Adaptation dynamique par tissage d'aspects d'assemblage

Daniel Cheung-Foo-Wo

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention : Informatique

présentée et soutenu par

Daniel CHEUNG-FOO-WO

ADAPTATION DYNAMIQUE PAR TISSAGE D'ASPECTS D'ASSEMBLAGE

Thèse dirigée par *Michel RIVEILL*

soutenue le *5 mars 2009*

Jury :

Mme Noëmie SIMONI	Professeur	Présidente
M. Lionel SEINTURIER	Professeur	Rapporteur
M. Didier DONSEZ	Professeur	Rapporteur
M. Michel RIVEILL	Professeur	Directeur
M. Jean-Yves TIGLI	Docteur	Co-Directeur
M. Stéphane LAVIROTTE	Docteur	Invité
M. Eric PASCUAL	Ingénieur	Invité

Université de Nice-Sophia Antipolis - UFR Sciences

École doctorale STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

Thèse

pour obtenir le grade de

Docteur en Sciences

de l'université de Nice-Sophia Antipolis

Mention : Informatique

présentée par

Daniel CHEUNG-FOO-WO

Adaptation dynamique par tissage d'aspects d'assemblage

Professeur	Noëmie	SIMONI	Présidente du jury
Professeur	Lionel	SEINTURIER	Rapporteur
Professeur	Didier	DONSEZ	Rapporteur
Professeur	Michel	RIVEILL	Directeur
Docteur	Jean-Yves	TIGLI	Co-directeur
Docteur	Stéphane	LAVIROTTE	Invité
Ingénieur	Éric	PASCUAL	Invité

Table des matières

Résumé

L'Informatique Ambiante (IAm) engage de nombreux capteurs et actionneurs variés intégrés aux objets du quotidien. Ces dispositifs collaborent pour faire émerger de manière spontanée de nouvelles applications logicielles. Les fonctionnalités de ces applications sont adaptées aux dispositifs disponibles de l'environnement physique. L'objectif est de proposer une approche originale pour développer ces nouvelles applications IAm. Ces dernières doivent prendre en compte un ensemble de dispositifs temporaires et non connus à priori. Nous présentons notre approche s'appuyant sur les techniques de programmation à base de composants logiciels et sur le concept d'*aspect d'assemblage*, une évolution de l'AOP de Kiczales, qui spécifie les mécanismes de composition automatique et les principes de gestion d'interférences potentielles entre les spécifications d'adaptation. Une approche à base de règles logiques permet de résoudre les problèmes d'interférence. Une application s'adapte alors par tissage – composition et gestion d'interférences – entre ces aspects d'assemblage. Ces travaux ont permis la réalisation d'une extension de la plate-forme logicielle WComp et la mise en œuvre de plusieurs prototypes d'expérimentation IAm, notamment dans le domaine du bâtiment intelligent.

Summary

Ambient computing uses various devices integrated to objects of our everyday life. Those devices collaborate to build dynamically new applications according to the set of ambient devices which are not necessarily known in advance. A new approach is proposed to develop componentized applications for ambient computing which rely on a new concept called *aspect of assembly* to manage independent, automatic and structural composition of Ambient applications and their possible intrinsic interferences. These interferences are automatically solved by using merging rules specified in a logical formalism. An application adaptation is then developed by weaving (composing interferences solving) aspects of assembly. This work enabled the implementation of an extension of the WComp component framework and several experimental prototypes in Ambient computing, namely for smart building domain.

Première partie

Introduction

Chapitre 1

Introduction

Sommaire

1.1 De l'informatique à l'intelligence ambiante

L'informatique ambiante nous entoure. Elle est invisible. Cela ne signifie cependant pas que l'on ne peut plus la voir avec nos yeux. Cela concerne ce sur quoi notre attention et notre conscience se focalisent. Ainsi, les technologies tendent à disparaître pour se mélanger aux éléments de la vie de tous les jours [1]. Par exemple, des panneaux d'information (angl. *live boards*), qui mesurent de l'ordre d'un mètre de long, portent sur un événement précis qui est affiché en direct. Dans la figure 1.1, nous avons représenté un de ces panneaux. Celui-ci permet de retransmettre les résultats d'un match de base-ball en direct chez soi sur un panneau d'affichage réduit.

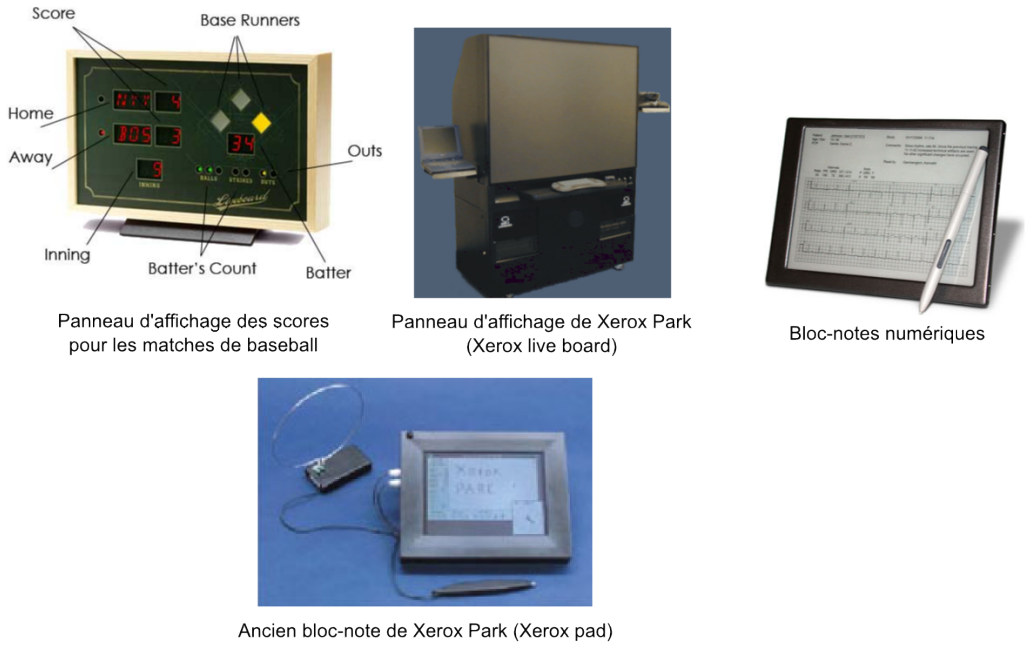


FIGURE 1.1 – Les technologies se fondent dans notre environnement de tous les jours : exemples de blocs-notes numériques et de panneaux d'affichage. Ces dispositifs informatiques fournissent de l'information, du savoir. Sont-ils suffisants pour matérialiser le vaste domaine que constitue l'intelligence ambiante ?

On trouve également des blocs-notes électroniques (angl. *pads*) qui sont des ordinateurs de la taille d'une page A4 et on les considère comme du "papier brouillon" ou "ordinateur brouillon" (angl. *scrap computer*). Pour les utiliser, on en dispose quelques-uns sur un bureau. On peut alors les organiser. Chacun constitue alors un travail à faire, ou bien on peut les utiliser comme des pense-bêtes. Enfin, il existe des badges actifs mesurant quelques centimètres utilisés pour décliner son identité. Ils servent à ouvrir des portes automatiques, déclencher des transferts d'appel lorsque l'on se déplace d'un bureau à un autre. Ils peuvent également servir à personnaliser l'affichage d'applications sur des écrans d'ordinateurs [1].

En informatique ambiante, les problèmes du développement d'applications adaptatives sont (1) l'impossibilité de prévoir toutes les situations dans lesquelles l'application sera utilisée et (2) la complexité grandissante de l'application à cause du mélange entre le code chargé de l'adapter et le code fonctionnel (métier) dans un système aux ressources limitées (espace mémoire, vitesse de calcul, faible portée des communications, etc.). L'objectif de cette thèse est de proposer une méthode de développement d'applications adaptatives pour l'informatique ambiante qui prend en compte ces deux problèmes. La thèse défendue se résume à rendre possible la reconfiguration dynamique de l'architecture logicielle en considérant les adaptations comme des `aspects` développés de manière séparée tout en gérant les problèmes d'interférences entre les adaptations.

1.2 L'intelligence ambiante (IAm)

L'intelligence ambiante (IAm) [2] se réfère aux environnements informatiques sensibles au monde physique. Elle est également connue sous les appellations anglo-saxonnes suivantes : *ubiquitous computing*, *calm technology*, *pervasive computing* et *everyware*. L'IAm va au-delà des objets réels informatisés, elle s'intègre à l'environnement de tous les jours en s'imprégnant des phénomènes physiques qui s'y manifestent.

1.2.1 L'IAm regroupe divers domaines informatiques

Plusieurs travaux ont été menés en informatique ambiante [3], notamment en ingénierie logicielle pour ce qui concerne la programmation de systèmes informatiques ambiants pour, par exemple, proposer une `programmation ambiante` [4]. L'informatique ambiante a pour origine l'*ubiquituous computing* de M. Weiser dont le but originel est d'intégrer de manière transparente des dispositifs informatiques au monde physique. Cette intégration offre aujourd'hui des modes d'interaction plus naturels entre les utilisateurs et leur milieu. L'utilisateur n'a pas conscience d'utiliser les services fournis par des unités de calcul l'entourant. L'informatique ambiante apparaît comme une convergence entre l'étude des objets sensibles à leur environnement (`médias tangibles` [5]) et les accès distants à nos informations personnelles ainsi qu'à leur traitement.

Il en découle ainsi le besoin d'embarquer des données de nature diverse dans des ordinateurs de plus en plus légers et spécialisés. De cette convergence, nous voyons la naissance du néologisme *everyware* d'Adam Greenfield [6] formé par la fusion du terme anglais *everywhere* signifiant "partout" et *software* pour "logiciel". Toutefois, Greenfield va au-delà de l'intégration de ces accès distants et de médias tangibles. Il propose d'y intégrer également les principes liés à l'informatique diffuse [7] et la prise en compte du contexte que nous allons expliquer plus en détail. Intéressons-nous d'abord à la définition des termes *ubiquitaire*, *diffus*, *ambiant*, *contextuel* et *tangible*. Voici les définitions extraites du dictionnaire :

- Ubiquitaire [1] (angl. *ubiquitous*) : qui est partout ;
- Diffus [7, 8, 9] (angl. *pervasive, disappearing*) : qui est intimement mêlé à ce qui l'entoure ;
- Ambiant [3] (angl. *ambient*) : qui entoure ou circule autour, qui environne ;
- Tangible [5] (angl. *tangible*) : qui est perceptible par le toucher ;
- Sensible au contexte [10] (angl. *context-aware*) : qui prend en compte ce qui l'entoure ;
- Autonome [11] (angl. *autonomic*) : qui est construit ou qui fonctionne comme un tout indépendant.

Nous apportons une nuance dans l'emploi du terme *informatique diffuse*, notion traduite de l'anglais *pervasive* [7, 8]. En effet, l'informatique diffuse fait référence à la miniaturisation des dispositifs communicants (processeurs et capteurs) pour une utilisation répandue. L'adjectif *ambiant* désigne plutôt l'environnement artificiel et caractérise mieux le domaine de notre étude. Dans cet environnement, les réseaux d'unités de calcul évoluent dans le monde réel où les médias de communication et les interfaces tangibles intègrent des objets de la vie quotidienne.

En conclusion, la recherche en *informatique ambiante* constitue alors l'étude de l'évolution logicielle dans un `espace ambiant`. Il prend en compte l'hétérogénéité des dispositifs. L'informatique diffuse se concentre sur la miniaturisation tout en conservant leurs capacités mémoire et de calcul. L'informatique ambiante met l'accent sur la gestion de l'hétérogénéité et sur les changements de l'environnement du système.

1.2.2 Quelques scénarios

L'intelligence ambiante est la description d'un monde dans lequel on envisage d'entourer les personnes de différents dispositifs informatiques intelligents ayant une utilisation intuitive. Ces dispositifs sont alors enfouis dans les objets de tous les jours. Ils reconnaissent et répondent à la présence et au comportement d'un ou de plusieurs individus de manière personnalisée et utile.

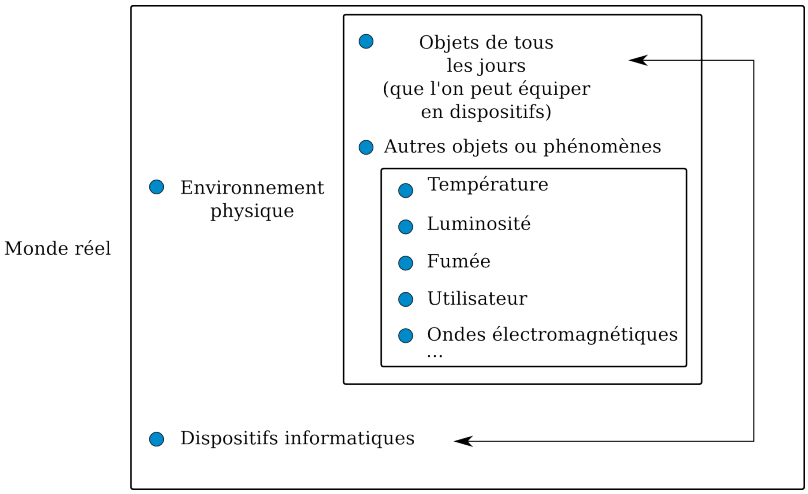


FIGURE 1.2 – Intelligence ambiante : des personnes entourées de dispositifs enfouis dans des objets de tous les jours. Toutefois, dans ce monde, tout élément de l'environnement physique ne peut pas être équipé, de manière exhaustive, de dispositifs informatiques pour la capture d'informations physiques (température, fumée, ...).

1.2.2.1 Scénarios en général

L'informatique ambiante (IAm) constitue le résultat de la fusion de deux grands domaines d'étude scientifique : les dispositifs intelligents et l'informatique sensible au contexte. Les dispositifs intelligents sont des dispositifs informatiques communiquant entre eux et intégrés à l'environnement physique. L'informatique sensible au contexte désigne les dispositifs informatiques qui peuvent reconnaître les utilisateurs, leurs comportements, leurs situations, etc.

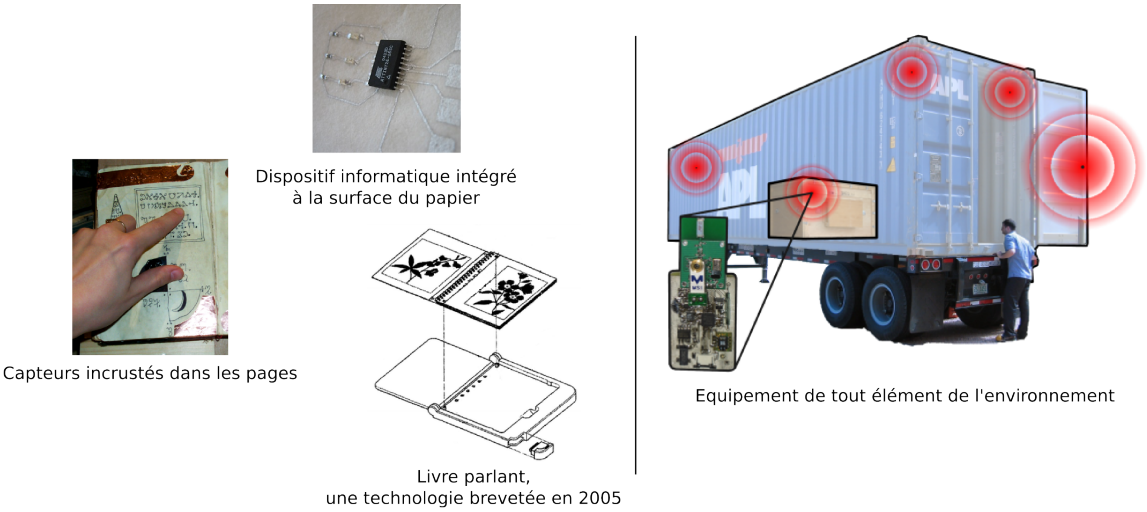


FIGURE 1.3 – L'expérience de la vie de tous les jours est enrichie par les dispositifs informatiques. En plus de l'information et du savoir, ces dispositifs fournissent du sens à l'environnement. Ils le complètent en synthétisant de nouvelles *réactions physiques* (comme l'exemple des livres d'H. Liu que l'on juxtapose pour faire apparaître leurs points communs).

Par exemple, H. Liu illustre l'IAm par ce qu'il appelle la "sémantique ambiante", c'est-à-dire un enrichissement de l'expérience de la vie de tous les jours. Il propose d'abord des livres intelligents : si le livre a été lu par des célébrités ou des amis du lecteur, ce dernier en est informé. Il lui est

même précisé quels passages ont été appréciés par ces précédents lecteurs. Lorsque l'on juxtapose deux de ces livres, on obtient alors immédiatement des informations sur les points communs qui ne sont pas connus à première vue comme leur thème, etc.

1.2.2.2 Scénario appliqué au bâtiment

Pour illustrer plus en détail les applications de l'informatique ambiante, nous présentons le scénario suivant :

...Hier, Marc a acheté une maison intelligente à partir d'un catalogue sorti au début de l'année 2010. Le catalogue proposait des façades équipées de lampes RF (sans fil), des murs intérieurs pré-équipés de divers capteurs et de luminaires. Il a sélectionné les différents murs pour la cuisine, pour le salon et la chambre comme s'il choisissait dans une boîte de jeu pour enfant des pièces composables pour construire la maquette de ses rêves.

Aujourd'hui, les techniciens préparent et installent la maison de Marc. Ils fabriquent des murs en plâtre contenant quelques câbles d'alimentation pour des boîtiers dits de collaboration. Ces derniers ressemblent à des automates industriels miniatures. On trouve également des capteurs et des actionneurs sans fil pour les luminaires et les interrupteurs et des câbles supplémentaires pour les volets roulants électriques.

Ces techniciens ont eu une formation en "informatique ambiante" sur la connexion de réseaux numériques, les câbles d'alimentation, la configuration de la localisation géographique de divers capteurs et actionneurs. Au niveau logiciel, ils savent également configurer les boîtiers de collaboration consistant à sélectionner du "câblage virtuel" (association capteurs-actionneurs) pour la maison. Ils savent également que ces boîtiers contiennent des programmes adaptatifs qui, selon le `contexte`*, seront capables de faire collaborer efficacement les objets tangibles et tout l'électronique de la maison. Les techniciens vérifient que leur installation est terminée grâce à un boîtier de collaboration indiquant les stratégies opérationnelles pour la gestion de la maison et donnant la liste des dispositifs ambiants découverts et utilisés.*

Marc emménage le lendemain. La maison peut le reconnaître grâce à une lecture de badge RF. Mais ce qui intéresse Marc, c'est le caractère non-intrusif des machines, c'est-à-dire leur capacité à fonctionner sans intervention humaine. Le système ambiant que constitue maintenant la maison récupère, grâce aux capteurs, des informations stockées et analysées sur certains boîtiers électroniques. Ces boîtiers fournissent des informations de haut niveau à des boîtiers de stratégies pour rendre la maison confortable et autonome.

Ainsi, les différents sous-systèmes de la maison communiquent entre eux pour, par exemple, tenir compte de la chaleur dégagée lors de la préparation d'un repas et ainsi optimiser les consignes de chauffage. Le système de gestion des luminaires s'imbrique implicitement avec le système de détection de présence et d'ouverture électrique pour optimiser l'éclairage. Différentes stratégies peuvent être mises en place, soit par l'occupant s'il connaît le fonctionnement des boîtiers de stratégie, soit par l'entreprise d'installation qui offre plusieurs gammes de services stratégiques pour différents espaces ambiants.

Marc sait que chaque objet tangible peut être facilement remplacé par un autre en modifiant physiquement l'adresse du nouvel objet. Après une première configuration par les techniciens, ces objets s'adaptent en permanence à l'espace ambiant. Le système peut alors évoluer tout en conservant cette configuration initiale en ajoutant ou soustrayant des fonctionnalités. L'ordinateur portable de Marc, équipé d'une carte réseau "ambiante" (capable de communiquer avec le système informatique ambiant), peut s'interfacer avec le `contexte` *pour le modifier et le contrôler...*

Dans ce scénario, un certain nombre de concepts clés de l'informatique ambiante ont été mis en valeur. Par exemple, il en ressort que l'installation du système se révèle fortement *modulaire* et la configuration des modules demeure simple. Les modules s'interfacent automatiquement par correspondance d'identifiants (uniques à chaque dispositif). L'association entre les dispositifs est en partie gérée par l'application logicielle, ce qui permet que l'installation reste simple. Nous y découvrons la capacité qu'a l'ensemble des dispositifs et des communications – que nous appellerons

`espace ambiant` – à se reconfigurer de manière optimale en faisant en sorte de *n'activer que les stratégies qui ont du sens* en fonction des dispositifs disponibles.

Un espace ambiant est un espace physique composé de dispositifs communiquant de manière invisible et qui sont introduits dans les objets de la vie de tous les jours ; ils entourent ainsi les utilisateurs de manière non-intrusive. Étant également capables de répondre à certaines indications gestuelles, sonores, etc., ils peuvent même engager un dialogue interactif avec un utilisateur ou entre machines. Au niveau des moyens de communication mis en œuvre, nous retrouvons aussi bien des communications filaires que des communications sans fil. Nous y voyons des systèmes informatiques ambiants figés aussi bien que des systèmes informatiques ambiants mobiles.

> *Enfin, la fonctionnalité principale est la combinaison de ces nombreux dispositifs collaborant pour réaliser dynamiquement de nouvelles applications. Il semble alors important d'analyser les nouveaux défis qu'apporte l'informatique ambiante.*

1.2.3 Particularité technologique

La technologie qui dérive de la vision de l'IAm que nous avons décrite ci-dessus est ambiante, c'est-à-dire invisible et intelligente. Par opposition à l'ordinateur de bureau avec lequel, une seule personne utilise consciemment un seul ordinateur, quand la technologie est ambiante, elle engage beaucoup plus de dispositifs. Ces dispositifs sont disponibles en tout lieu et sont intégrés à l'environnement physique et aux objets.

Lorsque la technologie est invisible, elle dissimule les dispositifs informatiques à l'intérieur des objets de tous les jours. Cela a pour effet de faire ressortir intuitivement leurs fonctions principales, contrairement aux ordinateurs de bureau qui sont toujours équipés d'un clavier, d'une souris et d'un écran qui, tout en leur permettant de rester génériques, masquent les fonctions qu'ils peuvent avoir.

La technologie dite intelligente se décompose en trois fonctions qui ont un rapport direct avec ses utilisateurs : elle est utile, discrète et enrichit le sens que l'on associe aux objets de tous les jours. La fonction *utilité* a pour but de s'assurer que les dispositifs, grâce à leur sensibilité au contexte, soient utiles aux utilisateurs. La fonction *discrétion* renforce la discrétion des dispositifs (qui se dit "non intrusive" en anglais), c'est-à-dire que les utilisateurs n'ont pas conscience d'utiliser un dispositif informatique, mais ont toujours l'impression de manipuler des objets de tous les jours. Nous avons l'habitude que les ordinateurs nous transmettent ou nous permettent de stocker du savoir et des informations. Enfin, la fonction *fournisseur de sens* est complètement différente de cette vision. Elle a pour but d'augmenter le sens que nous affectons aux objets de tous les jours. Elle ne nous transmet pas simplement des informations, mais ajoute de nouvelles réactions à l'environnement qui entoure les utilisateurs, des réactions synthétisées par les dispositifs informatiques.

1.3 Conséquences sur l'architecture des systèmes

Après le *mainframe* qui a concrétisé l'idée d'un ordinateur pour plusieurs utilisateurs dans les entreprises, le PC a concrétisé la volonté d'avoir un ordinateur pour un utilisateur chez soi ; nous entrons maintenant dans une troisième phase de l'informatique. Selon Mark Weiser, plusieurs utilisateurs utiliseront plusieurs ordinateurs à la fois enfouis dans les objets de la vie quotidienne comme le téléphone, le réfrigérateur, la machine à laver, la voiture, etc, une tendance illustrée dans la Figure 1.4. L'ordinateur devient alors implicite, invisible et omniprésent. L'utilisateur se retrouve dans un mode d'interaction inconsciente avec l'ordinateur et surtout avec les applications que ce dernier exécute. C'est à ces applications qu'incombe la tâche de prendre en compte ce mode d'interaction et de s'adapter à la volonté de l'utilisateur qui peut être inconnue à l'avance. Il peut être également possible d'accéder à ces applications en toute situation.

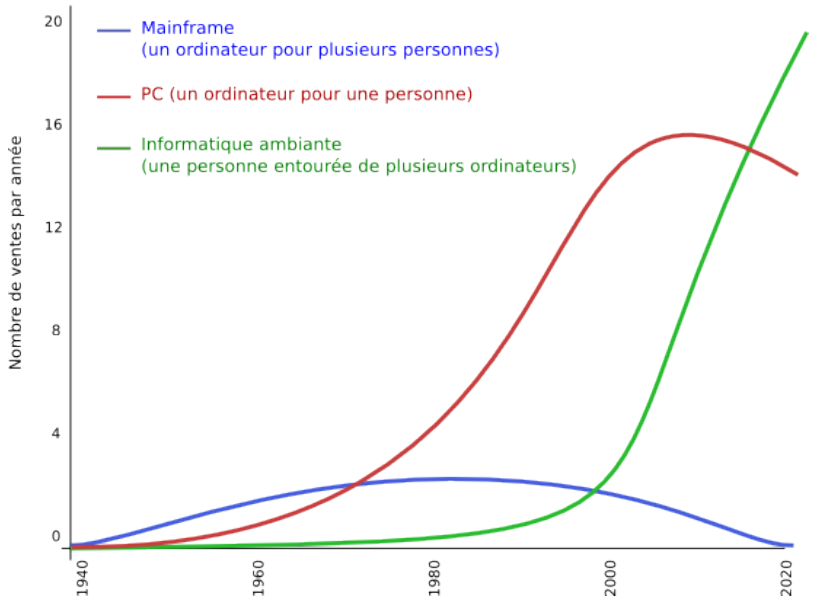


FIGURE 1.4 – Évolution des relations entre le système informatique et son environnement [1]

1.3.1 Les contraintes imposées par l'IAm

En règle général, un système informatique a un organe d'entrée, un organe de traitement de l'information et un organe de sortie. Les organes d'entrée sont le clavier, la souris, la voix, etc. Les organes de sortie sont principalement l'écran, l'imprimante, le graveur, etc. Ces systèmes utilisent, pour beaucoup encore, les concepts de Von Neumann [12].

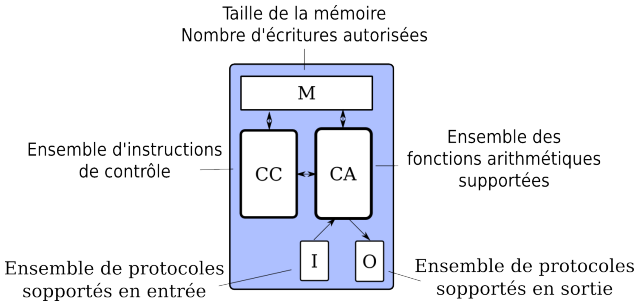


FIGURE 1.5 – Architecture Von Neumann des systèmes informatiques

L'architecture de Von Neumann décompose un système informatique en trois parties spécifiques de traitement et deux parties spécifiques sensorielles. Les parties spécifiques de traitement sont les suivantes :

1. L'unité de traitement arithmétique CA (opérations de base telles que l'addition, la soustraction, la multiplication, la division, etc.).
2. L'unité de contrôle logique CC (pour déterminer l'ordre correct dans lequel exécuter les opérations).
3. La mémoire M (pour supporter les séquences longues et compliquées d'opérations en fournissant un moyen de se souvenir des résultats intermédiaires).

Tous les transferts numériques (ou autres) d'information entre les parties C (CA et CC) et M du système informatique doivent être effectuées par des mécanismes contenus dans ces parties. Il reste alors la nécessité de transférer des informations de l'environnement vers le système informatique

et aussi de transférer l'information finale (les résultats) du système vers l'environnement. Par environnement, Von Neumann se réfère aux ressources telles que l'écran, l'imprimante, la souris, etc. L'environnement est ce qui peut être ressenti par nos cinq sens. Notons R ces ressources. Il nous reste donc deux parties spécifiques au système :

1. Les entrées (organes de transfert numérique, analogique, etc. d'informations de R vers ces parties spécifiques, C ou M).
2. Les sorties (organes de transfert d'informations de ces parties spécifiques C et M dans R).

L'intelligence ambiante est une refonte de l'interaction entre le système informatique et tout ce qu'il lui est extérieur. Comme nous l'avons vu à travers les exemples et les scénarios des sections précédentes, ce sont les objets de la vie de tous les jours qui détiennent le rôle d'organe de transfert d'information depuis et vers l'environnement.

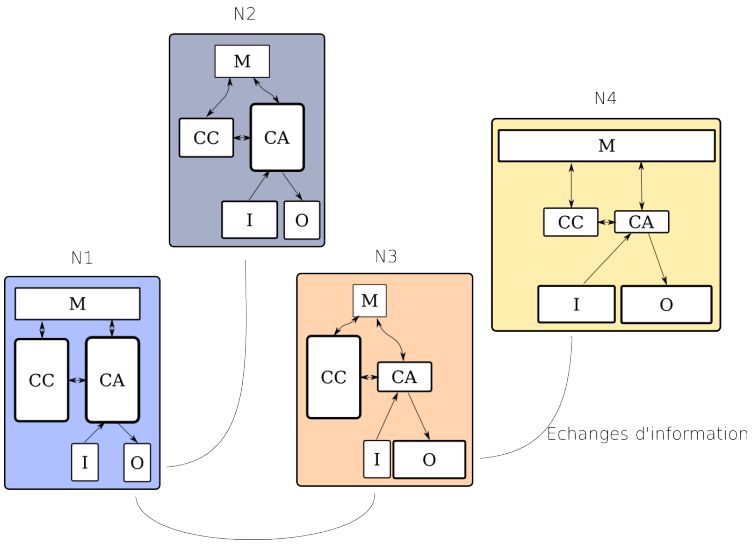


FIGURE 1.6 – Contraintes de l'IAm sur les systèmes informatiques

Système	**Mémoire**	**Capacité de calcul**	**Protocole**	**Exemple d'utilisation**
N1	Moyenne	Grande	Restreint	Contrôleur
N2	Petite	Grande	Standard	Conversion de signaux
N3	Petite	Petite	Standard	Capteur
N4	Grande	Petite	Complet	Stockage d'informations

Tableau 1.1 – Des systèmes ayant des capacités différentes

Le système découvre et utilise de nouvelles informations telles que la date et l'heure, la proximité d'utilisateurs, la proximité de dispositifs communicants, le niveau sonore ambiant au fur et à mesure de l'évolution du contexte. Ainsi, l'application appréhende et peut réagir aux situations telles que les pannes ou les ajouts et les retraits de `dispositifs communicants` de son contexte.
Les ressources de ces dispositifs sont généralement limitées, une limitation physique d'ordre spatial (espace mémoire et espace physique limités) et une limitation d'ordre énergétique (consommation restreinte). Les dispositifs font intervenir : des capteurs, des actionneurs et des services. Dans notre étude, ces dispositifs communicants sont des objets de la vie de tous les jours et réalisent cette interaction contexte-application. Ces sont des tasses (indiquant si elles sont chaudes ou froides), des balles (indiquant si elles sont en mouvement), des interrupteurs (fournissant leur état) ou des pots de fleurs (qui nous informent que les fleurs manquent d'eau).

Dans notre étude, il en ressort que la manière de communiquer avec ces dispositifs diffère de ce que nous connaissons déjà avec les PCs. Nous n'avons pas d'accès à des commandes évoluées comme des consoles de gestion afin d'instrumenter au minimum ces dispositifs. Les raisons de ces limitations sont très simples. Elles proviennent de l'économie d'énergie et de l'invisibilité des dispositifs. Nous pouvons nous poser les questions suivantes : doivent-ils être programmables et jusqu'à quel point ? Comment intégrer les dispositifs dans des applications déjà existantes ?

Nous proposons l'analyse de l'évolution de l'informatique ubiquitaire et tentons de formuler une réponse à ces questions à l'aide d'un modèle décrivant l'intégration des dispositifs communicants et l'adaptation logicielle résultante. Dans ce chapitre, nous positionnons la problématique de ces travaux par rapport aux idées d'adaptation logicielle issues de la littérature. Nous décrivons, à la fin du chapitre, le plan et le cadre de la thèse.

1.3.2 Les enjeux

À partir de l'analyse de la littérature du domaine, nous étudions les caractéristiques d'un système informatique ambiant (SIA). Nous proposons de dresser une liste des défis de l'informatique ambiante. L'informatique ambiante est constituée d'applications logicielles s'exécutant dans un espace peuplé de dispositifs communicants où chaque entité est intégrée dans un objet de la vie de tous les jours (tasse, chaise, etc.). Ces objets sont informatisés et sont capables de construire un `contexte` pour l'application en fournissant à leur environnement des informations sur leur état et également en agissant sur leur environnement. La détection et l'analyse des situations contribuant à cette perception sont deux tâches inconscientes que nous avons l'habitude d'effectuer tout au long de la journée. Cependant, lorsque ces tâches sont confiées à ces unités de calcul, le traitement devient nettement plus délicat. La complexité du monde réel est telle qu'il est difficile au niveau de l'unité de calcul d'identifier les sources d'information pertinentes.

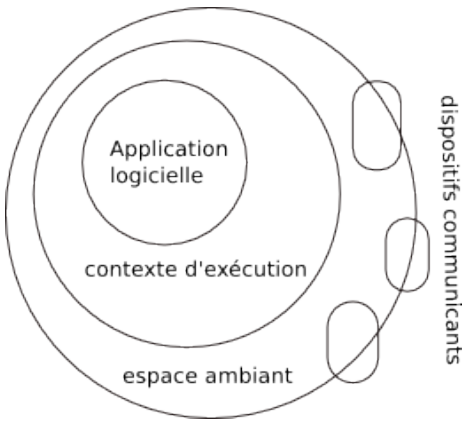


FIGURE 1.7 – Le problème de l'existence de l'application logicielle immergée dans l'environnement physique

Weiser [13] soulignait que les systèmes informatiques ambiants devaient être silencieux, calmes, et devenir des serviteurs invisibles, qu'ils devraient étendre notre inconscient en augmentant notre intuition. D'un autre côté, Adam Greenfield [6] perçoit l'informatique ambiante la manière dont nous subissons aujourd'hui les technologies numériques. Au lieu du lien binaire entre un service et un dispositif ce qui est caractéristique de l'ère du PC, l'utilisateur de l'*everyware* interagit simultanément avec un grand nombre de dispositifs. Ces dispositifs sont enfouis dans les objets de la vie quotidienne, d'une manière plus ou moins perceptible. Les idées de Weiser et de Greenfield se rejoignent dans le concept d'une informatique qui accompagne nos actions sur l'environnement.

Étant proche finalement du domaine de l'*autonomic computing* au service de l'utilisateur, nous ajoutons à cette démarche d'accompagnement de l'utilisateur, la collaboration des unités de calculs et leur autonomie dans leur environnement physique. Enfin, l'informatique ambiante n'est pas uniquement centrée sur l'utilisateur, mais prend en compte l'environnement avec, à la fois, ses grandeurs physiques et les unités de calcul.

L'environnement entourant un système informatique ambiant est constitué de machines qui forment globalement l'espace ambiant et plus précisément l'infrastructure de dispositifs de l'application logicielle (Figure 1.7). L'infrastructure de dispositifs est un concept dépendant du temps. Cela apporte de nouveaux enjeux aux logiciels sous la forme de cinq propriétés [2] :

1. Capacité d'anticipation des besoins selon le contenu de l'infrastructure de dispositifs,
2. Capacité d'adaptation des réponses du système par rapport à l'infrastructure de dispositifs,
3. Personnalisation du système selon des profils,
4. Découverte d'objets communicants,
5. Prise en compte d'informations tangibles.

Le premier enjeu permet à un système de prévoir un changement d'état du système logiciel et matériel et d'anticiper sur son mode de fonctionnement. Le deuxième enjeu se réfère à la manière de réagir aux modifications de l'infrastructure de dispositifs. La personnalisation consiste à fixer certains paramètres du système : c'est une des manières de prendre en compte l'utilisateur par exemple. La découverte d'objets communicants permet d'ajouter la caractéristique dynamique au système. Enfin, la prise en compte d'informations tangibles permet d'obtenir également des informations sur le comportement des utilisateurs.

La plupart des scénarios en informatique ambiante sont le résultat d'une collaboration entre plusieurs dispositifs. Ces dispositifs coopèrent et sont entourés d'entités communicantes qui évoluent. Nous analysons d'abord les systèmes dits figés et les systèmes évolutifs en s'appuyant sur la plate-forme conceptuelle développée par Mascolo et al. [14]. Nous étudions donc trois parties successives dans les trois paragraphes suivants : les types de dispositifs, la communication entre dispositifs et les types d'environnement d'exécution.

Types de dispositifs. Les systèmes figés partent des ordinateurs de bureau et des serveurs jusqu'aux systèmes électroniques à base de `microcontrôleurs` embarqués dans des objets tels que notre machine à laver. Rappelons qu'un microcontrôleur est un ordinateur simplifié et miniaturisé. Il est doté d'une unité de calcul (CPU), une mémoire contenant le programme à exécuter, une mémoire RAM (mémoire à accès aléatoire) pour stocker des variables temporaires et des dispositifs d'entrée-sortie (convertisseurs de signaux analogiques-numériques et entrées-sorties logiques). Les dispositifs mobiles varient des ordinateurs portables, PDAs, téléphones portables vers les systèmes électroniques embarqués dans des entités mobiles telles que les montres. Toutefois, à cause du caractère collaboratif de l'informatique ambiante, le terme "évolutif" englobe à la fois le découplage avec l'environnement physique et la collaboration avec les dispositifs de l'environnement. La mobilité physique devient facultative.

Communication. Au niveau de la communication entre les entités, dans le cas des systèmes figés, les ordinateurs sont connectés à travers des liens permanents. Ces liens physiques peuvent être à haut-débit et supporter potentiellement de la redondance. Ce n'est pas l'hypothèse d'un lien permanent qui nous intéresse en ce point, mais le fait que les applications logicielles s'attendent à utiliser ces moyens de communication pour échanger des informations. Un système figé peut également être un unique ordinateur contenant plusieurs applications communiquant entre elles.

Environnement d'exécution. Dans tous les cas, des déconnexions peuvent survenir à cause non pas de maintenance planifiée à l'avance ou de pannes, mais à cause des limitations des zones d'utilisation de dispositifs. Ces zones peuvent être définies par les technologies de communication utilisées souvent sans fil (GPRS, Bluetooth, Wifi, etc.). Enfin, un système informatique

ambiant est hétérogène et peut être constitué de dispositifs figés et de dispositifs évolutifs peuplant l'espace ambiant.

À partir de ces observations, nous pouvons conclure que les systèmes évolutifs sont connectés de manière intermittente, combinés avec des systèmes figés qui ont habituellement des connexions permanentes. Pour un système évolutif, l'établissement d'une connexion entre dispositifs dépend de l'espace dans lequel on se trouve. Mais, pour un système figé, la connexion est généralement permanente puisque les systèmes peuvent potentiellement offrir localement des fonctionnalités requises. Le logiciel d'un système évolutif est impacté par ces changements d'espace. Finalement, l'infrastructure des dispositifs relative aux applications adaptatives est changeant. Nous avons vu la notion de réseau évolutif qui incorpore à la fois la notion de coopération et de changement dans un espace où les dispositifs évolutifs sont combinés aux dispositifs figés. L'infrastructure de dispositifs est du même ordre que la disponibilité des dispositifs.
Notons qu'il existe une influence au niveau des dimensions sociales. Bien qu'importantes, ces dimensions ne concernent pas nos travaux. Le lecteur trouvera des travaux concernant ce domaine à travers ceux de Persson, Zimmerman, Tang et Keyani [15, 16, 17, 18, 19].
Enfin, avant d'aller plus loin dans l'exploration d'une architecture pour l'informatique ambiante, nous nous positionnons parmi les terminologies existantes dans ce domaine de recherche (Chapitre 2). Nous proposons alors une définition à la notion d'informatique ambiante (Chapitre 3). Nous précisons les problématiques auxquelles nous désirons apporter une solution dans le Tableau 1.2.

1	Enrichir les capacités sensibles, calculatoires et communicatives des objets de tous les jours.
2	Observer et modéliser le comportement des entités de l'environnement physique (qui comprend les utilisateurs, les phénomènes, etc.).
3	Déduire des observations les intentions et les intérêts des entités de l'environnement telles que les utilisateurs.
4	Concevoir des interfaces dynamiques qui interagissent judicieusement et de manière pertinente avec l'environnement physique.
5	Intégrer naturellement ces interfaces dans l'environnement physique.

Tableau 1.2 – Problèmes de l'informatique ambiante

Nous nous focalisons sur la problématique de l'adaptation logicielle pour l'informatique ambiante. Notre étude comporte deux parties. D'abord, nous analysons ce qui se passe au niveau matériel dans un environnement ambiant et leur impact sur la structure du logiciel. C'est le but de l'étude de la partie suivante. Ensuite, nous voyons plus en détail l'adaptation logicielle en découplant les notions d'adaptabilité et d'adaptativité afin de nous positionner précisément parmi les différents travaux de recherche dans le domaine.

1.4 Bilan

En conclusion de ce chapitre, nous présentons dans un premier temps l'orientation de notre recherche par rapport aux problématiques soulevées après l'analyse du domaine de l'informatique ambiante. Nous exposons ensuite le plan du mémoire de thèse en résumant les différentes étapes de notre analyse. Enfin, nous présentons le cadre dans lequel nous avons effectué ces travaux.

1.4.1 Objectifs de la thèse

Les travaux actuellement réalisés proposent de traiter la complexité de la construction des applications informatiques ambiantes adaptatives, en abstrayant, par des composants logiciels, les ressources et les services fournis par les dispositifs communicants. À partir du principe d'intégration de nouvelles fonctionnalités apportées par les dispositifs communicants, de nombreuses propositions

ont été effectuées sur des modèles à base de composants, des modèles orientés services [20] et à partir des approches par aspects [21]. Des cas problématiques apparaissent cependant comme celui de l'intégration en continu de nouvelles fonctionnalités au sein d'une application, de leurs interférences [22] et des conflits liés au partage des ressources. Ces problèmes sont complémentaires à celui de l'adaptation sémantique des interfaces [23] et celui de la séparation des préoccupations [24] (dont le but est toujours de diviser un logiciel en parties plus petites pour en maîtriser la complexité).

Composants. Les modèles à base de composants [25, 26] consistent à utiliser une approche modulaire au niveau de l'architecture, ce qui permet d'assurer au logiciel une meilleure lisibilité et une meilleure maintenance. Dans notre cas, elle permet un découpage adéquat pour rendre une application adaptable. Au lieu de créer un exécutable monolithique, ces modèles fournissent un ensemble de briques logicielles réutilisables qu'il s'agit ensuite d'assembler. Ce modèle n'est pas sans similitude avec le modèle orienté objet, puisqu'il peut revenir, dans certains cas, à s'appuyer sur l'approche objet au niveau de l'architecture générale et de son implémentation. Par exemple, plusieurs objets peuvent contribuer à l'élaboration du concept de composant logiciel [27].

Services et composition. Une architecture orientée service [28] forme une architecture de médiation qui met en œuvre des services avec une forte cohérence interne (dont la nouveauté se trouve dans l'échange de documents et l'utilisation d'un format d'échange pivot, comme XML) et des faibles couplages externes et des échanges asynchrones (par l'utilisation d'une couche d'interface interopérable). Par exemple, un service WEB est un modèle d'architecture orientée service. Le service est une action exécutée par un producteur à l'attention d'un consommateur, cependant l'interaction entre consommateur et producteur est faite par le biais d'un médiateur (qui peut être un bus) responsable de la mise en relation des composants. Ces systèmes peuvent aussi être définis comme des couches applicatives.

Aspects. Dans le développement traditionnel par aspects [21], il est question d'une modularité s'exprimant à travers la séparation des préoccupations transverses [1] dans des domaines de recherche variés [29]. Le code logiciel peut être écrit de manière indépendante en s'appuyant sur de nouvelles entités appelées aspects. Une séparation modulaire classique consiste à écrire le code "fonctionnel" (métier) indépendamment du code "non-fonctionnel" qui est, quant à lui, regroupé dans des aspects. Ces codes peuvent donc être maintenus et améliorés de manière découplée. Ensuite, les aspects sont tissés au code fonctionnel pour former l'application complète. À la base du développement par aspects repose donc l'identification du phénomène d'entrelacement de code [2].

L'approche par aspect ne se limite cependant pas aux seules préoccupations non-fonctionnelles (services dits techniques). En effet, toute fonctionnalité transverse peut être modularisée par un aspect. Par définition, l'approche par aspect considère que toute décomposition conduit, à un moment donné, à ces phénomènes d'entrelacement de code. Elle fournit alors des méthodes pour une modularisation explicite.

Interférence. Le phénomène d'interférence (conflit entre plusieurs spécifications) se retrouve dans, au moins, deux modèles : celui des aspects appelé d'*aspect interactions* [22] et celui des modèles réactifs [30] sous la forme d'interactions logicielles.

Notre objectif. Dans les architectures à composants, on a toujours peu de mécanismes de gestion de superpositions d'interactions. Ces mécanismes se trouvent souvent mêlés à l'assemblage ce qui complique la maintenance, l'évolution et la réutilisation du code. De plus, la complexité de l'assemblage est augmentée à chaque nouvel ajout fonctionnel. Ce couplage entre fonctionnalités intrinsèques et interactions inhibe alors fortement les concepts de base des modèles de composant. L'objectif de notre travail est l'obtention d'un mécanisme d'intégration dynamique de nouvelles

1. qui est difficilement exprimable de façon modulaire
2. dispersion et duplication d'un code à plusieurs endroits de l'application

fonctionnalités en réponse aux variations du contenu de l'infrastructure de dispositifs (partie *capture informatique* des informations de l'environnement physique). Nous proposons une approche logicielle qui résout le traitement, à posteriori, des problèmes de superposition de fonctionnalités en s'appuyant sur une base de règles logiques. Les contributions principales de ce travail peuvent être résumées en trois points-clés :

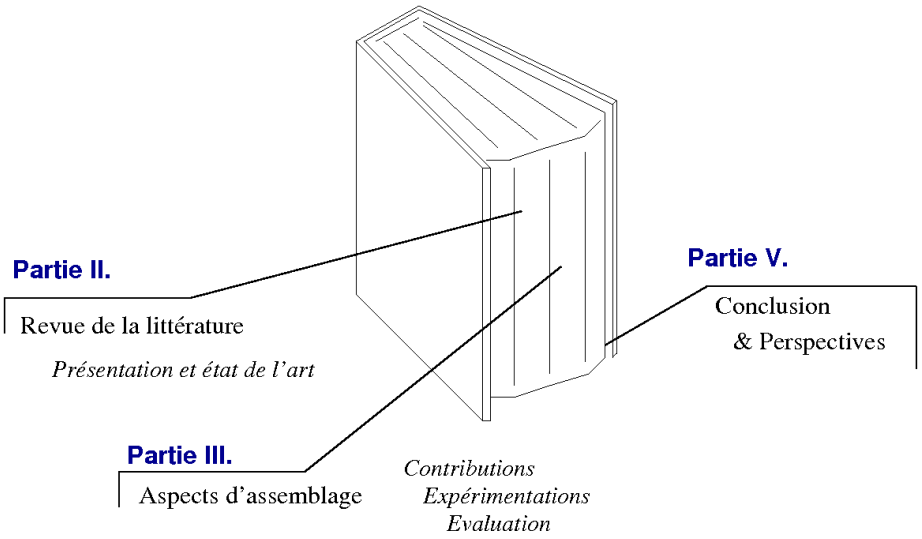
1. Un modèle de service composite ;
2. La notion d'aspect d'assemblage ;
3. Un mécanisme de tissage d'aspects d'assemblage à partir de règles logiques ;

Notre démarche est la suivante. Nous exposons d'abord les limites des modèles existants et présentons ensuite notre approche en précisant les fondements de notre modèle. Puis, nous débattons des mécanismes d'adaptation.

1.4.2 Points-clés

Dans cette section, nous présentons les différents chapitres. Nous étudions et proposons un modèle pour l'adaptation logicielle dynamique dans les environnements des systèmes informatiques ambiants. Cette thèse vise à résoudre la difficulté liée à l'intégration de plusieurs fonctionnalités sous la forme d'assemblages de composants en fonction de l'infrastructure de dispositifs. Cette recherche nous conduit à définir un modèle d'adaptation d'applications afin de permettre l'ajout et la suppression dynamique des adaptations. Cependant, nous souhaitons que notre modèle soit suffisamment générique pour qu'il puisse être facilement implémenté sur différentes plates-formes. Ainsi, ce mémoire est organisé en trois parties :

- État de l'art des architectures pour l'informatique ambiante
- Aspects d'assemblage (Principe, Mise en œuvre, Expérimentation et Évaluation)
- Conclusion et perspectives



Partie II. état de l'art

Beaucoup de travaux proposent des modèles dont le but est de simplifier la programmation des systèmes informatiques ambiants et des systèmes informatiques en général. Cette partie passe en revue quatre domaines que sont les composants, les services, les aspects et la gestion des interférences.

> **Chapitre 2.** *Composants, Services, Aspect et Interférences.* Ce chapitre regroupe ces quatres domaines et présente les principales approches existantes. Nous analysons leur manière de

prendre en compte le contexte et la manière de simplifier la programmation pour l'adaptation. Nous dressons un bilan des apports originaux de chaque travail de recherche (portant sur le découplage, la généricité, l'automatisation, l'unification, etc.) et situons nos travaux sur un sujet encore peu exploré : l'adaptation par superposition d'assemblages de composants logiciels.

Chapitre 3. *Synthèse et objectif.* Dans ce chapitre, nous résumons les techniques utiles de l'état de l'art et les apports des travaux présentés. Nous situons ensuite nos travaux sur l'adaptation par superposition d'assemblages de composants logiciels.

Partie III. Un modèle de composition pour l'adaptation : aspects d'assemblage

L'objectif de cette partie est la définition d'un modèle d'Aspects d'Assemblage (AA) offrant une programmation simple de l'adaptation. Ce modèle comprend deux sous parties : un modèle d'assemblage et de composant et un modèle d'adaptation. Y sont décrits la mise en œuvre de l'architecture que nous avons proposé pour la gestion des aspects d'assemblage et leur évaluation. Nous montrons comment les AAs s'intègrent dans le modèle de composant.

Chapitre 4. *Modèle de service, composant et assemblage.* Ce chapitre expose les principes de base de l'abstraction que nous proposons pour l'adaptation logicielle par modifications structurelles de l'assemblage : les *services composites*.

Chapitre 5. *Aspects pour l'assemblage de composants.* Nous présentons, de manière incrémentale, la notion d'aspect d'assemblage en partant de la définition des points de coupe et les greffons. Nous voyons le tissage et présentons notre manière d'ordonnancer les greffons d'adaptation. Nous proposons une première extension du tissage qui consiste à ordonnancer en s'appuyant sur des règles logiques mathématiques. La seconde extension permet de fournir une approche qui prend en compte des conflits plus spécifiques.

Chapitre 6. *Expérimentations.* Ce chapitre montre, dans un premier temps, les expérimentations effectuées sur l'implémentation des services composites. Nous présentons les réalisations effectuées avec la plate-forme Ubiquarium. Dans un second temps, nous présentons les expérimentations effectuées sur des plates-formes embarquées utilisant une maquette électronique reproduisant un studio appareillé d'équipements communicants. Nous décrivons enfin les caractéristiques d'une expérimentation en grandeur réelle dans le laboratoire GERHOME.

Chapitre 7. *Évaluation.* Dans ce chapitre, nous proposons un modèle des performances et de l'impact mémoire de notre système adaptatif. Ce modèle s'appuie sur les algorithmes décrits dans les chapitres 4 et 5. Nous montrons que les performances de notre solution pour des cas d'utilisation concrets sont optimales pour une utilisation dans le cadre attendu. Enfin, nous montrons comment utiliser les résultats de ces travaux pour une extrapolation à d'autres domaines.

Partie IV. Conclusion et perspectives

Le dernier chapitre présente le bilan de notre approche et les perspectives des aspects d'assemblage à travers divers domaines d'application.

Chapitre 8. *Conclusion et perspectives.* Dans ce chapitre, nous fournissons des éléments de réponse aux nouveaux questionnements issus de notre approche, en identifiant tout d'abord les apports essentiels de notre travail et de son modèle sous-jacent, ainsi que leurs limites. Nous affinons et généralisons par ailleurs notre approche, sur la base des enseignements que nous avons pu déduire de la validation des concepts de la plate-forme. Nous y exposons également les perspectives de nos travaux.

1.4.3 Contexte de recherche

Cette thèse s'est déroulée dans l'équipe Rainbow du laboratoire I3S [3] dans le cadre d'un contrat de collaboration entre le Centre National de la Recherche Scientifique (CNRS) et le Centre Scientifique et Technique du Bâtiment (CSTB) (contrat de recherche CNRS n°04101).

L'équipe RAINBOW du laboratoire I3S (CNRS - UNSA) est une des équipes de la thématique "génie du logiciel et de la connaissance" du laboratoire I3S. Son objet de recherche est le génie logiciel avec pour objectif de minimiser les sources d'erreurs. En effet, les logiciels deviennent de plus en plus complexes :

- par leur taille (des millions de lignes de code)
- par leur combinatoire (génération d'un grand nombre d'événements simultanés)
- par leur architecture (adaptation, hétérogénéité, contexte [4])

Les services qu'ils doivent rendre sont multiples et, bien souvent, pas entièrement connus au moment de leur conception. Le logiciel doit alors être capable d'évoluer et de s'adapter aux conditions changeantes de son exécution. Il doit aussi découvrir son environnement et y associer les dispositifs intéressants. La plupart des logiciels sont aujourd'hui construits par assemblage. Sans être une révolution, l'assemblage vise à mieux réutiliser ce qui a déjà été produit et à mieux contrôler l'ensemble du cycle de développement des logiciels. Ainsi, le monde du génie logiciel rejoint peu à peu celui de la connaissance par sa structure, la manipulation et le partage de ces descriptions.

Le projet WComp de l'équipe Rainbow recouvre les aspects liés à la maîtrise de la complexité logicielle, l'adaptation dynamique logicielle, en particulier en fonction de l'évolution de son environnement d'exécution. Cette recherche a pour ambition d'aboutir à une plate-forme d'étude pour l'intégration de composants dans le domaine de l'informatique ambiante et de manière plus générale, en *autonomic computing*.

Collaboration I3S - CSTB. Le CNRS et le CSTB ont financé cette thèse au travers d'une bourse docteur-ingénieur (BDI). Le CSTB est placé sous la tutelle du ministère du logement, direction générale de l'urbanisme, de l'habitat et de la construction. Il a pour finalité l'amélioration du bien-être et de la sécurité dans les bâtiments et agit dans quatre domaines complémentaires : la recherche, la consultance, l'évaluation et la diffusion du savoir. C'est dans son département TIDS [5], qu'est étudiée l'application des technologies de l'information et de la communication à des nouveaux usages et pratiques de partage d'informations dans le secteur de la construction. Un des services de ce département s'ouvre sur un nouveau champ d'étude, les Objects Communicants dans le cadre du Bâtiment Intelligent (OCBI). Ce domaine s'intéresse aux problématiques techniques liées à l'utilisation d'objets enfouis dans le bâtiment (capteurs, actionneurs, automatismes,...) communiquant entre eux dans le but d'apporter des services de plus haut niveau. Les travaux qui sont menés dans ce cadre concernent aussi bien les liens logiciel-matériel, que les préoccupations au niveau logiciel et architecture des systèmes. Nous contribuons à ce programme de recherche en abordant notamment les préoccupations d'adaptation logicielle en informatique ambiante dans le cadre du bâtiment intelligent. L'OCBI apporte ainsi un cadre d'expérimentation et d'application des travaux d'adaptation dynamique et de maîtrise de la complexité logicielle à l'ensemble des travaux que nous avons effectués pendant cette thèse. Des expérimentations ont été effectuées dans le cadre de l'infrastructure du laboratoire GERHOME.

Cadres d'expérimentation supplémentaires. Nous avons également réalisé des expérimentations dans l'Ubiquarium informatique au laboratoire I3S. L'Ubiquarium met en œuvre une architecture orientée service à la fois réelle et simulée dans un espace ambiant tel qu'au domicile, au

3. Informatique, Signaux et Systèmes de Sophia-Antipolis

4. par exemple : exécution de processus multiples mobiles, appartenant à des institutions indépendantes, interconnectées de manière ad hoc

5. Technologie de l'Information et Diffusion du Savoir

bureau, à l'hôpital tel qu'il a été fait dans le projet RNTS Ergodyn. Les résultats de cette thèse ont été intégrés à la plate-forme WComp, le principal moteur logiciel de l'Ubiquarium.

Deuxième partie

État de l'art

Chapitre 2

Composants, Services et Aspects

Sommaire

Afin de justifier les différents besoins en termes d'adaptation dynamique des applications, nous nous appuyons sur l'exemple décrit dans la première partie (page 13). Nous rappelons, de manière synthétique, les caractéristiques-clé de l'informatique ambiante :

- Forte modularité
- Mécanisme d'association et de découverte d'entités matérielles et prise en compte au niveau logiciel
- Combinaison et collaboration des fonctionnalités existantes pour construire des fonctionnalités plus complexes

En effet, nous considérons l'application logicielle comme étant constituée d'entités indépendantes assemblées pour réaliser les fonctionnalités attendues, c'est-à-dire une approche à base de composants logiciels.
Analyse de l'adaptation : nous allons maintenant analyser l'adaptation logicielle d'assemblages de composants, en d'autres termes ce qu'est la décomposition logicielle simple et comment elle est améliorée en utilisant les `aspects`. Nous verrons la notion de *service* que l'on confrontera à celle de *composant* pour faire face à la problématique de la distribution. Nous nous intéresserons à la gestion des interférences entre différentes adaptations pour en extraire une synthèse des propriétés.
Dans une première partie, nous analysons ces caractéristiques que nous mettons en regard avec l'adaptation d'assemblages d'entités logicielles à leur environnement (2.1). Les autres parties s'intéresseront aux plates-formes pour l'adaptation (2.2), à la modélisation à base de composants logiciels (2.3), aux plates-formes à services (2.4) et aux plates-formes orientées aspect (2.5).

2.1 Analyse de l'adaptation des assemblages

Nous étudions d'abord comment décrire les entités logicielles qui seront ultérieurement assemblées. Cela nous amène ensuite à poser la question de la recherche de services composites pour effectuer ces assemblages. Ces services composites peuvent être combinés pour fournir des fonctionnalités plus complexes. Enfin, la complexité de l'assemblage de composants et des services peut être résolue par l'emploi des aspects qui apportent une notion de modularité avancée, dite, *transverse* (par duplication et dispersion du code).

2.1.1 Description d'entités logicielles ultérieurement assemblées

Une première description du logiciel peut se faire à l'aide d'un `modèle orienté objet` [31] qui introduit un petit nombre de concepts (objet, classe, méthode, attributs, etc.). L'intérêt de ce modèle réside d'abord dans sa modularité, c'est-à-dire que la structure et les méthodes d'accès sont regroupées dans une seule entité logique : l'objet. Ensuite, il fournit la notion d'encapsulation, c'est-à-dire que le programmeur d'une classe peut n'avoir accès qu'à l'interface des objets. Enfin, le principe de réutilisabilité se situe au niveau de la classe qui n'est pas créée pour une application particulière, mais qui peut être utilisée pour d'autres applications.

Bien que cette vision *objet* ait été un pas en avant en ingénierie logicielle et en modélisation, elle n'a cependant pas réussi à apporter une solution bien établie sur la manière d'assembler ultérieurement les entités logicielles. Une des réponses à cette question est fondée sur la notion d'architecture [32]. Une partie de la communauté scientifique considère qu'il s'agit d'une tentative d'aller au-delà des objets [32] en fournissant les modèles et les outils nécessaires pour capturer, maîtriser et manipuler l'architecture d'un système logiciel. Dans tous les cas, cette problématique d'assemblage [33] se trouve au cœur de nos travaux. Nous présentons deux moyens de structurer des entités logicielles qui peuvent être ultérieurement assemblées. La première est la modélisation par une architecture à composants logiciels [32] permettant de rendre le logiciel modulaire. Et la seconde est la modélisation par une architecture orientée service (SOA) [34] se focalisant sur les services rendus par une partie du logiciel.

2.1.1.1 Architectures à composants logiciels

Les architectures à composants logiciels [32] introduisent la notion d'assemblage comme technique de réutilisation. La plupart des plates-formes à composants facilite l'évolution statique ou dynamique d'une application construite à partir d'assemblages de composants par la possibilité d'ajout, de retrait ou de remplacement de composants. La définition ci-dessous faite par Szyperski [26] décrit une vision abstraite des composants logiciels en soulignant les propriétés qui font d'eux les briques de bases d'un système logiciel conçu par assemblage.

> "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.*"
>
> "Un `composant logiciel` est une unité de composition ayant des interfaces spécifiées de façon contractuelle et possédant uniquement des dépendances de contexte explicites. Un composant peut être déployé de manière indépendante et est sujet à composition par des tiers."

Un modèle de composant définit la structure de l'assemblage, la structure des composants et la structure des connecteurs. Il spécifie également la méthode de déploiement.

2.1.1.2 Architectures orientées service

Une `architecture orientée service` [34] est, quant à elle, fondée sur la définition de services coopérants et non pas nécessairement sur la modularisation du code logiciel. Elle définit de façon contractuelle le comportement d'une application.

Service. Un service définit une fonctionnalité réutilisable dont le comportement est décrit par un contrat d'utilisation dans un descripteur de service. Un descripteur de service inclut des informations sur l'interface de service (syntaxe, comportement).

Fournisseur de service. Un fournisseur de service implémente la fonctionnalité définie par un service. Il est découvert à l'exécution de l'application par une entité intermédiaire.

Un service correspond à une fonctionnalité dont le comportement est défini par un contrat d'utilisation et est fourni par une entité nommée `fournisseur de service`. Les fournisseurs de service sont découverts à l'exécution de l'application à l'aide d'une entité intermédiaire (section 2.1.2.2). Ainsi, une application construite à partir de services utilise un ensemble de services. Cette approche, au même titre que l'approche à composants, promeut la réutilisation de code, car un même service peut être utilisé dans différentes applications :

- Forte cohérence interne (besoin d'un format d'échange),
- Couplages externes lâches (couche d'interface interopérable).

Le service est une action exécutée par un `fournisseur` à l'attention d'un `consommateur`, cependant l'interaction entre consommateur et producteur est faite par le biais d'un médiateur, comme nous l'avons déjà indiqué, responsable de la mise en relation des services. Ces services peuvent aussi être définis comme des couches d'une application. Une caractéristique fondamentale de l'approche à services est que l'assemblage d'une application est alors réalisé à partir des descripteurs de services.

La découverte et la liaison avec des fournisseurs de services n'ont lieu que de façon tardive, c'est-à-dire avant ou pendant l'exécution de l'application. En conséquence, l'approche à service se focalise sur les notions de description et d'organisation, de façon à supporter la découverte dynamique des fournisseurs à l'exécution. L'architecture orientée service est une réponse aux problématiques de réutilisabilité, d'interopérabilité et de réduction de couplage entre les différents systèmes qui implémentent leurs systèmes d'information.

2.1.2 Comment assembler des entités logicielles (services, composants) ?

Nous voyons dans ce paragraphe comment s'assemblent des entités logicielles de types service et composant. Puis, nous voyons l'amélioration qu'apportent les modularités logicielles transverses.

2.1.2.1 Assemblage de composants

Nous distinguons deux types d'assemblage de composants [35] :
- l'assemblage horizontal (connexion des entrées sorties)
- l'assemblage vertical (encapsulation des fonctionnalités).

Assemblage horizontal Nous appelons assemblage horizontal, un assemblage qui met en relation les entrées-sorties des composants logiciels.

`Connecteur`. "*Je sais avec qui je communique.*" Un connecteur est une liaison explicite, une entité qui représente les interactions entre composants logiciels [36]. Un composant appelle directement un autre composant à travers un connecteur qui permet de spécifier le type d'interaction qu'il peut y avoir entre les composants (exemple : appel distant, local, ...). Un connecteur est basé sur l'utilisation de langages décrivant des relations entre des concepts propres à la composition. Bien que ces langages soient proches des langages de description d'architecture ADL [37], ils se différencient de ces derniers par le fait que le résultat de la composition est toujours exécutable. Des exemples de langages déclaratifs sont décrits dans les travaux de Birngruber et Wuyts [38, 39].

`Appel de service` : "*Je ne sais pas avec qui je communique.*" Un appel de service est une liaison implicite. Une application construite à partir de services implique que les fournisseurs de ces services ne sont pas câblés dans l'application : un service n'appelle jamais directement un autre service et les fournisseurs peuvent changer à travers les différentes exécutions de celle-ci. De nouveaux services peuvent être découverts à l'exécution. Les langages de script, tels que CorbaScript [40] ont tendance à être non-typés et interprétés [41].

Assemblage vertical Nous appelons assemblage vertical, un assemblage qui encapsule les composants logiciels pour en créer un nouveau. L'*assemblage hiérarchique* utilise l'encapsulation pour construire des composants en imbriquant des composants à l'intérieur d'autres composants dits composites. L'intérêt d'assembler les composants de manière hiérarchique réside dans la diminution de la complexité de l'architecture grâce au groupement des composants réalisant des tâches communes. L'assemblage hiérarchique nécessite de réaliser la connexion d'éléments présents dans les vues externes et internes de la description [42].

2.1.2.2 Composition de services

Les services peuvent être composés pour former un nouveau service. Des exemples de composition sont par exemple l'`orchestration` ou les `gestionnaires de workflow`.

Une `orchestration` est un ensemble de mécanismes conçus pour construire un nouveau service à partir de services disponibles. Une orchestration comprend des variables qui permettent de partager des informations entre les invocations de services. Les notations, comme celles répertoriées par List [43] associées aux orchestrations, visent à favoriser le maintien d'un faible couplage et un contrôle plus précis de la composition. Ainsi, sans orchestration, la composition de services est réduite à la résolution par la plate-forme des références vers les autres services (donc de manière cachée).

Un `gestionnaire de workflow` sert à décrire et à contrôler des applications à grande échelle demandant le calcul d'ensemble d'informations qui peuvent être manipulées indépendamment à partir d'une seule analyse. Il existe un large potentiel de parallélisme à exploiter pour rendre ces applications performantes [44].

La recherche de service Lorsque l'on ne connaît pas l'entité avec laquelle on communique, notamment dans le cas de l'`appel de service`, on a recours à un système de recherche de service appelé `annuaire`. Un annuaire référence l'ensemble des entités disponibles dans une architecture. Il participe ainsi activement à la mise en œuvre d'une cartographie dynamique des entités logicielles. Il est un intermédiaire entre les fournisseurs et les consommateurs de services. L'annuaire contient

un ensemble de descripteurs de service ainsi que des références vers les fournisseurs de ces services. Il fournit également des mécanismes permettant de l'interroger pour obtenir des références vers les fournisseurs de services. L'ensemble des descripteurs de service (se trouvant dans l'annuaire) peut changer de manière constante.

Au niveau utilisation, le consommateur de services interroge le registre pour découvrir des fournisseurs d'un service particulier à partir d'un ensemble de critères relatifs au descripteur. Si des fournisseurs répondant aux critères ont été publiés au préalable dans l'annuaire, ce dernier renvoie des références du fournisseur au demandeur. Le demandeur doit ensuite choisir des fournisseurs appropriés et effectuer ses appels au service. Nous avons répertorié quatre manières d'utiliser la fonction de recherche d'un annuaire :

- Recherche à priori : recherche effectuée lors de la conception dans un assemblage statique ;
- Recherche dynamique : celui qui réalise le service (celui qui l'implémente et le fournit) est connu au moment de l'exécution ;
- Recherche par type d'interface : "*Je recherche un capteur qui a l'interface Présence et qui se trouve dans la cuisine.*" ;
- Recherche par besoin : "*Je veux ouvrir les volets électriques du salon !*"

2.1.2.3 Modularités transverses

Les composants et les services proposent les techniques d'assemblage de composants et de composition de services pour créer l'architecture de l'application. La `programmation orientée aspect` peut être vue comme une approche orthogonale à l'assemblage. La programmation orientée aspect a été proposée par Kiczales *et al.* en 1997 [21]. Les paradigmes de programmation habituels reposent sur le fait que les différentes préoccupations de l'application puissent se décomposer en une collection d'abstractions (procédures, fonctions, objets, composants, classes, etc.). Cependant certaines préoccupations `transverses`, ne peuvent pas respecter cette décomposition et se retrouvent éparpillées entre plusieurs abstractions. Un exemple simple de préoccupation transverse est la mise au point des programmes : de nombreux ajouts sont nécessaires pour tester des assertions ou tracer une valeur. Ils se retrouvent entrelacés dans tout le code. De manière non exhaustive, la gestion de la sécurité, la mise en œuvre de transactions et la gestion de la distribution (synchronisation des accès concurrents) sont des préoccupations transverses.

La programmation par aspects se base sur l'idée qu'une application est mieux décrite en séparant toutes les préoccupations et propose de représenter de manière séparée ces préoccupations transverses dans les abstractions appelées `aspects`. De façon plus générale, l'expressivité du programmeur est parfois contrainte par le paradigme de programmation utilisé. Celui-ci choisit une manière de décomposer l'application. Inévitablement certaines préoccupations ne respectent malheureusement pas cette décomposition au sein du paradigme. Ce problème de la "tyrannie de la décomposition dominante" se produit avec tous les paradigmes de programmation. Même si la majorité des travaux sur l'AOP est appliquée à la programmation orientée objet, différentes recherches ont montré que ce paradigme pouvait aussi être appliqué à la programmation impérative [45], fonctionnelle [46] et réactive [47].

La programmation par aspects repose sur un `langage d'aspects` conçu pour décrire les préoccupations transverses et un `tisseur` qui intègre automatiquement les aspects au programme de base. De plus, les aspects permettent également de modifier le comportement de l'application de base et sont décrits dans des langages d'aspects qui décrivent où et comment modifier le programme et son exécution. Comme les langages d'aspects sont généralement basés sur des transformations syntaxiques, une des difficultés de la programmation par aspects réside au niveau du raisonnement sur l'application sans visualiser le code tissé (le code obtenu après tissage). De plus, l'application tissée peut avoir des comportements non souhaités à cause de potentielles interactions néfastes entre certains aspects. Un des défis majeurs de la programmation par aspects est donc de permettre un raisonnement modulaire, c'est-à-dire *raisonner sur l'application de base et les aspects sans étudier le code tissé.*

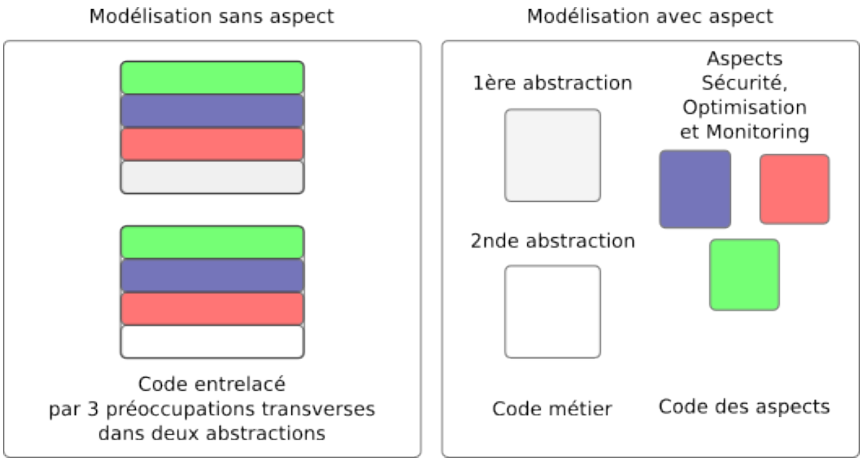


FIGURE 2.1 – Éviter l'éparpillement du code

Dès les années 80, la programmation structurée et la programmation orientée-objet ont introduit de nouvelles approches pour concevoir un programme et un ensemble de règles et de conventions pour aider les programmeurs à produire du code plus lisible et plus facilement réutilisable à l'aide par exemple de l'éradication des *gotos* au profit des boucles [48], l'introduction des *types* et des *structurations* à partir de composants logiciels [49]. Dans cette même direction, et 30 années plus tard, l'*aspect* résout des problèmes qui peuvent, bien sûr, être traités dans des approches classiques, mais d'une manière très élégante.

2.1.3 Synthèse des propriétés étudiées

A partir de cette étude préliminaire, nous pouvons déduire qu'il est important de rendre indépendantes les adaptations par rapport à l'application : c'est la propriété permettant aux programmeurs de spécifier des adaptations sans connaître, de manière spécifique, les applications qui seront déployées. Nous étudions les propriétés d'indépendance des adaptations, celles de la composition et celles de la gestion de conflits. Nous analysons de manière conceptuelle les apports originaux des plates-formes d'adaptation logicielle. Ces apports proviennent de domaines d'étude "orthogonaux" (adaptation, service, composant et aspect). Nous allons alors étudier ces domaines en commençant par les plates-formes pour l'adaptation.

2.2 Plates-formes pour l'adaptation dynamique

Dans cette section, nous étudions les `intergiciels` pour les systèmes informatiques ambiants. Ces systèmes *multi-dispositifs* sont intrinsèquement modulaires et cette modularité matérielle se retrouve au niveau du logiciel. Comme nous l'avons vu précédemment, la modification du contexte opérationnel motive une architecture adaptative, et nous allons voir comment cette modularité évolue et se décompose en plusieurs concepts. Nous analysons les intergiciels existants qui se situent entre l'application et l'infrastructure matérielle. Nous étudions leur adéquation avec les caractéristiques principales de l'informatique ambiante telles que l'implication qu'a la modularité de l'infrastructure sur le logiciel et son adaptation.

