le *feature* peut faire partie de la configuration.
- *requires*: le *feature* exige qu'un autre soit présent
- *excludes*: le *feature* exige qu'un autre soit absent.

Une expression de *features* est écrite selon la grammaire BNF suivante:

E= *feature* | *and*(E,E) | *or*(E,E) | *xor*(E,E) | *requires*(E,E) | *excludes*(E,E) | *optional*(E).

Les opérateurs de composition permettent de construire comme suit un ensemble de *features* tel qu'illustré dans les exemples suivants:

- *and*(f1,f2): les deux *features* sont obligatoires. L'ensemble des *features* est {{f1,f2}}.
- *or*(f1,f2): les deux *features* sont obligatoires ou optionnels. Leurs ensembles sont {{f1}, {f2}, {f1, f2}}.

- *xor*(f1,f2): les deux *features* sont obligatoires ou optionnels. Leurs ensembles sont {{f1}, {f2}}.

Définition 7.2 Expression et combinaison de configurations. La construction des *features* permet de construire des ensembles de *features SF* (pour *Set of Features*) pour chaque *expression* qui représente une configuration. Ces ensembles sont construits comme suit:

SF désigne un ensemble d'ensembles de *features* (exemple : {{f1}, {f2}, {f1, f2}})

Algorithme de construction de l'ensemble de SF :

- SF(F) = {{F}} si F est un feature.
- Si F = *and*(F1, F2), alors S(F) = F.{SF(F1) >< SF(F2)}
- Si F = *or*(F1, F2), alors S(F) = F.{SF(F1), SF(F2), S(F1) >< S(F2)}
- SI F = *xor* (F1, F2), alors S(F) = F.{SF(F1), SF(F2)}
- Si F = *opt*(F1), alors S(F) = F.{{}, SF(F)}

L'opérateur >< est défini comme suit:

$$A >< B = \{a \cup b \mid a \, in \, A, b \, in \, B\}$$

L'opérateur . permet d'ajouter un *feature* à tous les éléments d'un ensemble de *features*:

$$f.A == \{<f, a> \mid a \text{ in } A\}$$

Nous donnons un exemple de modèle simple:

- f1 = *or*(f11, f12)
  - SF(f1) = {{f1, f11}, {f1, f12}, {f1, f11, f12}}
- f2 = *xor*(f21, f22)

- SF(f2) = {{f2, f21}, {f2, f22}}

- f = *and*(f1, f2)

  - SF(f) = {{f, f1, f11, f2, f21}, {f, f1, f11, f2, f22}, {f, f1, f12, f2, f21}, {f, f1, f12, f2, f22}, {f, f1, f11, f12, f2, f21}, {f, f1, f11, f12, f2, f22}}

Nous pouvons illustrer cette procédure à l'aide d'une représentation graphique (voir figure 7.7):

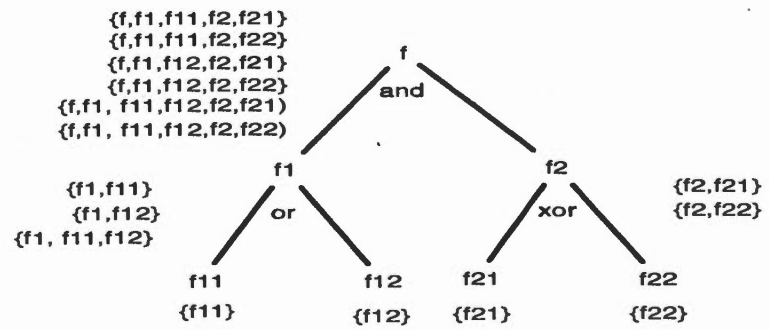


Figure 7.7 Représentation graphique

Nous pouvons représenter ces configurations comme suit (voir figure 7.8):

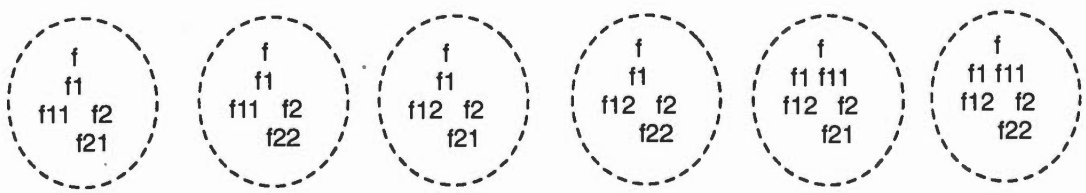


Figure 7.8 Représentation des configurations

Les *features* qui font partie du même ensemble sont celles qui seront activées en même temps pour une valeur de contexte donné. Elles forment une configuration possible.

Pour définir l'ensemble global de *features* d'une application, nous utilisons les clauses *theSetOfFeatures/1* et *isFeature/1* qui déterminent si un élément appartient à cette liste comme:

```
theSetOfFeatures({f, f1, f11, f12, f2, f21, f22}).
isFeature(F):-
        theSetOfFeatures(Set),
        memberOf(F,Set).
```

Pour exprimer les définitions d'un *feature* à l'aide d'autres *features* dans une configuration, nous utilisons la clause *def/2* comme dans l'exemple suivant:

```
def(f, andop(f1, f2)).
def(f1, orop(f11, f12)).
def(f2, xorop(f21, f22)).
```

Pour déclarer les *features* optionnels, nous utilisons la clause *opt/1*. Par exemple, si f11 est optionnel, nous écrirons:

```
opt(f11).
```

Pour obtenir l'ensemble des *features* d'une configuration (fonction SF), nous utilisons les clauses suivantes :

```
setOfFeatures(F,[[F]]):-
        feature(F),!.
setOfFeatures(and(F1,F2), S31):-
        setOfFeatures(F1,S1),
        setOfFeatures(F2,S2),
        unionAnd(S1,S2,S3),S3=[S31].
setOfFeatures(or(F1,F2), Res):-
        setOfFeatures(F1, S1),
        setOfFeatures(F2, S2),
        unionAnd(S1,S2,[S3]),
        union3(S1,S2,S3,Res).
setOfFeatures(xor(F1,F2), S):-
        setOfFeatures(F1,S1),
```

```
                setOfFeatures(F2,S2),
                union(S1,S2,S).
setOfFeatures(opt(F1), [[]|S1]):-
                setOfFeatures(F1,S1),
                union([],S1,S).
```

La clause *go47/0* effectue l'appel à cette fonction.

```
go47 :- def(f, A),
        write('Input: '), write(A),nl,
        setOfFeatures(A, B),
        write('Resultat: '), write(B),
        nl.
```

Dans l'exemple ci-dessous, nous faisons appel à cette procédure:

```
?- go47.
Input: and(f1,or(f2,f3))
Resultat: [[f1,f2],[f1,f3],[f1,f2,f3]]
```

Ce qui indique que l'expression *and(f1,or(f2,f3))* génère trois ensembles de *features* qui sont:*{f1,f2}, {f1,f3}* et *{f1,f2,f3}*

Dans ce qui suit, nous donnons un exemple simple dans lequel nous appliquons la méthode. Un exemple concret sera donné ultérieurement.

```
def(f, andop(f1, or(f2,f3))).
def(f1, andop(f12,f22)).
def(f2, andop(f21,f22)).
def(f3, orop(f31, f32)).
def(f4, xorop(f41, f42, f43)).
def(f41, orop(f411, f412)).
def(f6, andop(f61, f62)).
def(f5, orop(f51, f52)).
def(f61, andop(f611,f612)).
def(f62, andop(f621, f622)).
```

Nous parcourons la hiérarchie des *features* une étape à la fois. À chaque étape, nous donnons les ensembles SF. Nous faisons cela grâce aux clauses ci-dessous.

```
configureApplication(E,S):-
                def(E,F), !,
                setOfFeatures(F, S),
                loopOnFeatures(S, Sel),
                configureApplication(F,_).

configureApplication(E,S):-
                setOfFeatures(E, S),
                write('Configuration : '), write(S),nl,
                loopOnFeatures(S, Sel),
                write('Go to next step (y/n) ?'),read(Answer), write('-Ok-'),
                configureApplication(F,_).

loopOnFeatures([], _):-!.
loopOnFeatures([H|T],Sel):-
        write('Set: '), write(H), nl,
        loopItem(H), loopOnFeatures(T, Sel).

loopItem([]):-!.
loopItem([H|T]):-
        write('Feature: '), write(H),nl,
        listDep(H), loopItem(T).

listDep([]):-!.
listDep(H):-
   atomic(H),
   dep(H,B),
   write('Definition: '), write(H), write(' = '), write(B), nl,
   setOfFeatures(B,S),
   write('   Current feature subset:  '),write(S),nl.
listDep([H|T]):-
   atomic(H),
   dep(H,B),
   write('  -'), write(H), write(' = '), write(B),nl,
   setOfFeatures(B,S),
   write('   Feature subset:  '),write(S),nl,
   listDep(T)
```

Nous déployons ce programme et nous obtenons le résultat ci-dessous. Par exemple pour l'expression *and(f1,or(f2,f3))*, nous obtenons la configuration {{f1,f2,f},{f1,f3,f},{f1,f2,f3,f}} et pour chacun de ces sous-ensembles, le programme donne les configurations correspondantes. Comme dans le sous-ensemble {f1,f2,f}, nous pouvons déployé l'arbre des dépendances des features f1, f2 et f. Sachant que f1=and(f12,f22), cela donne une sous-configuration qui est {{f21,f22,f}} et ainsi de suite. La clause *go48* permet d'implanter ce mécanisme.

?- go48.
Model: and(f1, or(f2,f3))
Configuration : [[f1,f2,f],[f1,f3,f],[f1,f2,f3,f]]
Set: [f1,f2,f]
Feature: f1
    Definition: f1 = and(f12,f22)
    Current feature subset: [[f12,f22,f]]
Feature: f2
    Definition: f2 = and(f21,f22)
    Current feature subset: [[f21,f22,f]]
Feature: f
    Definition: f = and(f1,or(f2,f3))
    Current feature subset: [[f1,f2,f],[f1,f3,f],[f1,f2,f3,f]]
Set: [f1,f3,f]
Feature: f1
    Definition: f1 = and(f12,f22)
    Current feature subset: [[f12,f22,f]]
Feature: f3
    Definition: f3 = or(f31,f32)
    Current feature subset: [[f31,f],[f32,f],[f31,f32,f]]
Feature: f
    Definition: f = and(f1,or(f2,f3))
    Current feature subset: [[f1,f2,f],[f1,f3,f],[f1,f2,f3,f]]
Set: [f1,f2,f3,f]
Feature: f1
    Definition: f1 = and(f12,f22)
    Current feature subset: [[f12,f22,f]]
Feature: f2
    Definition: f2 = and(f21,f22)
    Current feature subset: [[f21,f22,f]]
Feature: f3

Definition: f3 = or(f31,f32)
Current feature subset: [[f31,f],[f32,f],[f31,f32,f]]
Feature: f
Definition: f = and(f1,or(f2,f3))
Current feature subset: [[f1,f2,f],[f1,f3,f],[f1,f2,f3,f]]

Go to next step (y/n) ?

7.3.3.2 Contraintes de précédence

Dans les modèles de variabilité, nous pouvons avoir des contraintes de précédence entre deux *features* en utilisant la relation *requires* et *excludes* définies par les clauses *requires/2* et *excludes/2* ci-dessous :

- Le *feature* F1 exige que le *feature* F2 soit présent: F1 *requires* F2
- Si le *feature* F1 est présent, alors le *feature* F2 ne peut être présent: F1 *excludes* F2

Pour implanter des contraintes, nous utilisons les règles suivantes :

requires(F1,F2) :- *isFeature*(F1), *isFeature*(F2).

excludes(F1,F2) :- *isFeature*(F1), *not*(*isFeature* F2)).

Dans notre exemple, nous avons les contraintes suivantes:

- F5Alarme *requires* (F412PrioriteAlerte ^ F422AvertisseurSonore)
- F412Priorite *requires* F422AvertisseurSonore
- F411Fifo *excludes* F422AvertisseurSonore
- F5Alarme *excludes* F7PlanSoins

Ces contraintes sont représentées par les règles suivantes:

- *requires*(F5Alarme, [F412PrioriteAlerte, F422AvertisseurSonore].
- *requires*(F412Priorite, [F422AvertisseurSonore]).
- *excludes*(F411Fifo, [F422AvertisseurSonore]).
- *requires*(F5Alarme, [F7PlanSoins]).

7.3.3.3 Exemple

Pour prendre un exemple plus concret, considérons le modèle de *features* donnée dans la section 5.1 du chapitre 5 (voir figure 7.9).

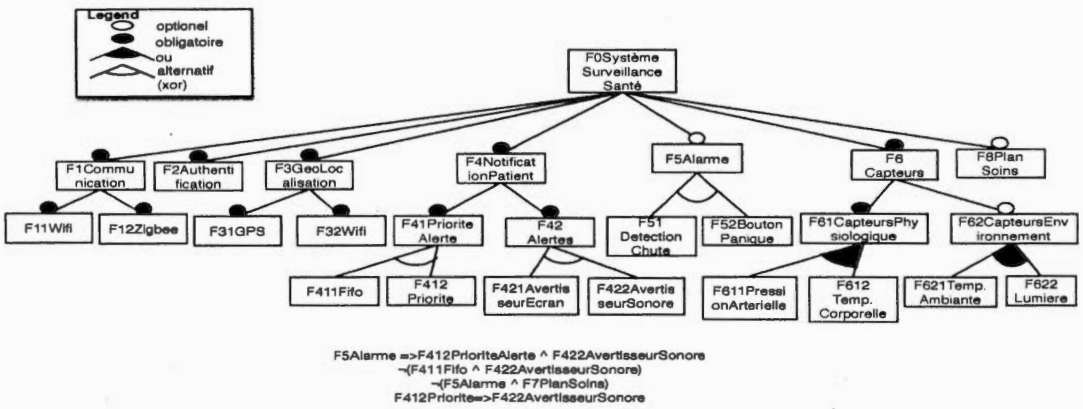


Figure 7.9 Modèle de *features* d'un système de surveillance de l'état de santé de patients

Nous appliquons cette procédure aux *features* de notre système de surveillance de l'état de santé de patients:

theSetOfFeatures(
[F0SystemeSurveillanceSante, F1Communication, F11Wifi, F12Zigbee, F2Authentification, F3GeoLocalisation, F31GPS, F32Wifi, F4NotificationPatient,
F41PrioriteAlerte, F41PrioriteAlerte, F411Fifo, F412Priorite, F42Alertes, F421AvertisseurEcran, F422avertisseurSonore, F5Alarme, F51DetectionChute, F52BoutonPanique, F6Capteurs,

F61CapteursPhysiologique, F611PressionArterielle, F612TempCorporelle, F62CapteursEnvironnement, F621TempAmbiante, F622Lumiere, F7PlanSoins]
).

def(F0SystemeSurveillanceSante,
andop(F1Communication,F2Authentification,F3GeoLocalisation,
F4NotificationPatient, F5Alarme, F6Capteur, F7PlanSoins)
).

def(F1Communication, andop(F11Wifi, F12Zigbee)).
def(F3GeoLocalisation, andop(F31GPS, F32Wifi)).
def(F4NotificationPatient, andop(F41PrioriteAlerte, F42Alertes)).
def(F41PrioriteAlerte, xorop(F411Fifo, F412Priorite)).
def (F42Alertes, xorop(F421AvertisseurEcran, F422AvertisseurSonore)).
def (F5Alarme, xorop(F51DetectionChute, F52BoutonPanique)).
def (F6Capteurs,
andop(F61CapteursPhysiologique, F62CapteursEnvironnement)).
def (F61CapteursPhysiologique, orop(F611PressionArterielle, F612TempCorporelle)).
def (F62CapteursEnvironnement , orop(F621TempAmbiante, F622Lumiere)).

opt(F5Alarm).
opt(F7PlanSoins).
opt(F62CapteursEnvironnement).

À l'aide d'une représentation graphique, nous illustrons les configurations possibles du système (voir figure 7.10).

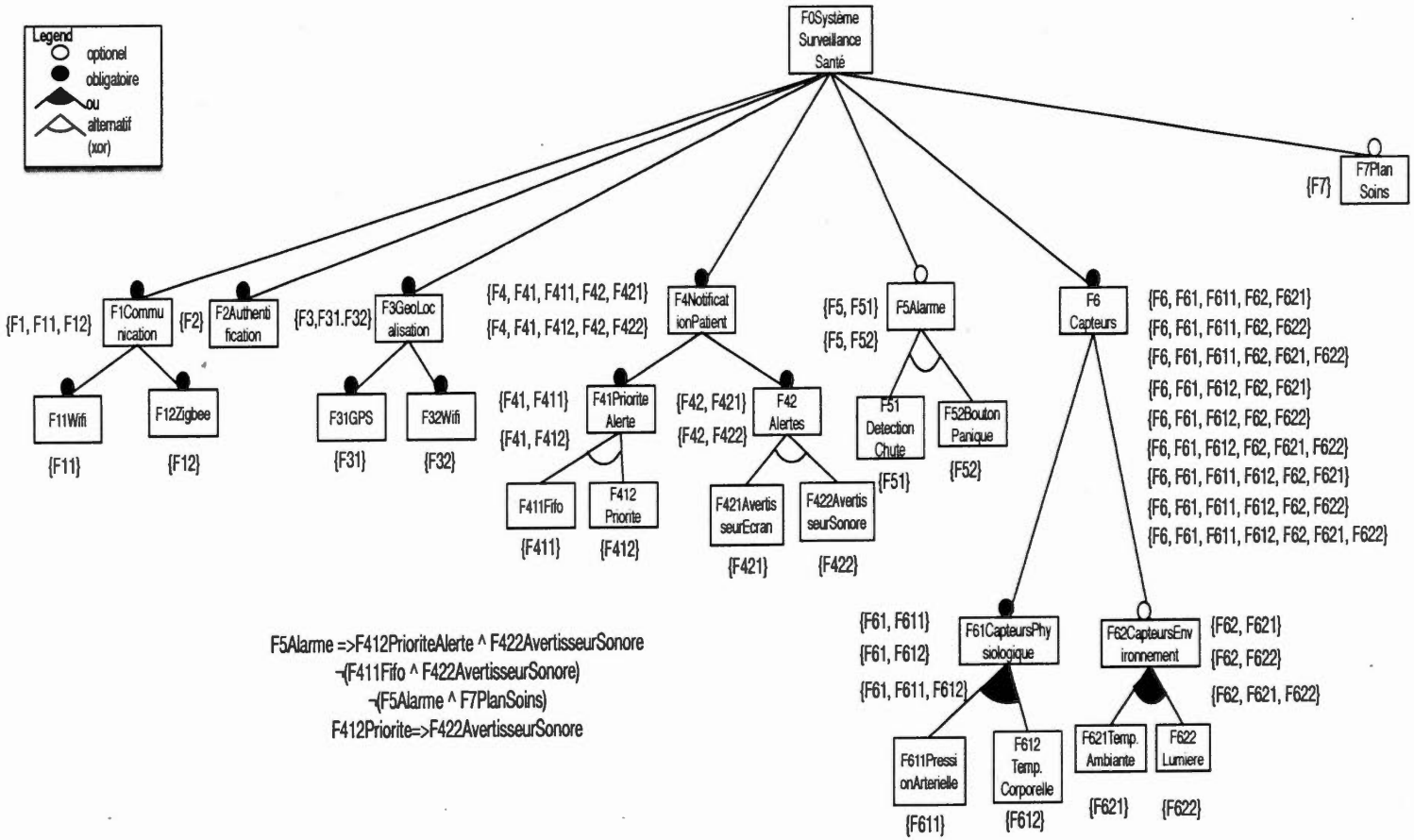


Figure 7.10 Représentation graphique des configurations du système de surveillance de l'état de santé de patients

Dans la section 5.5.3 du chapitre 5, nous avions utilisé 3 exemples de configurations valides à partir du modèle de variabilité du système de surveillance de l'état de santé de patients représenté par la figure 7.10. Nous obtenons 36 configurations valides pour le système de surveillance de l'état de santé de patients que nous donnons dans le tableau 7.1. Nous montrons ces exemples ci-dessous. Les identificateurs de *features* que nous avons utilisé sont celles qui apparaissent dans les nœuds du modèle de variabilité de la figure 7.10.