Le but de cette section est donc de montrer que la modularité logicielle se retrouve de manière générale imbriquée et dispersée dans des assemblages existants. Nous passons en revue les systèmes informatiques ambiants dotés de la capacité d'adaptation et analysons ensuite la caractéristique principale qui ressort de ces travaux. L'informatique ambiante encourage la prolifération des dispositifs embarqués spécialisés. Un des principaux aspects de l'informatique ambiante devient son invisibilité. Les utilisateurs perçoivent les fonctionnalités mais ne voient pas les dispositifs qui offrent ces fonctionnalités. La capacité d'adaptation et d'évolution du logiciel au sein de ces dispositifs devient donc un atout pour leur condition d'utilisation. Il existe un certain nombre de systèmes informatiques ambiants disponibles. Dans la suite, nous nous intéressons aux principales approches permettant l'adaptation dynamique. Elles ont été choisies pour leurs différences en terme d'expression des adaptations et de gestion du changement du contexte. Nous voyons ensuite d'autres approches apportant des notions importantes en adaptation dynamique.

2.2.1 *Aura* : minimisation de l'intervention de l'utilisateur

La plate-forme Aura de Garlan et Soussa [50, 51] s'appuie sur l'hypothèse que la ressource la plus précieuse en informatique n'est plus son processeur, ni sa mémoire, ni la taille de son disque ni même le réseau, c'est l'utilisateur. Aura est un intergiciel sensible au contexte qui permet de concevoir des applications mobiles. Le but d'Aura est de fournir à chaque utilisateur un ensemble de `services` implicites de traitement et d'information qui s'étend aux PDAs et aux ordinateurs de bureau. Ces travaux sont alors centrés sur l'utilisateur. Lorsque l'utilisateur se déplace d'une localisation géographique à un autre, et lorsque les ressources s'en vont et reviennent, le système s'adapte et offre à l'utilisateur un accès optimal, auto-ajusté et continu aux données et aux traitements.

Un des scénarios d'utilisation d'Aura [50] est celui de l'assistance à une utilisatrice, appelée Jane, qui se trouve à la porte 23 de l'aéroport de Pittsburgh au nord-est des état-Unis, attendant une correspondance. Elle écrit plusieurs documents assez volumineux et voudrait les envoyer par email en utilisant la connexion sans fil. Malheureusement, la bande passante est limitée parce que la plupart des passagers des portes environnantes surfent sur Internet. La plate-forme Aura observe qu'effectivement, avec une telle bande passante, Jane ne sera pas capable d'envoyer ses documents avant le départ de son vol. Ayant accès au service de bande-passante de l'aéroport et au service horaire des vols, Aura découvre qu'une bande passante excellente est disponible à la porte 15 et qu'il n'y a pas de départ ni d'arrivée près de cette porte dans la demi-heure qui suit. Une fenêtre popup s'ouvre alors sur l'écran de Jane pour lui suggérer d'aller à la porte 15, qui ne se trouve qu'à trois minutes de l'actuelle porte...

Dans ce scénario, la situation physique de Jane est adaptée et la fenêtre *popup* en constitue le moyen. Ce scénario comporte beaucoup de points difficiles comme l'observation à priori d'états, la gestion de la distraction de l'utilisatrice par le *popup*, l'accès aux services de l'aéroport. Au niveau technique, la localisation géographique de l'utilisateur est faite par triangulation des requêtes-réponses par rapport à des points-d'accès réseaux. Au point de vue théorique, le point important à comprendre dans la plate-forme Aura est comment, de manière générique, modéliser et construire ces mécanismes particuliers de découverte, proche de l'adaptation, au sein de la plate-forme Aura. Nous étudions dans les paragraphes suivants les principes-clés d'Aura.

Aura dispose de deux concepts accessibles à tout niveau de conception du système : la proactivité et l'auto-ajustement. La proactivité est la capacité à anticiper des requêtes. `Auto-ajustement` (Angl. *self-tuning*) L'auto-ajustement consiste, par observation, à ajuster leur performance et l'utilisation des ressources en fonction de la demande. Garlan [50] souligne l'importance d'une adaptation qui se fait à tous les niveaux hiérarchiques de conception et pendant l'exécution des applications :

> "*Research is needed not just on the building blocks of pervasive computing but in their seamless integration.*" [1]

La plate-forme Aura se décompose ainsi en quatre sous-systèmes nommés Coda [2], Odyssey, Spectra et Prism. Coda est un système de fichiers distribués [52] fournissant un support pour l'accès nomade. Les fichiers sont répliqués sur plusieurs serveurs. Odyssey fournit un support pour l'observation des ressources et l'adaptation des applications. Spectra gère un mécanisme d'exécution distante et utilise les observations d'Odyssey pour décider de la manière d'exécuter les appels distants. Enfin, Prism fournit un support logiciel pour mesurer l'attention de l'utilisateur. Nous voyons deux points particuliers qui sont la gestion de l'environnement et la gestion de l'attention de l'utilisateur. Dans Odyssey, l'environnement d'une application [53] est composé des ressources disponibles pour cette application. Il y a deux types de ressources : les `ressources génériques` et les `ressources spécifiques`. Les ressources génériques ont du sens pour toute application. On y retrouve la bande-passante, l'espace disque disponible, la puissance restante de la batterie. Les ressources spécifiques ont du sens uniquement pour certaines applications comme le nombre de requêtes sur Internet par jour. À partir de cet environnement, Odyssey fournit la disponibilité des ressources mesurée par un unique scalaire. Chaque scalaire a une unité particulière. Ainsi, la bande-passante s'exprime en nombre de bits par seconde, la durée d'utilisation d'une batterie en minutes. Prism est un gestionnaire de tâches [51] dont le but est de minimiser la distraction de l'utilisateur face aux quatre catégories de changement suivantes :

- Changement de localisation géographique et nouvel environnement pour l'application,
- Amélioration de la qualité de service par action de l'utilisateur,
- Changement par Prism de la tâche principale de l'utilisateur,
- Modification de la correspondance entre les contraintes de l'application et l'état de l'environnement.

L'idée de Prism est de travailler sur une description indépendante des tâches. Ce travail élève ainsi le niveau d'abstraction vers une notion de `service` qui leur est propre avec des commandes simples comme *edit text* et *play video*.

Aura apporte deux contributions originales à l'adaptation. D'abord, les requêtes faites par les couches logicielles de haut-niveau sont anticipées par les couches logicielles de niveau inférieur. Ensuite, la couche logicielle de bas niveau adapte la sollicitation des ressources en fonction des requêtes des couches logicielles de plus haut niveau.

2.2.2 *ExORB* : reconfiguration dynamique pendant l'exécution

ExORB [54] suivant démontre la capacité de mise en place d'abstractions au sein d'un intergiciel pour supporter sa configurabilité, la possibilité de sa mise-à-jour et son amélioration. ExORB examine en particulier les intergiciels pour les téléphones portables. De tels dispositifs requièrent un intergiciel sur lequel il est possible de configurer de nouveaux logiciels, d'améliorer les logiciels déjà intégrés sans intervention manuelle de l'utilisateur final. ExORB a pour but principal de contribuer à la construction de services pour intergiciel en renforçant les caractéristiques principales de configurabilité, de capacité de mise à jour et d'amélioration. ExORB étend l'approche UIC (Angl. *Universally Interoperable Core*) [55]. L'UIC est un intergiciel réflectif dont la conception s'appuie sur

1. La recherche n'est pas uniquement sollicitée sur les composants d'une application diffuse mais sur leur intégration invisible à l'utilisateur.
2. http ://www.cs.cmu.edu/~coda/

l'architecture DynamicTAO [56]. Le but d'UIC est d'offrir un intergiciel que l'on peut reconstruire pour chaque cible matérielle qui peut changer d'un type de service à une autre. Chaque configuration personnalisée est connue dans cette architecture sous le terme de `personnalité` (Angl. *personality*). Par exemple, une instance de l'intergiciel peut être configurée pour offrir un bus ORB CORBA pour interagir avec d'autres applications s'apuyant sur le même bus, tandis qu'une alternative pour un autre dispositif serait d'implémenter une personnalité basée sur Java RMI ou SOAP.

Les informations sur l'intergiciel sont disponibles à distance sur le bus. Cela permet de contrôler la configuration logicielle par un logiciel tiers. Cette technique s'appuie sur les trois abstractions suivantes :

1. Micro-blocks de construction (MBB [3]) : la plus petite unité logicielle adressable du système. Par exemple, cela peut être une méthode.
2. Action : l'ordre dans lequel les MBBs s'exécutent. Elle définit donc la logique du système. Ainsi, une action peut être modélisée par un graphe orienté déterministe dont les nœuds sont des MBBs et les arrêtes, les transitions d'exécution.
3. Domaine : cela regroupe des MBBs. Il stocke une liste des MBBs et les actions qui y font référence en plus du lieu géographique du domaine. Un domaine agrège donc un ensemble de MBBs au sein d'une même unité qui peut être suspendue, reprise, insérée et retirée.

L'évolution du logiciel peut ainsi être contrôlée lors de son exécution. La technique d'intercepteur est utilisée dans l'intergiciel. Des opérations additionnelles peuvent être invoquées avant et après l'envoi de requêtes ORB. Par exemple, ils peuvent construire des intercepteurs pour encrypter et décrypter un buffer de messages ce qui a pour conséquence de faire évoluer la plate-forme avec de nouvelles fonctionnalités telles que la sécurité des données. L'interception est accomplie par l'ajout d'un nouveau composant implémentant la fonctionnalité additionnelle à la structure `externalisée`. Enfin, l'`action` est ensuite mise à jour pour s'assurer que l'opération est appelée avant et après l'opération attendue comme 'Send MBB' (envoyer un MBB).

> *ExORB apporte deux contributions originales à l'adaptation. D'abord, il abstrait le code exécutable sous la forme de blocks MBB (la plus petite unité logicielle dynamiquement remplaçable dans son modèle). Un MBB est indépendant du langage de programmation. L'exécution des MBBs est gérée par un ordonnanceur. Ensuite, il permet l'indication des points de sûreté dans l'application où l'ordonnanceur va s'arrêter avant d'effectuer la reconfiguration.*

2.2.3 *DoAmI* : automatisation de la reconfiguration logicielle

DoAmI [57] porte sur l'agrégation dynamique de services distribués. DoAmI (DOmain-specific AMbient Intelligence) propose un modèle d'architecture qui s'appuie sur un intergiciel orienté service pour l'intégration et l'activation de services à l'exécution. Ce travail est le fruit du projet BelAmI [58] (Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems) [4].

Le système a pour but de gérer l'hétérogénéité au niveau logiciel. Il propose un service centralisé de découverte. Ainsi, selon les services découverts et selon le contexte courant, DoAmI les interconnecte et les active. DoAmI est une approche hiérarchique à base de quatre couches logicielles : couche applicative, couche intergicielle, couche de communication et couche physique. Nous nous intéressons aux couches applicatives et intergicielles, et plus précisément à la couche intergicielle. Elle contient les services de gestion des nœuds et de la configuration. Le service de gestion des nœuds est responsable de la gestion du cycle de vie et de l'exécution des applications sur les systèmes matériels. Le service de configuration s'occupe de la connexion dynamique des services. Enfin, des systèmes matériels dont les ressources sont limitées (pas de possibilité d'implémentation des

3. Micro Building Blocks
4. Systèmes de collaboration en recherche bilatérale germano-hongrois en l'intelligence ambiante

services) sont regroupés sur une entité appelée "baie des dispositifs" (Angl. *Device Bay*). DoAmI s'appuie sur un modèle orienté service. Un service est une entité modulaire de déploiement dans ces travaux, une définition différente des notions de services plus classiques. Cette notion contient des interfaces requises et fournies qui sont les points d'entrée et les dépendances de ces entités modulaires. Au niveau logiciel, le modèle d'un service se sépare selon deux vues : la vue-type (*type view*) et la vue-instance (*instance view*).

La vue-type est utilisée à la conception pour décider de la configuration finale à établir. La vue-type est alors statiquement couplée au niveau des variables des services. Les types statiques sont vérifiées à l'exécution par le service de configuration par `réflexion`. La vue-instance décrit l'implémentation d'un service. Une implémentation peut supporter plusieurs configurations ordonnées de manière hiérarchique afin de simplifier les décisions à prendre lorsque plusieurs configurations sont possibles pour une configuration demandée. La "meilleure" configuration (offrant le plus de fonctionnalités) se trouve à la racine. Le service de configuration gère les types et a pour but de les garder consistants. Il les utilise pour la configuration dynamique du système. Lorsque des services sont déconnectés du service de configuration, les références distantes dépendantes de ces services sont toutes retirées et les services dépendants sont arrêtés.

La configuration dynamique de DoAmI est régie par un algorithme simple d'adaptation dynamique. Au niveau de la découverte, le service de configuration se réserve la négociation des connexions des dépendances. Au niveau implémentation, DoAmI est écrit en Java et C++. Il utilise CORBA comme bus de communication qui permet l'intégration de plusieurs langages de programmation.

> *DoAmI apporte trois contributions originales à l'adaptation. D'abord, il permet la déclaration de connexions automatiques (sans intervention du programmeur) des composants logiciels. Ensuite, il permet la recherche de services par la plate-forme et non pas par le composant. Cela simplifie le code des composants en réduisant le nombre d'interfaces pré-définies. Enfin, il permet l'ajout d'interfaces de services pendant l'exécution de l'application.*

2.2.4 *Gaia* : gestion de l'hétérogénéité des dispositifs

Gaia [59] est une étude sur les applications "conscientes" dans le domaine des transports. Gaia travaille sur les bâtiments équipés de capteurs et d'actionneurs appelés espaces actifs. Gaia est un système créé à l'Université d'Illinois à Champaign aux états-Unis qui s'appuie sur la notion d'`espaces actifs` qui est la dénomination utilisée pour qualifier les pièces d'un bâtiment équipé en capteurs et actionneurs tels que des écrans, des télécommandes, etc. Toutes les entrées, sorties et unités de traitement dans une pièce sont modélisées comme étant les périphériques gérés par un unique système d'exploitation. Gaia repose sur un dépôt de composants logiciels représentant chaque entité logique de l'application et une approche centralisée pour la gestion des événements. Il est également doté d'un système de découverte de services.

Tout cela confère à Gaia la capacité de mettre à jour des composants de l'application en remplaçant les composants du dépôt. Le cycle de fonctionnement de Gaia est constitué (1) d'un algorithme (appelé aussi protocole) de *bootstrap* qui permet de démarrer les services noyau du système qui s'exécute en premier, (2) ensuite, d'un protocole d'instanciation des applications, (3) enfin, d'un protocole d'interaction avec les nœuds d'exécution pour créer des composants et les assembler. Une phase de test est également prévue pour pouvoir expérimenter le fonctionnement individuel des composants logiciels et leurs applications. Gaia utilise un langage de script impératif LuaOrb [60]. Ce langage permet de programmer les différentes phases du cycle de fonctionnement. Ce langage a été conçu afin de réduire la quantité de code en abstrayant le caractère distribué du système.

Gaia s'appuie sur un modèle inspiré de MVC [5] ; lorsque nous parlons de `modèle` et de `présentation` (angl. *view*), c'est à ce modèle que nous nous référons. Un script Gaia utilise le dépôt d'espaces

5. *Model-View-Controller pattern*, Pr. Trygve Reenskaug (1979)

actifs pour obtenir un pointeur sur un dispositif de sortie (audio, par exemple), un nœud d'exécution de modèles et un nœud d'exécution pour le coordonnateur. Par exemple, il peut également contenir un écran tactile servant de capteur d'entrée d'informations. Il utilise ensuite les fonctionnalités de base offertes par Gaia pour la gestion des composants logiciels pour créer le coordonnateur, le modèle, la présentation et le capteur. Nous distinguons trois niveaux d'interaction avec Gaia : un premier niveau direct avec la plate-forme pour effectuer une recherche des nœuds et stocker leurs pointeurs localement, un deuxième niveau intermédiaire permettant, à partir des nœuds, de construire les composants logiciels toujours localement et enfin, un troisième niveau permettant d'associer les différents composants.

Gaia offre deux contributions originales à l'adaptation. D'abord, il propose un cache logique des propriétés bas-niveau, des capteurs et des actionneurs pour une exportation des paramètres ayant du sens pour l'application en cours. Ensuite, il propose une composition dynamique des services selon les tâches effectuées par l'utilisateur.

2.2.5 *CORTEX* : généralisation de la notion d'événement

CORTEX [61] a pour but de construire un intergiciel à base de composants pour accueillir des applications dites conscientes en environnement extérieur : notamment dans le domaine du transport. Il s'agit de mettre en évidence les challenges soulevés par ce projet. L'étude de CORTEX se fait en deux parties : en environnement contrôlé (en laboratoire) et en environnement réel. En environnement contrôlé, la problématique s'énonce de la manière suivante :

- Avoir un comportement coopératif sans contrôle humain,
- Avoir une navigation autonome selon des coordonnées GPS de points par lesquels passer définis à l'avance.

Les points clés d'une telle application sont d'abord, les réseaux sans fil ad hoc à haut débit pour la communication entre les véhicules et l'utilisation de technologies de localisation ne nécessitant aucune intervention sur l'environnement physique telles que le GPS et les systèmes de détection de bordure de chaussée. L'application consiste à faire coopérer des véhicules sans l'assistance de l'utilisateur pour naviguer selon un itinéraire donné. Ces véhicules deviennent sensibles au contexte et autonomes.

Pour accomplir ces applications, CORTEX a besoin de fonctionnalités et des services locaux pour des prises de décision locales. Le système participe à un système global coopératif. Ainsi, il est doté d'un système de perception temps-réel de son environnement immédiat (obstacles). La coopération entre les véhicules permet d'éviter les collisions, de réaliser des chaînes de véhicules. De plus, les systèmes locaux peuvent être dotés de fonctionnalités supplémentaires comme la prise en compte des feux de signalisation ou un mécanisme pour laisser passer les piétons (capteur de présence, évitement d'obstacles).

En environnement réel, la problématique s'énonce avec un peu moins de liberté. Ainsi, les spécifications sont différentes : l'absence de contrôle humain devient par exemple une assistance minimum au conducteur (dont le rôle est de surveiller et de prendre le contrôle lorsque le système a un comportement non attendu). CORTEX définit les objets conscients (*sentient*). Ce sont des objets mobiles qui ont un comportement autonome et sont responsables des interactions avec l'environnement physique. Ces comportements se font en fonction des entrées des capteurs et de l'état interne à chaque objet. L'organisation de ces objets permet de construire des patterns d'interactions non prévus à l'avance selon par exemple la proximité géographique. Ces objets conscients permettent au système de fonctionner de manière indépendante de l'utilisateur. Est mis en place un réseau de communication sans fil ad-hoc pour faire face au problème de la mobilité.

Pour prendre en compte les problèmes de coordination, de contrôle, d'adaptabilité et de passage à l'échelle, CORTEX propose un premier paradigme de programmation à partir d'`objets conscients` [62]. Le modèle de programmation de CORTEX s'appuie sur un modèle de communication événementiel non bloquant. Le modèle inclut des mécanismes pour spécifier les contraintes

de propagation et de livraison d'événements. Ils doivent cependant mettre en place une technique optimale de filtre pour que le paradigme événement passe à l'échelle. Les événements sont également regroupés en zones pour limiter leur portée. Une interaction dans CORTEX englobe les mécanismes de coordination et de communication. Un mécanisme à événement permet la génération spontanée de messages plutôt qu'un style RPC. Ils proposent un modèle gérant l'indépendance des objets : la communication est anonyme s'appuyant sur des canaux typés reliant les producteurs aux consommateurs. Le modèle n'oblige pas à avoir un transfert de contrôle explicite ni de synchronisation. Voici donc l'hypothèse de base : les objets sensibles communiquent via le réseau, indirectement à travers l'environnement en le modifiant.

Les objets conscients apportent deux contributions originales à l'adaptation. D'abord, CORTEX propose un modèle de composant représentant des entités logicielles et matérielles capables d'interagir avec l'environnement. Ensuite, il propose une communication entre objets basée sur la notion d'événement générique qui homogénéise la communication avec le monde réel et les entités logicielles.

2.2.6 *K-Component* : propriétés et mécanismes de reconfiguration

K-Component [63, 64, 65] est un modèle de systèmes adaptatifs fonctionnant dans un environnement décentralisé. Il utilise les architectures dynamiques et la réflexion structurelle pour la construction de systèmes adaptatifs. Il aborde les problèmes d'intégrité et de sécurité de l'évolution dynamique du logiciel en modélisant la reconfiguration dynamique par des transformations de graphe d'une architecture logicielle. Ils séparent aussi le code spécifique à l'adaptation du code fonctionnel en l'encapsulant dans des programmes réflexifs appelés contrats d'adaptation.

Le modèle inclut un langage de description d'interface de composant K-IDL, une extension d'IDL [66]. Il supporte la définition d'états de composants logiciels, d'actions d'adaptation ainsi que les interfaces requises. Les états de composants et les actions d'adaptation sont utilisés par des programmes de prise de décision pour raisonner sur les opérations des composants afin de les adapter. Un composant au niveau méta, appelé le gestionnaire de configuration, stocke et gère le graphe de configuration de l'architecture logicielle.

Aussi, une transformation de graphe est une modification à partir des règles du graphe de configuration. Ces règles définissent comment et quand un graphe doit être transformé. Cette transformation consiste à remplacer des composants logiciels appartenant à une configuration particulière et à modifier les stratégies des connecteurs. L'alternative qui consiste à permettre l'introduction de nouveaux services dans un système pendant son exécution laisse ouvert le problème suivant : "comment les composants et les clients deviennent conscients et accèdent aux nouveaux services à travers leurs interfaces à l'exécution ?" Dans le cadre d'un logiciel auto-adaptatif, leur modèle permet de contraindre les reconfigurations dynamiques possibles du système aux reconfigurations les plus significatives. Ils peuvent garantir l'intégrité et la consistance du système si les règles sont des opérations transactionnelles sur le graphe, au lieu de complètement arrêter le fonctionnement de l'application.

K-Component apporte une contribution originale à l'adaptation. Chaque entité a une vue partielle du système. Chacune a pour but d'augmenter la qualité (en terme de coût des communications réseau par exemple) en fonction des retours des entités environnantes. Il définit des fonctions d'évaluation (sélection des bonnes actions) et la politique de sélection.

2.2.7 *SmartSpace* : modélisation de l'infrastructure de dispositifs

SmartSpace [67, 68] est un système qui s'appuie sur un modèle de programmation par espace d'états et un compilateur fonctionnant, à la fois, à la conception et à l'exécution des programmes. Des règles logiques permettent de passer d'un état à un autre et chaque état correspondant à un ensemble d'actions accomplies. Nous avons extrait de [67] un exemple de programmation d'un scénario d'application dans un langage centré sur cette notion d'état pour les *smart spaces* nommé RCAL [68] [6]. Ce programme a pour but de calculer l'état courant de l'espace à partir des observations effectuées à partir de capteurs du *smart space*. Il se compose de règles logiques définissant les transitions d'état. Notons qu'un état peut être "actif" dans le sens où il peut déclencher un actionneur. Les directives de communication entre les différents espaces ne sont pas exprimées dans cet exemple afin de simplifier le code.

```
1  space(office) {
     interface:
       sensor(indoorTemperature/1),
4      sensor(outdoorTemperature/1),
       sensor(motion/1),
       sensor(light/1),
7      actuator(window/1),
       actuator(lightSwitch/1).

10   pre_state(lightOn) [
       bright(X) :- X>20.
       state :- light.average(X), bright(X).
13     state :- motion(X), X<2.
     ] post_states(lightShouldBeOff).

16   pre_state(lightShouldBeOff) [
       state :- lightSwitch(0).
     ] post_states(lightOff).
19
     ...
     ...
22 }
```

FIGURE 2.2 – Exemple d'implémentation d'un scénario applicatif

Nous expliquons à travers cet exemple le fonctionnement et la programmation des espaces intelligents. Les lignes 2 à 8 de l'exemple dans la Figure 2.2 déclarent l'interface de l'espace appelé '*office*'. A la ligne 3, par exemple, un capteur nommé **indoorTemperature** est déclaré. Ce capteur délivre une valeur de température à la fois, ce que nous pouvons voir à travers la déclaration du nombre de paramètres "/1" du prédicat. Après la déclaration des capteurs et des actionneurs, nous trouvons deux règles aux lignes 10 et 16. Ces règles expriment les transitions d'état. La première règle (10-14) est appliquée à condition que l'espace soit dans l'état **lightOn**, c'est-à-dire que la lampe soit allumée. La règle définissant cet état n'est pas représentée dans l'exemple. Ensuite, à la ligne 11, nous avons la déclaration d'un nouveau prédicat qui définit ce que signifie **bright/1**, c'est-à-dire que son paramètre (sous-entendu représentant une intensité lumineuse) doit être supérieur à la constante 20. Puis, nous avons deux déclarations de prédicats **states** juxtaposées aux lignes 12 et 13 formant un OU logique. Enfin, la dernière règle à la ligne 16 définit l'état **lightShouldBeOff** comme l'action d'éteindre les lampes à travers le prédicat de la ligne 17 fournissant le paramètre "0" au prédicat **lightSwitch/1**. L'état courant se change ensuite en l'état **lightOff** qui n'est pas décrit dans l'exemple.

Le second paradigme de Smart Space concerne la correspondance entre les prédicats logiques du programme en RCAL et des services sous-jacents (capteurs et actionneurs). Ces deux modèles sont ensuite transformés en un modèle exécutable interprété par un intergiciel.

6. RuleCaster Application Language

SmartSpace propose ainsi deux contributions dont la première est le découplage des entités logicielles avec l'infrastructure réseau. La seconde concerne l'utilisation de règles logiques pour contrôler la transition entre les états du système, ce qui rend plus simple l'analyse des adaptations effectuées.

2.2.8 Autres approches

Dans les sections précédentes, nous avons présenté les modèles pour l'adaptation logicielle proposant des apports différents en termes d'architecture : modèle d'intercepteur pour optimiser les échanges entre couches logicielles pour Aura, prise en compte des conflits de fonctionnalités dans ExORB, découplage entre entités logicielles dans le reste des travaux avec une prise en compte du contexte et une gestion de l'hétérogénéité pour les objets conscients. Cette section présente les spécificités d'autres modèles méritant aussi d'être soulignées.

La plate-forme Oxygen [69] adresse la problématique de l'interface utilisateur en proposant une communication à travers la parole et la vision. Le système modélise d'abord le profil d'une personne. Il définit ensuite des réseaux de communication ayant des topologies dynamiques en fonction de la position des dispositifs d'interaction. Cette topologie distingue quatre types de dispositifs : dispositifs fixes, dispositifs mobiles, dispositifs capteur/actionneur ou dispositif embarquant un logiciel. Dans ce dernier cas, le code logiciel peut être automatiquement mis à jour. Enfin, des règles de connexion du réseau peuvent être définies afin de permettre à des ensembles d'utilisateurs d'avoir accès à des ressources particulières fournies par l'environnement.

La plate-forme Scorpio [70] propose un travail sur l'adaptation structurelle de composants logiciels. Scorpio ré-assemble les composants pour créer une correspondance entre les interfaces hétérogènes lorsque de nouveaux composants logiciels doivent être intégrés. De plus, ils proposent de diviser les services comportementaux en plusieurs groupes afin de les déployer séparément sur différents systèmes pour permettre une répartition des charges.

La plate-forme Satin [71] s'appuie sur des règles logiques pour exprimer la mobilité. Satin signifie *Self Adaptation targeting Integrated Networks*. Elle défend la thèse qu'une application de règles logiques pour exprimer la mobilité dans un système à base de composants permet d'assister la construction des systèmes auto-organisateurs. Ils définissent un intergiciel à base de composants disposant d'une mise à jour dynamique selon le changement de contexte.

L'injection de dépendances [72] : devant le constat de nombreux défauts de lourdeur d'utilisation et de mise en place des plates-formes telles que CCM, des approches dites à `conteneurs légers` sont apparues. En effet, il existe une disproportion du temps passé à utiliser les services techniques par rapport aux services fonctionnels dans les approches à conteneurs comme EJB. Le canevas Spring (Spring Framework) est un canevas open source pour la plate-forme Java. Il s'impose comme une alternative aux serveurs d'application EJB en implémentant, entre autres, un système pour l'inversion de dépendances (IoC - Inversion of Control en anglais) qui permet de faire de l'injection de code. L'inversion de dépendances est une des méthodes pour faire de l'aspect. Pour réaliser son inversion de contrôle, Spring est doté d'un noyau aspect qui montre qu'aspect et composant peuvent être associés. Ici les aspects sont utilisés pour faire la "glu" entre les services techniques et le code fonctionnel. Spring est une approche à conteneur et ne définit pas de représentations au niveau du code métier.

Il nous semble primordial de rendre explicite les déclarations de dépendances entre composants, en mettant en avant la notion d'architecture logicielle et, en particulier, de donner la possibilité de définir des hiérarchies de composants. Un conteneur léger prend en charge la création d'entités et leur mise en relation. L'inversion de contrôle consiste à laisser au programmeur la définition des composants sans relation dure entre eux. C'est une entité tierce (la plate-forme) qui se charge de les mettre en relation.

2.2.9 Synthèse et conclusion

Avec Aura, les requêtes faites par les couches logicielles de haut-niveau sont anticipées par les couches logicielles de niveau inférieur. La couche logicielle de bas niveau adapte la sollicitation des ressources en fonction des requêtes des couches logicielles de haut-niveau. ExORB abstrait le code exécutable sous la forme de blocs MBB dont l'ordonnancement est contrôlé. Il propose d'indiquer des points de sûreté dans l'application où l'ordonnanceur va s'arrêter avant d'effectuer la reconfiguration. DoAmI permet la connexion automatique sans intervention du programmeur des composants logiciels et la recherche de services par la plate-forme et non pas par le composant. Cela simplifie le code des composants en réduisant le nombre d'interfaces pré-définies. Il permet aussi l'ajout d'interfaces de services pendant l'exécution de l'application. La plate-forme Gaia, elle, propose la mise en place d'un cache logique des propriétés bas-niveau des capteurs et actionneurs. Le but est de n'exporter que les paramètres ayant du sens pour l'application en cours : un filtre logique. Il propose une composition dynamique des services selon les tâches effectuées par l'utilisateur. Les objets conscients apportent deux contributions originales à l'adaptation. D'abord, il propose un modèle de composant représentant des entités logicielles et matérielles capables d'interagir avec l'environnement. Ensuite, il propose une communication entre objets basée sur la notion d'événement générique qui homogénéise la communication avec le monde réel et les entités logicielles. Dans K-Component, chaque entité a une vue partielle du système et a pour but d'augmenter la qualité (en terme de coût des communications réseau par exemple) en fonction des retours des entités environnantes. Il définit des fonctions d'évaluation (sélection des bonnes actions) et la politique de sélection.

Les plates-formes pour l'adaptation s'articulent autour de concepts fondamentaux tels que les composants logiciels. Nous voyons dans la section suivante l'utilisation du paradigme à composants pour l'adaptation.

2.3 Plates-formes à composants

Dans cette introduction, nous positionnons dans un premier temps l'abstraction des composants logiciels parmi les modèles de programmation. COM (Component Object Model) [73] fait partie des premiers modèles de composants. COM est plus un modèle binaire qui résulte d'une réflexion sur "comment implémenter des composants logiciels présents au cours de l'exécution ?" qu'une réflexion conceptuelle comme c'est le cas dans la plupart des modèles à composants. De ce fait, COM met en place un modèle qui permet la définition d'interactions entre composants logiciels éventuellement écrits avec des langages différents. OpenCOM [74] propose ensuite une version étendue du modèle de composant COM, version issue des travaux de l'Université de Lancaster pour implémenter l'intergiciel OpenORB. C'est une évolution en vue de la mise en place d'intergiciels réflexifs et reconfigurables. Les contributions de ce modèle à COM commencent par la définition des services requis (optionnel dans COM) et par l'identification des connexions comme étant des éléments architecturaux. La deuxième contribution majeure d'OpenCOM est d'avoir défini un méta-niveau pour tous les composants. Ce méta-niveau fournit essentiellement des services d'introspection et de reconfiguration dynamique des composants. Le lecteur pourra trouver une discussion précise des travaux que nous ne présenterons pas dans la thèse de Bures [75] (à savoir Darwin, Unicon, Rapide, C2, Acme, Wright, Olan, Reo, CommUnity, Wireless Connectors, SYNTHESIS, Exogeneous connectors et Java/A). L'auteur a travaillé sur la problématique de l'adaptation/génération dynamique de connecteurs, donnant la deuxième version du modèle SOFA [76].

Description de composant. La description d'un composant (aussi appelé *type de composant* dans un modèle orienté objet) est une entité logicielle caractérisée par des interfaces de deux sortes :

1. Interfaces d'entrée (ou fournies) du composant définissent les fonctionnalités offertes par une description de composant donnée en énumérant un ensemble de ports ;
2. Interfaces de sortie (ou requises) du composant définissent les fonctionnalités dont le composant a besoin pour tout ou partie de son fonctionnement interne.

Les interfaces définissent les points de communication des composants et sont utilisées pour leur interconnexion. Les modes de communication sont spécifiés pour chaque interface requise pour préciser le mode d'interaction (synchrone, asynchrone, synchronisation, etc.).

Implémentation d'un composant. L'implémentation d'une description de composant (aussi appelée implantation) regroupe deux notions : l'implémentation du contrôle et l'implémentation des interfaces d'entrée. L'implémentation des interfaces d'entrée représente sa mise en œuvre. L'implémentation du contrôle (parfois référencée comme services techniques) regroupe, quant à elle, les fonctionnalités du modèle de composant (connexion, cycle de vie, voire, sécurité, persistance). Elle est souvent juste décrite car déjà intégrée dans l'implémentation du modèle de composant.

Instance de composant (à laquelle nous faisons référence dans la suite par le terme "*composant*"). Une instance de composant est une entité s'exécutant localement sur un système. Elle est caractérisée par une référence unique, une description et une implémentation. Les instances de composant peuvent communiquer entre elles à travers des ports constituant leurs interfaces (par invocation de méthode, envoi de messages, etc.).

Afin de clarifier la notion de composant, nous proposons d'identifier les apports des modèles par rapport à nos critères concernant les systèmes informatiques ambiants adaptatifs. Un ensemble de plates-formes à composants est décrit dans les sections suivantes. Nous avons choisi les travaux autour des modèles et des plates-formes mKernel, CORBA "Connector" Model, Interface d'adaptation, Constructions élémentaires pour les patterns d'architecture, Cache d'assemblage, SOFA 2.0 et Fractal. Ils ont été choisis pour leurs différences en terme d'architecture et de mise en œuvre de l'assemblage dynamique de composants logiciels. Nous voyons ensuite d'autres approches apportant des idées importantes pour les modèles de composant.

2.3.1 *mKernel* : administration en EJB en environnement changeant

Enterprise Java Beans (EJB) [77] est un modèle proposé par Sun qui est destiné à la mise en place des serveurs d'application. Un composant EJB est essentiellement un objet Java qui fournit un certain nombre d'opérations de contrôle. Pour cette raison, un composant EJB n'a qu'une seule interface serveur. Il existe trois types de composants EJB : les composants session, message et entité. Les composants session sont destinés à être instanciés pour chaque session d'interaction entre un client et le serveur. Leur durée de vie est équivalente à la durée de l'interaction. Les composants session sont dédiés à l'implémentation de la logique métier (du comportement), et sont distingués par le fait qu'ils sont une optimisation de l'implémentation. Les composants message ont essentiellement le même rôle que les composants session mais offrent un mode de communication asynchrone. Enfin, les composants entité sont de durée de vie indéfinie et sont dédiés à la réification des données persistantes. mKernel [78] apporte des fonctionnalités supplémentaires aux EJBs pour y ajouter la dimension dynamique : pouvoir gérer les composants EJB à l'exécution.

Les composants EJB sont assemblés et exécutés au sein de conteneurs. Les conteneurs implémentent la création et la destruction des composants EJB et ont la charge de leur fournir un environnement d'exécution comprenant des services non-fonctionnels de type persistance, gestion des transactions, etc. Un conteneur s'exécute dans une machine virtuelle Java, et peut gérer plusieurs types d'EJB. L'instanciation de composants se fait par l'interface de composant `Home` fournie par les conteneurs. Les utilisateurs peuvent alors étendre cette interface pour supporter des fonctionnalités additionnelles. Les conteneurs sont aussi responsables de la gestion des accès aux composants qu'ils accueillent. Alors que les composants se trouvant dans le même conteneur peuvent interagir directement, l'exportation des interfaces par le conteneur est requise pour donner accès aux fonctions métier depuis l'extérieur. Les conteneurs peuvent intercepter les interfaces qu'ils exportent pour effectuer des pré- et post-traitements.

L'approche EJB propose de séparer l'implémentation des fonctions métier de certains aspects non-fonctionnels. Les composants encapsulent des codes métier et bénéficient des environnements d'exécution enrichis fournis par les conteneurs. Pourtant, cette approche ne propose aucun support de génération de code. Parmi les services génériques, nous pouvons citer le service de nommage et de répertoire supporté par JNDI et les communications synchrones et asynchrone supportées respectivement par Java-RMI et JMS. La mise en œuvre des composants et des conteneurs EJB se fait en Java en étendant deux interfaces prédéfinies. L'interface de base pour un composant EJB s'appelle EJBObject. Via cette interface, nous pouvons obtenir une référence unique pour chaque instance de composant, et l'on peut accéder au conteneur qui l'accueille. Cette interface de base doit être étendue pour définir des types de composants EJB avec leurs fonctions métier. Pour mettre en œuvre un conteneur EJB (EJBHome), l'interface de base fournit une référence unique au le conteneur. Cela permet d'effectuer des recherches parmi les composants lui appartenant. Cette interface doit être étendue pour spécifier les opérations spécifiques à un type de conteneur donné. Par exemple, une interface Home spécifique qui permet la création et la destruction des composants.

```
1  public interface VideoStreamer extends javax.ejb.EJBObject {
       byte{[}] getContent() throws RemoteExeption;
       intgetHeight() throws RemoteException;
4      intgetWidth() throws RemoteException;
   }
   public interface VideoStreamerHome extendsjavax.ejb.EJBHome {
7      VideoStreamer create() throws RemoteException, CreateException;
       void remove(VideoStreamerinstance) throws RemoteException,
              RemoveException;
10 }
```

FIGURE 2.3 – Définition d'un type et d'un conteneur

Les EJBs sont largement utilisés pour la mise en place des serveurs de commerce électronique. Le fait que ce modèle soit simple et directement compatible avec les programmes Java existants, la présence d'outils de conception graphique et le support de la société Sun Microsystems expliquent les raisons de son succès.

Bruhn [78] apporte quatre contributions originales à l'approche composant. D'abord, il propose l'inspection structurelle et l'observation des interactions entre les composants par l'utilisation de callbacks et permet la reconfiguration structurelle. Ensuite, il ne se base pas sur les modifications de l'implémentation des plates-formes d'exécution ni des containers. Il permet finalement l'insertion de code (pour l'observation et la reconfiguration) de manière transparente pour le développeur.

2.3.2 *CORBA "Connector" Model* : interactions en CCM

CORBA (Common Object Request Broker Architecture) [79] est un standard proposé par l'OMG (Object Management Group) pour la programmation des objets répartis. CORBA spécifie des interfaces de programmation et des protocoles de communication de type client/serveur qui permettent de déléguer la mise en œuvre de tous les aspects liés à la communication et à la synchronisation des objets répartis avec leurs environnements d'exécution sous-jacents. Le CCM (CORBA Component Model) spécifie une structure de composant et un environnement d'exécution fournissant des services non-fonctionnels.

Un composant CCM est une entité logicielle qui peut fournir ou recevoir des opérations à travers des interfaces typées, appelées `ports`. Les composants peuvent avoir de multiples ports, ainsi que des attributs qui donnent accès à leur configuration. CCM distingue quatre sortes de ports : les facettes et les réceptacles sont respectivement les interfaces serveur et client en mode de communication synchrone, alors que les `puits` et les `sources` sont utilisés en mode de communication asynchrone. Parmi les interfaces de contrôle générées, nous avons l'`interface de référence` des composants CCM qui permet d'introspecter les ports d'un composant en cours d'exécution.

Les conteneurs fournissent tout d'abord aux composants un environnement d'exécution, c'est-à-dire un emplacement mémoire et un flot d'exécution. Cet environnement fournit aussi des services non-fonctionnels comme des transactions, de la persistance, de la sécurité ou de la tolérance aux fautes ainsi que les moyens de communication basés sur le bus CORBA. Chaque composant est exécuté dans un conteneur, et les conteneurs sont spécifiques à chaque type de composant. Chaque conteneur implémente une dépôt de composants. Le dépôt est responsable du contrôle de ses composants. Les services de contrôle implémentés par un dépôt sont rendus accessibles à travers deux interfaces de type fabrique et recherche qui permettent respectivement de créer ou de détruire des instances de composants et d'effectuer des opérations de connexion et d'introspection des interfaces des composants contrôlés par la maison. Les conteneurs peuvent en quelque sorte être considérés comme des adaptateurs des composants CCM à l'environnement CORBA, fournissant les services non-fonctionnels cités précédemment. Hachichi *et al.* [80] proposent d'opter pour un langage de configuration de haut niveau combiné avec un mécanisme d'adaptation dynamique pour répondre au problème de spécialisation (monolithique, nombre fixe de services non-fonctionnels) des conteneurs pour les composants CCM.

CCM [81] apporte réellement trois contributions originales à l'approche composant dans CORBA. D'abord, il intègre la notion de connecteur dans CCM. Ensuite, il capitalise l'expertise sur la gestion des interactions. Enfin, il permet l'indépendance (par le recours à n'importe quelle couche de communication et la substituabilité des connecteurs) d'un composant par rapport à CORBA.

2.3.3 *SOFA 2.0* : Rééquilibrage des fonctionnalités avancées

SOFA 2.0 [76] est un modèle de composant développé à l'Université Charles à Pragues. SOFA propose quatre points-clé qui sont la conception orientée ADL, la spécification comportementale utilisant des protocoles comportementaux, les connecteurs automatiquement générés supportant une distribution transparente et un environnement d'exécution distribué avec la mise à jour automatique de composants. Les centres d'intérêt de SOFA sont la reconfiguration dynamique des `architectures` de composants [7], l'accès aux composants à travers des concepts SOA, sa modélisation des connecteurs et l'introduction des aspects dans les composants afin de séparer clairement le contrôle (non-fonctionnel) au niveau implémentation. SOFA est à la fois un outil de modélisation et une plate-forme expérimentale supportant toutes les étapes du cycle de vie de l'application du développement jusqu'à l'exécution. Après une présentation des concepts de base, nous présentons un exemple d'application et voyons en détail le modèle de composant.

Le `dépôt` est le concept central de SOFA. C'est une unité de stockage des métadonnées sur les composants ainsi que de leurs implémentations à la fois pendant la phase de conception et pendant la phase d'exécution. Chaque entité stockée dans le dépot est versionnée ce qui permet l'usage de plusieurs versions d'un même composant. Le dépôt supporte le clonage. Ainsi, le développeur peut travailler sur l'amélioration d'un composant en effectuant des modifications sur un clone. Le clone une fois terminé peut être réintégré dans le dépôt. Ce dernier reste dans un état cohérent. Un mécanisme de `commit` permet de tester la cohérence des clones avant d'effectuer le remplacement. Des composants peuvent être exportés ou importés dans le dépôt. `Cushion` est le nom de l'outil de développement de composants SOFA 2.0.

L'architecture d'une application qui recueille des informations des capteurs est simple. Elle comprend trois composants dont deux sont des capteurs et le troisième enregistre les données provenant de ces capteurs. Pour réaliser une application, nous avons besoin de la plate-forme SOFA et de Cushion. Le dépôt de la plate-forme SOFA doit être lancée. Ensuite, tout le travail de développement se fait dans l'environnement Cushion.

Nous commençons par déclarer de manière abstraite une interface de composant nommée **Data**. Un fichier de description ADL est généré. Nous devons renseigner à quelle interface Java cette interface de composant fait référence. Nous pouvons, par exemple, indiquer qu'il s'agit d'une interface Java identifiée par **foo.Data**.

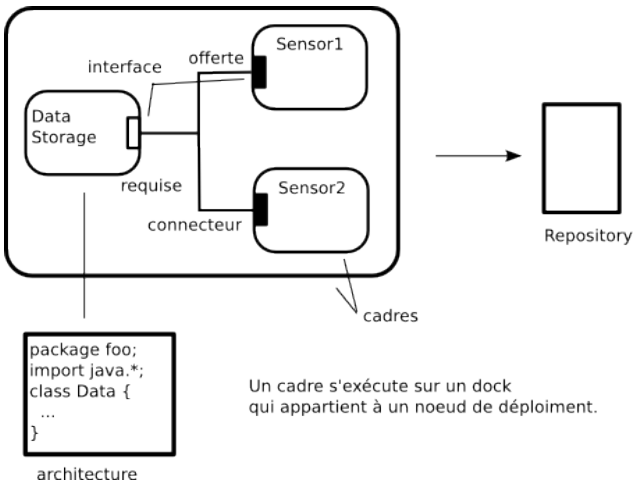


FIGURE 2.4 – Exemple d'application SOFA

Nous déclarons ensuite un cadre nommé qui représente le composant **Data Storage**. Un fichier ADL est généré. Le cadre est initialement vide. Nous renseignons alors l'ADL avec une interface offerte dont le nom est bus et le type celui de l'interface de composant Data que nous avons créée

7. Bures *et al.*[75] nomment `architecture` l'implémentation (le corps ou code) d'un composant logiciel.

ci-dessus. Nous déclarons de la même façon deux autres cadres représentant les capteurs. Nous renseignons alors les ADL avec une interface requise dont le nom est bus également. Enfin, il s'agit de créer un dernier cadre qui représente le composant composite de plus haut niveau. Ce cadre reste vide.

Ensuite, pour chaque cadre de composant, nous concevons une architecture, qui a pour conséquence de générer une description ADL. Nous associons pour les trois premiers cadres déclarés ci-dessus en complétant les ADL une classe Java qui implémente les composants. Pour le dernier cadre de composant, la démarche est différente, car nous désirons en faire un composite. Pour cela, nous déclarons dans l'ADL généré les trois sous-cadres de composant, qui le constituent (**Sensor1** et **Sensor2**) ainsi que les connecteurs entre les interfaces. Enfin, nous effectuons un `commit` du travail. Le dépot est mis à jour.

> *SOFA 2.0 apporte deux contributions originales à l'approche composant. D'abord, il propose la reconfiguration dynamique d'architectures hiérarchiques. Ensuite, il modélise le contrôleur de composants. Enfin, il démontre l'existence d'un ensemble de constructions de base à partir de l'analyse des patterns d'architecture les plus utilisés (incluant les design patterns).*

2.3.4 *Fractal* : un modèle de composant ouvert

FRACTAL [82] est un modèle de composant ouvert et réflexif dédié à la construction de systèmes logiciels en général. Il est utilisé dans divers projets de recherche allant de la mise en place de systèmes d'exploitation que l'on peut spécialiser pour des plates-formes embarquées, jusqu'à la mise en place de systèmes autonomes pour des serveurs d'applications d'entreprises en passant par des systèmes multimédia pour des applications mobiles. La suite de cette sous-section présente brièvement le modèle FRACTAL pour le situer dans le cadre de l'état de l'art sur les modèles de composants. FRACTAL est un modèle hiérarchique [83, 84] et permet ainsi une construction récursive qui s'arrête aux composants primitifs qui sont directement programmés dans un langage de programmation donné. Le modèle FRACTAL permet le partage de composants, c'est-à-dire que plusieurs composants composites peuvent contenir une même instance d'un composant, que ce soit un composite ou un primitif.

Un composant peut avoir plusieurs interfaces. Deux types d'interfaces sont distingués comme dans la plupart des modèles de composant. Les interfaces clientes permettent d'émettre des opérations vers d'autres composants alors que les interfaces serveur permettent d'en recevoir. En plus de la nature client ou serveur, des propriétés de contingence (obligatoire ou optionnelle) et de cardinalité (acceptation de connexions simples ou multiples) sont attribuées aux interfaces. Ces propriétés permettent d'améliorer le niveau de description de l'architecture logicielle, et enrichissent entre autres les vérifications architecturales que l'on peut implémenter. La communication entre une interface client et une interface serveur n'est possible que si ces deux interfaces sont interconnectées par une liaison. FRACTAL distingue deux types de liaisons pour permettre l'implémentation de communications inter-composants de sémantiques différentes. Les liaisons primitives sont des liaisons *langage* et par conséquent sont conformes à la sémantique définie par le langage d'implémentation des composants (comme les appels de méthode synchrones en Java). Des sémantiques autres que celle qui est proposée par le langage d'implémentation peuvent êtres implémentées avec des liaisons complexes. En effet, une liaison complexe est réifiée par un composant de complexité arbitraire, dont le rôle est d'implémenter une forme de communication donnée (par exemple, RPC).

FRACTAL permet de séparer l'implémentation du métier du composant de sa partie contrôle. Dans le cas de composants primitifs, le contenu est considéré comme une blackbox dont la fonction est programmée dans un langage de programmation. Dans le cas des composites, le métier est implémenté par la composition d'un ensemble de sous-composants. Schématiquement, le contenu d'un composant est entouré par une membrane qui implémente la partie contrôle. Parmi les fonctions implémentées dans les membranes, nous pouvons, par exemple, citer les fonctions d'interception

pour tracer ou filtrer les interactions d'un composant avec son environnement. La membrane implémente aussi des interfaces additionnelles en plus des interfaces métier du composant afin de donner à l'environnement du composant des points d'accès pour les aspects de type contrôle. L'ensemble et la nature des interfaces de contrôle implémentées par des composants ne sont pas fixées par le modèle : FRACTAL en propose quelques unes pour le contrôle de l'identité, des liaisons, des attributs, et de contenu basées sur des opérations réflexives comme l'introspection et l'intercession ; d'autres peuvent être spécifiées et rajoutées en fonction des besoins spécifiques liés au contexte applicatif. Le modèle FRACTAL est indépendant de son implémentation ainsi que des langages qui en découlent. Il existe plusieurs implémentations du modèle dans divers langages. Cela permet de viser des contextes applicatifs différents. Parmi ces implémentations, nous pouvons citer :

- Julia [85] en Java,
- AOKell [86] en AspectJ,
- Think (*THink Is Not a Kernel*) [87] en C,
- FractNet [88] en .Net.

Polakovic [89] étudie la construction d'OS par assemblage de composants avec la problématique de reconfiguration dynamique. Il propose un langage de haut niveau pour résoudre les problèmes du passage à l'échelle et propose également de définir un modèle d'exécution événementielle pour une mise en œuvre dans Think. Dans ce modèle, les `composants` interagissent uniquement par `échanges de messages`. Chaque composant possède une ou plusieurs files d'entrée dans lesquelles il stocke les messages qui lui sont destinés et un ensemble de `traitants de messages`. Ces traitants réagissent de façon atomique à l'occurrence d'un message correspondant à un certain `pattern`. Il définit l'état d'un composant comme l'ensemble de ses `observables` qui sont les messages internes, les messages en attente de traitement et traitants. Chaque traitant de message s'exécutant de manière atomique, chaque état observable d'un composant correspond à un état stable. La reconfiguration ne pouvant se faire qu'au moment où les traitants ne sont pas exécutés, il devient donc possible de reconfigurer le système à tout instant.

> *Fractal apporte trois contributions originales à l'approche composant. D'abord, il propose un modèle de composant hiérarchique avec partage de composant entre plusieurs composants composites. Ensuite, les composants sont dotés d'un ensemble de capacités réflexives paramétrables (contrôleurs et intercepteur). Enfin, la modélisation de connecteur se fait par l'intermédiaire de bindings composites qui sont des composants jouant le rôle de connecteurs s'appuyant sur les bindings primitifs.*

2.3.5 *ArchJava : une extension de Java*

ArchJava [90], proposé par l'Université Carnegie Mellon, occupe une place toute particulière dans l'état de l'art de la programmation à base de composants, car il constitue une des premières propositions qui visent à définir un langage à composants. La particularité de cette approche vient de l'intégration des éléments architecturaux et de l'implémentation des composants au sein d'un même langage prenant la forme d'une extension du langage Java.

Les composants sont les seules unités d'encapsulation définies dans le langage ArchJava. Ils communiquent avec d'autres composants via la notion de port. Un port définit un ensemble de méthodes (telles que celles de Java) qui représentent les points d'accès à un composant. Dans ce sens, les ports ArchJava peuvent être considérés comme les interfaces des composants. Un composant peut alors avoir plusieurs ports. Contrairement à la plupart des modèles de composant qui définissent un sens associé à chaque interface (serveur/client ou fourni/requis), aucune propriété de ce type n'est associée aux ports d'ArchJava. En effet, ce sont les méthodes associées à un port qui vont donner ce type d'informations. Dans ces termes, une méthode peut être soit fournie (implémentée par le composant), soit requise (appelée par le composant). Une méthode requise n'a pas de valeur de retour ; un port contenant des méthodes requises peut être également connecté à plusieurs composants, auquel cas l'invocation de la méthode sera diffusée à tous.

Les ports des composants doivent êtres interconnectés pour établir la communication. Est-ce si évident ? ArchJava intègre la spécification des connexions et l'implémentation des composants dans un même langage de programmation. Il est capable et utilise sa capacité pour interdire toute communication inter-composants qui ne passe pas via des connexions établies. Autrement dit, contrairement à ce qui est valable pour les objets en générale, il ne suffit pas d'obtenir une référence à un autre composant pour en appeler les opérations. Ceci est une propriété très forte, car la cohérence entre l'implémentation et l'architecture est ainsi assurée. De plus, cette propriété est assurée pour tout composant instancié dynamiquement, ce qui permet d'observer de manière rigoureuse l'évolution de l'architecture du logiciel au cours de son exécution en termes de composants et de leurs interconnexions.

ArchJava permet la composition hiérarchique des composants dans le sens où un composant peut posséder des sous-composants. Ainsi, l'implémentation des méthodes fournies par un composant peut être faite directement dans le composant en tant que méthode classique, ou être déléguée à une implémentation fournie par un sous-composant. Les connexions entre les composants sont gérées par leur père, c'est à dire le composant le plus proche à qui ils appartiennent. La structure hiérarchique d'une configuration ArchJava est représentée par un arbre, le partage des composants n'étant pas supporté. À l'origine, le modèle de communication entre les composants ArchJava était limité à des invocations classiques de méthode [90]. Récemment, une abstraction a été proposée pour mettre en œuvre des connecteurs supportant diverses sémantiques de communication [91]. Cette proposition consiste en la définition de classes de base pour les ports et les connecteurs qui fournissent des opérations d'introspection sur la signature des méthodes et qui spécifient les méthodes abstraites que doivent fournir les implémentations concrètes des connecteurs. De cette manière, les programmeurs de connecteurs peuvent mettre en place des `stubs` et des `skeletons` génériques en utilisant la puissance de la programmation réflexive. Cette approche fournit une abstraction homogène et robuste pour la construction de moyens de communication supportant tous les types de communication énumérés dans l'article de Shaw *et al.* [92]. Néanmoins, l'utilisation de la réflexivité semble un choix peu efficace en ce qui concerne la performance à l'exécution.

> *Alexandrescu* et al. *[93] présente une évaluation de l'implémentation d'ArchJava. Les conclusions de ce rapport montrent qu'ArchJava apporte des bénéfices considérables en matière de conception de logiciels flexibles alors qu'il présente certaines surcharges en termes de taille de code binaire et de performance d'exécution. Parmi les critiques au langage ArchJava, nous pouvons citer l'absence de méta-niveau qui permettrait de contrôler le comportement des composants, la dépendance envers l'environnement Java qui limiterait son utilisation dans un cadre applicatif restreint, et le fait que la programmation des connecteurs flexibles soit uniquement basée sur la technologie de réflexivité, ce qui nous interroge sur les performances du système en général.*

2.3.6 *SystemC : une extension de C++*

SystemC [94, 95] est une extension du langage C++ proposée par OSCI (Open SystemC Initiative) qui regroupe un nombre important d'industriels dans le domaine de la conception de produits microélectroniques pour modéliser des systèmes matériels. Depuis sa version 2.1 publiée en 2005, SystemC intègre un modèle de composant qui est intéressant pour la modélisation des systèmes. Dans la suite de cette section, nous présentons le modèle de composant de SystemC et nous présentons son environnement d'exécution sous-jacent.

Les modules constituent les éléments d'encapsulation de comportements et de données en SystemC. Les modules peuvent interagir avec d'autres modules se trouvant dans leur environnement par le biais de leurs ports. Les ports permettent des échanges de données en entrée ou en sortie et sont définis par un type d'interface et par une cardinalité. En d'autres termes, un port encapsule un nombre défini d'interfaces de même type. Une interface définit le type des données qui peuvent être échangées ainsi que le sens de l'échange. Les ports des modules ont besoin d'être connectés

pour établir leurs interactions. SystemC propose une classe de base qui définit certaines opérations par défaut pour la mise en place de différents types de connecteurs afin de permettre l'échange de différents types de données entre les modules, éventuellement avec des protocoles d'interactions sophistiqués. Un ensemble de connecteurs pré-définis est aussi proposé. Cet ensemble comprend les signaux pour échanger des valeurs avec une synchronisation événementielle, des tampons avec des sémantiques de remplissage et de vidage différentes pour des échanges de données asynchrones, et des mutex et des sémaphores pour la synchronisation. Enfin, SystemC permet l'utilisation des objets, qui ne font pas partie des éléments architecturaux, mais qui sont fournis pour aider la programmation de contrôles dynamiques. Les modules SystemC peuvent êtres actifs et/ou réactifs. Deux modèles d'exécution sont proposés pour les modules actifs : **sc-thread** associe un flot d'exécution classique à une opération implémentée dans un module alors que **sc-cthread** (clocked thread) associe un flot d'exécution synchrone, contrôlé par une base de temps gérée via une horloge présente dans le système. Les opérations d'un module peuvent aussi être réactives (**sc-method**), ce qui permet de les exécuter en réaction à l'occurrence d'un événement. Dans ce cas, une liste de ports d'entrée qui peuvent déclencher une réaction est spécifiée pour chaque opération réactive. Il est important de noter que les actions et les réactions sont associées aux opérations et non pas aux modules, ce qui permet aux modules d'encapsuler à la fois des opérations actives et réactives.

Les modules SystemC sont définis pour la modélisation des composants matériels. Par conséquent, ils sont instanciés une fois pour toute avant la simulation du système, et ne peuvent êtres détruits ou reconfigurés au cours de la simulation. Le cycle de vie d'un programme SystemC commence par une phase d'élaboration où tous les modules sont instanciés et interconnectés et continue par la simulation du système jusqu'à ce qu'un des modules fasse appel à la procédure de terminaison. En revanche, les objets qui ne font pas partie des éléments de modélisation architecturale, peuvent être instanciés et détruits dynamiquement. Alors que la première version de SystemC supportait uniquement un modèle de composition *plat*, certaines constructions ont été rajoutées à partir de la version 2 pour permettre la composition hiérarchique. Les modules peuvent donc contenir d'autres instances de module, et configurer leurs connections. Pour ce faire, ils doivent décrire cette partie de gestion de configuration dans une procédure pré-définie qui sera appelée pendant la procédure d'élaboration du système global. Le partage de module n'est pas permis ; une configuration de système en SystemC est forcément représentée sous forme d'arbre. Enfin, un support d'introspection est défini par SystemC pour découvrir, au cours de l'exécution, la configuration du système en prenant en compte les objets qui sont instanciés dynamiquement.

SystemC fournit deux éléments pour étendre le langage C++ afin d'y intégrer les concepts présentés ci-dessus. Premièrement, une librairie est fournie pour la définition des classes modèles et des procédures qui sont destinées à être utilisées par les programmeurs. Cette libraire comprend, entre autres, un fichier en-tête principal qui fournit les définitions des nombreuses macros qui doivent être utilisées pour définir et manipuler des modules, des ports, des interfaces, etc. Le deuxième élément consiste en une librairie runtime qui est destinée à être liée avec les programmes SystemC. Cette librairie fournit le point d'entrée des programmes SystemC et assure leur élaboration et leur simulation. Cette dernière fournit aussi un certain nombre d'outils qui permettent d'observer et de contrôler la simulation au cours de son exécution.

> *SystemC est largement utilisé pour la modélisation dans le domaine de la conception de systèmes électroniques en raison de ses atouts en ce qui concerne les outils de programmation, de débogage et d'analyse. L'absence de générateur RTL depuis des modèles SystemC empêche cependant son déploiement dans un processus complet de conception de circuits.*

2.3.7 Autres approches

Dans les sections précédentes, nous avons présenté les modèles pour l'adaptation logicielle proposant des apports différents en termes d'architecture. Cette section présente les spécificités d'autres

modèles méritant d'être soulignées.

Les propositions de Sun Microsystems dans le domaine des composants logiciels commencent avec les *Java Beans* [96] qui sont destinés à la mise en place des interfaces graphiques. Les composants Java Beans sont en effet un modèle de programmation spécifique contenant des métadonnées sur les événements auxquels ils répondent, et sont destinés à être assemblés. Les Java Beans constituent un pas en avant par rapport au langage Java en définissant un patron de conception pour y retrouver des composants logiciels. En effet, les Java Beans proposent de mettre en place des objets Java avec des interfaces bien définies, et de documenter les paquetages créés avec des informations contenues sous forme de Bean Infos. J. Sasitorn [97] apporte dans sa thèse deux contributions originales à l'approche composant. D'abord, il propose une analyse des techniques logicielles pour réaliser une plate-forme de composants pour les langages typés orientés objet supportant la généricité. Ensuite, il propose sa plate-forme CGEN qui permet l'héritage entre composants, l'analyse statique des types entre composants (la réflexion enlevait cette possibilité).

La plate-forme AESOP [98] est un système créé à l'université de Carnegie Mellon pour développer des environnements de conception orientée-style (appelés *Fables*). Il offre un dépot étendu de composants, de connecteurs et de configurations (pattern de conception). Ils peuvent être utilisés pendant la construction de support pour des `styles architecturaux`. AESOP offre également un éditeur graphique qui peut être spécialisé afin de visualiser différents styles d'architecture. Il permet la vérification de type et l'analyse statique utilisant une description CSP. Il y a aussi une option de génération simple de code reflétant les connecteurs architecturaux. Cependant, pour chaque type de connecteur, un programme C++ implémentant le code du connecteur doit être fait. Il n'y a pas de solution par contre pour avoir des connecteurs modulaires ou les rendre configurables.

La plate-forme Classages [99] est une proposition académique de langage qui s'insère dans le cadre de la programmation basée sur des interactions (IOP). L'implémentation actuelle du modèle Classages fournit un support complet pour le langage. Mais le downcasting nécessaire à l'implémentation des connecteurs et des pluggers et la duplication du code en raison de la mise à plat de l'héritage pour les opérations de mixage compliquent la mise en œuvre. Classages est un nouveau langage qui est conçu pour fournir une isolation forte des composants logiciels en maîtrisant leurs interactions dynamiques et statiques. Le modèle d'interaction dynamique proposé par Classages est proche de celui d'ArchJava. Il est basé sur des ports qui implémentent des communications bidirectionnelles. Le modèle d'interaction statique est plutôt original ; il utilise la composition de composants à grain fin (mixage de méthodes) pour remplacer plus clairement l'héritage classique. Parmi les contributions de ce modèle, nous tenons aussi à citer la distinction entre les interactions d'un composant avec ses éléments internes (domaine de possession) et externes (domaine de collaboration).

D'une part, Senthil [100] apporte deux contributions originales à l'approche composant. D'abord, il montre qu'il peut obtenir de meilleures performances et une moindre complexité du code en s'appuyant sur la notion d'*interface d'adaptation*. Cela réduit le couplage entre les composants en abstrayant les données échangées au niveau des ports de communication. D'autre part, Zdun [101] apporte deux contributions originales à l'approche composant. Il propose de modéliser des patterns [98] d'architecture en utilisant neuf constructions élémentaires décrivant les interactions et la structure des entités de l'architecture. Ensuite, il démontre l'existence de ces constructions élémentaires à partir de l'analyse des patterns d'architecture les plus utilisées (incluant les design patterns). Aussi, Fissi [102] apporte une contribution originale à l'approche composant. Il propose l'ajout d'une unité de déploiement gérant le cycle de vie des composants. La découverte et la sélection de service se fait par l'utilisateur et est ensuite mémorisée et réutilisée automatiquement (cache d'assemblage). Enfin, on peut citer également les travaux de Chefrour [103] qui portent sur un modèle de composant adaptatif dans ACEEL : l'adaptation est centrée sur le composant logiciel.

2.3.8 Synthèse et conclusion

Sasitorn [97] propose une analyse des techniques logicielles pour réaliser une plate-forme de composants pour les langages orientés objet supportant la généricité en la plate-forme CGEN. Il permet

l'*héritage entre composants*, l'*analyse statique de types* entre composants. Bruhn propose l'inspection structurelle et l'*observation des interactions entre les composants* par l'utilisation de callbacks. Il ne se base pas sur les modifications de l'implémentation des plates-formes d'exécution ni des containers et permet l'insertion de code (pour l'observation et la reconfiguration) de manière transparente pour le développeur. S. Robert *et al.*intègre la notion de connecteur dans CCM et capitalise l'expertise sur la gestion des interactions. Cela permet l'*indépendance d'un composant* par rapport à CORBA. Senthil [100] pour améliorer les performances et diminuer la complexité utilise la notion d'*interface d'adaptation*. Il s'agit d'abstraire les données échangées au niveau des ports de communication. Zdun [101] propose de modéliser des *patterns d'architecture* [98] en utilisant des constructions élémentaires décrivant les interactions et la structure des entités de l'architecture. Fissi [102] propose plutôt l'ajout d'une *unité de déploiement* gérant le cycle de vie des composants. La découverte et la sélection de services se font par une demande à l'utilisateur et sont mémorisées. SOFA 2.0 propose la reconfiguration dynamique d'architecture hiérarchique en s'appuyant sur un modèle du *contrôleur de composants*. Fractal propose un modèle de composant hiérarchique avec *partage de composants* entre plusieurs composants composites. Les composants sont aussi dotés d'un ensemble de capacités réflexives paramétrables (contrôleurs et intercepteurs). Enfin, la modélisation de connecteur se fait par l'intermédiaire de bindings composites qui sont des composants jouant le rôle de connecteurs.

Nous avons vu la première notion fondamentale constituant les systèmes adaptatifs. Nous voyons dans la section suivante la seconde notion plus liée à la distribution logicielle : celle de service.

2.4 Plates-formes à services et composition de services

Un service peut être décrit comme une fonctionnalité dont le comportement est défini de façon contractuelle. Dans cette approche, les entités fournissant une telle fonctionnalité ne sont découvertes qu'à l'exécution de l'application. Une application construite à partir de services est uniquement décrite par un ensemble de services.

Découverte de service et connexion. La découverte et la connexion aux fournisseurs de services n'a lieu qu'au moment de l'exécution de l'application. Ainsi, l'approche à services se focalise sur la description des services.

Un service n'appelle pas directement un autre service. Un fournisseur peut changer au cours de l'exécution de l'application. De nouveaux services peuvent être découverts à l'exécution. L'approche à service n'est pas nouvelle et ses concepts sont déjà présents dans les mécanismes de courtage des systèmes distribués [104]. La propriété la plus importante des architectures orientés service est leur flexibilité dans la manière de prendre en compte les adaptations dynamiques et la simplicité de l'intégration de nouveaux services. Ils sont aussi pertinents dans le domaine des systèmes distribués. Les services peuvent apparaître et disparaître du système et notifier au système qui peut recalculer les dépendances afin de remplacer le service manquant. Les services ne peuvent en effet pas envoyer des événements applicatifs aux autres services. Dans la suite, nous nous intéressons aux principales approches permettant la découverte de services : ContextBox, OSGi et JINI. Elles ont été choisies pour leurs différences en terme d'expression des services et de leur mise en œuvre en environnement changeant. Nous voyons ensuite d'autres approches apportant des idées importantes pour la découverte de services.

2.4.1 *ContextBox* : notion de service pour un modèle de composant

Le concept de contexte des composants JavaBeans a été introduit dans une spécification postérieure à celle du modèle de composant [105]. Ce concept a été introduit pour fournir des moyens de grouper des instances de JavaBeans dans des contextes d'exécution, pouvant à leur tour être organisés dans des hiérarchies, et pour permettre aux instances d'obtenir des services de la part du contexte qui les contient lors de l'exécution d'une application. Le domaine d'application de cette plate-forme est celui des applications centralisées pouvant être assemblées de façon visuelle et qui normalement supportent l'interaction avec un utilisateur. Dans le BeanContext, pour un service particulier, un seul fournisseur peut être enregistré dans le registre à un moment donné.

Le BeanContext est également un système centralisé. La spécification permet la création de hiérarchies de contextes et décrit la possibilité pour un contexte de déléguer une demande de service vers un contexte parent.

ContextBox apporte une contribution originale à l'approche service. Il résout les conflits de la prise en charge de la notion de découverte de services en s'appuyant sur la composition de composants JavaBeans.

2.4.2 OSGi

OSGI est une corporation qui se charge de définir et promouvoir des spécifications ouvertes pour réaliser la livraison de services administrés dans des réseaux résidentiels ou autres types d'environnements restreints (voitures, etc.). La spécification OSGI [106] définit une plate-forme de services Java non distribuée ainsi que des moyens permettant de réaliser le déploiement des fournisseurs et des demandeurs de services à l'intérieur de la plate-forme (appelée framework). Dans OSGI, les services sont livrés et déployés dans des unités logiques et physiques appelées bundles. Du côté physique, un bundle correspond à un fichier JAR contenant du code et des ressources ; du côté logique, un bundle correspond à un demandeur ou fournisseur de services (un bundle peut aussi

jouer les deux rôles en même temps). Le framework fournit des mécanismes d'administration permettant de réaliser l'installation, l'activation, la désactivation, la mise à jour et la désinstallation des bundles physiques de façon continue. L'activation ou désactivation d'un bundle physique résulte dans l'activation ou désactivation du bundle logique correspondant. Lorsque le bundle logique est actif, il peut publier ou découvrir des services et se lier avec d'autres bundles à travers un registre de services fourni par la plate-forme.

OSGI est un système centralisé avec un seul registre. La spécification OSGI prend en compte les aspects "sécurité" et définit aussi un standard pour la notion de d'architecture orientée service (LogService, HttpService, etc.). iPOJO [107] ajoute la notion de dispositif dans un modèle de composant orienté service. Beanome ajoute la notion de composant pour un système orienté services au-dessus d'OSGi.

De manière globale, OSGI accompagné d'iPOJO apporte une contribution originale à l'approche service. Le couple propose un modèle de composant orienté service gérant l'adaptation dynamique de la découverte et de la publication des services. Beanome apporte deux contributions originales à l'approche service. D'abord, il propose un modèle de composant hiérarchique et un exécutif s'appuyant sur les services OSGi en tant qu'unités de déploiement contenant les descriptions de composants. Ensuite, il permet le déploiement de graphes de dépendances qui représentent les applications.

2.4.3 *Jini* : services dynamiques en environnement changeant.

JINI [108] est une plate-forme de services distribués qui partage un grand nombre de concepts avec la plate-forme de courtiers de CORBA. JINI repose sur le langage Java et bénéficie particulièrement de la capacité de télécharger du code à travers le réseau. Par conséquent, il supporte le fait que l'objet de service soit envoyé sur le même emplacement où se trouve le demandeur de services (bien que la distribution reste possible si cet objet joue le rôle de souche - *stub*). Cette caractéristique différencie JINI des courtiers CORBA dans lesquels la communication entre demandeurs de services et objet d'implémentation se fait toujours à travers un appel à distance. JINI supporte de façon explicite les politiques de libération des objets de service par expiration de bail. Une autre caractéristique de JINI est que clients et fournisseurs de services doivent trouver un registre avant de commencer à interagir. Un registre peut être connu par une adresse fixe ou bien il peut être découvert à partir de la diffusion d'une demande. Les services dans JINI peuvent être organisés par groupes (par exemple, groupe service d'impression) et un registre peut héberger un groupe de services particulier. Bien que JINI supporte l'existence de registres multiples, il n'offre pas les mécanismes permettant aux registres de déléguer une demande de service ; au lieu de cela, les fournisseurs de services enregistrent leurs services dans de multiples registres.

Dans JINI, un service est décrit par une interface Java et un nombre variable de propriétés (objets de sous-classes de la classe **Entry**). La publication de services est réalisée à travers la méthode register. Lors de la publication, un fournisseur de services définit un temps de bail (angl. *lease*) pendant lequel le service sera disponible. Avant que ce délai termine, le fournisseur doit renouveler le bail pour éviter le retrait de ses services du registre. Pour retirer un service, le fournisseur peut attendre la fin du bail ou bien forcer son expiration.

La découverte des services se fait à travers la méthode **lookup** qui permet de spécifier un nombre maximum de réponses. JINI ne propose cependant pas de mécanisme flexible pour réaliser le filtrage des fournisseurs appropriés du côté du registre. Le critère pour déterminer si un service correspond à une demande est que les interfaces du service coïncident avec celles demandées et que les propriétés envoyées par le demandeur soient présents dans la description de service. Lors de la publication d'un service, le fournisseur inclut une référence vers l'objet de service et ce dernier est copié dans le registre. Quand un demandeur de service obtient une réponse du registre, il obtient à son tour une copie de l'objet de service. Par défaut, la politique de création est donc un objet par liaison, cependant si l'objet obtenu joue le rôle de souche (**stub**) envers un objet distant, la politique de

création devient un objet partagé, car tous les demandeurs interagissent alors avec le même objet distant. JINI fournit des mécanismes de notification asynchrones permettant aux clients d'être informés des changements au niveau des services :

- TRANSITION_NOMATCH_MATCH signale l'arrivée d'un service,
- TRANSITION_MATCH_NOMATCH signale le départ d'un service,
- TRANSITION_MATCH_MATCH signale un changement au niveau d'un service (par exemple la valeur de ses propriétés). Un client doit s'enregistrer auprès du registre à travers la méthode notify.