- Configuration valide pour l'état normal : {F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F611, F612}
- Configuration valide pour l'état de pression artérielle faible : {F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F611}
- Configuration valide pour l'état de fièvre et de pression artérielle élevée : {F0, F1, F11, F412, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612}

Tableau 7.1 Configurations valides pour le système de surveillance de l'état de santé de patients

Configurations du système de surveillance de l'état de santé de patients
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F62, F621, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F612, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F612, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F612, F62, F621, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F62, F621, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F62, F621, F7}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F62, F622, F7}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F62, F621, F622, F7}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F612, F62, F621, F7}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F612, F62, F622, F7}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F612, F62, F621, F622, F7}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F62, F621, F7}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F62, F622, F7}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F62, F621, F622, F7}

{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F611, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F611, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F611, F62, F621, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F612, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F612, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F612, F62, F621, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F611, F612, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F611, F612, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F51, F6, F61, F611, F612, F62, F621, F622}

{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F611, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F611, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F611, F62, F621, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F612, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F612, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F612, F62, F621, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F611, F612, F62, F621}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F611, F612, F62, F622}
{F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F5, F52, F6, F61, F611, F612, F62, F621, F622}

7.3.4 L'exécuteur

7.3.4.4 Gestion des configurations

La procédure décrite précédemment permet de déterminer l'ensemble des configurations qu'un système peut offrir. Durant son exécution, un système dépendant du contexte peut passer d'une configuration à une autre en fonction des changements d'état.

À chaque changement d'état, nous proposons une ou plusieurs configurations qui lui sont associées. Dans certains cas, ces différentes configurations ont des *features* en commun. Pour adapter un système à ces changements, nous pouvons déterminer la différence entre elles et celles de l'état précédent.

Cette adaptation en "temps réel" permet de faire une adaptation dynamique.

Définition 7.3 Différences entre configurations. Soit $C_A = \{af_1, af_2, \ldots, af_n\}$ et $C_B = C_A = \{bf_1, bf_2, \ldots, bf_m\}$ deux ensembles de configurations. Nous construisons les deux ensembles suivants:

- *plus*(C_A, C_B) = {b – c, c in C_A , b in C_B} est l'ensemble des *features* qui sont dans une configuration de C_A et non dans une configuration de C_B.

- *less*(C_A,C_B) = {c – b, c in C_A , b in C_B} est l'ensemble des *features* qui sont dans une configuration de C_B et non dans une configuration de C_A.

Prenons C_A = {{f1, f2, f3, f4}, {f2, f4, f6}} et C_B = {{f1, f5}, {f2, f4}}. *plus*(C_A,C_B) et *less*(C_A,C_B) sont résumés dans les tableaux 7.2 et 7.3.

Tableau 7.2 Configuration *plus*(C_A,C_B)

C_A	C_B	*plus*(C_A,C_B)	**Description**
{f1, f2, f3, f4}	{f1, f5}	{f2, f3, f4}	Les *features* à ajouter pour passer de la configuration C_A à C_B sont l'authentification, la géolocalisation et la notification du patient.
{f1, f2, f3, f4}	{f2, f4}	{}	Il n'y a pas de *feature* à ajouter pour passer de la configuration C_A à C_B.
{f2, f4, f6}	{f1, f5}	{f2, f4, f6}	Les *features* à ajouter pour passer de la configuration C_A à C_B sont l'authentification, la notification du patient et les capteurs.
{f2, f4, f6}	{f2, f4}	{f6}	Le *feature* à ajouter pour passer de la configuration C_A à C_B est le capteur.

Tableau 7.3 Configuration *less*(C_A,C_B)

C_B	C_A	*less*(C_A,C_B)	**Description**
{f1, f5}	{f1, f2, f3, f4}	{f5}	Le *feature* à ajouter pour passer de la configuration C_B à C_A est l'alarme.
{f1, f5}	{f2, f4, f6}	{f1, f5}	Les *features* à ajouter pour passer de la configuration C_B à C_A sont la communication et l'alarme.
{f2, f4}	{f1, f2, f3, f4}	{}	Il n'y a pas de *feature* à ajouter pour passer de la configuration C_B à C_A.
{f2, f4}	{f2, f4, f6}	{}	Il n'y a pas de *feature* à ajouter pour passer de la configuration C_B à C_A.

La première ligne du tableau 7.2 indique que pour passer de la configuration {f1, f2, f3, f4} à {f1, f5}, il faut ajouter les *features* {f2, f3, f4}. La ligne 1 du tableau 7.3 indique le changement inverse, c.-à-d. que pour passer de la configuration {f1, f5} à {f1, f2, f3, f4}, il faut ajouter les *features* {f5}.

7.3.4.5 Graphe d'adaptation

Le graphe d'adaptation permet de décrire ces changements de configurations à l'aide d'un graphe G qui décrit le passage d'une étape à la suivante dans l'exécution du système. G = <n_0, N, T> où n_0 est le nœud de configuration initiale, N est l'ensemble de nœuds et L est un ensemble de transitions dans NxLxN -> N de la forme <n_1, l, n_2> où n_1 et n_2 sont des nœuds (représentant un état de configuration) et l est une fonction de transition définie par l=*plus*(n_1, n_2)/*less*(n_2,n_1) (voir figure 7.11).

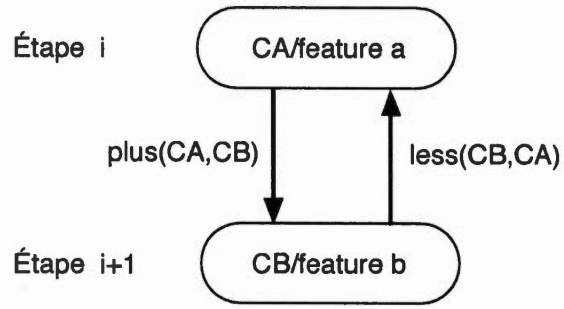


Figure 7.11 Exemple de transition

En utilisant les configurations C_A et C_B ci-dessus, nous obtenons les graphes illustrés par la figure 7.12.

Étape i: $C_A = \{\{f1, f2, f3, f4\}, \{f2, f4, f6\}\}$

Étape i+1: $C_B = \{\{f1, f5\}, \{f2, f4\}\}$

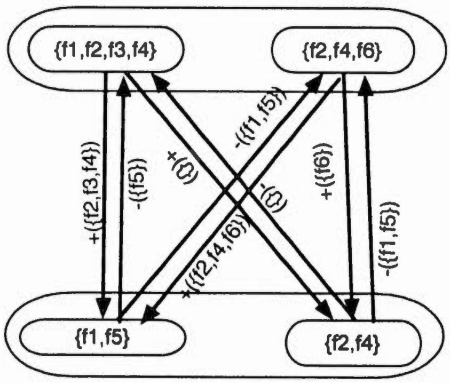


Figure 7.12 Graphes de configuration

Pour calculer les différences entre les configurations, nous utilisons les clauses *checkOneSide/3*, *difference/3* et *checkall/2* listées ci-dessous.

Les clauses *checkOneSide/3*, *difference/3* et *checkall/3* déterminent la différence entre deux configurations dans deux étapes consécutives (étape i et étape i+1). Elles déterminent également les différences dans le sens inverse (c.-à-d. de l'étape i+1 vers

l'étape i). Ce qui permet de passer d'un état à l'état précédent et ainsi éviter des traitements inutiles (ce qui risque d'être assez souvent le cas) :

```
checkOneSide([],_,_):-!.
checkOneSide([H1|T],L,[R|Res]):-
        difference(H1,L,R),
        write('Difference entre '), write(H1), write(' et '), write(L),
        write(': + '), write(R),nl,
checkOneSide(T,L,Res).

checkall(X1, L, X2):-
        checkOneSide(X1, L, X2),
        checkOneSide(X2, L, X1).

difference(_,[],[]):-!.
difference(H1,[H2|L],[R|Res]):-
        notcommon(H1,H2,R),
difference(H1,L,Res).
```

L'adaptation consiste à utiliser le graphe d'adaptation à chaque étape et à suivre les actions spécifiées dans les transitions. Dans notre approche, nous utilisons une approche itérative. À chaque étape, nous déterminons les ensembles de *features*. Ensuite, pour chaque *feature* non terminale appartenant à un de ces ensembles, nous pouvons la déployer pour compléter la configuration.

Dans ce qui suit, nous appliquons cette méthode au système F0SystemeSurveillanceSante.

Supposons les configurations C_A et C_B. Chacune de ces configurations est composée de deux ensembles de configurations pour un contexte donné. C_A et C_B sont respectivement des ensembles de configurations pour les contextes d'état normal et de basse pression artérielle d'un patient comme suit :

- C_A= {

  {F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612},

  {F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F62, F621, F622, F7}

  }

- C_B= {

  {F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F412, F42, F421, F5, F51, F6, F61, F611, F62, F621},

  {F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F412, F42, F421, F5, F51, F6, F61, F611, F62, F621, F622}

  }

Notons que nous identifions les ensembles de C_A comme C_{A1} et C_{A2} pour le premier et second ensemble, respectivement. Nous identifions également les ensembles de C_B comme C_{B1} et C_{B2} pour le premier et second ensemble, respectivement. En utilisant les opérateurs *plus* et *less*, nous obtenons les résultats présentés dans le tableau 7.4 et 7.5.

Tableau 7.4 Configuration *plus*(C_A, C_B)

C_A	C_B	*plus*(C_A, C_B)	Description
C_{A1}	C_{B1}	{F411, F421, F612}	Les *features* à ajouter pour passer de C_{A1} à C_{B1} sont FIFO (priorité), l'avertisseur sonore et la temp. ambiante (capteur).
C_{A1}	C_{B2}	{F411, F421, F612}	Les *features* à ajouter pour passer de C_{A1} à C_{B2} sont FIFO (priorité), l'avertisseur sonore et la temp. ambiante (capteur).
C_{A2}	C_{B1}	{F411, F421, F612, F622, F7}	Les *features* à ajouter pour passer de C_{A2} à C_{B1} sont FIFO (priorité), l'avertisseur sonore, la temp. ambiante (capteur), la lumière (capteur) et le plan pour des soins.
C_{A2}	C_{B2}	{F411, F421, F612, F7}	Les *features* à ajouter pour passer de C_{A2} à C_{B2} sont FIFO (priorité), l'avertisseur sonore, la temp. ambiante (capteur) et le plan pour des soins.

Tableau 7.5 Configuration *less*(C_A, C_B)

C_B	C_A	*less*(C_A, C_B)	Description
C_{B1}	C_{A1}	{F412, F422, F5, F51, F62, F621}	Les *features* à ajouter pour passer de C_{B1} à C_{A1} sont la priorité (alerte), l'avertisseur sonore, l'alarme, la détection de chute, les capteurs d'environnement et la temp. ambiante (capteur).
C_{B2}	C_{A1}	{F412, F422, F5, F51}	Les *features* à ajouter pour passer de C_{B2} à C_{A1} sont la priorité (alerte), l'avertisseur sonore, l'alarme et la détection de chute.
C_{B1}	C_{A2}	{F412, F422, F5, F51, F62, F621, F622}	Les *features* à ajouter pour passer de C_{B1} à C_{A2} sont la priorité (alerte), l'avertisseur sonore, l'alarme, le bouton de panique, les capteurs d'environnement, la temp. ambiante (capteur) et la lumière (capteur).
C_{B2}	C_{A2}	{F412, F422, F5, F51}	Les *features* à ajouter pour passer de C_{B2} à C_{A2} sont la priorité (alerte), l'avertisseur sonore, l'alarme et le bouton de panique.

La première ligne du tableau 7.4 indique que pour passer de la configuration C_{A1} à C_{B1}, il faut ajouter les *features* {F411, F421, F612}. La première ligne du tableau 7.5 indique le changement inverse, c'est-à-dire que pour passer de la configuration C_{B1} à C_{A1}, il faut ajouter les *features* {F412, F422, F5, F51, F62, F621}.

Le graphe d'adaptation pour cet exemple est présenté par la figure 7.13.

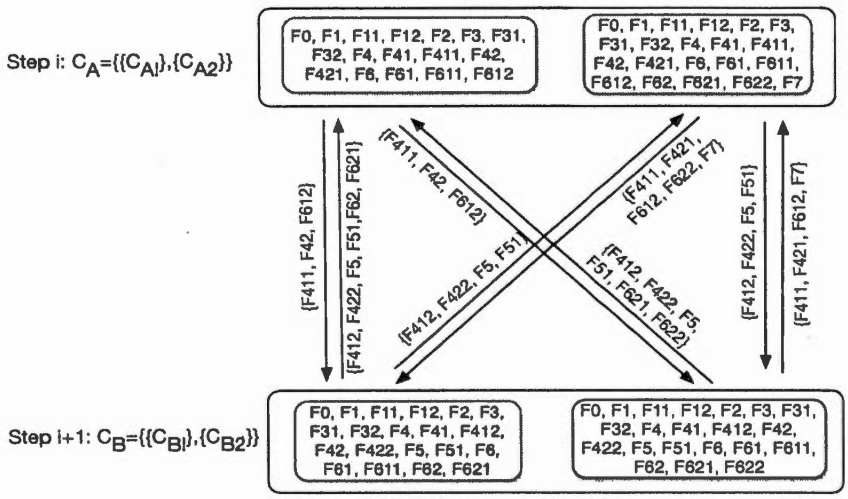


Figure 7.13 Graphe d'adaptation

Nous avons adapté le système dans le cas ou nous avons deux contextes: la température corporelle et pression artérielle avec les données suivantes:

Tableau 7.6 Intervalles de valeurs de contexte

Contexte	État	Catégorie	Intervalle
Température corporelle	hypo	temp	30 - 35.5
	normal	temp	35.5 - 37.5
	fever	temp	37.5 - 42
Pression artérielle	high	diast	90 - 120
		syst	140 - 180
	low	diast	40 - 60
		syst	60 - 90
	normal	diast	60 - 80
		syst	90 - 120

Les directives émises par la base de connaissances au moniteur sont exprimées par les règles suivantes:

profiles([bodytemperature, bloodpressure]).

profile(bodytemperature, [id:id_bt, type:float, detla:1,
svalues:[temp:[30, 42]],
intervals:[hypo: [temp:[30, 35.5]],
normal: [temp:[35.5, 37.5]],
fever: [temp:[37.5, 42]]]]).
profile(bloodpressure, [id: id_bp, type: integer,
svalues:[diast:[40, 120], syst:[60, 180]],
intervals:[high: [diast:[90, 120], syst:[140, 180]],
low: [diast:[40, 60], syst:[60, 90]],
normal: [diast:[60, 80], syst:[90, 120]]]]).

Notons que nous avons considéré trois cas de combinaisons de valeurs de contexte. Les autres ont été omis. Nous avons donc défini les configurations suivantes:

1. **Configuration 1:**
    a. État: Température normale et pression artérielle normale
    b. *Features*:
        i. {f0, f1, f11, f12, f2, f3, f31, f32, f4, f41, f411, f42, f421, f6, f61, f611, f612}
        ii. {f0, f1, f11, f12, f2, f3, f31, f32, f4, f41, f411, f42, f421, f6, f61, f611, f612, f7}
2. **Configuration 2**:
    a. État: Fièvre et pression artérielle élevée
    b. *Features*:
        i. {f0, f1, f11, f12, f2, f3, f31, f32, f4, f41, f412, f42, f422, f5, f52, f6, f61, f611, f612}
        ii. {f0, f1, f11, f12, f2, f3, f31, f32, f4, f41, f412, f42, f422, f5, f51, f6, f61, f611, f612}
3. **Configuration 3:**
    a. État: Hypothermie et pression artérielle basse
    b. *Features*:

i. {f0, f1, f11, f12, f2, f3, f31, f32, f4, f41, f412, f42, f422, f5, f52, f6, f61, f611, f612}
ii. {f0, f1, f11, f12, f2, f3, f31, f32, f4, f41, f412, f42, f422, f5, f52, f6, f61, f611, f612}

Nous montrons un exemple d'exécution de notre simulateur sur cet exemple.

Les règles créées par la base de connaissances pour le fonctionnement du système sont:

Regle: id(bodytemperature,id_bt)
Regle: type(bodytemperature,float)
Regle: detla(bodytemperature,1)
Regle: svalues(bodytemperature,[temp:[30,42]])
Regle: intervals(bodytemperature,[
hypo:[temp:[30,35.5]],
normal:[temp:[35.5,37.5]],
fever:[temp:[37.5,42]]])
Regle: id(bloodpressure,id_bp)
Regle: type(bloodpressure,integer)
Regle: svalues(bloodpressure,[diast:[40, 120], syst:[60, 180]])
Regle: intervals(bloodpressure,[
high:[diast:[90, 120], syst:[140, 180]],
low:[diast:[40, 60], syst:[60, 90]],
normal:[diast:[60, 80], syst:[90, 120]]])

Le moniteur détermine les états du système en considérant les deux contextes. Par exemple, la ligne 3 indique que dans l'état normal la configuration est [F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612]. La ligne 7 indique que le passage de l'état fièvreux à l'état normal reconfigure le système à [F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F7].

Tableau 7.7 État du système selon le contexte

Numéro de ligne	Contexte	Valeur	*Time stamp*	État
1	bodytemperature	35.4	1470855360.804492	hypo
2	bloodpressure	78,120	1470855360.804727	normal
3	Reconfig.	[F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612]		
4	bodytemperature	37.6	1470855360.805003	fever
5	bloodpressure	80,120	1470855360.805064	
6	bodytemperature	36.6	1470855360.805134	normal
7	Reconfig.	[F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612, F7]		
8	bloodpressure	90,125	1470855360.805304	
9	bodytemperature	35.6	1470855360.805373	
10	bloodpressure	100, 140	1470855360.805406	high
11	bodytemperature	35.7	1470855360.805495	
12	bloodpressure	100, 142	1470855360.805528	
13	bodytemperature	35.8	1470855360.805591	
14	bloodpressure	60, 110	1470855360.805623	normal
15	bodytemperature	35.7	1470855360.805758	
16	bloodpressure	60, 120	1470855360.805793	
17	bodytemperature	35.9	1470855360.805885	
18	bloodpressure	80, 120	1470855360.805933	
19	bodytemperature	36.0	1470855360.805998	
20	bloodpressure	100, 160	1470855360.806031	high
21	bodytemperature	36.2	1470855360.806123	
22	bloodpressure	100, 150:	1470855360.806161	
23	bodytemperature	36.1	1470855360.806214	
24	bloodpressure	100, 170	1470855360.806244	
25	bodytemperature	37.6	1470855360.806292	fever
26	bloodpressure	50,90	1470855360.806337	low
27	bodytemperature	37.5	1470855360.806406	normal

28	Reconfig.	[F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F411, F42, F421, F6, F61, F611, F612]		
29	bloodpressure	60, 80	1470855360.806636	
30	bodytemperature	37.9	1470855360.806693	fever
31	bloodpressure	100, 160	1470855360.806755	high
32	bodytemperature	32.0	1470855360.806831	hypo
33	Reconfig.	[F0, F1, F11, F12, F2, F3, F31, F32, F4, F41, F412, F42, F422, F6, F61, F611, F612]		
34	...	...	...	...

7.4 Adaptation par recomposition

Grâce à des opérateurs de transformation, nous pouvons substituer certains opérateurs par d'autres. Nous avons également introduit le concept de contraintes qui peut être utilisé dans le processus d'adaptation comme nous l'expliquerons plus loin. Ces deux mécanismes permettent ainsi de faire des changements d'architecture et faire en sorte que le système s'adapte aux changements des valeurs du contexte.

Pour effectuer cette adaptation, nous utiliserons les éléments suivants:

- Conditions sur les valeurs du contexte : Ces valeurs seront acquises (par l'intermédiaire de capteurs) et analysées pour déterminer si nous devons effectuer l'adaptation.
- Conditions sur des intervalles de valeurs du contexte : Les valeurs captées ne sont pas toutes considérées par le système. Ce qui nous intéresse souvent ce sont les intervalles de valeurs qui correspondent à des contextes différents

plutôt que des valeurs spécifiques. Par exemple, l'état fièvre correspond à l'intervalle [37.5, 42].

- Les traces d'exécution qui déterminent les points de transformation dans le comportement du système : Selon l'état dans le comportement du système, des transformations doivent être appliquées. Pour atteindre l'état ciblé, nous devons passer par des états intermédiaires. Chacun de ces états est identifié par une séquence d'actions formant une trace d'exécution.
- Les contraintes à appliquer au système afin de restreindre son comportement : Selon l'état du système et la valeur du contexte, nous pouvons restreindre le comportement global du système à un sous-comportement en particulier. Une contrainte est elle-même considérée comme un composant qui peut être utilisé comme masque pour le comportement global.

Pour effectuer cette adaptation, nous allons suivre le même modèle de boucle de contrôle ayant une base de connaissances, un moniteur, un analyseur, un planificateur et un exécuteur.

La base de connaissances est structurée autour d'un *Gabarit d'adaptation.*

Définition 7.4 Gabarit d'adaptation. Il s'agit d'un fichier qui donne les directives pour effectuer l'adaptation par composition et transformation des systèmes. Ce gabarit contient les données suivantes:

- *name*: contient le nom du contexte. Exemple:
  - name:bodytemperature.

- *svalues: categorie :*$<e_1:i_1, e_2:i_2, ..., e_n:i_n>$ indique l'intervalle des valeurs possibles d'une catégorie. Catégorie désigne la catégorie des données. Par exemple, dans le cas des données sur les températures corporelles, il existe une seule catégorie que nous avons appelée *oneset.*

- *Intervals : {Categorie : <e_1:i_1, e_2:i_2, ..., e_n:i_n>, ...}* indique un ensemble d'intervalles de valeurs possibles pour ce contexte. À chaque intervalle est associé un état qui l'identifie et auquel nous devons associer un traitement spécifique. Ce dernier pourra être implémenté par un composant ou un *feature*.

- *transforms*: désigne des transformations à appliquer après avoir exécuté une séquence d'actions décrite dans la rubrique *sequences* présentée plus loin. Exemple:
    - *transforms: [[t1, seq1, par, alt], [t2, seq2, par, pipe]]*: indique deux transformations: *t1* est appliquée après la séquence *seq1* et qui remplace l'opérateur *par* par *alt* et *t2* qui s'applique après *seq2* et qui transforme *par* par *pipe*.
- *transformApply*: décrit les états dans lesquels nous devons appliquer chaque transformation. Nous y indiquons également, le nom de la transformation et le niveau dans lequel nous devons appliquer cette transformation. Exemple:
    - *transformApply:[[normal,t1,0,0],[fever,t2,0,0]]*: indique que *t1* s'applique dans l'état *normal* et *t2* dans l'état *fever*. Nous indiquons également les transformations se font à la racine de chaque nœud.
- *constraintApply*: indique une liste d'états et la contrainte à appliquer à chaque état. Exemple:
    - *constraintApply:[[normal,c2],[fever,c1]]*: indique que la contrainte *c1* s'applique dans l'état fiévreux et que la contrainte *c2* s'applique dans l'état normal.
- *constraintAfter*: désigne la séquence après laquelle nous devons appliquer les contraintes. Cela permet d'être plus spécifique quant à l'étape durant laquelle nous devons appliquer chaque contrainte. Si cette directive n'est pas présente, cela veut dire que nous l'appliquons depuis le début du comportement du système. Exemple:

- *constraintAfter:[[c1,seq1],[c2,seq2]]*: indique que la contrainte *c1* s'applique après avoir exécuté la séquence *seq1* et que *c2* s'applique après *seq2*.

- *constraints* : donne la liste des contraintes à appliquer. Chaque contrainte est identifiée par son identificateur. Exemple:
  - *constraints: [*
    [c1,seq(location,seq(hospital,seq(fever,seq(medication,null)))),
    [c2,seq(location,seq(hospital,seq(normal,seq(nomedication,null))))]]

- *sequences*: indique la liste des identificateurs des séquences et leurs contenus. Exemple:
  - *sequences:[*
    [seq1,[location,hospital]],
    [seq2,[location,hospital]]]

Voici en exemple de gabarit pour le contexte *bodytemperature* (voir figure 7.14). Il est représenté par une clause appelée *gabarit/2*.

```
gabarit(bodytemperature,
        [name: bodytemperature,
        sValues:[35,41],
        Intervals:[[normal, [36,37.5]], [fever, [37.5,40]]],
        transforms:[[t1, seq1, par, alt],[t2,seq2,par,pipe]],
        transformApply:[[normal,t1,0,0],[fever,t2,0,0]],
        constraintApply:[[normal,c2],[fever,c1]],
        constraintAfter:[[c1,seq1],[c2,seq1]],
        constraints:[
                [c1,seq(location,seq(hospital,seq(fever,seq(medication,null))))],
                [c2,seq(location,seq(hospital,seq(normal,seq(nomedication,null))))]]
        sequences:[
                [seq1,[location,hospital]],
                [seq2,[location,hospital]]]
    ]).
```

Figure 7.14 Gabarit d'adaptation du contexte *bodytemperature*

À partir de cette description, nous pouvons effectuer l'adaptation grâce à l'application de contraintes et de transformations des opérateurs.

Nous donnons ici un exemple de système simplifié qui consiste à traiter les symptômes de la température corporelle chez un patient. Nous supposons que le patient sent une montée de fièvre. Si cette hausse est importante, il doit être traité en se rendant à un hôpital qui se trouve dans son voisinage. La spécification simplifiée d'un tel système appelé *bodytemperature* et son architecture sont données ci-dessous (voir figure 7.15). Nous en fournissons par la suite une implantation et nous l'exécutons en nous basant sur l'implémentation en Prolog de nos règles sémantiques décrites dans le chapitre 6.

Dans cet exemple, le composant *transport* s'exécute en pipeline avant le reste du système. Une fois que se composant s'exécute, le système donne deux choix qui sont représentés par les composants *treatment* et *notreatment*. Le composant treatment est mis en parallèle avec 2 composants qui correspondent l'un au cas fiévreux et l'autre au cas de température normale. Ces deux derniers composants sont en parallèle, mais sans synchronisation.

```
bodytemperature =
                    call(transport, {location}, {hospital. nohospital}) *
                    ( (call(treatment, {hospital}, {normal,fever})
                      ||  (call(feverState, {fever},{medication})
                        ||| call(normalState, {normal}, {nomedication})
                          )
                      )
                      + call(notreatment, {nohospital}, {nomedication})
                    )
```

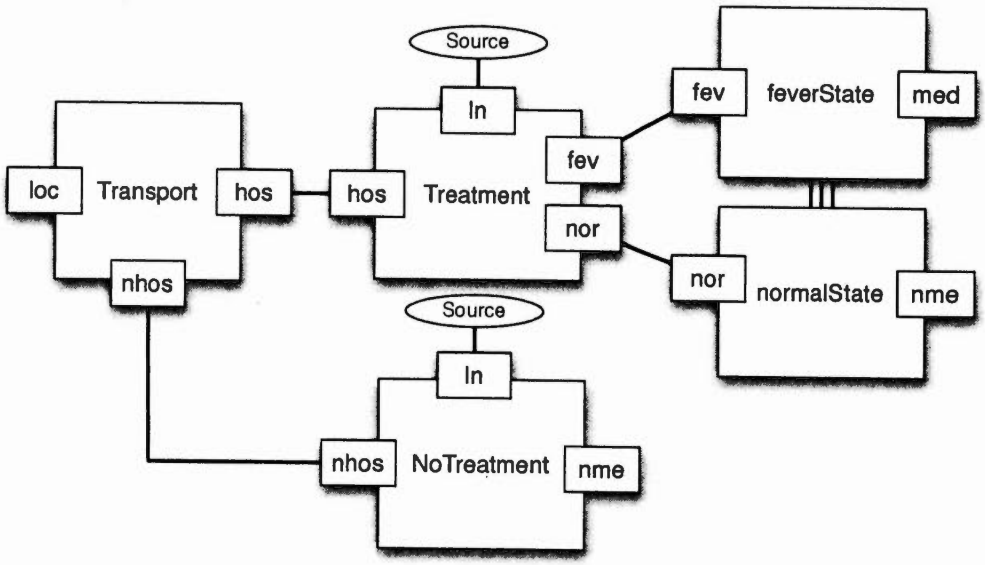


Figure 7.15 Architecture de l'application *bodytemperature*

Sa représentation en Prolog est :

```
process(bodytemperature,
        pipe(
             call(transport, [location], [nohospital,hospital]),
             alt(par(
                      call(treatment, [hospital], [normal,fever]),
                          intl(
                               call(feverState, [fever], [medication]),
                               call(normalState, [normal],[nomedication])
                              )
                     ),
                     call(notreatment, [nohospital], [nomedication]))
                )
           ).
```

Dans un premier temps, nous présentons une partie de l'exécution du système sans utiliser les outils d'adaptation. Pour cela, nous allons utiliser la clause *loop/1* qui utilise les clauses *do/2* et *nth/3* données ci-dessous. La clause *loop/1* permet de parcourir le système et d'exécuter une action à la fois. À chaque étape, elle demande

à l'utilisateur de choisir l'action qu'il veut exécuter. Les clauses *do/2* et *nth/2* permettent d'afficher le résultat.

```
loop(P):-  bagof([Action, Next], infer(P, Action, Next), Res),
        do2(Res,1),
        read(N),
        (       N == "exit" -> true;
                nth(Res, N, [A,New]),
                loop(New)
        ).
do2([],_):-!.
do2([[A,R]|X],N):-
        write(N), write("-"), write(A),write("->"), nl,
        write(R) , nl,
        N1 is N+1, do2(X,N1).

nth([[A,Next]|X],1,[A,Next]):-!.
nth([_|X], N, New):-  N > 1, N1 is N-1, nth(X,N1, New).
```

Le résultat de l'exécution est :

```
?-process(bodytemperature,P),loop(P).
1-location->
        pipe(call(transport,[],[nohospital,hospital]),
        alt(par(call(treatment,[hospital],[normal,fever]),
        intl(call(feverState,[fever],[medication]),
        call(normalState,[normal],[nomedication]))),
        call(notreatment,[nohospital],[nomedication])))
|: 1.
1-nohospital->
        call(notreatment,[],[nomedication])
2-hospital->
        par(call(treatment,[],[normal,fever]),
        intl(call(feverState,[fever],[medication]),
        call(normalState,[normal],[nomedication])))
|: 2.
1-normal->
        par(call(treatment,[],[fever]),
        intl(call(feverState,[fever],[medication]),
        call(normalState,[normal],[nomedication])))
2-fever->
```

```
        par(call(treatment,[],[fever]),
        intl(call(feverState,[fever],[medication]),
        call(normalState,[normal],[nomedication])))
3-fever->
        par(call(treatment,[],[normal,fever]),
        intl(call(feverState,[],[medication]),
        call(normalState,[normal],[nomedication])))
4-normal->
        par(call(treatment,[],[normal,fever]),
        intl(call(feverState,[fever],[medication]),
        call(normalState,[],[nomedication])))
5-normal->
        par(call(treatment,[],[fever]),
        intl(call(feverState,[fever],[medication]),
        call(normalState,[],[nomedication])))
6-fever->
        par(call(treatment,[],[fever]),
        intl(call(feverState,[],[medication]),
        call(normalState,[normal],[nomedication])))
|: 2.
1-fever->
        par(call(treatment,[],[]),
        intl(call(feverState,[fever],[medication]),
        call(normalState,[normal],[nomedication])))
2-fever->
        par(call(treatment,[],[fever]),
        intl(call(feverState,[],[medication]),
        call(normalState,[normal],[nomedication])))
3-normal->
        par(call(treatment,[],[fever]),
        intl(call(feverState,[fever],[medication]),
        call(normalState,[],[nomedication])))
4-fever->
        par(call(treatment,[],[]),
        intl(call(feverState,[],[medication]),
        call(normalState,[normal],[nomedication])))
|: ...
```

Lorsque nous déployons le système pour montrer son exécution, nous obtenons l'arbre d'exécution qui est donné par la figure 7.16. Tout d'abord le système offre l'action location suivie du choix des actions *hospital* et *nohospital*. Suite à

l'exécution de l'action *hospital*, le système nous offre plusieurs actions *normal*, *fever*, etc. Cet arbre a été obtenu en utilisant les règles d'inférence présentées dans le chapitre 6.

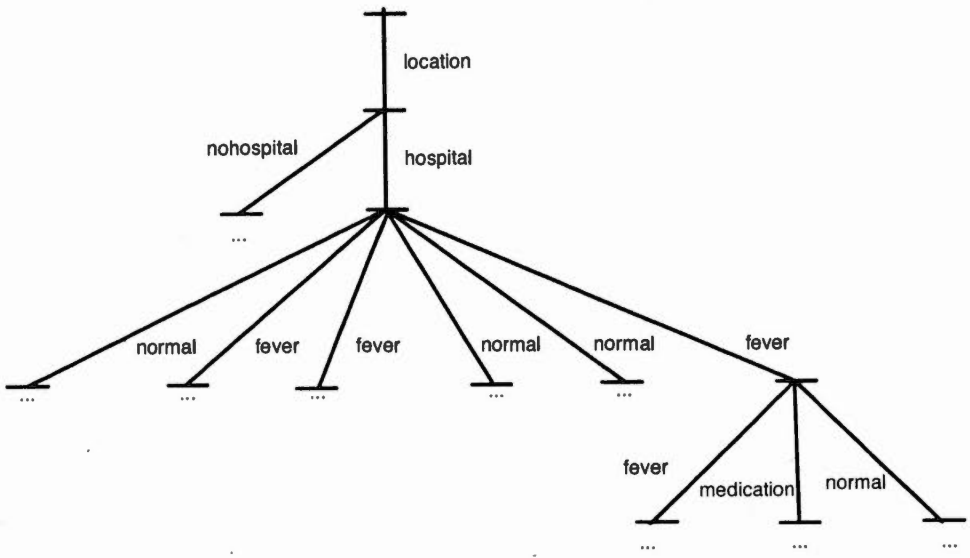


Figure 7.16 Graphe d'exécution du système *bodytemperature*

Nous donnons ci-dessous un exemple d'exécution de ce système en suivant le modèle de la boucle de contrôle décrite dans la section 7.2.

7.4.1 La base de connaissances

La base de connaissances contient les règles de génération du gabarit, sa transmission et les règles d'exécution du système (partagé avec le planificateur).

Après avoir exécuté la clause *context(bodytemperature)*, nous construisons l'ensemble de règles qui représentent le gabarit.

La base de connaissances contient également les règles qui implantent la logique des opérateurs définis dans le chapitre 6 à l'aide de la clause *infer/3*. À titre d'exemple, nous rappelons les règles qui définissent la sémantique de l'opérateur de parallélisme:

```
infer(par(P1,P2),A,par(P11,P2)):-
infer(P1,A,P11).
infer(par(P1,P2),A,par(P1,P21)):-
infer(P2,A,P21).
infer(par(P1,P2),sync(A1,A2),par(P11,P21)):-
                infer(P1,A1,P11),
                infer(P2,A2,P21),
                synch(A1,A2,A).
```

La base de connaissances représentant le gabarit est transmise au moniteur pour qu'il fasse la lecture des données du contexte. Cette base contient les éléments suivants qui seront utilisés directement par le moniteur :

```
context(bodytemperature)
svalues([35,41])
state(hypo,[30,35.5])
state(normal,[35.5,37.5])
state(fever,[37.5,42])
transform(t1,seq1,par,alt)
transform(t2,seq2,par,pipe)
transformApply(normal,t1,0,0)
transformApply(fever,t2,0,0)
applyConstraint(normal,c2)
applyConstraint(fever,c1)
constAfter(c1,seq1)
constAfter(c2,seq1)
const(c1,seq(location,seq(hospital,seq(fever,seq(s(fever),seq(medication,null))))))
const(c2,seq(location,seq(hospital,seq(normal,seq(nomedication,null)))))
seq(seq1,[location,hospital])
seq(seq2,[location,hospital])
```

7.4.2 Le moniteur

La base de connaissances communique le contenu du gabarit au moniteur pour que ce dernier fasse l'analyse des données.

Une fois que le moniteur reçoit le gabarit, il lit les données et les classe en fonction de leurs états. Pour implanter cette fonction, nous utilisons la clause *monitor/0.*

```
monitor:-
        open('data.txt', read, Stream),
        read(Stream, Value),
        context(System),
        process(System, SystemCode),
        processOneValue(Value, Stream, SystemCode),
        close(Stream).

processOneValue(end_of_file, Stream , _):- !.
processOneValue(V1,     Stream, SystemCode):-
        getInterval(V1, State, SystemCode),
        read(Stream,V2),
        processOneValue(V2, Stream, SystemCode).
```

La clause *processOneValue/3* analyse chaque valeur lue et détermine l'état auquel elle correspond ainsi que les contraintes et les transformations qui s'appliquent.