JINI est un système distribué qui supporte l'existence d'un nombre indéfini de registres de services et prend en compte des contrôles tels que la sécurité et les transactions.

> *Jini apporte deux contributions originales à l'approche service. D'abord, il propose un ensemble de services ayant des attributs comme la localisation, par exemple, qui s'adaptent en changeant de Proxy, c'est-à-dire l'implémentation des interfaces de service, selon l'état courant des dispositifs et selon la topologie des services, ce qui constitue une implémentation évolutive et réutilisation d'anciens dispositifs.*

2.4.4 Composition de services

Nous allons voir dans un premier temps les mécanismes de base d'exécution à distance. Nous voyons dans un second temps les travaux sur la composition de services : le contrôle de l'exécution à distance.

Les `mécanismes d'exécution à distance` [109, 110, 111] reposent sur quelques patrons de conception (*design patterns*) dont les principaux sont les Proxy, les Factory, les Pools, les Adapters, les Interceptors. Cependant, les Pools et les Factories créent des instances distantes d'objets applicatifs et dans le cas des Pools à réduire le coût de la création d'objets. Ceux qui nous intéressent le plus ce sont les Proxys, Adapters et Interceptors qui ont d'étroites relations mutuelles puisqu'ils reposent tous les trois sur un module logiciel inséré entre le demandeur et le fournisseur d'un service.

	Adapter	**Proxy**	**Interceptor**
Préserve l'interface	Non	Oui	Oui
Accès distant	Local	Souvent	Souvent
Modifie un service existant	Oui	Non	Oui
Transforme l'implémentation	Non	Non	Oui

Tableau 2.1 – Mécanismes d'exécution à distance pour une construction hiérarchique de composants en IAm

Le tableau 2.1 présente une synthèse des similitudes et des différences entre les patrons dans le cadre de leur utilisation spécifique en IAm. Lorsque l'on dit que l'on modifie un service existant, cela signifie que soit on ajoute de nouvelles capacités à un service existant, soit on fournit le service par un moyen différent.

Les unités d'assemblages sont réparties et communiquent au moyen d'appels à distance : un client demande un service fourni par un composant distant. Dans notre architecture, il s'agit de définir un mécanisme d'accès qui n'implique pas de coder en dur l'emplacement du servant au niveau du code du client, et qui ne nécessite pas une connaissance détaillée des protocoles de communication par ce dernier. L'accès doit être efficace à l'exécution. La programmation doit être simple pour le client. La principale contrainte est que le client et le serveur puissent se situer dans des espaces d'adressage différents. Par conséquent, la solution consiste à utiliser un représentant local des unités d'assemblage sur le site du client. Ce représentant a exactement la même interface que le servant. Toute information relative au système de communication est cachée par le représentant.

La structure interne d'un Proxy suit un schéma bien défini :

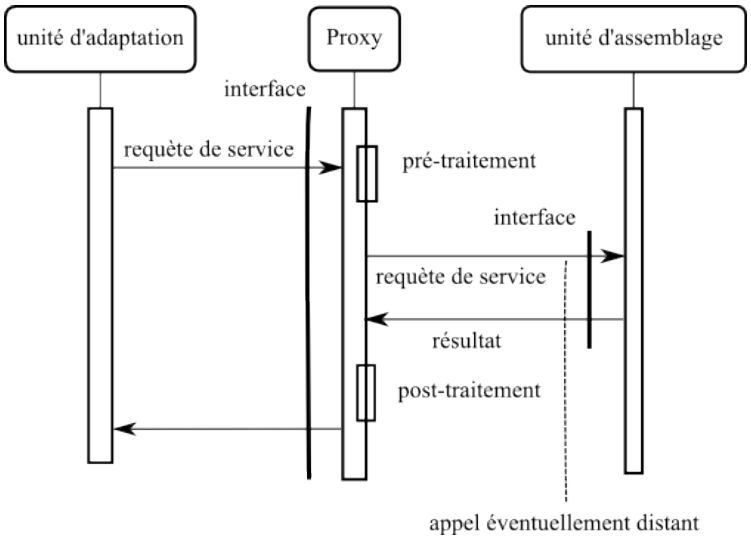


FIGURE 2.5 – Le patron Proxy

- Une phase de pré-traitement, qui consiste essentiellement à emballer les paramètres et à préparer le message de requête,
- L'appel effectif du servant, utilisant le protocole de communication sous-jacente pour envoyer et pour recevoir la réponse,
- Une phase de post-traitement, qui consiste essentiellement à déballer les valeurs de retour.

Dans les sections précédents, nous avons présenté les modèles de service pour l'adaptation logicielle proposant des apports différents en termes d'architecture et de fonctionnalité. Ce paragraphe présente les spécificités d'autres modèles méritant d'être soulignées.

AO4BPEL [112] apporte deux contributions à la combinaison de services. D'abord, il permet d'exprimer la composition de services sous la forme de modules. Ensuite, il permet d'effectuer la modification dynamique de la composition de services. JAsCo [113] y apporte aussi deux contributions. Il définit des connecteurs transverses permettant d'exprimer la composition des services et offre la possibilité de contrôler la combinaison des connecteurs transverses par l'utilisation de la précédence et des stratégies dans JAsCo. Self-Serv [114], quant à lui, propose une composition déclarative basée sur les diagrammes à état. La sélection des services se fait par le concept de communauté de services qui choisit le fournisseur le plus approprié. Les travaux sur JAC permettent d'exprimer la combinaison des services en UML en s'appuyant sur l'AOP [115]. Enfin, Verheecke [116] s'appuie sur la plate-forme JAsCo précédemment citée pour effectuer la sélection dynamique et l'intégration de services en utilisant également WSML [8].

2.4.5 Synthèse et conclusion

Au niveau des modèles à services, ContextBox résout les conflits entre la notion de découverte de services en décomposant la notion de découverte de services sous la forme d'*assemblage de composants* JavaBeans. iPOJO [107] propose un modèle de composant orienté service gérant l'*adaptation dynamique* des mécanismes de découverte, d'invocation et de publication des services. Beanome apporte deux contributions originales à l'approche service en proposant un modèle de *composant hiérarchique et un exécutif* s'appuyant sur les services OSGi (unité de déploiement contenant les descriptions de composants). Ensuite, il permet le déploiement de graphes de dépendances (qui représentent les applications). Jini met plutôt en relief la notion d'*attributs de services* pour prendre en compte la localisation pour la gestion de la dynamicité.

8. Web Service Modeling Language

Au niveau de la composition de services, AO4BPEL [117] permet d'exprimer la composition de services sous la forme de *modules*. Ensuite, il permet d'effectuer la *modification dynamique de la composition de services*. JAsCo [113] apporte deux contributions à la combinaison de services en définissant d'abord des *connecteurs transverses* permettant d'exprimer la composition des services. Ensuite, il offre la possibilité de contrôler la combinaison des connecteurs transverses par l'utilisation de la *précédence* et des stratégies dans JAsCo. Self-Serv [114] apporte une contribution originale à la combinaison de services avec la notion de *composition déclarative basée sur les diagrammes à état*. La sélection des services se fait par le concept de communauté de services qui choisit le fournisseur le plus approprié.

Enfin, nous voyons la dernière notion fondamentale qui se trouve maintenant liée à l'adaptation : la programmation orientée aspect.

2.5 Plates-formes utilisant des aspects et gestion des interférences

La programmation par aspect est de plus en plus utilisée par les travaux sur l'adaptation. En effet, les aspects permettent de séparer les préoccupations au niveau de l'implémentation fonctionnelle et l'implémentation pour l'adaptation. Cependant, nous constatons que les aspects sont rarement indépendants les uns des autres [22]. Ainsi, le problème des `interactions entre aspects` prend de l'importance. Début 2002, il n'y avait presque aucun support pour la détection et la résolution de telles interactions. Le programmeur est responsable de l'identification de ces interactions entre les aspects en conflit en proposant un code de résolution de ces conflits. Nous analysons les solutions proposées pour permettre leur analyse et la résolution des conflits. Des méthodologies à base d'automates et de règles logiques ont été proposées dans la littérature du domaine. Nous analyserons ces méthodes, en particulier la méthode fusionnelle qui permet d'amoindrir cette dépendance tout en gardant une expressivité générale des spécifications des adaptations. Nous n'étudions que les aspects liés aux composants et non pas, par exemple, directement AspectJ [118], CaesarJ [119], Arachne [120] pour le C.

Aspect. Un aspect est une abstraction qui définit une structure et un comportement qui se superposent à l'application.

Point de coupe. Un point de coupe (appelé aussi coupe) désigne un ensemble de points de jonction de l'assemblage à adapter où l'aspect va intervenir.

Point de jonction. Un point de jonction (appelé aussi point de jointure) est un point du flot d'exécution de l'application qui peut être contrôlé et où les aspects peuvent intervenir (invocation de méthode, envoi de message, etc.).

Greffon. Un greffon (appelé aussi Angl. *advice*) est une structure d'aspect permettant de décrire les actions à effectuer lorsqu'un point de coupe est identifié.

La `programmation orientés aspect` [21] consiste donc en un ensemble de points de jonction, de points de coupe, de greffons et une boucle de tissage dont le rôle est de calculer, pendant l'exécution de l'application, les modifications à effectuer. La tendance consiste à considérer les adaptations comme des composants transversaux tissés comme des aspects classiques. Par exemple, David et Ledoux [121] ont conçu un DSL [9] et exprimé des préoccupations d'adaptation en tant que composants aspectuels afin de monitorer des systèmes auto-adaptatifs. Ils ont aussi proposé d'exprimer des points de coupe en terme de script de connexion. Cependant, cette approche n'offre pas une combinaison collaborative et ne permet pas d'éviter les conflits sémantiques en déclarant des connexions. Les langages d'aspect généraux respectent cette séparation entre le "où" et le "comment". Le modèle de points de jonction définit l'ensemble des points de l'exécution visibles par les aspects. Ces points de jonction comportent une partie statique et une partie dynamique. Le langage de points de coupe permet de choisir un sous-ensemble des points de jonction (motifs, opérateurs logiques,...) pour définir des ensembles de points de jonction. Le langage de greffon offre des mécanismes pour ajouter du comportement aux points de coupe à l'aide des directives **after**, **before** et **around**. Un greffon **before** (respectivement after) est exécuté avant (respectivement après) le point de coupe. Ces aspects généraux reposent sur des transformations d'ordre plus syntaxique que sémantique et donc obligent à raisonner sur le code tissé.

Dans la suite, nous nous intéressons aux principales approches orientées aspect permettant le tissage dynamique d'aspects (ajout et retrait dynamique d'aspects) à partir de modèles de composant : FAC et SAFRAN. Elles ont été choisies pour leurs différences en terme d'expression des aspects et pour leur mise en œuvre de la composition d'aspects.

9. langage dédié spécifique à un domaine

2.5.1 *FAC* : sûreté d'utilisation d'aspects dans un modèle de composant

FAC est un framework pour l'adaptation logicielle [122]. La plate-forme se focalise sur un paradigme orienté composant incluant un ADL. Son modèle inclut trois niveaux d'adaptation. Son argumentation tient du fait que les modifications fonctionnelles ou non-fonctionnelles demandent divers types d'outils d'adaptation travaillant à différents niveaux de granularité et à différentes étapes du le cycle de vie du système ayant des paramètres variés.

Dans un modèle de composant doté d'un ADL, nous voulons adapter les points d'accès (ports des composants), les connecteurs entre des composants hétérogènes, la composition d'un composant composite ou des programmes qui représentent le comportement du composant. Ces travaux identifient trois niveaux distincts d'adaptation, chacun d'entre eux travaille à une granularité différente :

1. L'architecture
2. Le composant
3. L'implémentation

FAC est une extension du modèle de composant Fractal. Il y intègre la notion d'aspect. Le but est de capturer les propriétés transversales du système. Dans FAC, les aspects sont représentés par des composants Fractal. Ils sont appelés des composants d'aspect (*Aspect Components* ou AC). Ils sont dotés d'une interface serveur spécifique qui implémente l'API AOP Alliance. Les points de coupe sont définis à travers des expressions régulières au niveau du connecteur. Un point de coupe sélectionne les composants, les interfaces et les méthodes sur lesquels l'AC sera appliqué. FAC permet des points de coupe structurels ou comportementaux. Un AC est tissé et détissé à l'exécution. Le processus de tissage est très similaire au processus de connexion entre une interface client fonctionnelle et une interface serveur dans Fractal. Est appelé connecteur, cette interaction entre un ensemble de composants et un AC. Ces connecteurs *transverses* sont ainsi définis par une API ou à travers un niveau d'ADL. FAC travaille sur plusieurs niveaux et précisément sur plusieurs paradigmes (composants, langages). Certains problèmes ont été soulevés comme celui de la cohérence entre les différents niveaux d'expression. Appliquer les modifications au moment de l'exécution ou au moment de la conception représente également un problème. Appliquer des modifications à l'exécution en utilisant des transformations requiert des précautions particulières.

> *FAC offre cinq contributions originales à l'approche aspect. D'abord, il définit un aspect encapsulé dans un composant logiciel. Il définit un ordre d'exécution des greffons au niveau de chaque point de jonction. Ensuite, il rend possible l'extraction des préoccupations transverses. De plus, il permet la sûreté de l'intégration des aspects par le contrôle des aspects agissant sur un point de jonction par une interface particulière de tissage. Enfin, il autorise le contrôle de l'intérieur des composants composites.*

2.5.2 *Safran* : composants adaptatifs et aspect d'adaptation

SAFRAN [123, 121] (Self-Adaptive Fractal CompoNents) est une plate-forme `auto-adaptative` à composants Fractal. C'est un système qui étend Fractal afin de faciliter le développement d'applications adaptatives. SAFRAN considère l'adaptation d'une application à son infrastructure de dispositifs sous la forme d'un aspect démontré dans l'article de David et Ledoux [121] qui est appelé `aspect d'adaptation`. Ce dernier doit être développé séparément du reste de l'application en utilisant un formalisme spécifique. Il peut ainsi être intégré (ou retiré) dynamiquement du système. P.-C. David propose aux programmeurs d'applications un modèle de développement et des outils qui facilitent la création d'applications adaptatives. L'adaptation considérée est la reconfiguration volontairement restreinte de l'application aux attributs de configuration et à la structure de l'application pour pouvoir offrir des garanties de cohérence. L'architecture de SAFRAN se divise en deux parties : une extension de Fractal et un système de programmation des politiques d'adaptation.

Avant de décrire l'architecture et le fonctionnement de ce modèle d'architecture logicielle, nous proposons de définir les notions-clés de ce modèle qui sont en outre nécessaire à sa compréhension.

Une action de reconfiguration est un programme écrit dans le langage procédural dédié comme FScript [124] permettant de reconfigurer une plate-forme à composants dynamiques. Il donne accès aux opérations de création de composants, d'introspection et de reconfiguration d'architecture en manipulant le contenu des composites et les connexions entre les interfaces. Un événement endogène est un terme consacré à SAFRAN et correspond à un `point d'exécution` dans le `programme de base` qui est un ensemble de composants Fractal. Un `point d'exécution` correspond soit à l'invocation de messages des `interfaces Fractal`, soit aux reconfigurations possibles des composants Fractal (création de composants, démarrage et arrêt, modification du contenu et manipulation des connexions). Un événement endogène est une `réification` des informations implicites de l'infrastructure de dispositifs de l'application par l'utilisation d'une plate-forme (WildCAT [123]) pour le développement d'applications sensibles au contexte.

Une coupe est la description de la notification d'un événement endogène ou exogène. Sa définition coïncide avec celle des aspects académiques, car elle représente également un ensemble de points de jonction, mais qui correspondent de manière originale à l'occurrence d'événements provenant du contexte ou de l'application. C'est une extension du modèle théorique de Douence *et al.* [22]. Ces définitions sont utilisées, d'une part, pour la déclaration de l'adaptation qui constitue l'architecture du modèle SAFRAN et d'autre part, pour son association et son exécution qui constituent la description fonctionnelle du modèle. SAFRAN reprend le schéma générique d'un aspect. Une adaptation logicielle y est exprimée sous la forme d'un ensemble de déclarations dont un `programme de base` correspondant à une architecture à base de composants Fractal, des coupes correspondant à la notification d'événements endogènes ou exogènes et des actions restreintes à la reconfiguration de l'architecture. Enfin, est appelé aspect d'adaptation, le regroupement composé des coupes, des actions et des politiques d'adaptation modulaires tissées dynamiquement grâce à un nouveau contrôleur Fractal.

La plate-forme Fractal a du être étendue afin de pouvoir associer dynamiquement des politiques d'adaptation aux composants d'une application. Comme toute extension de Fractal, celle-ci se traduit par la spécification d'une nouvelle interface de contrôle : le `contrôleur d'adaptation`. Ce mécanisme d'association dynamique constitue le mécanisme de `tissage` de l'AOP. Le contrôleur d'adaptation implémente le tissage des politiques d'adaptation. Il permet ainsi d'ajouter la capacité d'adaptation à certains composants. Par conséquent, nous pouvons distinguer deux familles de composants : les composants Fractal non-adaptatifs et les composants Fractal adaptatif. Là où un composant non-adaptatif permet à un acteur externe de le reconfigurer, un composant adaptatif intègre grâce à cette interface le code d'adaptation lui-même et devient ainsi adaptatif, c'est à dire `autonome` par rapport à sa propre adaptation. Lorsqu'une politique d'adaptation est attachée à un composant, son contrôleur d'adaptation l'analyse et, en fonction des `coupes` spécifiées dans les règles, instrumente le composant cible pour générer les `événements endogènes` appropriés et s'enregistre auprès de WildCAT afin d'être notifié de l'occurrence des `événements exogènes`. Après cette phase d'initialisation, lorsque le contrôleur reçoit un événement, qu'il soit endogène ou exogène, il détermine la réaction appropriée en fonction de l'ensemble courant de politiques et de règles du composant cible, puis exécute cette réaction afin d'adapter le composant aux nouvelles circonstances. Ce schéma d'exécution correspond à la nature réactive du processus d'adaptation, en reprenant les trois phases : observation, décision et action.

Ce modèle montre comment les principes de la programmation par aspects peuvent être utilisés pour faciliter la construction d'applications adaptatives. Sur le plan conceptuel, P.-C. David a montré que l'adaptation peut être considérée comme une préoccupation transverse et que, par l'étude d'un cas particulier, la transposition des concepts de la programmation par aspects (programme de base, coupes, actions et tisseur) pour modéliser l'adaptation est possible. Il étend également la notion de point de jonction, par rapport au modèle événementiel de Douence, en y intégrant les `événements exogènes`. Par conséquent, il ajoute à l'adaptation la `prise en compte du contexte`. SAFRAN fait face à la réactivité par l'utilisation de règles ECA [125] similaires à celles des bases

de données actives [126]. Les principes de l'AOP [127] introduisent cette réactivité de façon non-invasive. Les conflits entre les adaptations n'est pas doté d'outils d'analyse du comportement des règles. Leur cohérence définie par le contrôle des `interactions` [22] (c'est-à-dire, en conflit) entre les règles est à la charge du programmeur. Au niveau de la dispersion des préoccupations, une politique n'adapte un composant qu'individuellement (même si plusieurs composants peuvent chacun être affectés par une instance distincte d'une même politique). Absence de mécanisme de gestion des aspects d'adaptation qui sont transverses à plusieurs composants. Le système est extensible. La reconfiguration et le tissage sont `dynamiques` et permettent donc une implémentation incrémentale au deux niveaux de représentation de l'application : architecture et adaptation. Enfin, l'adaptation apparaît comme une préoccupation **transverse** [10] au `code métier` que nous aimerions modulariser pour garantir une meilleure réutilisation et maintenabilité [121].

Selon P.-C. David, combiner les contrôleurs d'adaptation des composants affectés en les partageant au sein d'un même composite conviendrait au problème de la dispersion des préoccupations. En effet, nous pourrions alors spécifier une `action` qui confère à ces composants leurs `politiques d'adaptation`. Globalement, l'auteur démontre en outre dans ce modèle que les techniques de la `programmation par aspects` offrent un cadre de solution adéquat aux problématiques de la modularisation logicielle.

> *Safran propose trois contributions originales. Il permet l'ajout de deux sous-systèmes au-dessus du modèle de composant : un langage de script pour reconfigurer l'application avec la garantie de atomicité de l'opération, un toolkit pour observer le contexte de l'application. Il propose un contrôleur d'adaptation connectant ces deux entitées paramétrables par des règles ECA, événements détectés par WildCat et les actions entreprises par FScript.*

2.5.3 Gestion des interférences

Les interactions logicielles [128] sont des notions parentées aux aspects. Afin de montrer les limitations des approches classiques par aspects, nous introduisons de nouveaux concepts tels que : les points d'exécution d'entrée/sortie d'un greffon. Le point d'exécution d'entrée (PEE) d'un greffon constitue le point d'entrée d'un assemblage de composants constituant son greffon. Nous n'en dénombrons qu'un seul. Nous définissons les points d'exécution de sortie (PES) d'un greffon comme étant les appels à des services extérieurs au greffon. L'inversion de contrôle n'est effectuée que sur un PEE et le contrôle des PES demeure interne au greffon. Les mécanismes de coordination sont regroupés en deux catégories : arbitrage (subsomption, événements discrets, décision baysien, apprentissage, réseau d'activation) et fusion (logique floue, schémas moteur).

Le problème de superposition appelé aussi `interaction d'aspect` se décline dans plusieurs disciplines. Dans le domaine des composants logiciels, nous nous intéressons au problème de l'application simultanée de plusieurs assemblages de composants. Donnons un exemple dans le domaine d'étude des machines parallèles, le problème de superposition se résume par exemple à celui du modèle d'accès simultané à une case mémoire [129].

2.5.3.1 *EAOP* : Généralisation de la notion de point de coupe

Les premiers travaux de Douence [130] offre trois contributions originales à la détection et la résolution de conflits. D'abord, il offre un cadre formel d'expression des points de coupe. Ensuite, il définit deux propriétés générales d'indépendance : l'indépendance forte (points de coupe disjoints entre aspects) et l'indépendance par rapport à un programme (points de coupe disjoints utilisés par le programme). Il fournit l'analyse statique de ces propriétés. Enfin, il propose deux types de commandes de résolution de conflits : d'abord, la composition explicite des greffons au même point de

10. se dit de la description programmatique d'une fonctionnalité qui est dispersée dans un programme structuré et qui est partagée par plusieurs encapsulations

jonction (a. séquence, b. sélection d'un unique aspect, c. séquence arbitraire décidée par l'analyseur et le contrôle de la visibilité des aspects entre eux par encapsulation d'aspects.

Avec EAOP, Douence définit une machine à état fini et des priorités pour gérer la superposition des schémas [130]. La plupart des formalisations de la programmation par aspects (calculs minimaux, sémantiques) prennent en compte des langages d'aspects généraux sur lesquels il est difficile de raisonner. Dans la thèse [131], une approche formelle est adoptée. Les aspects peuvent être en effet vus comme des propriétés sur les traces d'exécution du programme. Son approche a l'avantage de permettre de raisonner à la fois sur l'application de base et sur les aspects. Sa formalisation repose sur les automates pour représenter l'application de base et les aspects. Le tissage est défini comme une composition d'automates et peut être vu comme une restriction du comportement de l'application selon la propriété. Dit autrement, l'aspect spécifie l'ensemble des traces d'exécution autorisées. Ce cadre pour la programmation par aspects a été appliqué à différents domaines dont l'ordonnancement de programmes parallèles.

Les aspects dans ce modèle sont `régulatifs` : ils ne changent pas les variables de l'application de base et peuvent seulement sélectionner ou couper une trace d'exécution de l'application. C'est une catégorie d'aspects qui garantit que les propriétés, par exemple, de sûreté de l'application de base et les invariants sur les états sont préservés dans l'application tissée. Cette catégorie d'aspects garantit aussi que si le programme tissé termine normalement alors il retourne les mêmes résultats que le programme de base. Cette approche a aussi l'avantage que les `interactions` entre les aspects sont maîtrisées comme le tissage est `associatif` et `commutatif`. Il est possible de construire un aspect représentant les autres aspects en les combinant par produit et de le tisser de la même manière.

Cette approche prend en entrée un programme et un aspect, et produit automatiquement un programme tissé respectant l'aspect. La composition se fait automatiquement par produit des automates. Par exemple, le premier automate permet de reconnaître les mots **ddacabc** et **abcabc**. Le second automate permet de reconnaître les mots **ccab** et **abcabc**. Le produit des deux automates permet de reconnaître uniquement le mot *abcabc* parmi les mots ci-dessus. Ces travaux sont applicables en effet pour une catégorie spécifique des aspects. Il ne permet cependant pas de traiter les aspects dans leur généralité.

> *EAOP offre trois contributions originales à la détection des conflits. D'abord, il définit l'aspect en termes de ticks provenant de l'exécution de l'application. Ensuite, il propose des points de coupe rattachés à des séquences de ticks avec ou sans sauvegarde d'un état. Enfin, il associe une action à la détection d'un point de coupe.*

2.5.3.2 *ISL* : interactions en environnement distribué, compilé et typé

Dans un modèle orienté objet, les *interactions logicielles* [30] modélisent les *interactions entre appels de méthodes* entre différents modules d'un programme logiciel. ISL (sigle signifiant Interaction Specification Language – langage de spécifications d'interactions) est un langage de spécification d'interactions entre des objets logiciels. Il a été développé en 2001 par Berger au laboratoire I3S et permet de fusionner des spécifications écrites dans ce langage afin de modifier la structure de programmes orientés objet. Un appel de méthode est la transmission d'un flot de contrôle entre objets accompagnée d'un passage par paramètre de données. Cette étude sur les `interactions logicielles` concerne l'ajout de nouvelles fonctionnalités en terme de réécriture de réception d'appel de méthodes. Ces ajouts s'effectuent au cours de l'exécution de l'application, par conséquent, lors de l'ajout de nouvelles fonctionnalités, de manière sûre. L'étude de la sûreté de l'application lors de l'ajout de ces fonctionnalités a été réalisée dans [132] consistant à vérifier que les modifications se font en respectant l'état de l'application. Nous ne traitons pas de la sécurité des interactions dans cette thèse, mais nous étudions les mécanismes permettant de reconfigurer un plus large spectre d'applications au cours de leur exécution en recherchant les mécanismes essentiels de la reconfiguration dynamique.

En résumé, l'objectif général du travail de Berger était de faciliter "l'adaptabilité dynamique des codes" pour des applications logicielles complexes telles que les frameworks et les applications réparties. L'adaptabilité dynamique consiste à ajouter et enlever dynamiquement des contrôles dans le but d'accomplir une adaptation fonctionnelle. En prenant l'exemple typique de l'agenda, nous pouvons associer l'adaptation fonctionnelle à l'exemple de l'ajout de la fonctionnalité d'authentification à la communication entre deux agendas différents. La solution traditionnelle consiste à analyser le problème et mettre en œuvre une solution à travers l'utilisation de design patterns, de méta-programmation, d'AOP (Event service, RelationShip Service, DII, Interceptors, Bus reflexif). Les interactions se retrouvent enfouies dans le code de l'application. Ainsi, la solution de L. Berger consiste à faire émerger dans des fichiers extérieurs au code ces interactions. Il propose la mise en place d'un langage de spécification d'interactions (ISL) pour cette occasion. Ainsi pour reprendre l'exemple de l'agenda, deux interactions peuvent être définies pour les deux agendas de la manière suivante :

```
agenda1.addRendezVous(rdv) ->
  call; if rdv.concerne(P) then agenta2.addRendezVous(rdv) endif
```

FIGURE 2.6 – Agenda

Cette première interaction spécifie qu'au lieu d'effectuer le comportement originel de la méthode add de l'objet **Ag1**, nous effectuons plutôt séquentiellement l'action d'ajouter un rendez-vous **Ardv** sur un certain agenda **Ag1** et puis celle de vérifier si le rendez-vous (Ardv) concerne bien la personne (P) et si tel est le cas, alors le rendez-vous est également ajouté à l'agenda Ag2.

```
agenda.addRendezVous(rdv) -> if controleur.accept() then call endif
agenda.setPasswd() -> key.setPasswd()
```

FIGURE 2.7 – Agenda avec mot de passe

Dans l'exemple précédent, nous retrouvons deux règles d'interaction. La première règle signifie que le comportement de la méthode add de l'objet agenda est réécrit de la manière suivante : si le contrôleur d'accès C l'accepte, alors nous ajoutons le rendez-vous (Ardv) dans l'agenda. La deuxième règle redéfinit le comportement de la méthode setPasswd de l'objet Agenda en spécifiant d'appeler plutôt cette même méthode sur l'objet Key. Ces règles d'interactions sont regroupées dans un schéma d'interaction qui a l'allure suivante :

```
Interaction teamDiary2Diary(Diary TD, Diary D, Person P) {
  TD.addRendezVous(RDV aMeeting) ->
    call ; if aMeeting.concern(P) then
    try { D.addRendezVous(aMeeting) }
    catch NotFree { TD.removeRendezVous(aMeeting) }
    end if,
  D.addRendezVous(RDV aMeeting) ->
    call || if not TD.contains(aMeeting) then
      TD.notify(aMeeting)
    end if
}
```

FIGURE 2.8 – Interaction

ISL offre six contributions originales. D'abord, il définit un langage spécifique pour décrire les interactions c'est-à-dire les relations entre les entités logicielles. Ensuite, il définit la notion de schémas d'interaction qui sont des descriptions des comportements réactifs d'un ensemble d'interactions et les interactions sont dissociées des classes et des objets. Ensuite, il permet la modification des interactions à l'exécution de l'application. Ensuite, il définit la notion de dépôt de schémas d'interactions. Enfin, il permet l'envoi dynamique de messages pour des environnements compilés et fortement typés. Enfin, il conserve la sémantique par combinaison des comportements réactifs indépendants. C'est la fusion comportementale.

2.5.4 Autres approches

Dans les sections précédentes, nous avons présenté les modèles pour l'adaptation logicielle proposant des apports différents en termes d'architecture. Cette section présente les spécificités d'autres modèles méritant d'être soulignées.

Plates-formes

Bockish [133] offre une contribution originale à l'approche aspect. Il permet (1) la construction d'un méta modèle (ensemble de concepts) existant à l'exécution de l'application, (2) de conserver la notion d'aspect et (3) l'application des optimisations déjà existantes mais statiques lors du tissage d'aspects pendant l'exécution de l'application.

Ubayashi [134] offre une contribution originale à l'approche aspect en séparant l'interface d'un aspect (weaving interface) de l'implémentation d'un aspect afin de simplifier la composition des aspects sans se soucier de son implémentation.

Les travaux autour de Caesar [135] proposent une approche en découplant l'interface de l'implémentation des aspects. Un aspect devient un composant que l'on assemble. D'autres travaux comme ceux sur FuseJ [136] et Hyper/J [137] proposent une unification naturelle d'aspect et de composant. Ils l'appellent approche `symétrique`. Plusieurs travaux proposent l'application de l'AOP mais sans prendre en compte la décomposition dominante avec les composants [138, 135]. Enfin, certains proposent de conserver une trace de l'abstraction des aspects en intégrant deux intructions supplémentaires à l'exécution : l'association et la dissociation de pattern de point de jonction (ex : bytecode) dans Nu [139]. D'autres proposent un métamodèle comme Reflex [140] pour structurer l'adaptation par aspect.

COOL et RIDL ont été proposés dans l'article de Lopes *et al.* [141]. Dans ce travail, une application distribuée est constituée d'un ensemble d'objets distribués et concurrents qui communiquent par des appels à leurs méthodes partagées. Ces deux langages d'aspect dédiés facilitent le développement des systèmes distribués en séparant le code de base de l'application, qui est décrit dans un langage restreint sans instruction pour gérer les préoccupations transverses liées au cadre distribué. Le langage COOL permet au programmeur d'exprimer la coordination entre les différents processus en spécifiant les contraintes pour appeler les méthodes partagées. Ce langage permet de spécifier les exclusions entre les différents appels de méthodes avec les deux constructions dédiées *self-exclusif* et *mut-exclusif*. Le langage RIDL fournit les moyens de paramétrer pour chaque méthode la manière dont les données (arguments des méthodes) sont effectivement transmises (par copie ou par référence). Par comparaison, le langage Java permet de définir pour chaque classe comment les attributs sont effectivement transmis (référence ou copie) ; il est possible d'avoir une spécification par méthode, mais de manière transverse et ad hoc (en décrivant plusieurs classes héritées fictives, et en spécifiant pour chacune de ces classes la façon de la transmettre, puis en forçant le type de l'objet). Chaque méthode inclue dans l'ensemble "selfexclusif" ne peut pas être exécutée par plusieurs processus en même temps. Les méthodes appartenant au même ensemble "mutexclusif" ne peuvent pas être exécutées en concurrence. Ce langage permet aussi de décrire des conditions qui bloquent

l'entrée dans une méthode tant qu'elles ne sont pas respectées. Ces conditions sont spécifiées par la construction **require** et reposent sur des variables qui définissent l'état de synchronisation de l'objet. Les évolutions de l'état de synchronisation sont uniquement possible en entrée ou en sortie d'une méthode, et sont décrites par les constructions **on entry** et **on exit**. Une autre difficulté dans les programmes distribues réside dans la transmission des paramètres lors d'un appel de méthode. Les solutions triviales de transmettre la référence de l'objet ou une copie complète de l'objet sont inefficaces.

Nous pouvons citer le langage unifié pour l'adaptation et l'AOP/DAJ [142] : c'est une extension d'AspectJ afin d'intégrer les concepts de Demeter (diagramme de classes, stratégie, visiteur) pour enrichir les mécanismes d'adaptation logicielle. Enfin, Lämmel [143] propose d'améliorer la programmation générique en apportant une modélisation par stratégies qui sépare deux préoccupations de l'application : les données et le contrôle qu'il regroupe dans deux ensembles d'aspects disjoints.

Gestion des interférences

Au niveau de la gestion des interférences, les travaux de Moreira [144] reposent sur un modèle orienté préoccupation. Ensuite, la détection de conflit direct entre aspects est effectuée par Tessier [145]. Enfin, d'autres proposent l'ajout d'annotations d'aspect pour la détection de conflits [146].

Zambrano [147] travaille sur les interférences entre les aspects qui ne sont pas d'ordre syntaxique. Il donne les exemples tels que la minimisation de l'utilisation de la mémoire par rapport à l'optimisation de la batterie. En effet, il n'y a aucun point de jonction en commun entre ces aspects mais ils sont quand même en conflit. Il appelle cela des conflits de comportement inconsistant, faisant ainsi écho à la taxonomie de Tessier *et al.* [145]. Ce que l'on retient c'est la manière dont Zambrano pose des étiquettes sous la forme de méta-données sur un aspect. Cela permet d'abstraire l'étiquette de l'aspect. Il offre ainsi deux contributions originales à la détection de conflits. D'abord, il permet l'étiquetage sous la forme d'ajout de métadonnées aux aspect. Ensuite, il définit un coordinateur qui scrute des ressources et active ou désactive des aspects selon les informations indiquées par les étiquettes.

2.5.5 Synthèse et conclusion

Bockish [133] propose la construction d'un méta modèle de l'application en *conservant la notion d'aspect à l'exécution.* Des optimisations lors du tissage d'aspects peut alors se faire pendant l'exécution de l'application. FAC [122] définit un *aspect encapsulé* dans un composant logiciel et un *ordre d'exécution* des greffons au niveau de chaque point de jonction. Ensuite, il rend possible l'*extraction des préoccupations transverses.* Enfin, la sûreté de l'intégration des aspects par le contrôle des aspects agissant sur un point de jonction par une interface particulière de tissage peut être mise en place. Safran ajoute deux sous-systèmes au modèle à composants : un *langage de script* pour reconfigurer l'application, un toolkit pour *observer le contexte de l'application.* Ensuite, il propose un contrôleur d'adaptation connectant ces deux entités paramétrables par des règles ECA (événements détectés par WildCat et les actions entreprises par FScript). Ubayashi offre une contribution originale permettant la séparation de l'interface d'un aspect (*weaving interface*) et de l'implémentation d'un aspect afin de simplifier la composition des aspects sans se soucier de son implémentation.

Au niveau de la détection de conflit, EAOP définit l'aspect en termes de *ticks* provenant de l'exécution de l'application. Il associe une *action à la détection* d'un point de coupe. ISL définit un langage spécifique pour décrire les interactions (relations entre les entités logicielles). Ensuite, il définit la notion de *schémas d'interaction* (description des comportements réactifs d'un ensemble d'interactions) et les interactions sont dissociées des classes et des objets. Ensuite, il permet la modification des interactions à l'exécution de l'application. Ensuite, il définit la notion de *dépôt de schémas* d'interactions. Enfin, il permet l'envoi dynamique de messages pour des environnements compilés et fortement typés. Enfin, il *conserve la sémantique par combinaison des comportements réactifs indépendants.* C'est la fusion comportementale. Zambrano permet l'étiquetage sous la forme

d'ajout de métadonnées aux aspects et définit un *coordinateur* qui scrute des ressources et active ou désactive des aspects selon les informations indiquées par les étiquettes.

Nous allons, dans le chapitre suivant, faire une synthèse de toutes les approches que nous avons vues dans cet état de l'art pour en dégager les problématiques non résolues sur lesquelles s'appuie notre travail.

Chapitre 3

Synthèse générale et objectifs

Sommaire

Dans notre environnement quotidien, nous voyons émerger des environnements constellés de dispositifs : capteurs, actionneurs, services. Ces dispositifs sont à priori indépendants, mais pourtant connectés les uns aux autres. On les qualifie parfois d'intelligents, car ils peuvent être impliqués dans des cas d'utilisations multiples : assistance, contrôle à distance des équipements, gestion du confort par le réglage automatique du chauffage en fonction de la température ambiante. Cette nouvelle façon de combiner les dispositifs et les applications a donné naissance à l'`intelligence ambiante` (page 11).

Le chapitre 2 explique les raisons pour lesquelles l'intelligence ambiante implique une technique de programmation différente de celles utilisées en informatique classique. En effet, les applications qui vont être exécutées à un instant donné vont être dynamiquement construites. Ensuite, les dispositifs physiques utilisés pour une application donnée peuvent être de nature différente, tout en fournissant une fonctionnalité équivalente. Enfin, le fait que tous ces systèmes doivent pouvoir fonctionner en situation réelle impose des contraintes sur l'économie d'énergie, la longévité des systèmes, etc, et justifie les configurations diverses autour d'une application principale.

La recherche en informatique ambiante a donc pour objectif de faciliter la mise en œuvre de tels systèmes et de fournir un modèle d'adaptation logicielle en prenant en compte ses contraintes.

3.1 Les contraintes de l'IAm

Nous dressons dans cette section un récapitulatif des différents enjeux en informatique ambiante. Un système informatique ambiant évolue dans un `espace ambiant` constitué d'une infrastructure de dispositifs et de l'environnement physique comprenant les utilisateurs, les objets de la vie de tous les jours, etc. Cet espace ambiant est caractérisé par deux grandes notions [148] : l'hétérogénéité des dispositifs et la variation permanente de l'infrastructure de dispositifs.

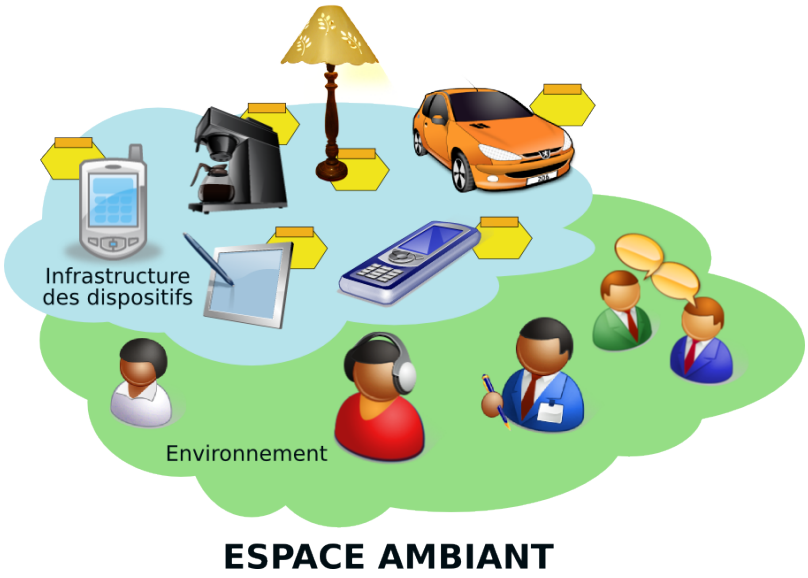


FIGURE 3.1 – Architecture des systèmes IAm

Hétérogénéité des dispositifs : Les utilisateurs interagissent avec les applications Internet à travers une variété de dispositifs dont les caractéristiques et les performances ont pu passer à l'échelle. Entre un PC très performant, un téléphone mobile et un PDA, les variations de la bande passante, la puissance de calcul local, les dimensions de l'écran, la capacité à afficher des couleurs sont extrêmement différents. Il était difficile de prévoir qu'un serveur puisse offrir un niveau d'accès aussi large et adapté à chaque client. Toutefois, imposer un format uniforme pour tous les clients

serait restreindre leurs fonctionnalités en les ramenant à un niveau commun (écrans textuels, noir et blanc, etc.).

Composants blackbox : Les anciennes applications (patrimoniales), par exemple, ne peuvent être utilisées qu'à travers leur interface spécifique et ne peuvent pas être modifiées. Dans plusieurs cas, le coût de la conception et de la réécriture d'une application patrimoniale est onéreux et ces applications ont besoin d'être intégrées sans modification.

Dynamiques propres à l'espace ambiant : L'application en informatique ambiante s'adapte aux variations permanentes de l'espace ambiant. Deux caractéristiques peuvent être déduites de cette variabilité. La première est la mobilité : l'application subit des variations permanentes, le système est dit réactif. La seconde est la nomadicité : l'espace ambiant s'ajuste à l'application, le système est dit interactif. Notre étude se situe dans le premier cas intégrant ainsi la notion de mobilité. C'est ce que nous appelons l'adaptation réactive.

Un système de communication approprié pour de telles situations est un système de bus de messages, c'est-à-dire un canal de communication commun auquel les différentes entités communicantes sont connectées. La communication est asynchrone et peut impliquer temporairement plusieurs participants. Il existe plusieurs possibilités pour savoir à qui envoyer un message, comme être membre d'un groupe prédéfini ou bien être souscripteur à un événement spécifique.

3.1.1 Grande diversité des dispositifs

Dans cette section, nous caractérisons les principes fondamentaux des systèmes adaptatifs ambiants et la problématique de l'évolution des interactions entre ordinateurs. En fournissant des exemples de cas réels d'utilisation, nous aboutissons au modèle d'architecture à deux étages. Nous proposons d'analyser une architecture à deux niveaux : un niveau machine et un niveau infrastructure. Nous étudions leur impact sur le niveau applicatif. Nous établissons d'abord que les systèmes informatiques ambiants sont des systèmes multi-dispositifs. Les systèmes informatiques ambiants que nous appelons systèmes multi-dispositifs ont ainsi une structure intrinsèque modulaire et dynamique. Une modélisation en utilisant un paradigme orienté objet n'est cependant pas satisfaisante à cause du fort couplage en les objets. Nous convergerons vers une modélisation à partir de composants logiciels.

Un système multi-dispositif est un système dont la principale caractéristique est le grand nombre de dispositifs d'entrée-sortie. Les moyens de communication de ces dispositifs avec l'utilisateur et le programmeur sont très limités. Le nombre de standards se multipliant, nous faisons face à des systèmes multi-dispositifs qui requièrent une connaissance à priori de leur infrastructure de dispositifs. La notion d'infrastructure est définie à partir de l'environnement physique et de son incidence sur l'infrastructure matérielle. La modification de ce contexte opérationnel motive alors une nouvelle architecture logicielle pour l'adaptation. Nous exposons une architecture s'appuyant sur la dualité systèmes multi-dispositifs locaux - systèmes multi-dispositifs distribués. Un espace ambiant peut contenir un grand nombre de dispositifs réalisant chacun une tâche précise comme relever la température ou actionner un volet roulant. Chacun de ces systèmes constitue une décomposition du système global. Le grand nombre de dispositifs est une des caractéristiques principales de notre vision de l'informatique ambiante.

Nous proposons un modèle simple de dispositifs. Soit un ensemble d'entités : une unité de calcul, une application logicielle, un utilisateur et un dispositif d'entrée-sortie avec le monde physique réel. Il existe des relations entre ces entités : communications fortement couplées (connexion permanente), communications faiblement couplées (connexion intermittente). A l'aide de ce modèle, nous pouvons, par exemple, différencier des catégories d'ordinateurs telles que les mainframes, les PCs, les grilles et les systèmes informatiques ambiants. Une modélisation classique orientée-objet peut convenir pour la programmation de tels systèmes. Cependant, il existe plusieurs problèmes qui restent encore ouverts tels que :

– Multiples copies d'objets entre systèmes hôtes,
– Mécanisme de réplication de classes.

Le premier point concerne les applications distribuées. On peut prendre comme exemple une hypothétique application où tous les objets sont passés en tant que références distantes sur le réseau. Une référence distante est utilisée pour envoyer un message sur le réseau. Tous les paramètres de la méthode distante invoquée sont alors passés comme références distantes pour que la valeur référée ne puisse pas être accédée localement. Ainsi, le passage de paramètres dans le contexte de ce message distant demande au moins que certains objets soient copiés entre les différentes machines hôtes [149, 150, 151, 152].

En raison de la forte modularité vue dans un premier temps et de la complexité de la modélisation orientée-objet avec un mécanisme de classe, nous avons sélectionné une modélisation orientée-composant qui permet de travailler avec du code modulaire, accessible uniquement à travers une interface pour l'informatique ambiante. Une analyse plus détaillée est réalisée sur les différents modèles à composants existants.

3.1.2 Un espace ambiant intrinsèquement réactif

Dans cette section, nous étudions la manière dont les systèmes informatiques ambiants interagissent avec l'environnement extérieur. Nous y définissons les termes de systèmes transformationnels, interactifs et réactifs en précisant qui effectue le contrôle du système ambiant. La réactivité est un élément clé pour l'informatique ambiante [153]. Dans la mesure où les intergiciels doivent réagir aux modifications du contexte, ils doivent prendre en compte des notifications d'événements selon un mécanisme de publication/souscription par exemple. De manière générale, nous classons les systèmes informatiques en trois grandes catégories selon la relation qu'ils entretiennent avec leur environnement : systèmes transformationnels, systèmes interactifs et systèmes réactifs.

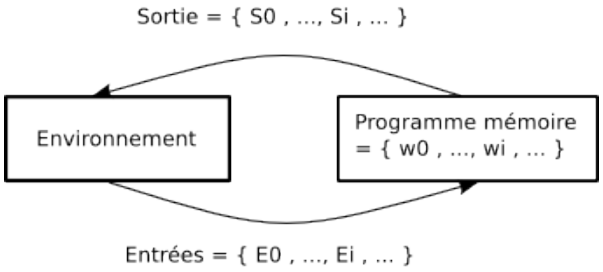


FIGURE 3.2 – Interaction discrétisée entre un système réactif et son environnement

La première catégorie est celle des `systèmes transformationnels`. Ces systèmes sont conçus pour calculer un résultat final à partir d'un ensemble de données d'entrée disponibles au début de leur exécution. L'interaction qui existe entre un tel système et son environnement est très simple : lecture de données fournies par l'environnement au début de l'exécution et écriture d'un résultat fourni à l'environnement en fin d'exécution. Un compilateur est un exemple typique de système transformationnel. Un système transformationnel lit les données fournies par l'environnement, calcule et écrit le résultat fourni à l'environnement. Le contrôleur n'est pas spécifié. Un autre exemple est le SOA [34]. Le SOA est une notion s'appuyant sur le concept de service qui est défini par un contrat et une implémentation. Ils sont typiquement bien adaptés à la modélisation des systèmes transformationnels. Nous retrouvons ce modèle mis en œuvre dans les Web Services ou WSOA. Ces derniers représentent de possibles implémentations techniques.

La seconde catégorie est celle des systèmes dits `interactifs`, tels que les environnements graphiques que nous utilisons tous sur nos ordinateurs. L'interaction entre le système et l'environnement est ici plus fine, puisque les échanges peuvent être fréquents pendant l'exécution du système,

mais elle est principalement dirigée par le système : c'est lui qui en impose le rythme. Un système interactif lit les données fournies par l'environnement, calcule et écrit le résultat fourni à l'environnement. Le contrôleur est le système informatique.

La troisième et dernière catégorie, qui nous intéresse plus particulièrement dans ce travail, est celle des systèmes dits `réactifs` [154]. L'interaction environnement/système est encore ici permanente, mais c'est le rythme des échanges qui est maintenant fixé par l'environnement (voir Fig. 3.2). Le comportement d'un tel système peut être discrétisé et vu comme une succession de valeurs d'entrées E et de sorties S aux différents instants de l'exécution :

$$\langle E_1, S_1 \rangle, \langle E_2, S_2 \rangle, \ldots, \langle E_i, S_i \rangle, \ldots$$

La fréquence des événements est imposée par l'environnement et l'unité de calcul doit suivre cette cadence. Un tel système a une mémoire, notée ω, qui représente une abstraction de l'histoire de l'environnement et peut être utilisée pour calculer les nouvelles valeurs des sorties. Un système réactif lit les données fournies par l'environnement, calcule et écrit le résultat fourni à l'environnement. Le contrôleur n'est pas le système informatique mais l'environnement.

Dans tous les cas, les systèmes informatiques ambiants sont contrôlés par leur environnement physique qui établit leur rythme d'échange d'informations. Nous devons mettre en place un mécanisme de gestion simple de ces stimulus. Dans notre modèle, un mécanisme d'événements permet de garder à la fois un découplage fort entre l'environnement et le logiciel. Les entités logicielles de l'espace ambiant interagissent entre elles mais aussi avec un environnement physique. Ainsi, traditionnellement, l'utilisateur est aux commandes d'un système informatique. Les relations entre les entités logicielles sont de type interactive : la cadence des échanges est fixée par le système informatique. Par contre, un espace ambiant comprend des objets physiques qui contribuent aux applications : la cadence des échanges est fixée par l'espace ambiant.

> *Un système informatique ambiant est un système réactif c'est-à-dire dont le contrôle est effectué par l'environnement. Pour garder un découplage entre l'environnement et le logiciel, un mécanisme de gestion d'événement est intégré à notre modèle.*

3.2 Besoins en adaptation logicielle

Variable et imprévisible ? C'est le cas des systèmes de communication mobiles et nomades tels que les téléphones portables ou les PDA. Le communication avec de tels dispositifs utilise des technologies sans fil qui sont sujets à des variations de performance imprévisibles. L'informatique ambiante, dans lequel une variété de dispositifs peut être connectés temporairement ou de manière permanente au système, font aussi partie de cette catégorie. En plus de faciliter la mise en œuvre d'applications en informatique ambiante, nous devons simplifier leur adaptation à l'espace ambiant. En effet, les dispositifs et les objets de l'environnement réel convergent pour ne plus faire qu'un. On ne voit plus les dispositifs comme des entités particulières mais comme des objets de la vie de tous les jours. Les contraintes subies par les dispositifs sont les mêmes que celles subies par les objets réels. L'application informatique ambiante doit alors s'adapter aux variations permanentes de l'espace ambiant et de l'environnement physique.

3.2.1 L'adaptation et l'IAm

Ces systèmes informatiques ambiants évoluent dans un environnement dynamique constellé de nombreux dispositifs. Leur gestion complexifie l'application et il est nécessaire de trouver un juste milieu entre l'adaptation et les fonctions métier. De nombreux intergiciels sont apparus en informatique ambiante afin d'adapter les applications logicielles aux changements opérés dans l'environnement physique et plus particulièrement sur l'infrastructure de dispositifs pour répondre au problème de

la conception d'applications qui s'adaptent [155, 156, 157, 57]. L'étude de l'adaptation des systèmes informatiques n'est cependant pas un phénomène nouveau. Il existe tant dans le domaine des interactions humain-machine [158] que dans le domaine de l'informatique autonome [159]. Historiquement, l'Intelligence Artificielle (IA), dont l'objectif est de construire des machines intelligentes, semble être la première à s'intéresser à l'automatisation de l'adaptation. En ingénierie logicielle, nous avons beaucoup travaillé sur l'identification des paramètres de personnalisation des applications logicielles.

Les travaux de recherche en adaptation logicielle peuvent être classés en deux catégories : ceux sur les interactions entre l'humain et la machine (qui n'entrent pas dans notre cas d'étude) et ceux qui considèrent les interactions avec l'environnement au sens général (notre domaine d'étude).

Interactions humain-machine. Au niveau interaction humain-machine, les travaux comme ceux de Browne [160] sur le dialogue intelligent adaptatif et les `objectifs de l'adaptation` qui la motivent et ceux de Dieterich [161] sur les niveaux d'adaptation, en définissent les concepts principaux en déterminant les `acteurs de l'adaptation`. Le niveau d'adaptation est résolument le concept fondamental de l'adaptation logicielle. Il permet de classer les travaux accomplis dans le domaine et d'étudier leur complémentarité. Cependant, bien que certaines notions soient assez générales, ces travaux demeurent orientés vers une adaptation des systèmes à l'utilisateur, ignorant, de manière justifiée, leur adaptation à l'infrastructure de dispositifs.

Machine et espace ambiant. Au niveau interaction généralisée entre la machine et son espace ambiant, les travaux sur l'unification de l'adaptation et l'interface [162, 163, 158] résument les principaux concepts dans le domaine de l'adaptation généralisée des systèmes informatiques. Ces travaux posent les fondements conceptuels généraux de l'adaptation logicielle. De ces concepts peuvent être dégagés les problèmes de l'adaptation.

3.2.2 De l'adaptabilité à l'adaptativité

Selon les propriétés utilisées pour spécifier un système adaptatif, le `niveau t'adaptations` d'un système change. Nous proposons de construire une classification des machines adaptatives en fonction de ces propriétés. Thevenin [158] réalise une liste des systèmes adaptatifs en fonction de leur capacité d'adaptation. Il définit cinq catégories de systèmes adaptatifs (sans compter les systèmes dits `câblés`) : adaptable, adaptatif, autorégulateur, automédiateur et automodificateur. Nous ne nous intéressons pas aux cas des deux derniers types de systèmes (automédiateur et automodificateur), car ils ne concernent pas nos travaux.

Un `système adaptable` est un système qui est personnalisable sur intervention explicite de l'utilisateur. Un `système adaptatif` sait reconnaître la situation et est capable de modifier son comportement selon les déclencheurs qui sont mis en œuvre. Un `système autorégulateur` reconnaît la situation et utilise une fonction de boucle de commande pour effectuer son choix de réaction. Nous avons constitué une liste des challenges de l'adaptation logicielle :

1. *Intervention de l'utilisateur* : comment l'utilisateur doit-il interagir avec le système ?
2. *Adaptabilité* : quels sont les mécanismes à mettre en place pour modifier le logiciel ?
3. *Reconnaissance de situations* : quels sont les indices à collecter pour que l'application puisse reconnaître des situations et s'adapter en conséquence ?
4. *Boucle de rétroaction* : comment faire en sorte que la machine puisse avoir un retour sur les actions qu'elle effectue sur son contexte et puisse ajuster son comportement ?

L'intervention de l'utilisateur pose le problème de quand, comment et où il doit intervenir. Celui de l'adaptabilité est le problème de comment le logiciel doit être construit pour qu'il soit adaptable. Celui de la reconnaissance de situations est de construire un modèle capable de conduire une reconfiguration pour des situations données. Enfin, celui de la boucle de commande concerne la logique capable d'ajuster les propriétés du système en fonction de l'environnement. Les systèmes sont regroupés dans le tableau 3.1.

Type de système adaptatif	Intervention utilisateur	**Logiciel adaptable**	**Prise en compte de situations**	Boucle de commande
	1	**2**	**3**	4
Adaptable	•	•		
Adaptatif	•	•	•	
Autorégulateur		•	•	•

Tableau 3.1 – Classification de Thevenin : les catégories des systèmes adaptatifs

Un `système autoadaptatif` – manquant dans la classification de Thevenin – est un système adaptatif qui ne nécessite pas d'intervention directe de l'utilisateur. Le problème de la spécification de l'adaptation logicielle est une préoccupation commune aux systèmes au moins autoadaptatifs. Toutefois, nous pouvons noter que les deux points que sont la détermination des acteurs et la motivation des objectifs demeurent des branches importantes de recherche dans le domaine de l'informatique ambiante. En conclusion, deux points nous intéressent particulièrement. Il s'agit de déterminer des mécanismes pour gérer l'adaptation intégrée au logiciel, des mécanismes de prise en compte des différentes situations et des mécanismes de prise de décision des adaptations à effectuer.

Nous nous intéressons aux problématiques de l'adaptation du logiciel et de la prise en compte de situations non prévues à l'avance. Ceci nous amène à traiter un problème récurant : celui du conflit structurel. Nous décomposons l'adaptation en deux parties : l'adaptativité, partie décisionnelle, et l'adaptabilité, partie effectrice (Fig. 3.3).

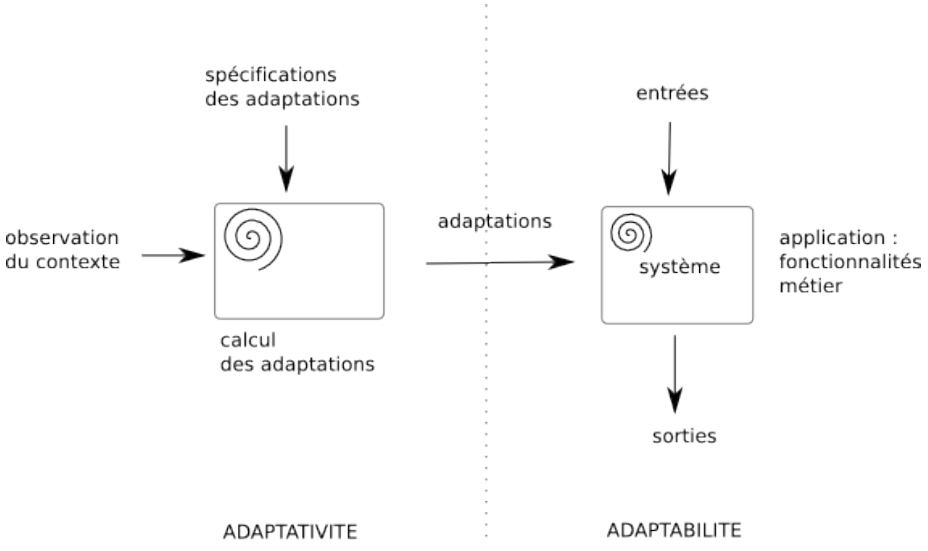


FIGURE 3.3 – Une décomposition de l'adaptation : partie décisionnelle et partie effectrice

Déf. 1: `Adaptativité`. L'adaptativité d'un système est la manière dont le système s'adapte. Elle est constituée de stratégies qui permettent de déclencher des modifications du système en fonction de stimuli.

Déf. 2: `Adaptabilité`. L'adaptabilité d'un système constitue les mécanismes logiciels qui permettent de réaliser les modifications du système. Ce sont ces mécanismes qui permettent de modifier le comportement du système tout en préservant des propriétés du système. Les stratégies d'adaptativité contrôlent les mécanismes d'adaptabilité.

Les deux disciplines (interactions humain-machine et informatique autonome) se rejoignent sur ce problème. Si l'IA visait l'`adaptativité` des systèmes, l'ingénierie logicielle traitait essentiellement de leur `adaptabilité`. Pour être plus précis, nous distinguons également l'adaptation de l'évolution logicielle. En effet, contrairement à l'`évolution logicielle` qui suppose en général l'accès au code des composants et à la personnalisation qui suppose un certain nombre de possibilités de modification prévues à l'avance, l'objectif de l'`adaptation logicielle` [164] est de réaliser des modifications de façon non intrusive (pas de modification directe de code qui est non accessible), de manière systématique (pas ou peu d'intervention humaine) et de manière dynamique (la structure de l'adaptateur n'est pas définie à priori et l'adaptation se fait au cours de l'exécution).

> *Dans un système informatique ambiant, l'adaptation logicielle est réactive et infrastructurelle. Des stratégies sont alors nécessaires pour effectuer les modifications, c'est ce que nous appelons le modèle d'adaptativité. Un tel modèle s'appuie sur un modèle de composant sous-jacent adaptable.*

3.3 Synthèse et critique de l'existant

Nous récapitulons dans ce paragraphe les principales contributions de la littérature pour chaque besoin précédemment identifié pour l'IAm.

3.3.1 Principes de la littérature

Nous allons voir que les travaux issus de la littérature prennent partiellement en compte les différentes nouvelles caractéristiques et nouveaux besoins de l'informatique ambiante telle que nous l'avons décrite. Nous présentons d'abord une synthèse des travaux étudiés au chapitre 2 à partir de laquelle nous retirons les problèmes qui n'ont pas encore de solution.

Synthèse : Nous pouvons classer les travaux selon trois catégories de systèmes pour l'adaptation : ceux qui prennent en compte la dynamique de l'infrastructure, ceux qui s'adaptent à tous les niveaux hiérarchiques et ceux qui s'intéressent plus à la réutilisabilité.

Prise en compte de la dynamique de l'infrastructure : Les systèmes de cette catégorie [51] proposent d'anticiper les actions du contexte et d'allouer les ressources uniquement en cas de besoin. Ces systèmes consistent de manière générale à filtrer les données remontantes de l'infrastructure. De même, une classe de système, par exemple, Roman [59] dans Meditrina modifie la manière dont les propriétés des capteurs et des actionneurs apparaissent à l'application (API).

Adaptation à tous les niveaux : Les travaux d'adaptation à tout niveau [50, 54] proposent une vision globale de l'application où chaque composant a accès à une évaluation globale du système.

Réutilisabilité : Les travaux sur la réutilisabilité [58, 65] proposent une architecture s'appuyant sur des composants pour faciliter le remplacement dynamique des entités du modèle. La décomposition est directement issue de la structure de l'espace ambiant. Ils exploitent cette décomposition naturelle de l'espace ambiant pour en déduire la connexion entre les composants logiciels : ces travaux infèrent alors les dépendances entre les objets de l'espace ambiant.

Problèmes liés à l'IAm : Les solutions précédentes sont cependant très liées au langage, même pour les systèmes favorisant la réutilisabilité où l'adaptativité est limitée à un système d'inférence. Ces solutions, cependant, limitent à la composition et la mise en œuvre des adaptations logicielles. Lorsque plusieurs adaptations sont en conflit, leur exécution doit alors être contrôlée afin de conserver la sémantique des différentes fonctionnalités proposées. Parmi les solutions proposées, l'environnement n'a d'impact que sur des événements de déclenchement d'actions [123] ou bien sur la manière dont les objets vont communiquer étant donnée leur localisation [54]. Il serait intéressant d'aller plus loin en permettant à l'environnement d'influencer sur la constitution même des adaptations.

Pour maîtriser la distribution, nous nous appuyons sur une architecture orientée service. Cela permet de représenter les entités distribuées par des services. La découverte de service est la capacité à trouver et à accéder aux services existants.

3.3.2 Principes de notre approche

Prise en compte des dispositifs blackbox : On ne peut pas modifier la structure et le comportement d'un dispositif de l'infrastructure (*blackbox*), car un système informatique ambiant subit les variations de l'environnement. Nous devons proposer un modèle qui permet de prendre en compte cette isolation tout en offrant la possibilité d'adapter la structure et le comportement global de l'application.

Communication événementielle : Nous pouvons distinguer deux grandes approches dans la programmation des systèmes informatiques. La première approche prend essentiellement place dans les systèmes d'information destinés à traiter des données tout en respectant les algorithmes programmés. La seconde approche est plutôt celle qui produit des réponses programmées aux stimulus produits par l'environnement. Plus spécifiquement, elle correspond à la mise en œuvre dans les systèmes réactifs et interactifs qui sont parfois couplés à un environnement physique comme les Systèmes Autonomes ou à des utilisateurs comme les Interfaces Hommes Machines. Dans cette seconde approche, on parlera alors de *respect de la réactivité du système* par opposition au *strict respect du processus de transformation des données* [154].

Cette dualité alimente les plus récents travaux sur les architectures logicielles. Elle s'impose notamment aux architectures orientées service (SOA) pour palier les limitations de la classique décomposition d'un *workflow* en séries d'activités ou d'invocations de services dans la prise en compte d'*informations spontanées* [165]. Après avoir promu un modèle de SOA adapté aux systèmes d'information, la problématique s'étend aux espaces ambiants et interactifs avec des services non plus basés sur des invocations de méthodes comme seul mode d'interaction, mais basés aussi sur des émissions d'événements.

La programmation événementielle présente alors deux avantages majeurs. Le premier avantage repose sur notion d'`inversion de contrôle` souvent associée à celle d'événement. En effet, plutôt que de spécifier des séquences d'appels de méthodes à invoquer pendant l'exécution d'un processus, l'inversion de contrôle consiste à programmer les réactions en réponse aux événements émis par d'autres processus de traitement ou à programmer des réactions pour des modifications de l'environnement physique aux travers les entrées/sorties du système. Le second avantage souvent associé à la notion d'`injection de dépendance` est le découplage des relations de dépendance entre les entités logicielles du système, ce qui facilite ainsi les insertions et les retraits.

Toute ces raisons nous ont conduit à nous appuyer sur une approche événementielle comme nous le verrons plus en détail dans la suite.

La modularité transverse à son paroxysme : Nous mettons en avant deux principes. D'abord, l'idée d'une "programmation transverse" que nous proposons s'appuie sur une méthodologie où tout

est aspect. Il n'y a plus d'application initiale enrichie à posteriori par des modularités transverses. Toute entité logicielle devient une modularité transverse qui décrit une partie de l'application. Nous pouvons toujours représenter une modularité standard par une modularité transverse, c'est-à-dire que nous ne perdons pas le pouvoir d'expression. Ces modules transverses coïncident partiellement en des points au niveau logiciel pour engendrer un assemblage de composants représentant l'application. Après la vision de l'AOP de Kiczales, on aborde maintenant l'AO4BPEL de Charfi [117] qui permet de reconfigurer la composition des services à l'aide des aspects. Nous avançons dans notre recherche en éliminant l'application de base et en exprimant les compositions de services essentiellement à partir d'aspects particuliers.

3.4 Notre approche

Dans notre modèle, toute application se présente comme un assemblage de composants encapsulé dans un service composite. Nous avons vu que les solutions de la littérature sont basées sur des langages. Notre approche s'appuie sur des assemblages de composants locaux et légers communiquant par flot d'événements. Les points ainsi énoncés propose une solution pour l'adaptabilité (capacité d'adaptation) à l'application informatique ambiante. Enfin, nous proposons une approche pour gérer l'adaptativité (activité de l'adaptation) des applications par la notion d'aspect d'assemblage.

3.4.1 Modèle de service composite

Un service est adapté à la distribution, car elle n'impose pas de structure particulière pour l'unité de déploiement. Par opposition, un composant est adapté à la construction d'applications localisées puisqu'il n'impose pas de mécanisme de recherche distribué à l'exécution imposée par l'adaptation réactive caractéristique de l'IAm (section 3.1). Ainsi, nous mettons en place la notion de service composite (section 4.2) qui consiste à implémenter un service distribué par un assemblage de composants logiciels légers localisés. Les services composites sont constitués de deux interfaces : une interface de modification structurelle pour gérer l'adaptabilité (section 3.2.2) et un service exportant les fonctionnalités internes pour garder la capacité de réutilisabilité (section 3.3.1).

Nous avons vu que les applications se composent en deux parties : transformationnelles et interactive/réactive (section 3.1.2). Les JavaBeans [96] constituent un exemple de communication interactive/réactive. L'analyse des problèmes de communication entre les entités modulaires dans le cadre des applications transformationnelles a trouvé une solution dans l'architecture SOA. Cependant, l'analyse de la communication dans le cadre des applications interactives/réactives ne peut pas s'appuyer sur la même solution. Elle s'appuie dans notre cas sur une modélisation de la communication par événements au niveau des services composites pour les espaces ambiants (section 4.1.2.1).

3.4.2 Modèle de composant

Préserver la sémantique des programmes au niveau d'une architecture à service n'est pas une tâche simple pour deux raisons :

- les pannes ou modes dégradés sont différents dans les cas d'un logiciel local ou d'un logiciel distribué ; dans ce dernier, le client, le serveur et le réseau peuvent tomber en panne de manière indépendante ;
- même en l'absence de panne, la sémantique du passage de paramètres est différente (par exemple, le passage par pointeur ne fonctionne pas en environnement distribué, car le processus appelant et la procédure appelée sont dans des espaces d'adressage différents).

Problèmes de latence. Au niveau du client, une thread doit être bloquée lorsque celle-ci attend le retour d'un appel à une procédure distante. Afin d'éviter ces temps de latence, une solution alternative consiste à gérer un pool de taille finie de worker-threads. Ces threads attendent qu'un travail leur soit fourni, l'exécute et retourne dans le pool. C'est en ce sens que nous proposons

un modèle de composant local s'appuyant sur un paradigme à événement afin de réduire le temps de latence. Au niveau local, nous proposons une modélisation par composant léger (section 4.2.3). Un composant léger est un composant logiciel dont l'implémentation ne contient pas des espaces d'adressage multiples (c'est-à-dire de distribution) [166] et toutes les conséquences qui en découlent comme la gestion de la synchronisation par exemple. Cela permet, sans sur-coût (*overhead*), une connexion de deux composants à travers leur interface. Les composants légers sont ainsi assemblés pour constituer une application au plus proche de la cible. De plus, les systèmes informatiques ambiants ont des ressources limitées [167].

Composants spécifiques au service composite : les sondes et les puits (section 4.3.1) permettent de construire une interface offerte au service composite et exporte les données confinées à l'assemblage. Cela permet d'exporter les fonctionnalités internes pour garder la capacité de réutilisabilité (section 3.3.1). Enfin, pour gérer la réactivité (section 3.1.2), les interfaces des composants sont constituées d'un ensemble de points d'entrée et d'événements (section 4.2.3).

3.4.3 Modèle d'adaptativité transverse

L'adaptativité doit prendre en compte le caractère hautement dynamique de l'espace ambiant (décrit dans la section 3.1). Par conséquent, elle doit fournir un mécanisme de reconfiguration logicielle qui réagit à chaque modification de l'espace ambiant (section 5.5). Pour cela, nous avons choisi d'exprimer une adaptation par une entité transverse à l'instar d'un aspect, mais poussée à son paroxysme. Une adaptation ne s'appuiera non plus sur une application déjà existante, mais construira, sur mesure, l'application dont les objectifs correspondent à la situation réelle de l'espace ambiant (section 5.2.2).

Nous allons, à présent, entrer dans le détail de notre modèle. Ainsi, dans la partie suivante, nous présenterons les éléments fondamentaux des assemblages de composants logiciels. Puis, nous nous intéresserons à leur adaptation. Pour cela, nous reprendrons les principales notions issues (section 2.1.3) de l'étude que nous avons effectuée dans la partie 2 sur les plates-formes à composants, de services et d'aspects. À partir de là, nous bâtirons notre approche en suivant les trois directives énoncées précédemment :

- Service composite (designer et container),
- Composant générique local,
- Adaptativité transverse logique et sans application initiale.

Troisième partie

Modèle de composition d'adaptations

Chapitre 4

Modèle de service composite pour l'IAm

Sommaire

Un système informatique ambiant est un système dont les entités qui le composent (tels que les capteurs, les actionneurs, les dispositifs de traitement, etc.) changent constamment. Ces changements sont causés par l'infrastructure du réseau (Manet ou Mesh). Nous ne pouvons alors faire aucune hypothèse sur la présence et l'interface de ces entités distribuées. Il s'agit de trouver des paradigmes pour gérer de façon efficace le développement de ces applications et faire en sorte que des principes généraux puissent être appliqués et que des briques de base puissent être réutilisées.

Pour cela, nous proposons un modèle de service composite adaptable pour l'informatique ambiante qui prend en compte au mieux les principes liés à l'adaptabilité énoncés dans le chapitre de synthèse. Ce modèle met en avant deux principes généraux :

- Des services s'appuyant sur la notion d'événement :
  Les applications informatiques ambiantes sont alors des graphes de services basés sur des événements.
- Des services composites constitués de composants légers :
  Un service composite basé sur les événements est dynamiquement géré par un assemblage interne de composants légers.

Notre modèle de service permet d'adapter dynamiquement les applications pour l'informatique ambiante. Ce chapitre présente une approche à base de composants (modèle SLCA) pour concevoir des services composites de haut niveau : une application est un graphe de services coopérants.

4.1 Service pour l'informatique ambiante

La notion de service est souvent raccrochée à la problématique de la distribution logicielle [168]. Les architectures orientées service permet de séparer les préoccupations en ne permettant aux éléments de base comme les services de communiquer avec d'autres services uniquement à travers de leur interface. Le principal avantage est le fort découplage entre les services, ce qui permet la réactivité et la réutilisabilité. La notion de service permet de travailler avec la problématique de l'adaptation réactive. Les concepts de service et celui de composant sont bien définis, mais peuvent entrer en conflit sur certains points. Nous étudions d'abord ces conflits et ensuite nous nous focaliserons sur les caractéristiques de notre modèle de service.