Le fichier de données se présente sous forme de valeurs des températures (en °C) lues à partir d'un capteur:

```
35.4.
37.6.
36.6.
...
...
38.3.
38.9.
39.2.
```

Le moniteur ajoute l'horodate correspondant à chaque lecture. Ce qui permettrait éventuellement de s'adapter aux traitements qui dépendent du temps. Dans l'exemple ci-dessous, nous montrons l'exécution de ce moniteur.

```
=> Value: 35.4 - 1464979307.706663 ms
State: noState
Constraint: noConstraint
Transform: noTransform
Sequence: noSeq
Execution trace: []
```

...

=> Value: 35.5 - 1464979388.889627 ms
State: noState
Constraint: noConstraint
Transform: noTransform
Sequence: noSeq
Execution trace: [location]

...

Pour lire les valeurs du contexte (à partir du fichier context.txt), nous faisons appel à la clause *getContext/0* :

```
getContext(Context):-
        gabarit(Context, Gabarit),
        analyse(Gabarit)
```

Celle-ci fait appel à une clause appelée *processValues/3* qui permet d'analyser les données lues et crée des règles dynamiquement qui représentent les éléments du contexte ainsi lus. Elle fait appel aux clauses *checkName/2*, *checkValues/2*, *checkiValues/2*, *checktTransform/2* et *checkConst/2*.

Par exemple, *checkiValues/2* crée des règles qui décrivent l'état courant du système en fonction des valeurs du contexte.

checkiValues(H,T):- createStates(H,T).

Nous avons défini trois états et leurs intervalles de valeurs associées qui sont : hypothermie pour les valeurs de température entre 30 et 35.5, normal pour les valeurs de température entre 35.5 et 37.5, et fiévreux pour les valeurs de température entre 37.5 et 42. La clause *createStates/2* permet, grâce à *assert/1*, de créer des clauses qui représentent ces états, soit:

state(hypo,[30,35.5]).

state(normal,[35.5,37.5]).
state(fever,[37.5,42]).

Nous créons également des règles qui représentent les contraintes, les transformations et les états auxquels ils sont associés.

Nous avons défini les contraintes et les séquences grâce aux clauses *const/2* et *seq/2*:

const(c1,seq(location,seq(hospital,seq(fever,seq(s(fever),seq(medication,null))))))
const(c2,seq(location,seq(hospital,seq(normal,seq(nomedication,null)))))
seq(seq1,[location,hospital])
seq(seq2,[location,hospital])

Nous avons également les règles qui représentent les transformations que l'analyseur doit utiliser. Les clauses ci-dessous décrivent une transformation appelée *t1* qui est appliquée dans l'état normal. Elle est appliquée au début du comportement du système et jusqu'à sa fin telle qu'indiquée dans les deux derniers paramètres (0 et 0):

transform(t1,seq1,par,alt)
applyTransform(normal,t1,0,0)
applyConstraint(normal,c2)
applyConstraint(fever,c1)

Dans cet exemple, la transformation consiste à remplacer l'opérateur parallèle (*par*) par l'opérateur alternatif (*alt*). Cette transformation doit s'appliquer après avoir exécuté la séquence appelée *seq1* (qui correspond à *location.hospital*) tel qu'indiqué dans la clause *seq/2* ci-dessus.

Nous avons mis en place toutes les contraintes et les transformations et leurs états associés.

7.4.3 L'analyseur

L'analyseur détermine l'état dans lequel se trouve le système et détecte les changements d'état qui ont eu lieu. Il détermine également les contraintes et les

transformations qui s'appliquent dans ce nouvel état. La clause *analyse/1* effectue ce travail.

```
analyse(Gabarit):-
        write('Le Gabarit :'), nl,
        write(Gabarit),
        analyseGabarit(Gabarit).

analyseGabarit([]).
analyseGabarit([Case:H|T]):-
        (Case == name ->          checkName(H) ;
         Case == allValues ->     checkValues(H);
         Case == iValues ->       checkIntervalValues(H);
         Case == transforms ->    checktTransforms(H);
         Case == transformApply -> checktTransformApply(H);
         Case == constraintAfter -> checkConstAfter(H);
         Case == constraintApply ->checkConstraintApply(H);
         Case == constraints ->   checkConstraints(H);
         Case == sequences ->     checkSequences(H); nl
         ),
         analyseGabarit(T).
```

La clause *analyseGabarit/1* fait appel aux clauses *checkName(H)*, *checkValues/1*, *checkIntervalValue/1*, *checktTransforms/1*, *checktTransformApply/1*, *checkConstAfter/1*, *checkConstraintApply/1*, *checkConstraints/1* et *checkSequences/1* qui permettent d'analyser les données reçues par le moniteur et de construire la base de connaissances.

L'analyseur permet aussi de mettre à jour la base de connaissances avec ces règles qui seront utilisées lors de la phase de planification.

7.4.4 Le planificateur

Le planificateur permet d'organiser les actions du système en utilisant les règles qui sont stockées dans la base de connaissances. Il effectue les tâches suivantes:

a) Exécution d'une action

Étant donné une action, il détermine le comportement du système qui découle de cette action. Par exemple, il permet de savoir le résultat de l'exécution d'une action du système :

```
go0(Action) :-
    process(bodytemperature, Process),
    infer(Process, Action, X),
    write("-> "), write(X), nl, fail.

go0(location).
->
        pipe(call(transport,[],[nohospital,hospital]),alt(par(call(treatment,[hospital],
        [normal,fever]),intl(call(feverState,[fever],[medication]),call(normalState,[no
        rmal],[nomedication]))),call(notreatment,[nohospital],[nomedication])))
```

Il peut également donner un ensemble de comportements si une même action mène à des résultats différents.

b) Exécuter une séquence d'actions

Nous pouvons soumettre une séquence d'actions et vérifier (avec la clause *trace/3*) si elle est acceptable pour un composant et si oui, obtenir le ou les résultats de l'exécution de cette séquence. Il s'agit de l'implémentation de la fonction α() décrite dans la section 6.21.

```
trace(P,[],P):-!.
trace(P,[S1|S], Res):-
  infer(P,S1,Next),
  trace(Next,S,Res).
```

L'exemple ci-dessous montre l'exécution de la séquence [location, hospital] sur *bodytemperature*:

```
go3(S):-
        process(bodytemperature,P),
```

```
        trace(P,S,Res),
        write('Sequence: '), write(S),nl,
        write(' ->'),nl, write(Res).

?-go3([location,hospital]).
Sequence: [location,hospital]
->
        par(call(treatment,[],[normal,fever]),intl(call(feverState,[fever],[medication],
        call(normalState,[normal],[nomedication])))
```

Lorsque les données indiquent que le système doit être dans un nouvel état, le planificateur détermine les étapes à suivre pour arriver à celui-ci. Il détermine cela en utilisant différents outils:

- Les séquences à appliquer en comparant la séquence à exécuter tel qu'indiqué dans le gabarit et la séquence d'actions actuellement exécutées par le système depuis le début.
- Les contraintes à appliquer à la suite de cette séquence et en fonction de l'état du système.
- Les transformations à effectuer en fonction de l'état et les séquences suite auxquelles ces transformations doivent être effectuées.

La clause *getStateConstraint/2* détermine le code de la contrainte à appliquer en fonction de l'état.

```
getStateConstraint(State,ConstraintId):-
        applyConstraint(State,ConstraintId),!.
getStateConstraint(State,noConstraint):- !
```

La clause *updateConstraint/2* permet de mettre à jour la contrainte en fonction des séquences d'actions exécutées.

```
updateConstraintCode(noConstCode,noConstCode):- !.
updateConstraintCode(ConstraintCode, NewConstraintCode):-
                actualSequence(Sequence),
```

trace(ConstraintCode, Sequence1, NewConstraintCode).

La clause *actualSequence/1* est une clause dynamique dans laquelle nous stockons la séquence d'exécution courante.

7.4.5 L'exécuteur

Une fois que les contraintes sont mises à jour, l'exécuteur les applique pour déterminer l'action courante à exécuter.

Dans notre exemple, il utilise deux contraintes qui sont *c1* et *c2* citées précédemment et qui sont associés à leurs états comme suit:

const(c1,seq(location,seq(hospital,seq(fever,seq(s(fever),seq(medication,null))))))
const(c2,seq(location,seq(hospital,seq(normal,seq(nomedication,null)))))
applyConstraint(fever,c1)
applyConstraint(normal,c2)

La contrainte *c1* est la séquence d'actions *location.hospital.fever.medication.* Ce qui correspond à la séquence d'exécution dans le cas d'un patient à l'état fiévreux qui se termine par une médication. La contrainte *c2* est la séquence d'actions *location.hospital.normal.nomedication.* Ce qui correspond à la séquence d'exécution dans le cas d'un patient à l'état normal qui ne nécessite aucune médication. Les deux contraintes obligent à passer à un hôpital après avoir été localisé.

À titre d'exemple, la contrainte *c1* impose l'exécution de la séquence *location.hospital.fever.s_fever.medication* dans l'état *fever*.

Nous utilisons également des transformations d'opérateurs en fonction de l'état courant :

transform(t1,seq1,par,alt)
transform(t2,seq2,par,pipe)
transformApply(normal,t1,0,0)
transformApply(fever,t2,0,0)
seq(seq1,[location,hospital])

seq(seq2,[location,hospital])

À titre d'exemple, la transformation *t1* s'applique dans l'état normal et elle consiste à transformer le parallélisme en alternative. Cette transformation a lieu après l'exécution de la séquence *location.hospital*.

7.4.5.1 Exécution avec contraintes et transformations

Nous avons déjà montré l'exécution du système sans utiliser ni les contraintes ni les transformations.

En utilisant les contraintes et les transformations, nous exécutons le système en fonction de son état qui est lui-même déterminé par les valeurs du contexte. La figure 7.17 montre les différentes étapes de ce processus.

1. Nous lisons les valeurs du contexte à partir d'un fichier (représentant un capteur).
2. Nous déterminons l'état courant et le changement d'état.
3. Nous exécutons le système (en déterminant les actions qu'il peut exécuter selon le modèle de sémantique des opérateurs).
4. S'il y a eu un changement d'état, nous déterminons les contraintes et les transformations et nous démarrons le système depuis de début de son comportement.
5. À chaque étape, nous calculons la séquence des actions qui ont été exécutées afin de déterminer si une contrainte doit s'exécuter après une séquence donnée (voir la règle *constAfter/2*).
6. S'il n'y a pas eu de changement d'état, nous continuons dans l'exécution du système et nous mettons à jour la séquence d'exécution. Cette séquence sera utilisée pour savoir si nous avons atteint un point pour l'application d'une transformation (voir règle *transform/4*).

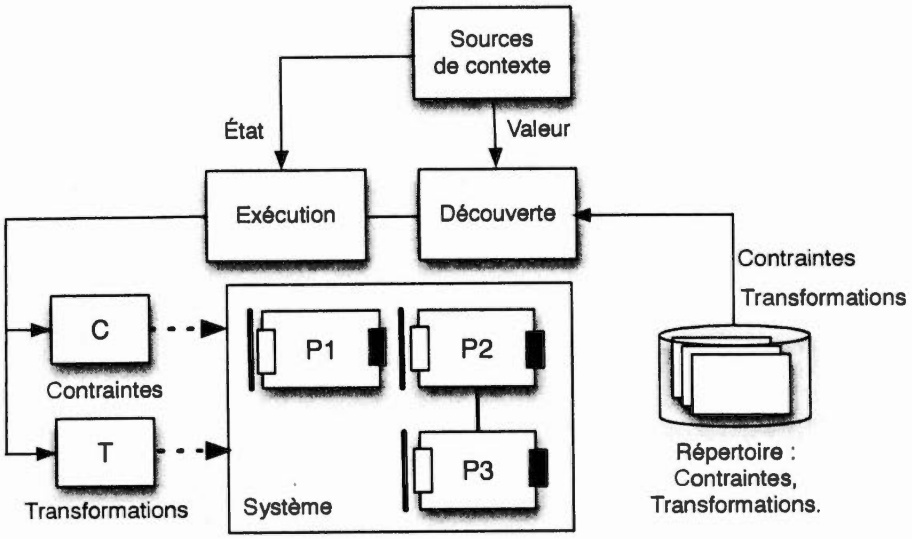


Figure 7.17 Modèle d'exécution avec contraintes et transformations

Le code Prolog qui permet cette exécution est le suivant (par manque d'espace, nous fournissons juste un extrait de ce code!).

```
getIntervalState(V,SystemCode):-
        write('\nContext value:\t  '), write(V),
        write('\tTime stamp: '), get_time(TS), write(TS),
        getState(V,State),
        write('\tState:  '), write(State),
        (newState(State)  ->
                write('\t- New State.'), getStateConstraint(State,ConstraintId),
                getConstraint(ConstraintId, ConstraintCode),
                actualSequence(ExecutionSequence),
                write("\nConstraint: \t"), write(ConstraintCode),
                write("\nExecution Sequence: \t"), write(ExecutionSequence),
                getTransform(State, Op1, Op2, Sequence),
                write("\nTransform:   \t"), write(Op1), write("/"), write(Op2),
                write("\nTransform Sequence: \t"), write(Sequence),
                actualProcess(X), emptyTrace,  process(X,ProcessCode),
                executeSystem(ProcessCode, ConstraintCode,Transform, Seq)
         not(newState(State)) ->
                write('\t- Same State.'), getStateConstraint(State,ConstraintId),
                getConstraint(ConstraintId, ConstraintCode),
                actualSequence(ExecutionSequence),
```

```
write("\nConstraint: \t"), write(ConstraintCode),
updateConstraintCode(ConstraintCode, NewConstraintCode),
write("\nUpdated Constraint: \t"), write(NewConstraintCode),
getTransform(State, Op1, Op2, Sequence),
write("\nExecution Sequence: \t"), write(ExecutionSequence),
transformCode(SystemCode, Op1, Op2, Sequence, NewSystemCode),
write("\nTransform:  \t"), write(Op1), write("/"), write(Op2),
write("\nTransform Sequence: \t"), write(Sequence),
executeSystem(NewSystemCode,NewConstraintCode,Transform,Seq)
; nl).
```

7.4.5.2 Exécution avec contrainte uniquement

Nous donnons un exemple d'application des contraintes. L'inclusion des contraintes est illsutré dans la figure 7.18.

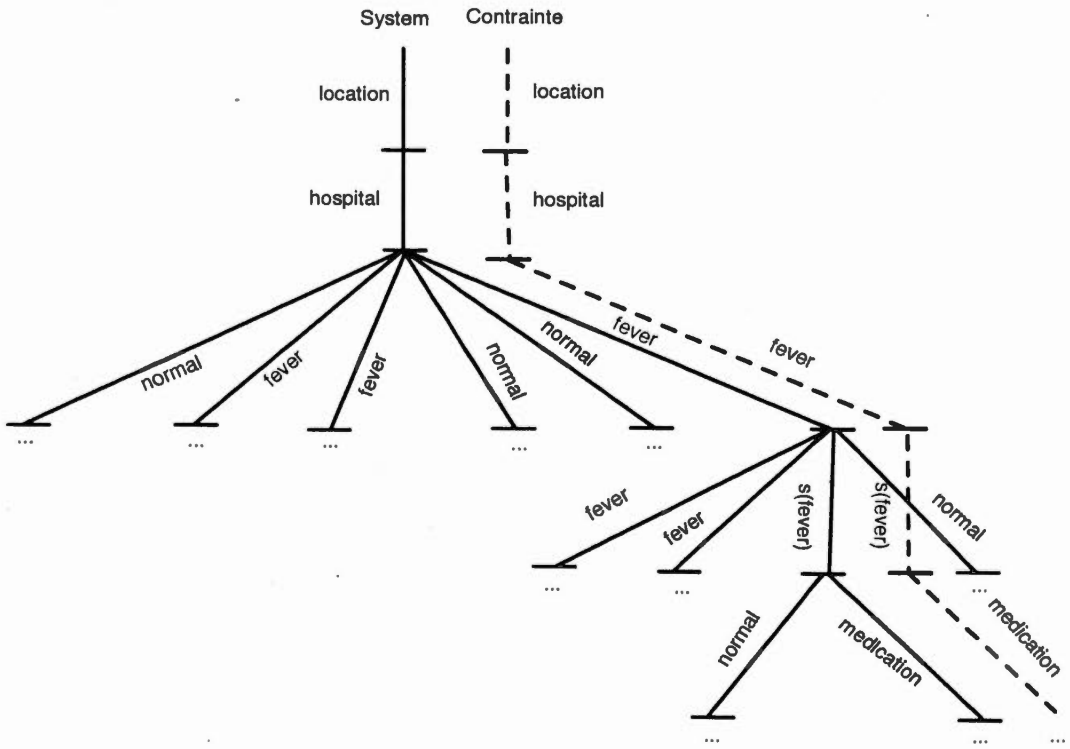


Figure 7.18 Exécution avec les contraintes

Pour exécuter le système, nous devons d'abord construire le contexte avec *getContext/1* :

```
?- getContext(bodytemperature).
La base de connaissances:
-------
context(bodytemperature)
svalues([35,41])
state(hypo,[30,35.5])
state(normal,[35.5,37.5])
state(fever,[37.5,42])
transform(t1,seq1,par,alt)
transform(t2,seq2,par,pipe)
transformApply(normal,t1,0,0)
transformApply(fever,t2,0,0)
applyConstraint(normal,c2)
applyConstraint(fever,c1)
constAfter(c1,seq1)
constAfter(c2,seq1)
const(c1,seq(location,seq(hospital,seq(fever,seq(s(fever),seq(medication,null))))))
const(c2,seq(location,seq(hospital,seq(normal,seq(nomedication,null)))))
seq(seq1,[location,hospital])
seq(seq2,[location,hospital])
true.
```

Nous appelons ensuite la clause *monitor/0*:

```
?- monitor.
Context value: 35.4 Time stamp: 1471555312.574275 State: hypo - New State.
Constraint: noConstCode
1-location->
        pipe(call(transport,[],[nohospital,hospital]),alt(par(call(treatment,[hospital],
        [normal,fever]),intl(call(feverState,[fever],[medication]),call(normalState,[no
        rmal],[nomedication]))),call(noTreatment,[nohospital],[nomedication])))
|: 1.

Context value: 37.6 Time stamp: 1471555315.553934 State: fever - New State.
Constraint: seq(location,seq(hospital,seq(fever,seq(s(fever),seq(medication,null)))))
1-location->
        const(seq(hospital,seq(fever,seq(s(fever),seq(medication,null)))),pipe(call(tra
        nsport,[],[nohospital,hospital]),alt(par(call(treatment,[hospital],[normal,feve
```

r]),intl(call(feverState,[fever],[medication]),call(normalState,[normal],[nome dication]))),call(noTreatment,[nohospital],[nomedication])))

|: 1.

*Context value: 36.6 Time stamp: 1471555317.419354 State: normal - **New State.***
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
1-location->
const(seq(hospital,seq(normal,seq(nomedication,null))),pipe(call(transport,[], [nohospital,hospital]),alt(par(call(treatment,[hospital],[normal,fever]),intl(ca ll(feverState,[fever],[medication]),call(normalState,[normal],[nomedication]))),call(noTreatment,[nohospital],[nomedication])))

|: 1.

*Context value: 35.6 Time stamp: 1471555319.063276 State: normal - **Same State.***
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Actual Sequence: [location]
Constraint Code: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Updated Constraint: seq(hospital,seq(normal,seq(nomedication,null)))
1-hospital->
const(seq(normal,seq(nomedication,null)),const(seq(normal,seq(nomedication ,null)),par(call(treatment,[],[normal,fever]),intl(call(feverState,[fever],[medic ation]),call(normalState,[normal],[nomedication]))))

|: 1.

*Context value: 35.7 Time stamp: 1471555322.140635 State: normal - **Same State.***
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Actual Sequence: [hospital,location]
Constraint Code: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Updated Constraint: seq(normal,seq(nomedication,null))
1-normal->
const(seq(nomedication,null),const(seq(nomedication,null),const(seq(nomedic ation,null),par(call(treatment,[],[fever]),intl(call(feverState,[fever],[medicati on]),call(normalState,[normal],[nomedication])))))
2-normal->
const(seq(nomedication,null),const(seq(nomedication,null),const(seq(nomedic ation,null),par(call(treatment,[],[normal,fever]),intl(call(feverState,[fever],[m edication]),call(normalState,[],[nomedication])))))

|: 2.

*Context value: 35.8 Time stamp: 1471555325.426799 State: normal - **Same State.***
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Actual Sequence: [normal,hospital,location]
Constraint Code: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))

Updated Constraint: seq(nomedication,null)
1-nomedication->
const(null,const(null,const(null,const(null,par(call(treatment,[],[normal,fever
]),intl(call(feverState,[fever],[medication]),call(normalState,[],[])))))))
|: 1.

Context value: 35.7 Time stamp: 1471555327.371858 State: normal - ***Same State.***
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Actual Sequence: [nomedication,normal,hospital,location]
Constraint Code: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
...

7.4.5.3 Exécution avec transformations

Le gabarit de l'exemple indique que des transformations doivent s'appliquer dans des états spécifiques et après avoir exécuté des séquences. Ces transformations sont:

- La transformation *t1* (qui consiste à modifier le parallélisme par l'alternative) s'applique dans l'état normal.
- La transformation *t2* (qui consiste à modifier le parallélisme par le pipeline) s'applique dans l'état *fever*.

Pour cela, nous utilisons les clauses suivantes qui sont basées sur l'opération de transformation implémentée par la clause *transformOperator/4* (opérateur [op1/op2] décrit dans le chapitre 6) :

```
getTransform(State, Op1, Op2, Sequence):-
        transformApply(State,TransformId, Start, End),
        transform(TransformId,SeqId,Op1,Op2),
        seq(SeqId,Sequence),!.
getTransform(State, none, none, none):-!.

transformCode(SystemCode, noop, Op2, Sequence, SystemCode).
transformCode(SystemCode, Op1, Op2, Sequence, NewSystemCode):-
        transformOperator(SystemCode, Op1, Op2, NewSystemCode), !.
```

En appelant le moniteur, nous obtenons l'exécution suivante, dont nous fournissons un extrait seulement:

?- monitor.

*Context value: 35.4 Time stamp: 1472203288.025767 State: hypo - **New State**.*
Constraint: noConstCode
Execution Sequence: []
Transform: none/none
Transform Sequence: none
1-location->
pipe(call(transport,[],[nohospital,hospital]),alt(par(call(treatement,[hospital],[normal,fever]),intl(call(feverState,[fever],[medication]),call(nomralState,[normal],[nomedication]))),call(noTreatment,[nohospital],[nomedication])))
|: 1.

*Context value: 37.6 Time stamp: 1472203292.896072 State: fever - **New State**.*
Constraint: seq(location,seq(hospital,seq(fever,seq(s(fever),seq(medication,null)))))
Execution Sequence: [location]
Transform: par/pipe
Transform Sequence: [location,hospital]
1-location->
const(seq(hospital,seq(fever,seq(s(fever),seq(medication,null)))),pipe(call(transport,[],[nohospital,hospital]),alt(par(call(treatement,[hospital],[normal,fever]),intl(call(feverState,[fever],[medication]),call(nomralState,[normal],[nomedication]))),call(noTreatment,[nohospital],[nomedication]))))
|: 1.

*Context value: 36.6 Time stamp: 1472203297.462028 State: normal - **New State**.*
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Execution Sequence: [location]
Transform: par/alt
Transform Sequence: [location,hospital]
1-location->
const(seq(hospital,seq(normal,seq(nomedication,null))),pipe(call(transport,[],[nohospital,hospital]),alt(par(call(treatement,[hospital],[normal,fever]),intl(call(feverState,[fever],[medication]),call(nomralState,[normal],[nomedication]))),call(noTreatment,[nohospital],[nomedication]))))
|: 1.

*Context value: 35.6 Time stamp: 1472203299.131506 State: normal - **Same State**.*
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))

Execution Sequence: [location]
Transform: par/alt
Transform Sequence: [location,hospital]
1-hospital->
const(seq(normal,seq(nomedication,null)),par(call(treatement,[],[normal,fever]),intl(call(feverState,[fever],[medication]),call(nomralState,[normal],[nomedication]))))
|: 1.

*Context value: 35.7 Time stamp: 1472203301.621152 State: normal - **Same State**.*
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Execution Sequence: [location,hospital]
Transform: par/alt
Transform Sequence: [location,hospital]
1-normal->
const(seq(nomedication,null),call(treatement,[],[fever]))
|: 1.

*Context value: 35.8 Time stamp: 1472203303.282617 State: normal - **Same State**.*
Constraint: seq(location,seq(hospital,seq(normal,seq(nomedication,null))))
Execution Sequence: [hospital,location,normal]
Transform: par/alt
Transform Sequence: [location,hospital]
...

7.5 Conclusion

Dans ce chapitre, nous avons présenté deux méthodes d'adaptation et leurs implantations : adaptation par *features* et adaptation par composition. L'adaptation par *features* se base sur le modèle de variabilité que nous avons représentée sous forme d'ontologie en établissant les liens entre les *features* et le contexte modélisé à partir de l'analyse relationnelle de concepts. Nous proposons une méthode d'adaptation comportementale qui consiste à modifier le comportement du système par l'inclusion ou l'exclusion de features selon le contexte. Quant à l'adaptation par composition, nous avons proposé un modèle sémantique spécifiant le comportement d'un système en termes de comportement de ses composants. L'architecture est exprimée à l'aide d'opérateurs. Nous avons proposé une méthode d'adaptation architecturale par l'imposition de contraintes et par la transformation d'opérateurs

que nous avons simulés. Pour chacune de ces méthodes, nous avons basé notre exécution sur la boucle de contrôle MAPE-K qui comprend 4 activités, à savoir : l'observation, l'analyse, la planification et l'exécution. Ces activités partagent la même base de connaissances. Enfin, nous avons implanté les deux méthodes en Prolog.

CHAPITRE VIII

CONCLUSION

8.1 Résumé

Le contexte suit un cycle de vie qui se retrouve dans les applications sensibles au contexte, à savoir l'acquisition de contexte, la modélisation de contexte, le raisonnement sur le contexte et la dissémination du contexte (Perera *et al.*, 2014). Nous nous sommes penchés sur deux aspects importants de ce cycle, soit : la modélisation et le raisonnement.

Nous avons présenté une approche de modélisation du contexte à partir de l'analyse relationnelle de concepts. Afin d'obtenir des contextes formels mono-valués, nous avons proposé la méthode *k-means* pour les données numériques et nous avons utilisé la méthode d'échelonnage conceptuel de Wille (Ganter et Wille, 1989) pour les données catégoriques. Notre approche permet d'établir des liens entre les services d'un système sensible au contexte et leurs informations contextuelles. Nous avons utilisé l'outil RCAExplore (Dolques, 2014) pour illustrer notre méthode à partir d'un exemple, à savoir un système de surveillance de l'état de santé de patients.

Nous avons proposé d'utiliser la logique descriptive pour raisonner sur le contexte. Cette étape consiste à valider le modèle de contexte et à effectuer des opérations de raisonnement sur le contexte. Ce choix s'est basé sur notre objectif de proposer une méthode de raisonnement qui ne dépende pas de la méthode de modélisation, mais plutôt sur une approche qui l'accompagne. À partir des règles de correspondance entre les entités de l'analyse relationnelle de concepts et la logique descriptive, nous

avons montré le raisonnement sur le contexte, à savoir l'instanciation de concepts et la subsomption, la comparaison de concepts et l'analyse du co-domaine d'une relation. En d'autres termes, le raisonnement sur le contexte des applications sensibles au contexte permet de déterminer le concept formel d'un service, les contextes dont dépend un service particulier, les services appartenant au même concept formel et le concept formel d'un service selon ses attributs. Nous avons utilisé l'éditeur d'ontologie Protégé ("Protégé Ontology Editor," 2015) et le raisonneur Pellet ("Pellet," 2015).

Vu les limites de la logique descriptive, nous avons décidé d'utiliser la logique à base de règles pour construire des règles de contexte qui associent une situation contextuelle à des configurations d'une application sensible au contexte. Pour ce faire, nous avons décidé d'utiliser les modèles de variabilité pour représenter les fonctionnalités de ce type d'application. Par conséquent, nous avons modélisé un modèle de variabilité notre système de surveillance de l'état de santé de patients en ontologie dans le but de valider la cohérence du modèle et de ses configurations. Nous avons implanté un programme en Java et Groovy qui permet, à partir d'un fichier OWL, d'ajouter de nouvelles configurations ou de modifier des configurations existantes. Ces configurations sont validées par le raisonneur Pellet. Nous avons construit des règles de contexte sous format SWRL et elles sont également validées par le même raisonneur.

Un aspect essentiel des applications sensibles au contexte est l'adaptation. Le concept d'adaptation a pour objectif de permettre à ces systèmes de s'adapter, c'est-à-dire de modifier leur comportement et/ou leur architecture suite au changement des valeurs du contexte de l'utilisateur et/ou de son environnement. Nous avons proposé deux méthodes d'adaptation, soit : l'adaptation par composition et l'adaptation par *features*.

La première méthode d'adaptation repose sur un modèle sémantique qui permet de spécifier l'architecture d'un système en termes de composition des composants grâce à des opérateurs de transformation d'architectures. De plus, nous avons introduit le concept de contraintes qui est utilisé pour restreindre le comportement global du système à un sous-comportement. Ces deux mécanismes permettent d'effectuer des changements d'architecture forçant ainsi un système à s'adapter aux changements des valeurs du contexte.

La seconde méthode d'adaptation repose sur un modèle sémantique de la variabilité qui consiste à préciser la composition d'un système en choisissant une configuration particulière de ses *features*. Cette méthode consiste à activer et/ou désactiver des fonctionnalités selon certaines conditions appliquées sur les valeurs de contexte, et ce, en respectant les types de *features* ainsi que leurs décompositions dans le modèle de variabilité.

Les différentes activités d'une boucle de contrôle fournissent le mécanisme général pour l'auto-adaptation de ces systèmes. Parmi les nombreuses boucles de contrôle proposées dans la littérature, nous avons utilisé le modèle de référence MAPE-K (IBM, 2006; Jacob *et al.*, 2004) dont les activités impliquées, c'est-à-dire la surveillance, l'analyse, la planification et l'exécution, partagent des informations avec une base de connaissances décrite en Prolog. Nous avons implanté cette boucle de contrôle en Prolog pour les deux méthodes d'adaptation, à savoir l'adaptation par composition et par *features* pour notre système simple de surveillance de l'état de santé de patients.

8.2 Contributions

Dans le cadre de cette thèse, nous amenons les contributions suivantes :

a) *Modélisation du contexte* :

Nous avons proposé une méthode de modélisation de contexte qui permet d'établir des liens et des dépendances entre les contextes et les services. Notre méthode de modélisation décrit formellement les contextes et les services, et elle supporte le raisonnement sur le contexte.

b) *Raisonnement sur le contexte* :

Nous avons effectué un raisonnement formel sur le contexte en utilisant la logique descriptive et la logique à base de règles. D'une part, nous avons validé formellement le modèle de contexte et nous avons aussi introduit des opérations de raisonnement sur le contexte à partir de la logique descriptive.

D'autre part, nous avons utilisé la logique à base de règles pour définir des situations contextuelles qui représentent des états de système nécessitant d'être activés selon le contexte de l'utilisateur. Ainsi, nous avons représenté un modèle de *features*, c'est-à-dire les fonctionnalités contextuelles d'un système, sous forme d'ontologie OWL afin de le décrire sémantiquement, et nous l'avons validé en considérant les contraintes de *features* et les dépendances entre les *features*-parents et les *features*-enfants à partir d'un raisonneur. De plus, nous avons établi un lien entre un contexte modélisé à partir de l'analyse relationnelle de concepts et un modèle de *features* représenté par une ontologie OWL. En utilisant un langage de règles pour le Web sémantique, nous avons défini des règles de contexte qui associent des situations contextuelles provenant du modèle de contexte à des configurations valides du modèle de *features*.

c) *Adaptation d'une application dépendante du contexte* :

Nous avons proposé une méthode d'adaptation architecturale basée sur un modèle sémantique de composant et une méthode d'adaptation comportementale basée sur un modèle sémantique de représentation des fonctionnalités contextuelles, respectivement.

8.3 Directions futures de recherche

Le travail effectué dans la présente thèse ouvre certaines pistes de recherche, notamment :

a) *Acquisition et prétraitement des données contextuelles* :

L'acquisition de données contextuelles est une étape importante qui précède la modélisation du contexte. Elle consiste à effectuer un prétraitement des données acquises, à savoir : le nettoyage, le filtrage et l'agrégation des données contextuelles. Les techniques d'acquisition du contexte varient selon la responsabilité, la fréquence, la source et le processus d'acquisition (Perera et al., 2014).

Toutefois, l'acquisition et le prétraitement de contexte ne sont pas des étapes à négliger et elles requièrent des algorithmes qui prennent en considération les divers types de contextes. L'objectif est de fournir à l'étape de modélisation du contexte des données contextuelles qui peuvent être représentées.

b) *Modélisation du contexte* :

La modélisation du contexte est une étape importante dans le cadre des applications sensibles au contexte. Dans cette thèse, nous avons proposé une nouvelle méthode de modélisation du contexte et nous l'avons validée. Cependant, la modélisation du contexte ne s'arrête pas ni sur sa représentation ni sur sa validation. Elle nécessite d'effectuer des techniques de mise à jour du modèle existant en considérant certains aspects tels que : la fusion (*clustering*) de données contextuelles similaires, la gestion d'informations contextuelles imprécises ou incohérentes, l'ajout ou la suppression de nouvelles informations contextuelles, la validité et la pertinence de l'information contextuelle, et les transitions d'état du contexte.

c) *Modèle de variabilité* :

Dans ce travail, nous avons utilisé un modèle de *features* pour représenter les fonctionnalités contextuelles d'une application sensible au contexte. Nous l'avons validé et relié au modèle de contexte. Vu le manque de temps, nous n'avons pas tenu compte des ajouts ou des modifications qui peuvent être apportées au modèle de *features*. Il serait intéressant de proposer une solution sur cet aspect du modèle durant l'exécution et l'auto-adaptation d'une application sensible au contexte.

d) *Interrogation des fichiers d'ontologie OWL à partir de Prolog :*

Nous avons utilisé la logique de description ou l'équivalence par une ontologie OWL pour valider le modèle de contexte et raisonner sur le contexte. Nous avons également utilisé cette logique pour représenter le modèle de *features* et le valider. Par manque de temps, nous n'avons pas pu utiliser cette logique pour effectuer l'adaptation par la boucle de contrôle MAPE-K et nous n'avons pas pu explorer des techniques qui permettraient d'interroger ou de transformer l'ontologie en logique de premier ordre, en l'occurrence Prolog. Une telle solution amènerait une consolidation de la modélisation du contexte et de son raisonnement avec l'adaptation du système selon le contexte.

e) Raisonnement sur les informations contextuelles imprécises :

Dans cette thèse, nous avons utilisé deux types de logique pour raisonner sur le contexte, soit la logique descriptive et la logique à base de règles. Toutefois, ces logiques ne permettent pas de gérer des informations contextuelles imprécises ou ambiguës. Les applications dépendantes du contexte sont notamment utilisées dans des environnements mobiles et ubiquitaires. Ainsi, il serait pertinent d'étendre le modèle proposé pour raisonner sur le contexte en tenant compte de ces types d'informations contextuelles. Une piste intéressante à explorer serait celle de la

logique probabiliste puisqu'elle permet de gérer des situations inconnues ou imprécises par un raisonnement numérique.

f) *Modèle d'adaptation mixte :*

L'adaptation est une partie importante des applications sensibles au contexte. Nous avons montré une méthode d'adaptation comportementale et une méthode d'adaptation architecturale. Nos méthodes sont indépendantes l'une de l'autre. Idéalement, l'adaptation devrait se faire aux deux niveaux. Ainsi, un composant devrait grouper un ou plusieurs *features* représentant. L'invocation des composants en question devrait uniquement activer les *features* requis tout en fournissant une configuration valide du modèle de *features* en temps réel. Vu le manque de temps, nous n'avons pas pu trouver une solution à cette problématique.

ANNEXE A

DÉTAILS DES TREILLIS DE CONCEPTS

L'annexe A est un complément du chapitre 3 qui présente en détail les concepts formels obtenus à partir de l'analyse relationnelle de concepts.

A.1 Treillis de concepts à l'état initial

Les concepts formels du treillis de concepts pour le contexte de *Localisation* sont les suivants :

a) Concept_ContexteLocalisation_0 (CL0) :
   - *Intension du concept* : ValeurLocalisationCluster1 et ValeurLocalisationCluster2.
   - *Extension du concept* : Non applicable.
   - *Définition du concept :* Les localisations ayant toutes les propriétés.

b) Concept_ContexteLocalisation_1 (CL1) :
   - *Intension du concept* : ValeurLocalisationCluster1.
   - *Extension du concept* : LocalisationUtilisateur1.
   - *Définition du concept :* Les valeurs de localisation appartenant au cluster 1.

c) Concept_ContexteLocalisation_2 (CL2) :
   - *Intension du concept* : ValeurLocalisationCluster2
   - *Extension du concept* : LocalisationUtilisateur2 et LocalisationUtilisateur3.
   - *Définition du concept :* Les valeurs de localisation appartenant au cluster 2.

d) Concept_ContexteLocalisation_3 (CL3) :

- *Intension du concept* : Non applicable.
- *Extension du concept* : LocalisationUtilisateur1, LocalisationUtilisateur2 et LocalisationUtilisateur3.
- *Définition du concept :* Localisation.