4.1.1 Service et composant

La frontière entre la notion de `service` et celle de `composant` peut paraître floue à première vue. Des modèles de composant complexes offrent parfois des fonctionnalités similaires à celles fournies par les services : programmation distribuée, découverte dynamique, interopérabilité, etc. Ils permettent alors le déploiement d'applications sur plusieurs noeuds. Le point important à mettre en valeur en informatique ambiante est de pouvoir créer des applications qui sont capables de s'adapter aux modifications de l'environnement telles que l'apparition de dispositifs ou la modification de leur interface, bien plus que de provoquer ces changements. De plus, la notion de service inclut une existence autonome. Cela est un avantage lorsque nous les utilisons pour des dispositifs physiques. Par conséquent, nous basons notre approche sur le concept de service qui représentera la communication entre les entités diverses dans l'environnement incluant les dispositifs et pour la capacité d'adaptation des composants logiciels. La notion de service n'est pas contraire au concept de localité, ni à celui de blackbox qui sont souvent raccrochés à la notion de composant.

Service et localité. La localité détermine l'espace d'exécution des entités (local/distribué). Rassembler des entités dans un même espace d'adressage permet d'obtenir de meilleures performances au niveau de la communication, mais permet surtout de garder un état stable d'une application sans se soucier des variations de l'environnement. Les services OSGi illustrent ces entités s'exécutant dans un même espace d'adressage.

Service et blackbox. D'une part, une entité blackbox limite les interactions à travers la notion de service en restreignant les communications et les modifications aux interfaces fournies ou requises et en interdisant tout accès direct à l'implémentation (par exemple, pour effectuer des adaptations). D'autre part, utiliser des entités blackbox améliore la réutilisabilité lors de l'exécution puisque l'on dispose de la garantie que son implémentation est exactement la même.

Nous emploierons la notion de service dans sa définition la plus minimale et stricte : une interface contractuelle entre les entités de notre application. Cette notion autorise la distribution et améliore la réutilisabilité en interdisant la modification de l'implémentation des entités blackbox.

4.1.2 Caractéristiques conférées aux services

Les dispositifs de l'infrastructure en informatique ambiante ressemblent aux services de part leur autonomie, leur indépendance et parce qu'ils offrent un ensemble de fonctionnalités que l'on peut exploiter. Ces dispositifs évoluent, changent de position géographique, changent d'interface ; par conséquent les schémas de communication ne sont plus les mêmes.

Les modèles de service ont beaucoup évolués depuis 10 ans [108, 169, 170, 171] et incluent les concepts de services à ceux de plates-formes à événements en plus de la découverte dynamique. De nouvelles caractéristiques telles que l'interopérabilité contribuent maintenant à la résoudre le problème d'adaptation réactive.

4.1.2.1 Réactivité

Traditionnellement, les invocations distantes (RPC) permettent de passer le flot de contrôle d'un service à un autre. Ces invocations conviennent bien aux applications de traitement d'information. En informatique ambiante, les utilisateurs (que ce soit des êtres humains ou d'autres services) souhaitent être notifiés d'un changement. Les mécanismes de *publish/subscribe* ou de notification d'événements proviennent de ce besoin de réactivité dans les applications. De plus, le mécanisme d'événement apporte un découplage plus important entre les services.

4.1.2.2 Interopérabilité.

Nous distinguons deux types d'interopérabilité : celle des entités et celle de la plate-forme.

L'interopérabilité des services (système s'exprimant par un protocole qui lui est propre), c'est-à-dire au niveau de la communication, est réalisée par l'ajout de *stubs/skeleton* en CORBA ou encore de l'utilisation d'une technologie commune comme TCP/IP, HTTP ou XML dans le cas des services Web. Elle est résolue par une couche logicielle intermédiaire.

L'interopérabilité de la plate-forme (exécution d'une même entité logicielle sur des unités physiques différentes) est réalisée par la présence d'une machine virtuelle. Le programme exécutable est écrit dans un langage intermédiaire interprété par une machine virtuelle. Cela permet de simplifier le mécanisme de déploiement des applications.

Nous avons vu que les services ne sont plus réservés à des applications à grande échelle (comme le Web par exemple) pour des systèmes d'information, mais sont de plus en plus tournés vers l'informatique ambiante par exemple. En effet, ils offrent à présent des communications basées sur les événements et une gestion de l'interopérabilité. Toutefois, un fossé est toujours présent entre le développement d'applications pour l'informatique ambiante essentiellement basé sur des services à cause du *découplage trop fort* qu'il occasionne (moins de flexibilité), le manque de *confiance* au niveau des interfaces et leur *disponibilité* que temporaire (dynamisme de l'infrastructure).

Nous proposons dans les sections suivantes un modèle de service incluant les propriétés énoncées et traitant du problème de fort découplage et de forte variabilité.

4.1.3 Modèle de service pour l'IAm

L'infrastructure est constituée de l'ensemble des services accessibles et des dispositifs utilisables. Notre modèle s'appuie sur une infrastructure de services utilisant les événements et que l'on peut découvrir dynamiquement. Ces services représentent les dispositifs utilisés aussi bien dans les applications IAm que dans les services composites. Le problème d'interopérabilité des services est résolu par ce modèle, ce qui permet de programmer les dispositifs en utilisant le langage que l'on souhaite. Celle de la plate-forme dépend de la cible utilisée.

Un service est un ensemble de fonctions fournies par un système logiciel à un autre système logiciel. Il est généralement accessible à travers une API. Nous distinguons deux catégories de services :

- Service classique
- Service pour dispositif

Les services classiques constituent une approche pour faciliter leur interopérabilité indépendamment des technologies sous-jacentes comme c'est le cas pour la technologie Web Service standard [28]. Utiliser cette approche pour accéder à des dispositifs permet de gérer l'apparition et la disparition de dispositifs comme l'arrivée ou le départ d'un service. Cette approche s'appuie sur la technologie Web Service pour Dispositifs [172] dont une première mise en œuvre est apparue avec le standard UPnP.

La notion de service pour dispositif requiert deux extensions majeures du modèle de service classique :

- un mécanisme de recherche et de découverte dynamique de nouveaux services pour dispositifs dans l'environnement proche du système IAm ;
- un mode de communication par événements pour garantir un minimum de réactivité aux variations des entrées-sorties des dispositifs.

Les dispositifs étant le plus souvent connectés à l'environnement physique, les services associés doivent offrir les caractéristiques pour prendre en compte la réactivité de l'application aux variations de l'environnement. Le rythme des échanges environnement/système est fixé par l'environnement et a également pour effet d'accroître également le découplage entre les entités qui le composent [173]

La publication et la découverte d'un service s'accompagne d'une description dynamique de son interface qui assure l'indépendance du fournisseur de service par rapport au consommateur [174]. Pour avoir un découplage plus fort, cette description peut être étendue [175, 176] par l'ajout de méta-données.

Les services pour dispositifs sont contraints par les cibles auxquels ils sont associés [28]. Certains mécanismes permettent de déterminer les services les mieux adaptés à la résolution d'une tâche [177]. En fonction des approches, il peut s'agir de prendre en compte l'auto-organisation [178]. Nous conditionnons la découverte d'un service au contexte. plus généralement, sa disponibilité peut être alors conditionnée à des conditions contextuelles d'utilisation [179, 180].

Notre modèle de service. Dans notre modèle, nous avons une unique notion de service. Un service a une interface qui détermine les ports par lesquels le service est connecté à l'infrastructure. Nous distinguons deux catégories de port :

- l'ensemble des points d'entrée noté I
- l'ensemble des événements noté O.

La paire (I, O) caractérise la notion d'interface (parfois notée $(I \blacktriangleright O)$ [181]).

Ce modèle supporte cinq fonctionnalités :

- *Découverte des dispositifs :* les dispositifs peuvent être découverts dans l'infrastructure du système IAm passivement (attente d'une notification) ou activement (envoi régulier de messages) ;
- *Entités de l'infrastructure multi-service :* les entités de l'infrastructure peuvent fournir un ou plusieurs services. Ces capacités sont fixes et spécifiées à travers la description d d'une entité ;

– *Invocations synchrones des points d'entrée :* on attend la fin de l'exécution des fonctionnalités offertes ;
– *Mécanisme de souscription aux événements :* pouvoir s'abonner aux événements diffusés par l'entité de l'infrastructure ;
– *Description générale d'une entité :* possibilité d'obtenir une description plus générale de l'entité de l'infrastructure.

Notons que nous ne considérons pas les notions de sécurité qui peuvent être ajoutées a posteriori à ce modèle.

4.1.4 Système informatique ambiant

Un modèle de service pour l'informatique ambiante est la solution que nous proposons à partir de l'étude que nous avons faite : faire face à la variabilité de l'infrastructure logicielle liée à sa distribution. Un système informatique ambiante est un graphe de services qui collaborent pour construire plusieurs applications pour plusieurs utilisateurs. Les modèles d'orchestrations centralisées ne peuvent plus s'appliquer.

4.1.4.1 Graphe de services

Notre modèle définit une modèle d'architecture composée de services et s'appuyant sur des événements. Il permet de construire des graphes de coopération entre services et applications. L'environnement physique est constitué d'utilisateurs mobiles interagissant avec le monde réel et d'autres utilisateurs portant des dispositifs sur eux. Le modèle permet de les percevoir comme des services disponibles momentanément dans l'infrastructure. Nous constatons qu'il n'existe plus d'application prédéfinie, mais seulement un graphe de services qui collaborent.

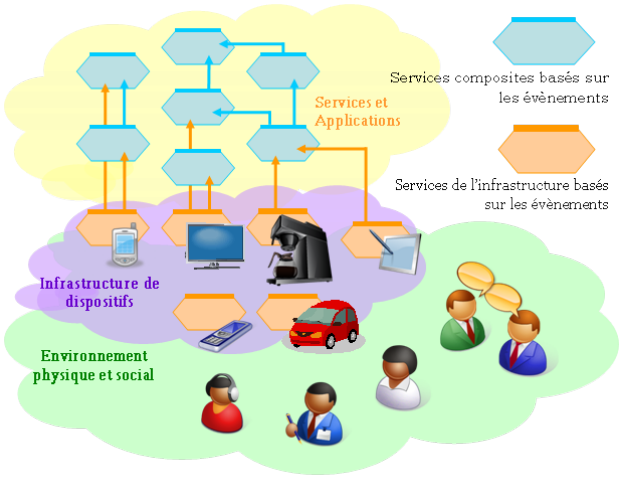


FIGURE 4.1 – Graphe de services basés sur des événements

4.1.4.2 Pas d'orchestration centralisée

L'enjeu principal de l'informatique ambiante est de trouver une solution pour résoudre le problème de l'informatique *multiple*. Plusieurs utilisateurs peuvent utiliser simultanément plusieurs applications fragmentées en plusieurs services. Ces utilisateurs interagissent avec plusieurs dispositifs afin de communiquer avec d'autres utilisateurs localisés en différents espaces physiques et environnements. En effet, l'informatique ambiante est multi-application, multi-user et multi-dispositif. Il ne peut plus y avoir d'orchestration centralisée.

4.2 Service composite

Les services sont aussi utilisés comme axes d'échanges entre les sous-systèmes hétérogènes. Dans ce contexte d'utilisation deux points de vues sont possibles :

- *Chorégraphie :* spécification externe décrivant l'enchaînement des services ;
- *Orchestration :* réalisation interne des échanges entre services réalisant la chorégraphie.

Notre modèle ne fait pas la distinction entre ces deux formes de composition. Nous nous intéressons de manière plus générale à la composition de services.

Nous voyons dans cette partie comment s'effectue la conception d'un service composite à travers les approches statiques de la littérature. Nous proposons alors une approche dynamique de la composition de services qui s'appuie sur un modèle de composant dits `légers`.

4.2.1 Composition de services

Nous avions précédemment le modèle d'un service à travers son interface. Nous voyons à présent comment construire un service à partir de services existants de l'infrastructure. Ce nouveau service est appelé `service composite`.

La composition d'une interface (I, O) peut être décrite par une fonction F de l'ensemble des points d'entrée I dans une partie de l'ensemble des événements $\mathcal{P}(O)$.

La composition de services mettent en jeu la connaissance et l'entrelacement de services. Les services sont particulièrement difficiles à composer pour prendre en compte la gestion des erreurs, de contextes, d'ordonnancement de l'exécution, etc. Nous pouvons voir la composition sous deux angles :

- considérer les éléments de l'infrastructure (services primitifs et services composites) comme des boîtes noires accessibles uniquement via leur interface ; Dans ce cas, se pose le problème d'exécutions multiples de services communs, d'absence de partage du contexte de gestion des erreurs, etc.
- considérer les orchestrations comme des boîtes blanches ; Ce cas suppose une bonne connaissance des services et des compositions déjà existantes.

Nous voyons dans la section suivante les approches de la littérature pour la composition de services.

4.2.2 Plusieurs approches dans la littérature

Différents formalismes ont été définis pour la composition des services [182] ayant surtout pour but de minimiser le couplage entre les services composites et les services de l'infrastructure. Le résultat d'une orchestration est un nouveau service, ce qui permet la composition récursive d'orchestrations.

Outre les approches ad hoc de construction des services composites, des approches formelles permettent la validation de la composition des services en dérivant du langage BPEL4WS des vérifications formelles (voir page 55). D'autres approches consistent à améliorer l'expression de la composition en proposant de regrouper la spécification d'une composition dans un module logiciel (AO4BPEL), de construire des connecteurs transverses (JAsCo) ou de spécifier de manière déclarative la composition (Self-Serv), enfin de considérer la composition comme un assemblage de composants (SCA).

Cependant, lorsque les compositions de services sont définies statiquement (cas des approches relatives à BPEL), il n'est pas possible d'adapter la liaison à un service. Or, cette adaptation est nécessaire pour sélectionner le service approprié en fonction des choix de l'utilisateur [183].

Il s'agit alors de définir des règles de sélection des services. Les travaux relatifs au web sémantique abordent cette problématique via l'usage d'ontologies [177] qui permettent de déterminer en fonction d'une tâche les services les mieux adaptés ou en cas d'échec pour substituer de nouveaux services répondant mieux aux exigences utilisateur [184]. En fonction des approches, il s'agit de prendre en compte soit l'adaptation par l'utilisateur, soit l'auto-configuration [178].

4.2.3 Notre modèle de composant léger LCA

L'adaptation réactive en l'informatique ambiante intervient au niveau la distribution et la présence intermittente des dispositifs. Pour garantir et maîtriser le cycle de vie logicielle, il s'agit de ne pas subir directement et sans contrôle à priori les variations de l'environnement et de l'infrastructure.

Pour ce faire, nous interdisons, dans notre modèle de composant, les mécanismes de distribution. Cela implique que les composants sont exécutés dans un même espace d'adressage et dans un même processus. Leurs interactions sont alors réduites aux mécanismes les plus simples et les plus efficaces : flot de contrôle et de données. Les composants n'embarquent alors plus de gestion non-fonctionnelle liée aux intergiciels. Leur impact mémoire est réduite. Nous qualifions pour ces raisons ce modèle de léger.

4.2.3.1 Assemblage de composants légers

Dans notre modèle LCA (Lightweight Component Architecture) [1], l'application est un assemblage de composants.

Structure d'un assemblage de composants. Un assemblage est une entité d'exécution qui comprend deux ensembles : l'ensemble des composants et l'ensemble des connecteurs. L'ensemble des composants contient les identifiants des composants participant à l'assemblage.

L'ensemble des connecteurs contient les connecteurs entre ces composants. Chaque connecteur est représentée par un couple de ports $(\hat{}o, i)$ tel que i soit un point d'entrée et $\hat{}o$ un événement.

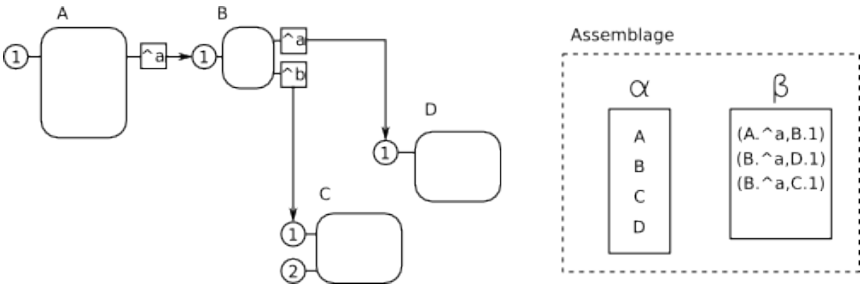


FIGURE 4.2 – Assemblage de composants

La figure 4.2 représente un exemple d'assemblage. L'assemblage est représenté par un rectangle aux traits discontinus. Il est modélisé par deux ensembles (tel qu'il est illustré à gauche de la figure) : α – qui correspond aux composants participant à l'assemblage –, et β – qui correspond aux connecteurs entre ces composants. Les éléments de l'ensemble α sont des identifiants des composants (A, B, C et D) et ceux de l'ensemble β sont des couples de ports tels que $(A.\hat{}a, B.1)$, $(B.\hat{}a, D.1)$ et $(B.\hat{}b, C.1)$.

Structure d'un composant : Un composant possède une interface (I, O). De la même manière qu'une interface de service, il existe deux catégories de ports qui constituent l'interface d'un composant :

- les points d'entrées du composant
- les événements qui est une liste de wrappers vers des points d'entrées

Un composant léger respecte les trois propriétés suivantes :

- Un composant ne communique avec d'autres composants qu'à travers leur interface et ne contiennent aucune référence directe vers d'autres composants à leur conception. Il respecte à ce niveau la propriété blackbox (approche contraire à MOP ou aux interactions [30]).
- La seule gestion non-fonctionnelle présente dans un composant est la gestion des événements.

1. Architecture à Composants Légers

– Pour réaliser dynamiquement leur interconnexion avec d'autres composants logiciels, nous utilisons une indirection qui associe un pointeur prédéfini dans une blackbox à un pointeur externe que l'on peut modifier. C'est ce que nous appelons wrapper. Lorsqu'un événement est émis, chaque point d'entrée associé est exécuté de manière synchrone et séquentiellement et dans l'ordre de leur abonnement. Nous verrons qu'en ajoutant un composant de séquence, il est possible de rendre la diffusion d'événements complètement déterministe.

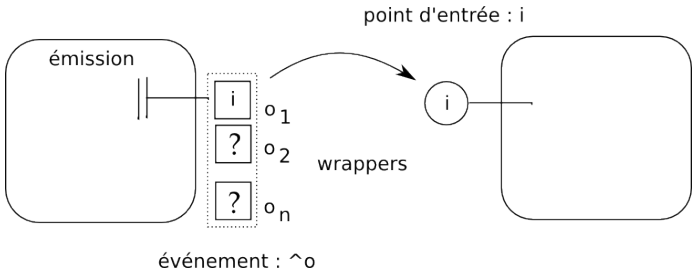


FIGURE 4.3 – Wrapper et événement

Un composant est conforme à une description qui contient les informations nécessaires à son instanciation. Les descriptions sont regroupées dans un dépôt modifiable à l'exécution.

Lorsqu'un service de l'infrastructure est découvert, une ou plusieurs descriptions sont dynamiquement crées à l'image de leurs interfaces. Des composants logiciels sont instanciés immédiatement. Ces composants logiciels sont appelés composants mixtes, car ils ne sont pas purement logiciels et représentent les services de l'infrastructure.

4.2.3.2 Composition par flots d'événements

Les composants d'un assemblage peuvent être composés en les reliant par des connecteurs.

Structure d'un connecteur. Nous avons deux sortes de connecteurs : les connecteurs de base et les connecteurs complexes. Les connecteurs permettent aux composants de communiquer entre eux. Nous avons deux types de connecteurs : les connecteurs de base et les connecteurs complexes.

Connecteur de base. Les connecteurs de base transmettent un flot de contrôle et éventuellement des informations entre un *wrapper* et un point d'entrée.

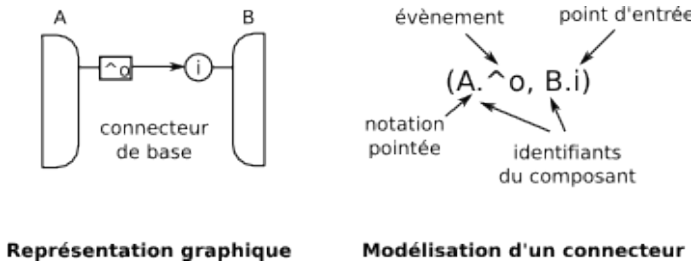


FIGURE 4.4 – Modélisation d'un connecteur

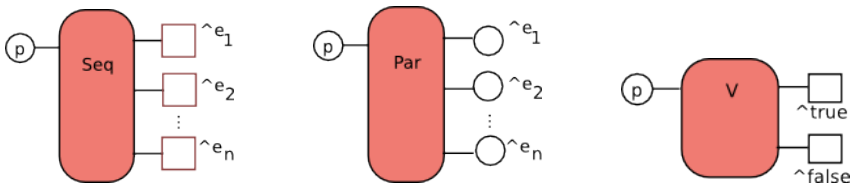
Un connecteur est définie par un couple de ports $(A.\hat{}o, B.i) \in O_A \times I_B$ où i et $\hat{}o$ désignent respectivement points d'entrée et les événements qui sont préfixés par l'identifiant du composant auquel ils appartiennent. Comme il est illustré dans la figure 4.4, nous utilisons une notation pointée pour désigner ces ports. Ainsi, la représentation graphique d'un connecteur de base (à gauche de la figure) est modélisée par le couple $(A.\hat{}o, B.i)$.

Connecteur complexe. Un connecteur complexe met en relation n points d'entrée à m événements où n, m sont des entiers supérieurs strictement à 1. Contrairement aux composants blackbox, notre

modèle se focalise plutôt sur des connecteurs complexes de sémantique connue. Un connecteur complexe est toujours de sémantique connue. Exemple : le parallélisme, la disjonction, etc.

La représentation structurelle de notre modèle est valable pour le flot de contrôle et le flot de données.

Flot de contrôle Dans notre modèle, nous définissons quelques composants qui permettent de représenter les manipulations du flot de contrôle. Les données transmises à ces composants ne sont pas modifiées.

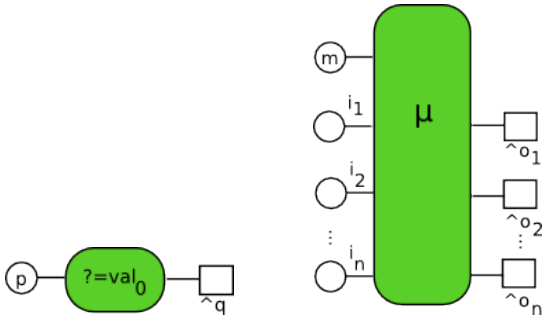


- Un composant de séquence noté 'Seq' où $I_{Seq} = \{p\}$ et $O_{Seq} = \{\hat{}e_1, \ldots, \hat{}e_n\}$ entre plusieurs événements est le plus simple : l'exécution du point d'entrée p émet en séquence synchrone respectivement les événements $\hat{}e_1, \ldots, \hat{}e_n$.

- Nous définissons la concurrence notée Par ($I_{Par} = \{p\}$ et $O_{Par} = \{\hat{}e_1, \ldots, \hat{}e_n\}$) entre plusieurs émissions d'événements : l'exécution du point d'entrée p provoque l'exécution en synchrone respectivement les événements $\hat{}e_1, \ldots, \hat{}e_n$.

- Un composant de disjonction noté '$\vee$' où $I_\vee = \{p(b \in BOOL)\}$ et $O_\vee = \{\hat{}true, \hat{}false\}$ provoque l'émission de l'événement $\hat{}true$ si la valeur booléenne b est vraie et l'émission de $\hat{}false$ sinon. Notons que la donnée transmise avec le flot de contrôle n'est pas modifiée.

Flot de données Nous définissons trois composants qui permettent de représenter une transformation de données (sans modification du flot de contrôle) :

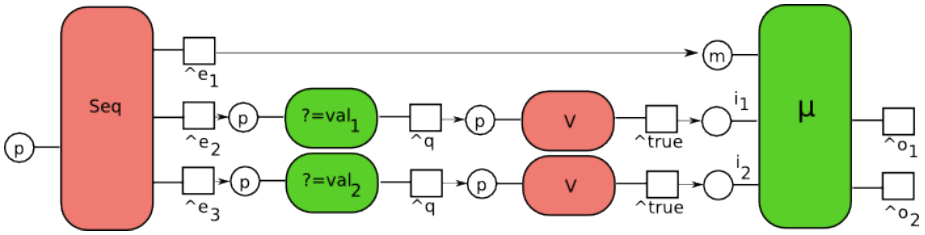


- Un composant de test noté '?' où $I_? = \{p\}$ et $O_? = \{q(b \in BOOL)\}$ transforme d'une information i en information booléenne b en fonction de l'égalité avec une donnée interne i_0.

- Un composant de mémoire noté 'μ' où $I_\vee = \{m(val), i_1, \ldots, i_n\}$ et $O_\vee = \{\hat{}o_1, \ldots, \hat{}o_n\}$ une information i est mémorisée lors de l'exécution du point d'entrée $m(i)$. Cette information est ensuite émise par l'un des événements $\hat{}o_1, \ldots, \hat{}o_n$ que lorsqu'un des points d'entrée respectivement $i_1, \ldots, i_n$ est exécuté.

Composants hybrides contrôle/donnée Nous présentons dans ce paragraphe plusieurs descriptions de composants complexes qui s'expriment à partir des flots de contrôle et de données auxquels on se rattache pour nos travaux pour des raisons de flexibilité.

Condition. Voici un exemple de connecteur complexe constitué de composants élémentaires de flot de données et de flot de contrôle. Ce connecteur étend en quelque sorte le composant de disjonction en permettant de émettre un événement $\hat{}o_1$ ou $\hat{}o_2$ et l'information i associée à l'exécution du point d'entrée $Seq.p$ respectivement si $i = val_1$ ou $i = val_2$.



4.2.3.3 Container et Designer

L'architecture de notre modèle LCA se compose de deux entités logicielles qui permettent de gérer les assemblages de composants.

Container La première entité logicielle appelée container encapsule l'assemblage de composants. Un container permet la création et la destruction d'assemblages de composants et la modification des connecteurs.

La figure 4.5 résume et relie les entités du modèle LCA à la notion de container. Les descriptions de composants qui peuvent être ajoutés dans un container dépendent du contenu du dépôt.

Designer Un designer est une entité logicielle qui permet la manipulation interactive/réactive d'un assemblage de composants.

Nous avons créé plusieurs designers encapsulés dans des services : designer graphique (modification de l'interface graphique), designer visuel (modification interactive de l'assemblage), designer pour les aspects d'assemblage (adaptation réactive des assemblages de composants), etc.

4.3 Service composite SLCA

Un `service composite` SLCA s'appuie sur un assemblage interne de composants légers pour gérer la composition de plusieurs autres services s'appuyant sur les événements et pour concevoir l'interface d'un nouveau service composite de plus haut niveau. Nous appelons ce paradigme Architecture de Service à Composants Légers (SLCA) qui s'appuie fondamentalement sur les événements et propose un minimum de fonctionnalités contrairement à SCA [185].

Un service composite gère alors un assemblage dynamique de composants légers et offre également des services s'appuyant sur l'assemblage en question. Pour cela, nous introduisons des composants spécifiques pour l'interception et la diffusion d'événements et l'exécution de points d'entrée dans l'assemblage géré par le service composite. Nous les appelons les composants sondes.

4.3.1 Composants spécialisés dans l'interaction avec les autres services

Les composants sondes sont de deux types :

- les composants source d'événements (interception d'événement et interception d'exécution de points d'entrée) pour le service du container
- les composants puits d'événements (déclenchement d'événements et déclenchement de points d'entrée) pour le service du container

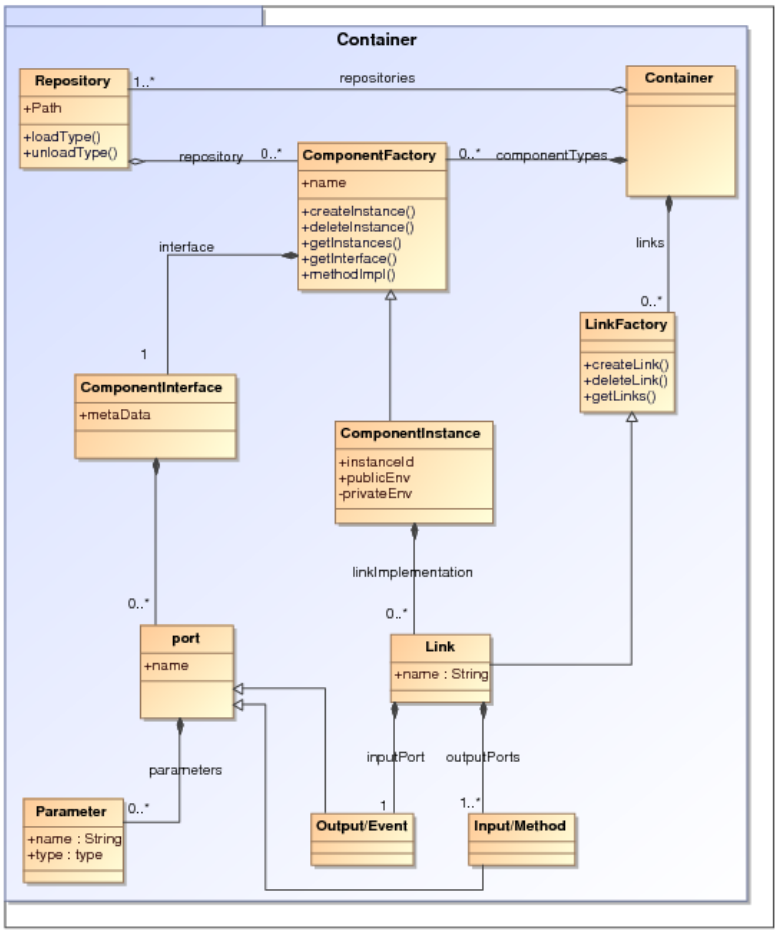


FIGURE 4.5 – Modèle LCA : composants légers assemblés

Ces composants permettent à d'autres services d'intervenir sur les flots d'événements de l'assemblage de l'application. Ils correspondent à autant d'événements et de points d'entrée que l'on peut invoquer sur le service composite au travers du service correspondant.

Les autres événements de l'application gérés par le container, comme les événements de démarrage de l'application et des modifications élémentaires de l'assemblage.

Les opérations sont proches d'un ADL :
- add_component Type Id
- remove_component Id
- link Id source_Event Target_Point
- unlink Id

Le service composite présente donc deux interfaces de service : l'une permet d'interagir avec l'assemblage de ses composants et l'autre avec le flot de ses événements.

Hiérarchie. De plus, puisqu'un assemblage est à l'intérieur d'un service composite, il contribue à une interface de service et aux fonctionnalités que d'autres services peuvent utiliser. Un service composite peut donc créer des applications communiquant avec d'autres services composites. Le concept de hiérarchie est alors introduit au travers les couches de service.

4.3.2 Interface de contrôle pour les modifications

Les applications sont construites par composition de services à travers des assemblages de composants. Le modèle SLCA consiste en encapsulation d'un assemblage de composants légers dans un service composite.

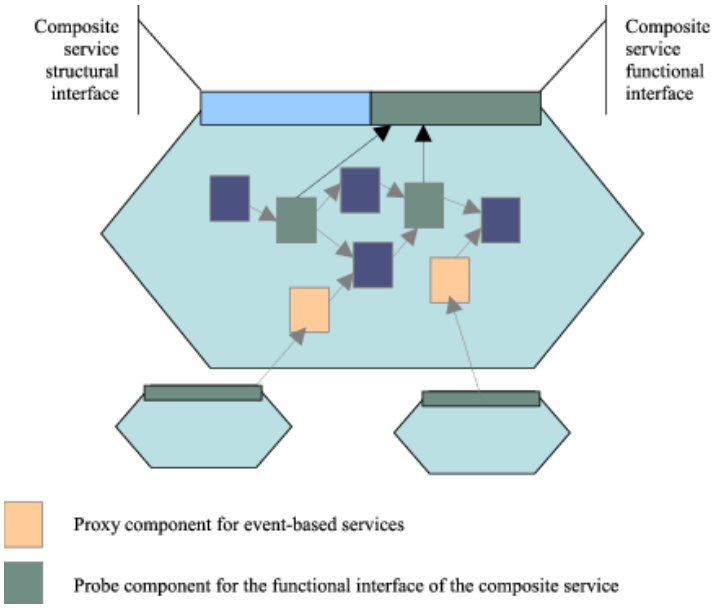


FIGURE 4.6 – Modèle SCLA : Encapsulation d'un assemblage de composants dans un service composite

Lorsqu'un service composite est doté également d'une interface de contrôle, il permet d'adresser la problématique des modifications dynamiques de l'assemblage interne. Il offre des points d'entrée pour ajouter et retirer des composants, leur description, des connecteurs et permet également d'obtenir des informations sur les composants. Par conséquent, un service dit client (qui peut également être composite) peut agir sur la structure du service composite. L'adaptabilité structurelle est alors rendue possible.

Designer et Container. Nous avons synthétisé deux notions pour désigner ces services particuliers liés à l'adaptation [186], des notions qui émergent d'une méthodologie de développement dynamique à base de composants logiciels [187]. Nous appelons un service composite qui dispose d'une interface de contrôle pour les modifications container. Nous appelons un service qui utilise des containers pour effectuer des modifications structurelles des designers.

4.3.3 Système informatique ambiant

Notre modèle fédère deux principaux axes :

- services basé sur les événements
- un assemblage de composants légers à l'intérieur d'un service composite

Notre modèle consiste en une approche classique par composants pour construire des services composites de plus haut-niveau ce qui incrémente le graphe de coopération de services de l'application. Cette approche convient à la conception d'application dans un contexte usuel, commun et connu. Nous appelons cette approche *composition pour les services de haut-niveau.*

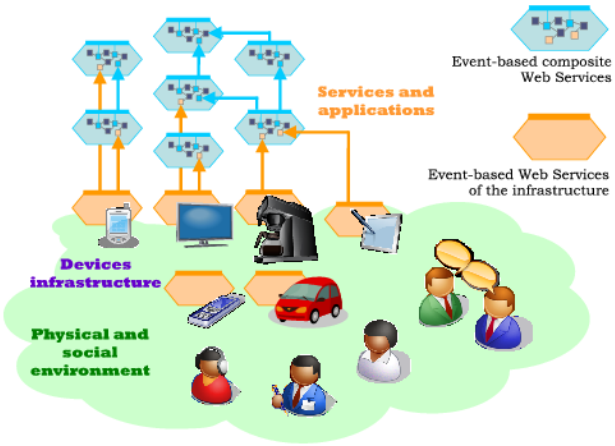


FIGURE 4.7 – Graphe de services composites d'assemblages de composants légers basés sur des événements

Dans cette architecture, les applications d'informatique ambiante sont des graphes de services composites s'appuyant sur un réseau de communication événementielle. Ainsi, notre solution permet d'adopter une approche par composition ou par adaptation pour concevoir dynamiquement des applications pour l'informatique ambiante. C'est une approche globalement orientée-composants qui a pour but de concevoir des services composites.

En résumé, le modèle SLCA apporte la notion de distribution à travers les services au modèle de composant LCA.

Cependant, ce modèle de service composite ne suffit pas à lui seul pour construire un système adaptatif. Nous allons voir dans le chapitre suivant comment construire un tel système.

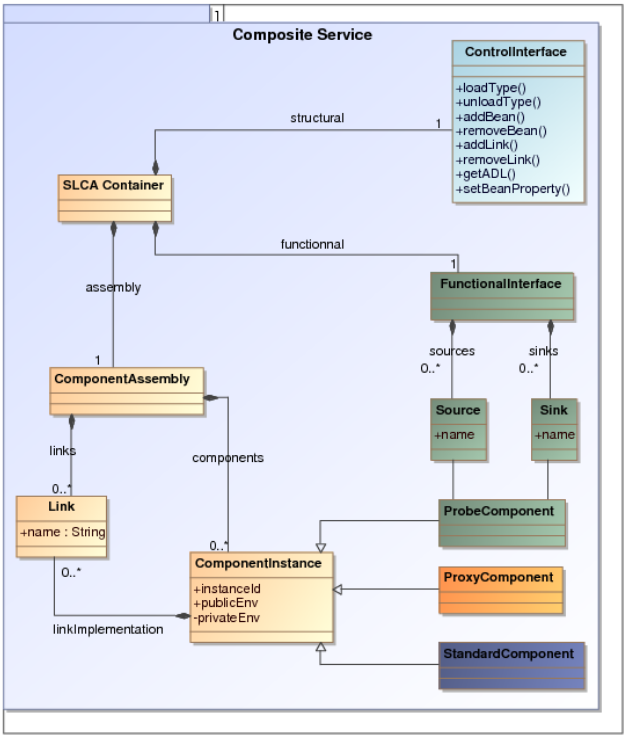


FIGURE 4.8 – Modèle SLCA : Distribution à travers les services et modèle LCA

Chapitre 5

Aspects d'assemblage

Sommaire

5.1 Introduction

Nous proposons un modèle d'adaptation utilisant un concept original appelé aspect d'assemblage. Ce concept permet de tisser des schémas d'adaptation *indépendants* et *transversaux* qui se fusionnent en suivant des règles logiques en cas de conflit. Ces schémas sont applicables à tous les services composites de l'application, qui ne seraient pas totalement connus à priori. Le modèle d'adaptation prend en compte au mieux les principes liés à l'adaptativité énoncé dans le chapitre de synthèse et met en avant le principe de modification structurelle.

Les adaptations sont un ensemble d'aspects d'assemblage qui ont été conçus à partir d'une connaissance à priori mais partielle des services sur lesquels ils s'appliquent. Le but de l'AOP est de modifier un programme. Notre approche consiste, elle, à modifier la structure d'un assemblage de composants dans deux cas : un assemblage qui représente localement une application (p. 87) ou un assemblage qui est encapsulé dans un service composite (p. 90).

5.2 Comparatifs des aspects

Ce comparatif permet, dans un premier temps, de voir les relations entre la programmation par aspect et les aspects d'assemblage, puis étudie l'implication des aspects d'assemblage dans chaque processus d'adaptation.

5.2.1 Programmation orientée aspect classique

La programmation par aspect est un paradigme qui adresse explicitement la modularisation des préoccupations transverses dans les systèmes logiciels complexes. L'hypothèse de travail est que les mécanismes de modularité sont supportés jusqu'à maintenant par l'*unique* décomposition hiérarchique logicielle basée, par exemple, sur une structure de données (décomposition orientée objet) ou sur la fonctionnalité offerte (décomposition fonctionnelle). L'AOP fournit une alternative à cette décomposition. Elle introduit une nouvelle unité de modularité appelée `aspect` qui a pour but de modulariser les préoccupations transverses des systèmes complexes au moyen de trois concepts-clés : le point de coupe, le greffon et le tisseur.

Point de coupe. Les points de coupe représentent l'ensemble des points de l'exécution d'un programme appelés `points de jonction`. Par exemple, puisque AspectJ est une extension orientée aspect du langage Java, ses points de jonction sont définis selon la manière d'exécuter les programmes orientés objet, c'est-à-dire en s'appuyant sur les appels de méthode, les appels au constructeur, les attributs, etc. Afin de modulariser les préoccupations, les points de coupe sont définis en tant que prédicats sur les points de jonction. Ils peuvent sélectionner les points d'exécution relatifs à l'exécution d'une méthode basée sur le type de leurs paramètres ou leur valeur de retour, sur leur nom, etc. Les langages d'aspect viennent avec des constructions prédéfinies de points de coupe (appelées `pointcut designator` en AspectJ).

Greffon. La fonctionnalité commune transverse d'un ensemble de points de jonction est spécifiée au moyen d'un greffon. Le greffon est un bout de code qui est exécuté lorsqu'un point de jonction de l'ensemble identifié par le point de coupe est atteint. Le greffon peut être exécuté avant (before), après (after), ou à la place (instead) du point de jonction sélectionné par le point de coupe. Avec un greffon autour (around), un aspect peut intégrer l'exécution d'un point de jonction intercepté au milieu d'un code qui sera donc exécuté autour de ce dernier (mot-clé proceed en AspectJ). Un greffon a également accès au contexte d'exécution d'un point de jonction donné. Ainsi, en plus d'identifier un point de l'exécution du programme, un point de coupe déclare également les parties du contexte du point de jonction disponible au code de l'aspect. AspectJ offre par exemple un langage dédié pour fournir ce contexte à l'appel de méthode, aux arguments passés à l'appel, etc.

Aspect. Un aspect consiste en général à plusieurs déclarations de point de coupe et d'un greffon qui leur est associé. De plus, il peut définir son propre état ainsi que ses propres méthodes qui peuvent être utilisées à leur tour dans le code du greffon.

Tissage. L'intégration des aspects dans l'exécution de la fonctionnalité de base est appelée tissage. En AOP statique comme AspectJ, le tissage se passe à la compilation ou au chargement. En AOP dynamique, les aspects peuvent être déployés à l'exécution.

```
public aspect Logging {
  // où ?
  pointcut loggableMethods(Object o): call (* bar(..)) && this(o);

  // greffon (comportement à insérer) :
  before(Object o):loggableMethods(o) { // quand ?
    // quoi ?
    System.out.println("code_appelé_par_l'objet_" + o.toString());
  }
}
```

FIGURE 5.1 – Un aspect de Logging en AspectJ

La figure 5.1 montre un exemple simple de logging en AspectJ. Cet aspect définit un point de coupe, loggableMethod qui spécifie où la préoccupation de logging devrait joindre l'exécution de la fonctionnalité de base. Dans cet exemple, les points de jonction intéressants sont les appels (call) à toutes les méthodes nommées *bar* indépendamment de la classe dans laquelle ils sont définis, de leur valeur de retour, ou du nombre et du type des paramètres. L'objet qui exécute l'appel de méthode est exposé au greffon par l'utilisation du mot-clé *this*.

L'aspect de logging permet à une préoccupation transverse d'être modularisée. Si la fonctionnalité de logging est requise à d'autres endroits du code, nous devons juste modifier la définition du point de coupe. Dans une solution orientée-objet mais non orientée-aspect, il aurait été nécessaire d'aller à tous les endroits où le logging est requis et de modifier la classe appropriée. Avec les aspects, un seul morceau de code doit être modifié et de plus, la fonctionnalité de logging peut être activée ou désactivée sans modifier le code de l'application de base.

Les objectifs de la programmation par aspect pour des applications à base de composants sont : (1) fournir un support pour la définition d'aspects et leur réutilisation, (2) pouvoir utiliser des composants développés par des tiers (préservation d'un modèle de composition blackbox) et (3) la séparation stricte entre l'implémentation des aspects et leur liaison aux composants (développement de bibliothèques d'aspects indépendants des composants). Cette approche classique des aspects s'appuie sur une réalisation par insertion/retrait de code dans l'application ; nous pouvons cependant envisager une modification originale comme une reconfiguration, c'est-à-dire que ce sont des endroits bien précis de l'architecture que nous modifions.

5.2.2 Aspects pour la reconfiguration

Il existe deux différences fondamentales entre un aspect classique [188] et un aspect d'assemblage. Tout d'abord, un aspect classique est une partie d'un programme qui est transverse au concept de module du langage : l'encapsulation objet dans le cas de la POO. Un aspect viole donc par définition le principe d'encapsulation. Un AA s'appuie sur ce même principe en agissant cependant à l'encontre de l'encapsulation des *assemblages* de composants. Il modifie soit la structure d'un assemblage de composants représentant une application locale (LCA), soit la structure d'un assemblage encapsulée dans un service composite (SLCA).

Le tissage d'AAs opère uniquement sur la structure des assemblages de composants. Un AA est lui-même un assemblage de composants logiciels dont certains composants représentent potentiellement des points de coupe et d'autres sont réservés au contrôle du tissage. Ensuite, dans le modèle AA, un `point de jonction` est toujours un point du flot d'événements (opération, événement), mais pas de la même granularité qu'en AOP. Un point de jonction est un composant logiciel. En AA, le `tissage` est scindé en deux opérations : la `composition` d'`assemblages` intermédiaires et la résolution de conflits. Cette distinction permet de rendre la résolution de conflits indépendante du tissage.

```
AA Chauffage {
  // où ?
  pointcut {
        gestion_maison = gestion_maison*
  }

  // greffon (la nouvelle structure) :
  add chauffage : ComposantChauffage
    // description d'un assemblage de composants :
  gestion_maison.Initialisation →seq( call , chauffage.On )
  chauffage.Consommation →gestion_maison.Conso
}
```

FIGURE 5.2 – Un aspect d'assemblage pour la prise en compte d'un système de chauffage

La figure 5.2 montre qu'un aspect d'assemblage permet d'exprimer les deux principales notions des aspects : le point de coupe et le greffon. Le greffon d'un aspect d'assemblage désigne la structure d'une application, alors que le greffon d'un aspect classique décrit le comportement de l'application à un moment donné. En particulier, dans cet exemple, le point de coupe désigne le composant de gestion d'une maison auquel on fait correspondre la variable *gestion_maison*. Au niveau structurel, on crée un composant *chauffage* que l'on rattache au composant de gestion (initialisation et gestion de la consommation).

La recherche classique sur les aspects concerne l'unification des composants et des aspects. Les aspects d'assemblage ne consiste pas à un portage des aspects dans le paradigme à composants [189, 190], mais plutôt à une utilisation des mécanismes des aspects pour reconfigurer des assemblages de composants. C'est un mécanisme de composition d'assemblages de composants. Par rapport aux interactions [30], l'expression des AAs est indépendante des opérateurs du langage qui correspondent à des composants logiciels spécifiques. Nous proposons une modélisation indépendante du langage et liée à la reconfiguration d'assemblages de composants.

> *En conclusion, le modèle d'aspects d'assemblage permet la composition d'assemblages de composants. Ces assemblages peuvent être exprimés à un plus haut niveau d'abstraction grâce aux notions de points de coupe.*

5.3 Les aspects d'assemblage

La modélisation des modifications logicielles à partir d'aspects par une reconfiguration résout les problèmes de blackbox et fait émerger une nouvelle façon de programmer. Nous présentons dans cette section les caractéristiques originales des aspects d'assemblage :

Impact global. Nous pouvons construire entièrement une application à partir d'aspects d'assemblage. Dans les approches classiques [123], les greffons sont rattachés à une application de base. En informatique ambiante, l'application de base peut ne pas exister. Elle est construite en fonction de l'environnement. Ainsi, l'expression des adaptations doit être la plus générale possible.

Préservation des boîtes noires. Les aspects d'assemblage préservent l'encapsulation des composants blackbox. Nous avons vu qu'au niveau du modèle de services, nous prenions en considération la notion de service *blackbox* qui est la conséquence de la présence de dispositifs physiques dans l'espace ambiant. Ainsi, les manières de modifier l'application se trouvent contraintes à la simple structure de l'assemblage de composants. Donc, cette prise en compte de services *blackbox* implique une modification uniquement structurelle lors des projections des modifications transverses.

Développement modulaire. Les aspects d'assemblage permettent un développement modulaire indépendant du contexte et ne comportant aucun état interne. Autrement dit, la sélection et l'application d'un de ces aspects d'assemblage avec le même assemblage initial produit exactement le

même résultat, quelque soit le moment et le contexte de la sélection. Les AAs sont indépendants de l'ordre dans lequel nous les sélectionnons. On dit que les AAs ont une approche 'déclarative'. Les modèles existants comme SAFRAN [123] se focalisent sur les modifications à effectuer lors d'un changement dans l'environnement physique. Le tissage de propriétés [131], prend en entrée un programme de base et un aspect qui représente une propriété comme la sûreté. L'aspect représente alors les traces d'exécution autorisées. Nous ne tissons pas des propriétés sur un programme. Nous affectons des propriétés à la fonction de tissage en conférant à l'opération d'adaptation la propriété mathématique de symétrie.

Paradigme logique. Les aspects d'assemblage permettent de gérer de manière déclarative les conflits d'assemblage de composants décrits dans les AAs. La gestion des conflits entre plusieurs adaptations est un enjeu important pour obtenir un système adaptatif utilisable. Nous présentons deux mécanismes de résolution s'appuyant sur des règles logiques qui permettent de gérer des conflits d'assemblage au niveau de l'exécution des composants.

Nous avons construits à partir de ces caractéristiques un modèle d'adaptativité et un tisseur que nous allons voir dans les sections suivantes.

5.3.1 Modèle d'adaptativité

Les motivations et la mise en œuvre de la notion d'aspect a évolué depuis sa création en 1997. L'AOP avait pour but original de minimiser la dispersion du code en réduisant le nombre d'instructions à écrire pour réaliser un programme complexe. En analysant de plus près les préoccupations visées, on a constaté que ces préoccupations étaient souvent de nature non fonctionnelle. Mais, cela dépendait du domaine d'application que l'on considère. Ainsi, on peut également considérer l'aspect comme une simple modularité transverse. En tant que telle, un aspect peut être déclenché non seulement par le programme, mais également par des événements *exogènes* (sollicitation de l'environnement physique extérieur au programme). Enfin, nous pouvons considérer l'aspect comme un paradigme de programmation à partir duquel on peut construire entièrement une application.

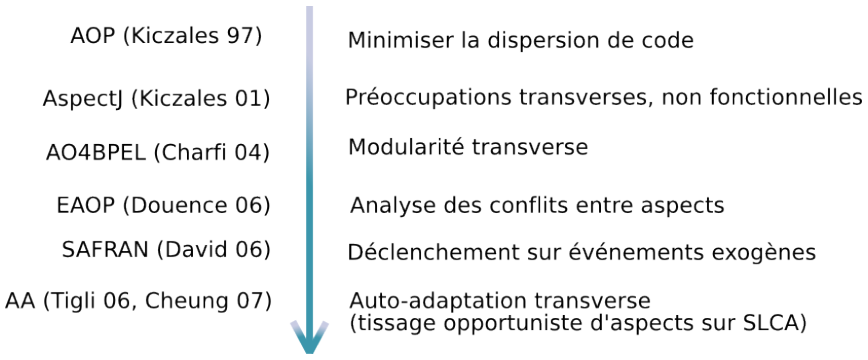


FIGURE 5.3 – Aspects d'assemblage : une approche inspirée des aspects

En particulier, les motivations des aspects ont changé. En effet, de l'analyse d'un problème de complexité d'un système d'information, nous nous intéressons à présent à la complexité de l'adaptation dynamique logicielle en informatique ambiante.

De manière générale, nous constatons que l'utilisation des aspects sert à modifier à priori le comportement d'une application existante. L'originalité de notre démarche est que l'on dispose d'un ensemble d'aspects d'assemblage prédéfinis dont les points de coupe servent de conditions de déclenchement de l'adaptation et donc, de mécanisme d'adaptativité.

Le concept d'aspect d'assemblage se définit en trois volets :

- Des points de coupe de l'assemblage à modifier (assemblage d'origine)
- Un greffon : schéma d'assemblage à intégrer
- Un tisseur : composition des schémas à intégrer à l'assemblage

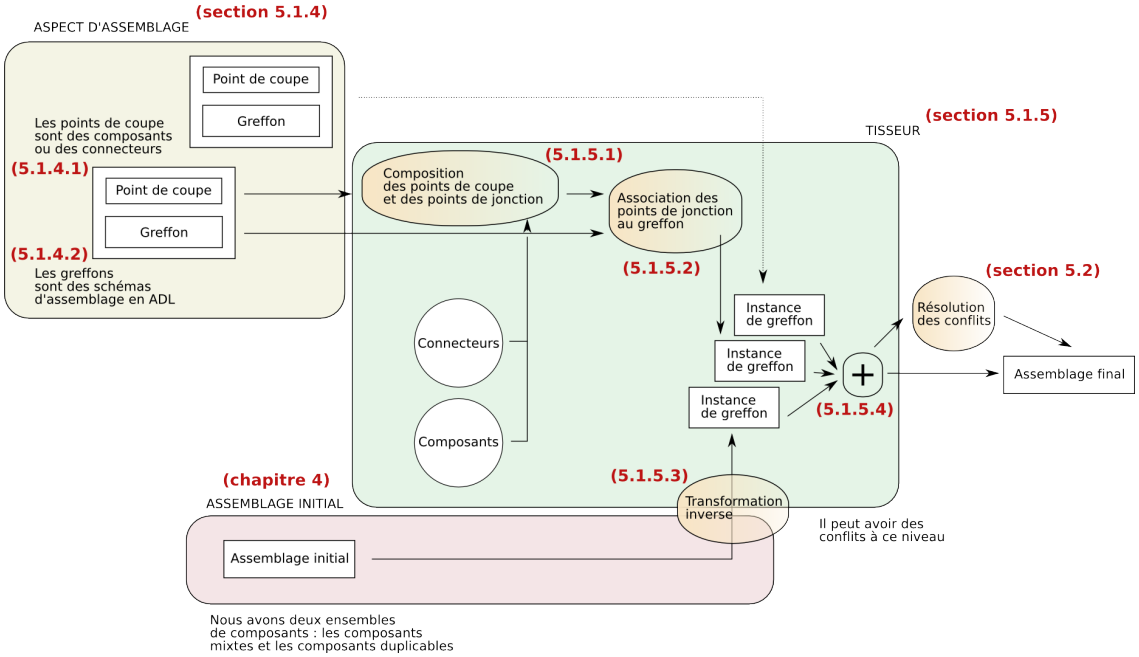


FIGURE 5.4 – Schéma général de la mise en œuvre des aspects d'assemblage

Le premier volet identifie les points d'ancrage de la modification d'assemblage où les greffons vont être intégrés. Le deuxième volet décrit l'assemblage à intégrer, en utilisant des opérateurs spécifiques au greffon. Le troisième volet consiste à traduire les spécifications décrite dans le second volet en un ensemble de modifications élémentaires qui vont entraîner une modification de l'assemblage d'origine.

5.3.1.1 Les points de coupe des AAs

Un point de coupe pour les aspects d'assemblage permet de désigner des endroits spécifiques de l'assemblage. Nous voyons d'abord un modèle général du point de coupe et puis analysons le cas particulier d'utilisation des expressions régulières pour effectuer le matching.

A. Point de coupe AA général : Un point de coupe permet de spécifier l'endroit du logiciel où est inséré un greffon par le tisseur. C'est une combinaison de points de jonctions dont le but est d'interfacer chaque aspect d'assemblage avec l'application.

Nous décrivons un point de coupe par une succession de filtres sur les composants de l'application. Ces filtres prennent en entrée la liste de tous les composants de l'application et produisent une ou plusieurs listes de composants.

L'application du greffon ne sera pas juste une juxtaposition du schéma sur un assemblage de base mais le résultat d'une duplication du greffon en fonction du nombre de composants blackbox qui ont été sélectionnés et de la juxtaposition des schémas dupliqués sur l'assemblage de base.

B. Cas particulier du matching syntaxique par expression régulière : Les points de coupe des AAs décrivent, par une expression régulière, les points de jonction où le greffon doit se positionner. Ces expressions régulières portent sur les identifiants des composants et sur leurs ports. Un point de coupe est donc représenté par cette expression régulière. Un point de jonction est un des identifiants de l'assemblage désigné par l'expression régulière.

Nous avons défini une grammaire :

```
1 <variable>:=(nom du filtre) parametres ; [ | autre filtre ... ]
```

FIGURE 5.5 – Grammaire

Ainsi par exemple, nous pouvons spécifier le point de coupe suivant :

```
1 isalon := /i*/
```

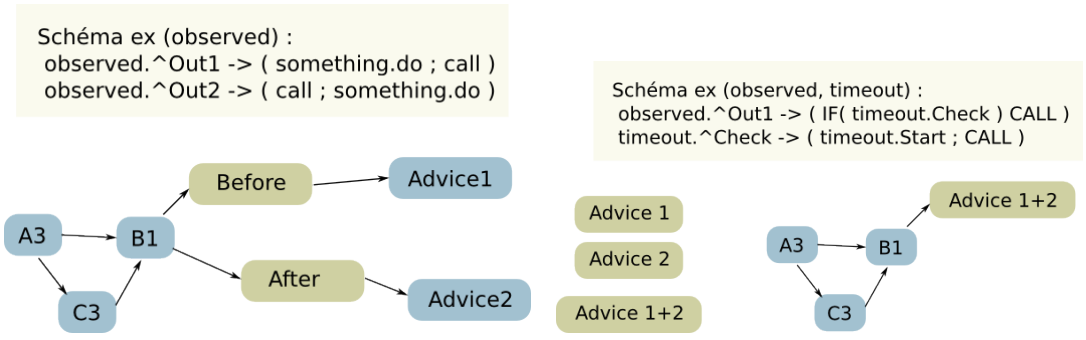
FIGURE 5.6 – Exemple de point de coupe

Instanciation d'un schéma sur un service composite. Grâce aux points de coupe, le mécanisme de tissage des aspects d'assemblage instancie le schéma à plusieurs endroits de l'assemblage de base.

Nous avons deux cas : lorsqu'une seule liste est construite, le greffon est intégré seulement une fois dans l'assemblage de base et les variables symboliques sont syntaxiquement remplacées dans le greffon afin de correspondre aux points de jonction de l'assemblage de base. Si plusieurs listes sont produites, le greffon est instancié autant de fois et chaque ensemble de variables, avec une occurrence de chaque point de jonction, est respectivement remplacé.

5.3.1.2 Les greffons des AAs

Nous présentons la notion générale de greffon en tant que fabrique de sous-assemblages. Nous considérons deux types de greffons : les greffons blackbox (fig. 5.7(a)) et greybox (fig. 5.7(b)).



(a) Greffons blackbox

(b) Greffons greybox

FIGURE 5.7 – AA : deux types d'interactions au niveau des greffons

Les greffons blackbox encapsulent leurs fonctionnalités auxquelles nous n'avons accès qu'à travers leurs interfaces. Nous ne pouvons gérer que l'ordre d'appel entre ces fonctionnalités. C'est ce que nous appelons le *tissage externe* entre des greffons blackbox.

Les greffons boîte-grise exposent partiellement la sémantique de l'assemblage de composants qui les constitue : on ne connaît le fonctionnement que d'une partie des composants de l'assemblage. Étant donnée cette connaissance, nous pouvons alors travailler sur une manière de les composer manuellement ou automatiquement : c'est ce que nous appelons la *fusion* des greffons boîte-grise.

Dans ce dernier cas, cela nous ouvre la possibilité de fusion en cas de conflit.

Cette notion est ensuite illustrée à travers deux mises en œuvre : l'un à partir du langage de spécification des interactions ISL [30] et l'autre est un modèle de coordination que nous avons mis au point en s'inspirant des travaux sur la coordination des comportements [191].

A. Greffon, fabrique de sous-assemblages : De manière générale, un greffon est une entité logicielle définissant le comportement d'un aspect associé à une coupe. Le comportement décrit par un greffon est exécuté à l'emplacement indiqué par un point de jonction.

B. Mise en œuvre à partir d'ISL, ISL4WComp : Les greffons ISL4WComp peuvent par exemple reposer sur un schéma d'assemblage qui s'inspire du langage de spécification d'interactions ISL [30]. ISL est d'une part un langage de description de schémas d'interactions entre des composants logiciels. Notre modèle adapte ces spécifications pour la prise en compte d'interactions basées sur des événements.

D'autre part, ces langages ont la particularité d'être composables : plusieurs schémas d'interactions peuvent se composer et fusionner en un seul schéma combinant leur comportement respectif. Nous exploitons la propriété de *symétrie* [192] de cette composition, c'est-à-dire que l'ordre d'application des aspects n'influence pas le résultat de leur composition. En effet, cela nous permet de faire face au caractère imprévisible de l'espace ambiant, puisqu'à chaque changement de l'espace, l'application des AAs est recalculée. Notons que cela n'interdit pas l'analyse de l'ordre dans lequel les dispositifs de l'infrastructure se sont connectés au système.

Nous avons défini deux façons de modifier le comportement d'une application :
– redirection d'un point d'entrée
– redirection de la diffusion d'événement
Dans les deux cas, on associe un nouveau comportement avec les opérateurs du langage :

Opérateurs	Description
... ; ...	Exprime une séquence
... || ...	Exprime un ordre indifférent entre les composants
if ... else ...	Exprime une condition
call ...	Désigne l'événement ou le point d'entrée correspondant au point de jonction
delegate ...	Désigne l'émission d'un événement ou l'exécution d'un point d'entrée
^...	exprime l'émission d'un événement

Tableau 5.1 – Opérateurs pour ISL4WComp

Un greffon décrit un ensemble de règles portant sur les points de jonction (l'émission d'un événement ou l'exécution d'un point d'entrée). Certains mots-clés du langage comme call et delegate permettent de contrôler la manière dont la composition des schémas d'assemblage va être effectuée. Ces mots-clés se rapprochent alors des mots-clés comme before, around et after de l'AOP.

La composition des schémas d'assemblage correspond à la mise en œuvre des travaux formels sur la composition des règles logiques [193].

```
advice Ex (observed):
  alarm : AlarmDesc ( timeout = 25 ) ;
  observed.^DoAction →
    ( if (alarm.CheckTimeout) call )
  alarm.CheckTimeout →
    ( alarm.Trigger ; call )
```

FIGURE 5.8 – Exemple de greffon ISL4WComp

Description de l'exemple ISL4WComp. Le greffon d'adaptation appelé **Ex** crée un composant **alarm**, redéfinit des ports de communication et s'applique à un ensemble de composants observés symboliquement représentés par la variable **observed** :

Premièrement, le greffon ajoute un composant **alarm** qui correspond à la description **AlarmDesc**. La propriété **timeout** de ce composant (dont l'unité est la seconde) est initialisée à la valeur 25. Le greffon redéfinit ensuite l'événement **^DoAction** du composant **observed** en spécifiant que les actions pouvant être définies par d'autres adaptations sont exécutées si et seulement si le composant **alarm** l'autorise à travers l'opération **CheckTimeout**. Deuxièmement, le greffon redéfinit le point d'entrée CheckTimeout de l'alarm, ce qui signifie qu'avant l'exécution du point d'entrée qui est peut-être connecté à d'autres composants, l'alarme doit être déclenchée, c'est-à-dire que le point d'entrée Start doit être exécutée.

Ces spécifications sont traduites en un ensemble de modifications structurelles élémentaires que l'on appelle MSE. Toute modification peut être vue comme une transformation d'un assemblage vers un nouvel assemblage. Ainsi, un service client implémentant les AAs peut par exemple communiquer les MSEs à un service composite afin de l'adapter.

C. Mise en œuvre d'un langage de coordination (BSL) : Le langage de coordination est composé des opérateurs d'arbitrage, d'ordonnancement et de déclenchement d'événements (tableau 5.2).

Opérateurs	Description
... ARB ...	Opérateur d'arbitrage
... RR ...	Opérateur d'ordonnancement temporel (H)
Event(...)	Exprime le déclenchement d'un événement
nop	Ne fait rien

Tableau 5.2 – Opérateurs pour BSL

```
advice Ex (observed):
  ARB(alarm1.^CheckTimeout,alarm2.^CheckTimeout)
     →obs.DoAction
  RR(c1.^a,c2.^b)
     →alarm.Trigger
```

FIGURE 5.9 – Exemple de greffon BSL

Description de l'exemple BSL. Un arbitrage est effectué entre deux événements envoyés par deux alarmes. Seuls ces événements ont le droit d'exécuter le point d'entrée DoAction et la connexion de l'alarm1 est prioritaire sur celle de l'alarm2. Un ordonnancement est effectué lorsque les événements a et b arrivent vers Trigger. a est toujours émis avant b, peu importe l'ordre réel d'émission des événements.

5.3.2 Tisseur d'aspects d'assemblage

La partie qui est mise en œuvre doit être décrite, mais de manière indépendante du modèle qui permet d'exprimer les points de coupe et les greffons. La composition d'assemblages de composants est une expression qui construit systématiquement un unique assemblage résultant à partir de plusieurs assemblages de composants intermédiaires.

Nous avons illustré la composition d'assemblages par un exemple théorique. Deux assemblages notés A et B sont composés pour donner un assemblage C. Les deux assemblages de départ sont en conflit, car ils ont deux points de jonction en commun qui sont les composants a et c. Ainsi, l'assemblage résultant C contient des composants indiquant les conflits au niveau du port de sortie 1 de a et du port d'entrée 1 de c.

Notre objectif est de construire un modèle suffisamment abstrait pour pouvoir être indépendant des plates-formes et ayant suffisamment de concepts proches de celles-ci pour être opérationnel.

Puisque nous travaillons sur la simplification du processus de développement et essayons de mettre en place des comportements prédictifs au sein de l'évolution de l'application, il nous faut une approche à la fois simple pour pouvoir être embarquée dans les plates-formes visées et pouvoir y exprimer nos propriétés.

5.3.2.1 Matching de point de coupe et instanciation de greffon

Jagadeesan [194] utilise par exemple pour décrire des points de coupe une logique simple s'appuyant sur les primitives d'exécution et d'appel. Comme nous l'avons vu dans la partie 2, la manière de modéliser le traitement de l'adaptation varie selon les modèles [131, 22] et est dépendante des langages de programmation.

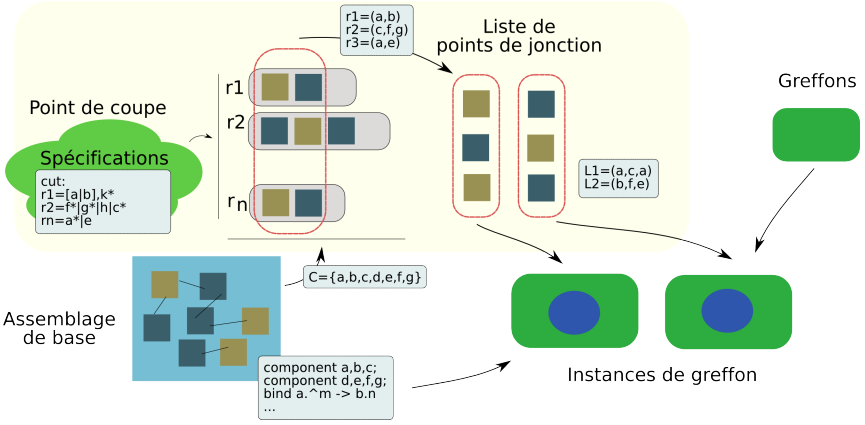


FIGURE 5.10 – Mécanisme de matching de point de coupe

Le matching de point de coupe se passe en deux étapes (Fig. 5.3.2.1) : construction des tables de points de jonction ($r_1 \dots r_n$) et l'établissement des listes de points de jonction. Ces étapes permettent d'extraire de l'assemblage de base les informations nécessaires pour construire des instances de greffon.

La construction des tables de points de jonction se fait par un filtrage de la liste des composants extraite de l'assemblage de base. Elle crée un ensemble de tables $r_1 \dots r_n$ qui contiennent une liste ordonnée de points de jonction. À partir des listes $r_1 \dots r_n$, nous en construisons de nouvelles comme L_1 et L_2 par exemple. Ces nouvelles listes se composent des éléments de rang i des listes $r_1 \dots r_n$. Ces nouvelles listes doivent toutes comporter le même nombre d'éléments. Elles constituent les paramètres pour les instances de greffon.

5.3.2.2 Composition d'instances de greffon

Nous définissons les concepts de composition et de fusion des AAs dans un formalisme logique de premier ordre. Nous définissons un ensemble de règles logiques pour chaque étape. Le mécanisme de composition s'appuie sur l'unification en logique. Nous allons voir dans un premier temps le principe de l'unification et présenter ensuite la composition d'instances de greffon.

Phase d'unification. L'unification est la notion centrale de la logique du premier ordre. L'unification de deux termes t_1 et t_2 consiste à trouver, s'il existe, un troisième terme t tel qu'on puisse passer de t à t_1 et à t_2 en instanciant (substituant) certaines variables. t est appelé un unificateur de t_1 et de t_2. Parmi les algorithmes d'unification [1], il est difficile de se mettre d'accord sur le

1. algorithme linéaire avec la plus faible complexité dans le pire des cas impliquant cependant une implémentation de taille importante [195], algorithme de Martelli et Montanari [196], de Robinson [197] pour la plupart des applications malgré leur complexité exponentielle dans le pire des cas.

meilleur algorithme pour une utilisation pratique. Nous nous appuierons sur l'algorithme de Martelli et Montanari pour sa simplicité de mise en œuvre. Cet algorithme travaille sur un ensemble fini d'équations notées :

$$E = \{s_1 = t_1, ..., s_n = t_n\}$$

Il travaille également sur un ensemble de substitutions noté θ. A l'instant initial, $E_0 = \{s = t\}$, où s et t sont des termes à unifier et $\theta = \emptyset$. L'algorithme choisit arbitrairement dans E une équation e et effectue des actions selon les règles suivantes :

1. si e est de la forme $f(s_1, ..., s_n) = f(t_1, ..., t_n)$, alors nous remplaçons e par les équations $s_1 = t_1$, ..., $s_n = t_n$.
2. si e est de la forme $f(s_1, ..., s_n) = g(t_1, ..., t_n)$ où g est différent de f, alors l'unification échoue.
3. si e est de la forme $X = X$, alors nous retirons e de l'ensemble E.
4. si e est de la forme $X = t$ où X n'apparaît pas dans t, alors nous ajoutons la substitution $X \leftarrow t$ à l'ensemble θ et nous appliquons la substitution $\{X \leftarrow t\}$ à l'ensemble E et à l'ensemble θ.
5. si e est de la forme $X = t$ où X apparaît dans t et $X \neq t$, alors l'unification échoue.

Cette procédure est répétée jusqu'à ce que l'ensemble E devienne vide ou bien que la procédure échoue. Nous noterons que, lorsque nous raisonnons en logique du premier ordre, où les variables ne peuvent être unifiées qu'à des constantes, savoir si deux termes t_1 et t_2 sont unifiables est décidable, c'est-à-dire que l'on sait le résoudre à l'aide d'un algorithme en un temps fini.

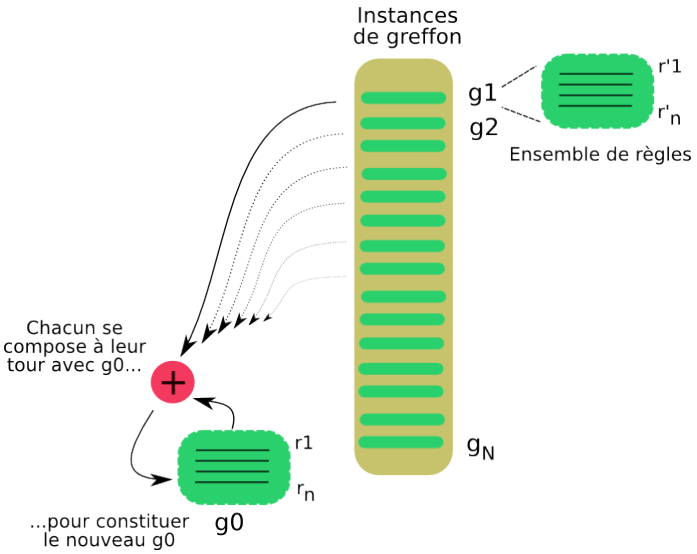


FIGURE 5.11 – Mécanisme de composition

Phase de composition. Un AA est modélisé par un ensemble de faits. L'algorithme d'unification est utilisé pour la composition des instances de greffon. La composition consiste à prendre, un par un, chaque instance de greffon g_i et à le composer avec un unique instance de greffon g_0. g_0 est l'instance de greffon initial (vide au départ). Le résultat de cette composition remplace g_0.

Un greffon est constitué d'un ensemble de règles logiques $r_1 \dots r_n$ qui vont devoir s'unifier.

5.3.2.3 Association des points de jonction au greffon

Ces schémas sont ensuite traduits dans un autre langage intermédiaire qui permet une écriture plus simple dédiée à un programmeur. Voici l'exemple du greffon1 exprimé dans ce langage de connexion. Nous précisons que cet exemple a été produit par l'outil associé (Figure 5.12).

```
advice greffon1 (isalon):
  isalon : Interrupteur ;
  isalon.^Up →( isalon.On )
```

FIGURE 5.12 – Greffon 1 en ISL4Wcomp

La grammaire de ce langage sera explicitée plus loin dans ce chapitre. Pour le moment nous pouvons d'ores-et-déjà l'analyser et l'interpréter. La première ligne nous donne des renseignements sur le nom du schéma qui est ici greffon1 ainsi que la liste des composants variables qui sont entre parenthèses. *isalon* est donc un composant variable. La ligne 2 spécifie que le composant isalon est un composant blackbox qui correspond à la description Interrupteur. La ligne 3 enfin spécifie la connexion entre l'événement ^Up avec le point d'entrée **On** du composant *isalon*. Ainsi toutes les connexions ainsi que les composants que nous retrouvons dans les schémas graphiques peuvent être représentés textuellement dans ce langage. Nous adoptons une approche *non-invasive* [198] qui ne s'appuie pas sur l'adjonction classique d'intercepteurs au niveau des interfaces. Cette approche s'effectue en trois étapes.

1ère étape : La première étape concerne la transformation d'un assemblage de base G_0 vers un aspect d'assemblage A_0. Cet aspect d'assemblage A_0 sera composé avec d'autres aspects d'assemblage.

2ème étape : La seconde étape concerne la spécification de modifications que nous voudrions appliquer à un assemblage de composants regroupé au sein d'un aspect d'assemblage noté A_i où $i \geq 1$. Cet AA contient les informations partielles portant sur l'application.

3ème étape : La troisième étape consiste à construire un aspect d'assemblage noté B_0 à partir de $A_0, \ldots, A_n$ où n est le nombre d'aspects d'assemblage qui modifient l'application initiale. Cet aspect est enfin traduit en une liste de modifications à porter sur l'assemblage de base G_0.

Les AAs de base comme A_0 et résultants comme B_0 sont les AAs qui ont une relation avec les modèles de composants. Typiquement, l'AA résultant est transformé en un assemblage de composant F_0.