Les concepts formels du treillis de concepts pour le contexte de *Température Corporelle* sont les suivants :

a) Concept_ContexteTemperatureCorporelle_0 (CT0) :
   - *Intension du concept* : {PrecisionElevee, PrecisionMoyenne, PrecisionFaible, ValeurTemperatureCluster1, ValeurTemperatureCluster2, ValeurTemperatureCluster3, Capteur1, Capteur2}
   - *Extension du concept* : {}.
   - *Définition du concept* : Les températures corporelles ayant toutes les propriétés.

b) Concept_ContexteTemperatureCorporelle_1 (CT1) :
   - *Intension du concept* : {ValeurTemperatureCluster2, PrecisionMoyenne, Capteur2}
   - *Extension du concept* : {TemperatureCorporelle3}
   - *Définition du concept* : Les valeurs de température corporelle ayant une précision moyenne, appartenant au cluster 2 et ayant été captée par le capteur source 2.

c) Concept_ContexteTemperatureCorporelle_2 (CT2) :
   - *Intension du concept* : {ValeurTemperatureCluster3, PrecisionElevee, Capteur1}
   - *Extension du concept* : {TemperatureCorporelle2}
   - *Définition du concept* : Les valeurs de température corporelle ayant une précision élevée, appartenant au cluster 3 et ayant été captés par le capteur source 1.

d) Concept_ContexteTemperatureCorporelle_3 (CT3) :
   - *Intension du concept* : {ValeurTemperatureCluster1, PrecisionFaible, Capteur1}
   - *Extension du concept* : {TemperatureCorporelle1}
   - *Définition du concept* : Les valeurs de température ayant une précision faible, appartenant au cluster 1 et ayant été captée par le capteur source 1.

e) Concept_ContexteTemperatureCorporelle_4 (CT4) :
   - *Intension du concept* : {Capteur1}
   - *Extension du concept* : {TemperatureCorporelle1, TemperatureCorporelle2}
   - *Définition du concept* : Les valeurs de température corporelle ayant été captée par le capteur source 1.

f) Concept_ContexteTemperatureCorporelle_5 (CT5) :
   - *Intension du concept* : {}
   - *Extension du concept* : {TemperatureCorporelle1, TemperatureCorporelle2, TemperatureCorporelle3}
   - *Définition du concept* : Température corporelle.

Les concepts formels du treillis de concepts pour le contexte de *Pression Artérielle* sont les suivants :

a) Concept_ContextePressionArterielle_0 (CT0)
   - *Intension du concept* : {PrecisionElevee, PrecisionMoyenne, PrecisionFaible, ValeurPressionArterielleCluster1, ValeurPressionArterielleCluster2, ValeurPressionArterielleCluster3, Capteur1, Capteur2}
   - *Extension du concept* :{}
   - *Définition du concept* : Les pressions artérielles ayant toutes les propriétés.

b) Concept_ContextePressionArterielle_1 (CT1)
   - *Intension du concept* : {PrecisionElevee, ValeurPressionArterielleCluster3, Capteur2}

- *Extension du concept* : {PressionArterielle3}
- *Définition du concept* : Les valeurs de pression artérielle ayant une précision élevée, appartenant au cluster 3 et ayant été captés par le capteur source 2.

c) Concept_ContextePressionArterielle_2 (CT2)

- *Intension du concept* : {ValeurPressionArterielleCluster2, PrecisionMoyenne, Capteur2}.
- *Extension du concept* : {PressionArterielle2}
- *Définition du concept* : Les valeurs de pression artérielle ayant une précision moyenne, appartenant au cluster 2 et ayant été captée par le capteur source 2.

d) Concept_ContextePressionArterielle_3 (CT3)

- *Intension du concept* : {ValeurTemperatureCluster1, PrecisionFaible, Capteur1}
- *Extension du concept* : {PressionArterielle1}
- *Définition du concept* : Les valeurs de pression artérielle ayant une précision faible, appartenant au cluster 1 et ayant été captée par le capteur source 1.

e) Concept_ContextePressionArterielle_4 (CT4)

- *Intension du concept* : {Capteur2}.
- *Extension du concept* : {PressionArterielle2, PressionArterielle3}
- *Définition du concept* : Les valeurs de pression artérielle ayant été captée par le capteur source 2.

f) Concept_ContextePressionArterielle_5 (CT5)

- *Intension du concept* : {}
- *Extension du concept* : {PressionArterielle1, PressionArterielle2, PressionArterielle3}
- *Définition du concept* : Pression artérielle.

Les concepts formels du treillis de concepts des services sont les suivants :

a) Concept_Services_0 (CS0) :
   - *Intension du concept* : {F1Communication, F11Wifi, F12Zigbee, F2Authentification, F3GeoLocalisation, F31GPS, F32Wifi, F4NotificationPatient, F41PrioriteAlerte, F411Fifo, F412Priorite, F42Alertes, F421AvertisseurEcran, F422AvertisseurSonore, F5Alarme, F51DetectionChute, F52BoutonPanique, F6Capteur, F61CapteurPhysiologique, F611PressionArterielle, F612TempCorporelle, F62CapteurEnvironnement, F621TempAmbiante, F622Lumiere, F7PlanSoins}
   - *Extension du concept* : {}
   - *Définition du concept* : Les services ayant toutes les propriétés.
b) Concept_Services_1 (CS1) :
   - *Intension du concept* : {F622Lumiere, F621TempAmbiante}
   - *Extension du concept* : {ServiceEnvironnement}
   - *Définition du concept* : Les services offrant les fonctionnalités F622Lumières et F621TempAmbiante.
c) Concept_Services_2 (CS2) :
   - *Intension du concept* : {F611PressionArterielle, F612TempCorporelle}
   - *Extension du concept* : {ServiceSignesVitaux}
   - *Définition du concept* : Le service offrant les fonctionnalités F611PressionArterielle et F612TempCorporelle.
d) Concept_Services_3 (CS3) :
   - *Intension du concept* : {F51DetectionChute, F52BoutonPanique}
   - *Extension du concept* : {ServiceOptionAlarme}
   - *Définition du concept* : Le service offrant les fonctionnalités F51DetectionChute et F52BoutonPanique.

e) Concept_Services_4 (CS4) :

- *Intension du concept* : {F421AvertisseurEcran, F422AvertisseurSonore}
- *Extension du concept* : {ServiceAvertisseur}
- *Définition du concept* : Le service offrant les fonctionnalités F421AvertisseurEcran et F422AvertisseurSonore.

f) Concept_Services_5 (CS5) :

- *Intension du concept* : {F412Priorite, F411Fifo}
- *Extension du concept* : {ServiceOptionPrioriteAlerte}
- *Définition du concept* : Le service offrant les fonctionnalités F412Priorite et F411Fifo.

g) Concept_Services_6 (CS6) :

- *Intension du concept* : {F7PlanSoins}
- *Extension du concept* : {ServiceSoins}
- *Définition du concept* : Le service offrant la fonctionnalité F7PlanSoins.

h) Concept_Services_7 (CS7) :

- *Intension du concept* : {F6Capteur, F61CapteurPhysiologique, F62CapteurEnvironnement}
- *Extension du concept* : {ServiceCapteur}
- *Définition du concept* : Le service offrant les fonctionnalités F6Capteur, F61CapteurPhysiologique et F62CapteurEnvironnement.

i) Concept_Services_8 (CS8) :

- *Intension du concept* : {F5Alarme}
- *Extension du concept* : {ServiceAlarme}
- *Définition du concept* : Le service offrant la fonctionnalité F5Alarme.

j) Concept_Services_9 (CS9) :

- *Intension du concept* : {F41PrioriteAlerte, F42Alertes, F4NotificationPatient}
- *Extension du concept* : {ServiceNotificationPatient}

- *Définition du concept* : Le service offrant les fonctionnalités F41PrioriteAlerte, F42Alertes et F4NotificationPatient.

k) Concept_Services_10 (CS10) :

- *Intension du concept* : {F3GeoLocalisation, F31GPS, F32Wifi}
- *Extension du concept* : {ServiceGeoLocalisation}
- *Définition du concept* : Le service offrant les fonctionnalités F3GeoLocalisation, F31GPS et F32Wifi.

l) Concept_Services_11 (CS11) :

- *Intension du concept* : {F2Authentification}
- *Extension du concept* : {ServiceAuthentification}
- *Définition du concept* : Le service offrant la fonctionnalité F2Authentification.

m) Concept_Services_12 (CS12) :

- *Intension du concept* : {F1Communication, F12Zigbee, F11Wifi}
- *Extension du concept* : {ServiceCommunication}
- *Définition du concept* : Le service offrant les fonctionnalités F12Zigbee, F11Wifi et F1Communication.

n) Concept_Services_13 (CS13) :

- *Intension du concept* : {}
- *Extension du concept* : {ServiceCommunication, ServiceAuthentification, ServiceGeoLocalisation, ServiceNotificationPatient, ServiceAlarme, ServiceCapteur, ServiceSoins, ServiceOptionPrioriteAlerte, ServiceAvertisseur, ServiceOptionAlarme, ServiceSignesVitaux, ServiceEnvironnement}
- *Définition du concept* : Services.

A.2 Treillis de concepts après l'échelonnage relationnel

Les concepts formels du treillis de concepts des services sont les suivants :

a) Concept_Service_0 (CS0) :

- *Intension du concept* : {F5Alarme, F611PressionArterielle, F52BoutonPanique, dependLocalisation(CL1), dependLocalisation(CL2), dependLocalisation(CL3), F422AvertisseurSonore, F61CapteurPhysiologique, F6Capteur, dependLocalisation(CL0), F62CapteurEnvironnement, F4NotificationPatient, F7PlanSoins, F622Lumiere, F3GeoLocalisation, F421AvertisseurEcran, F42Alertes, dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP0), F12Zigbee, dependPressionArterielle(CP3), dependPressionArterielle(CP1), F412Priorite, dependPressionArterielle(CP2), F41PrioriteAlerte, F2Authentification, F11Wifi, F1Communication, dependTemperature(CT0), F51DetectionChute, dependTemperature(CT3), dependTemperature(CT2), F621TempAmbiante, dependTemperature(CT4), dependTemperature(CT1), F411Fifo, F612TempCorporelle, F31GPS, F32Wifi, dependTemperature(CT5)}
- *Extension du concept* : {}
- *Définition du concept* : Les services offrant toutes les fonctionnalités.

b) Concept_Service_2 (CS2) :

- *Intension du concept* : {dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP3), F611PressionArterielle, dependPressionArterielle(CP1), dependTemperature(CT3), dependTemperature(CT2), dependTemperature(CT4), dependTemperature(CT1), dependPressionArterielle(CP2), F612TempCorporelle, dependTemperature(CT5)}
- *Extension du concept* : ServiceSignesVitaux.
- *Définition du concept* : Le service offrant les fonctionnalités F611PressionArterielle et F612TempCorporelle. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle

appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

c) Concept_Service_3 (CS3) :

- *Intension du concept* : {dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP3), F52BoutonPanique, dependPressionArterielle(CP1), dependLocalisation(CL1), dependLocalisation(CL2), dependLocalisation(CL3), dependPressionArterielle(CP2), F51DetectionChute, dependTemperature(CT3), dependTemperature(CT2), dependTemperature(CT4), dependTemperature(CT1), dependTemperature(CT5)}
- *Extension du concept* : {ServiceAlarme}
- *Définition du concept* : Le service offrant les fonctionnalités F52BoutonPanique et F51DetectionChute. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le

capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

d) Concept_Service_6 (CS6) :

- *Intension du concept* : {dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP3), dependPressionArterielle(CP1), dependLocalisation(CL1), dependLocalisation(CL2), dependLocalisation(CL3), dependPressionArterielle(CP2), F7PlanSoins, dependTemperature(CT3), dependTemperature(CT2), dependTemperature(CT4), dependTemperature(CT1), dependTemperature(CT5)}
- *Extension du concept* : {ServiceSoins}
- *Définition du concept* : Le service offrant la fonctionnalité de F7PlanSoins. Ce service dépend de la pression artérielle selon 4 scénarios: 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend

également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

e) Concept_Service_7 (CS7) :

- *Intension du concept* : {dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP3), dependPressionArterielle(CP1), dependLocalisation(CL1), dependLocalisation(CL2), dependLocalisation(CL3), F61CapteurPhysiologique, dependPressionArterielle(CP2), F6Capteur, F62CapteurEnvironnement, F51DetectionChute, dependTemperature(CT3), dependTemperature(CT2), dependTemperature(CT4), dependTemperature(CT1), dependTemperature(CT5)}
- *Extension du concept* : {ServiceCapteur}
- *Définition du concept* : Le service offrant les fonctionnalités F6Capteurm F62CapteurEnvironnement et F51DetectionChute. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle

appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

f) Concept_Service_8 (CS8) :

- *Intension du concept* : {dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP3), F5Alarme, dependPressionArterielle(CP1), dependLocalisation(CL1), dependLocalisation(CL2), dependLocalisation(CL3), F61CapteurPhysiologique, dependPressionArterielle(CP2), dependTemperature(CT3), dependTemperature(CT2), dependTemperature(CT4), dependTemperature(CT1), dependTemperature(CT5)}
- *Extension du concept* : {ServiceAlarme}
- *Définition du concept* : Le service offrant la fonctionnalité F5Alarme. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la

température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

g) Concept_Service_10 (CS10) :

- *Intension du concept* : {dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP3), dependPressionArterielle(CP1), dependLocalisation(CL1), dependLocalisation(CL2), dependLocalisation(CL3), dependPressionArterielle(CP2), dependTemperature(CT3), dependTemperature(CT2), F3GeoLocalisation, dependTemperature(CT4), dependTemperature(CT1), F31GPS, F32Wifi, dependTemperature(CT5)}
- *Extension du concept* : {ServiceGeoLocalisation}
- *Définition du concept* : Le service offrant les fonctionnalités F3GeoLocalisation, F31GPS et F32Wifi. Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

h) Concept_Service_14 (CS14) :

- *Intension du concept* : {dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP3), dependPressionArterielle(CP1), dependLocalisation(CL1), dependLocalisation(CL2), dependLocalisation(CL3), dependPressionArterielle(CP2), dependTemperature(CT3), dependTemperature(CT2), dependTemperature(CT4), dependTemperature(CT1), dependTemperature(CT5)}
- *Extension du concept* : {ServiceGeoLocalisation, ServiceAlarme, ServiceCapteur, ServiceSoins, ServiceOptionAlarme}
- *Définition du concept* : Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend de la localisation de l'utilisateur. La valeur de la localisation appartient soit au cluster1, soit au cluster2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

i) Concept_Service_15 (CS15) :

- *Intension du concept* : {dependPressionArterielle(CP4), dependPressionArterielle(CP5), dependPressionArterielle(CP3),

- dependPressionArterielle(CP1), dependTemperature(CT3), dependTemperature(CT2), dependTemperature(CT4), dependTemperature(CT1), dependPressionArterielle(CP2), dependTemperature(CT5)}
- *Extension du concept* : {ServiceGeoLocalisation, ServiceAlarme, ServiceCapteur, ServiceSoins, ServiceOptionAlarme, ServiceSignesVitaux}
- *Définition du concept* : Ce service dépend de la pression artérielle selon 4 scénarios : 1) La valeur de la pression artérielle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la pression artérielle est captée par le capteur2. 3) La valeur de la pression artérielle appartient au cluster3, a une précision élevée et est captée par le capteur2. 4) La valeur de la pression artérielle appartient au cluster2, a une précision moyenne et est captée par le capteur2. Ce service dépend également de la température corporelle selon 4 scénarios : 1) La valeur de la température corporelle appartient au cluster1, a une précision faible et est capté par le capteur1. 2) La valeur de la température corporelle appartient au cluster3, a une précision élevée et est captée par le capteur1. 3) La valeur de la température corporelle appartient au cluster2, a une précision moyenne et est captée par le capteur2. 4) La valeur de la température corporelle est captée par le capteur1.

ANNEXE B

DÉTAILS DE LA BASE DE CONNAISSANCES

L'annexe B est un complément du chapitre 4 qui présente en détail la base de connaissances obtenu à partir de la logique descriptive.

B.1 TBox de la base de connaissances

Le tableau B.1 présente tous les concepts primitifs et les rôles de la base de connaissances.

Tableau B.1 TBox : Concepts primitifs et rôles

Concepts primitifs et rôles
ValeurLocalisationCluster1, ValeurLocalisationCluster2, TempPrecisionFaible, TempPrecisionMoyenne, TempPrecisionElevee, ValeurTemperatureCluster1, ValeurTemperatureCluster2, TempCapteur1, TempCapteur2, PressionPrecisionFaible, PressionPrecisionMoyenne, PressionPrecisionElevee, ValeurPressionArterielleCluster1, ValeurPressionArterielleCluster2, PressionCapteur1, PressionCapteur2, F1Communication, F11Wifi, F12Zigbee, F2Authentification, F3GeoLocalisation, F31GPS, F32Wifi, F4NotificationPatient, F41PrioriteAlerte, F411Fifo, F412Priorite, F42Alertes, F421AvertisseurEcran, F422AvertisseurSonore, F5Alarme, F51DetectionChute, F52BoutonPanique, F6Capteur, F61CapteurPhysiologique, F611PressionArterielle, F612TempCorporelle, F62CapteurEnvironnement, F621TempAmbiante, F622Lumiere, F7PlanSoins, dependLocalisation, dependTempCorporelle, dependPressionArterielle

Le tableau B.2 présente tous les concepts définis de la base de connaissances.