5.3.2.4 Transformation inverse

Soit un assemblage de base à un instant t donné. Cet assemblage se situe toujours avant le premier calcul d'adaptation. Nous notons cet assemblage G_t. Lorsque l'algorithme se réinitialise, l'assemblage G_t est transformé en un greffon A_t selon des règles de transformation spécifiques. Ensuite, le moteur d'aspects d'assemblage effectue ses calculs d'adaptation en cohérence avec les spécifications fournies potentiellement par d'autres greffons. Ces calculs produisent un greffon résultant A_{t+1} qui est ensuite transformé en un assemblage de composants résultant G_{t+1}. Ces étapes sont regroupées dans la Figure 5.13.

Dans cet exemple, nous avons un assemblage de composants logiciels légers constitué de :
- deux composants interrupteurs **s0** et **s1** qui correspondent à la description **Switch** avec respectivement la propriété d'adresse dont la valeur est 39 et 40 (lignes 2 et 3),
- deux composants lampe **l0** et **l1** correspondant à la description **Light** aux adresses 56 et 57 (lignes 4 et 5).

Les composants interrupteurs envoient un message sur l'événement **^On** lorsqu'ils passent dans l'état actif et envoient un message sur l'événement **^Off** lorsqu'ils passent dans l'état inactif. Ces deux événements sont respectivement connectés aux points d'entrée **On** et **Off** des lampes. A la réception d'un message sur le point d'entrée **On** d'une lampe, celle-ci s'allume et la réception d'un message sur le point d'entrée **Off** éteint cette dernière. Dans le cadre expérimental, ces composants logiciels utilisés dans cet assemblage sont des proxies logiciels vers des services pour dispositif [28].

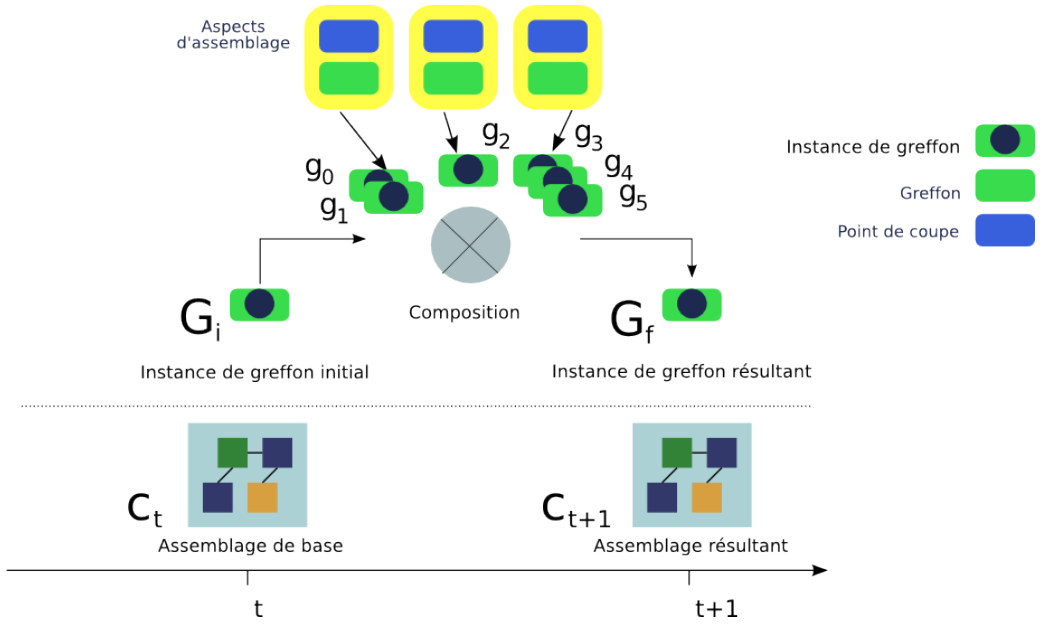


FIGURE 5.13 – Assemblage de base G_t

5.3.2.5 Détection de conflit

Nous donnons un exemple dans la figure 5.14 qui illustre un conflit. Dans cet exemple, les points de coupe P1 et P2 sont respectivement couplés aux greffons G1 et G2. Après le calcul des points de coupe, les variables u et e des greffons G1 et G2 valent respectivement $\{u1, e1\}$ et $\{u1, e2\}$. Donc, les deux adaptations proposent simultanément de modifier structurellement des connexions qui portent sur le composant $u1$. C'est à cet endroit que se situe le conflit et que l'algorithme de fusion doit intervenir afin de produire une solution en s'appuyant sur une base de règles logiques de composition.

```
pointcut P1:
  u ←/u1/
  e ←/e1/

pointcut P2:
  u ←/u1/
  e ←/e2/
```

```
advice G1 (u, e):
 u.^Out1 →( if (e.Check1) call )
 e.Check1 →( e.Start1 ; call )

advice G2 (u, e):
 u.^Out2 →( if (e.Check2) call )
 e.Check2 →( e.Start2 ; call )
```

FIGURE 5.14 – Exemple de détection de conflit entre deux greffons ISL4WComp

Dans la Figure 5.15, nous illustrons un exemple de fonctionnement de la composition-fusion. Dans ce cas, il va travailler avec trois aspects d'assemblage A_0, A_1, A_2. Le tisseur de travaille pas directement sur les points de coupe, cette partie est déléguée au moteur de point de coupe noté PC. Rappelons que le moteur de point de coupe permet d'extraire à partir des points de coupes de différents aspects d'assemblage comme par exemple P_0, P_1, P_2, les points de jonction J_0, J_1, J_2 dans l'assemblage de base G_0. Le tisseur prend donc en entrée les greffons d'adaptations $\alpha_0, \alpha_1, \alpha_2$

issus des aspects d'assemblage et les points de jonction J_0, J_1, J_2. À partir de là, il construit un unique aspect d'assemblage A_{012} composé des fonctionnalités exprimées dans chacun des aspects d'assemblage A_0, A_1, A_2.

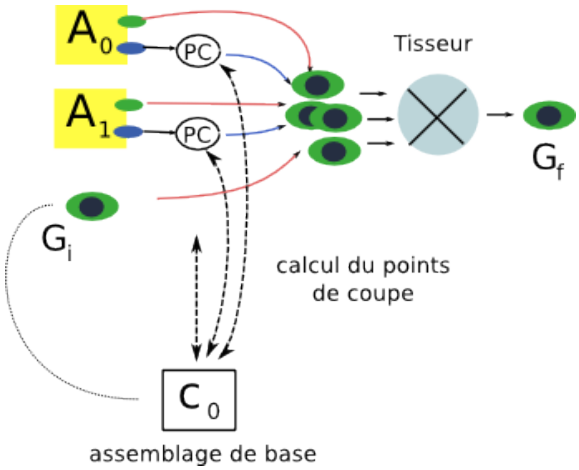


FIGURE 5.15 – Composition et fusion

On notera que l'architecture du tissage tel qu'il est présenté inclut par défaut l'AA A_0 qui est la transformation de l'assemblage de base G_0. Cela implique que les informations contenues dans l'assemblage de base sont traduites en A_0 et sont composées aux autres aspects d'assemblage afin de construire A_{012}. La composition ne s'applique pas uniquement aux AA multiples que nous voulons tisser mais également à l'assemblage de base. Cela permet d'inclure dans la phase de fusion et de gestion de conflit les fonctionnalités déjà présentes dans l'assemblage de base. La dernière transformation AA→Assemblage est par conséquent simplifiée et n'a pas besoin d'intégrer un mécanisme de gestion de conflits puisque ce dernier a été déjà effectué si nécessaire en amont par la fusion.

5.4 Gestion des conflits d'aspects

La résolution de conflits est associée à la composition d'assemblages de composants. Elle permet de gérer les cas précédemment énoncés où la superposition des assemblages de composants s'appuie sur les mêmes points de jonction. Il existe alors un conflit d'utilisation de ces composants logiciels. C'est à partir des composants spécifiques '$\oplus$' que le mécanisme de résolution de conflit va construire sa solution. Le résultat de la résolution de conflits est un assemblage de composants.

Reprenons l'assemblage C de l'exemple précédent contenant deux composants '$\oplus$'. Nous proposons de résoudre ces conflits en produisant une solution en terme d'assemblage aux points identifiés selon un ensemble de règles logiques pré-définies. Nous énonçons pour cet exemple deux exemples de règles logiques de manière intuitive :

1. *tout port en conflit avec le point d'entrée de f détient la priorité d'exécution et le flot de contrôle doit être retardé avant d'arriver à f*
2. *tout port en conflit avec l'événement de g doit être synchronisé et l'on doit effectuer une sommation de leurs paramètres avant toute diffusion*

La résolution de conflits produit à partir de ces règles un assemblage final noté D ayant certains conflits résolus. Dans cet exemple, un composant de séquence (**seq**) a été ajouté à l'événement de a suivi d'un composant retardant le flot d'événement (**delay**) avant d'atteindre f. De la même manière, on retrouve un composant de synchronisation et de mémorisation des paramètres (**Buffer**) après l'événement de g suivi d'un composant de sommation (**sum**).

Dans cette partie, nous nous intéressons aux conflits entre les préoccupations transverses [146]. Nous analysons graduellement les approches pour la résolution de ces conflits vers les approches les plus intrusives. Enfin, nous proposons notre approche qui s'appuie sur la notion de fusion [30].

5.4.1 Les conflits entre les aspects dans la littérature

Il y a plusieurs catégories de conflit entre les aspects [199] :
- Spécifications
- Entre aspects
- Entre le programme de base et l'aspect
- Entre les préoccupations

Ainsi, lorsqu'un conflit se passe au niveau des spécifications, il peut être soit accidentel au niveau des points de jonction soit au niveau de la récursivité. Entre les aspects, cela concerne l'exécution conditionnelle, l'exclusion mutuelle, l'ordonnancement statique ou dynamique. Entre les préoccupations, cela signifie qu'il y eu changement des fonctionnalités, des comportements inconsistants ou des anomalies de composition.

Un premier niveau de gestion de la superposition est la description d'un aspect d'assemblage par un composant blackbox [136, 189, 190]. Dans ce cas, on gère les conflits en spécifiant localement les aspects qui sont prioritaires. On parle de précédence d'un aspect sur un autre. Cette gestion est réalisée de manière externe au tissage en identifiant les composants blackbox prioritaires. Voici un exemple de gestionnaire de précédence entre les ports de composants :

---

blackbox1.Input + blackbox2.Input = (blackbox1.Input ; blackbox2.Input)
blackbox1.Input + A = (blackbox1.Input ; A)
blackbox1.Input + A = (blackbox2.Input in A) **and** (blackbox1.Input ; blackbox2.Input)

Ce programme est ensuite traduit en règles logiques et est utilisé par le tisseur. Par exemple, AspectualACME [200] propose deux opérateurs : précédence (priorité) et XOR. Hyper/J [201] propose une gestion par programmation de la résolution. Mais ces approches demandent soit l'apprentissage d'un nouveau langage dédié soit de choisir de manière ad hoc entre plusieurs adaptations.

5.4.2 Gestion des conflits d'aspects d'assemblage

Pour éviter de traiter les conflits de manière ad hoc, nous proposons une approche qui permet de composer les greffons en les considérant comme des whitebox. Des opérateurs spécifiques dont on connaît la sémantique de composition sont mis en œuvre. Cela permet de composer automatiquement les greffons.

5.4.2.1 Notre motivation

L'état de l'application à un instant donné ne doit pas dépendre de l'ordre d'application des AAs. C'est le but que nous nous sommes fixés pour favoriser une approche où l'on ne fixe pas à priori de priorité entre les adaptations, une approche où une partie des décisions peut être prise par le système tel que des choix de coordination purement logiques par exemple.

Pour ce faire, nous cherchons à mettre en œuvre la propriété de symétrie dans l'opération de composition des aspects d'assemblage. Cette propriété peut être décomposées en trois sous-propriétés qui sont l'associativité, la commutativité et l'idempotence. Nous proposons donc une approche qui permet de composer symétriquement des assemblages de composants. L'avantage pour notre approche est de promouvoir une approche déclarative à l'utilisateur final (cf. page 5.3) des aspects d'assemblage pour une adaptation dirigée par l'utilisateur ou par l'environnement.

5.4.2.2 Fusion ISL

Les opérations de fusion des greffons sont régies par un ensemble de règles logiques. Ces règles sont regroupées dans la matrice que nous retrouvons dans la figure 5.16. Les propriétés de cette fusion sont la commutativité, l'associativité et l'idempotence (cf. Annexe).

Nous avons construit, à partir du modèle de Berger [30], un nouveau modèle de fusion d'interactions qui s'applique à notre modèle de composant. Comme un aspect classique peut être représenté dans le modèle de composant, un comportement peut également être représenté dans notre modèle.

Un comportement est un arbre composé d'un sommet qui sont les entêtes des règles de réécriture. Ces entêtes caractérisent les méthodes à rediriger dans [30]. Chaque opérateur de Berger est modélisé par un composant dont la sémantique opérationnelle est la même.

Nous nous sommes inspirés de ce langage afin de construire un moteur de fusion qui s'appuie sur la fusion de programmes écrits sous la forme d'assemblages de composants. Chaque composant logiciel réalise dans l'assemblage un ensemble d'instructions. Le moteur de fusion d'ISL est un moteur de composition permettant de composer plusieurs schémas d'interaction indépendamment de leur ordre de fusion.

De part la capacité réduite des calculateurs envisagés, nous avons réduit le nombre d'opérateurs que nous utilisons pour la fusion au strict minimum (séquence, indéterminisme, conditionnel, call, message et délégation). Les règles de fusion sont également différentes et des propriétés supplémentaires sont en jeu (comme l'idempotence). La fusion est associative, commutative et idempotente et permet de gérer les cas pratiques de superposition de descriptions de scénarios applicatifs dans le cas d'un système informatique ambiant. Cette fusion est décrite dans ce chapitre.

> *Nous avons vu que les aspects d'assemblage décrivent des adaptations indépendantes. Ces adaptations sont exprimées en juxtaposant les schémas d'assemblage. Nous voyons dans cette section une fonction qui permet de fusionner plusieurs schémas d'assemblage dans le but de construire un unique schéma d'assemblage. Le schéma d'assemblage résultant peut être ambigu.*

Notion de pivot. Nous ajoutons aux spécifications la notion de pivot qui permet de contrôler la manière dont la fusion s'opère. Le pivot sert à guider la fusion afin qu'elle se fasse en des points précis des espaces.

Conflits non résolus. Lorsque plusieurs AAs utilisent les mêmes points de jonction, le mécanisme de composition détecte ces conflits et ajoute au niveau de ces points de jonction provoquant le conflit, un composant spécifique mémorisant ainsi l'endroit où se situe le conflit.

Nous allons, dans un premier temps, construire cette fonction de fusion. La fonction de fusion binaire est notée $\otimes$ et prend deux espaces d'interaction en entrée pour fournir un espace d'interaction. Nous désirons pour notre cas d'étude construire $\otimes$ de sorte qu'elle soit idempotente, commutative et associative. Nous démontrerons ensuite qu'elle satisfait bien ces trois propriétés. Nous avons regroupé les différents axiomes définissant la fonction $\otimes$. Nous notons que $\otimes$ se définit à partir de règles logiques qui sont décrites en annexe.

Limites d'ISL : Les problèmes de superposition peuvent se gérer à plusieurs niveaux. Nous distinguons trois niveaux de gestion de superposition. Le premier niveau est semblable aux approches actuelles des aspects dans lequel nous gérons la *concurrence* entre les aspects soit à l'intérieur des aspects même ou bien de manière externe (dans un fichier de configuration par exemple). Le deuxième niveau concerne une amélioration par la gestion de la concurrence des aspects au niveau de la diffusion d'événements. Enfin, le troisième niveau permet de gérer plus finement la superposition des aspects en proposant une fusion interne des greffons.

Les composants mis en œuvre peuvent modifier et contrôler la manière dont les communications vont se passer entre les composants du système. La fusion analyse les composants selon leur position dans l'assemblage de composants constituant le greffon. Le moteur de fusion idéal serait capable de déduire automatiquement la composition finale sans avoir recours à la mise en concurrence des schémas. Il n'est cependant pas possible d'atteindre cet idéal, car les paramètres pour prendre les décisions ne sont pas complètement disponibles.

	$\text{par}_L(\vec{\beta})$		
$\text{par}_L([\vec{a}])$	$\text{par}_L(\vec{\beta})$	si $a = \beta_1$	(1.1)
	$\text{par}_L([\beta_1]) \cup \left(\text{par}_L([\vec{a}]) \oplus \text{par}_L(\vec{\beta} \setminus \beta_1)\right)$	si $a \neq \beta_1$	(1.2)
$\text{par}_K(\vec{\emptyset})$	$\text{par}_L(\vec{\beta})$		(1.3)
$\text{par}_K(\vec{\alpha})$	$\text{par}_K(\vec{\alpha}) \cup \text{par}_L(\vec{\beta})$	si $K \neq L$	(1.4)

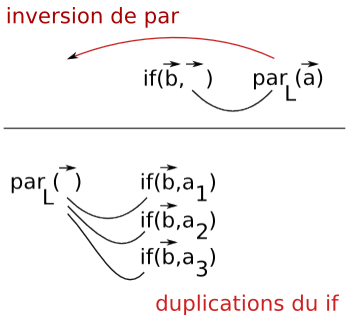
(a) Règles de concurrence

	$if\left(\vec{b}, \vec{\beta}\right)$		
$if\,(\vec{a}, \vec{\alpha})$	$if\left([\vec{b}], [\vec{\beta}] \cup [\vec{\alpha_1}]\right) \oplus if\,(\vec{a} \setminus a_1, \vec{\alpha} \setminus \alpha_1)$	si $b = a_1$	(1.5)
	$if\left([\vec{a_1}], [\vec{\alpha_1}]\right) \cup_{if} \left(if\left([\vec{b}], [\vec{\beta}]\right) \oplus if\,(\vec{a} \setminus a_1, \vec{\alpha} \setminus \alpha_1)\right)$	si $b \neq a_1$	(1.6)
$if\left(\vec{\emptyset}, \vec{\emptyset}\right)$	$if\left(\vec{b}, \vec{\beta}\right)$		(1.7)

(b) Règles conditionnelles

	β		$del(\beta)$		
β	β	(1.8)	commuter		(1.14)
$del(\alpha)$	$del(\alpha)$	(1.10)	erreur !	si $\alpha \neq \beta$	(1.9)
$call$	β	(1.11)	commuter		(1.14)
nop	β	(1.12)	commuter		(1.14)
α	erreur !	(1.13)	commuter		(1.14)

(c) Règles finales



(d) Classement des opérateurs

FIGURE 5.16 – Matrices de fusion ISL4WComp

Exemples Le premier exemple concerne la fusion d'une alarme et d'un système de distribution d'événements. Deux points de coupe spécifient respectivement la variable observed et la variable worker qui sont en conflit (c'est-à-dire, produisent les mêmes points de jonction).

```
advice Alarm (observed,alarm):
    observed.^Out -> ( if( alarm.Check ) call )
    alarm.Check -> ( alarm.Start ; call )
advice Dispatcher (producer,worker,consumer):
    producer.^Out -> ( worker.In )
    worker.^Out -> ( consumer.In )
```

Les règles de spécifications (RSs) ne sont pas en conflit. Ainsi, elles sont recopiées dans le greffon résultant. Cependant, les RSs sont en conflit parce qu'ils redéfinissent le même événement ^Out du composant confondu observed/worker. Par conséquent, leur programme de spécification respectif sont fusionnés de manière logique et le greffon résultant AA0+Ex est calculé en utilisant la matrice de fusion de la Figure 5.16. L'opérateur + correspond à un couple non-ordonné d'opérations à exécuter. Le processus de fusion remplace CALL à la ligne 3 par CALL+consumer.In dans AA0+Ex. L'AA résultant est alors traduit dans un ensemble de MSEs. Par exemple, l'opérateur IF est interprété comme l'ajout d'un composant générique de type IF.

Code en sortie :

```
advice Ex+AA0 (observed,alarm,producer,consumer):
    observed.^Out →( if( alarm.Check ) call )
    alarm.Check →( alarm.Start ; call )
    producer.^Out →( observed.In )
```

Nous avons vu le processus de conception spécifique aux AAs ainsi qu'un cycle d'adaptation d'une application pour l'informatique ambiante. Dans la partie suivante, nous présentons les cycles du processus d'auto-adaptation. La composition dans ce cas se fait à partir des composants spécifiques que nous utilisons pour composer l'assemblage.

5.4.2.3 Fusion BSL

L'extension du tissage vient ici du fait qu'il se révèle nécessaire dans plusieurs cas de gérer les conflits lorsque plusieurs aspects d'assemblage émettent un événement vers un même point d'entrée (c'est à dire la précédence au niveau des sorties). Nous avons mis en place quelques opérateurs qui permettent de gérer de manière locale les conflits en sortie des greffons. Ces opérateurs peuvent être combinés afin d'affiner la fusion.

Un premier exemple représente la connexion d'un composant variable commande à un composant variable lampe. Cette connexion est arbitrée. Cela signifie que l'événement **On** du composant variable commande doit être spécifié également par les autres greffons sinon, il ne sera pas pris en compte dans le schéma de composition final. D'autre part, le second exemple décrit la composition en cascade de deux composants spécifiques : l'arbitrage et le découpage. Comme le premier exemple, le composant variable commande est arbitré. Le second composant variable n'est pas sujet à l'arbitrage. Les deux sources d'événements sont sujets au découpage temporel de période p.

Nous avons deux opérateurs de coordination : l'opérateur d'arbitrage noté A et l'opérateur de découpage temporel noté H. Chacun de ces deux opérateurs est constitué de deux entrées. Il y a une entrée In qui permet de cascader les opérateurs et une entrée event qui est connectée aux sorties des composants variables ou blackbox. Cette opération est régie par un ensemble de règles de fusion que nous avons réunies dans une matrice. Les opérations de fusion des greffons sont régies par un ensemble de règles logiques. Ces règles sont regroupées dans la matrice que nous retrouvons dans la figure 5.17. Les propriétés de cette fusion sont la commutativité, l'associativité et l'idempotence. Ces propriétés sont étudiés dans le chapitre suivant.

Cette fusion modifie la manière dont le tissage s'opère. Nous l'analysons dans la section suivante. Le tissage est donc incrémenté d'un gestionnaire de conflit sur les sorties des composants et prend

	$a(\vec{y})$		$o(\vec{y})$		$conc(\vec{y})$	
$a(\vec{x})$	$a(\text{tri}_1(\vec{x},\vec{y}))$	(1.14)				
$o(\vec{x})$			$o(\text{tri}_2(\vec{x},\vec{y}))$	(1.15)		
$conc(\vec{x})$					$conc(\text{concat}_1(\vec{x},\vec{y}))$	(1.16)

(a) Règles de fusion des opérateurs BSL

Classement	Opérateur	Règles de transformation
1	Arbitrage $a(\vec{a})$	
2	Découpage $b(\vec{d})$	$\forall x \in \vec{a}, x \notin \vec{d}$
3	Concurrence $conc(\vec{c})$	$\forall x \in \vec{a}, x \notin \vec{d}$ et $\forall y \in \vec{d}, y \notin \vec{c}$

(b) Fusion des opérateurs BSL

FIGURE 5.17 – Matrices de fusion BSL

en compte les composants spécifiques tels qu'ils sont décrits dans ce chapitre. Le reste du mécanisme de tissage par unification et par duplication reste identique. Ce mécanisme de gestion est rajouté à la fin du tissage.

5.5 Aspects d'assemblage et informatique ambiante

En informatique ambiante, l'adaptation peut être provoquée par un mécanisme de réaction cyclique orchestré par l'utilisateur ou son environnement.

Utilisateur : L'adaptation centrée utilisateur consiste à (dé-)sélectionner des AAs afin d'intégrer ou retirer des fonctionnalités et ne fait qu'*orienter* plutôt que de *programmer* les applications *émergentes*. Dans un système classique, l'utilisateur agit également sur l'assemblage de composants et faire des modifications directement sur l'assemblage. Cela requiert une grande expertise, car il devra bien connaître le domaine d'utilisation des composants logiciels.

Infrastructure : L'adaptation provoquée par les modifications de l'infrastructure est organisée autour d'une boucle de programme qui a pour but de vérifier si les dispositifs sont toujours présents dans l'environnement ou bien si de nouveaux apparaissent. Lorsqu'un dispositif disparaît de l'environnement du système, le composant logiciel le représentant est retiré de l'assemblage. Le processus d'adaptation consiste à détecter ces modifications structurelles de l'assemblage de base. À chaque cycle du processus, est vérifié si un nouvel AA est intégré ou non.

5.6 Conclusion

Le modèle que nous avons présenté s'appuie sur trois principaux concepts du modèle des aspects d'assemblage. Il s'inspire de l'AOP notamment les points de coupe que nous avons fait évoluer et les greffons. Nous avons créé un nouveau mécanisme de tissage, scindé en deux opérations de composition et de résolution de conflit, renfermant le mécanisme de superposition des adaptations. Ces concepts s'enchaînent au moment de l'exécution de l'application pour produire des adaptations.

Cette approche est particulièrement bien adaptée pour un ensemble de services basés sur les événements en réaction à une variation du contexte ou même des préférences de l'utilisateur. Nous appelons cette approche la *composition pour l'adaptation*.

Chapitre 6

Plate-forme WComp et Expérimentations

Sommaire

Dans ce chapitre, nous présentons l'implémentation des concepts que nous avons présentés précédemment : les services composites et le modèle d'adaptativité pour l'informatique ambiante. Pour cela, nous nous appuyons sur la plate-forme WComp que nous avons étendue.

Cette présentation s'articule autour de trois implémentations clés : l'implémentation du modèle LCA, l'implémentation modèle AA et la mise en œuvre dans un espace ambiant.

6.1 Présentation de la plate-forme WComp

WComp est la mise en œuvre d'expérimentations de modèles présentés dans cette thèse et des travaux de RAINBOW. L'architecture de WComp [1] s'organise autour de containers et de designers. L'objectif des containers est de prendre en charge dynamiquement la gestion de la structure tels que l'instanciation, la désignation, la destruction de composants logiciels et de liaisons. Les designers permettent la manipulation dynamique des assemblages de composants en utilisant les formalismes adaptés.
Voici les différentes projections de WComp :

La plate-forme WComp Java est la première implémentation du modèle de composant léger pour le prototypage rapide d'applications embarquées.

La plate-forme SharpWComp reprend les principes de WComp Java en les intégrant dans un réel environnement de développement SharpDevelop en s'appuyant sur une machine virtuelle .NET.

La plate-forme WComp C++ est une implémentation du modèle de composant léger indépendante de toute machine virtuelle ; elle est ainsi faite pour s'exécuter en environnement très restreint.

6.2 WComp et LCA

Dans cette section, nous présenterons les implémentations de base de WComp (container, designers). Nous discutons ensuite de ces implémentations.

6.2.1 Container WComp/LCA

La plate-forme WComp utilisant le framework .NET est dotée d'un container implémenté en langage C#. Les composants logiciels sont implémentés par un objet C#. D'une part, nous verrons dans la section 6.3.2 le modèle d'implémentation des composants dans ce langage. D'autre part, nous verrons d'autres containers implémentés en C et en Java, dans la section 6.6, destinés aux cibles légères de l'espace ambiant.

6.2.2 Designers de base WComp

Un environnement de développement doit offrir des représentations variées et des outils pour les programmeurs. La plate-forme WComp propose une approche multi-design qui permet de modifier les applications selon différentes approches. Nous présentons les principales approches ci-dessous :

Le designer de Code : Le premier designer est celui du code-source qui représente l'application décrite sous la forme d'un assemblage de composants. Lorsque le programmeur modifie le code-source, le designer communique les modifications au container. Ces modifications consistent au remplacement de composants lorsque leur code a été modifié. Le designer permet également la modification des connexions entre les composants. Ce designer ne fonctionne que dans un environnement .NET et pour le langage C#.

Le designer Graphique : Une application graphique a un rendu graphique à l'écran. Ce rendu est manipulé par le programmeur à travers le designer graphique. Tous les composants logiciels ne sont pas dotés d'une représentation graphique comme un bouton, une image, un label, une barre de progression. Ainsi, uniquement ceux qui sont annotés comme étant graphiques sont représentés

1. Nos travaux s'appliquent à partir de la révision 176 de SharpWComp.NET.

dans ce mode. On peut alors modifier leurs propriétés graphiques comme leur position, leur couleur, leur forme, etc.

Le designer Console : Le designer console représente un cas particulier. Il permet d'envoyer des commandes de reconfiguration structurelle au container grâce à un langage ADL simple. Sont implémentées quatre commandes d'intercession représentant l'ajout, la suppression, l'accès et la modification des propriétés, la connexion et la déconnexion de composants et cinq commandes d'introspection pour obtenir les types de composants disponibles, les composants présents dans l'assemblage, la liste des connecteurs, l'interface des composants (point d'entrée et événement).

Le designer Visuel : L'assemblage de composants constituant l'application peut être manipulée graphiquement par le designer visuel. Le programmeur manipule intuitivement les concepts d'assemblage de composants logiciels : un composant logiciel est représenté par un rectangle et les connecteurs par des traits entre les interfaces de composants (disposées à gauche et à droite des rectangles). Il peut ainsi grâce à un menu contextuel, ajouter, enlever des composants ou des connecteurs et modifier leurs paramètres.

La plate-forme WComp.NET est une plate-forme logicielle constituée d'un addon et de SharpDevelop lui-même. SharpDevelop est un environnement de développement intégré (IDE) basé sur la plate-forme .NET qui offre un environnement de qualité et libre (licence GPL) comparable à VisualStudio.

L'addon nous permet de bénéficier de l'ensemble des fonctionnalités de SharpDevelop (génération de code correspondant à l'assemblage WComp, manipulation de la représentation graphique de l'application, génération du code exécutable correspondant à l'application, etc.) Afin de mieux comprendre WComp.NET, la première section décrit l'architecture de cet environnement.

6.2.3 Les implémentations WComp/LCA (Java, C#/SharpDevelop)

Un composant peut gérer lui-même son cycle de vie en s'appuyant sur l'état de ses connexions locales. Nous pouvons également gérer son cycle de vie de manière externe comme c'est fait par exemple dans la plupart des modèles de composant. Le cycle de fabrication d'une application pour les autres modèles de composant s'appliquent également à notre cas.

```
[Bean] public class IntegerAdd : EventedDrawable {
        private int add, val;
        public event ValueEventHandler EmitValue;
        public int Val { get { return val; } set { val = value; FireValue(val+add); } }
        public int IntAdd { get { return add; } set { add = value; FireIntValue(val+add); } }
        private void FireIntValue(int v) { if (EmitIntValue ≠null) EmitIntValue(v); }
}
```

FIGURE 6.1 – Exemple d'implémentation d'un composant addition

Un assemblage de composants est contenu dans une entité logicielle notée Container. Cette entité est représentée par un objet de type Container.

L'implémentation d'un container contient trois grande parties : la déclaration des types de composant, la création de leurs instances et leur initialisation et enfin la déclaration des connecteurs simples et complexes.

Connecteur Simple et Complexe Un connecteur simple est implémenté en utilisant les mécanismes de délégation et d'événement de la plate-forme .NET.

Un connecteur complexe dans la plate-forme WComp.NET est une modification de flot d'événements. Le flot d'événements est redirigé vers d'autres points d'entrée du composant source afin de récupérer d'autres données à rajouter aux paramètres.

Un connecteur complexe est implémenté par la classe EventHandler. Cette classe prend en paramètre les méthodes permettant de récupérer les paramètres manquants de la méthode appelée.

```
public class Container1 : System.Windows.Forms.Form {

  // Declaration des composants
  [BeanDesignLocation(200,184)] private System.Windows.Forms.Label label1;
  [BeanDesignLocation(48,168)] private System.Windows.Forms.Button button1;

  public Container1() {
        InitializeComponent();
        InitializeBeans();
  }

  [STAThread]
  public static void Main(string[] args) {
    Application.Run(new Container1());
  }

  private void InitializeComponent() {
    this.label1 = new System.Windows.Forms.Label(); // Création des composants
    this.button1 = new System.Windows.Forms.Button();

    this.label1.Size = new System.Drawing.Size(100, 32); // Initialisation des paramètres
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    (...)

    this.Controls.Add(this.label1); // Ajout des composants ayant une interface graphique aux Controls
    this.Controls.Add(this.button1);
  }

  // Initialisation des connecteurs simples et complexes
  private void InitializeBeans() {
    this.button1.Click += new System.EventHandler(this.__button1_to_label1_0);
  }
  private void __button1_to_label1_0(object sender, System.EventArgs e) {
    this.label1.Text = this.button1.Name;
  }
}
```

FIGURE 6.2 – Description de l'implémentation d'un Container en C#

```
C1.Event(param) => C2.Method(param)
C1.Event(param) => C2.Method(C1.GetProp())
```

FIGURE 6.3 – Connecteur Event Simple et Complexe

6.3 WComp et SLCA

Dans cette section, nous verrons comment relier les concepts de container et de designer de WComp au concept de service. Nous présenterons d'abord notre implémentation des services à partir d'UPnP. Nous verrons ensuite l'implémentation des services composites – interfaces structurelles et fonctionnelles et enfin, l'implémentation du service Designer.

6.3.1 Infrastructure de Services pour Dispositif de type UPnP

C'est pour ne pas limiter notre approche à des problèmes d'interopérabilité (section 4.1.2.2) entre différentes technologies que nous proposons une extension aux Web Services comprenant la notion de Web Services pour Dispositifs dont l'objectif est d'abstraire la communication entre dispositifs sous un standard commun. Nous nous sommes basés sur le standard UPnP.

Architecture UPnP : L'UPnP remplit les conditions pour correspondre à notre notion de Web Service pour Dispositif. Cependant, l'utilisation de fichiers de descriptions, la particularité syn-

taxique des événements et l'absence de tout mécanisme de sécurité tel que l'authentification et le cryptage sont des lacunes qui ont motivés un standard plus proches des Web Services DPWS (la technologie qui remplacera notre choix actuel).

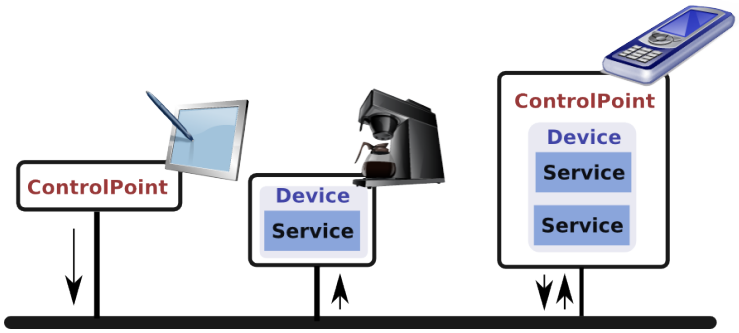


FIGURE 6.4 – Architecture UPnP

UPnP est une initiative industrielle conçue pour permettre la connectivité simple et robuste entre dispositifs de différents fournisseurs sur un réseau IP/LAN. L'architecture UPnP permet par sa nature la découverte et le contrôle de dispositifs sur un réseau, indépendamment des systèmes d'exploitation (utilisation des sockets), des langages de programmation (stubs disponibles pour plusieurs langages) et des réseaux (grâce à l'utilisation des normes existantes : IP, UDP, TCP, HTTP, XML et SOAP).

UPnP s'appuie sur deux principaux concepts : ControlPoint et Device. Un ControlPoint est une entité logicielle client qui permet de contrôler un ServiceUPnP. Un Device, quant à lui, est une entité logicielle serveur qui reçoit donc des commandes d'un ControlPoint. Un dispositif peut contenir ces deux entités en même temps. Les services sont contenus dans les DeviceUPnP.

Terminologie UPnP : Au niveau fonctionnement, UPnP se décompose en 4 niveaux logiciels :

- *Niveau 4 :* Contrôle, Événements et Présentation
- *Niveau 3 :* Description
- *Niveau 2 :* Découverte
- *Niveau 1 :* Adressage

Au niveau le plus bas, le niveau 1, un ControlPoint et un Device obtiennent une adresse IP. Au niveau 2, un ControlPoint découvre les Devices disponibles. Au niveau 3, un ControlPoint découvre les propriétés et les capacités d'un Device. Enfin, au niveau le plus haut, le niveau 4, un ControlPoint peut exécuter une *action*, c'est-à-dire un point d'entrée, d'un dispositif. Il peut également s'abonner à un événement et être à l'écoute de changement d'état des dispositifs. Il peut aussi commander des dispositifs.

Produits intégrant l'UPnP : Plusieurs produits commerciaux logiciels et matériels intègrent l'UPnP. Cependant, cela ne représente qu'une petite partie des objets de l'espace ambiant. Notre approche réside alors dans la conception de bridges UPnP pour adresser ces objets.

Voici quelques exemples de bridges que nous avons utilisé :

- Cyrlink - UPnP (Ubiquarium, section 6.5.1.2)
- XBee - UPnP (Bâtiment Haute Technologie, section 6.6)
- Phidget - UPnP (Ubiquarium)
- Nabaztag, Téléphones portables, etc.

6.3.2 Container WComp/SLCA

Nous allons voir l'implémentation du modèle de composant (structure d'un composant léger, la structure d'un assemblage, l'implémentation des connecteurs simples et complexes) et ensuite l'implémentation du modèle SLCA avec les sondes et les puits et l'interface structurelle.

6.3.2.1 Interface structurelle

L'interface structurelle de WComp.NET est très riche. Elle dispose de 10 points d'entrées et de 5 variables d'état génératrices d'événements. Nous les avons regroupés dans 5 grands principes : l'intercession avec l'ajout, le retrait de composants, la connexion et la déconnexion, enfin les mécanismes d'introspection. La liste des commandes et events de l'interface structurelle est donnée dans le tableau 6.1.

Nom	Description
CheckBeanProperties	Donne la liste des noms et des types de propriétés pour un type de composant (prend actuellement un nom d'instance à la place)
CheckBeanPropertyValue	Donne la valeur et le type d'une propriété d'une instance, sérialisée au format XML
CheckBeans	Donne la liste des instances de l'assemblage
CheckBeanType	Donne le type de composant d'une instance
CheckBeanTypes	Donne la liste des types de composants chargés dans le container
CheckEvents	Donne la liste des événements de l'interface d'un type de composant (prend actuellement un nom d'instance à la place)
CheckLinkedBeansFrom	Donne la liste des noms des instances qui sont à la source d'une liaison en commun avec l'argument
CheckLinkedBeansTo	Donne la liste des noms des instances qui sont à la destination d'une liaison en commun avec l'argument
CheckMethods	Donne la liste des méthodes de l'interface d'un type de composant (prend actuellement un nom d'instance à la place)
CreateBean	Crée une instance de composant
CreateLink	Crée une liaison entre deux composants
CreateNamedBean	Crée une instance de composant, et on peut en spécifier le nom
LoadType	Charge un type de composants dans le container
RemoveBean	Supprime une instance de composants de l'assemblage
RemoveLink	Supprime une liaison de l'assemblage
SetBeanPropertyValue	Modifie la valeur d'une propriété d'une instance donnée, sérialisée au format XML
UnloadType	Décharge un type de composant dans le container

Tableau 6.1 – Liste des commandes et events

Remarque : Un service Designer est un service qui contrôle un container doté d'une interface structurelle.

6.3.2.2 Interface fonctionnelle

Deux composants spécifiques appelés sondes et puits EmitProbe et SourceProbe permettent de spécifier dynamiquement l'interface du service structurel. Chaque modification des sondes ou des puits dans l'assemblage modifie immédiatement l'interface du service structurel.

6.4 WComp et AA

Nous présentons deux designers pour l'adaptation et la continuité des applications en informatique ambiante : Un designer pour construire des applications et des services composites.

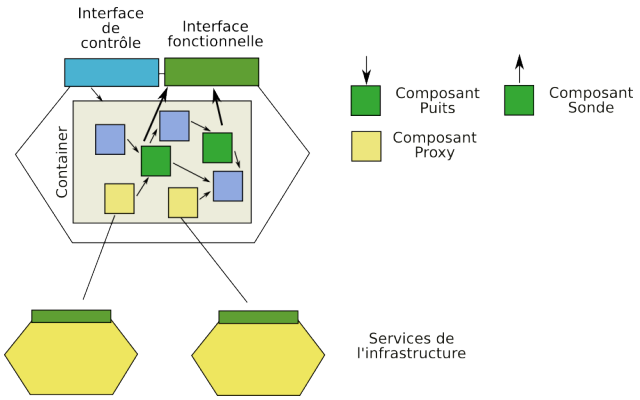


FIGURE 6.5 – Encapsulation d'un container dans un service UPnP, dit Service Composite

Le premier designer découvre l'ensemble des services pour dispositifs accessibles et instancie les composants Proxy correspondants (Designer UPnP).

Le second designer construit des applications et des services composites avec une connaissance a priori partielle de l'espace ambiant (Designer AA).

6.4.1 Les designers AA

Nous avons implémenté les deux types de designers AA : ISL et BSL. Ces designers sont des services composites destinés à être connectés à une interface structurelle de service pour l'adapter en continue.

Nous avons développé une interface graphique permettant d'éditer dynamiquement les aspects d'assemblage. C'est une même interface qui sera utilisée pour éditer BSL ou ISL4WComp. Il se compose, comme illustrée dans la figure 6.6, de quatre panneaux.

Le premier panneau Advice est un éditeur permettant d'éditer la spécification d'un greffon (section 6.4.1.2). Le deuxième panneau Pointcut permet d'y inscrire les points de coupe (section 6.4.1.1). Enfin, les deux derniers panneaux permettent de spécifier le nom de l'AA que l'on édite actuellement et de sélectionner puis visualiser les AAs effectivement appliquées à l'assemblage.

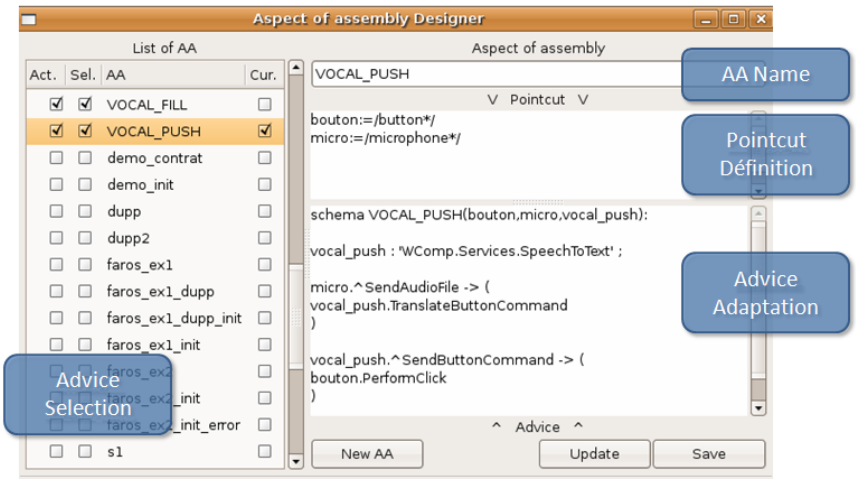


FIGURE 6.6 – Désigner ISL4WComp

Le designer ISL4WComp a pour but de sélectionner les AAs et de les tisser afin de modifier

l'assemblage du Service Composite cible.

Le designer BSL a pour but de sélectionner les AAs et de les tisser afin de modifier l'assemblage du Service Composite cible.

6.4.1.1 Expression et méthodes de calcul des points de coupe

Le calcul du point de coupe est effectué en deux parties : un mécanisme de filtrage des composants fournis en entrée suivie d'une sélection syntaxique effectuée par matching d'expressions régulières en s'appuyant sur le langage AWK.

Nous utilisons à des fins d'expérimentation le langage AWK afin d'exprimer les points de coupe. En effet, AWK peut manipuler des fichiers, des chaînes de caractères, faire des opérations mathématiques, ou des séquences logiques complètes. AWK simplifie l'extraction des champs contenus dans des chaînes de caractères. Nous l'utilisons en l'appliquant à un flux de données provenant de son entrée. AWK décompose chaque flux en champs, les champs étant des chaînes de caractères séparés les unes des autres par un caractère particulier appelé le séparateur. Dans notre cas d'utilisation le flux d'entrée correspond à la liste des composants blackbox présents dans l'assemblage de base.

Implémentation des filtres : Nous avons vu le modèle dans la section 5.3.1.1 de la grammaire des filtres des points de coupe que nous employons.

```
1 <variable>←(nom du filtre) parametres ; [ | autre filtre ... ]
```

FIGURE 6.7 – Grammaire

Le composant d'implémentation des filtres est nommé **Pointcut**. Il comporte un point d'entrée prenant en paramètres une liste de composants et un point de coupe conforme à la grammaire ci-dessus. Il comporte un événement qui délègue l'interprétation des paramètres du filtre (Effectuée par la librairie AWK, section suivante).

```
for (int k=0 ; k<pointcut.size() ; k++) { // Decomposition of a pointcut
        split( pointcut[k], "←", data ); // Separate variable (index 0) from program (index 1)
        results += interprete( data[1] , components ); // Run the program
}
```

FIGURE 6.8 – Implémentation simplifiée des filtres en C++

Librairie GAWK : Nous avons utilisé le code source du projet Busybox [2] dont l'implémentation de l'interpréteur du langage AWK bien que réduite au minimum est complète.

Nous avons extrait le code de l'interpréteur afin de l'encapsuler dans un composant logiciel. Ce composant logiciel a pour point d'entrée deux chaînes de caractères qui sont les paramètres du point de coupe exprimé en AWK et la liste des composants.

Les lignes d'un script AWK sont de la forme :

```
motif action
```

Awk comprend deux motifs, c'est-à-dire deux sélections particulières BEGIN et END, interprétées respectivement en Debut et Fin de script quelque soit leur place. BEGIN est interprété au démarrage du programme et END avant la terminaison du script. AWK intègre un langage interprété proche du C.

Nous définissons les descriptions de points de coupe comme des ensembles de filtres sur les caractéristiques d'un assemblage de composants légers. Ces filtres produisent une liste de paramètres

2. http ://www.busybox.net

qui satisfont la liste des variables d'un greffon afin d'être intégré dans l'assemblage de base. Si uniquement une liste est construite, le greffon est intégré seulement une fois dans l'assemblage de base et les variables symboliques sont syntaxiquement remplacées dans le greffon afin de correspondre aux points de jonction de l'assemblage de base. Si plusieurs listes sont produites, le greffon est dupliqué et chaque ensemble de variables, avec une occurrence de chaque point de jonction, est respectivement remplacé. Le mécanisme du point de coupe intervient à ce niveau de correspondance et permet de relier une variable à plusieurs composants Proxy.

Syntaxe de base : Ainsi par exemple, nous pouvons spécifier le point de coupe suivant :

```
isalon←/i{*}/
```

Cet exemple est constituée de deux parties. La partie de gauche correspond à l'identifiant d'un composant variable. La partie de droite correspond à un pattern matching de composant blackbox décrit dans le langage AWK. Ici, i* est une expression régulière (mise entre les barres obliques) qui permet de désigner les composants Proxy dont l'identifiant commence par un i.

Ainsi, le résultat de l'évaluation de cette expression par le moteur de points de coupe est une liste de composants Proxy auxquels la variable correspond.

L'application du greffon ne sera donc pas juste une juxtaposition du schéma sur un assemblage de base mais le résultat d'abord d'une duplication du greffon en fonction du nombre de composants blackbox qui ont été sélectionnés et puis la juxtaposition des schémas dupliqués sur l'assemblage de base.

Le but d'un point de coupe est de construire pour chaque composant variable i une liste de composants blackbox correspondant L_i contenant n_i éléments. Toutes les listes doivent être de la même taille que nous notons n. Ainsi, le schéma auquel est sujet un point de coupe est dupliqué n fois. Notons l_i^k le k-ième élément de la liste L_i. Chaque schéma dupliqué aura comme liste de composant variables assigné les valeurs de $l_i^1 \dots l_i^n$. Nous donnons un exemple dans la section suivante.

Instanciations multiples : Soit S un schéma ayant les composants variables suivants : ivariables et lvariables.

Voici un exemple de liste de composants

```
1 isalon
2 icuisine
3 lchambre
4 lsalon
```

Voici un exemple de point de coupe :

```
1 ivariables←/i{*}/
2 lvariables←/i{*}/ { print 'l'␣substr($1,2)␣}
```

Le schéma S est dupliqué deux fois. Notons S_1 et S_2 les duplications. Le calcul du point de coupe appliqué à la liste des composants nous donne :

ivariables, isalon, icuisine

lvariables, lsalon, lcuisine

Le schéma S_1 va pouvoir s'appliquer car isalon substituant ivariable et lsalon substituant lvariable existent. S_2 ne s'appliquera pas car le composant lcuisine n'existe pas.

Exemple utilisant un programme AWK : La variable *observed* est confrontée au nom du composant qui commence par *user*, et *alarm* à celui qui commence par *err*. Le second filtre (ligne 3-4) est un programme AWK qui, en plus de les faire correspondre aux débuts des noms des composants, les trie par ordre alphanumérique. La ligne 3 enregistre les noms dans un tableau en fonction de leur suffixe. Et à la fin de la mise en correspondance, le programme affiche les noms stockés et triés.

```
observed ←/user∗/ ;
alarm ←
  /err∗/ { a[substr($1,3)]=$1 }
  END { for(i=1;i≤NR;i++){print a[i]} } ;
```

FIGURE 6.9 – Exemple de point de coupe

L'ordre des composants n'est pas spécifié et peut être aléatoire lorsqu'un programme spécifique de tri n'est pas implémenté dans AWK. Dans cet exemple, le premier point de coupe n'a pas d'ordre de spécifié. Le second est ordonné. Considérons un assemblage de base constitué de cinq composants : err1 à err3 et user1 et user2. Le greffon est dupliqué en deux greffons que l'on peut appliquer notés Ex1 et Ex2. Le résultat global est un tableau à deux dimensions dont les paramètres des greffons sont représentés en colonne.

6.4.1.2 Expression des greffons

L'implémentation des greffons comporte plusieurs parties. Nous avons d'abord un compilateur de greffon qui permet de transformer un greffon écrit dans un langage dédié ISL4WComp ou BSL en règles logiques. Enfin, nous voyons l'implémentation de la composition.

Nous avons un unique compilateur dédié qui traduit de façon générique les spécifications du greffon en règles logiques. Le langage s'écrit sous la forme de règle de transformation[30]. Voici la structure générale de la grammaire :

```
<Greffon>: advice ':' <Regles>
<Regles>: <Regles> | <Regle>
<Regle>: VARIABLE '(' <Parametres> ')' '−>' <Assemblage> ';' | ADD '(' NOM ',' TYPE ')'
<Assemblage>: NOP | <operateur> '(' Parametres ')'
<Parametres>: <Parametres> | VARIABLE | <Assemblage> | Empty ';'
```

FIGURE 6.10 – Grammaire générale d'un greffon

Les variables et les opérateurs de cette grammaire sont des chaînes de caractères. Le mot-clé 'NOP' représente un assemblage vide. Avec cette grammaire, un greffon est un ensemble de règles de transformation. Chaque règle désigne une coupe d'un assemblage associé à un sous-assemblage (Assemblage). Un assemblage est constitué d'opérateurs, c'est-à-dire de composants logiciels spécifiques à un AA donné. Ces opérateurs ne sont pas spécifiés dans le compilateur, mais analysés a posteriori lors de la construction de l'arbre sémantique. Dans les sections suivantes, nous analyserons ces opérateurs.

Greffon ISL4WComp : Les greffons ISL4WComp sont constitués des opérateurs de condition (if), de séquence (seq), de concurrence (par), de la délégation (del) et du call. Ces opérateurs permettent de spécifier un assemblage de composants à travers un langage dédié.

Cependant, pour faciliter la composition d'assemblage, nous n'allons pas travailler directement sur un assemblage mais sur une représentation par des prédicats.

Transformation Greffon → Règles logiques : Chaque opérateur représente un prédicat, leurs paramètres sont également des prédicats. Cette transformation est simplifiée par le compilateur et le langage de greffon que nous avons vu précédemment, car il correspond à l'arbre syntaxique de la spécification.

```
if(luminosity(bad),light(on),light(off))
```

FIGURE 6.11 – Exemple de représentation logique

L'ensemble de ces règles logiques constitue les *faits* du système. Ces faits sont confrontés à une base de règles correspondant aux règles de composition (section 6.4.2.1) et aux matrices de fusion que nous avons vu dans le chapitre précédent. On peut alors en déduire un ensemble unique de règles de transformations qui constituera l'assemblage final.

Composants légers spécifiques ISL4WComp : Chacun des opérateurs énoncés correspond à un composant logiciel dans un des langages cibles. Nous avons ci-dessous donné l'exemple du composant de séquence en C#. Les autres composants sont décrits dans le dépôt logiciel SharpWComp [202].

```
[Bean(Category="AA Beans")]
public class SEQ : EventedDrawable {
        public object Inp(params object[] o) {
                if (Out1 ≠null) Out1(o);
                if (Out2 ≠null) return Out2(o);
                return new object();
        }
        public event Generic Out1, Out2;
}
```

FIGURE 6.12 – Implémentation d'un composant de séquence en C#

Ce composant a des ports génériques (section 6.4.2) et permet de transmettre le flot de contrôle à l'événement ^Out1 puis à ^Out2. Il correspond au modèle de séquence à la page 89.

Greffon BSL : Le greffon BSL est constitué des opérateurs d'arbitrage (arb) et d'ordonnancement (ord). Ces opérateurs fonctionnent en s'appuyant sur une notion de temps relative aux événements. Par exemple, l'ordonnancement consiste à trier des événements arrivant dans une période T. Si deux événements a et b arrivent pendant cette période, ils sont retransmis quand T est écoulé dans un ordre prédéfini (par exemple, b puis a).

Règles logiques et greffon BSL : De la même manière que l'assemblage en ISL4WComp, nous pouvons exprimer les spécifications BSL en logique du premier ordre. La spécification d'un greffon BSL est traduite en règles logiques comme illustré ci-dessous :

```
arb(a,b,c,d,ord(f,g,h))
```

L'arbitrage et l'ordonnancement consistent principalement à ordonner et à sélectionner les flots d'événements qui vont exécuter un point d'entrée.

Composants légers spécifiques BSL : Un composant d'arbitrage a une implémentation en C# qui en fonction de propriétés internes fixées à sa construction, va pouvoir ordonnancer à nouveau le flot d'événements.

Ainsi, pendant une fenêtre de temps T, si un événement *priority* n'a pas été reçu, tout événement peut alors passer, sinon, seul l'événement désigné peut transiter.

6.4.2 Composants génériques pour les opérateurs d'AAs

Pour éviter la création dynamique de composants et réutiliser au maximum les composants déjà présents dans le dépot, nous proposons d'implémenter des ports génériques dans le modèle de composant léger. Cela permet de réutiliser des composants spécifiques comme la condition, la séquence, etc. pour des signatures de méthodes différentes.

```
[Bean(Category="AA_Beans")]
public class ARB : EventedDrawable {
   string priority = "ErrorEvent"
        Timer t;
        public object Inp(params object[] o) {
                if (o[0] == priority && t.ok() && Out1 ≠null)
                        Out(o);
                else
                        if (Out ≠null) return Out(o);
        }
        public event Generic Out;
}
```

FIGURE 6.13 – Implémentation d'un composant d'arbitrage

```
params object[] o
```

Port générique : Un port générique en C# peut s'écrire en utilisant un paramètre de type :

Lors de la connexion d'un point d'entrée et d'un événement, nous devons alors ajouter un cas particulier pour transformer les paramètres. Rappelons qu'un connecteur complexe consiste à la création dynamique d'une méthode anonyme C# pour récupérer les paramètres à passer au point d'entrée. Nous créons un nouveau cas qui consiste également en la création d'une telle méthode anonyme, mais pour faire correspondre un événement ayant des paramètres typés à la structure de données non typé *params object*[] *o*. Cette correspondance se fait dans les deux sens.

Sens Generique → Typé : On construit par réflexion un appel de méthode avec, comme paramètre chaque objet transmis dans la table *o*.

```
        public class Generic_to_Signature {
                        MethodInfo mi = method.GetType().GetMethod(method);
                        int nbParamDest = mi.GetParameters().Length;
                        object[] p = new object[nbParamDest];
                        int n = 0; // Parameter Index
                        foreach(object j in o) {
                                p[n] = j;
                                n++;
                        }
                        return mi.Invoke(method,p);
        }
```

FIGURE 6.14 – Passage de paramètres sans tenir compte du typage en C# (Framework .NET version 1.1)

Sens Typé → Générique : On effectue l'opération inverse. On construit une table non typé dans laquelle on ajoute chaque objet typé en paramètre de l'événement.

La création d'un connecteur générique consiste à modifier le projet ApplicationArchitecture (container) de WComp.NET. En particulier, la classe LinkInfos contient les méthodes de connexion complexe.

Cas ISL4WComp : En ISL4WComp, nous avons créé 4 catégories de composants dotés de ces ports génériques :

- If
- Delegate
- Seq
- Par

Cas BSL : En BSL, trois composants ont été créés tout en conservant la sémantique des spécifications selon les priorités (1) arbitrage, (2) découpage et (3) concurrence :

- Arbitrage
- Découpage
- Concurrence

Ces composants génériques doivent supporter n'importe quel type d'événements en entrée.

6.4.2.1 Composition / Fusion des greffons

Composition La composition est un algorithme que l'on peut simplement exprimer en logique. Elle consiste à vérifier qu'il n'y a pas de conflit entre les règles de transformation. S'il n'y en a pas, alors les règles sont juxtaposés. S'il y a un conflit, alors les règles en conflits sont transmises au composant de fusion (dans leur langage respectif) et le résultat est juxtaposé.

```
islw_fusion0([A|TA],R) :-
    retractall(error1(_)),assert(error1(0)), % error1 = 0
    islw_fusionL(A,TA,R),
    (error1(0);throw('cannot_resolve_ambiguities')).
```

FIGURE 6.15 – Implémentation de la composition en Prolog

La composition suit l'algorithme d'unification, mais s'appuyant sur les règles des greffons (section 5.3.2.1). Le prédicat 'compose' que nous avons implémenté prend trois paramètres : les règles non-composées (entrée), les règles juxtaposées (sortie). Il prend les règles non composées et produit les règles juxtaposées dont les conflits ont été résolues.

La composition est la même pour ISL4WComp et BSL, c'est la fusion qui change.

Fusion L'implémentation de la fusion est une base de règles en Prolog que l'on unifie avec les faits que constituent les greffons. Les règles de fusion sont regroupées en annexe.

```
% for each a in E|TE append an if(a) in the result
cif(L,C,[par(L,E)|TE],T) :-
    cif(L,C, TE, R), % search and resolve conflicts in TE
    cif(L,C, E, R1), % search and resolve conflicts in E
    append(R1,R,T).
cif(L,C,[E|TE],T) :-
    cif(L,C, TE, R),
    append([if([C],[[E]])],R,T).
cif(_,_,[],[]).
```

FIGURE 6.16 – Implémentation de la fusion de la délégation en Prolog (ISL4WComp)

Un prédicat réalise la fusion de deux arborescences représentant chacune une règle. Il prend comme paramètres le premier arbre (entrée), le second arbre (entrée) et l'arbre résultat (sortie).

6.4.3 Cycle d'adaptativité dans WComp

Le cycle d'adaptativité consiste à réagir à chaque changement de l'infrastructure de dispositifs. Un calcul de la composition des AAs concernés par ces changements est alors effectué.

6.5 Mise en œuvre en espace ambiant de Services pour Dispositifs

Notre second objectif est de construire plusieurs prototypes de base implémentant notre modèle de composant. Chaque prototype est adapté à un niveau de granularité donné de l'infrastructure

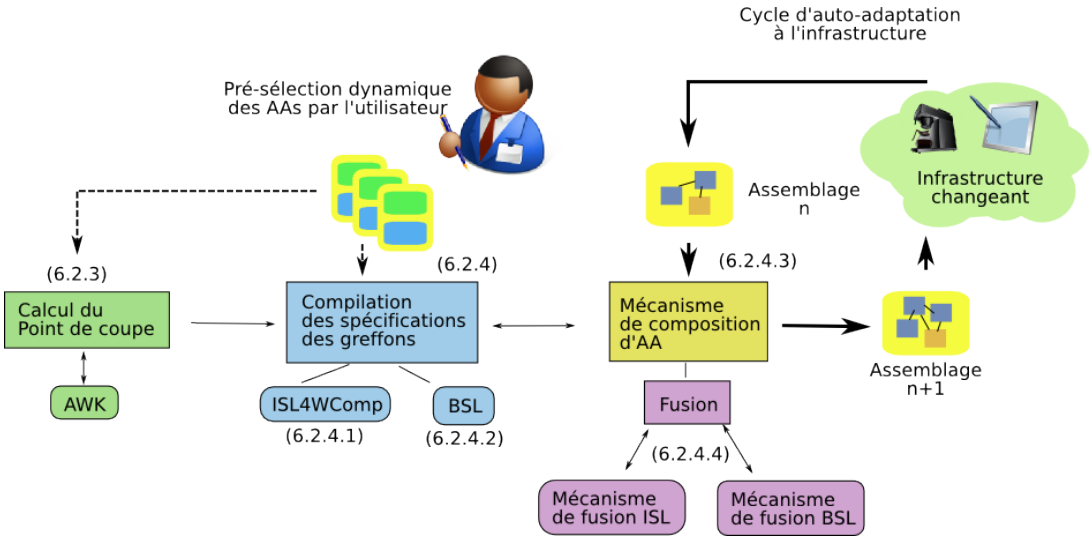


FIGURE 6.17 – Architecture de l'implémentation du calcul des AAs

matérielle : microcontrôleur, ordinateur de type personnel et intergiciel.

6.5.1 Contexte expérimental

6.5.1.1 Architecture matérielle cible

Ces dispositifs peuvent être virtuels ou réels dans l'environnement personnel d'un utilisateur ou bien enfouis dans l'environnement physique. L'infrastructure offre ainsi un certain nombre de services variés tels que :

- des services UPnP virtuelles d'une scène 3D dans laquelle l'utilisateur est immergé
- des services UPnP réels utilisant des technologies propriétaires dont nous avons réalisé une passerelle UPnP (ex : Newstéo, Mobile et les commandes Hayes on AT)
- Web Services standard comme les services météo sur Internet

Cet environnement permet l'évaluation des nouvelles applications de l'informatique mobile et ambiante telles que les usages des wearable computers. C'est un cadre d'expérimentation pour le prototypage de ce type d'application. L'infrastructure logicielle de l'Ubiquarium repose sur le plate-forme WComp qui gère les applications sous la forme d'assemblages de composants logiciels ainsi que la dépendance de certains composants aux ressources.

6.5.1.2 Infrastructure logicielle

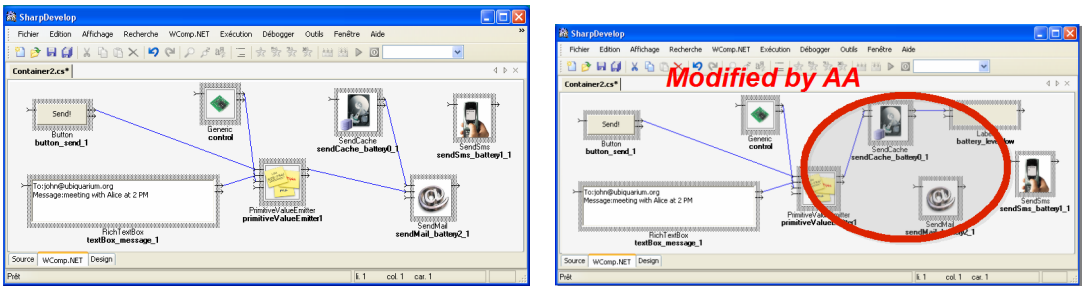
L'architecture de nos exemples dans Ubiquarium informatique s'appuie sur un designer UPnP. C'est un service qui a deux fonctionnalités : créer les composants Proxy des dispositifs UPnP découverts et détruire les composants Proxy des dispositifs UPnP disparus. Ce service nous permet d'effectuer les adaptations dirigées par l'infrastructure des dispositifs.

Nous voyons deux exemples : l'économie d'énergie et l'adaptation d'une application pour l'utilisation de la synthèse vocale.

6.5.2 Scénarios et mise en œuvre de WComp

Économie d'énergie Les aspects d'assemblage fournissant les fonctionnalités de base sont utilisés pour construire l'application en fonction des composants légers disponibles ainsi que les services. Chaque aspect d'assemblage spécifie une méthode pour tisser afin d'ajouter ces fonctionnalités. Ces

fonctionnalités peuvent être de l'ordre des communications GSM ou WIFI ou des fonctionnalités de cache, par exemple.



(a) Avant Adaptation

(b) Après Adaptation

FIGURE 6.18 – Adaptation basée sur l'activation du Wifi

```
pointcut pcEngine:
  mail ←/sendMail∗/
  cache ←/sendCache∗/
  value ←/primitiveValueEmitter∗/
  battery_low ←/batteryLow∗/
advice ConsoMinimale(mail,cache,battery_low):
        value.^Emit −> ( cache.Send() )
   cache.^Sending −> ( battery_low.ShutDown() )
```

FIGURE 6.19 – AA Consommation minimale

Au niveau des politiques d'économie d'énergie, nous avons défini trois aspects d'assemblage qui s'appuient sur une politique de consommation énergétique. Nous retrouvons donc :

- Consommation minimale : les dispositifs WIFI et GSM seront déconnectés de l'application et toute communication sera routée au système de cache afin de stocker les messages.
- Consommation standard : les messages sont envoyés par SMS sur le réseau GSM.
- Pas de limitation de la consommation : le WIFI est utilisé et tous les messages sont directement routés vers ce service.

L'application considérée est un gestionnaire d'email. Elle est constituée d'un cache, et de plusieurs composants d'édition (bouton, champ de texte).

En effet, l'utilisateur ne peut pas prendre la décision d'utiliser une communication WIFI dans l'infrastructure si le WIFI n'est pas présent dans l'environnement. Mais tous les autres AAs peuvent être activés à la fois par l'utilisateur et par le contexte de l'application.

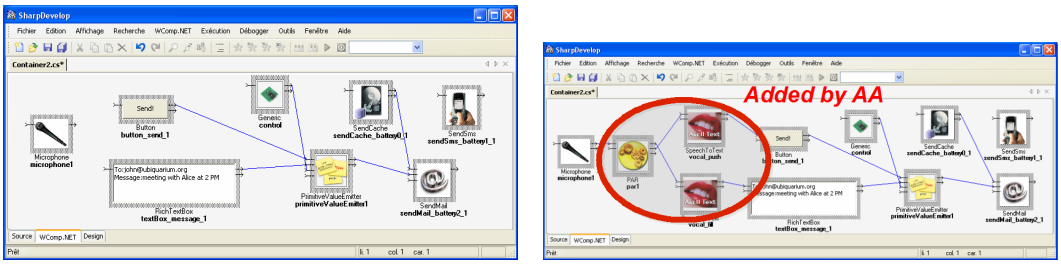
Dans la figure 6.19, nous avons représenté l'AA de consommation minimale. L'AA détecte à partir de son point de coupe, l'apparition du composant battery-low qui spécifie que la batterie est presque vide. L'AA est alors appliqué et l'email à envoyé est mis en cache et une notification est également envoyée à la batterie pour lui signaler de ne plus fournir d'énergie.

> *Un aspect d'assemblage permet de recalculer l'assemblage de composants pour rediriger le flot d'événements. En effet dans cet exemple, le flot d'événements modifié agit sur la consommation du système en changeant les services utilisés de l'infrastructure.*

Contrôle vocal : Voici un exemple de l'utilisation d'un système main-libre. Nous définissons deux types d'aspects :

– Aspect de contrôle vocal : ce module est dédié à la connexion du système de contrôle vocal à certains widgets de l'interface graphique.
– Aspect d'entrée vocale : ce module est utilisé pour effectuer la reconnaissance vocale afin de permettre d'entrer des messages sous forme de texte dans le système.

Ces aspects d'assemblage sont appliqués à l'application de base afin de dynamiquement construire les connexions entre les composants et les services pour affecter au système un comportement cohérent. Les fonctionnalités de base sont sélectionnées exclusivement par le contexte.



(a) Avant l'adaptation (b) Après l'adaptation

FIGURE 6.20 – Adaptation à l'application pilote en utilisant la voix

```
assembly base (vAddress,vTitle,recogAddress,recogTitle):
  buttonAddress : 'Windows.Forms.Button' ;
  fieldAddress : 'Windows.Forms.TextField' ;

  buttonTitle : 'Windows.Forms.Button' ;
  fieldTitle : 'Windows.Forms.TextField' ;

  buttonText : 'Windows.Forms.Button' ;
  fieldText : 'Windows.Forms.TextField' ;
```

FIGURE 6.21 – Assemblage de base

L'assemblage de base constitue un système d'envoi d'email que nous allons adapter à une situation de modification du contrôle. Ainsi, nous passons d'un mode d'entrée par les dispositifs standards tels que le clavier et la souris à un mode d'entrée par reconnaissance vocale.

Un premier aspect d'assemblage permet d'intégrer à l'assemblage de base la reconnaissance vocale de l'adresse et du sujet.

```
pointcut pcAddrTitle:
  vAddress := /fieldAddress*/
  vTitle := /fieldAddress*/ { print 'fieldTitle' substr($1,12) }
  recogAddress := /recogAddress/
  recogTitle := /recogTitle/

advice RecogAddrTitle (vAddress,vTitle,recogAddress,recogTitle):
  sttAddress : 'WComp.Services.SpeechToText' ; ttsAddress : 'WComp.Services.TextToSpeech' ;
  recogAddress.^EmitErase →( vAddress.Clear ) ; recogAddress.^EmitBool →( sttAddress.set_listenFlag )
  recogAddress.^EmitText →( sttAddress.ListenToText ) ; recogAddress.^DisplayReport →( ttsAddress.Speak )
  sttAddress.^SendSpokenText →( vAddress.set_Text )

  sttTitle : 'WComp.Services.SpeechToText' ; ttsTitle : 'WComp.Services.TextToSpeech' ;
  recogTitle.^EmitErase →( vTitle.Clear ) ; recogTitle.^EmitBool →( sttTitle.set_listenFlag )
  recogTitle.^EmitText →( sttTitle.ListenToText ) ; recogTitle.^DisplayReport →( ttsTitle.Speak )
  sttTitle.^SendSpokenText →( vTitle.set_Text )
```

FIGURE 6.22 – AA de contrôle vocal (adresse et titre du message)

Un second AA permet d'instrumenter le contenu du message et le sujet de l'email :

Nous avons un premier point de coupe qui permet l'instanciation des variables du premier AA déclaré dans le paragraphe précédent. Son code permet de construire des paires de composants adresse et titre ayant un même préfixe. Le dernier point de coupe permet de contrôler l'application de l'AA correspondant au moteur vocal.

Leur fusion et composition sont mis en annexe.

> *Ce qu'il faut retenir ici, c'est que la propriété de symétrie permet en effet de ne pas avoir à mettre en place un mécanisme qui permet de faire la distinction entre les aspects d'assemblage. Les conflits entre les aspects sont gérés à partir d'un ensemble de règles logiques prédéfinies.*

6.6 Mise en œuvre dans un Bâtiment Haute Technologie

Nous allons présenter les scénarios que nous avons développés au CSTB. Un aspect est utilisé pour décrire un scénario d'application particulier. L'architecture des composants peut évoluer de manière indépendante.

Nous présentons, dans cette section, quatre sous-systèmes adaptatifs : le câblage par profil, le système d'alarme, le système d'éclairage et l'intégration du volet.

```
pointcut pcMsgTxt:
  vMessage := /fieldMessage*/
  vText := /fieldMessage*/ { print 'fieldText' substr($1,12) }
  recogMessage := /recogMessage/
  recogText := /recogText/

advice RecogMesTxt(vMessage,vText,recogMessage,recogText):
  sttMessage : 'WComp.Services.SpeechToText' ; ttsMessage : 'WComp.Services.TextToSpeech' ;
  recogMessage.^EmitErase →( vMessage.Clear ) ; recogMessage.^DisplayReport →( ttsMessage.Speak )
  recogMessage.^EmitBool →( sttMessage.set_listenFlag ) ; recogMessage.^EmitText →
( sttMessage.ListenToText )

  sttText : 'WComp.Services.SpeechToText' ; ttsText : 'WComp.Services.TextToSpeech' ;
  recogText.^EmitErase →( vText.Clear ) ; recogText.^DisplayReport →( ttsText.Speak )
  recogText.^EmitBool →( sttText.set_listenFlag ) ; recogText.^EmitText →( sttText.ListenToText )
```

FIGURE 6.23 – AA de contrôle vocal (message et sujet de l'email)

Dans l'implémentation des scénarios, nous voyons d'une part que les greffons expriment les relations structurelles entre les composants logiciels. D'autre part, les points de coupe permettent d'exprimer d'autres caractéristiques de ces scénarios comme ce qu'est, de manière générique (sous la forme d'une expression régulière), un système d'éclairage dans une maison. Un système d'éclairage ne peut pas être décrit et caractérisé (uniquement) par un schéma d'interconnexion.

6.6.1 Contexte expérimental

6.6.1.1 Architecture matérielle cible

Le but principal de cette plate-forme est de programmer des dispositifs pour une utilisation dans un studio équipé, en guise de laboratoire expérimental. En particulier, nous voulons offrir un ensemble de dispositifs matériels que les utilisateurs peuvent choisir et assembler pour construire son propre espace informatique ambiant. Nous ne disposons cependant pas de la plate-forme .NET sur le routeur. Nous avons utilisé des microcontrôleurs (AVR) et un routeur (WL500g Premium) constituant le système informatique. Le routeur contient l'implémentation complète de l'unité de conception AA Dans cette section, nous présentons les différents scénarios que nous avons mis en place dans le laboratoire GERHOME.

6.6.1.2 Infrastructure logicielle

Ne pouvant pas utiliser le container C# pour contenir l'assemblage de composants, nous avons donc porté et construit deux containers supplémentaires : Container Java (supposant l'installation d'un environnement Java) et Container C++ (sans pré-requis).