Tableau B.2 TBox : Concepts définis

ID du treillis	Nom du concept	Concept défini
Concept_Services_0 (CS0)	CS0	$\bot$
Concept_Services_1 (CS1)	CS1	$\exists F622Lumiere.\top \sqcap \exists F621TempAmbiante.\top$
Concept_Services_2 (CS2)	CS2	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap \exists F611PressionArterielle.\top \sqcap dependPressionArterielle.CP1 \sqcap \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap dependPressionArterielle.CP2 \sqcap \exists F612TempCorporelle.\top \sqcap \exists dependTemperatureCorporelle.CT5$
Concept_Services_3 (CS3)	CS3	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap \exists F52BoutonPanique.\top \sqcap dependPressionArterielle.CP1 \sqcap dependLocalisation.CL1 \sqcap dependLocalisation.CL2 \sqcap dependLocalisation.CL3 \sqcap dependPressionArterielle.CP2 \sqcap \exists F51DetectionChute.\top \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap \exists dependTemperatureCorporelle.CT5$
Concept_Services_4 (CS4)	CS4	$\exists F421AvertisseurEcran.\top \sqcap \exists F422AvertisseurSonore.\top$
Concept_Services_5 (CS5)	CS5	$\exists F412Priorite.\top \sqcap \exists F411Fifo.\top$
Concept_Services_6 (CS6)	CS6	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap dependPressionArterielle.CP1 \sqcap dependLocalisation.CL1 \sqcap dependLocalisation.CL2 \sqcap dependLocalisation.CL3 \sqcap dependPressionArterielle.CP2 \sqcap \exists F7PlanSoins.\top \sqcap dependTemperatureCorporelle.CT3 \sqcap dependTemperatureCorporelle.CT2 \sqcap dependTemperatureCorporelle.CT4 \sqcap dependTemperatureCorporelle.CT1 \sqcap dependTemperatureCorporelle.CT5$

Concept_Services_7 (CS7)	CS7	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap dependPressionArterielle.CP1 \sqcap dependLocalisation.CL1 \sqcap dependLocalisation.CL2 \sqcap dependLocalisation.CL3 \sqcap \exists F61CapteurPhysiologique.\top \sqcap dependPressionArterielle.CP2 \sqcap \exists F61Capteur.\top \sqcap \exists F62CapteurEnvironnement \sqcap \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap \exists dependTemperatureCorporelle.CT5$
Concept_Services_8 (CS8)	CS8	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap \exists F5Alarme.\top \sqcap dependPressionArterielle.CP1 \sqcap dependLocalisation.CL1 \sqcap dependLocalisation.CL2 \sqcap dependLocalisation.CL3 \sqcap dependPressionArterielle.CP2 \sqcap \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap \exists dependTemperatureCorporelle.CT5$
Concept_Services_9 (CS9)	CS9	$\exists F41PrioriteAlerte.\top \sqcap \exists F42Alertes.\top \sqcap \exists F4NotificationPatient.\top$
Concept_Services_10 (CS10)	CS10	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap dependPressionArterielle.CP1 \sqcap dependLocalisation.CL1 \sqcap dependLocalisation.CL2 \sqcap dependLocalisation.CL3 \sqcap dependPressionArterielle.CP2 \sqcap dependTemperatureCorporelle.CT3 \sqcap dependTemperatureCorporelle.CT2 \sqcap \exists F3GeoLocalisation.\top \sqcap dependTemperatureCorporelle.CT4 \sqcap dependTemperatureCorporelle.CT1 \sqcap \exists F31GPS.\top \sqcap \exists F32Wifi.\top \sqcap dependTemperatureCorporelle.CT5$
Concept_Services_11 (CS11)	CS11	$\exists F2Authentification.$
Concept_Services_12 (CS12)	CS12	$\exists F12Zigbee.\top \sqcap \exists F11Wifi.\top \sqcap \exists F1Communication.\top$
Concept_Services_13 (CS13)	CS13	*Service*

Concept_Services_14 (CS14)	CS14	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap dependPressionArterielle.CP1 \sqcap dependLocalisation.CL1 \sqcap dependLocalisation.CL2 \sqcap dependLocalisation.CL3 \sqcap dependPressionArterielle.CP2 \sqcap \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap \exists dependTemperatureCorporelle.CT5$
Concept_Services_15 (CS15)	CS15	$dependPressionArterielle.CP4 \sqcap dependPressionArterielle.CP5 \sqcap dependPressionArterielle.CP3 \sqcap dependPressionArterielle.CP1 \sqcap \exists dependTemperatureCorporelle.CT3 \sqcap \exists dependTemperatureCorporelle.CT2 \sqcap \exists dependTemperatureCorporelle.CT4 \sqcap \exists dependTemperatureCorporelle.CT1 \sqcap dependPressionArterielle.CP3 \sqcap \exists dependTemperatureCorporelle.CT5$
Concept_ContexteLocalisation_0 (CL0)	CL0	$\bot$
Concept_ContexteLocalisation_1 (CL1)	CL1	$\exists ValeurTemperatureCluster1.\top$
Concept_ContexteLocalisation_2 (CL2)	CL2	$\exists ValeurTemperatureCluster2.\top$
Concept_ContexteLocalisation_3 (CL3)	CL3	$Localisation$
Concept_ContexteTemperatureCorporelle_0 (CT0)	CT0	$\bot$
Concept_ContexteTemperatureCorporelle_1 (CT1)	CT1	$\exists TempPrecisionMoyenne.\top \sqcap \exists TempCapteur2.\top \sqcap \exists ValeurTemperatureCluster2.\top$
Concept_TemperatureCorporelle_2 (CT2)	CT2	$\exists TempCapteur1.\top \sqcap \exists ValeurTemperatureCluster3.\top \sqcap \exists TempPrecisionElevee.\top$
Concept_ContexteTemperatureCorporellle_3 (CT3)	CT3	$\exists TempCapteur1.\top \sqcap \exists ValeurTemperatureCluster1.\top \sqcap \exists TempPrecisionFaible.\top$
Concept_ContexteTemperatureCorporelle_4 (CT4)	CT4	$\exists ValeurTemperatureCluster1.\top$
Concept_ContexteTemperatureCorporelle_5 (CT5)	CT5	$TemperatureCorporelle$
Concept_ContextePressionArterielle_0 (CP0)	CP0	$\bot$
Concept_ContextePr	CP1	$\exists PressionPrecisionElevee.\top \sqcap \exists PressionCapteur2.\top$

essionArterielle_1 (CP1)		$\sqcap \exists ValeurPressionArterielleCluster3.\top$
Concept_ContextePr essionArterielle_2 (CP2)	CP2	$\exists PressionPrecisionMoyenne.\top \sqcap \exists PressionCapteur2.\top \sqcap \exists ValeurPressionArterielleCluster2.\top$
Concept_ContextePr essionArterielle_3 (CP3)	CP3	$\exists PressionPrecisionFaible.\top \sqcap \exists PressionCapteur1.\top \sqcap \exists ValeurPressionArterielleCluster1.\top$
Concept_ContextePr essionArterielle_4 (CP4)	CP4	$\exists PressionCapteur2.\top$
Concept_ContextePr essionArterielle_5 (CP5)	CP5	$PressionArterielle$

Le tableau B.3 présente tous les axiomes d'inclusion de la base de connaissances.

Tableau B.3 TBox : Axiomes d'inclusion

Axiomes d'inclusion
$CS12 \sqsubseteq CS13, CS11 \sqsubseteq CS13, CS9 \sqsubseteq CS13, CS4 \sqsubseteq CS13, CS15 \sqsubseteq CS13, CS1 \sqsubseteq CS13, CS5 \sqsubseteq CS13, CS14 \sqsubseteq CS15, CS2 \sqsubseteq CS15, CS3 \sqsubseteq CS14, CS7 \sqsubseteq CS14, CS6 \sqsubseteq CS14, CS10 \sqsubseteq CS14, CS8 \sqsubseteq CS14, CS0 \sqsubseteq CS12, CS0 \sqsubseteq CS11, CS0 \sqsubseteq CS9, CS0 \sqsubseteq CS5, CS0 \sqsubseteq CS4, CS0 \sqsubseteq CS3, CS0 \sqsubseteq CS7, CS0 \sqsubseteq CS6, CS0 \sqsubseteq CS10, CS0 \sqsubseteq CS8, CS0 \sqsubseteq CS2, CS0 \sqsubseteq CS1, CL1 \sqsubseteq CL3, CL2 \sqsubseteq CL3, CL0 \sqsubseteq CL1, CL0 \sqsubseteq CL2, CT4 \sqsubseteq CT5, CT1 \sqsubseteq CT5, CT3 \sqsubseteq CT4, CT2 \sqsubseteq CT4, CT0 \sqsubseteq CT3, CT0 \sqsubseteq CT2, CT0 \sqsubseteq CT1, CP3 \sqsubseteq CP5, CP4 \sqsubseteq CP5, CP2 \sqsubseteq CP4, CP1 \sqsubseteq CP4, CP0 \sqsubseteq CP3, CP0 \sqsubseteq CP2, CP0 \sqsubseteq CP1$

B.2 ABox de la base de connaissances

Le tableau B.4 présente tous les instances de rôles de la base de connaissances.

Tableau B.4 ABox : Individus

Individus
LocalisationUtilisateur1, LocalisationUtilisateur2, LocalisationUtilisateur3, TemperatureCorporelle1, TemperatureCorporelle2, TemperatureCorporelle3, PressionArterielle1, PressionArterielle2, PressionArterielle3, ServiceCommunication, ServiceAuthentification, ServiceGeoLocalisation, ServiceNotificationPatient, ServiceAlarme, ServiceCapteur, ServiceSoins, ServiceOptionPrioriteAlerte, ServiceAvertisseur, ServiceOptionAlarme, ServiceSignesVitaux, ServiceEnvironnement

Le tableau B.5 présente tous les instances de rôles de la base de connaissances.

Tableau B.5 ABox : Instance de rôle

Instance de rôle
dependLocalisation(ServiceGeoLocalisation, LocalisationUtilisateur1), dependLocalisation(ServiceGeoLocalisation, LocalisationUtilisateur2), dependLocalisation(ServiceGeoLocalisation, LocalisationUtilisateur3), dependLocalisation(ServiceAlarme, LocalisationUtilisateur1), dependLocalisation(ServiceAlarme, LocalisationUtilisateur2), dependLocalisation(ServiceAlarme, LocalisationUtilisateur3), dependLocalisation(ServiceCapteur, LocalisationUtilisateur1), dependLocalisation(ServiceCapteur, LocalisationUtilisateur2), dependLocalisation(ServiceCapteur, LocalisationUtilisateur3), dependLocalisation(ServiceSoins, LocalisationUtilisateur1), dependLocalisation(ServiceSoins, LocalisationUtilisateur2), dependLocalisation(ServiceSoins, LocalisationUtilisateur3) dependLocalisation(ServiceOptionAlarme, LocalisationUtilisateur1), dependLocalisation(ServiceOptionAlarme, LocalisationUtilisateur2), dependLocalisation(ServiceOptionAlarme, LocalisationUtilisateur3), dependLocalisation(ServiceSignesVitaux, LocalisationUtilisateur1), dependLocalisation(ServiceSignesVitaux, LocalisationUtilisateur2), dependLocalisation(ServiceSignesVitaux, LocalisationUtilisateur3), dependTempCorporelle(ServiceGeoLocalisation, TemperatureCorporelle1), dependTempCorporelle(ServiceGeoLocalisation, TemperatureCorporelle2), dependTempCorporelle(ServiceGeoLocalisation, TemperatureCorporelle3), dependTempCorporelle(ServiceAlarme, TemperatureCorporelle1), dependTempCorporelle(ServiceAlarme, TemperatureCorporelle2), dependTempCorporelle(ServiceAlarme, TemperatureCorporelle3), dependTempCorporelle(ServiceCapteur, TemperatureCorporelle1), dependTempCorporelle(ServiceCapteur, TemperatureCorporelle2), dependTempCorporelle(ServiceCapteur, TemperatureCorporelle3), dependTempCorporelle(ServiceSoins, TemperatureCorporelle1), dependTempCorporelle(ServiceSoins, TemperatureCorporelle2), dependTempCorporelle(ServiceSoins, TemperatureCorporelle3) dependTempCorporelle(ServiceOptionAlarme, TemperatureCorporelle1), dependTempCorporelle(ServiceOptionAlarme, TemperatureCorporelle2), dependTempCorporelle(ServiceOptionAlarme, TemperatureCorporelle3), dependTempCorporelle(ServiceSignesVitaux, TemperatureCorporelle1), dependTempCorporelle(ServiceSignesVitaux, TemperatureCorporelle2), dependTempCorporelle(ServiceSignesVitaux, TemperatureCorporelle3), dependPressionArterielle(ServiceGeoLocalisation, PressionArterielle1), dependPressionArterielle(ServiceGeoLocalisation, PressionArterielle2), dependPressionArterielle(ServiceGeoLocalisation, PressionArterielle3), dependPressionArterielle(ServiceAlarme, PressionArterielle1), dependPressionArterielle(ServiceAlarme, PressionArterielle2), dependPressionArterielle(ServiceAlarme, PressionArterielle3), dependPressionArterielle(ServiceCapteur, PressionArterielle1), dependPressionArterielle(ServiceCapteur, PressionArterielle2), dependPressionArterielle(ServiceCapteur, PressionArterielle3), dependPressionArterielle(ServiceSoins, PressionArterielle1), dependPressionArterielle(ServiceSoins, PressionArterielle2),

dependPressionArterielle(ServiceSoins, PressionArterielle3), dependPressionArterielle(ServiceOptionAlarme, PressionArterielle1), dependPressionArterielle(ServiceOptionAlarme, PressionArterielle2), dependPressionArterielle(ServiceOptionAlarme, PressionArterielle3), dependPressionArterielle(ServiceSignesVitaux, PressionArterielle1), dependPressionArterielle(ServiceSignesVitaux, PressionArterielle2), dependPressionArterielle(ServiceSignesVitaux, PressionArterielle3)

Le tableau B.6 présente tous les instanciation de concept défini de la base de connaissances.

Tableau B.6 ABox : Instanciation de concept défini

Instanciation de concept défini
CS1(ServiceEnvironnement), CS2(ServiceSignesVitaux), CS3(ServiceOptionAlarme), CS4(ServiceAvertisseur), CS5(ServiceOptionPrioriteAlerte), CS6(ServiceSoins), CS7(ServiceCapteur), CS8(ServiceAlarme), CS9(ServiceNotificationPatient), CS10(ServiceGeoLocalisation), CS11(ServiceAuthentification), CS12(ServiceCommunication), CS13(ServiceCommunication), CS13(ServiceAuthentification), CS13(ServiceGeoLocalisation), CS13(ServiceNotificationPatient), CS13(ServiceAlarme), CS13(ServiceCapteur), CS13(ServiceSoins), CS13(ServiceOptionPrioriteAlerte), CS13(ServiceAvertisseur), CS13(ServiceOptionAlarme), CS13(ServiceSignesVitaux), CS13(ServiceEnvironnement), CS14(ServiceGeoLocalisation), CS14(ServiceAlarme), CS14(ServiceCapteur), CS14(ServiceSoins), CS14(ServiceOptionAlarme), CS14(ServiceSignesVitaux), CS15(ServiceGeoLocalisation), CS15(ServiceAlarme), CS15(ServiceCapteur), CS15(ServiceOptionAlarme), CL3(LocalisationUtilisateur1), CL3(LocalisationUtilisateur2), CL3(LocalisationUtilisateur3), CL1(LocalisationUtilisateur1), CL2(LocalisationUtilisateur2), CL2(LocalisationUtilisateur3), CT1(TemperatureCorporelle3), CT2(TemperatureCorporelle2), CT3(TemperatureCorporelle1), CT4(TemperatureCorporelle1), CT4(TemperatureCorporelle2), CT5(TemperatureCorporelle1), CT5(TemperatureCorporelle2), CT5(TemperatureCorporelle3), CP1(PressionArterielle3), CP2(PressionArterielle2), CP3(PressionArterielle1), CP4(PressionArterielle2), CP4(PressionArterielle3), CP5(PressionArterielle1), CP5(PressionArterielle2), CP5(PressionArterielle3)

Le tableau B.7 présente tous les assertions de concept primitif de la base de connaissances.

Tableau B.7 ABox : Assertions de concept primitif

Instanciation/assertion de concept primitif
F2Authentification(ServiceAuthentification), F41PrioriteAlerte(ServiceNotificationPatient), F42Alertes(ServiceNotificationPatient), F4NotificationPatient(ServiceNotificationPatient), F421AvertisseurEcran(ServiceAvertisseur), F422AvertisseurSonore(ServiceAvertisseur),

F12Zigbee(ServiceCommunication), F11Wifi(ServiceCommunication),
F1Communication(ServiceCommunication), F412Priorite(ServiceOptionPrioriteAlerte),
F411Fifo(ServiceOptionPrioriteAlerte), F611PressionArterielle(ServiceSignesVitaux),
F612TempCorporelle(ServiceSignesVitaux), F622Lumiere(ServiceEnvironnement),
F621TempAmbiante(ServiceEnvironnement), F5Alarme(ServiceAlarme),
F3GeoLocalisation(ServiceGeoLocalisation), F31GPS(ServiceGeoLocalisation),
F32Wifi(ServiceGeoLocalisation), F7PlanSoins(ServiceSoins),
F61CapteurPhysiologique(ServiceCapteur), F6Capteur(ServiceCapteur),
F62CapteurEnvironnement(ServiceCapteur), F52BoutonPanique(ServiceOptionAlarme),
F51DetectionChute(ServiceOptionAlarme), ValeurLocalisationCluster1(LocalisationUtilisateur1),
ValeurLocalisationCluster2(LocalisationUtilisateur2),
ValeurLocalisationCluster2(LocalisationUtilisateur3), Capteur1(TemperatureCorporelle1),
Capteur(TemperatureCorporelle2), ValeurTemperatureCluster2(TemperatureCorporelle3),
TempPrecisionMoyenne(TemperatureCorporelle3), TempCapteur2(TemperatureCorporelle3),
ValeurTemperatureCluster1(TemperatureCorporelle1),
TempPrecisionFaible(TemperatureCorporelle1), TempCapteur1(TemperatureCorporelle1),
ValeurTemperatureCluster3(TemperatureCorporelle2),
TempPrecisionElevee(TemperatureCorporelle2), TempCapteur1(TemperatureCorporelle2),
ValeurPressionArterielleCluster1(PressionArterielle1), PressionPrecisionFaible(PressionArterielle1),
PressionCapteur1(PressionArterielle1), PressionCapteur2(PressionArterielle2),
PressionCapteur2(PressionArterielle3), ValeurPressionArterielleCluster2(PressionArterielle2),
PressionPrecisionMoyenne(PressionArterielle2),
ValeurPressionArterielleCluster3(PressionArterielle3), PressionPrecisionElevee(PressionArterielle3)

PUBLICATIONS

Amja, A. M., Obaid, A., et Mili, H. (2016). Combining Variability, RCA and Feature Model for Context-Awareness. *The Sixth International Conference on Innovative Computing Technology (INTECH 2016)*, 15–23.

Amja, A. M., Obaid, A., et Mili, H. (2017 – sous presse). A Formal Framework for Adaptation. In *The Third International Symposium on Ubiquitous Networking*. Springer.

Amja, A. M., Obaid, A., Mili, H., et Jarir, Z. (2016). Feature-Based Adaptation and Its Implementation. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, 321–328.

Amja, A. M., Obaid, A., Mili, H., et Valtchev, P. (2016). Linking relational concept analysis and variability model within context modeling of context-aware applications. In *2016 IEEE International Symposium on Systems Engineering (ISSE)* (pp. 1–8). Edinburgh: IEEE.

Amja, A. M., Obaid, A., et Seguin, N. (2011). A Distributed Mobile Database Architecture. In *2011 IEEE Asia-Pacific Services Computing Conference (APSCC)*, 62–69.

Amja, A. M., Obaid, A., et Valtchev, P. (2014). Modeling and reasoning in context-aware systems based on relational concept analysis and description logic. In *2014 IEEE Symposium on Computational Intelligence for Communication Systems and Networks (CIComms)*, 1–8.