Ces deux implémentations existent pour les raisons (1) de portabilité de l'environnement Java et (2) des performances que l'on pourrait obtenir en ayant un code compilé en langage C++. Les containers que nous avons construits utilisent la *libUPnP* pour implémenter les services composites. Nous présentons dans cette section les exemples, les AAs et quelques composants pour illustrer leur utilisation.

6.6.2 Scénarios et mise en œuvre de WComp

Couplage Système d'alarme et Système d'alarme : Dans cette section, nous présentons un exemple d'utilisation des aspects d'assemblage qui est utilisé dans une situation réelle d'observation et de déclenchement d'alarme. Cet exemple permet de déclencher une alarme lorsqu'une entité (qui peut être une personne, un appareil, etc.) n'a pas été active pendant une durée de 25 secondes.

Le greffon suivant (Figure 6.24) implémente la fonctionnalité de système d'alarme. C'est un système d'alarme basique qui comprend un capteur de présence et une sirène d'alarme. Le greffon

connecte les deux sorties du capteur de présence **^Somebody** et **^Nobody** aux entrées respectives **On** et **Off** de la sirène d'alarme. Nous indiquons, dans ce greffon, que d'autres connexions peuvent être faites avant le déclenchement de la sirène d'alarme.

```
advice presalarm (pres, siren) :
  pres.^Somebody →( call ; siren.On )
  pres.^Nobody →( call ; siren.Off )
```

FIGURE 6.24 – Système d'alarme

```
advice s0l0(genswitch, genlight):
  genswitch.^On →( call + genlight.LOn )
  genswitch.^Off →( call + genlight.LOff )
```

FIGURE 6.25 – Système d'éclairage – Greffon

Système d'éclairage : Le système d'éclairage est exprimé à la fois à travers un greffon et également par un point de coupe. D'abord, le greffon permet d'exprimer la relation structurelle basique qu'il existe entre une lampe et un interrupteur. Nous retrouvons cette relation dans la Figure 6.25. Elle dit que les événements **^On** et **^Off** sont connectés aux entrées respectives **LOn** et **LOff** d'une lampe et que ces signaux peuvent être connectés à d'autres composants sans problème d'ordonnancement.

```
pointcut pc1:
  genlight=/light*/
  genswitch=/light*/ { a=substr(\$1,6); print "switch" a; }
```

FIGURE 6.26 – Système d'éclairage – Point de coupe

Nous exprimons maintenant le caractère générique d'un système d'éclairage. Dans cet exemple, nous faisons correspondre chaque interrupteur portant le numéro i (comme **switch97** portant le numéro 97) à la lampe ayant le même numéro i. Nous remarquerons que le programme AWK de la seconde ligne effectue un remplacement du préfixe "light" du nom de composant par le préfixe "switch".

L'exemple fig. 6.27 peut impliquer deux greffons dupliqués Ex1 et Ex2, les paramètres ne sont pas associés aux paramètres ayant le même nom : user2 est associé à err1 et user1 à err2.

La décision d'intégrer le greffon d'adaptation à un point de coupe spécifié suit les règles suivantes : (1) uniquement les premières colonnes complètes du tableau deviennent des paramètres des greffons dupliqués (dans cet exemple, seulement les deux premières colonnes deviennent des paramètres). (2) L'ordre des noms dans la première ligne user2, user1 peut changer. C'est pourquoi, pour appliquer un greffon de manière déterministe, les lignes doivent être triées.

> *Cet exemple de système d'alarme nous permet d'illustrer la manière dont les aspects d'assemblage peuvent être utilisés pour répartir des modifications à travers tout un assemblage de composants. Cette duplication est spécifiée au niveau des points de coupe.*

Câblage par profil Nous spécifions d'abord trois greffons ISL. Le premier greffon contient la spécification du schéma d'assemblage des capteurs et actionneurs de l'appartement correspondant à un certain profile de personne dite rouge (fig. 6.28).

Au niveau du langage intermédiaire ISL4WComp, nous distinguons quatre règles d'adaptation et spécifions que l'adaptation se fait sur quatre composants identifiés par s_0, l_0, s_1 et l_1. La première

```
advice Ex1 (user2, err1):
 user2.^Out →( if (err1.Check) call )
 err1.Check →( err1.Start ; call )

advice Ex2 (user1, err2):
 user1.^Out →( if (err2.Check) call )
 err2.Check →( err2.Start ; call )
```

FIGURE 6.27 – Duplication et système d'alarme

règle (ligne 2) concerne la sortie **^On** que l'on reconnaît grâce au préfixe '^' du composant s_0. Elle indique que la diffusion de l'événement **On** au niveau des connexions de cette sortie qui sont déjà établies vont être mis en concurrence avec l'activation de l'entrée **LOn** du composant l_0. Les trois lignes suivantes (lignes 3 à 5) reflètent la même adaptation mais pour des composants et des entrées-sorties différentes.

```
advice persoRouge (s0, l0, s1, l1) :
  s0.^On →( call + l0.LOn )
  s0.^Off →( call + l0.LOff )
  s1.^On →( call + l1.LOn )
  s1.^Off →( call + l1.LOff )
```

FIGURE 6.28 – Greffon d'adaptation pour le profile rouge

Un deuxième AA (fig. 6.29) contient la spécification du schéma de câblage des capteurs et actionneurs de l'appartement correspondant un certain profile de personne que nous avons qualifié de bleu.

Comme pour le greffon ci-dessus, au niveau du langage intermédiaire ISL4WComp, nous distinguons quatre règles d'adaptation et spécifions que l'adaptation se fait sur quatre composants identifiés par s_0, l_0, s_1 et l_1. La première règle (ligne 2) concerne la sortie **^On** du composant s_0. Elle indique que la diffusion de l'événement **On** au niveau des connexions de cette sortie qui sont déjà établies vont être mis en concurrence avec l'activation des entrées **LOn** du composant l_0 et l_1. La ligne suivante reflètent la même adaptation mais pour des composants et des entrées-sorties différentes. Les deux dernières lignes (4-5) sont les mêmes que celles du greffon précédent. Cette différence a pour effet d'actionner au même moment les composants l_0 et l_1 en fonction de s_0.

```
advice persoBlue (s0, l0, s1, l1, h) :
  s0.^On →( call + l0.LOn + l1.LOn )
  s0.^Off →( call + l0.LOff + l1.LOff )
  s1.^On →( call + l1.LOn )
  s1.^Off →( call + l1.LOff )
```

FIGURE 6.29 – Profile bleu

Enfin, le dernier greffon contient l'assemblage de composants nécessaire à l'ajout d'un composant **Human** à l'assemblage de composants. En effet, il ajoute à l'assemblage lorsqu'il est sélectionné, le composant **creator** (ligne 2). Il connecte ce composant de manière conditionnelle à l'entrée **Create** de **creator** ou à **Destroy** (ligne 5 et 7). Le composant **creator** est également connecté au conteneur pour l'ajout effectif et le retrait de composants.

Nous sélectionnons trois aspects d'assemblage. Le premier est l'assemblage virtuel pour l'utilisateur, le second pour l'utilisateur, le troisième assemblage est un assemblage permettant d'intégrer au firmware du contrôleur le logiciel permettant de reconnaître un utilisateur particulier représenté par l'aimant.

Cela peut servir dans le cadre effectif de l'aide des personnes âgées en modifiant dynamiquement l'infrastructure des capteurs/actionneurs pour cibler certaines déficiences.

```
advice creator (id, container) :
  creator : Creator ;
  id.^Change →(
      if (id.Someone) {
        creator.Create
      } else {
        creator.Destroy
      }
  )
  creator.^AddComponent →(container.AddComponent)
  creator.^RemoveComponent →(container.RemoveComponent)
```

FIGURE 6.30 – Créateur

Cet exemple permet de montrer que les aspects d'assemblage peuvent être déclenchés par la présente d'individus dans l'environnement. Pour que cela soit possible, chaque individu est représenté par un service de l'infrastructure.

Intégration d'un nouveau dispositif : Le nouveau dispositif est accompagné de son aspect d'assemblage d'intégration.

Dans le greffon de la Figure 6.31, nous substituons la connexion à l'entrée **On** de la lampe **light** par le comportement suivant. Nous effectuons un test de la luminosité renvoyée par un capteur équipant le volet **shutter**. Si celui-ci renvoie vrai (sous-entendu que la luminosité est suffisante), alors nous ouvrons le volet électrique. Dans le cas contraire, nous exécutons un **call** c'est-à-dire l'action déroutée, à savoir l'allumage de la lampe. Nous remarquerons à la ligne 9 que l'extinction de la lampe s'accompagne également de l'ordre de fermeture des volets électriques.

```
advice volet (light, shutter) :
  light.LOn →(
    if (shutter.Bright) {
      shutter.Open
    } else {
      call
    }
  )
  light.LOff →( call + shutter.Close )
```

FIGURE 6.31 – Intégration du volet

Écriture des points de coupe : L'écriture des points de coupe permet de raccrocher les greffons que nous avons définis précédemment aux composants logiciels de l'application.

Le premier point de coupe concerne le système de câblage par profil. Ce point de coupe est très simple. C'est le point de coupe tautologique. Cela signifie que les variables utilisées dans les greffons correspondent aux identifiants des composants logiciels. Aucun traitement n'est donc effectués au préalable sur celles-ci.

```
pointcut pc0: tautologie
```

FIGURE 6.32 – Câblage par profil

Ensuite, au niveau du point de coupe pour le système d'alarme, les identifiants des composants logiciels ne correspondent pas aux variables utilisées dans les greffons. Nous définissons donc une correspondance ici 1-1 entre les variables et les identifiants des composants. Notamment, la variable **pres** doit correspondre à un ID **p0** et la variable **siren** à **a**.

```
pointcut pc10:
  pres←/p0/
  siren←/a/
```

FIGURE 6.33 – Système d'alarme

Puis, nous voyons dans l'exemple qui suit un exemple d'enchaînement de deux points de coupe **pc1** et **pc2**. Ce dernier point de coupe permet de traduire les variables fournies par le greffon et le calcul du premier point de coupe afin de les faire correspondre aux dénominations utilisées dans l'assemblage de composants de l'application.

```
pointcut pc2 after pc1:
  genlight←/light∗/ { a=substr($1,6); print "l" a; }
  genswitch←/switch∗/ { a=substr($1,7); print "s" a; }
```

FIGURE 6.34 – Système d'éclairage

Enfin, nous retrouvons dans ce dernier exemple une correspondance 1-1 au niveau du point de coupe.

```
pointcut pc3:
  light←/l0/
  shutter←/v0/
```

FIGURE 6.35 – Intégration du volet

Deux sous-systèmes alarme et éclairage ont été mis en place. On greffe un assemblage de composants pour partager l'information de présence du sous-système d'alarme pour en faire bénéficier le sous-système d'éclairage. On fait une transition sur la manière de greffer aux endroits précis les assemblages de composants.

> *Cet exemple nous montre l'intégration d'un volet électrique qui est un dispositif de l'infrastructure apportant avec lui son propre AA d'intégration.*

6.7 Conclusion et retour sur expérience

Application des AAs : Un aspect d'assemblage permet de recalculer l'assemblage de composants pour rediriger le flot d'événements. Le flot d'événement agit sur la consommation du système lorsque ce dernier déclenche l'utilisation de services de l'infrastructure qui en consomment. Les aspects d'assemblage peuvent être utilisés pour répartir des modifications à travers tout un assemblage de composants. Cette duplication est spécifiée au niveau des points de coupe.

AAs et composition : La propriété de symétrie permet en effet de ne pas avoir à mettre en place un mécanisme qui permet de faire la distinction entre les aspects d'assemblage. Les conflits entre les aspects sont gérés de manière globale à partir d'un ensemble de règles logiques prédéfinies.

Environnement physique et AAs : Les aspects d'assemblage peuvent être déclenchés par la présence d'individus dans l'environnement. Pour que cela soit possible, chaque individu est représenté par un service de l'infrastructure. L'intégration d'un dispositif de l'infrastructure consiste à apporter avec lui son propre AA d'intégration.

Chapitre 7

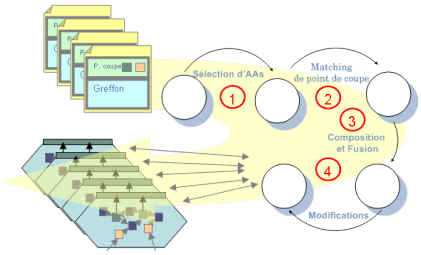
Évaluation des aspects d'assemblage

Sommaire

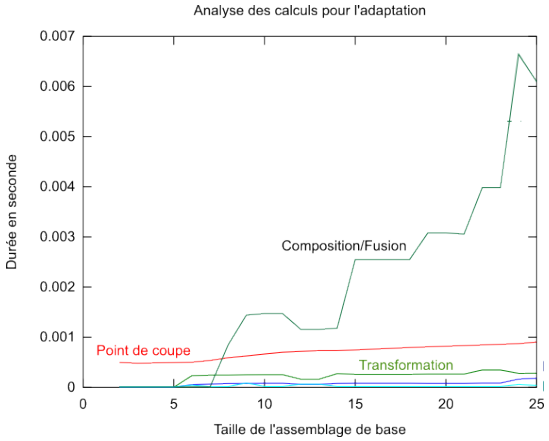
Dans ce chapitre, nous évaluons notre approche en commentant quelques expériences sur des cas que nous avons mis en œuvre. Le but de cette évaluation est de montrer les avantages de l'utilisation des aspects d'assemblage.

7.1 Le tissage

Nous avons vu que le tissage se découpe en plusieurs sous-étapes. Nous allons analyser les étapes qui sont critiques pour le temps de réponse de l'adaptation du système. D'après les algorithmes décrits dans le chapitre 5, le calcul du point de coupe (étape 2) dépend de la taille de l'assemblage de base et du nombre d'aspects d'assemblage – plus précisément, du nombre d'instances de greffon dans l'application. La composition et la transformation Greffon/Assemblage (étapes 3 et 4) dépendent seulement du nombre d'instances de greffon (figure 7.1(a)).



(a) Exemple d'évolution du temps de calcul du cycle d'adaptation.



(b) Comment évoluent ces grandeurs ?

FIGURE 7.1 – Mécanisme de tissage

La figure 7.1(b) illustre à petite échelle (une dizaine d'AAs), dans les conditions expérimentales décrites dans la section 7.1.1, l'évolution des différentes parties du mécanisme de composition dans le cas d'ISL4WComp. Nous analysons dans ce chapitre la manière dont le temps de calcul peut évoluer en fonction du nombre de composants et du nombre d'AAs que nous mettons en œuvre.

Dans les deux sections suivantes, nous décrivons les conditions expérimentales, puis nous distinguons les parties les plus coûteuses du mécanisme de composition.

7.1.1 Conditions expérimentales

Les expérimentations ont été effectuées sur un PC de bureau (1.6GHz). Pendant ces expérimentations, nous avons utilisé un scénario du Bâtiment Intelligent qui consiste à créer dynamiquement les interactions entre des lampes et des interrupteurs en créant un maillage complet.

Nous avons fait évoluer artificiellement le système en instanciant des composants Proxy d'interrupteur et de lampe de sorte à augmenter le nombre de composants et d'instances de greffon. Les composants apparaissent de manière aléatoire suivant une loi binomiale. Nous pouvons déjà déduire que le nombre d'instanciations énoncé précédemment est dépendant de cette loi.
Dans ce système, nous programmons une application de la manière suivante : nous considérons les actionneurs l_n (lampes) et les capteurs s_m (interrupteurs). Tous les dixièmes de seconde (pendant

```
pointcut pcAA0:
  s0← BEGIN {i=0;j=0} \
        /si([[:alnum:]]+)/ {i++;si[i]=$0} \
        /li([[:alnum:]]+)/ {j++} \
      END { for(k=1;k≤j&&k≤i;k++) print si[k]}
  l0← BEGIN {i=0;j=0} \
        /si([[:alnum:]]+)/ {i++} \
        /li([[:alnum:]]+)/ {j++;li[j]=$0} \
      END { for(k=1;k≤j&&k≤i;k++) print li[k]}

advice AA0 (s0,l0):
  s0.^On -> ( call ; l0.On )
  s0.^Off -> ( call ; l0.Off )
```

FIGURE 7.2 – Aspect d'assemblage utilisé pour l'évaluation

environ 20 secondes), un nouveau composant parmi les capteurs/actionneurs s_m et l_n est ajouté dans le container. Les indices n et m sont incrémentés de 0 à 300. A chaque fois qu'un nouveau composant est ajouté, le mécanisme de point de coupe est exécuté et un ensemble d'aspects d'assemblage est sélectionné et appliqué.

7.1.2 Coût des points de coupe et de la composition

Notre approche permet de factoriser ces modifications structurelles élémentaires dans un aspect d'assemblage qui peut alors être dupliqué en spécifiant des points de coupe adéquats. Un AA est alors défini une fois et puis appliqué n fois. De plus, les points de coupe isolent le schéma d'assemblage de l'assemblage de base. Il est responsable de l'affectation des variables du schéma (indépendantes de l'application) aux points de jonction (dépendants de l'application). Cela permet de renforcer l'indépendance et la réutilisabilité des schémas d'assemblage des AA.

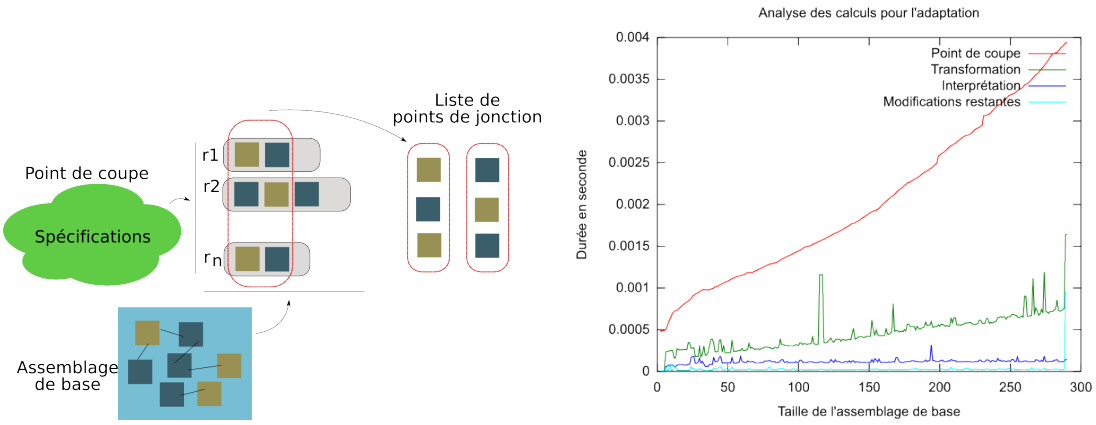
7.2 Coût des points de coupe

Pour le calcul des points de coupe, nous présenterons dans un premier temps le modèle général du calcul du point de coupe pour les aspects d'assemblage. Ensuite, nous étudierons deux cas particuliers : qui prennent en compte la construction d'une seule ou de plusieurs listes de paramètres pour les instances de greffon.

7.2.1 Modèle général - point de coupe AA

Le modèle général du calcul des points de coupe (figure 7.3(a)) consiste à prendre comme entrées les spécifications sous forme de règles $r_1, \ldots, r_n$ et l'assemblage de base afin d'indiquer pour chaque instance de greffon dupliqué, la liste des points de jonction associée.
La figure 7.3(b) illustre le temps de calcul des différentes parties de la composition d'aspects d'assemblage, et notamment, la prédominance du temps de calcul des points de coupe. Ces mesures ont été réalisés selon les conditions exprimées dans la section 7.1.1.

Soit D la durée du processus de correspondance des points de coupe. Soient $a_1, \ldots, a_n$ les paramètres du modèle et c le nombre de composants de l'assemblage de base. Soit κ le nombre d'AAs disponibles en moyenne pendant l'exécution de l'application. Notons δ_i le nombre de duplications de l'AA numéro i. Nous avons κ aspects d'assemblage et chacun d'entre eux est associé à un point de coupe. Ainsi, chaque point de coupe donne un nombre δ_i de duplications d'instances de greffon. Chaque duplication est soumise à un calcul pour définir le corps de son greffon. Nous avons la formule suivante pour le modèle de la correspondance du point de coupe :



(a) (b)

FIGURE 7.3 – Parmi les temps de calcul des différentes parties. On remarque que 3 parties sont coûteuses : le point de coupe, la composition (non représentée ici) et la transformation Greffon → Assemblage

D : durée du calcul des points de coupe
a_1, a_2 : paramètres du modèle
c : nombre de composants dans l'assemblage de base
κ : nombre d'AAs
δ_i : nombre de duplications de l'AA_i

$$D = a_1 . \sum_{i=1}^{\kappa} (\delta_i + 1) . c^2 + a_2$$

Dans l'algorithme que nous avons implémenté, l'ensemble des composants est parcouru deux fois. L'implémentation pourra être améliorée en optimisant cet algorithme qui n'est pas, pour ce premier prototype, optimale.
La quantité δ_i n'est pas facile à déterminer parce qu'elle peut dépendre des caractéristiques de l'usage ou bien des caractéristiques internes à l'application. Typiquement, cette quantité dépend de la manière dont les composants apparaissent dans l'application.

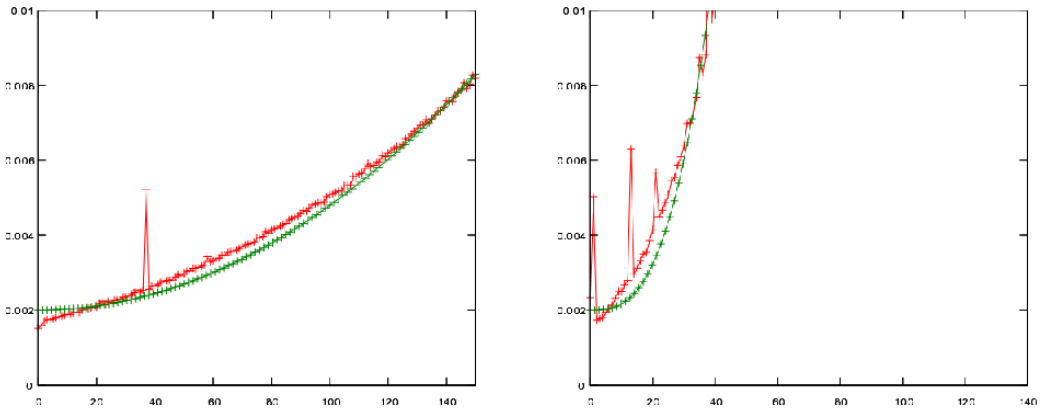
Nous avons expérimenté un modèle probabiliste de la durée du processus et en avons déduit une estimation de la quantité δ_i. Ces expérimentations sont décrite dans la section suivante.

7.2.2 Deux cas particuliers

Nous explorons alors deux cas : le premier consiste à ne pas permettre la duplication de l'aspect d'assemblage AA_0 ($\forall i, \delta_i = 0$) tandis que le second le permet ($\forall i, \delta_i \geq 0$). Nous confrontons ensuite notre modèle aux mesures réelles.

7.2.2.1 Sans duplication

Le premier cas que nous analysons ne contient pas de duplication. Nous obtenons le graphe de la figure 7.4(a). Les conditions expérimentales concernant les mesures illustrées dans la figure 7.4 sont celles énoncées dans la section 7.1.1.



(a) Sans duplication

(b) Avec duplications

FIGURE 7.4 – Modèle/Expérimentation des points de coupe

Cela signifie que pour chaque aspect d'assemblage i, le nombre de duplication d_i est égale à 0. Nous n'utilisons qu'un seul AA, ainsi la valeur de κ est fixée à 1. La formule est alors simplifiée :

D_1 : durée du calcul des points de coupe sans duplication
a_1, a_2 : paramètres du modèle
c : nombre de composants dans l'assemblage de base
κ : nombre d'AAs
δ_i : nombre de duplications de l'AA$_i$ $= 0$

$$D_1 = a_1.\sum_{i=1}^{\kappa}(0+1).c^2 + a_2 = a_1.\kappa.c^2 + a_2 = a_1.c^2 + a_2$$

Nous avons également représenté dans la figure 7.4(a) les résultats expérimentaux en plus du modèle de prédiction. Nous remarquons que le modèle correspond aux résultats expérimentaux après un calcul de régression. Cette première expérience nous permet de déterminer les paramètres du modèle. Nous obtenons :

$$a_1 = 280.10^{-9} \text{ et } a_2 = 2.10^{-3}$$

7.2.2.2 Avec duplications

Le second cas consiste à dupliquer systématiquement les AAs. Cependant, dans un tel cas, nous avons besoin de connaître l'allure de la courbe représentant le nombre d'instances de greffons d_i. En fait, nous savons qu'un composant apparaît de manière aléatoire, choisi parmi les descriptions l_x et s_y. Par conséquent, le greffon a une probabilité d'être instancié de $\frac{1}{2}$. Nous prenons ainsi $\delta_i = \frac{1}{2}$ qui correspond également à la probabilité d'obtenir le bon paramètre. Après simplification, nous obtenons la formule suivante pour le modèle de performance :

D_2 : durée du calcul des points de coupe avec duplications
a_1, a_2 : paramètres du modèle
c : nombre de composants dans l'assemblage de base
κ : nombre d'AAs $= 1$
δ_i : nombre de duplications de l'AA$_i$ $= \frac{1}{2}$

$$D_2 = a_1.\kappa.(1 - \frac{1}{2}).c^2 + a_2 = a_1.\frac{3}{2}.c^2 + a_2$$

La figure 7.4(a) montre que les paramètres correspondent effectivement et peuvent être utilisés pour caractériser le système matériel. Nous pouvons également voir des irrégularités dans les résultats expérimentaux. Elles sont causées par la collecte et l'allocation de mémoire qui sont des opérations coûteuses.
Le second calcul coûteux du tissage est la composition que nous allons voir dans la section suivante.

7.3 Coût de la composition

Comme le calcul du point de coupe, nous présenterons d'abord un modèle général de la composition d'AAs. Ensuite, nous étudierons un cas particulier.

7.3.1 Modèle général : composition d'AAs dans le cas d'ISL4WComp

La durée prise par l'algorithme de composition (incluant la fusion logique d'AAs) dépend du greffon de l'AA. En effet, même si nous pouvons affecter une mesure à un AA en termes du nombre de composants et de connecteurs qui sont nécessaires pour sa description, il est toujours difficile de prédire les règles qui seront exécutées afin de composer et de fusionner éventuellement les AAs. Par exemple, les règles terminales telles que la règle de fusion d'un message avec un call coûte moins que des règles de fusion récursives telles que celles d'une fusion entre IF. D'ailleurs, la sélection de la règle dépend du greffon. Par conséquent, nous ne pouvons pas fournir de modèle précis du processus de composition, mais plutôt un modèle approché.

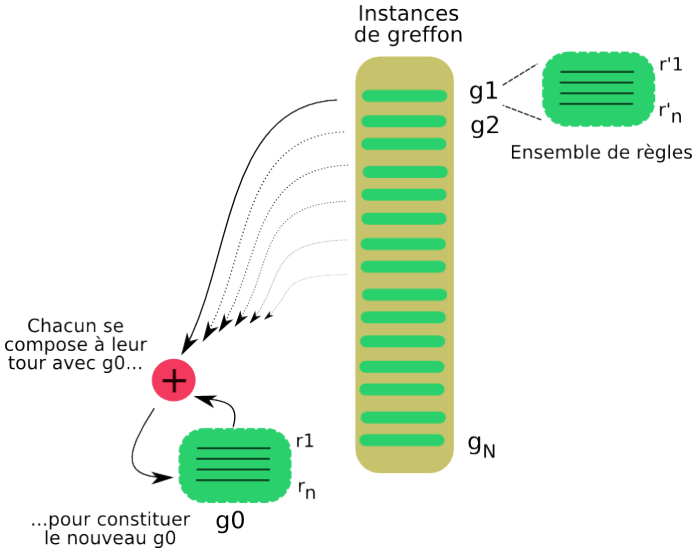


FIGURE 7.5 – Composition

Nous prenons deux (sous-)arbres α et β. Nous définissons ensuite en fonction de la hauteur des arbres h_α et h_β. Ce calcul du coût de la composition est probabiliste. La composition est un mécanisme qui est composé de deux opérations : la juxtaposition et la fusion. Nous venons de voir ci-dessus le coût probabiliste de la fusion en fonction de la hauteur maximum de deux arbres à fusionner. Le coût de la juxtaposition est d'une unité de temps par instance de greffon.

Nous appelons l'instance de greffon de base, l'assemblage de base traduite sous la forme de règles en instance de greffon, notée g_0 (nous notons n^0 le nombre de règles qui le compose). Pour donner une allure de la composition, nous allons considérer qu'il est donné un certain nombre N d'instances de

greffon. Notons n^i, le nombre de règles décrites dans un greffon, p_i, la probabilité que l'instance de greffon g_i soit en conflit avec l'instance du greffon de base g_0. La quantité C représente la hauteur des arbres à fusionner est à priori dépendante des greffons g_i et de greffon de base g_0. Nous obtenons ainsi la formule générale suivante :

K : durée de la composition
τ_0, τ_1, τ_2 : paramètres du modèle
n^0 : le nombre de règles décrites dans un greffon de base g_0
n^i : le nombre de règles décrites dans un greffon g_i
p_i : probabilité que l'instance g_i soit en conflit avec l'instance de greffon de base
$C(g_0, g_i)$: représente la hauteur des arbres à fusionner

$$K = \tau_0 . n^0 \sum_{i=1}^{N} n^i \left(1 + p_i \tau_1 C(g_0, g_i)\right) + \tau_2$$

Notons que τ_0, τ_1 et τ_2 sont des constantes qui dépendent du système sous-jacent. Nous simplifions le modèle de la fusion en disant que la quantité $C(g_0, g_i)$ est proportionnelle à la hauteur minimale entre les deux arbres considérés. Ainsi :

$$C(g_0, g_i) = k_0 . min(h_\alpha, h_\beta)$$

où k_0 est une constante dépendante du système sous-jacent.

7.3.2 Étude d'un cas particulier

Toujours en suivant les conditions expérimentales de la section 7.1.1, nous obtenons la courbe de Composition/Fusion de la figure 7.6 dans laquelle nous avons superposé une courbe représentant le modèle théorique. Le paragraphe suivant explique la démarche que nous avons eu pour construire le modèle théorique correspondant.

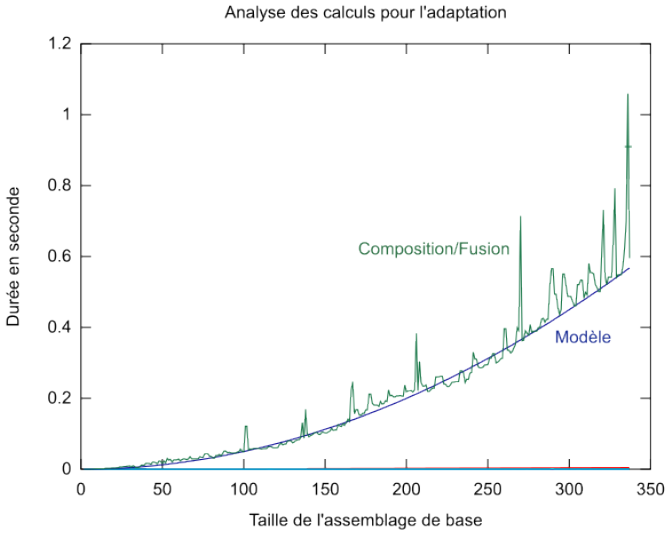


FIGURE 7.6 – Évolution de la composition

Dans ce cas particulier, on note j le nombre de fois que la composition a été effectuée depuis le démarrage du système. D'après les hypothèses énoncées, nous pouvons déjà simplifier la formule générale de la composition en procédant par étapes. D'abord, l'unique aspect d'assemblage AA_0 se

compose d'un greffon ayant deux règles. Donc, $n^i = 2$ pour tout i. Ensuite, concernant le greffon de base, à chaque fois qu'un composant lampe ou interrupteur est ajouté à l'assemblage, il a une chance sur deux d'être instancié une fois une de plus que précédemment. Donc, nous proposons $n^0 = \frac{1}{2}.j$. De la même manière, une fusion a lieu toutes les fois ou une nouvelle instance de greffon est appliquée. Par conséquent, une fusion a lieu une fois sur deux. D'où, $p_i = \frac{1}{2}$. La hauteur minimale des arbres constituant g_i et g_0 est de 2 : $C = 2.k_0$ est constante.

Nous obtenons :

$K(j)$: durée de la composition en fonction de j
$\tau_0, \tau_1, \tau_2, \tau'$: paramètres du modèle
n^0 : le nombre de règles décrites dans un greffon de base $\mathrm{g}_0 = \frac{1}{2}j$
n^i : le nombre de règles décrites dans un greffon $\mathrm{g}_i = 2$
p_i : probabilité que l'instance g_i soit en conflit avec l'instance de greffon de base $= \frac{1}{2}$
$C(g_0, g_i)$: représente la hauteur des arbres à fusionner $= 2.k_0$

$$K(j) = \tau_0 \sum_{i=1}^{n} 2 \left(\frac{1}{2}j\right) \left(1 + \frac{1}{2}\tau_1 \times 2.k_0\right) + \tau' \tag{7.1}$$

$$= 2\tau j^2 + \tau' \tag{7.2}$$

Nous avons représenté dans la figure 7.6 les résultats expérimentaux. Nous pouvons en déduire les valeurs des constantes :

$$K(j) = 5.2 \times 10^{-6} j^2$$

Ainsi, on trouve $\tau = 2.6 \times 10^{-6}$.

7.4 Synthèse

Nous avons deux principales approches pour l'adaptation des systèmes informatiques ambiants. La première concerne l'adaptation par composition d'assemblage et la seconde concerne la réutilisation par modification des points de coupe. Nous pouvons jouer sur les performances des adaptations proposées en manipulant trois leviers : (A) les algorithmes de composition/point de coupe/transformation, (B) ajuster les paramètres en changeant le système matériel, (C) écrire efficacement les spécifications pour l'adaptation.

A. Composition, Point de coupe et Transformation : Les mécanismes de composition, de point de coupe et de transformation sont rangés du plus coûteux vers le moins coûteux. Par exemple, au niveau de la performance, un facteur 10 peut séparer ces trois concepts. Nous devons répartir correctement les manières de décrire une adaptation entre les points de coupe et la composition/fusion.

B. Identification de paramètres : Nous avons identifié les paramètres de notre modèle s'appuyant sur un PC. Nous obtenons une approximation des paramètres du modèle en effectuant une régression linéaire : nous obtenons alors une approximation des paramètres a_1 et a_2. Ces caractéristiques sont constantes pour l'évaluation des trois mécanismes : point de coupe, composition et résolution de conflit. Ils constituent les paramètres du système matériel :

$a_1 = 280.10^{-9}$, $a_2 = 2.10^{-3}$ *et* $\tau = 2.6.10^{-6}$ *pour un PC de bureau servant de gateway, pour le cas d'une gestion de l'éclairage avec un maillage complet.*

C. Simplification des spécifications pour l'adaptation : Au niveau de l'implémentation, nous avons construit un indicateur à partir du nombre de modifications élémentaires afin de mesurer, pour des cas d'utilisation réels, le degré à partir duquel l'application de notre technique peut simplifier les spécifications de l'adaptation. Cette mesure a pour but de montrer combien de modifications structurelles simples (MSE) sont nécessaires afin de se rapprocher et d'accomplir de tels services d'adaptation sur des systèmes orientés composant et de montrer la complémentarité de chaque approche. Pour prendre un exemple, pour un ensemble de programmes aléatoires, l'AA simple donné dans le Figure 7.2 produit 2 MSE pour 1 intégration. Pour 30 duplications, nous atteignons une moyenne de 36 MSE.

Les aspects d'assemblage adaptent des services composites et des assemblages de composants. Lorsque l'on note N, le nombre d'aspects d'assemblage qui sont disponibles et p, le nombre d'aspects d'assemblage qui sont sélectionnés, nous pouvons effectuer une composition de ces p aspects d'assemblage parmi N aspects d'assemblage disponibles. Nous comparons ainsi cette approche à celle du scripting et la modification directe de l'assemblage. Soit m, le nombre de ports et n, le nombre de liaisons, on doit alors effectuer dans le cas général m fois n compositions. Les aspects d'assemblage proposent ainsi de réduire la complexité en réduisant le nombre de lignes de code à fournir.

Nous avons, dans ce chapitre, construit un modèle probabiliste des performances du service d'adaptation dynamique. Ce modèle est paramétré ce qui permet de l'adapter à différentes cibles matérielles et pouvoir estimer le temps de calcul de chaque partie du système. Nous avons deux manières d'utiliser des aspects d'assemblage. La première consiste à autoriser la sélection des AAs par l'utilisateur. La seconde peut être du contexte-aware en sélectionnant les AAs sur des conditions contextuelles.

Quatrième partie

Conclusion et perspectives

Chapitre 8

Conclusion & Perspectives

Nous avons mis en évidence que la problématique de l'adaptation dynamique dans un environnement informatique ambiant consiste à ajuster les fonctionnalités offertes par l'application en fonction de l'infrastructure des dispositifs. Notre concept d'adaptation dynamique a été utilisé dans divers projets dont FAROS, ErgoDyn et GERHOME. Dans le projet RNTL FAROS, nous avons réalisé une correspondance entre l'application des aspects d'assemblage et la vérification de contrats d'assemblage de composants logiciels dans des espaces ambiants. Dans le projet RNTS ErgoDyn, il a été utilisé pour l'adaptation des interfaces de services à la diversité des téléthèses de myopathes. Une téléthèse est un logiciel qui permet à un grand handicapé d'agir sur son environnement. Le logiciel permet d'augmenter les capacités d'adaptation. Enfin, dans le projet GER'HOME, nous avons utilisé les locaux du laboratoire expérimental (studio équipé en capteurs) pour expérimenter en grandeur réelle des scénarios illustrant l'*intelligence ambiante*.

L'objectif de cette thèse était de proposer une approche originale pour le développement d'adaptations d'applications logicielles pour l'informatique ambiante. L'informatique ambiante engage de *nombreux* dispositifs *variés* intégrés aux objets de tous les jours qui collaborent pour adapter des assemblages de composants. Cette adaptation peut être partielle (on modifie une partie de l'application existante) ou totale (on reconstitue intégralement une application) à partir d'aspects d'assemblage. Dans ce dernier cas, des applications peuvent alors *émerger* dynamiquement puisque l'instanciation des aspects d'assemblage est dirigée par la présence des dispositifs de l'infrastructure. Nous nous intéressons à la construction des applications étant donné un ensemble de dispositifs communicants selon les principes suivants :

1. Respect des modules blackbox (dispositifs de l'espace ambiants auxquels nous n'avons pas accès hormis à travers leurs interfaces – implémentation masquée volontairement par des brevets par exemple) ou les limites logiciel/matériel,
2. Communication événementielle pour gérer efficacement la réactivité intrinsèque des systèmes informatiques ambiants,
3. Modularité transverse à son paroxysme pour gérer les adaptations : création d'une application entière à partir d'aspects produisant des reconfigurations structurelles plutôt que de l'injection de code pour respecter le premier point : de l'adaptation à l'émergence.

Notre modèle décrit une plate-forme qui a pour objectif de gérer l'émergence d'applications informatiques ambiantes. Il définit une solution complète en s'appuyant sur l'IDE WComp.NET pour la conception, le déploiement et l'adaptation des dispositifs évoluant dans des espaces ambiants.

Nous avons d'abord développé un *designer* pour les aspects d'assemblage intégrant deux langages dédiés : le premier inspiré d'ISL [30] – un langage de spécification d'interactions – et un second BSL – gestion de la coordination – permettant de gérer le flot d'événements en incluant la notion de temps. Ensuite, nous avons réalisé plusieurs extensions de la plate-forme WComp afin de porter le modèle sur des systèmes enfouis ne disposant pas de la technologie .NET.

Perspectives

Nous avons travaillé sur les différents modèles de la plate-forme WComp et dressé une liste de nombreuses évolutions possibles. Elles proviennent des limitations de l'implémentation actuelle ou des standards utilisés (voir chapitre 6). Voici donc quelques perspectives que nous jugeons intéressantes.

Échanges implicites dans le modèle de composant légers : La définition de Szyperski [203] établit qu'un composant est une unité de composition ayant des interfaces spécifiées de façon contractuelle et possédant uniquement des dépendances de contexte explicites (voir page 30). Cependant, nous pouvons toujours, par programmation, échanger des informations entre des composants sans avoir recours aux ports. Les raisons peuvent être justifiées (garantir la clarté de l'assemblage en ne faisant apparaître que les connexions importantes pour l'application). Par exemple, nous pouvons programmer des accès à une ressource déjà utilisée. Comment et devons-nous alors exprimer ce partage de ressources dans notre modèle ? Cette approche nécessite une analyse fine.

ADL et greffons d'aspects d'assemblage : Dans notre travail, nous avons développé deux langages dédiés ISL4WComp et BSL pour exprimer la composition des greffons. Ces langages décrivent implicitement les composants spécifiques qui sont les projections des aspects d'assemblage (composants de séquence, de condition, etc.). Cependant, il n'est pas souvent facile de faire la correspondance entre les sous-assemblages déduits des AAs et la spécification décrite dans une grammaire ad hoc. Une piste serait de formuler une grammaire unique qui exprimerait une adaptation sous la forme d'un assemblage : une étude approfondie serait nécessaire.

Enrichissement du modèle de points de coupe : Les mécanismes de déclenchement des adaptations sont en deux étapes dans nos travaux : d'abord, le système est sollicité par une modification de l'infrastructure des dispositifs ; ensuite, en fonction de ces modifications, une sélection dynamique des aspects d'assemblage à appliquer est réalisée. Cependant, cette deuxième phase peut être enrichie en proposant d'autres modèles que celui des filtres à composants que nous avons mis en place. Par exemple, une analyse incluant la notion de temps peut y être intégrée. D'autre part, la correspondance entre les points de jonction (composants) et les points de coupe les désignant peut être rendue plus flexible en proposant un modèle par classe d'équivalence comme une approche par ontologies distribuées par exemple.

Pré-sélection des aspects d'assemblage : La pré-sélection des aspects d'assemblage est effectuée par l'utilisateur. Nous pouvons aller plus loin en proposant une semi-automatisation de ces sélections. À la différence des sélections effectuées au niveau du point de coupe qui est une sélection s'appuyant sur les informations émanant de l'infrastructure, cette sélection peut avoir des critères liés à l'environnement physique. Ainsi, en plus d'un enrichissement du modèle des points de coupe, nous pouvons envisager faire intervenir des critères originaux par ce point d'entrée du système (mécanisme d'apprentissage des pré-sélections, etc.).

Vers une approche soustractive des greffons : Enfin, les aspects d'assemblage suivent une approche *additive* au sens où chaque aspect ajoute des éléments à l'assemblage, aux exceptions près des opérateurs de conditionnement de la composition (call, delegate). À aucun moment, nous pouvons spécifier le retrait d'une fonctionnalité à travers un greffon. S'agit-il d'un sous-assemblage à effacer s'il est présent ? Comment spécifier ces "*anti-greffons*" ?

Traitements des cas particuliers : Peut-on appliquer un aspect d'assemblage sur un composant spécifique ? En théorie, ce n'est pas impossible. Par contre, l'implémentation actuelle ne le prend pas en compte, en particulier, la fusion actuelle ne les gère pas. Cependant, cela ne prend pas tout

son sens, puisque le but est de faire émerger des applications à partir des services de l'infrastructure et non pas à partir d'applications déjà existantes.

Ce travail, j'espère, constitue un socle sur lequel nous pouvons de manière simple et intuitive déclarer des formes d'adaptation. Cela permet de s'affranchir de la rigidité des langages d'adaptation.

Liste des publications

Voici la liste des publications effectuées en relation avec les travaux de cette thèse :

Revue Internationale

1. J.-Y. Tigli, S. Lavirotte, G. Rey, V. Hourdin, **D. Cheung-Foo-Wo**, E. Callegari, M. Riveill. *WComp Middleware for Ubiquitous Computing : Aspects and Composite Event-based Web Services*, Annals of Telecom, 2008, à paraître.

Conférences internationales

1. **D. Cheung-Foo-Wo**, J.-Y. Tigli, S. Lavirotte, M. Riveill. *Self-adaptation of event-driven component-oriented Middleware using Aspects of Assembly* in Proceedings of the 5th International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC), California, USA, 26-30 november 2007
2. N. Bussière, **D. Cheung-Foo-Wo**, V. Hourdin, S. Lavirotte, M. Riveill, J.-Y. Tigli. *Optimized Contextual Discovery of Web Services for Devices* in Proceedings of the International Workshop on Context Modeling and Management for Smart Environments (CMMSE), IEEE, Lyon, France, 28-31 october 2007
3. **D. Cheung-Foo-Wo**, J.-Y. Tigli, S. Lavirotte, M. Riveill. *Contextual Adaptation for Ubiquitous Computing Systems using Components and Aspect of Assembly* in Proceedings of the Applied Computing (IADIS), IADIS, pages 563–567, Salamanca, Spain, 18-20 feb february 2007
4. **D. Cheung-Foo-Wo**, J.-Y. Tigli, S. Lavirotte, M. Riveill. *Wcomp : a Multi-Design Approach for Prototyping Applications using Heterogeneous Resources* in Proceedings of the 17th IEEE International Workshop on Rapid System Prototyping (RSP), pages 119–125, Chania, Crete, jun 2006
5. **D. Cheung-Foo-Wo**, G. Joulié, F. Grillon, J. Fuchet, J.-Y. Tigli. *Wcomp : Rapid Application Development Toolkit for Wearable Computer Based on Java* in Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, vol. 5, pages 4198–4203, Washington DC, USA, 5-8 oct 2003
6. A. Ressouche, V. Roy, J.-Y. Tigli, **D. Cheung-Foo-Wo**. *SAS architecture : Verification Oriented Formal Modeling of Concrete Critical Systems* in Proceedings of the Proceedings of the International Conference on Systems, Man and Cybernectics (IEEE SMC), IEEE, oct 2003

Articles de journaux nationaux

1. **D. Cheung-Foo-Wo**, M. Blay-Fornarino, J.-Y. Tigli, A.-M. Pinna-Déry, D. Emsellem, M. Riveill. *Langage d'aspects pour la composition dynamique de composants embarqués* Numéro spécial : Développement de logiciels par aspects : JFDLPA 2005 dans L'objet : coopération dans les systèmes à objets, 12 (2-3), pages 89-112, Hermes, avril 2006, ISBN 2-7462-1521-7

2. J.-Y. Tigli, **D. Cheung-Foo-Wo**, S. Lavirotte, M. Riveill. *Adaptation au contexte par tissage d'aspects d'assemblage de composants déclenchés par des conditions contextuelles* Numéro spécial : Adaptation et Gestion du Contexte dans RTSI Série ISI, 11 (5), pages 89–114, 2006, ISBN 2-7462-1672-8

3. J.-Y. Tigli, S. Lavirotte, **D. Cheung-Foo-Wo**. *Mobilité et Enseignement à Distance* Numéro spécial : Colloque TICE dans Journal International des Sciences de l'Information et de la Communication (ISDM), numéro 10, octobre 2003, ISSN 1265-499X

Conférences nationales

1. V. Hourdin, **D. Cheung-Foo-Wo**, S. Lavirotte, J.-Y. Tigli. *Ubiquarium Informatique : Une plate-forme pour l'étude des équipements informatiques mobiles en environnement simulé* (poster) in Proceedings of the Troisièmes Journées Francophones Mobilité et Ubiquité (UbiMob), Paris, France, 5-8 sep september 2006

2. **D. Cheung-Foo-Wo**, M. Blay-Fornarino, J.-Y. Tigli, S. Lavirotte, M. Riveill. *Adaptation dynamique d'assemblages de dispositifs par des modèles* in Proceedings of the 2èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM), jun 2006

3. **D. Cheung-Foo-Wo**, M. Blay-Fornarino, J.-Y. Tigli, A.-M. Pinna-Déry, David Emsellem, M. Riveill. *Langage d'aspects pour la composition dynamique de composants embarqués* in Proceedings of the 2ème Journée Francophone sur le Développement de Logiciels par Aspects (JFDLPA) , 2005

4. **D. Cheung-Foo-Wo**, J.-Y. Tigli, M. Riveill. *Architecture orientée composant et interactions implicites, application aux ordinateurs corporels* in Proceedings of the Premières Journées Francophones : Mobilité et Ubiquité, 2004

Rapports techniques

1. J.-Y. Tigli, M. Riveill, G. Rey, S. Lavirotte, V. Hourdin, **D. Cheung-Foo-Wo**, E. Callegari. *A Middleware for Ubiquitous Computing : WComp* Research Report I3S (University of Nice - Sophia Antipolis - CNRS), number I3S/RR-2008-01-FR, 23 pages, Janvier 2008

2. F. Balligand, M. Blay-Fornarino, H. Chang, **D. Cheung-Foo-Wo**, P. Collet, G. Dufrêne, V. Hourdin, S. Lavirotte, S. Mosser, A. Ozanne, A.-M. Pinna-Déry, N. Rivierre, L. Seinturier, J.-Y. Tigli. *Identification des modalités de prise en charge des contrats pour chaque plate-forme cible* Research Report RNTL Faros, number F.3.1, 1–93 pages, jun 2007 (BibTeX)

Logiciels et Tutoriaux

1. S. Weibel, J.-Y. Tigli, S. Lavirotte, **D. Cheung-Foo-Wo**, V. Hourdin, M. Riveill. *SharpWComp* Septembre 2006, IDDN.FR.001.090016.000.S.C.2007.000.10600

2. J.-Y. Tigli, **D. Cheung-Foo-Wo**. *Tutorial Wcomp : Environnement de développement pour prototypage rapide multi-dispositif orienté-composant et programmation visuelle*, Septembre 2005

Bibliographie

[1] M. Weiser, « The Computer for the 21st Century », *Scientific American*, Vol. 1, p. 94–104, sep 1991.

[2] ISTAG, « Ambient Intelligence : From vision to reality », Rapport Technique, ISTAG, sep 2003. Draft report.

[3] M. Satyanarayanan, « Pervasive computing : Vision and challenges », *IEEE Personal Communications*, p. 10–17, 2001.

[4] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt et W. D. Meuter, « Ambient-oriented programming », *in OOPSLA'05 : Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (New York, NY, USA), p. 31–40, ACM Press, 2005.

[5] O. Shaer, N. Leland, E. H. Calvillo-Gamez et R. J. K. Jacob, « The TAC paradigm : specifying tangible user interfaces », *Personal Ubiquitous Comput.*, Vol. 8, No. 5, p. 359–369, 2004.

[6] A. Greenfield et C. Fievet, *Everyware : la révolution de l'Ubimédia.* FYP Éditions, jun 2007.

[7] J. Euzenat, J. Pierson et F. Ramparany, « Gestion dynamique de contexte pour l'informatique diffuse », *in 15ème édition du congrès Reconnaissance des Formes et Intelligence Artificielle (RFIA)*, jan 2006.

[8] G. Cliquet et S. Richir, « La réalité virtuelle comme technique d'évaluation des interfaces de l'informatique diffuse », *in colloque CONFERE sur les Sciences de l'Innovation*, (Marrakech), jul 2006.

[9] N. Streitz et P. Nixon, « The disappearing computer - introduction », *Commun. ACM*, Vol. 48, No. 3, p. 32–35, 2005.

[10] A. K. Dey, *Providing architectural support for building context-aware applications.* Thèse de doctorat, College of Computing, Georgia Institute of Technology, dec 2000.

[11] J. O. Kephart, « Research challenges of autonomic computing », *in ICSE '05 : Proceedings of the 27th Intern. conference on Software engineering*, (New York, NY, USA), p. 15–22, ACM Press, 2005.

[12] J. V. Neumann, « First draft of a report on the EDVAC », Rapport Technique Contract No. W-670ORD-492, Moore School of Electrical Engineering, Univ. of Penn., Philadelphia, 1945.

[13] M. Weiser et J. Brown, « Designing calm technology », *PowerGrid Journal*, Vol. 1, p. 1, 1996.

[14] C. Mascolo, L. Capra et W. Emmerich, « Mobile computing middleware », *Advanced Lectures on Networking*, Vol. 2497, p. 20–58, 2002.

[15] P. Persson, « Social ubiquitous computing », *in Position paper to the workshop Building the Ubiquitous Computing User Experience ACM/SIGCHI'01*, (Seattle), 2001.

[16] J. Zimmerman et K. F. Ozenc, « Exploring social relationships between smart homes and their occupants », *in First International Workshop on Social Implications of Ubiquitous Computing*, 2005.

[17] J. C. Tang, « Ubiquitous computing : Individual productivity at the expense of social good ? », *in First International Workshop on Social Implications of Ubiquitous Computing*, 2005.

[18] P. Keyani et J. I. Hong, « Potential effects of ubiquitous computing on civic engagement », *in First International Workshop on Social Implications of Ubiquitous Computing*, 2005.

[19] C. Magerkurth, « Social implications of ubiquitous computing », *in Workshop on social implications of ubiquitous computing in ACM SIGCHI Conference Human Factors in Computing Systems 2005*, 2005.

[20] J. Oprescu, *Service Discovery and Composition in Ambient Networks.* Thèse de doctorat, Institut National Polytechnique Grenoble, France, Décembre 2004.

[21] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier et J. Irwin, « Aspect-oriented programming », *in Proceedings European Conference on Object-Oriented Programming*, Vol. 1241, p. 220–242, Berlin, Heidelberg and New York : Springer-Verlag, 1997.

[22] R. Douence, P. Fradet et M. Südholt, « A framework for the detection and resolution of aspect interactions », *in Generative Programming and Component Engineering : ACM SIGPLAN/SIGSOFT Conference* (C. C. D. Batory et W. T. Editors, éds), Vol. 2487, p. 173–188, October 2002.

[23] S. R. Ponnenkanti et A. Fox, « Application-service interoperation without standardized interfaces », *in IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, 2003.

[24] P. Cointe, J. Noyé, R. Douence, T. Ledoux, Jean-MarcMenaud, G. Muller et M. Südholt, « Programmation post-objets : Des langages d'aspects aux langages de composants », *RSTI L'Objet*, Vol. 10, No. 4, p. 119–143, 2004.

[25] M. D. McIlroy, « Mass produced software components », *in Software Engineering : Report on a Conference Sponsored by the NATO Science Committee* (B. R. E. P. Naur, éd.), 1968.

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

[27] E. Bruneton, T. Coupaye et J. B. Stefani, « Recursive and dynamic software composition with sharing », *in ECOOP Workshop on Component-Oriented Programming*, (Malaga, Spain), jun 2002.

[28] V. Hourdin, S. Lavirotte et J.-Y. Tigli, « Comparaison des systèmes de services pour dispositifs », Rapport Technique ISRN I3S/RR-2006-25-FR, Laboratoire I3S - CNRS - UNSA, aug 2006.

[29] A. Ait Ali Slimane et M. Usman Bhatti, « Utilisation des services et des aspects pour la réutilisabilité du logiciel d'un automate pour l'analyse de plasma », *in 3ème Journée Francophone sur le Développement de Logiciels Par Aspects*, 2007.

[30] L. Berger, *Mise en Œuvre des Interactions en Environnements Distribués, Compilés et Fortement Typés : le Modèle MICADO.* Thèse de doctorat, Université de Nice-Sophia Antipolis - Faculté des sciences et techniques, École doctorale STIC - Informatique, oct 2001.

[31] D. Thomas, « The impact of object oriented languages on design », *Special Issue on object-oriented design*, Vol. 33, No. 9, 1990.

[32] C. Szyperski, *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley. 2nd ed., 2002.

[33] T. Meijler et O. Nierstrasz, « Beyond objects : Components », *in Cooperative Information Systems : Current Trends and Directions* (A. Press, éd.), 1997.

[34] T. Erl, *Service-Oriented Architecture : Concepts, Technology, and Design.* The Prentice Hall, 2004.

[35] S. Krakowiak, « Patrons et canevas pour l'intergiciel », *in Cinquième École d'été sur les Intergiciels et sur la Construction d'Applications Rparties*, (Autrans, Isère, France), 2006.

[36] D. Balek, *Connectors in Software Architecture.* Thèse de doctorat, Charles University, Prague, Czech Republic, 2002.

[37] P. Clements, « A survey of architecture description languages », *in 8th Int. Workshop on Software Specification and Design*, p. 16–25, ACM, 1996.

[38] D. Birngruber, « A software composition language and its implementation », *in Perspectives of System informatics*, 2001.

[39] R. Wuyts et S. Ducasse, « Composition languages for black-box components », *in First OOPSLA Workshop on Language Mechanisms for Programming SoftwareComponents*, 2001.

[40] P. Merle, C. Gransart et J.-M. Geib, « Using and implementing corba objects with corbascript », *in Object Based Parallel and Distributed Computing Workshop*, 1997.

[41] J. K. Ousterhout, « Scripting : Higher-level programming for the 21st century », *Computer*, Vol. 3, No. 31, p. 23–30, 1998.

[42] H. Cervantes, J. M. Favre et F. Duclos, « Describing hierarchical compositions of javabeans with the beanome language », *in Workshop on Software Composition*, 2002.

[43] B. List et B. Korherr, « An evaluation of conceptual business process modelling languages », *in SAC '06 : Proceedings of the 2006 ACM symposium on Applied computing*, (New York, NY, USA), p. 1532–1539, ACM, 2006.

[44] T. Glatard, J. Montagnat, D. Emsellem et D. Lingrand, « A Service-Oriented Architecture enabling dynamic services grouping for optimizing distributed workflows execution », *Future Generation Computer Systems*, Vol. 24, p. 720–730, juil. 2008.

[45] J. H. Andrews, « Process-algebraic foundations of aspect-oriented programming », *in REFLECTION'01 : Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crusscutting Concerns*, p. 187–209, 2001.

[46] H. Tatsuzawa, H. Masuhara et A. Yonezawa, « Aspectual CamL : an aspect-oriented functional language », *in Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, p. 39–50, 2005.

[47] K. Altisen, F. Maraninchi et D. Stauch, « Exploring aspects in the context of reactive systems », *in Workshop on the Foundations of Aspect-Oriented Languages (FOAL)*, 2004.

[48] E. Dijkstra, « Go to statement considered harmful », *Classics in software engineering*, p. 27–33, 1979.

[49] B. J. Cox et A. J. Novobilski, *Object-Oriented Programming : An Evolutionary Approach.* Addison-Wesley (2 Sub edition), 1991.

[50] D. Garlan, D. P. Siewiorek, A. Smailagic et P. Steenkiste, « Aura : Toward distraction-free pervasive computing », *IEEE Pervasive Computing*, 2002.

[51] J. P. Sousa et D. Garlan, « Aura : an architectural framework for user mobility in ubiquitous computing environments », *in Third Working IEEE/IFIP Conference on Software Architecture*, p. 29–43, 2002.

[52] P. J. Braam, « The coda distributed file system », *Linux Journal*, Vol. 50, 1998.

[53] B. Noble, M. Price et M. Satyanarayanan, « A programming interface for application-aware adaptation in mobile computing », *in Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, 1995.

[54] M. Roman et N. Islam, « Dynamically programmable and reconfigurable middleware services », *in Middleware 2004*, Vol. 3231 *in LNCS*, p. 372–396, Springer, 2004.

[55] M. Roman, F. Kon et R. Campbell, « Reflective middleware : From your desk to your hand », *IEEE Distributed Systems Online*, Vol. 2, 2001.

[56] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes et R. Campbell, « Monitoring, security and dynamic configuration with the dynamictao reflective orb », *in Middleware'2000*, (New York, USA), 2000.

[57] M. Anastasopoulos, H. Klus, J. Koch, D. Niebuhr et E. Werkman, « DoAmI - a middleware platform facilitating re-configuration in ubiquitous systems », *in System Support for Ubiquitous Computing Workshop. At the 8th Annual Conference on Ubiquitous Computing (Ubicomp 2006)*, sep 2006.

[58] J. Nehmer, A. Karshmer, M. Becker et R. Lamm, « Living assistance systems - an ambient intelligence approach », *in Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[59] M. Roman, C. K. Hess, R. Cerqueira, A. R. R. H. Campbell et K. Nahrstedt, « Gaia : A middleware infrastructure to enable active spaces », *IEEE Pervasive Computing*, Vol. 1, No. 4, p. 74–83, 2002.

[60] R. Cerqueira, C. Cassino et R. Ierusalimschy, « Dynamic component gluing across different componentware systems », *in International Symposium on Distributed Objects and Applications (DOA'99)*, 1999.

[61] T. Sivaharan, G. S. Blair, A. Friday, M. Wu, H. Duran-Limon, P. Odanka et C. F. Sorensen, « Cooperating sentient vehicles for next generation automobiles », *in ACM MobiSys 2004 workshop on Applications of Mobile Embedded Systems (WAMES 2004)*, juin 2004.

[62] P. Verissimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday et J. Kaiser, « Cortex : Towards supporting autonomous and cooperating sentient entities », *in Proceedings of European Wireless 2002*, 2002.

[63] J. Dowling et V. Cahill, « Self-managed decentralised systems using k-components and collaborative reinforcement learning », *in Proceedings of the Workshop on Self-Managed Systems (WOSS'04)*, 2004.

[64] J. Dowling, V. Cahill et S. Clarke, « Dynamic software evolution and the k-component model », *in Workshop on Software Evolution, OOPSLA*, 2001.

[65] J. Dowling et V. Cahill, « The K-Component architecture meta-model for self-adaptive software », *in Proceedings of Reflection 2001, The Third International Conference on MetalevelArchitectures and Separation of Crosscutting Concerns, Kyoto, Japan* (A. Yonezawa et S. Matsuoka, éds), LNCS 2192, p. 81–88, AITO, Springer-Verlag, sep 2001.

[66] OMG, « Corba 3.0 - omg idl syntax and semantics chapter », Rapport Technique, Object Management Group, 2002.

[67] U. Bischoff et G. Kortuem, « A compiler for the smart home », *in Proceedings of the European Conference on Ambient Intelligence (AmI-07)*, 2007.

[68] U. Bischoff, V. Sundramoorthy et G. Kortuem, « Programming the smart home », *in Proceedings of the Third IET International Conference on Intelligent Environments*, 2007.

[69] M. L. for Computer Science, « Oxygen project ». http ://oxygen.lcs.mit.edu/, jun 2000.

[70] G. Bastide, A. Seriai et M. Oussalah, « Adapting software components by structure fragmentation », *in Proceedings of ACM Symposium on Applied Computing*, 2006.

[71] S. Zachariadis, C. Mascolo et W. Emmerich, « The SATIN component system - a meta model for engineering adaptable mobile systems », *IEEE Trans. on Softw. Eng.*, Vol. 32, p. 910–927, nov. 2006.

[72] Fitech, « xTier(tm) IoC-based system configuration », Rapport Technique, Fitech Laboratories Inc., 2003.

[73] D. Box, *Essential COM*. Addison-Wesley, 1998.

[74] G. Blair, G. Coulson, J. Ueyama, K. Lee et A. Joolia, « OpenCOM v2 : A component model for building systems software », *in IASTED Software Engineering and Applications*, 2004.

[75] T. Bures, *Generating Connectors for Homogeneous and heterogeneous deployment.* Thèse de doctorat, Charles University, Prague, sep 2006.

[76] T. Bures, P. Hnetynka et F. Plasil, « Sofa 2.0 : Balancing advanced features in a hierarchical component model », *in Proceedings of SERA 2006*, 2006.

[77] L. DeMichiel et M. Keith, « JSR 220 : Enterprise javabeans, version 3.0 », Rapport Technique, Sun Microsystems, 2006.

[78] J. Bruhn et G. Wirtz, « mkernel : a manageable kernel for ejb-based systems », *in Autonomics '07 : Proceedings of the 1st international conference on Autonomic computing and communication systems*, (ICST, Brussels, Belgium, Belgium), p. 1–10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[79] I. OMG Object Management Group, « Documents associated with corba, 3.1 ». http ://www.omg.org/spec/CORBA/3.1/, june 2002.

[80] A. Hachichi, G. Thomas, C. Martin, B. Folliot et S. Patarin, « A generic language for dynamic adaptation », *Euro-Par*, Vol. 3648, p. 40–49, 2005.

[81] S. Robert, A. Radermacher, V. Seignole, S. Gérard, V. Watine et F. Terrier, « The CORBA Connector Model », *in SEM'05*, (Lisbon, Portugal), sep 2005.

[82] E. Bruneton, V. Quéma, T. Coupaye, M. Leclerc et J.-B. Stefani, « An open component model and its support in java », *in 7th International Symposium on Component-based Software Engineering*, 2004.

[83] J.-B. Stefani, « A calculus of kells », *in 2nd International Workshop on Foundations of Global Computing*, 2003.

[84] A. Schmitt et J.-B. Stefani, « The kell calculus : A family of higher-order distributed process calculi », *in Global Computing*, p. 146–178, 2004.

[85] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma et J.-B. Stefani, « The fractal component model and its support in java », *Software Practice and Experience, special issue on Experiences with Auto-Adaptive and Reconfigurable Systems*, Vol. 36, p. 11–12, 2006.

[86] L. Seinturier, N. Pessemier, L. Duchien et T. Coupaye, « A component model engineered with components and aspects », *in Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, Vol. 4063 *in Lecture Notes in Computer Science*, p. 139–153, Springer, juin 2006.

[87] J.-P. Fassino, J.-B. Stefani et G. Muller, « Think : A software framework for component-based operating system kernels », *in USENIX 2002 Annual Conference*, may 2002.

[88] C. Escoffier et D. Donsez, « Fracnet ». http ://www-adele.imag.fr/fractnet/, jan 2008.

[89] J. Polakovic, A. E. Özcan et J.-B. Stefani, « Reconfiguration dynamique d'un système à composants », *in RENPAR'16 / CFSE'4 / SympAAA'05 / Journées Composants*, (Le Croisic, France), avril 2005.

[90] J. Aldrich, C. Chambers et D. Notkin, « Archjava : Connecting software architecture to implementation », *in The 24th International Conference on Software Engineering (ICSE)*, (Orlando, USA), 2002.

[91] J. Aldrich, V. Sazawal, C. Chambers et D. Notkin, « Language support for connector abstractions », *in European Conference on Object-Oriented Programming (ECOOP)*, (Darmstadt, Germany), jul 2003.

[92] M. Shaw, R. DeLine et G. Zelesnik, « Abstractions and implementations for architectural connections », *in 3rd International Conference on Configurable Distributed Systems*, May 1996.

[93] A. Alexandrescu et K. Lorincz, « Archjava : An evaluation », Rapport Technique, University of Washington, 2003.

[94] OSCI, « Draft standard systemc language reference manual », Rapport Technique, Open SystemC Initiative, 2005.

[95] M. Thompson et A. D. Pimentel, *A high-level programming paradigm for systemC*, p. 530–539. Springer, 2004.

[96] G. Hamilton, « Java beans specifications v1.01 », Rapport Technique, Sun Microsystems, August 1997.

[97] J. Sasitorn, *Component NextGen : A sound and expressive component framework for Java.* Thèse de doctorat, Rice University, Houston, Texas, USA, 2007.

[98] R. T. Monroe et D. Garlan, « Style-based reuse for software architectures », *in 4th international conference on software reuse*, 1996.

[99] Y. D. Liu et S. F. Smith, « Interaction-based programming with classages », *in OOPSLA'05*, p. 192–209, 2005.

[100] R. Senthil, D. S. Kushwaha et A. K. Misra, « An improved component model for component based software engineering », *ACM SIGSOFT Software Engineering Notes*, Vol. 32, p. 1–9, jul 2007.

[101] U. Zdun et P. Avgeriou, « Modeling architectural patterns unsing architectural primitives », *in OOPSLA'05*, p. 133–146, 2005.

[102] A. Flissi, C. Gransart et P. Merle, « Une infrastructure à composants pour des applications ubiquitaires », *in UbiMob'05* (ACM, éd.), (Grenoble, France), jun 2005.

[103] D. Chefrour et F. André, « Auto-adaptation de composants aceel coopérants », *in 3ème Conférence Francaise sur les Systèmes d'Exploitation (CFSE'3)*, 2003.

[104] H. Cervantes, *Vers un modèle à composants orienté services pour supporter la disponibilié dynamique.* Thèse de doctorat, Université Joseph Fourier, 2004.

[105] L. Cable, *Extensible Runtime Containment and Services Protocol for JavaBeans Version 1.0.* Sun Microsystems, 1998.

[106] D. Marples et P. Kriens, « The open gateway initiative : an introductory overview », *in IEEE Commun. Mag.*, p. 110–114, 2001.

[107] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier et C. Marin, « A dynamic-SOA home control gateway », *IEEE International Conference on Services Computing, SCC'06*, p. 463–470, 2006.

[108] K. Arnold, B. O'Sullivan, R. W. Waldo et A. Wollrath, *The Jini Specification.* Addison-Wesley, 1999.

[109] E. Gamma, R. Helm, R. Johnson et J. Vlissides, *Design Patterns.* Addison-Wesley, 1995.

[110] F. Buschmann, R. Meunier, H. Rohnert, M. Stal et P. Sommerlad, *Pattern Oriented Software Architecture.* Wiley, 1996.

[111] D. Schmidt, M. Stal, H. Rohnert et F. Buschmann, *Pattern-oriented software architecture - Patterns for Concurrent and Networked Objects*, Vol. 2. John Wiley & Sons, Ltd, 2000.

[112] A. Charfi et M. Mezini, « Aspect-oriented web service composition with AO4BPEL », *in ECOWS'04*, 2004.

[113] D. Suvée et W. Vanderperren, « JAsCo : an aspect-oriented approach tailored for component based software development », *in 2nd Int. Conf. on AOSD*, (Boston, USA), 2003.

[114] B. Benatallah, Q. Sheng et M. Dumas, « The self-serv environment for web services composition », *IEEE Internet Computing*, 2003.

[115] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier et L. Martelli, « An UML notation for aspect-oriented software design », *in Workshop on Aspect-Oriented Modeling with UML at AOSD'02*, avril 2002.

[116] B. Verheecke et M. A. Cibrán, « AOP for dynamic configuration and management of web services », *in ICWS'03*, 2003.

[117] A. Charfi et M. Mezini, « AO4BPEL : An Aspect-Oriented Extension to BPEL », *World Wide Web Journal : Recent Advances on Web Services (special issue)*, 2007.

[118] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, JeffreyPalm et W. G. Griswold, « An overview of AspectJ », *Lecture Notes in Computer Science*, Vol. 2072, p. 327–355, 2001.

[119] I. Aracic, V. Gasiunas, M. Mezini et K. Ostermann, « Overview of CaesarJ », *in Transactions on Aspect-Oriented Software Development*, 2006.

[120] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise et M. Südholt, « An expressive aspect language for system applications with arachne », *in AOSD'05 : Proceedings of the 4th international conference on aspect-oriented software development*, (NY), p. 27–38, ACM Press, 2005.

[121] P.-C. David et T. Ledoux, « Une approche par aspects pour le développement de composants fractal adaptatifs », *RSTI - L'Objet*, Vol. 12, No. 2-3, p. 113–132, 2006.

[122] N. Pessemier, L. Seinturier, T. Coupaye et L. Duchien, « A safe aspect-oriented programming support for component-oriented programming », *in Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming (WCOP'06)*, Vol. 2006–11 *in Technical Report*, (Nantes, France), Karlsruhe University, jul 2006.

[123] P.-C. David, *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. Thèse de doctorat, École des Mines de Nantes et Université de Nantes, 2005.

[124] A. van Deursen, P. Klint et J. Visser, « Domain-specific languages : an annotated bibliography », *ACM SIGPLAN Notices*, Vol. 35, p. 26–36, jun 2000.

[125] F. Bounaas, « Using eca rules for object and schema evolution in an object-oriented system », *in Technology of Object-Oriented Languages and Systems (TOOLS 17)*, (USA), 1995.

[126] C. Collet, « Bases de données actives : des systèmes relationnels aux systèmes à objets ». Mémoire Habilitation à Diriger des Recherches (HDR), oct 1996. LSR-IMAG, Grenoble, RR 965-I-LSR 4.

[127] R. E. Filman, T. Elrad, S. Clarke et M. Aksit, *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[128] M. Blay-Fornarino, A. Charfi, D. Emsellem, A.-M. Pinna-Dery et M. Riveill, « Software interactions », *Journal Of Object Technology*, Vol. 3, No. 10, p. 161–180, 2004.

[129] A. Legrand et Y. Robert, *Algorithmique parallèle*. DUNOD, 2003.

[130] R. Douence et M. Südholt, « A model and a tool for event-based aspect-oriented programming (EAOP) », Rapport Technique TR 02/l 1/INFO (french version accepted at LMO'03), Ecole des Mines de Nantes, 2002.

[131] S. Hong Tuan Ha, *Programmation par aspects et tissage de propriétés - Application à l'ordonnancement et à la disponibilité*. Thèse de doctorat, Institut de recherche en informatique et systèmes aléatoires (IRISA), Rennes, 2006.

[132] A. Occello et A.-M. Dery-Pinna, « Safe runtime adaptations of components : a uml metamodel with ocl constraints », *in First International Workshop on Foundations of Unanticipated Software Evolution (FUSE'04)*, (Barcelona, Spain), mar 2004.

[133] C. Bockisch et M. Mezini, « A flexible architecture for pointcut-advice language implementations », *in Workshop VMIL'07*, (Vancouver, British Columbia, Canada), ACM, mar 2007.

[134] N. Ubayashi, A. Sakai et T. Tamai, « An aspect-oriented weaving mechanism based on component and connector architecture », *in ASE* (ACM, éd.), (Atlanta, Georgia, USA), p. 154–163, nov 2007.

[135] M. Mezini et K. Ostermann, « Conquering aspects with Caesar », *in Proc. International Conference on Aspect-Oriented Software Development (AOSD '03)*, (Boston, USA), 2003.

[136] D. Suvée, W. Vanderperren et V. Jonckers, « FuseJ : An architectural description language for unifying aspects and components », *in Workshop Software-engineering Properties of Languages and Aspect Technologies*, 2005.

[137] P. L. Tarr, H. Ossher, W. Harrison et S. M. Sutton, « N degrees of separation : Multi-dimensionnal separation of concerns », *in 21st Int'l Conf. Software Eng. (ICSE)* (A. Press, éd.), p. 107–119, 1999.

[138] B. Lagaisse et W. Joosen, « Component-based open middleware supporting aspect-oriented software composition », *in CBSE*, p. 139–154, 2005.

[139] H. Rajan, R. Dyer, Y. Hanna et H. Narayanappa, « Preserving separation of concerns through compilation », Rapport Technique, Dept. of Computer Sc., Iowa State University, 2006.

[140] E. Tanter et R. Toledo, « A versatile kernel for distributed AOP », *in International Federation for Information Processing (DAIS 2006)* (F. Eliassen et A. M. (Eds.), éds), No. 4025, p. 316–331, LNCS, 2006.

[141] C. V. Lopes, *A Language Framework for Distributed Programming.* Thèse de doctorat, College of Computer Science, Northeastern University, 1997.

[142] Lieberherr, « Lieberherr », *Lieberherr*, 2004.

[143] R. Lämmel, E. Visser et J. Visser, « Strategic programming meets adaptive programming », *in AOSD'2003*, (Boston, USA), p. 168–177, 2003.

[144] A. Moreira, J. ao Araújo et A. Rashid, « A concern-oriented requirements engineering model », *CAiSE 2005*, Vol. 3520, p. 293–308, 2005.

[145] F. Tessier, M. Badri et L. Badri, « A model-based detection of conflits between crosscutting concerns : Towards a formal approach », *in International Workshop on Aspect-Oriented Software Development (WAOSD 2004)* (H. M. M. Huang et e. J. Zhao, éds), sep 2004.

[146] L. M. J. Bergmans, « Towards detection of semantic conflicts between crosscutting concerns », *in ECOOP : AAOS'03. The first workshop on analysis of aspect-oriented software*, (Darmstadt, Germany, July), p. 1–6, jul 2003.

[147] A. Zambrano, T. Vera et S. Gordillo, « Solving aspectual semantic conflicts in resource-aware systems », *in RAM-SE'06 - ECOOP'06 Workshop on reflection, AOP, and meta-data for software evolution*, No. 79–88, (Nantes, France), jul 2006.

[148] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten et J.-C. Burgelman, « Scenarios for ambient intelligence in 2010 », Rapport Technique, ISTAG advisory group report, Seville, 2001.

[149] E. Jul, H. Levy, N. Hutchinson et A. Black, « Fine-grained mobility in the emerald system », *ACM Transactions on Computer Systems*, Vol. 6, p. 109–133, 1988.

[150] L. Cardelli, « A language with distributed scope », *in Conference Record of POPL'95 : 22nd annual ACM SIGPLAN-SIGACT Symposium on principles of programming languages*, p. 286–297, 1995.

[151] R. Tolksdorf et K. Knubben, « Programming distributed systems with the delegation-based object-oriented language dself », *in SAC'02 : Proceedings of the 2002 ACM symposium on applied computing*, p. 927–931, 2002.

[152] J. Dedecker, *Ambient-oriented programming.* Thèse de doctorat, Vrije Universiteit Brussel, 2006.

[153] R. Gupta et R. K. Shyamasundar, « Reactive framework for resource aware distributed computing », *in Asian Computing Science Conference (ASIAN)* (A. Springer, Berlin, éd.), LNCS, (Chiang Mai, Thailand), p. 452–467, 2004.

[154] D. Harel et A. Pnueli, « On the development of reactive systems », *in Logic and Models of Concurrent Systems* (Springer-Verlag, éd.), Vol. 13 *in NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, (New York), p. 477–498, 1985.

[155] L. Capra, W. Emmerich et C. Mascolo, « Reflective middleware solutions for context-aware applications », *in Proceedings of Reflection 2001, LNCS, Springer Verlag*, 2001.

[156] C. Mascolo, S. Hailes, L. Lymberopoulos, G. P. Picco, P. Costa, G. Blair, P. Okanda, T. Sivaharan, W. Fritsche, M. Karl, M. A. Rnai, K. Fodor et A. Boulis, « Survey of middleware for networked embedded systems », Rapport Technique D5.1, 2005.

[157] E. Niemela et J. Latvakoski, « Survey of requirements and solutions for ubiquitous software », *in MUM'04 : Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, (NY), p. 71–78, ACM, 2004.

[158] D. Thevenin et J. Coutaz, « Adaptation des interactions humain-machine : Taxonomies et architecture logicielle », *in IHM02*, p. 207 – 210, 2002.

[159] M. T. Ibrahim, R. J. Anthony, T. Eymann, A. Taleb-Bendiab et L. Gruenwald, « Exploring adaptation & self-adaptation in autonomic computing systems », *in 17th Int. Conf. on Database and Expert Systems Applications, 2006. DEXA '06.*, p. 129–138, 2006.

[160] D. Browne, P. Totterdell et M. Norman, « Adaptive user interfaces », *Computer and People Series*, 1990.

[161] H. Dieterich, U. Malinowski, T. Kühme et M. Schneider-Hufschmidt, « State of the art in adaptive user interface », *in Adaptive User Interfaces, Principles and Practice*, Human Factors in Information Technology, p. 13–48, 1993.

[162] C. Stephanidis, A. Paramythis, M. Sfyrakis et A. Savidis, « A case study in unified user interface development : The avanti web browser », *in User Interface for All - concepts, methods and tools*, p. 525–568, 2001.

[163] P. Brusilovsky, « Adaptive hypermedia », *User Modeling and User-Adapted Interaction*, p. 87–110, 2001.

[164] C. Canal, P. Poizat et G. Salaün, « Adaptation de composants logiciels. une approche automatisée basée sur des expressions régulières de vecteurs de synchronisation », *in Actes de la Première Conférence Francophone sur les Architectures Logicielles (CAL)*, 2006.

[165] R. Schulte, « Clarifying the terms 'event-driven' and 'service-oriented architecture' », *Gartner*, 2008.

[166] O. Spinczyk, A. Gal et W. Schroeder-Preikschat, « A lightweight component architecture for efficient information fusion », *in Proceedings of the International Workshop on Information Fusion 2001 (DBFusion 2001)*, (Magdeburd, Germany), apr 2001.

[167] S.-W. Lee, Y.-D. You et H. Choi, « Design and implementation c based lightweight component framework », *in IEEE Annual seminar, TECON* (IEEE, éd.), 2006.

[168] OASIS, « Oasis reference model for service oriented architecture 1.0 ». http ://www.oasis-open.org/committees/, aug 2006.

[169] M. Jeronimo et J. Weast, *UPnP Design by Example*. Intel Press, 2003.

[170] J. Schlimmer et J. Thelin, « Devices profile for web services ». schemas.xmlsoap.org/ws/2006/02/devprof, feb 2006.

[171] N. Bussière, D. Cheung-Foo-Wo, V. Hourdin, S. Lavirotte, M. Riveill et J.-Y. Tigli, « Optimized contextual discovery of web services for devices », *in IEEE Int. Workshop on Context Modeling and Management for Smart Environments*, 2007.

[172] V. Hourdin, S. Lavirotte et J.-Y. Tigli, « Service upnp pour dispositifs autonomes », *Technique de l'ingï¿½nieur*, Vol. H5002, feb 2007.

[173] J. Cohen et S. Aggarwal, « General Event Notification Architecture Base (GENA) ». http ://quimby.gnus.org/internet-drafts/draft-cohen-gena-p-base-01.txt, juil. 1998.

[174] OASIS, « Oasis uddi specification ». http ://www.uddi.org/specification.html, jul 2002.

[175] W3C, « Semantic web services ontology (swso) ». http ://www.daml.org/services/swsf/1.0/swso/, may 2005.

[176] A. Keller et H. Ludwig, « The wsla framework : Specifying and monitoring service level agreements for web services », *Journal of Network and Systems Management*, 2003.

[177] M. Paolucci, N. Srinivasan, K. P. Sycara et T. Nishimura, « Towards a Semantic Chroreography of Web Services : from WSDL to DAML-S », *in Proceedings of the International Conference on Web Services (ICWS'03)*, (Las Vegas, Nevada), p. 22–26, jun 2003.

[178] C. Prehofer et C. Bettsteller, « Self-organization in communication networks : principles and design paradigms », *Communications Magazins, IEEE*, Vol. 43, p. 78–85, jan 2004.

[179] J.-Y. Tigli, D. Cheung-Foo-Wo, S. Lavirotte et M. Riveill, « Adaptation au contexte par tissage d'aspects d'assemblage de composantsdéclenchés par des conditions contextuelles », *RTSI Série ISI*, Vol. 5, No. 11, p. 89–114, 2006.

[180] J. Kuck et M. Gnasa, « Context-Sensitive Service Discovery Meets Information Retrieval », *in Fifth IEEE Int. Conference on Pervasive Computing and Communications Workshops (PERCOMW)*, p. 601–605, IEEE Computer Society, 2007.

[181] M. Broy, I. H. Krüger et M. Meisinger, « A formal model of services », *ACM Transactions Software Engineering and Methodology*, Vol. 16, feb 2007.

[182] R. Khalaf et N. Mukhi, « Service-Oriented Composition in BPEL4WS ». WWW (Alternate Paper Tracks), 2003.

[183] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy et M.-C. Shan, « Adaptive and dynamic service composition in eFlow », *in Conference on Advanced Information Systems Engineering*, p. 13–31, 2000.

[184] W.-T. Balke et M. Wagner, « Cooperative Discovery for User-Centered Web Service Provisioning », *in Proceedings of the International Conference on Web Services (ICWS'03*, (Las Vegas, Nevada), p. 191–197, jun 2003.

[185] OSOA, « Service component architecture specifications ». http ://www.osoa.org/display/Main/Service+Component+Architecture+ Specifications, aug 2008.

[186] D. Cheung-Foo-Wo, J.-Y. Tigli, S. Lavirotte et M. Riveill, « Wcomp : a multi-design approach for prototyping applications using heterogeneousresources », *in 17th IEEE International Workshop on Rapid System Prototyping*, (Chania, Crete), jun 2006.

[187] D. Cheung-Foo-Wo, G. Joulie, F. Grillon, J. Fuchet et J.-Y. Tigli, « WComp : Rapid Application Development Toolkit for Wearable Computer Based on Java », *in IEEE International Conference on Systems, Man and Cybernetics*, Vol. 5, (Washington DC, USA), p. 4198–4203, oct 2003.

[188] M. N. Bouraquadi-Saâdani et T. Ledoux, « Le point sur la programmation par aspects », *Technique et science informatiques*, Vol. 20, No. 4, p. 289–512, 2001.

[189] N. Pessemier, L. Seinturier, T. Coupaye et L. Duchien., « A model for developing component-based and aspect-oriented systems », *in 5th International Symposium on Software Composition (SC'06)*, Vol. 4089 *in Lecture Notes in Computer Science*, (Vienna, Austria), p. 259–273, Springer-Verlag, March 2006.

[190] N. Pessemier, *Unification des approches par aspects et à composants.* Thèse de doctorat, Université Lille 1, juin 2007.

[191] F. Grillon, « Behavioral architecture », master de dea rsd, Université de Nice - Sophia Antipolis, 2003. 36 pages.

[192] D. Cheung-Foo-Wo, M. Blay-Fornarino, J.-Y. Tigli, S. Lavirotte et M. Riveill, « Adaptation dynamique d'assemblages de dispositifs dirigée par des modèles », *in 2ème journées sur l'Ingénieurie Dirigée par les Modèles (IDM)*, 2006.

[193] D. Cheung-Foo-Wo, M. Blay-Fornarino, J.-Y. Tigli, A.-M. Déry, D. Emsellem et M. Riveill, « Langage d'aspect pour la composition dynamique de composants embarqués », *RTSI - L'Objet*, Vol. 12, No. 2-3, p. 89–111, 2006.

[194] R. Jagadeesan, A. S. A. Jeffrey et J. Riely, « A calculus of untyped aspect-oriented programs », *in Proceedings of the European Confonference on Object-Oriented Programming* (Springer-Verlag, éd.), p. 415–427, 2003.

[195] M. S. Paterson et M. N. Wegman, « Linear unification », *Journal of Computer and System sciences*, Vol. 16, p. 158–167, 1978.

[196] A. Martelli et U. Montanari, « An efficient unification algorithm », *ACM Transactions on Programming Languages and Systems*, Vol. 4, p. 258–282, apr 1982.

[197] J. A. Robinson, « Computational logic : the unification computation », *Machine Intelligence*, Vol. 6, 1971.

[198] V. Issarny, M. Caporuscio et N. Georgantas, « A perspective on the future of middleware-based software engineering », *in FOSE '07 : 2007 Future of Software Engineering*, (Washington, DC, USA), p. 244–258, IEEE Computer Society, 2007.

[199] J. Hanneman, R. Chitchyan et A. Rashid, « Analysis of aspect-oriented software development », *in AOSD 2003*, (Darmstadt, Allemagne), 2003.

[200] T. Batista, C. Chavez, R. Garcia, U. Kulesza et C. Lucena, « Aspectual Connectors : Supporting the Seamless Integration of Aspects and ADLs », *in SBES'06*, 2006.

[201] C. Chavez, R. Garcia et C. Lucena, « Some insights on the use of aspectj and hyper/j », *in Tutorial and Workshop on AspectOriented Programming and Separation of Concerns*, p. 23–24, 2001.

[202] S. Weibel, J.-Y. Tigli, S. Lavirotte, D. Cheung-Foo-Wo, V. Hourdin et M. Riveill, « SharpWComp », *in IDDN.FR.001.090016.000.S.C.2007.000.10600*, Septembre 2006.

[203] C. Szyperski et C. Pfister, « Workshop on component-oriented programming, summary. », *in Special Issues in Object-Oriented Programming - ECOOP 96*, 1997.

ANNEXES

Table des matières

Annexe A

Aspect d'assemblage : les détails

Sommaire

Cet annexe fournit une description détaillée de notre modèle, que nous appelons modèle d'aspect d'assemblage. Il présente tout d'abord la syntaxe abstraite des aspects d'assemblage. Nous décrivons notre modèle indépendamment de toute implémentation du modèle à composant sous-jacent.

A.1 Greffon ISL4WComp

La syntaxe abstraite décrite s'appuie sur la grammaire BNF. Par souci de simplicité, la syntaxe est décomposée en plusieurs sous-ensembles, chacun faisant référence à l'autre.

A.1.1 Grammaire

Un greffon en ISL4WComp se compose de six opérateurs (délégué, conditionnel, call, nop, séquence et concurrence). Voici la grammaire des greffons :

```
all:
    | all ASPECT ID '(' var_list ')' ':' data
data:
        | data ID ':' classe ';'
        | data ID ':' '\'' classe '\'' ';'
        | data ID '.' '^' ID ARROW '(' target ')'
        | data ID '.' ID ARROW '(' target ')'
classe:
        ID
        | classe '.' ID
target:
        '(' target ')'
        | DELEGATE '(' target ')'
        | IF '(' method_target ')' '{' target '}' ELSE '{' target '}'
        | IF '(' method_target ')'     target
        | CALL
        | NOP
        | target ';' target
        | target '+' target
        | method_target
method_target:
     ID '.' ID
var_list:
        | var_list1
var_list1:
          var_list1 ',' ID
        | ID
```

La grammaire est beaucoup plus simple que le modèle présenté dans la fusion, car elle réalise la jonction entre les composants logiciels disponibles et le modèle de fusion. Par exemple, les composants logiciels de séquence sont binaires et n'ont que deux sorties, alors que dans le modèle, une séquence peut avoir autant de fils que possible.
C'est pour cela que nous avons mis en place des transformations entre la grammaire du greffon et le modèle de fusion (section A.1.2.2).

A.1.2 Du greffon aux règles logiques

Pour pouvoir travailler sur les spécifications des greffons, nous les intégrons dans un modèle où :
- les éléments (opérateurs) sont ordonnés,
- pour tout schéma, il existe une représentation simple unique (forme canonique),

– il existe une transformation entre le modèle et les spécifications.

Nous voyons dans cette partie la structure des éléments de ce modèle et les différentes transformations à partir des spécifications.

A.1.2.1 Structure d'une règle

Soit $\vec{a} = [a_1, a_2, a_3]$, $\vec{c} = [t, f]$ et $\vec{\vec{d}} = [[a_1], [b_1, b_2]]$. Chaque opérateur a une priorité de fusion, c'est-à-dire qu'il va dominer les opérateurs de priorité plus faibles.

Priorité	Notation mathématique
1	$\text{par}_L(\vec{a})$
2	$if\left(\vec{c}, \vec{\vec{d}}\right)$
3	$del(p)$
4	$call$
5	p
6	nop

Tableau A.1 – Tableau des priorités et de la notation mathématique des opérateurs

Un comportement se construit en imbriquant les différents opérateurs. On note une liste de comportements entre crochets. Voici un exemple de comportement :

$$seq\left([if([c], [[m_1]]), seq([m_2, m_3])]\right)$$

Cette forme est équivalente à la forme canonique (voir l'algorithme à la section C.2.3) suivante :

$$\left[par_0\left([if([c], [[m_1]])]\right),\ par_1\left([if([t], [[m_2]])]\right),\ par_2\left([if([t], [[m_3]])]\right)\right]$$

La notation t représente le booléen vrai.

A.1.2.2 De la spécification au modèle

Nous avons deux transformations à écrire. Nous voyons dans ce paragraphe la transformation des spécifications données par le greffon vers le modèle de calcul pour la composition et la fusion. Cette transformation s'accompagne de la transformation de la séquence dont l'algorithme est donné dans la section suivante (section A.1.2.3).

Il n'y a pas de difficulté à faire cette transformation sauf peut-être pour l'opérateur conditionnel à partir duquel il faut penser à rajouter la négation de la condition c. Le tableau A.2 regroupe les transformations syntaxiques. Une implémentation en Prolog est donné à la page 189.

Spécifications	Modèle
$par(\vec{a}, \vec{b})$	$par_0(\vec{a}, \vec{b})$
$seq(\vec{a}, \vec{b})$	$seq(\vec{a}, \vec{b})$
$if(c, \vec{a}, \vec{b})$	$if([c, \vec{\neg c}], [\vec{a}, \vec{b}])$
$del(m)$	$leaf(del(m))$
nop	$leaf(nop)$
$call$	$leaf(call)$
p	$leaf(p)$

Tableau A.2 – Transformations des spécifications

A.1.2.3 Transformation de la séquence

Il est difficile de fusionner un code d'apparence linéaire, mais qui suggère plusieurs actions dans le temps. C'est ce que le langage ISL déclare implicitement. La solution consistait à travailler sur la notion de pivot pour linéariser les comportements.
Nous ne nous appuyons pas sur des pivots dans notre approche. Nous utilisons une toute autre technique s'appuyant sur plusieurs niveaux de concurrence. Attention, c'est une concurrence de schémas dont nous parlons, et non pas la concurrence des systèmes d'exploitation. Ce qui nous intéresse, c'est comment calculer la superposition de plusieurs spécifications. Le fait d'intégrer plusieurs niveaux de concurrence simplifie la fusion, puisque pour chaque niveau, on s'attend à n'avoir qu'une opération à exécuter. Lorsque cette opération est exécutée, alors on passe au niveau suivant (voir figure A.1).

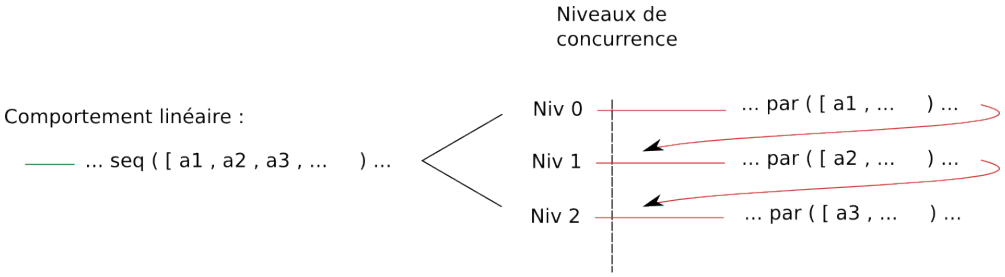


FIGURE A.1 – Niveaux de concurrence

L'algorithme d'élimination des séquences peut être décrite récursivement. Prenons $\vec{c}$, un comportement décrit en ISL4WComp, l, le niveau de concurrence est au départ fixé à 0. Enfin, appelons $P_l(\vec{c})$ l'algorithme récursif d'élimination des séquence pour un niveau l donné :

1. Si c_1 est une séquence $seq(\vec{d})$, alors éliminier récursivement les séquences dans le reste du comportement noté $e = P_l(\vec{d})$ et construire un opérateur de concurrence de niveau l à l'aide de e : $par_l(e)$. Le reste du vecteur $\vec{c}$ doit alors être exécuté à un niveau supérieur.
2. Si c_1 est un opérateur de concurrence, on y recherche récursivement la présence de séquence afin de les éliminer.
3. Si c_1 est un opérateur conditionnel, idem.
4. Si c_1 est un comportement final, la récursion se termine.

Une implémentation de cet algorithme en Prolog est donné à la page 191.

A.1.2.4 Du modèle vers la spécification

Une implémentation en Prolog est donnée à la page 190.

A.2 Fusion ISL4WComp

A.2.1 Introduction

L'approche d'origine ISL consiste à fusionner les opérateurs entre eux, indépendamment de leur sémantique. Cela a pour conséquence par exemple de vouloir définir une sémantique de la fusion entre l'envoi d'un message et une séquence, ce qui n'est pas intuitif. Notre approche consiste à établir une forme canonique des interactions en classant les opérateurs et à définir les règles de fusion uniquement pour la fusion ayant un sens intuitif comme la fusion de deux comportements conditionnels, de deux messages, etc.

On représente une liste de comportement par un vecteur $\vec{\alpha}$. Chaque coordonée de ce vecteur représente un comportement noté α_i. On a :

$$\vec{\alpha} = [\alpha_1, \ldots, \alpha_n]$$

Un vecteur n'ayant qu'un seul élément β est noté entre crochets $[\vec{\beta}]$. Enfin, on note $\vec{\beta} \setminus \beta_1$ l'ensemble $\vec{\beta}$ privé de son premier élément β_1.

Nous classons les opérateurs selon deux catégories : les opérateurs n-aires et les opérateurs finals.

A.2.2 Opérateurs n-aires

Un opérateur n-aire a un nombre fini n de paramètres. Nous avons implémenté la concurrence et le conditionnel. La séquence est un cas particulier de la concurrence à plusieurs niveaux. Les opérateurs n-aires sont définis récursivement.

A.2.2.1 Concurrence

L'opérateur de concurrence est un opérateur binaire commutatif et associatif. Il se note $\text{par}(\vec{\alpha})$ où le vecteur $\vec{\alpha}$ représente la liste des comportements $\alpha_1 \ldots \alpha_n$. L'opérateur de concurrence est paramétré par un niveau L. Chaque niveau est indépendant, par conséquent, des comportements concurrents ne peuvent fusionner qu'à un niveau L donné. Cette notion de niveau permet de représenter simplement la séquence dans notre modèle. Ainsi, un niveau peut être exécuté après l'exécution d'une opération. La notion de concurrence est différent de celle de Berger dans le sens où elle n'est présente qu'au niveau du modèle et pas de l'implémentation.

L'équation A.1 représente la fusion d'un comportement $\text{par}_L([\vec{a}])$ d'un niveau L avec un ensemble de comportements en concurrence $\text{par}_L(\vec{\beta})$. Le premier comportement s'intègre dans le second. Cela représente le cas idempotent.

$$\frac{\text{si } \vec{\beta} = [\beta_1, \ldots, \beta_n] \text{ et } a = \beta_1}{\text{par}_L([\vec{a}]) \oplus \text{par}_L(\vec{\beta}) = \text{par}_L(\vec{\beta})} \tag{A.1}$$

L'équation A.2 représente la fusion récursive du comportement précédent. On cherche à unifier le comportement $\text{par}_L([\vec{a}])$ avec le reste de l'ensemble $\text{par}_L(\vec{\beta})$ de manière récursive. Deux comportements qui ne peuvent pas s'unifier sont alors regroupés dans un même ensemble.

$$\frac{\text{si } a \neq \beta_1}{\text{par}_L([\vec{a}]) \oplus \text{par}_L(\vec{\beta}) = \text{par}_L([\vec{\beta_1}]) \cup \left(\text{par}_L([\vec{a}]) \oplus \text{par}_L(\vec{\beta} \setminus \beta_1)\right)} \tag{A.2}$$

Enfin, l'équation A.3 permet de mettre fin à la récurrence précédente en utilisant le vecteur nul *vvide* qui représente l'ensemble vide de comportement. N'importe quel comportement concurrent composé avec le comportement concurrent nul donne lui-même.

$$\text{par}_L(\vec{\beta}) \oplus \text{par}_{L'}(\vec{\emptyset}) = \text{par}_L(\vec{\beta}) \tag{A.3}$$

Pour conclure, nous présentons l'équation A.4 concernant le cas particulier des niveaux de concurrence. Les niveaux sont simplements juxtaposés dans l'ensemble des comportements concurrents.

$$\frac{\text{soient } K \text{ et } L \text{ des niveaux différents}}{\text{par}_K(\vec{\alpha}) \oplus \text{par}_L(\vec{\beta}) = \text{par}_K(\vec{\alpha}) \cup \text{par}_L(\vec{\beta})} \tag{A.4}$$

Ces quatres équations ont été implémentées en Prolog. L'implémentation est donnée à la page 192.

A.2.2.2 Condition

Nous définissons tout d'abord, l'opération d'union notée $\cup_{if}$ pour deux opérateurs *if*. Ainsi, l'union de deux comportements conditionnels est égale à l'union des deux conditions $\vec{a}$ et $\vec{b}$ et des deux comportements $\vec{\alpha}$ et $\vec{\beta}$.

$$if\ (\vec{a}, \vec{\alpha}) \cup_{if} if\ \left(\vec{b}, \vec{\beta}\right) = if\ \left(\vec{a} \cup \vec{b}, \vec{\alpha} \cup \vec{\beta}\right)$$

L'équation A.5 exprime la fusion idempotente de l'opérateur *if*.

$$\frac{\text{si } b = a_1}{if\ \left([\vec{b}], [\vec{\beta}]\right) \oplus if\ (\vec{a}, \vec{\alpha}) = if\ \left([\vec{b}], [\vec{\beta}] \cup [\vec{\alpha_1}]\right) \oplus if\ (\vec{a} \setminus a_1, \vec{\alpha} \setminus \alpha_1)} \tag{A.5}$$

L'équation A.6 exprime également une recursion sur la fusion de l'opérateur *if*.

$$\frac{\text{si } b \neq a_1}{if\ \left([\vec{b}], [\vec{\beta}]\right) \oplus if\ (\vec{a}, \vec{\alpha}) = if\ \left([\vec{a_1}], [\vec{\alpha_1}]\right) \cup_{if} \left(if\ \left([\vec{b}], [\vec{\beta}]\right) \oplus if\ (\vec{a} \setminus a_1, \vec{\alpha} \setminus \alpha_1)\right)} \tag{A.6}$$

L'équation A.7 représente la condition d'arrêt de la fusion récursive.

$$if\ (\vec{a}, \vec{\alpha}) \oplus if\ \left(\vec{\emptyset}, \vec{\emptyset}\right) = if\ (\vec{a}, \vec{\alpha}) \tag{A.7}$$

Ces trois équations ont été implémentées en Prolog. L'implémentation est donnée à la page 193.

> *La fusion des comportements concurrents consiste à ne pas réaliser de redondances et propose plusieurs niveaux de concurrence indépendants permettant de représenter la notion de séquence. La fusion des comportements conditionnels consiste à fusionner les conditions une à une. C'est une différence fondamentale par rapport à l'approche ISL d'origine.*

A.2.3 Opérateurs finals

A.2.3.1 Introduction

Les opérateurs finals sont la délégation, l'appel d'une opération, le nop et le call.

Selon l'équation A.8, la fusion de deux entités qui s'unifient donne la même entité. C'est la propriété d'idempotence de la fusion.

$$\frac{\text{si } \alpha = \beta}{leaf\ (\alpha) \oplus leaf\ (\beta) = leaf\ (\alpha) = leaf\ (\beta)} \tag{A.8}$$

On garantit la commutativité avec la règle suivante :

$$leaf\ (\alpha) \oplus leaf\ (\beta) = leaf\ (\beta) \oplus leaf\ (\alpha) \tag{A.9}$$

A.2.3.2 Délégation

La délégation signifie que l'on remplace l'appel du *message notifiant* par le comportement induit par la délégation. Elle a également la priorité sur les opérations que d'autres spécifications peuvent faire intervenir.

Comme décrit dans l'équation A.10, deux délégations ne fusionnent pas, car il est impossible de synthétiser un choix à partir des spécifications de l'adaptation.

$$\frac{\text{si } \alpha \neq \beta}{leaf\ (del(\alpha)) \oplus leaf\ (del(\beta)) = \text{erreur !}} \tag{A.10}$$

L'équation A.11 définit la priorité qu'a l'opérateur de délégation sur tous les autres.

$$\frac{\forall \beta}{leaf\ (del(\alpha)) \oplus leaf\ (\beta) = leaf\ (del(\alpha))} \tag{A.11}$$

A.2.3.3 Call et nop

La fusion de l'événement initiateur de l'appel de l'opération est très simple. Il laisse sa place à tout autre opérateur comme indiqué dans l'équation A.12.

$$\frac{\forall \beta}{leaf\ (call) \oplus leaf\ (\beta) = leaf\ (\beta)} \tag{A.12}$$

L'opérateur nop a cette même sémantique sauf qu'il ne représente pas l'événement initiateur (équation A.12).

$$\frac{\forall \beta}{leaf\ (nop) \oplus leaf\ (\beta) = leaf\ (\beta)} \tag{A.13}$$

A.2.3.4 Appel d'une opération

Deux opérations différentes ne peuvent pas fusionner (équation A.14).

$$\frac{\text{si } \alpha \neq \beta}{leaf\ (op(\alpha)) \oplus leaf\ (op(\beta)) = \text{erreur1 et } error1++} \tag{A.14}$$

Ces équations ont été implémentées en Prolog. L'implémentation est donnée à la page 194.

A.2.3.5 Erreurs graves et moins graves

Dans notre système, on considère deux types d'erreur de fusion : les erreurs graves $error_0$ et les erreurs moins graves $error_1$, c'est-à-dire des ambiguités l'on peut encore résoudre a posteriori. Seule une délégation peut résoudre une erreur de type 1.

$$\frac{\text{si } \alpha \neq \beta}{leaf\ (error1) \oplus leaf\ (del(\beta)) = leaf\ (del(\beta))\ \text{ et } error1--} \tag{A.15}$$

> *Nous avons exprimé uniquement la sémantique de la fusion ayant un sens intuitif. Aussi, l'ajout de nouveaux opérateurs est simplifié, car il ne s'agit que de rajouter la sémantique de la fusion avec lui-même ou les autres feuilles dans le cas d'un opérateur final et d'implémenter la fonction canonique.*

A.2.4 Forme canonique

La forme canonique a principalement deux emplois spécifiques. D'une part, il qualifie des formes d'expressions algébriques censément plus simples et en tout cas auxquelles se ramènent toutes les expressions. Cela permet de les distinguer et de les classifier.

L'existence d'une forme canonique, et d'une méthode générale pour mettre sous cette forme tous les éléments d'un ensemble donné est une propriété essentielle, et même nécessaire, à la calculabilité sur l'ensemble des opérateurs.

Classement	Opérateur
1	Concurrence
2	Conditionnel
3	Opérateurs finals

Tableau A.3 – Classification des opérateurs

A.2.4.1 Tri des opérateurs

D'abord, nous avons formulé un classement des différents opérateurs (tableau A.5).

Nous avons définit une opération permettant de classer les opérateurs de concurrence en première position. Notre langage étant simple, l'opérateur conditionnel est alors automatiquement classé à son tour puisque les opérateurs finals ne peuvent être désignés qu'après les opérateurs n-aires.

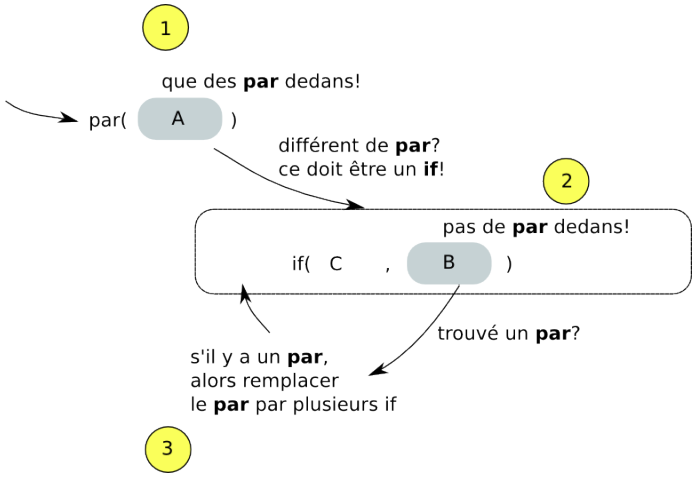


FIGURE A.2 – Algorithme de classification

L'algorithme de classement des opérateurs se fait en trois étapes (figure A.2). Ces trois étapes sont implémentés à l'aide de trois prédicats à la page 196. Il a pour but de faire en sorte que les opérateurs de concurrence soit en premier, puis, que les opérateurs conditionnels soit en second et enfin, le reste des opérateurs sont laissés en place.

La première étape de cet algorithme consiste à vérifier que la liste A des comportements concurrents contient bien des comportements concurrents. Lorsqu'un comportement d'une nature différente est détecté, on passe alors à la deuxième étape.

La deuxième étape consiste à vérifier d'une part que cet opérateur est bien un opérateur conditionnel. Si ce n'est ni un opérateur conditionnel, ni un opérateur concurrent alors il s'agit d'un opérateur final, on rajoute donc un opérateur conditionnel dont la condition est toujours vraie. On vérifie également que la liste B des comportements finals ne contient pas d'opérateurs concurrents ; si tel est le cas, alors on passe à l'étape 3.

Enfin, la troisième étape construit artificiellement des opérateurs conditionnels supplémentaires pour remplacer ces derniers, ce qui simule le fait d'inverser les deux types d'opérateur n-aire. Elle vérifie également à toutes les profondeurs qu'il ne reste pas d'opérateur concurrent après un opérateur conditionnel. Si c'est le cas, elle applique la technique d'inversion.

A.2.4.2 Application des équations de fusion

Une fois que les opérateurs sont triés, nous appliquons les équations de fusion. Ainsi, chaque opérateur fusionne uniquement avec les opérateurs ayant une sémantique similaire. Soit $\vec{a}$ un ensemble de comportements triés de différents niveaux de concurrence L_i, l'algorithme est le suivant :

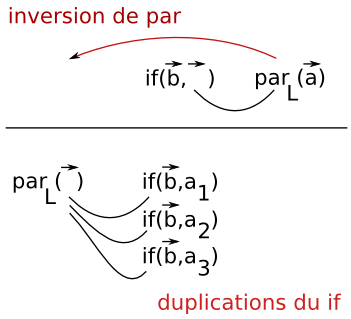


FIGURE A.3 – Inversion par / if

1. Appliquer les équations A.1, A.2, A.3 et A.4 au comportement $\vec{a}$ pour fusionner les comportements de concurrence, on obtient alors une liste de comportements conditionnels notée $\vec{b}$;
2. Appliquer les équations A.5, A.6 et A.7 à $\vec{b}$ pour fusionner les comportements conditionnels, on obtient alors une liste de comportements finals $\vec{c}$ et de conditions associées $\vec{k}$ pour chacune des conditions ;
3. Enfin, appliquer les équations de la section A.2.3 sur c pour résoudre les conflits des comportements finals. On obtient soit une erreur, soit une liste d dont les conflits sont résolus.

Pour finir, on peut construire le comportement sous sa forme canonique à l'aide de $\vec{b}$, c et d.

$$[\text{par}_{L_i}(if\ \left(\vec{k}, \vec{d}\right)), \dots]_i \text{ où } i \text{ dénote le niveau de concurrence}$$

Nous donnons un exemple d'implémentation de cet algorithme à la page C.3.5.

A.2.5 L'opération de fusion n-aire

L'opération de fusion n-aire prend en entrée une liste de spécifications. Elle consiste d'abord à mettre à zéro le compteur d'erreurs non résolues et appelle récursivement la fonction de fusion binaire tel que :

$$\bigoplus_{i=1}^{n} a_i = ((\dots(a_1 \oplus a_2) \oplus a_3) \oplus \dots \oplus a_n)$$

La fusion est réussie si le compteur d'erreurs non résolues est toujours à zéro et que la fusion binaire n'a pas levée d'erreur fatale (fusion de deux délégation différentes).

A.3 Greffon BSL

A.3.1 Grammaire

Un greffon en BSL se compose de quatre opérateurs (l'arbitrage, le découpage, la diffusion d'événements et nop). Voici la grammaire des greffons :

```
all:
    | all ASPECT ID '(' var_list ')' ':' data
data:
        | data ID ':' classe ';'
        | data ID ':' '\'' classe '\'' ';'
        | data '(' target ')' BACK_ARROW ID '.' '^' ID
```

```
        | data '(' target ')' BACK_ARROW ID '.' ID
classe:
        ID
        | classe '.' ID
target:
        '(' target ')'
        | ARBITRAGE '(' target ')'
        | HACHAGE '(' target ')'
        | event_target
event_target:
     ID '.' ID
var_list:
        | var_list1
var_list1:
          var_list1 ',' ID
        | ID
```

Pour les mêmes raisons qu'énoncées dans la section concernant la grammaire d'ISL4WComp, la grammaire de BSL est également plus simple que le modèle présenté dans la fusion. Nous avons mis en place des transformations entre la grammaire du greffon et le modèle de fusion (section A.1.2.2).

A.3.2 Du greffon aux règles logiques

Comme pour ISL4WComp, nous allons voir d'abord la structure d'une règle et ensuite les transformations entre les spécifications et le modèle.

A.3.2.1 Structure d'une règle

Soit $\vec{a} = [a_1, a_2, a_3]$. Chaque comportement a une priorité de fusion, c'est-à-dire qu'il va dominer les comportements de priorité plus faibles.

Priorité	Notation mathématique	Description
1	$a(\vec{a})$	Arbitrage
2	$o(\vec{a})$	Découpage temporel
3	$conc(\vec{a})$	Concurrence
-	e ou e^p	Événement

Tableau A.4 – Tableau des priorités et de la notation mathématique des comportements

A.3.2.2 Transformations

Aucun algorithme de transformation n'est nécessaire car la représentation au niveau modèle des opérateurs est similaire à celle décrite dans les spécifications du greffon.

A.4 Fusion BSL

A.4.1 Introduction

BSL est composé de deux opérateurs de base : l'arbitrage et le découpage. On peut y adjoindre la diffusion d'événements (qui ne fait qu'ajouter un événement supplémentaire dans la liste de ceux traités par les opérateurs de base) et l'opérateur nop qui n'influe pas sur le résultat de la fusion.
L'approche BSL (*Behavioral Specification Language*) provient du fait que l'on désire pouvoir ordonnancer non pas la manière dont les événements exécutent les opérations – comme le permet ISL4WComp, mais pouvoir distinguer, déclarer et manipuler dans le greffon tous les événements

qui peuvent déclencher une opération (figure A.4). Par conséquent, on peut alors les ordonnancer, les arbitrer, etc.

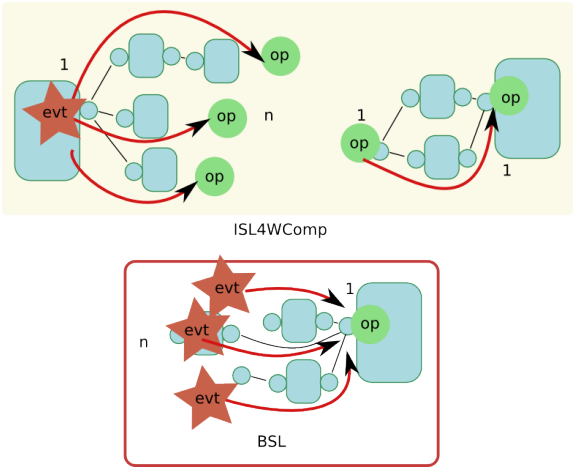


FIGURE A.4 – Différence entre l'approche BSL et ISL4WComp

A.4.2 Opérateur d'arbitrage

Nous appelons l'arbitrage d'un ensemble d'événements, le fait de pouvoir sélectionner des événements prioritaires par rapport à d'autres événements pendant un intervalle de temps déterminé. Pour réaliser cela, nous ajoutons une information à chaque événement déclaré dans la spécification du greffon permettant d'indiquer sa priorité. Ainsi, on note un événement e de priorité p comme cela : e^p.

L'équation A.16 représente la fusion de deux arbitrages $a(\vec{x})$ et $a(\vec{y})$ dont le résultat est également un arbitrage. La fusion se passe au niveau des listes. Nous introduisons la fonction tri_1 qui permet de fusionner deux listes et de trier leurs éléments dans l'ordre décroissant de leur priorité. De plus, deux événements de même priorité doivent être identiques, sinon la fusion échoue, car nous ne pouvons pas décider automatiquement de l'ordre entre ces événements.

$$a\left(\vec{x}\right) \oplus a\left(\vec{y}\right) = a\left(\text{tri}_1(\vec{x}, \vec{y})\right) \tag{A.16}$$

A.4.3 Opérateur de découpage temporel

Nous appelons le découpage temporel d'un ensemble d'événements, le fait de pouvoir bufférisér des événements et les transférer dans un ordre prédéfini. On ordonne ainsi les événements dans le temps.
Pour réaliser cela, nous ajoutons une information à chaque événement déclaré dans la spécification du greffon permettant d'indiquer son ordre de passage.

L'équation A.17 représente la fusion de deux découpages $o(\vec{x})$ et $o(\vec{y})$ dont le résultat est également un découpage. La fusion se passe au niveau des listes et est la même que pour l'arbitrage. Nous introduisons la fonction tri_2 qui permet de fusionner deux listes et de trier leurs éléments dans l'ordre décroissant de leur passage. Deux événements ayant le même ordre de passage doivent être identiques, sinon la fusion échoue, car nous ne pouvons pas décider automatiquement de l'ordre entre ces événements.

$$o\left(\vec{x}\right) \oplus o\left(\vec{y}\right) = o\left(\text{tri}_2(\vec{x}, \vec{y})\right) \tag{A.17}$$

A.4.4 Opérateur de concurrence

Nous appelons la concurrence d'un ensemble d'événements, le fait que ces événements sont déclarés comme pouvant déclencher une opération sans contraintes particulières.

L'équation A.18 représente la fusion de deux concurrences d'événements $conc(\vec{x})$ et $conc(\vec{y})$. La fusion consiste à concaténer les deux listes d'événements et d'enlever les doublons. Nous appelons cette opération concat_1.

$$conc\,(\vec{x}) \oplus conc\,(\vec{y}) = conc\,(\text{concat}_1(\vec{x}, \vec{y})) \tag{A.18}$$

A.4.5 Forme canonique

L'existence d'une forme canonique, et d'une méthode générale pour mettre sous cette forme tous les éléments d'un ensemble donné est une propriété essentielle, et nécessaire, à la calculabilité sur l'ensemble des opérateurs.

A.4.5.1 Tri des opérateurs

Les opérateurs sont spécifiés sans ordre précis dans une spécification. La forme canonique consiste à les classer par catégorie : concurrence, découpage et arbitrage (tableau A.5).

Classement	**Opérateur**
1	Concurrence
2	Découpage temporel
3	Arbitrage

Tableau A.5 – Classification des opérateurs BSL

A.4.5.2 Fusion locale et globale

Pour chaque catégorie d'opérateurs, on résout les conflits qu'il peut y avoir localement à une spécification donnée (voir sections A.4.2, A.4.3 et A.4.4). Ensuite, on résout les conflits entre les catégories d'opérateurs en s'appuyant sur les règles décrites dans le tableau A.6.

Catégorie	**Ensemble**	**Règles**
Événements arbitrés	$\vec{a}$	
Événements découpés	$\vec{d}$	$\forall x \in \vec{a}, x \notin \vec{d}$
Événements concurrents	$\vec{c}$	$\forall x \in \vec{a}, \forall y \in \vec{d},\ x$ et $y \notin \vec{c}$

Tableau A.6 – Règles de fusion globale

A.4.6 L'opération de fusion n-aire

Comme la fusion et la composition BSL se fait essentiellement par tri, concaténation de listes la fusion n-aire propose simplement en entrée deux listes.

Annexe B

Preuves des propriétés de commutativité, d'associativité et d'idempotence

Sommaire

B.1 Introduction

Nous avons étudié l'affectation de propriétés mathématiques de commutativité, d'associativité et d'idempotence à l'opération de fusion. Cela permet de faire des calculs sur les spécifications d'adaptation. Nous avons intégré ces propriétés dans les deux formes d'aspect d'assemblage ISL4WComp et BSL.

B.2 ISL4WComp

D'abord, deux transformations que nous avons présentées dans le chapitre des aspects d'assemblage n'influent pas sur les propriétés de la fusion, car la fusion ne concerne que la mise sous forme canonique :

- Transformation de la séquence et création de plusieurs niveaux de concurrence,
- Tri des opérateurs.

Ainsi, les spécifications mises sous forme canonique se fusionnent en plusieurs étapes. D'abord, on fusionne les concurrences, ensuite les conditions et enfin les comportements finals :

- Fusion des concurrences
- Fusion des conditions
- Fusion des comportements finals
  - Délégation
  - Call
  - nop
  - Opération

B.2.1 Introduction

Soient deux vecteurs α et β tels que :

$$\vec{\alpha} = [\text{par}_{K_1}(\vec{a_1}), \ldots, \text{par}_{L_n}(\vec{a_n})]$$

$$\vec{\beta} = [\text{par}_{L_1}(\vec{b_1}), \ldots, \text{par}_{K_m}(\vec{b_m})]$$

On suppose que ces deux vecteurs sont dans leur forme canonique (listes, séquences transformées et commençant par des *par*, puis if et terminant par des comportements finals).

B.2.2 Concaténation

D'abord, conformément au prédicat $islw\_fusion$, les deux vecteurs sont concaténés dans un nouveau vecteur que l'on note $\vec{\gamma}$:

$$\vec{\gamma} = \vec{\alpha}.\vec{\beta}$$

On calcule ensuite le vecteur résultat du traitement de $\vec{\gamma}$.

B.2.3 Fusion

Tous les *par* de même niveau sont ensuite fusionnés deux à deux et chaque résultat est fusionné avec le *par* suivant :

$$\vec{\gamma} = [\alpha_1, \ldots, \alpha_n, \ldots, \beta_1, \ldots, \beta_m]$$

Pour chaque élément, nous avons les propriétés suivantes : α_i est de niveau i ainsi que β_i, car les vecteurs de départ est sous forme canonique. Il n'y a également qu'un unique élément α_i de niveau i, de même pour β_i. Ainsi, la fusion des ceux vecteurs consiste à fusionner localement deux éléments

de même niveau. Conclusion, deux opérateurs *par* de niveaux différents ne fusionnent pas et deux opérateurs $\text{par}_L(a)$ et $\text{par}_L(b)$ donne $\text{par}_L(a \oplus b)$ où $\oplus$ denote l'opération de fusion d'opérateurs de condition.

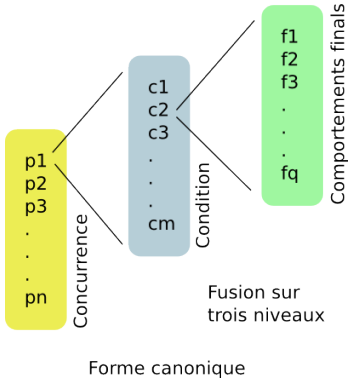


FIGURE B.1 – Forme canonique et propriétés de la fusion

Or, sous la forme canonique, l'opérateur conditionnel est également sans ambiguité B.1 : pour chaque élément de la liste des opérateurs conditionnels l, il en existe un seul élément l_i pour chaque condition c_i. Cela signifie qu'il s'agit de fusionner deux éléments l_i et l'_i de conditions identiques c_i lorsque l'on fusionne les deux vecteurs $\vec{\alpha}$ (apportant l_i) et $\vec{\beta}$ (apportant l'_i).

Niveaux	Conditions	Comportments finals	Fusion
identiques	identiques	identiques	idempotente
identiques	identiques	différents	fusion finale (1)
identiques	différentes	-	juxtaposition (2)
différents	-	-	juxtaposition (3)

Tableau B.1 – Quand y a-t-il fusion, juxtaposition et idempotence ?

Par conséquent, lorsque des éléments sont identiques de la colonne 'niveau', jusqu'à la colonne 'comportements finals', ils sont idempotents (tableau B.1). Sinon, une fusion ou une juxtaposition est faite.
La juxtaposition est commutative et associative.

B.2.4 Fusions finales

B.2.4.1 Idempotence et commutativité

Par construction, la fusion finale est idempotente et commutative.

	del	call	nop	op	err1
del	del (A.11)/err (A.10)	del (A.11)	del (A.11)	del (A.11)	del (A.15)
call	(A.9)	call (A.8)	nop (A.8)	op (A.8)	err1 (A.15)
nop	(A.9)	(A.9)	nop (A.13)	op (A.13)	err1 (A.15)
op	(A.9)	(A.9)	(A.9)	op (A.8) / err1 (A.14)	err1 (A.15)
err1	(A.9)	(A.9)	(A.9)	(A.9)	err1 (A.15)

Tableau B.2 – Fusion des comportements finals

En effet, l'équation A.8 nous donne pour tout a et pour tout b la propriété d'idempotence :
$f(a, b) = a$ si $a = b$

B.2.4.2 Associativité

Pour effectuer la preuve de l'associativité, nous avons fait les calculs de toutes les possibilités par un ensemble de prédicats.
Le premier prédicat passe une liste d'opérateurs qui vont ensuite être fusionnés. Il s'agit des opérateurs de délégation (3 délégations différentes), des opérateurs call et nop et trois appels d'opération (3 appels différents).

```
run :-
    assoc0([del(a),del(b),del(c),call,nop,op(a),op(b),op(c)]).
```

Le prédicat *assoc*0 met à zéro le compteur n (nombre de calculs effectués) et appel le prédicat *assoc* avec trois listes des opérateurs à fusionner.

```
assoc0(L) :- retractall(n(_)),assert(n(0)),assoc(L,L,L).
```

Le prédicat *assoc* prend chaque élément de la troisième liste avec lequel il appelle le prédicat *assoc*1.

```
assoc(_,_,[]).
assoc(L1,L2,[C|TL3]) :-
    assoc1(L1,L2,C),
    assoc(L1,L2,TL3).
```

Le prédicat *assoc*1 fait la même chose avec la deuxième liste en appelant le prédicat *assoc*2.

```
assoc1(_,[],_).
assoc1(L1,[B|TL2],C) :-
    assoc2(L1,B,C),
    assoc1(L1,TL2,C).
```

Enfin, le prédicat *assoc*2 appelle *assoc*3 avec chaque élément de la première liste. L'association des trois derniers prédicats permet de passer en revue toutes les combinaisons d'opérateurs pour le calcul de l'associativité.

```
assoc2([],_,_).
assoc2([A|TL1],B,C) :-
    assoc3(A,B,C),
    assoc2(TL1,B,C).
assoc3(A,B,C) :- disp(A,B,C).
```

Le prédicat suivant permet de créer le tableau de toutes les combinaisons possibles que l'on retrouve en annexe II. Mis à part le code LaTex produit, il appelle les prédicats *calc*1 et *calc*2 qui permettent de calculer le résultat des calculs respactifs : $a \oplus (b \oplus c)$ et $(a \oplus b) \oplus c$. Lorsqu'une erreur survient lors de la fusion, il affiche *error* dans le tableau des résultats. Enfin, il compare les deux résultats précédent et affiche OK lorsqu'ils sont égaux (associatifs) sinon BAD.

```
disp(A,B,C) :-
    n(N), SN is N+1, assign(n,SN),
    write(N),write('&$('),
        write(A),write('+'),write(B),write(')+'),write(C),
    write('=$&'),
    catch( calc1(A,B,C,R1), _, R1=error ),
    write('$'),write(R1),write('$&'),
    write('$'),write(A),write('+('),write(B),write('+'),write(C),
    write(')=$&'),
    catch( calc2(A,B,C,R2), _, R2=error ),
    write('$'),write(R2),write('$&'),
    (R1=R2, write('OK') ; write('BAD')), write('\\\\\\hline'),nl.
```

Les prédicats *calc*1 et *calc*2 sont similaires au parenthésage près, comme nous l'avons vu précédemment.
Le prédicat consiste à mettre *error*1 à zéro (erreurs de fusion d'opérations). Il calcule $AB = A \oplus B$ traduit le résultat en spécifications. Puis, il calcule $ABC = AB \oplus C$ et vérifie que toutes les erreurs de fusion d'opérateurs ont été résolues ($error1 = 0$).

```
calc1(A,B,C,R) :-
        retractall(error1(_)),assert(error1(0)), % error1 = 0
        islw_fusion(A,B,AB),
        %(error1(0);write(' (ambiguity found) ')),
        tri(AB,ABr),
        islw_fusion(ABr,C,ABC),
        (error1(0);throw('cannot resolve operation ambiguities')),
        tri(ABC,R).

        calc2(A,B,C,R) :-
        retractall(error1(_)),assert(error1(0)), % error1 = 0
        islw_fusion(B,C,BC),
        %(error1(0);write(' (ambiguity found) ')),
        tri(BC,BCr),
        islw_fusion(A,BCr,ABC),
        (error1(0);throw('cannot resolve operation ambiguities')),
        tri(ABC,R).
```

Le prédicat *assign* permet d'affecter une valeur V à un prédicat X.

```
assign(X,V) :- Old=..[X,_],retract(Old),New=..[X,V],assert(New).
```

B.3 BSL

La fusion et la composition en BSL consiste à classer les éléments d'une liste et à ordonner ensuite ces éléments tout en enlevant les doublons. C'est ce que nous appelons la fusion locale.

Ensuite, elle consiste à enlever les éléments déjà présents dans une première liste d'une seconde.

L'objectif de cette section est de montrer que ces opérations sont associatifs, commutatifs et idempotents.

B.3.1 Fusion locale : classement et doublons

Le classement des liste de concurrence $conc(C_i)$, $a(A_i)$ et $o(O_i)$ consiste à les ranger dans trois ensembles C, A et O. Cette opération est associative, commutative.
Les listes C_i (respectivement A_i et O_i) sont concaténées et les doublons sont enlevés. Cela permet de rendre l'opération non seulement associative et commutative, mais aussi idempotente.

B.3.2 Fusion globale

La fusion globale consiste à rendre *prioritaires* certaines listes. Cela signifie que des éléments déjà présent dans la liste prioritaire vont être retirés des listes moins prioritaires. Ainsi, les éléments présents dans l'ensemble A sont retirés des ensembles O et C ; les éléments présents dans les ensembles A et O sont retirés de la liste C.

- Arbitrage A
- Découpage temporel O
- Concurrence C

La fusion locale est associative, commutative et idempotente en considérant le contenu des trois ensembles A, O et C. La fusion globale est une opération sur ces trois ensembles qui est indépendante de l'ordre dans lequel les spécifications ont été émises, puisqu'elle ne prend entrée que ces trois ensembles.

En conclusion, la fusion et la composition BSL est une opération commutative, associative et idempotente.

Annexe C

Une implémentation minimale des aspects d'assemblage en Prolog embarqué

Sommaire

L'aspect d'assemblage est une notion travaillant principalement sur des algorithmes récursifs et sur le traitement des chaînes de caractères. Notre choix s'est porté naturellement sur une implémentation en Prolog pour être le plus proche possible des algorithmes proposés.
Nous voyons dans ce chapitre l'implémentation des greffons et de la fusion ISL4WComp et BSL. Quelques exemples d'utilisation sont donnés à la fin du chapitre. Mais avant d'aller dans le détail, nous présentons le logiciel cxprolog sur lequel nous avons implémenté nos algorithmes en Prolog.

C.1 Détails sur l'implémentation

CxProlog [1, 2] est un système Prolog basé sur la machine abstraite de Warren WAM [3] supportant presque la totalité de la première partie du standard ISO Prolog ainsi que le dialect Edinburgh. L'implémentation C de CxProlog a une taille raisonnable et peut être modifiée afin d'être intégrée à tout système embarqué. C'est à la fois un compilateur et il dispose d'une interface d'interprétation. C'est cette dernière que nous utilisons. Nous avons testé notre implémentation sous Linux 32-bit.

C.2 Greffon ISL4WComp

L'implémentation des greffons ISL4WComp s'appuie sur des règles logiques Prolog et sur la transformation de l'opérateur de séquence en niveaux de concurrence.

C.2.1 Règles en Prolog

Soit $\vec{a} = [a_1, a_2, a_3]$, $\vec{c} = [t, f]$ et $\vec{\vec{d}} = [[a_1], [b_1, b_2]]$. Chaque opérateur a une priorité de fusion, c'est-à-dire qu'il va dominer les opérateurs de priorité plus faibles.

Priorité	Notation mathématique	Notation Prolog
1	$\text{par}_L(\vec{a})$	$par(L, [a_1, a_2, a_3])$
2	$if\left(\vec{c}, \vec{\vec{d}}\right)$	$if([t, f], [[a_1], [b_1, b_2]])$
3	$del(p)$	$leaf(del(p))$
4	$call$	$leaf(call)$
5	p	$leaf(p)$
6	nop	$leaf(nop)$

Tableau C.1 – Tableau des priorités et de la notation Prolog des opérateurs

C.2.2 Transformation composants/opérateurs

La transformation entre les spécifications et le modèle de calcul se fait dans les deux sens. Nous décrivons dans cette section les deux transformations. Notons que nous n'avons pas pu utiliser, du moins pour cette thèse, la puissance de Prolog pour faire automatiquement le calcul de la transformation inverse. En effet, s'il s'avère simple dans un sens (section C.2.2.1), il demeurre difficile de transformer la forme canonique en spécification en raison de la multitude d'optimisations possibles (section C.2.2.2).

C.2.2.1 Des spécifications au modèle

La transformation spécifications/modèle nécessite un seul prédicat noté $tr(A, B^{\uparrow})$ qui prend en entrée le comportement à traduire. Les comportements binaires de la spécification sont traduits en opérateurs n-aires dans le modèle. Les fils sont récursivement traités. Nous noterons $not(C)$ la négation d'une entité booléenne en Prolog.

```
tr( par(A,B) , par(0,[Ar,Br]) ) :- tr(A,Ar), tr(B,Br).
tr( seq(A,B) , seq([Ar,Br]) ) :- tr(A,Ar), tr(B,Br).
tr( if(C,A,B) , if( [C,not(C)], [[Ar],[Br]] ) ) :- tr(A,Ar), tr(B,Br).
```

Les comportements finals sont traduits en rajoutant le prédicat *leaf* en amont. Ces règles constituent les terminaisons de la récursion.

```
tr( del(M), leaf(del(M)) ).
tr( nop , leaf(nop) ).
tr( call , leaf(call) ).
tr( M, leaf(M) ).
```

C.2.2.2 Du modèle aux spécifications

Comme nous l'avons déjà mentionné, cette transformation n'est pas simple. On ne peut pas s'appuyer (en l'état actuel) sur la première transformation pour créer celle-ci. Nous avons donc implémenté le prédicat $tri0(A, R^{\uparrow})$ (TRansformation Inverse 0) qui résume les deux premières étapes. Il s'agit d'abord de trier les opérateurs de concurrence en fonction de leur niveau (prédicat de tri rapide). Ensuite, on effectue la transformation (prédicat *tri*). On suppose avant toute chose que A se trouve sous forme canonique.

```
tri0( A , R ) :- quick_sort2(A,B), tri(B,R).
```

On a implémenté l'algorithme QuickSort avec accumulateur. Nous ne nous attarderons pas sur sa description.

```
quick_sort2(List,Sorted):-q_sort(List,[],Sorted).
q_sort([],Acc,Acc).
q_sort([H|T],Acc,Sorted):-
        pivoting(H,T,L1,L2),
        q_sort(L1,Acc,Sorted1),q_sort(L2,[H|Sorted1],Sorted).
```

Nous avons quand même modifié la manière de calculer le pivot en comparant les niveaux de concurrence.

```
pivoting(_,[],[],[]).
pivoting(par(H,HP),[par(X,XP)|T],[par(X,XP)|L],G) :-
  H=<X, pivoting(par(H,HP),T,L,G).
pivoting(par(H,HP),[par(X,XP)|T],L,[par(X,XP)|G]) :-
  H>X, pivoting(par(H,HP),T,L,G).
```

Le prédicat *tri* convertit dans un premier temps les foncteurs *leaf* en comportements finals (nop, call, appel d'opération, délégation).

```
tri( leaf(nop), nop ).
tri( leaf(call), call ).
tri( leaf(M), M).
tri( leaf(del(A)), del(A) ).
```

Ensuite, nous tranduisons les opérateurs n-aires. C'est la partie la plus difficile. Nous commençons par l'opérateur conditionnel. D'abord, une première règle permet de retrouver la "signature" de la transformation directe du comportement conditionnel $if(c, \bar{c} \ldots$. La seconde règle est une optimisation (falcultative), elle permet d'éliminer la forme conditionnelle inutile $if(true, do \ldots$ du comportement.

```
tri( if( [C,not(C)], [[A],[B]] ), S ) :-
    tri(A,Ar),tri(B,Br),S=if(C,Ar,Br).
tri( if( [not(C),C], [[B],[A]] ), S ) :-
```

```
    tri(A,Ar),tri(B,Br),S=if(C,Ar,Br).
tri( if( [true], [[A]] ), S ) :- % optimisation
    tri(A,Ar),
    S=Ar.
```

La règle suivante permet de créer un comportement conditionnel simple avec une seule branche. Et la dernière règle exprime un conflit entre plusieurs opérateurs de condition à un même niveau de concurrence.

```
tri( if( [C], [[A]] ), S ) :-
    tri(A,Ar),
    S=if(C,Ar,nop).
tri( if( [C|_], [[A]|_] ), S ) :-
    tri(A,Ar), writeln('error: cannot decide'),fail.
```

Enfin, les deux dernières règles concernent l'opérateur de concurrence. La première règle traite du cas le plus simple : celui où il n'y en a pas. La seconde règle profite du tri que nous avons effectué avec le prédicat *quick_sort*2 pour construire simplement des séquences imbriquées à partir des différents niveaux de concurrence.

```
tri( [par(_,[A])] , R) :-
    tri(A,R).
tri( [par(_,[A])|TP] , R) :-
    tri(A,Ar),
    tri(TP,R1),
    R=seq(Ar,R1).
```

Remarque : Ceci est une implémentation possible de la transformation. Cette dernière peut être optimisée ; on peut également rajouter des fonction de récupération d'erreurs pour une utilisation plus approfondie.

C.2.3 Transformation de la séquence

La transformation de la séquence en concurrence par niveau est implémentée avec quatre prédicats. Chacun effectue un traitement à un niveau de récursion différent.

C.2.3.1 Prédicat principal u : transformation de la séquence.

Le prédicat principal est u qui transforme un comportement incluant des séquences, noté a, en un comportement concurrent par niveau. Ce prédicat s'appuie sur t qui permet de calculer un comportement de niveau 0 noté S avec un reste de niveau différent noté $Temp$.

```
u( A, R ) :-
    t(0,_,A,S,Temp),
    append([S],Temp,R).
```

C.2.3.2 Prédicat secondaire t, élimination de séquence

La règle principale du prédicat t consiste à éliminer une séquence. Cela se fait en quatre étapes. D'abord, à partir d'une séquence notée $seq(\vec{c})$, on calcule récursivement un comportement sans séquence c' à partir de $c_1 \in \vec{c}$. On note alors $par_l(c')$ le résultat de niveau l. On calcule alors les comportements de niveau supérieur à partir du prédicat *incpar*. Le reste est alors concaténé dans $NewPar$.

```
t(L,H, seq([C|TC]) , R, NewPar) :-
    t(L,L1, C, C1, N1), % construct C1 from C
    R=par(L,[C1]),      % same level
    incpar(L1,H,TC,N2), % remaining behaviors are up-level
    append(N1,N2,NewPar).
```

La règle de terminaison (traitement d'un comportement final) est comprise dans la règle suivante. Mais, d'abord, elle permet de calculer récursivement l'élimination de la séquence dans l'opération de concurrence, puis dans l'opérateur de condition ainsi que ses comportements imbriqués (quatrième ligne).

```
t(L,H, A , R , NewPar ) :-
    ( A=par(K,B), tl(L,H, B,B1,NewPar), R=par(K,B1) );
    ( A=if(C,B),  tl(L,H, B,B1,NewPar), R=if(C,B1)  );
    ( A=[_|_], % inner behaviors (from IF)
        tl(L,H, A,R,NewPar) );
    ( A=leaf(C),   NewPar = [], H=L,    R=leaf(C)   ).
```

C.2.3.3 Prédicat secondaire incpar, création de niveaux supérieurs

Le prédicat *incpar* permet d'incrémenter le niveau de concurrence. Il vérifie qu'il n'y pas de séquence dans la nouvelle branche concurrente. Il traite récursivement tous les niveaux et construit la liste des concurrences de niveau différent.

```
incpar(L,J, [C|TC], NewPar) :-
    M is L+1,
    t(M,M1, C, C1, N1), % construct C1 from C
    P1 = [par(M,[C1])],
    incpar(M1,J, TC, P2),
    append(N1, P1, Te),
    append(Te, P2, NewPar).
incpar(J,J, [], []). % up-level
```

C.2.3.4 Prédicat secondaire tl, wrapper de t

Le prédicat suivant *tl* est un wrapper du prédicat *t* afin que ce dernier puisse être appliqué à une liste de comportements.

```
tl(V,W, [L|TL],R,N) :-
  t(V,W, L, R1, N1),
  tl(V,W, TL, R2, N2),
  append([R1],R2,R),
  append(N1,N2,N).
tl(_,_, [],[],[]).
```

C.3 Règles de fusion ISL4WComp

Le code source est constitué de plusieurs prédicats.

C.3.1 Prédicat fpar

Le prédicat fpar $(c, E, R^{\uparrow})$ prend les paramètres suivants :

1. Entrée c : le comportement de concurrence à fusionner avec l'ensemble ;
2. Entrée E : un ensemble de comportements de concurrence déjà fusionnés ;

3. Sortie R : le résultat de la fusion du comportement c dans E.

Nous avons implémenté les deux premières équations de la fusion de la concurrence (équations A.1 et A.2) dans ce même prédicat Prolog.

```
% MERGE par( A ) INTO par([..])
% HYP: par([..]) is correctly merged
fpar(
    par( Level, A ),
    par( Level, [B|TB] ),
    R
 ) :-
      ( A=B,
            R = par(Level,[B|TB])
      );
      ( % they are different
            fpar( par(Level,A) , par(Level,TB) , par( Level, TBR ) ),
            append([B],TBR, Temp), % PAR is now idempotent
            R = par( Level, Temp )
      ).
```

L'équation A.3 représente la condition d'arrêt de la récurrence déclenchée par le précidat ci-dessus. Nous l'avons implémenté de sorte qu'elle soit indépendante du niveau de concurrence. L'ensemble vide est implémenté par la liste vide [].

```
% HYP: if([],[]) is always correctly merged
fpar( par( Level, A ), par( _, []), par( Level, A ) ).
```

Enfin, l'équation A.4 qui réunit plusieurs niveaux de concurrence est est implémentée de la manière suivante :

```
fpar(
    par( Level1, A ),
    par( Level2, B ),
    R
 ) :-
            R = [par( Level1, A ),par( Level2, B )].
```

C.3.2 Prédicat fif

Le prédicat fpar $(c, E, R^{\uparrow})$ prend les paramètres suivants :

1. Entrée c : le comportement de condition à fusionner avec l'ensemble ;
2. Entrée E : un ensemble de comportements de condition déjà fusionnés ;
3. Sortie R : le résultat de la fusion du comportement c dans E.

Les équations A.5 et A.6 ont été implémentées dans un unique prédicat. Nous n'avons pas créé de prédicat particulier pour l'opération d'union qui est propre à l'opérateur de condition. Nous utilisons les techniques d'unification propre à Prolog.

```
% MERGE if(C1,A1) INTO if([..],[..])
% HYP: if([..],[..]) is correctly merged
fif(
    if( [C1] , [A1] ),
    if( [C2|TC2] , [A2|TA2] ),
    R
 ) :-
```

```
    ( (C1=C2 ; C2=true ),
        % Merge with head
        append(A1,A2,S),
        Temp=if([C1],[S]),
        % Merge result with rest
        fif( Temp , if( TC2, TA2 ), R )
    );
    ( C1=true,
        % Merge with head
        append(A1,A2,S),
        % Merge result with rest
        fif( if([C1],[A1]) , if( TC2, TA2 ), if( TC2R, TA2R ) ),
        % Append {C2,A2} to the result to build the final solution
        append([C2],TC2R, RC), append([S],TA2R, RA),
        R = if( RC , RA )
    );
    ( % they are different
        % Merge if(C1,A1) with the rest and keep {C2,A2} safe
        fif( if([C1],[A1]) , if( TC2, TA2 ) , if( TC2R, TA2R ) ),
        % Append {C2,A2} to the merge result to build the solution
        append([C2],TC2R, RC), append([A2],TA2R, RA),
        R = if( RC , RA )
    ).
```

Nous sommes confrontés dans l'implémentation de l'équation A.7 à définir deux prédicats pour faire face à deux manières d'écrire l'ensemble vide : l'ensemble contenant l'élément vide ou l'ensemble vide tout court.

```
fif( if( C , A ), if([],[[]]), if( C , A ) ).
fif( if( C , A ), if([],[]), if( C , A ) ).
```

C.3.3 Prédicat fleaf

Le prédicat fleaf $(c, E, R^{\uparrow})$ prend les paramètres suivants :

1. Entrée c_1 : un comportement final ;
2. Entrée c_2 : un autre comportement final ;
3. Sortie R : le résultat de la fusion des comportements c_1 et c_2.

Les équations de la section A.2.3 ont été implémentées dans un unique prédicat. Nous avons scindé vonlontairement le prédicat en plusieurs parties pour les expliquer un par un.

```
% MERGE leaf( A ) WITH leaf( B )
% HYP: leaf( B ) is correctly merged
fleaf(
    leaf( A ),
    leaf( B ),
    R
 ) :-
```

L'idempotence au niveau des comportements finals est la première règle à appliquer. On unifie les deux comportements donnés.

```
    % IDEM
    ( A=B,
        R = leaf(B)
    );
```

L'implémentation de la délégation est commutative. Elle génère ici une erreur lorsque deux délégations différentes (vu que l'on a déjà passé la règle d'idempotence ci-dessus) tentent de fusionner.
L'erreur numéro 1 peut être levée s'il y a fusion avec l'opérateur de délégation.

```
% DEL
( A=del(_),
      (( B=del(_),
           throw('cannot merge two different delegates')
      );(B=call,
           R=leaf(A)
      );(B=nop,
           R=leaf(A)
      );(B=op(_),
           R=leaf(A)
      );(B=error1,
           R=leaf(A),
           error1(N), PN is N-1, assign(error1,PN)
      ))
);
```

La fusion du call suit le même schéma que la délégation, mais sans lever de message d'erreur. Il a la priorité sur l'opérateur nop et sur les appels d'opération.

```
% CALL
( A=call,
      (( B=nop,
           R=leaf(nop)
      );(B=op(_),
           R=leaf(B)
      ))
);
```

L'implémentation du nop est la même que pour le call.

```
% NOP
( A=nop,
      (( B=op(_),
           R=leaf(B)
      ))
);
```

La fusion de deux opérations différentes émet un message d'erreur de niveau 1. Cela signifie que l'ambiguité peut être levée par une fusion avec l'opérateur de délégation. Si cette fusion n'a pas lieu, alors il y a échec de la fusion.

```
% OPER
( A=op(_),
      (( B=op(_),
           R=leaf(error1), error1(N),SN is N+1, assign(error1,SN)
      ))
);
```

Lorsque l'on a pas pu fusionner A avec B, on essaie de fusionner B avec A. C'est la règle qui exprime la commutativité :

```
(
  fleaf(B,A)
);
```

Pour conclure, on considère l'élément vide [] pour l'opération de fusion des comportements finals.

```
fleaf( leaf( A ), [], leaf( A ) ).
```

C.3.4 Tri des opérateurs

Les opérateurs sont triés selon l'ordre indiqué dans le tableau C.2. Pour trier les opérateurs d'un comportement donné, nous avons écrit trois prédicats.

Classement	Opérateur	Prédicats pour le tri
1	Concurrence	$can0\left(n, c, p, S^{\uparrow}\right)$, $c(\dots)$ et $cif(\dots)$
2	Conditionnel	-
3	Opérateurs finals	-

Tableau C.2 – Classification des opérateurs et prédicats concernés

C.3.4.1 Prédicat principal can0, détection des conflits

Le premier prédicat $can0\left(n, c, p, S^{\uparrow}\right)$ prend les paramètres suivants :

1. Entrée n : le niveau de concurrence ;
2. Entrée c : le comportement à trier ;
3. Entrée p : pointeur sur l'opérateur précédemment traité par rapport à c ;
4. Sortie S : comportement c trié.