Obaid, A., Amja, A. M., Mili, H., et Seguin, N. (2012). Query execution on a mobile database system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12)* , 569. New York: ACM Press.

Boudra, H., Obaid, A., et Amja, A. M. (2014). An intelligent medical monitoring system based on sensors and wireless sensor network. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 1650–1656.

Revello, A., Obaid, A., et Amja, A. M. (2012). Control flow management in the hospitality industry. *In Telecommunication Networks and Applications Conceference (ATNAC)*, 1-6.

BIBLIOGRAPHIE

Abowd, G. D. et Mynatt, E. D. (2000). Charting past, present, and future research in ubiquitous computing. *ACM Trans. Comput.-Hum. Interact.*, *7*(1), 29–58.

Apel, S., Batory, D., Kastner, C. et Saake, G. (2013). *Feature-Oriented Software Product Lines*. Berlin Heidelberg: Springer.

Arbab, F. (2004). Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, *14*(3), 329–366.

Arbab, F. (2006). Coordination for Component Composition. *Electronic Notes in Theoretical Computer Science*, *160*(1), 15–40.

Baader, F., Calvanese, D., McGuiness, D. L., Nardi, D. et Patel-Schneider, P. F. (2003). *The Description Logic Handbook: Theory, Implementation and Applications* (2e éd.). New York: Cambridge University Press.

Baldauf, M., Dustdar, S. et Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, *2*(4), 263–277.

Bauer, J. (2003). Identification and Modeling of Contexts for Different Information Scenarios in Air Traffic (Thèse de doctorat) Université Technique de Berlin. Récupéré de http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.7895&rep=rep1&type=pdf

Bellavista, P., Corradi, A., Fanelli, M. et Foschini, L. (2012). A survey of context data distribution for mobile ubiquitous systems. *ACM Computing Surveys*, *44*(4),

1–45.

Bendaoud, R., Hacene, M. R., Toussaint, Y., Delecroix, B. et Napoli, A. (2007). Text-based ontology construction using relational concept analysis. *In Proceedings of the International Workshop on Ontology Dynamics (IWOD-07) at the European Semantic Web Conference (ESWC)*, 55-68.

Bendaoud, R., Napoli, A. et Toussaint, Y. (2008). Formal Concept Analysis: A Unified Framework for Building and Refining Ontologies. In *Knowledge Engineering: Practice and Patterns (EKAW 2008), LNCS 5268,* 156–171. Berlin, Heidelberg: Springer.

Bendaoud, R. et Toussaint, Y. (2010). L'Analyse Formelle de Concepts au service de la construction et l'enrichissement d'une ontologie, *Revue des Nouvelles Technologies de L'Information. Fouilles des données complexes: avancées récentes*, *18*, 133–164.

Benghozi, P., Bureau, S. et Massit-Folea, F. (2011). *L'Internet des objets. Quels enjeux pour les Européens?* Paris: Éditions de la Maison des sciences de l'homme.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A. et Riboni, D. (2010). A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, *6*(2), 161–180.

Bidarra, R. et Bronsvoort, W. F. (2000). Semantic feature modelling. *CAD Computer Aided Design*, *32*(3), 201–225.

Bikakis, A., Patkos, T., Antoniou, G. et Plexousakis, D. (2008). A Survey of Semantics-Based Approaches for Context Reasoning in Ambient Intelligence, *Communications in Computer and Information Science: Constructing Ambient*

Intelligence (AMI 2007), 11, 14–23, Berlin, Heidelberg: Springer.

Birkhoff, G. (1940). *Lattice Theory*. American Mathematical Society.

Bradbury, J. S., Cordy, J. R., Dingel, J. et Wermelinger, M. (2004). A survey of self-management in dynamic software architecture specifications, *In Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systens* (WOSS'04) 28-33.

Brézillon, P. et Gonzalez, A. J. (2014). *Context in Computing: A Cross-Disciplinary Approach for Modeling the Real World*, New York : Springer.

Brown, P. J. (1996). The stick-e document : a framework for creating context-aware applications. *Electronic Publishing*, *8*(2), 259–272.

Brown, P. J., Bovey, J. D. et Chen, X. (1997). Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications*, *4*(5), 58–64.

Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Muller, H., Pezze, M. et Shaw, M. (2009). Engineering self-adaptive systems through feedback loops. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *LNCS 5525*, 48–70.

Capilla, R. et Bosch, J. (2011). The promise and challenge of runtime variability. *Computer*, *44*(12), 93–95.

Carpineto, C. et Romano, G. (2004). *Concept Data Analysis: Theory and Applications*. John Wiley & Sons.

Cetina, C., Giner, P., Fons, J. et Pelechano, V. (2009). Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, *42*(10), 37–43.

Chen, G. C. G. et Kotz, D. (2002). Context aggregation and dissemination in ubiquitous computing systems. *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, 105-114.

Chen, G. et Kotz, D. (2002a). Solar : A pervasive-computing infrastructure for context-aware mobile applications (Rapport de recherche TR2002-421). Récupéré de http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA440303

Chen, G. et Kotz, D. (2002b). Solar: An open platform for context-aware mobile applications. *Proceedings of the First International Conference on Pervasive Computing*, 41–47.

Cheverst, K., Mitchell, K. et Davies, N. (1999). Design of an Object Model for a Context-Sensitive Tourist Guide, *Elsevier Science, Computers & Graphics,* 23(6), 883-891.

Clements, P. et Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. Boston: Addison-Wesley.

Cook, D. J., Augusto, J. C. et Jakkula, V. R. (2009). Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, *5*(4), 277–298.

Cuesta, C. E., De La Fuente, P. et Barrio-Solárzano, M. (2001). Dynamic coordination architecture through the use of reflection. *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC '01)*, 134–140.

Da, K., Dalmau, M. et Roose, P. (2012). A Survey of Adaptation Systems. *International Journal on Internet and Distributed Computing Systems*, *2*(1), 1–

18.

Dargie, W. (2009). *Context-Aware Computing and Self-Managing Systems*. Boca Raton: Chapman and Hall/CRC.

Dey, A., Abowd, G. et Salber, D. (2001). A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction*, *16*(2), 97–166.

Dey, A. K. et Abowd, G. D. (1999). Towards a Better Understanding of Context and Context-Awareness. *Computing Systems*, *40*(3), 304–307.

Dolques, X. (2014). RCAExplore Relational Concept Analysis and Exploration. Récupéré de http://dolques.free.fr/rcaexplore/

Fokkink, W. (2007). *Introduction to process algebra. Computer Science Monograph,* Berlin, Heidelberg: Springer.

Franklin, D. et Flaschbart, J. (1998). All gadget and no representation makes jack a dull environment. *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environment*, 155–150.

Gan, G., Ma, C. et Wu, J. (2007). *Data Clustering: Theory, Algorithms, and Applications* (ASA-SIAM S). Philadelphia: Society for Industrial and Applied Mathematics.

Ganter, B. et Wille, R. (1989). Conceptual Scaling. Applications of Combinatorics and Graph Theory to the Biological and Social Sciences. The IMA Volumes in Mathematics and Its Applications, 17, 139-167, New York: Springer.

Ganter, B. et Wille, R. (1999). *Formal Concept Analysis*. Berlin, Heidelberg: Springer.

Gardini, L. M., Braga, R., Bringel, J., Oliveira, C., Andrade, R., Martin, H., Luiz, O., Andrade, M. et Oliveira, M. (2013). Clariisa, a context-aware framework based on geolocation for a health care governance system. *2013 IEEE 15th International Conference on E-Health Networking, Applications and Services, (Healthcom 2013)*, 334–339.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, *5*(2), 199–220.

Gu, T. (2004). A middleware for building context-aware mobile services. 2004 IEEE 59th *Vehicular Technology Conference (VTC 2004-Spring)*, 5, 2656–2660.

Hacene Rouane, M., Huchard, M., Napoli, A. et Valtchev, P. (2007). A Proposal for Combining Formal Concept Analysis and Description Logics for Mining Relational Data. *ICFCA'07: 5th International Conference Formal Concept Analysis*, *4390*, 51–65.

Held, A., Buchholz, S. et Schill, A. (2002). Modeling of context information for pervasive computing applications. *Procceding of the World Multiconference on Systemics, Cybernetics and Informatics,* 1–6.

Henricksen, K. et Indulska, J. (2004). Modelling and using imperfect context information. *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops (PerCom 2004)*, 33–37.

Henricksen, K., Indulska, J. et Rakotonirainy, A. (2003). Generating context management infrastructure from high-level context models. *In 4th International Conference on Mobile Data Management (MDM)-Industrial Track Proceedings*, 1–6.

Hoareau, C. et Satoh, I. (2009). Modeling and processing information for context-

aware computing: A survey. *New Generation Computing*, *27*(3), 177–196.

Huchard, M., Hacene, M. R., Roume, C. et Valtchev, P. (2007). Relational concept discovery in structured datasets. *Annals of Mathematics and Artificial Intelligence*, *49*(1–4), 39–76.

IBM. (2006). An architectural blueprint for autonomic computing. *IBM White Paper*.

Jacob, B., Lanyon-Hogg, R., Nadgir, D. K. et Yassin, A. F. (2004). *A Practical Guide to the IBM Autonomic Computing Toolkit*. IBM Redbooks.

Jongmans, S. S. T. Q. et Arbab, F. (2012). Overview of thirty semantic formalisms for Reo. *Scientific Annals of Computer Science*, *22*(1), 201–251.

Knappmeyer, M., Kiani, S. L., Frà, C., Moltchanov, B. et Baker, N. (2010). ContextML: A light-weight context representation and context management schema. *IEEE 5th International Symposium on Wireless Pervasive Computing (ISWPC 2010)*, 367–372.

Kramer, J. et Magee, J. (2007). Self-managed systems: An architectural challenge. *FoSE 2007: Future of Software Engineering*, 259–268.

Lalanda, P., McCann, J. A. et Diaconescu, A. (2013). *Autonomic Computing Principles, Design and Implementation. Technology*. London: Springer.

Lee, J. et Kang, K. C. (2004). Feature Binding Analysis for Product Line Component Development. In *Software Product-Family Engineering*, 250–260. Springer.

Lee, J. et Muthig, D. (2008). Feature-Oriented Analysis and Specification of Dynamic Product Reconfiguration. *High Confidence Software Reuse in Large Systems*, 154–165.

Lee, J. et Muthig, D. (2009). Feature Oriented Analysis and Design for Dynamically

Reconfigurable Product Lines. In *Applied Software Product Line Engineering*, 315–336. Auerbach Publications.

Levandoski, J. J. (2010). CareDB: A context and preference-aware location-based database system. *IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, 317–320.

Li, X., Eckert, M., Martinez, J. F. et Rubio, G. (2015). Context aware middleware architectures: Survey and challenges. *Sensors*, *15*(8), 20570–20607.

Loke, S. (2006). *Context-Aware Pervasive Systems*. Boston: Auerbach Publications.

Lyu, C. H., Choi, M. S., Li, Z. Y. et Youn, H. Y. (2010). Reasoning with imprecise context using improved dempster-shafer theory. *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (IAT 2010)*, *2*, 475–478.

MacQueen, J. B. (1967). Kmeans Some Methods for classification and Analysis of Multivariate Observations. *5th Berkeley Symposium on Mathematical Statistics and Probability 1967*, *1*(233), 281–297.

Mamo, B. et Ejigu, D. (2014). A generic layered architecture for context aware applications. *Procedia Computer Science*, *34*, 619–624.

McCarthy, J. et Buvac, S. (1997). *Formalizing Context (Expanded Notes)*. Computing Natural Language, 13-50. Récupéré de http://www-formal.stanford.edu/jmc/mccarthy-buvac-98/context.pdf

Milner, R. (1980). *A Calculus of Communicating Systems*. Berlin, Heidelberg: Springer.

Miraoui, M., Tadj, C. et Ben Amar, C. (2008). Architectural Survey of Context-

Aware Systems in Pervasive Computing Environment. *Ubiquitous Computing and Communication Journal*, *3*(3), 1–9.

Mokbel, M. F. et Levandoski, J. J. (2009). Toward context and preference-aware location-based services. *Proceedings of the Eighth ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE '09)*, 25-32.

Musumba, G. W. et Nyongesa, H. O. (2013). Context awareness in mobile computing: A review. *International Journal of Machine Learning and Applications*, *2*(1), 1–10.

Napoli, A. (1997). Une introduction aux logiques de descriptions, (RR-3314), INRIA. Récupéré de https://hal.archives-ouvertes.fr/inria-00073375/document

Naqvi, M. (2012). Claims and Supporting Evidence for Self-Adaptive Systems – A Literature Review (Mémoire de maîtrise). Récupéré de http://lnu.diva-portal.org/smash/record.jsf?pid=diva2%3A512439&dswid=3943

Noorian, M., Ensan, A., Bagheri, E., Boley, H. et Biletskiy, Y. (2011). Feature Model Debugging based on Description Logic Reasoning. *The 17th International Conference of Distributed Multimedia Systems (DMS 2011)*, 158–164.

Nosrati, M., Karimi, R. et Hasanvand, H. A. (2012). Mobile Computing : Principles , Devices and Operating Systems. *World Applied Programming*, *2*(7), 399–408.

Pascoe, J., Ryan, N. S. et Morse, D. R. (1998). Human Computer Giraffe Interaction: HCI in the Field (Rapport de recherche G98-1). Récupéré de https://www.cs.kent.ac.uk/pubs/1998/617/

Patikirikorala, T., Colman, A., Han, J. et Wang, L. (2012). A Systematic Survey on the Design of Self-Adaptive Software Systems Using Control Engineering Approaches. *7th International Symposium on Software Engineering for Adaptive*

and Self-Managing Systems (SEAMS 2012), 33–42.

Pellet. (2015). Récupéré de http://pellet.owldl.com/

Perera, C., Zaslavsky, A., Christen, P. et Georgakopoulos, D. (2014). Context aware computing for the internet of things: A survey. *IEEE Communications Surveys and Tutorials*, *16*(1), 414–454.

Perttunen, M., Riekki, J. et Lassila, O. (2009). Context Representation and Reasoning in Pervasive Computing: A Review. *International Journal of Multimedia and Ubiquitous Engineering, 4*(4), 1–28.

Piash, M. M., Ripon, S. et Hossain, S. A. (2013). A Semantic Web Approach to VerifyingProduct Line Variant Requirements. In *Fourth International Conference on Advances in Communication, Network and Computing*, 186–191.

Protégé Ontology Editor. (2015). Récupéré de http://protege.stanford.edu/

Ranganathan, A. et Campbell, R. H. (2003). An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, *7*(6), 353–364.

Ravindranathan, M. et Leitch, R. (1998). Heterogeneous intelligent control systems. *IEE Proceedings - Control Theory and Applications*, *145*(6), 551-558.

Rouane-Hacene, M., Huchard, M., Napoli, A. et Valtchev, P. (2013a). Relational concept analysis: Mining concept lattices from multi-relational data. *Annals of Mathematics and Artificial Intelligence*, *67*(1), 81–108.

Rouane-Hacene, M., Huchard, M., Napoli, A. et Valtchev, P. (2013b). Soundness and Completeness of Relational Concept Analysis. *11th International Conference on Formal Concept Analysis (ICFCA 2013)*, 228–243.

Royer, J.-C. et Arboleda, H. (2012). *Model-Driven and Software Product Line*

Engineering. Wiley-ISTE.

Ryan, N. S., Pascoe, J. et Morse, D. R. (1998). Enhanced Reality Fieldwork: the Context-aware Archaeological Assistant. *Computer Applications in Archaeology 1997*, 182–196.

Salehie, M., et Tahvildari, L. (2012). Towards a goal-driven approach to action selection in self-adaptive software. *Software: Practice and Experience*, *42*(2), 211–233.

Schilit, B. N., Adams, N. et Want, R. (1994). Context-aware computing applications. *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, 85-90.

Schilit, B. N. et Theimer, M. M. (1994). Disseminating active map information to mobile hosts. *IEEE Network*, *8*(5), 22–32.

Schmidt, A., Beigl, M. et Gellersen, H. W. (1999). There is more to context than location. *Computers and Graphics (Pergamon)*, *23*(6), 893–901.

Strang, T. et Linnhoff-Popien, C. (2004). A Context Modeling Survey. *First International Workshop on Advanced Context Modelling, Reasoning and Management at UbiComp*, 1–8.

Svahnberg, M., Van Gurp, J. et Bosch, J. (2005). A taxonomy of variability realization techniques. *Software: Practice and Experience*, *35*(8), 705–754.

SWI-Prolog. (2016). SWI-Prolog. Récupéré de http://www.swi-prolog.org/

Tamura, G., Villegas, N. M., Muller, H. A., Duchien, L. et Seinturier, L. (2013). Improving context-awareness in self-adaptation using the DYNAMICO reference model. *ICSE Workshop on Software Engineering for Adaptive and*

Self-Managing Systems, 153–162.

Turhan, A. Y., Springer, T. et Berger, M. (2006). Pushing doors for modeling contexts with OWL DL -a case study. *Proceedings - Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW 2006)*, 13–17.

Villegas, N. M., Muller, H. A. et Tamura, G. (2011). On Designing Self-Adaptive Software Systems. *Sistemas Y Telematica*, *9*, 29–51.

Vogel, T. et Giese, H. (2013). *Model-driven engineering of adaptation engines for self-adaptive software: Executable runtime megamodels*. (Techinical report 66) Récupéré de https://publishup.uni-potsdam.de/opus4-ubp/frontdoor/deliver/index/docId/6255/file/tbhpi66.pdf

Vogel, T. et Giese, H. (2014). Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems*, *8*(4), 1–33.

W3C. (2004). SWRL:A Semantic Web Rule Language Combining OWL and RuleML. Récupéré de https://www.w3.org/Submission/SWRL/

Wang, H. H., Li, Y. F., Sun, J., Zhang, H. et Pan, J. (2007). Verifying feature models using OWL. In *Web Semantics: Science, Services and Agents on the World Wide Web, 5*(2), 117–129.

Wang, H., Li, Y., Sun, J., Zhang, H. et Pan, J. (2005). A semantic web approach to feature modeling and verification. In *1st Workshop on Semantic Web Enabled Software Engineering,* 1-15.

Wang, X. H., Zhang, D. Q., Gu, T. et Pung, H. K. (2004). Ontology based context modeling and reasoning using OWL. In *IEEE Annual Conference on Pervasive*

Computing and Communications Workshops, 2004. Proceedings of the Second, 18–22.

Weiser, M. (2002). The computer for the 21st Century. *IEEE Pervasive Computing*, *1*(1), 19–25.

Weyns, D., Iftikhar, M. U., Malek, S. et Andersson, J. (2012). Claims and supporting evidence for self-adaptive systems: A literature study. *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 89–98.

Weyns, D., Malek, S. et Andersson, J. (2012). FORMS : Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems*, *7*(1), 8–61.

Zhang, D., Huang, H., Lai, C. F., Liang, X., Zou, Q. et Guo, M. (2013). Survey on context-awareness in ubiquitous media. *Multimedia Tools and Applications*, *67*(1), 179–211.