Cette première règle permet de traiter le cas simple où le comportement c à trier commence par l'opérateur de concurrence de niveau l portant sur une liste de comportement que l'on note $\vec{a}$. Il s'agit au premier abord, de modifier cet opérateur pour construire un nouvel opérateur de concurrence de même niveau l, mais avec une liste de comportements triés noté *Result* dans le code. Le calcul se fait en deux temps.

```
can0(L,  par(L,[A|TA]) , Prec , par(L,Result) ) :-
```

D'abord, si l'opérateur précédent était un opérateur de concurrence (*par*) ou bien si l'on se trouve au début du traitement du comportement c (*start*), on fusionne un unique élément $a_1 \in \vec{a}$ (extrait de c) avec le reste du comportement c privé de a_1. Aussi, on construit artificiellement son précédent $\text{par}_l(\vec{a})$. On obtient ainsi une première partie trié notée *Temp1*.
On appelle ensuite récursivement le prédicat avec le reste de la liste $(\vec{a} \setminus a_1)$ dont le précédent est *Prec*. On obtient la deuxième partie triée notée *Temp2*.
La solution est l'union des deux parties.

```
( (Prec=par(L,_);Prec=start),
    can0(L, A, par(L,[A|TA]), par(L,Temp1) ), % merge one element A
    can0(L, par(L,TA) , Prec, par(L,Temp2) ), % recursive call
    append(Temp1,Temp2,Result)
);
```

Ensuite, si l'opérateur précédent n'est pas bien trié, on va essayer d'inverser les opérateurs concernés à partir du comportement : $p = x(\dots \text{par}_l(\dots)\dots)$ où x est un opérateur n-aire autre que *par*. On réitère ensuite pour traiter de nouveaux inversements.

```
( % else
    c(L, Prec , Temp), % inverse BAD OPERATOR and PAR
    can0(L, Temp , Prec, Result ) % redo with corrected result
).
```

Voici la règle de terminaison du prédicat récursif :

```
can0(L,  par(L,[])  , _ , par(L,[])  ).
```

En plus de trier le comportement c, le prédicat rajoute, s'il est manquant, l'opérateur de concurrence, même si aucune concurrence n'est spécifiée afin de se ramener à une forme canonique.

```
can0(L, A, start , R ) :-
    can0(L,par(L,A),start ,R).
```

Si A n'est pas un opérateur de concurrence et que l'on ne se trouve pas au début du traitement du comportement, cette règle permet de passer la main au second prédicat c.

```
can0(L, A , _, Result ) :-
    c(L, A , Result).
```

C.3.4.2 Prédicat secondaire c, résolution par inversion d'opérateur

Ce deuxième prédicat $c\left(n, c, p, S^{\uparrow}\right)$ prend les paramètres suivants :

1. Entrée n : le niveau de concurrence ;
2. Entrée c : le comportement à trier ;
3. Entrée p : pointeur sur l'opérateur précédemment traité par rapport à c ;
4. Sortie S : comportement concurrentiel c trié.

Cette première règle stipule que lorsqu'un opérateur de concurrence se trouve dans la liste des comportements finals, on fait passer l'opérateur de concurrence avant l'opérateur de condition grâce au prédicat *cif*.
On appelle ensuite récursivement c pour traiter la suite de la liste. La solution est l'union des deux dernières opérations.

```
c(L, if( [C|TC] ,[[par(L,A)]|TA] ), Sol ) :-
      cif(L,C,A,T), % rebuild a correct par(L,if(C,A..)) -> in T
      c(L,if(TC,TA),par(L,R)), % recursive call with the rest
      append(T,R,LIST), Sol=par(L,LIST).
```

Voici la règle de terminaison du prédicat récursif :

```
c(L, if([],[]), par(L,[]) ).
```

Le but de ce prédicat est de produire des comportements concurrentiels triés : dans tous les cas, il doit contenir un opérateur concurrentiel à sa racine :

```
c(L, if(C,A), par(L,[if(C,A)]) ).
```

Nous rajoutons deux règles : la première permet de construire automatiquement une forme canonique à partir d'un comportement final. La seconde permet de construire une forme canonique à partir d'un comportement concurrent tout en vérifiant que son contenu est bien trié.

```
% Take care of other things
c(L, leaf(C), par(L,[if([true],[[leaf(C)]])]) ).
c(_, par(L1,C), par(L1,D) ) :-
      cif(L1, true , C, D).
```

C.3.4.3 Prédicat secondaire cif, ajout des opérateurs conditionnels manquants

Ce deuxième prédicat $cif\left(n, c, p, S^{\uparrow}\right)$ prend les paramètres suivants :

1. Entrée n : le niveau de concurrence ;
2. Entrée c : le comportement à trier ;
3. Entrée p : pointeur sur l'opérateur précédemment traité par rapport à c ;

4. Sortie S : comportement c trié.

Le prédicat *can0* détecte les conflits généraux d'ordre entre un opérateur de concurrence et un opérateur quelconque au niveau de la racine. Le prédicat c fait ce travail au niveau de l'opérateur conditionnel. Ce troisième prédicat est différent, il construit l'inversion entre un opérateur conditionnel et un opérateur de concurrence, en parcourant en profondeur les listes de comportements. Cette première règle cherche et résout les conflits à la fois dans la liste des comportements finals et en profondeur dans la liste des comportements conflictuels comme E.

```
% for each a in E|TE append an if(a) in the result
cif(L,C,[par(L,E)|TE],T) :-
    cif(L,C, TE, R), % search and resolve conflicts in TE
    cif(L,C, E, R1), % search and resolve conflicts in E
    append(R1,R,T).
```

S'il n'y a aucun conflit dans l'élément courant E de la liste, alors on peut construire une condition à partir de E et vérifier récursivement que le reste TE ne contient pas de conflit. Sous-entendu, on résout le conflit en éliminant l'opérateur de concurrence à cet endroit du comportement sachant qu'il va être rajouté à l'endroit précédent.

```
cif(L,C,[E|TE],T) :-
    cif(L,C, TE, R),
    append([if([C],[[E]])],R,T).
```

Voici la règle de terminaison du prédicat récursif :

```
cif(_,_,[],[]).
```

C.3.5 Application des équations de fusion

L'application des équations de fusion se fait par l'intermédiaire du prédicat $rr(\vec{a}, s^{\uparrow})$. Il prend les paramètres suivants :

1. Entrée $\vec{a}$: un comportement ;
2. Sortie s : le résultat de la fusion interne au comportement.

```
% merge things
rr([P|TP],ST) :-
    fpar(P,par(_,[]),par(L,T)),
    fif1(T,if([],[]),if(Cond,Actions)),
    fleaf1(Actions,[],AM),
    S=par(L,[if(Cond,AM)]),
    rr(TP,ST1),
    append([S],ST1,ST).
rr([],[]).
```

Nous ne pouvons pas appeler directement les prédicats de fusion (sauf pour la fusion de la concurrence, car *rr* est récursif sur la liste des comportements concurrents), car nous travaillons sur des listes. Ainsi, deux prédicats supplémentaires ont été mis en place pour traiter un à un, les éléments des listes de comportement.

Le premier prédicat *fif*1$(\vec{a}, \vec{b}, s^{\uparrow})$. Il prend les paramètres suivants :

1. Entrée $\vec{a}$: liste des comportements conditionnels ;
2. Entrée $\vec{b}$: comportement conditionnel déjà fusionnés ;
3. Sortie S : le résultat de la fusion des listes

La règle de récurrence ci-dessous calcule à chaque étape la nouvelle valeur de R. À aucun moment, le résultat S n'est calculé directement. Ce calcul se fait dans la règle de terminaison.

```
fif1([if([C|TC],[AA|TAA])|TA],S) :-
    fifcat(if(TC,TAA),TA,IF),  % build one IF from the list
    fif(if([C],[AA]), IF , S). % then merge recursively

fifcat(if(TC,TAA),[if(TD,TBB)|TA],S) :-
    append(TC,TD,T1),
    append(TAA,TBB,T2),
    fifcat(if(T1,T2),TA,S).
```

La règle de terminaison sert à copier le résultat $\vec{b}$ dans le paramètre de sortie S du prédicat.

```
fifcat(A,[],A).
```

Le second est un prédicat-double : $fleaf1(\vec{\vec{a}}, \vec{\vec{b}}, s^{\uparrow})$. Il prend les paramètres suivants :

1. Entrée $\vec{\vec{a}}$: double-liste des comportements finals ;
2. Entrée $\vec{\vec{b}}$: comportement finals déjà fusionnés ;
3. Sortie $\vec{\vec{S}}$: le résultat de la fusion des doubles-listes

Et $fleaf2(\vec{a}, \vec{b}, s^{\uparrow})$ prend les paramètres suivants :

1. Entrée $\vec{a}$: liste de comportements finals ;
2. Entrée $\vec{b}$: comportements finals déjà fusionnés ;
3. Sortie $\vec{S}$: le résultat de la fusion des listes

On fusionne en réalité dans les listes imbriquées. C'est pour cela que nous utilisons deux prédicats : le premier désencapsule la liste des comportements finals à fusionner et le second fait interface entre cette liste et la règle de fusion effective.

```
fleaf1([A|TA],D,S) :-
    fleaf2(A,[],T), % merge locally the elements of A -> results in T
    append(D,[[T]],SS),
    fleaf1(TA,SS,S).
```

La règle de terminaison sert à copier le résultat $\vec{\vec{b}}$ dans le paramètre de sortie $\vec{\vec{S}}$ du prédicat.

```
fleaf1([],R,R).
```

Le second prédicat appelle donc le prédicat de fusion pour chaque élément de la liste $\vec{a}$.

```
fleaf2([A|TA],R,S) :- fleaf(A,R,T), % merge effectively A with R
                      fleaf2(TA,T,S).
```

La règle de terminaison sert à copier le résultat $\vec{b}$ dans le paramètre de sortie $\vec{S}$ du prédicat.

```
fleaf2([],R,R).
```

C.3.6 Synthèse de la forme canonique

Le calcul complet de la forme canonique se fait en trois étapes. D'abord, on transforme les séquences en concurrences sur plusieurs niveaux (voir section C.2.3). Ensuite, pour chaque niveau de concurrence, on classe les opérateurs(voir section C.3.4). Enfin, on applique les équations de la fusion (voir section C.3.5).

Ce prédicat prend les paramètres suivants :

1. Entrée $\vec{a}$: comportement
2. Sortie $\vec{S}$: forme canonique

```
% class things then merge
i(A, S) :-
    u(A,A1),    % remove sequence
    j(A1,A2),   % class operator
    rr(A2, S). % merge equations
```

On ne peut pas appeler directement le prédicat de classement *can0*, car on manipule une liste. On utilise un prédicat j qui applique chaque élément à $can0$. Le résultat est concaténé.

```
j([A|TA],S) :-
    can0(_, A , start , S1),
    j(TA,S2), append([S1],S2,S).
j([],[]).
```

C.4 Prédicat de composition/fusion ISL4WComp

C.4.1 Prédicat islw_fusion0

Le prédicat islw_fusion0 $\left(L, R^{\uparrow}\right)$ prend les paramètres suivants :

1. Entrée L : liste de spécifications ISL4WComp ;
2. Sortie R : le résultat de la fusion des spécifications.

Avant de commencer la fusion de liste des spécifications, nous mettons à zéro le nombre d'erreur numéro 1. Ensuite, nous effectuons la fusion du premier élément de la liste avec le reste. Enfin, lorsque la fusion est terminée, nous vérifions que le nombre d'erreurs numéro 1 est bien zéro. Dans le cas contraire, cela signifie qu'il reste des ambiguités non résolues. La fusion échoue.

```
islw_fusion0([A|TA],R) :-
                retractall(error1(_)),assert(error1(0)), % error1 = 0
                islw_fusionL(A,TA,R),
                (error1(0);throw('cannot resolve ambiguities')).
```

La fusion effective s'effectue toujours deux par deux. De la liste précédente, nous avons extrait l'entête que nous fusionnons récursivement avec les entêtes du reste de la liste. Le résultat de chaque fusion binaire est traduit en spécification pour être réinjecté dans la récurrence.

```
islw_fusionL(A,[],A).
islw_fusionL(A,[B|TB],R) :-
                islw_fusion(A,B,Rt),
                tri(Rt,Rtp),
                islw_fusionL(Rtp,TB,R).
```

Le prédicat islw_fusion $\left(a, b, R^{\uparrow}\right)$ prend les paramètres suivants :

1. Entrée a : un comportement ISL4WComp sous forme canonique ;
2. Entrée b : un autre comportement ISL4WComp sour forme canonique ;
3. Sortie R : le résultat de la fusion des comportements a et b.

La fusion consiste d'abord à transformer la syntaxe binaire des spécifications des greffons en listes n-aires. C'est ce que fait le prédicat tr. Ensuite, on transforme toutes les séquences présentes en couples de concurrences *par* de niveau différent. Typiquement, une séquence a puis b est transformée en deux concurrences a_0 de niveau 0 et b_1 de niveau 1. On complète ensuite les comportements pour avoir une forme canonique *par* - if - comportements finals. Enfin, on concatène les listes de concurrences des deux comportements transformés pour ne plus former qu'un. La fusion (en place) va maintenant commencer : prédicat $rr1$.
Nous mettons à zéro le nombre d'erreurs qui peuvent être résolues par la fusion de délégation. Si ce nombre reste à zéro, alors la fusion n'est pas ambigüe.

```
islw_fusion(A,B,R) :-
    retractall(error1(_)),assert(error1(0)), % error1 = 0
    tr(A,Ar),tr(B,Br),  % 1) new syntax
    u(Ar,A1),u(Br,B1),  % 2) remove seq
    j(A1,A2),j(B1,B2),  % 3) canonical form : add missing par,if
    append(A2,B2,AB),
    rr1(AB,[],R),
    error1(0);throw('cannot merge different operators unambigously').
```

Le prédicat *rr*1 fusionne récursivement la première concurrence avec un accumulateur a qui est vide au départ et prend à chaque récurrence le résultat de la fusion courante. Une fois que toutes les concurrences ont été fusionnées avec l'accumulateur, ce dernier contient le résultat final de la fusion.

```
rr1([par(L0,P0)|TP], A, ST) :-
    rr2(par(L0,P0), A, ST1),
    rr1(TP,ST1, ST).
rr1([],R,R).
```

Le prédicat *rr*2 a pour rôle de fusionner une concurrence avec une liste de concurrences déjà fusionnées. Il se compose de deux parties : la fusion de deux opérateurs *par* de même niveau et la fusion de deux opérateurs *par* de niveaux différents.
La première règle fusionne deux concurrences du même niveau. Le résultat est une unique concurrence dont les sous-comportements (condition et comportements finals) ont été également fusionnée.

```
rr2(                    % HYP: same level
      par(L,P1),        % INPUT:  par to be merged
      [par(L,P0)|TQ],   % INPUT:  merged list
      ST                % OUTPUT: solution
  ) :-
    fpar(par(L,P1), par(L,P0) ,par(L,T)), % merge par
    fif1(T,if(Cond,Actions)),             % merge resulting if-list
    fleaf1(Actions,[],AM),                % merge resulting action-list
    S=par(L,[if(Cond,AM)]),               % S=Canonical Form
    rr2(S, TQ, ST).                       % merge S recursively

rr2(                    % HYP: different levels
      par(L0,P1),       % INPUT: par to be merged (level L0)
      [par(L1,P0)|TQ], % INPUT: merged list (level L1)
      ST                % SOLUTION
  ) :-
    rr2(par(L0,P1), TQ, ST1),
    insert([par(L1,P0)],ST1,ST).

rr2(par(L,P1),[],[S]) :-
    fpar(par(L,P1), par(_,[]) ,par(L,T)),
    fif1(T,if(Cond,Actions)),
    fleaf1(Actions,[],AM),
    S=par(L,[if(Cond,AM)]).
```

Le code suivant permet d'insérer un *par* de niveau i au bon endroit dans une liste ordonnée de *par*.

```
insert(X,[],X).
insert(X,L,R) :-
    X=[par(L1,_)],
    L=[par(L2,A2)|TL],
```

```
    ((
       L1>L2, insert(X,TL,R1), append([par(L2,A2)],R1,R) ;
              append(X,L,R)
    )).
```

C.4.2 Exemples d'utilisation

C.4.2.1 Fusionner deux spécifications

Prenons l'exemple de deux spécifications de greffon :

```
islw_fusion(
    seq(a,b)      ,
    if(c,a,nop)   ,
  R),
tri0(R,S).
```

```
R = [   par(1, [if([true]      , [[leaf(b)]                ])]),
        par(0, [if([c, not(c)], [[leaf(a)], [leaf(nop)]])])   ],
S = seq(if(c, a, nop), b) .
```

C.4.2.2 Fusionner une liste de spécifications

L'opération c ne peut pas fusionner avec l'opération d.

```
?- islw_fusion0([op(c),if(t,del(e),nop),seq(op(d),op(c))],R).
ERROR: Unhandled exception:
  Unknown message: cannot resolve merged-operation ambiguities
```

Aucune ambiguité dans l'exemple ci-dessous :

```
?- islw_fusion0([op(c),if(t,del(e),nop),seq(op(c),op(e))],R).
R = seq(if(t, del(e), op(c)), op(e)) .
```

C.5 Greffon BSL

L'implémentation des greffons BSL s'appuie également sur des règles logiques Prolog.

C.5.1 Règles en Prolog

Soit $\vec{a} = [a_1, a_2, a_3]$. Chaque opérateur a une priorité de fusion, c'est-à-dire qu'il va dominer les opérateurs de priorité plus faibles.

Priorité	Notation mathématique	Notation Prolog
1	$a(\vec{a})$	$a([a_1, a_2, a_3])$
2	$o(\vec{a})$	$o([a_1, a_2, a_3])$
3	$conc(\vec{a})$	$conc([a_1, a_2, a_3])$
-	e	$e(name)$
-	e^p	$e(name, p)$

Tableau C.3 – Tableau des priorités et de la notation Prolog des opérateurs

C.5.2 Transformation composants/opérateurs

La transformation entre les spécifications et le modèle de calcul se fait dans les deux sens. Nous décrivons dans cette section les deux transformations.

C.5.2.1 Des spécifications au modèle

L'implémentation actuelle prend directement les spécifications et la mise sous forme canonique comprend déjà la transformation spécification/modèle.

C.5.2.2 Du modèle aux spécifications

La traduction du modèle vers les spécifications au niveau du greffon est très simple. Elle consiste à concaténer les trois listes correspondant aux trois catégories d'opérateurs.

```
bsl_tr(can(A,B,C), R) :-
    append([A],[B],AB), append(AB,[C],R).
```

C.6 Règles de fusion BSL

C.6.1 Prédicat bsl_fa

Le prédicat $bsl\_fa(\vec{a}, \vec{b}, S^{\uparrow})$ est permet de fusionner deux listes d'événements arbitrés. Chaque événement a une priorité, les listes sont ordonnées. L'hypothèse est que les listes $\vec{a}$ et $\vec{b}$ sont déjà ordonnées.

```
bsl_fa( a(L1) , a(L2) , S ) :-
  mmerge(L1, L2, L), S=a(L).
```

Le prédicat *mmerge* fusionne les listes triées $\vec{a}$ et $\vec{b}$. Il trie également la liste finale en fonction des priorités des événements.

```
mmerge([],X,X).
mmerge(X,[],X).
mmerge([e(Dx,Ix)|Xs],[e(Dy,Iy)|Ys],[e(Dx,Ix)|Zs]) :-
    Ix<Iy,!,
    mmerge(Xs,[e(Dy,Iy)|Ys],Zs).
mmerge([e(Dx,Ix)|Xs],[e(Dy,Iy)|Ys],[e(Dy,Iy)|Zs]) :-
    Ix>Iy,
    mmerge([e(Dx,Ix)|Xs],Ys,Zs).
```

Cependant, il faut savoir que cette fusion peut échouer lorsque deux événements différents ont la même priorité.

```
mmerge([e(Dx,Ix)|Xs],[e(Dy,Iy)|Ys],[e(Dx,Ix)|Zs]) :- % idempotence
    Ix=Iy,!, (Dx=Dy; writeln('error cannot merge'),fail),
    mmerge(Xs,Ys,Zs).
```

C.6.2 Prédicat bsl_fo

Le prédicat $bsl\_fo(\vec{a}, \vec{b}, S^{\uparrow})$ permet de fusionner deux listes d'événements ordonnés dans le temps. Chaque événement est accompagné d'un numéro de passage. La fusion de ces deux listes est similaire à celle de l'arbitrage vue dans la section précédente.

```
bsl_fo( o(L1) , o(L2) , S ) :-
  mmergeo(L1, L2, L), S=o(L).
```

Le prédicat *mmergeo* permet de fusionner les listes. Une erreur apparaît lorsque des événements ont le même numéro de passage.

```
mmergeo([],X,X).
mmergeo(X,[],X).
mmergeo([e(Dx,Ix)|Xs],[e(Dy,Iy)|Ys],[e(Dx,Ix)|Zs]) :-
    Ix<Iy,!,
    mmerge(Xs,[e(Dy,Iy)|Ys],Zs).
mmergeo([e(Dx,Ix)|Xs],[e(Dy,Iy)|Ys],[e(Dy,Iy)|Zs]) :-
    Ix>Iy,
    mmergeo([e(Dx,Ix)|Xs],Ys,Zs).
mmergeo([e(Dx,Ix)|Xs],[e(Dy,Iy)|Ys],[e(Dx,Ix)|Zs]) :- % idempotence
    Ix=Iy,!, (Dx=Dy; writeln('error cannot merge'),fail),
    mmergeo(Xs,Ys,Zs).
```

C.6.3 Prédicat bsl_fc

Le prédicat $bsl\_fc(\vec{a},\vec{b},S^{\uparrow})$ permet de fusionner deux listes d'événements concurrents. Pour y parvenir, il concatène les deux listes et enlève les doublons de la liste résultante.

```
bsl_fc( conc(A) , conc(B) , conc(R) ) :-
        colle(A,B,R1), no_doublons(R1,R).
```

Pour fonctionner, ce prédicat s'appuie sur différents prédicats utilitaires. Le prédicat suivant permet de tester si un élément est membre d'une liste.

```
membre(Elem,[Elem|_]).
membre(Elem,[_|Suivants]):-membre(Elem,Suivants).
```

Le prédicat suivant permet de retirer les doublons d'une liste.

```
no_doublons([Tete|Queue],ResultSansDoublons):-
        membre(Tete,Queue),no_doublons(Queue,ResultSansDoublons).
no_doublons([Tete|Queue],[Tete|ResultSansDoublons]):-
        not(membre(Tete,Queue)),no_doublons(Queue,ResultSansDoublons).
no_doublons([],[]).
```

Enfin, ce dernier prédicat est équivalent au prédicat *append* qui permet de concaténer deux listes.

```
colle([X|SuiteDesX],ListeDesY,[X|SuiteDesXpuisLesY]) :-
   colle(SuiteDesX,ListeDesY,SuiteDesXpuisLesY).
colle([],Liste,Liste).
```

C.6.4 Synthèse de la forme canonique

Le calcul complet de la forme canonique se fait en deux étapes. D'abord, à partir d'une liste de déclarations d'opérateurs, on classe les opérateurs de concurrence, d'arbitrage et de découpage. Ensuite, on résout les conflits entre les déclarations ainsi classées. Le prédicat *canon*0 prend les paramètres suivants :

1. Entrée $\vec{a}$: comportement
2. Sortie $\vec{S}$: forme canonique

```
canon0(A, R) :-
   canon(A, [conc([])],[o([])],[a([])], R1),
   canon1(R1,R).
```

La classification des opérateurs est faite par le prédicat *canon*. Ce sont principalement des opérations sur les listes.

```
canon([conc(A)|TA], C,D,E, R) :-
    append([conc(A)],C,C1),
    canon(TA, C1,D,E, can(Ca,Da,Ea)),
    append(D,Da,Db),
    append(E,Ea,Eb),
    R=can(Ca,Db,Eb).
canon([o(A)|TA], C,D,E, R) :-
    append([o(A)],D,D1),
    canon(TA, C,D1,E, can(Ca,Da,Ea)),
    append(C,Ca,Cb),
    append(E,Ea,Eb),
    R=can(Cb,Da,Eb).
canon([a(A)|TA], C,D,E, R) :-
    append([a(A)],E,E1),
    canon(TA, C,D,E1, can(Ca,Da,Ea)),
    append(C,Ca,Cb),
    append(D,Da,Db),
    R=can(Cb,Db,Ea).
canon([], C,D,E, can(C,D,E)).
```

Le prédicat *canon*1 est responsable de deux principales actions.

- Il appelle les prédicats de fusion pour chaque groupe d'opérateur. Il résout alors à ce moment-là les conflits à l'intérieur de chaque groupe (prédicat *canon*_...);
- Ensuite, il résout les conflits entre les différents groupes. En effet, l'opérateur d'arbitrage est prioritaire sur les deux autres opérateurs. Les événements qui s'y trouvent ne doivent pas apparaître dans les autres listes. Il en est de même pour l'opérateur de découpage par rapport à la concurrence.

```
canon1(can([conc(A)|TA],[o(B)|TB],[a(C)|TC]),R) :-
    canon_c(conc(A),TA,CONC),       % local conflicts
    canon_o(o(B),TB,O),
    canon_a(a(C),TC,ARB),
    resolve_oc(O,CONC,RCONC),       % remove el. in O from CONC
    resolve_ao(ARB,O,RO),           % remove el. in A from O
    resolve_ac(ARB,RCONC,RRCONC), % remove el. in A from CONC
    R=can(RRCONC,RO,ARB).           % Solution
```

La résolution de conflits locaux consiste à parcourir la liste des opérateurs de même nature et d'appeler le prédicat de résolution adéquat.

```
canon_a(D,[B|TB],R) :-
    bsl_fa(D,B,E),          % conflict solving for arbitration
    canon_a(E,TB,R).
canon_a(R,_,R).

canon_o(D,[B|TB],R) :-
    bsl_fo(D,B,E),          % conflict solving for time-cutting
    canon_o(E,TB,R).
canon_o(R,_,R).

canon_c(D,[B|TB],R) :-
    bsl_fc(D,B,E),          % conflict solving for concurrency
    canon_c(E,TB,R).
canon_c(R,_,R).
```

Enfin, nous détaillons le prédicat $resolve\_(p, c, S^{\uparrow})$ qui résout globalement les conflits entre les opérateurs. p est l'opérateur de découpage portant sur une liste l_p d'événements. c est l'opérateur de concurrence portant sur une liste l_c d'événements. Ces deux listes sont localement fusionnées. D'abord, c'est un prédicat récursif, par conséquent nous définissons une règle de terminaison qui copie la solution :

```
resolve_oc(o([]),RC,RC).
```

Le calcul se fait à partir de deux règles. La première règle prend le premier événement $p_1 \in l_p$ de la liste de découpage. On traite le cas où p_1 se trouve à l'intérieur de la liste l_c. (Au passage, l'événement à l'intérieur d'une liste de concurrence n'a pas d'information rattachée.) On supprime l'élément de la liste l_c et continue la récursion sur la liste l_p privée de l'événement correspondant à p_1.

```
resolve_oc(o([e(D,_)|TA]),conc(C),RC) :-
    membre(e(D),C),!, supprime(e(D),C,RC1),
    resolve_oc(o(TA),conc(RC1),RC).
```

Pour la seconde règle, lorsque p_1 ne se trouve pas dans l_c, alors il n'y a rien à faire. On continue la récursion sur l_p privée de p_1.

```
resolve_oc(o([e(D,ID)|TA]),conc(C),RC) :-
    not(membre(e(D,ID),C)),
    resolve_oc(o(TA),conc(C),RC).
```

Voici les prédicats restant pour les deux autres opérateurs. L'explication reste la même que précédemment.

```
resolve_ac(a([]),RC,RC).
resolve_ac(a([e(D,_)|TA]),conc(C),RC) :-
    membre(e(D),C),!, supprime(e(D),C,RC1),
    resolve_ac(a(TA),conc(RC1),RC).
resolve_ac(a([e(D,ID)|TA]),conc(C),RC) :-
    not(membre(e(D,ID),C)),
    resolve_ac(a(TA),conc(C),RC).

resolve_ao(a([]),RO,RO).
resolve_ao(a([e(D,_)|TA]),o(O),RO) :-
    membre(e(D,ID),O),!, supprime(e(D,ID),O,RO1),
    resolve_ao(a(TA),o(RO1),RO).
resolve_ao(a([e(D,ID)|TA]),o(O),RO) :-
    not(membre(e(D,ID),O)),
    resolve_ao(a(TA),o(O),RO).
```

Le prédicat suivant est un prédicat utilitaire pour supprimer un élément d'une liste.

```
supprime(X,[X|ResteListeInitiale],ListeSansX) :-
        supprime(X,ResteListeInitiale,ListeSansX).
supprime(X,[PasX|ResteListeInitiale],[PasX|ListeSansX]) :-
        X\=PasX,supprime(X,ResteListeInitiale,ListeSansX).
supprime(_,[],[]).
```

C.7 Prédicat de composition/fusion BSL

Dans cette section, nous présentons le prédicat de fusion et donnons un exemple d'utilisation.

C.7.1 Prédicat bsl_fusion0

Le prédicat bsl_fusion0 $\left(a, b, R^{\uparrow}\right)$ prend les paramètres suivants :

1. Entrée a : spécifications BSL ;
2. Entrée b : spécifications BSL ;
3. Sortie R : le résultat de la fusion des spécifications a et b.

Le prédicat de fusion *bsl_fusion*0 traduit les spécifications A et B en formes canoniques et appelle le prédicat *bsl_fusion* pour effectuer la fusion.

```
bsl_fusion0(A,B,C) :-
    canon0(A,Ac),canon0(B,Bc),bsl_fusion(Ac,Bc,C).
```

C.7.2 Prédicat bsl_fusion

Le prédicat bsl_fusion $\left(a, b, R^{\uparrow}\right)$ prend les paramètres suivants :

1. Entrée a : un comportement BSL sous forme canonique ;
2. Entrée b : un autre comportement BSL sour forme canonique ;
3. Sortie R : le résultat de la fusion des comportements a et b.

```
bsl_fusion(can(A,B,C),can(D,E,F),R) :-
    append([A],[D],AD),append([B],[E],BE),append([C],[F],CF),
    canon1(can(AD,BE,CF),R).
```

La fusion consiste à utiliser le prédicat *canon*1 pour effectuer la fusion locale et globale (voir le détail à la page 205).

C.7.3 Exemple d'utilisation

C.7.3.1 Utilisation des prédicats *bsl_fusion* et *canon*0

Prenons l'exemple de deux spécifications de greffon : la première spécification décrit deux événements concurrents identiques e_m. Elle déclare également l'arbitrage de l'événement e_n^1 ayant la priorité 1.

```
?- canon0([conc([e(m),e(m)]),a([e(n,1)])],R).
R = can(conc([e(m)]), o([]), a([e(n, 1)])) .
```

La seconde spécification décrit deux événements concurrents e_t et e_m. Elle déclare aussi deux arbitrages déjà triés e_m^2 et e_y^3.

```
?- canon0([conc([e(t),e(m)]),a([e(m,2),e(y,3)])],R).
R = can(conc([e(t)]), o([]), a([e(m, 2), e(y, 3)])) .
```

Nous appliquons le prédicat de fusion aux deux résultats précédents, et nous obtenons ceci :

```
?- bsl_fusion(
       can(conc([e(m)]), o([]), a([e(n, 1)])),
       can(conc([e(t)]), o([]), a([e(m, 2), e(y, 3)])),
       R).
R = can(conc([e(t)]), o([]), a([e(n, 1), e(m, 2), e(y, 3)])) .
```

Nous pouvons retrouver son écriture sous la forme d'une spécification de greffon :

```
?- bsl_tr(can(conc([e(t)]), o([]), a([e(m, 2), e(y, 3)])),R).
R = [conc([e(t)]), o([]), a([e(m, 2), e(y, 3)])].
```

C.7.3.2 Utilisation du prédicat $bsl\_fusion0$

Voici un exemple de fusion de la diffusion d'événement m. Il est spécifié à la fois comme étant un événement concurrentiel et un événement arbitré. Le résultat de la fusion de ces deux spécifications est un arbitrage.

```
?- bsl_fusion0([conc([e(m)])],[a([e(m,1)])],R).
R = can(conc([]), o([]), a([e(m, 1)])) .
```

On peut inverser les spécifications et le résultat reste le même, car la fusion est commutative.

```
?- bsl_fusion0([a([e(m,1)])],[conc([e(m)])],R).
R = can(conc([]), o([]), a([e(m, 1)])) .
```

Annexe D

Calculs pour l'associativité

Sommaire

D.1 Calculs de toutes les possibilités

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
0	$(del(a) \oplus del(a)) \oplus del(a) =$	$del(a)$	$del(a) \oplus (del(a) \oplus del(a)) =$	$del(a)$	OK
1	$(del(b) \oplus del(a)) \oplus del(a) =$	$error$	$del(b) \oplus (del(a) \oplus del(a)) =$	$error$	OK
2	$(del(c) \oplus del(a)) \oplus del(a) =$	$error$	$del(c) \oplus (del(a) \oplus del(a)) =$	$error$	OK
3	$(call \oplus del(a)) \oplus del(a) =$	$del(a)$	$call \oplus (del(a) \oplus del(a)) =$	$del(a)$	OK
4	$(nop \oplus del(a)) \oplus del(a) =$	$del(a)$	$nop \oplus (del(a) \oplus del(a)) =$	$del(a)$	OK
5	$(op(a) \oplus del(a)) \oplus del(a) =$	$del(a)$	$op(a) \oplus (del(a) \oplus del(a)) =$	$del(a)$	OK
6	$(op(b) \oplus del(a)) \oplus del(a) =$	$del(a)$	$op(b) \oplus (del(a) \oplus del(a)) =$	$del(a)$	OK
7	$(op(c) \oplus del(a)) \oplus del(a) =$	$del(a)$	$op(c) \oplus (del(a) \oplus del(a)) =$	$del(a)$	OK
8	$(del(a) \oplus del(b)) \oplus del(a) =$	$error$	$del(a) \oplus (del(b) \oplus del(a)) =$	$error$	OK
9	$(del(b) \oplus del(b)) \oplus del(a) =$	$error$	$del(b) \oplus (del(b) \oplus del(a)) =$	$error$	OK
10	$(del(c) \oplus del(b)) \oplus del(a) =$	$error$	$del(c) \oplus (del(b) \oplus del(a)) =$	$error$	OK
11	$(call \oplus del(b)) \oplus del(a) =$	$error$	$call \oplus (del(b) \oplus del(a)) =$	$error$	OK
12	$(nop \oplus del(b)) \oplus del(a) =$	$error$	$nop \oplus (del(b) \oplus del(a)) =$	$error$	OK
13	$(op(a) \oplus del(b)) \oplus del(a) =$	$error$	$op(a) \oplus (del(b) \oplus del(a)) =$	$error$	OK
14	$(op(b) \oplus del(b)) \oplus del(a) =$	$error$	$op(b) \oplus (del(b) \oplus del(a)) =$	$error$	OK
15	$(op(c) \oplus del(b)) \oplus del(a) =$	$error$	$op(c) \oplus (del(b) \oplus del(a)) =$	$error$	OK
16	$(del(a) \oplus del(c)) \oplus del(a) =$	$error$	$del(a) \oplus (del(c) \oplus del(a)) =$	$error$	OK
17	$(del(b) \oplus del(c)) \oplus del(a) =$	$error$	$del(b) \oplus (del(c) \oplus del(a)) =$	$error$	OK
18	$(del(c) \oplus del(c)) \oplus del(a) =$	$error$	$del(c) \oplus (del(c) \oplus del(a)) =$	$error$	OK
19	$(call \oplus del(c)) \oplus del(a) =$	$error$	$call \oplus (del(c) \oplus del(a)) =$	$error$	OK
20	$(nop \oplus del(c)) \oplus del(a) =$	$error$	$nop \oplus (del(c) \oplus del(a)) =$	$error$	OK
21	$(op(a) \oplus del(c)) \oplus del(a) =$	$error$	$op(a) \oplus (del(c) \oplus del(a)) =$	$error$	OK
22	$(op(b) \oplus del(c)) \oplus del(a) =$	$error$	$op(b) \oplus (del(c) \oplus del(a)) =$	$error$	OK
23	$(op(c) \oplus del(c)) \oplus del(a) =$	$error$	$op(c) \oplus (del(c) \oplus del(a)) =$	$error$	OK
24	$(del(a) \oplus call) \oplus del(a) =$	$del(a)$	$del(a) \oplus (call \oplus del(a)) =$	$del(a)$	OK
25	$(del(b) \oplus call) \oplus del(a) =$	$error$	$del(b) \oplus (call \oplus del(a)) =$	$error$	OK
26	$(del(c) \oplus call) \oplus del(a) =$	$error$	$del(c) \oplus (call \oplus del(a)) =$	$error$	OK
27	$(call \oplus call) \oplus del(a) =$	$del(a)$	$call \oplus (call \oplus del(a)) =$	$del(a)$	OK
28	$(nop \oplus call) \oplus del(a) =$	$del(a)$	$nop \oplus (call \oplus del(a)) =$	$del(a)$	OK
29	$(op(a) \oplus call) \oplus del(a) =$	$del(a)$	$op(a) \oplus (call \oplus del(a)) =$	$del(a)$	OK
30	$(op(b) \oplus call) \oplus del(a) =$	$del(a)$	$op(b) \oplus (call \oplus del(a)) =$	$del(a)$	OK
31	$(op(c) \oplus call) \oplus del(a) =$	$del(a)$	$op(c) \oplus (call \oplus del(a)) =$	$del(a)$	OK
32	$(del(a) \oplus nop) \oplus del(a) =$	$del(a)$	$del(a) \oplus (nop \oplus del(a)) =$	$del(a)$	OK
33	$(del(b) \oplus nop) \oplus del(a) =$	$error$	$del(b) \oplus (nop \oplus del(a)) =$	$error$	OK
34	$(del(c) \oplus nop) \oplus del(a) =$	$error$	$del(c) \oplus (nop \oplus del(a)) =$	$error$	OK
35	$(call \oplus nop) \oplus del(a) =$	$del(a)$	$call \oplus (nop \oplus del(a)) =$	$del(a)$	OK
36	$(nop \oplus nop) \oplus del(a) =$	$del(a)$	$nop \oplus (nop \oplus del(a)) =$	$del(a)$	OK
37	$(op(a) \oplus nop) \oplus del(a) =$	$del(a)$	$op(a) \oplus (nop \oplus del(a)) =$	$del(a)$	OK
38	$(op(b) \oplus nop) \oplus del(a) =$	$del(a)$	$op(b) \oplus (nop \oplus del(a)) =$	$del(a)$	OK
39	$(op(c) \oplus nop) \oplus del(a) =$	$del(a)$	$op(c) \oplus (nop \oplus del(a)) =$	$del(a)$	OK
40	$(del(a) \oplus op(a)) \oplus del(a) =$	$del(a)$	$del(a) \oplus (op(a) \oplus del(a)) =$	$del(a)$	OK
41	$(del(b) \oplus op(a)) \oplus del(a) =$	$error$	$del(b) \oplus (op(a) \oplus del(a)) =$	$error$	OK
42	$(del(c) \oplus op(a)) \oplus del(a) =$	$error$	$del(c) \oplus (op(a) \oplus del(a)) =$	$error$	OK
43	$(call \oplus op(a)) \oplus del(a) =$	$del(a)$	$call \oplus (op(a) \oplus del(a)) =$	$del(a)$	OK
44	$(nop \oplus op(a)) \oplus del(a) =$	$del(a)$	$nop \oplus (op(a) \oplus del(a)) =$	$del(a)$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
45	$(op(a) \oplus op(a)) \oplus del(a) =$	$del(a)$	$op(a) \oplus (op(a) \oplus del(a)) =$	$del(a)$	OK
46	$(op(b) \oplus op(a)) \oplus del(a) =$	$del(a)$	$op(b) \oplus (op(a) \oplus del(a)) =$	$del(a)$	OK
47	$(op(c) \oplus op(a)) \oplus del(a) =$	$del(a)$	$op(c) \oplus (op(a) \oplus del(a)) =$	$del(a)$	OK
48	$(del(a) \oplus op(b)) \oplus del(a) =$	$del(a)$	$del(a) \oplus (op(b) \oplus del(a)) =$	$del(a)$	OK
49	$(del(b) \oplus op(b)) \oplus del(a) =$	$error$	$del(b) \oplus (op(b) \oplus del(a)) =$	$error$	OK
50	$(del(c) \oplus op(b)) \oplus del(a) =$	$error$	$del(c) \oplus (op(b) \oplus del(a)) =$	$error$	OK
51	$(call \oplus op(b)) \oplus del(a) =$	$del(a)$	$call \oplus (op(b) \oplus del(a)) =$	$del(a)$	OK
52	$(nop \oplus op(b)) \oplus del(a) =$	$del(a)$	$nop \oplus (op(b) \oplus del(a)) =$	$del(a)$	OK
53	$(op(a) \oplus op(b)) \oplus del(a) =$	$del(a)$	$op(a) \oplus (op(b) \oplus del(a)) =$	$del(a)$	OK
54	$(op(b) \oplus op(b)) \oplus del(a) =$	$del(a)$	$op(b) \oplus (op(b) \oplus del(a)) =$	$del(a)$	OK
55	$(op(c) \oplus op(b)) \oplus del(a) =$	$del(a)$	$op(c) \oplus (op(b) \oplus del(a)) =$	$del(a)$	OK
56	$(del(a) \oplus op(c)) \oplus del(a) =$	$del(a)$	$del(a) \oplus (op(c) \oplus del(a)) =$	$del(a)$	OK
57	$(del(b) \oplus op(c)) \oplus del(a) =$	$error$	$del(b) \oplus (op(c) \oplus del(a)) =$	$error$	OK
58	$(del(c) \oplus op(c)) \oplus del(a) =$	$error$	$del(c) \oplus (op(c) \oplus del(a)) =$	$error$	OK
59	$(call \oplus op(c)) \oplus del(a) =$	$del(a)$	$call \oplus (op(c) \oplus del(a)) =$	$del(a)$	OK
60	$(nop \oplus op(c)) \oplus del(a) =$	$del(a)$	$nop \oplus (op(c) \oplus del(a)) =$	$del(a)$	OK
61	$(op(a) \oplus op(c)) \oplus del(a) =$	$del(a)$	$op(a) \oplus (op(c) \oplus del(a)) =$	$del(a)$	OK
62	$(op(b) \oplus op(c)) \oplus del(a) =$	$del(a)$	$op(b) \oplus (op(c) \oplus del(a)) =$	$del(a)$	OK
63	$(op(c) \oplus op(c)) \oplus del(a) =$	$del(a)$	$op(c) \oplus (op(c) \oplus del(a)) =$	$del(a)$	OK
64	$(del(a) \oplus del(a)) \oplus del(b) =$	$error$	$del(a) \oplus (del(a) \oplus del(b)) =$	$error$	OK
65	$(del(b) \oplus del(a)) \oplus del(b) =$	$error$	$del(b) \oplus (del(a) \oplus del(b)) =$	$error$	OK
66	$(del(c) \oplus del(a)) \oplus del(b) =$	$error$	$del(c) \oplus (del(a) \oplus del(b)) =$	$error$	OK
67	$(call \oplus del(a)) \oplus del(b) =$	$error$	$call \oplus (del(a) \oplus del(b)) =$	$error$	OK
68	$(nop \oplus del(a)) \oplus del(b) =$	$error$	$nop \oplus (del(a) \oplus del(b)) =$	$error$	OK
69	$(op(a) \oplus del(a)) \oplus del(b) =$	$error$	$op(a) \oplus (del(a) \oplus del(b)) =$	$error$	OK
70	$(op(b) \oplus del(a)) \oplus del(b) =$	$error$	$op(b) \oplus (del(a) \oplus del(b)) =$	$error$	OK
71	$(op(c) \oplus del(a)) \oplus del(b) =$	$error$	$op(c) \oplus (del(a) \oplus del(b)) =$	$error$	OK
72	$(del(a) \oplus del(b)) \oplus del(b) =$	$error$	$del(a) \oplus (del(b) \oplus del(b)) =$	$error$	OK
73	$(del(b) \oplus del(b)) \oplus del(b) =$	$del(b)$	$del(b) \oplus (del(b) \oplus del(b)) =$	$del(b)$	OK
74	$(del(c) \oplus del(b)) \oplus del(b) =$	$error$	$del(c) \oplus (del(b) \oplus del(b)) =$	$error$	OK
75	$(call \oplus del(b)) \oplus del(b) =$	$del(b)$	$call \oplus (del(b) \oplus del(b)) =$	$del(b)$	OK
76	$(nop \oplus del(b)) \oplus del(b) =$	$del(b)$	$nop \oplus (del(b) \oplus del(b)) =$	$del(b)$	OK
77	$(op(a) \oplus del(b)) \oplus del(b) =$	$del(b)$	$op(a) \oplus (del(b) \oplus del(b)) =$	$del(b)$	OK
78	$(op(b) \oplus del(b)) \oplus del(b) =$	$del(b)$	$op(b) \oplus (del(b) \oplus del(b)) =$	$del(b)$	OK
79	$(op(c) \oplus del(b)) \oplus del(b) =$	$del(b)$	$op(c) \oplus (del(b) \oplus del(b)) =$	$del(b)$	OK
80	$(del(a) \oplus del(c)) \oplus del(b) =$	$error$	$del(a) \oplus (del(c) \oplus del(b)) =$	$error$	OK
81	$(del(b) \oplus del(c)) \oplus del(b) =$	$error$	$del(b) \oplus (del(c) \oplus del(b)) =$	$error$	OK
82	$(del(c) \oplus del(c)) \oplus del(b) =$	$error$	$del(c) \oplus (del(c) \oplus del(b)) =$	$error$	OK
83	$(call \oplus del(c)) \oplus del(b) =$	$error$	$call \oplus (del(c) \oplus del(b)) =$	$error$	OK
84	$(nop \oplus del(c)) \oplus del(b) =$	$error$	$nop \oplus (del(c) \oplus del(b)) =$	$error$	OK
85	$(op(a) \oplus del(c)) \oplus del(b) =$	$error$	$op(a) \oplus (del(c) \oplus del(b)) =$	$error$	OK
86	$(op(b) \oplus del(c)) \oplus del(b) =$	$error$	$op(b) \oplus (del(c) \oplus del(b)) =$	$error$	OK
87	$(op(c) \oplus del(c)) \oplus del(b) =$	$error$	$op(c) \oplus (del(c) \oplus del(b)) =$	$error$	OK
88	$(del(a) \oplus call) \oplus del(b) =$	$error$	$del(a) \oplus (call \oplus del(b)) =$	$error$	OK
89	$(del(b) \oplus call) \oplus del(b) =$	$del(b)$	$del(b) \oplus (call \oplus del(b)) =$	$del(b)$	OK
90	$(del(c) \oplus call) \oplus del(b) =$	$error$	$del(c) \oplus (call \oplus del(b)) =$	$error$	OK
91	$(call \oplus call) \oplus del(b) =$	$del(b)$	$call \oplus (call \oplus del(b)) =$	$del(b)$	OK
92	$(nop \oplus call) \oplus del(b) =$	$del(b)$	$nop \oplus (call \oplus del(b)) =$	$del(b)$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
93	$(op(a) \oplus call) \oplus del(b) =$	$del(b)$	$op(a) \oplus (call \oplus del(b)) =$	$del(b)$	OK
94	$(op(b) \oplus call) \oplus del(b) =$	$del(b)$	$op(b) \oplus (call \oplus del(b)) =$	$del(b)$	OK
95	$(op(c) \oplus call) \oplus del(b) =$	$del(b)$	$op(c) \oplus (call \oplus del(b)) =$	$del(b)$	OK
96	$(del(a) \oplus nop) \oplus del(b) =$	$error$	$del(a) \oplus (nop \oplus del(b)) =$	$error$	OK
97	$(del(b) \oplus nop) \oplus del(b) =$	$del(b)$	$del(b) \oplus (nop \oplus del(b)) =$	$del(b)$	OK
98	$(del(c) \oplus nop) \oplus del(b) =$	$error$	$del(c) \oplus (nop \oplus del(b)) =$	$error$	OK
99	$(call \oplus nop) \oplus del(b) =$	$del(b)$	$call \oplus (nop \oplus del(b)) =$	$del(b)$	OK
100	$(nop \oplus nop) \oplus del(b) =$	$del(b)$	$nop \oplus (nop \oplus del(b)) =$	$del(b)$	OK
101	$(op(a) \oplus nop) \oplus del(b) =$	$del(b)$	$op(a) \oplus (nop \oplus del(b)) =$	$del(b)$	OK
102	$(op(b) \oplus nop) \oplus del(b) =$	$del(b)$	$op(b) \oplus (nop \oplus del(b)) =$	$del(b)$	OK
103	$(op(c) \oplus nop) \oplus del(b) =$	$del(b)$	$op(c) \oplus (nop \oplus del(b)) =$	$del(b)$	OK
104	$(del(a) \oplus op(a)) \oplus del(b) =$	$error$	$del(a) \oplus (op(a) \oplus del(b)) =$	$error$	OK
105	$(del(b) \oplus op(a)) \oplus del(b) =$	$del(b)$	$del(b) \oplus (op(a) \oplus del(b)) =$	$del(b)$	OK
106	$(del(c) \oplus op(a)) \oplus del(b) =$	$error$	$del(c) \oplus (op(a) \oplus del(b)) =$	$error$	OK
107	$(call \oplus op(a)) \oplus del(b) =$	$del(b)$	$call \oplus (op(a) \oplus del(b)) =$	$del(b)$	OK
108	$(nop \oplus op(a)) \oplus del(b) =$	$del(b)$	$nop \oplus (op(a) \oplus del(b)) =$	$del(b)$	OK
109	$(op(a) \oplus op(a)) \oplus del(b) =$	$del(b)$	$op(a) \oplus (op(a) \oplus del(b)) =$	$del(b)$	OK
110	$(op(b) \oplus op(a)) \oplus del(b) =$	$del(b)$	$op(b) \oplus (op(a) \oplus del(b)) =$	$del(b)$	OK
111	$(op(c) \oplus op(a)) \oplus del(b) =$	$del(b)$	$op(c) \oplus (op(a) \oplus del(b)) =$	$del(b)$	OK
112	$(del(a) \oplus op(b)) \oplus del(b) =$	$error$	$del(a) \oplus (op(b) \oplus del(b)) =$	$error$	OK
113	$(del(b) \oplus op(b)) \oplus del(b) =$	$del(b)$	$del(b) \oplus (op(b) \oplus del(b)) =$	$del(b)$	OK
114	$(del(c) \oplus op(b)) \oplus del(b) =$	$error$	$del(c) \oplus (op(b) \oplus del(b)) =$	$error$	OK
115	$(call \oplus op(b)) \oplus del(b) =$	$del(b)$	$call \oplus (op(b) \oplus del(b)) =$	$del(b)$	OK
116	$(nop \oplus op(b)) \oplus del(b) =$	$del(b)$	$nop \oplus (op(b) \oplus del(b)) =$	$del(b)$	OK
117	$(op(a) \oplus op(b)) \oplus del(b) =$	$del(b)$	$op(a) \oplus (op(b) \oplus del(b)) =$	$del(b)$	OK
118	$(op(b) \oplus op(b)) \oplus del(b) =$	$del(b)$	$op(b) \oplus (op(b) \oplus del(b)) =$	$del(b)$	OK
119	$(op(c) \oplus op(b)) \oplus del(b) =$	$del(b)$	$op(c) \oplus (op(b) \oplus del(b)) =$	$del(b)$	OK
120	$(del(a) \oplus op(c)) \oplus del(b) =$	$error$	$del(a) \oplus (op(c) \oplus del(b)) =$	$error$	OK
121	$(del(b) \oplus op(c)) \oplus del(b) =$	$del(b)$	$del(b) \oplus (op(c) \oplus del(b)) =$	$del(b)$	OK
122	$(del(c) \oplus op(c)) \oplus del(b) =$	$error$	$del(c) \oplus (op(c) \oplus del(b)) =$	$error$	OK
123	$(call \oplus op(c)) \oplus del(b) =$	$del(b)$	$call \oplus (op(c) \oplus del(b)) =$	$del(b)$	OK
124	$(nop \oplus op(c)) \oplus del(b) =$	$del(b)$	$nop \oplus (op(c) \oplus del(b)) =$	$del(b)$	OK
125	$(op(a) \oplus op(c)) \oplus del(b) =$	$del(b)$	$op(a) \oplus (op(c) \oplus del(b)) =$	$del(b)$	OK
126	$(op(b) \oplus op(c)) \oplus del(b) =$	$del(b)$	$op(b) \oplus (op(c) \oplus del(b)) =$	$del(b)$	OK
127	$(op(c) \oplus op(c)) \oplus del(b) =$	$del(b)$	$op(c) \oplus (op(c) \oplus del(b)) =$	$del(b)$	OK
128	$(del(a) \oplus del(a)) \oplus del(c) =$	$error$	$del(a) \oplus (del(a) \oplus del(c)) =$	$error$	OK
129	$(del(b) \oplus del(a)) \oplus del(c) =$	$error$	$del(b) \oplus (del(a) \oplus del(c)) =$	$error$	OK
130	$(del(c) \oplus del(a)) \oplus del(c) =$	$error$	$del(c) \oplus (del(a) \oplus del(c)) =$	$error$	OK
131	$(call \oplus del(a)) \oplus del(c) =$	$error$	$call \oplus (del(a) \oplus del(c)) =$	$error$	OK
132	$(nop \oplus del(a)) \oplus del(c) =$	$error$	$nop \oplus (del(a) \oplus del(c)) =$	$error$	OK
133	$(op(a) \oplus del(a)) \oplus del(c) =$	$error$	$op(a) \oplus (del(a) \oplus del(c)) =$	$error$	OK
134	$(op(b) \oplus del(a)) \oplus del(c) =$	$error$	$op(b) \oplus (del(a) \oplus del(c)) =$	$error$	OK
135	$(op(c) \oplus del(a)) \oplus del(c) =$	$error$	$op(c) \oplus (del(a) \oplus del(c)) =$	$error$	OK
136	$(del(a) \oplus del(b)) \oplus del(c) =$	$error$	$del(a) \oplus (del(b) \oplus del(c)) =$	$error$	OK
137	$(del(b) \oplus del(b)) \oplus del(c) =$	$error$	$del(b) \oplus (del(b) \oplus del(c)) =$	$error$	OK
138	$(del(c) \oplus del(b)) \oplus del(c) =$	$error$	$del(c) \oplus (del(b) \oplus del(c)) =$	$error$	OK
139	$(call \oplus del(b)) \oplus del(c) =$	$error$	$call \oplus (del(b) \oplus del(c)) =$	$error$	OK
140	$(nop \oplus del(b)) \oplus del(c) =$	$error$	$nop \oplus (del(b) \oplus del(c)) =$	$error$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
141	$(op(a) \oplus del(b)) \oplus del(c) =$	$error$	$op(a) \oplus (del(b) \oplus del(c)) =$	$error$	OK
142	$(op(b) \oplus del(b)) \oplus del(c) =$	$error$	$op(b) \oplus (del(b) \oplus del(c)) =$	$error$	OK
143	$(op(c) \oplus del(b)) \oplus del(c) =$	$error$	$op(c) \oplus (del(b) \oplus del(c)) =$	$error$	OK
144	$(del(a) \oplus del(c)) \oplus del(c) =$	$error$	$del(a) \oplus (del(c) \oplus del(c)) =$	$error$	OK
145	$(del(b) \oplus del(c)) \oplus del(c) =$	$error$	$del(b) \oplus (del(c) \oplus del(c)) =$	$error$	OK
146	$(del(c) \oplus del(c)) \oplus del(c) =$	$del(c)$	$del(c) \oplus (del(c) \oplus del(c)) =$	$del(c)$	OK
147	$(call \oplus del(c)) \oplus del(c) =$	$del(c)$	$call \oplus (del(c) \oplus del(c)) =$	$del(c)$	OK
148	$(nop \oplus del(c)) \oplus del(c) =$	$del(c)$	$nop \oplus (del(c) \oplus del(c)) =$	$del(c)$	OK
149	$(op(a) \oplus del(c)) \oplus del(c) =$	$del(c)$	$op(a) \oplus (del(c) \oplus del(c)) =$	$del(c)$	OK
150	$(op(b) \oplus del(c)) \oplus del(c) =$	$del(c)$	$op(b) \oplus (del(c) \oplus del(c)) =$	$del(c)$	OK
151	$(op(c) \oplus del(c)) \oplus del(c) =$	$del(c)$	$op(c) \oplus (del(c) \oplus del(c)) =$	$del(c)$	OK
152	$(del(a) \oplus call) \oplus del(c) =$	$error$	$del(a) \oplus (call \oplus del(c)) =$	$error$	OK
153	$(del(b) \oplus call) \oplus del(c) =$	$error$	$del(b) \oplus (call \oplus del(c)) =$	$error$	OK
154	$(del(c) \oplus call) \oplus del(c) =$	$del(c)$	$del(c) \oplus (call \oplus del(c)) =$	$del(c)$	OK
155	$(call \oplus call) \oplus del(c) =$	$del(c)$	$call \oplus (call \oplus del(c)) =$	$del(c)$	OK
156	$(nop \oplus call) \oplus del(c) =$	$del(c)$	$nop \oplus (call \oplus del(c)) =$	$del(c)$	OK
157	$(op(a) \oplus call) \oplus del(c) =$	$del(c)$	$op(a) \oplus (call \oplus del(c)) =$	$del(c)$	OK
158	$(op(b) \oplus call) \oplus del(c) =$	$del(c)$	$op(b) \oplus (call \oplus del(c)) =$	$del(c)$	OK
159	$(op(c) \oplus call) \oplus del(c) =$	$del(c)$	$op(c) \oplus (call \oplus del(c)) =$	$del(c)$	OK
160	$(del(a) \oplus nop) \oplus del(c) =$	$error$	$del(a) \oplus (nop \oplus del(c)) =$	$error$	OK
161	$(del(b) \oplus nop) \oplus del(c) =$	$error$	$del(b) \oplus (nop \oplus del(c)) =$	$error$	OK
162	$(del(c) \oplus nop) \oplus del(c) =$	$del(c)$	$del(c) \oplus (nop \oplus del(c)) =$	$del(c)$	OK
163	$(call \oplus nop) \oplus del(c) =$	$del(c)$	$call \oplus (nop \oplus del(c)) =$	$del(c)$	OK
164	$(nop \oplus nop) \oplus del(c) =$	$del(c)$	$nop \oplus (nop \oplus del(c)) =$	$del(c)$	OK
165	$(op(a) \oplus nop) \oplus del(c) =$	$del(c)$	$op(a) \oplus (nop \oplus del(c)) =$	$del(c)$	OK
166	$(op(b) \oplus nop) \oplus del(c) =$	$del(c)$	$op(b) \oplus (nop \oplus del(c)) =$	$del(c)$	OK
167	$(op(c) \oplus nop) \oplus del(c) =$	$del(c)$	$op(c) \oplus (nop \oplus del(c)) =$	$del(c)$	OK
168	$(del(a) \oplus op(a)) \oplus del(c) =$	$error$	$del(a) \oplus (op(a) \oplus del(c)) =$	$error$	OK
169	$(del(b) \oplus op(a)) \oplus del(c) =$	$error$	$del(b) \oplus (op(a) \oplus del(c)) =$	$error$	OK
170	$(del(c) \oplus op(a)) \oplus del(c) =$	$del(c)$	$del(c) \oplus (op(a) \oplus del(c)) =$	$del(c)$	OK
171	$(call \oplus op(a)) \oplus del(c) =$	$del(c)$	$call \oplus (op(a) \oplus del(c)) =$	$del(c)$	OK
172	$(nop \oplus op(a)) \oplus del(c) =$	$del(c)$	$nop \oplus (op(a) \oplus del(c)) =$	$del(c)$	OK
173	$(op(a) \oplus op(a)) \oplus del(c) =$	$del(c)$	$op(a) \oplus (op(a) \oplus del(c)) =$	$del(c)$	OK
174	$(op(b) \oplus op(a)) \oplus del(c) =$	$del(c)$	$op(b) \oplus (op(a) \oplus del(c)) =$	$del(c)$	OK
175	$(op(c) \oplus op(a)) \oplus del(c) =$	$del(c)$	$op(c) \oplus (op(a) \oplus del(c)) =$	$del(c)$	OK
176	$(del(a) \oplus op(b)) \oplus del(c) =$	$error$	$del(a) \oplus (op(b) \oplus del(c)) =$	$error$	OK
177	$(del(b) \oplus op(b)) \oplus del(c) =$	$error$	$del(b) \oplus (op(b) \oplus del(c)) =$	$error$	OK
178	$(del(c) \oplus op(b)) \oplus del(c) =$	$del(c)$	$del(c) \oplus (op(b) \oplus del(c)) =$	$del(c)$	OK
179	$(call \oplus op(b)) \oplus del(c) =$	$del(c)$	$call \oplus (op(b) \oplus del(c)) =$	$del(c)$	OK
180	$(nop \oplus op(b)) \oplus del(c) =$	$del(c)$	$nop \oplus (op(b) \oplus del(c)) =$	$del(c)$	OK
181	$(op(a) \oplus op(b)) \oplus del(c) =$	$del(c)$	$op(a) \oplus (op(b) \oplus del(c)) =$	$del(c)$	OK
182	$(op(b) \oplus op(b)) \oplus del(c) =$	$del(c)$	$op(b) \oplus (op(b) \oplus del(c)) =$	$del(c)$	OK
183	$(op(c) \oplus op(b)) \oplus del(c) =$	$del(c)$	$op(c) \oplus (op(b) \oplus del(c)) =$	$del(c)$	OK
184	$(del(a) \oplus op(c)) \oplus del(c) =$	$error$	$del(a) \oplus (op(c) \oplus del(c)) =$	$error$	OK
185	$(del(b) \oplus op(c)) \oplus del(c) =$	$error$	$del(b) \oplus (op(c) \oplus del(c)) =$	$error$	OK
186	$(del(c) \oplus op(c)) \oplus del(c) =$	$del(c)$	$del(c) \oplus (op(c) \oplus del(c)) =$	$del(c)$	OK
187	$(call \oplus op(c)) \oplus del(c) =$	$del(c)$	$call \oplus (op(c) \oplus del(c)) =$	$del(c)$	OK
188	$(nop \oplus op(c)) \oplus del(c) =$	$del(c)$	$nop \oplus (op(c) \oplus del(c)) =$	$del(c)$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
189	$(op(a) \oplus op(c)) \oplus del(c) =$	$del(c)$	$op(a) \oplus (op(c) \oplus del(c)) =$	$del(c)$	OK
190	$(op(b) \oplus op(c)) \oplus del(c) =$	$del(c)$	$op(b) \oplus (op(c) \oplus del(c)) =$	$del(c)$	OK
191	$(op(c) \oplus op(c)) \oplus del(c) =$	$del(c)$	$op(c) \oplus (op(c) \oplus del(c)) =$	$del(c)$	OK
192	$(del(a) \oplus del(a)) \oplus call =$	$del(a)$	$del(a) \oplus (del(a) \oplus call) =$	$del(a)$	OK
193	$(del(b) \oplus del(a)) \oplus call =$	$error$	$del(b) \oplus (del(a) \oplus call) =$	$error$	OK
194	$(del(c) \oplus del(a)) \oplus call =$	$error$	$del(c) \oplus (del(a) \oplus call) =$	$error$	OK
195	$(call \oplus del(a)) \oplus call =$	$del(a)$	$call \oplus (del(a) \oplus call) =$	$del(a)$	OK
196	$(nop \oplus del(a)) \oplus call =$	$del(a)$	$nop \oplus (del(a) \oplus call) =$	$del(a)$	OK
197	$(op(a) \oplus del(a)) \oplus call =$	$del(a)$	$op(a) \oplus (del(a) \oplus call) =$	$del(a)$	OK
198	$(op(b) \oplus del(a)) \oplus call =$	$del(a)$	$op(b) \oplus (del(a) \oplus call) =$	$del(a)$	OK
199	$(op(c) \oplus del(a)) \oplus call =$	$del(a)$	$op(c) \oplus (del(a) \oplus call) =$	$del(a)$	OK
200	$(del(a) \oplus del(b)) \oplus call =$	$error$	$del(a) \oplus (del(b) \oplus call) =$	$error$	OK
201	$(del(b) \oplus del(b)) \oplus call =$	$del(b)$	$del(b) \oplus (del(b) \oplus call) =$	$del(b)$	OK
202	$(del(c) \oplus del(b)) \oplus call =$	$error$	$del(c) \oplus (del(b) \oplus call) =$	$error$	OK
203	$(call \oplus del(b)) \oplus call =$	$del(b)$	$call \oplus (del(b) \oplus call) =$	$del(b)$	OK
204	$(nop \oplus del(b)) \oplus call =$	$del(b)$	$nop \oplus (del(b) \oplus call) =$	$del(b)$	OK
205	$(op(a) \oplus del(b)) \oplus call =$	$del(b)$	$op(a) \oplus (del(b) \oplus call) =$	$del(b)$	OK
206	$(op(b) \oplus del(b)) \oplus call =$	$del(b)$	$op(b) \oplus (del(b) \oplus call) =$	$del(b)$	OK
207	$(op(c) \oplus del(b)) \oplus call =$	$del(b)$	$op(c) \oplus (del(b) \oplus call) =$	$del(b)$	OK
208	$(del(a) \oplus del(c)) \oplus call =$	$error$	$del(a) \oplus (del(c) \oplus call) =$	$error$	OK
209	$(del(b) \oplus del(c)) \oplus call =$	$error$	$del(b) \oplus (del(c) \oplus call) =$	$error$	OK
210	$(del(c) \oplus del(c)) \oplus call =$	$del(c)$	$del(c) \oplus (del(c) \oplus call) =$	$del(c)$	OK
211	$(call \oplus del(c)) \oplus call =$	$del(c)$	$call \oplus (del(c) \oplus call) =$	$del(c)$	OK
212	$(nop \oplus del(c)) \oplus call =$	$del(c)$	$nop \oplus (del(c) \oplus call) =$	$del(c)$	OK
213	$(op(a) \oplus del(c)) \oplus call =$	$del(c)$	$op(a) \oplus (del(c) \oplus call) =$	$del(c)$	OK
214	$(op(b) \oplus del(c)) \oplus call =$	$del(c)$	$op(b) \oplus (del(c) \oplus call) =$	$del(c)$	OK
215	$(op(c) \oplus del(c)) \oplus call =$	$del(c)$	$op(c) \oplus (del(c) \oplus call) =$	$del(c)$	OK
216	$(del(a) \oplus call) \oplus call =$	$del(a)$	$del(a) \oplus (call \oplus call) =$	$del(a)$	OK
217	$(del(b) \oplus call) \oplus call =$	$del(b)$	$del(b) \oplus (call \oplus call) =$	$del(b)$	OK
218	$(del(c) \oplus call) \oplus call =$	$del(c)$	$del(c) \oplus (call \oplus call) =$	$del(c)$	OK
219	$(call \oplus call) \oplus call =$	$call$	$call \oplus (call \oplus call) =$	$call$	OK
220	$(nop \oplus call) \oplus call =$	nop	$nop \oplus (call \oplus call) =$	nop	OK
221	$(op(a) \oplus call) \oplus call =$	$op(a)$	$op(a) \oplus (call \oplus call) =$	$op(a)$	OK
222	$(op(b) \oplus call) \oplus call =$	$op(b)$	$op(b) \oplus (call \oplus call) =$	$op(b)$	OK
223	$(op(c) \oplus call) \oplus call =$	$op(c)$	$op(c) \oplus (call \oplus call) =$	$op(c)$	OK
224	$(del(a) \oplus nop) \oplus call =$	$del(a)$	$del(a) \oplus (nop \oplus call) =$	$del(a)$	OK
225	$(del(b) \oplus nop) \oplus call =$	$del(b)$	$del(b) \oplus (nop \oplus call) =$	$del(b)$	OK
226	$(del(c) \oplus nop) \oplus call =$	$del(c)$	$del(c) \oplus (nop \oplus call) =$	$del(c)$	OK
227	$(call \oplus nop) \oplus call =$	nop	$call \oplus (nop \oplus call) =$	nop	OK
228	$(nop \oplus nop) \oplus call =$	nop	$nop \oplus (nop \oplus call) =$	nop	OK
229	$(op(a) \oplus nop) \oplus call =$	$op(a)$	$op(a) \oplus (nop \oplus call) =$	$op(a)$	OK
230	$(op(b) \oplus nop) \oplus call =$	$op(b)$	$op(b) \oplus (nop \oplus call) =$	$op(b)$	OK
231	$(op(c) \oplus nop) \oplus call =$	$op(c)$	$op(c) \oplus (nop \oplus call) =$	$op(c)$	OK
232	$(del(a) \oplus op(a)) \oplus call =$	$del(a)$	$del(a) \oplus (op(a) \oplus call) =$	$del(a)$	OK
233	$(del(b) \oplus op(a)) \oplus call =$	$del(b)$	$del(b) \oplus (op(a) \oplus call) =$	$del(b)$	OK
234	$(del(c) \oplus op(a)) \oplus call =$	$del(c)$	$del(c) \oplus (op(a) \oplus call) =$	$del(c)$	OK
235	$(call \oplus op(a)) \oplus call =$	$op(a)$	$call \oplus (op(a) \oplus call) =$	$op(a)$	OK
236	$(nop \oplus op(a)) \oplus call =$	$op(a)$	$nop \oplus (op(a) \oplus call) =$	$op(a)$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
237	$(op(a) \oplus op(a)) \oplus call =$	$op(a)$	$op(a) \oplus (op(a) \oplus call) =$	$op(a)$	OK
238	$(op(b) \oplus op(a)) \oplus call =$	$error$	$op(b) \oplus (op(a) \oplus call) =$	$error$	OK
239	$(op(c) \oplus op(a)) \oplus call =$	$error$	$op(c) \oplus (op(a) \oplus call) =$	$error$	OK
240	$(del(a) \oplus op(b)) \oplus call =$	$del(a)$	$del(a) \oplus (op(b) \oplus call) =$	$del(a)$	OK
241	$(del(b) \oplus op(b)) \oplus call =$	$del(b)$	$del(b) \oplus (op(b) \oplus call) =$	$del(b)$	OK
242	$(del(c) \oplus op(b)) \oplus call =$	$del(c)$	$del(c) \oplus (op(b) \oplus call) =$	$del(c)$	OK
243	$(call \oplus op(b)) \oplus call =$	$op(b)$	$call \oplus (op(b) \oplus call) =$	$op(b)$	OK
244	$(nop \oplus op(b)) \oplus call =$	$op(b)$	$nop \oplus (op(b) \oplus call) =$	$op(b)$	OK
245	$(op(a) \oplus op(b)) \oplus call =$	$error$	$op(a) \oplus (op(b) \oplus call) =$	$error$	OK
246	$(op(b) \oplus op(b)) \oplus call =$	$op(b)$	$op(b) \oplus (op(b) \oplus call) =$	$op(b)$	OK
247	$(op(c) \oplus op(b)) \oplus call =$	$error$	$op(c) \oplus (op(b) \oplus call) =$	$error$	OK
248	$(del(a) \oplus op(c)) \oplus call =$	$del(a)$	$del(a) \oplus (op(c) \oplus call) =$	$del(a)$	OK
249	$(del(b) \oplus op(c)) \oplus call =$	$del(b)$	$del(b) \oplus (op(c) \oplus call) =$	$del(b)$	OK
250	$(del(c) \oplus op(c)) \oplus call =$	$del(c)$	$del(c) \oplus (op(c) \oplus call) =$	$del(c)$	OK
251	$(call \oplus op(c)) \oplus call =$	$op(c)$	$call \oplus (op(c) \oplus call) =$	$op(c)$	OK
252	$(nop \oplus op(c)) \oplus call =$	$op(c)$	$nop \oplus (op(c) \oplus call) =$	$op(c)$	OK
253	$(op(a) \oplus op(c)) \oplus call =$	$error$	$op(a) \oplus (op(c) \oplus call) =$	$error$	OK
254	$(op(b) \oplus op(c)) \oplus call =$	$error$	$op(b) \oplus (op(c) \oplus call) =$	$error$	OK
255	$(op(c) \oplus op(c)) \oplus call =$	$op(c)$	$op(c) \oplus (op(c) \oplus call) =$	$op(c)$	OK
256	$(del(a) \oplus del(a)) \oplus nop =$	$del(a)$	$del(a) \oplus (del(a) \oplus nop) =$	$del(a)$	OK
257	$(del(b) \oplus del(a)) \oplus nop =$	$error$	$del(b) \oplus (del(a) \oplus nop) =$	$error$	OK
258	$(del(c) \oplus del(a)) \oplus nop =$	$error$	$del(c) \oplus (del(a) \oplus nop) =$	$error$	OK
259	$(call \oplus del(a)) \oplus nop =$	$del(a)$	$call \oplus (del(a) \oplus nop) =$	$del(a)$	OK
260	$(nop \oplus del(a)) \oplus nop =$	$del(a)$	$nop \oplus (del(a) \oplus nop) =$	$del(a)$	OK
261	$(op(a) \oplus del(a)) \oplus nop =$	$del(a)$	$op(a) \oplus (del(a) \oplus nop) =$	$del(a)$	OK
262	$(op(b) \oplus del(a)) \oplus nop =$	$del(a)$	$op(b) \oplus (del(a) \oplus nop) =$	$del(a)$	OK
263	$(op(c) \oplus del(a)) \oplus nop =$	$del(a)$	$op(c) \oplus (del(a) \oplus nop) =$	$del(a)$	OK
264	$(del(a) \oplus del(b)) \oplus nop =$	$error$	$del(a) \oplus (del(b) \oplus nop) =$	$error$	OK
265	$(del(b) \oplus del(b)) \oplus nop =$	$del(b)$	$del(b) \oplus (del(b) \oplus nop) =$	$del(b)$	OK
266	$(del(c) \oplus del(b)) \oplus nop =$	$error$	$del(c) \oplus (del(b) \oplus nop) =$	$error$	OK
267	$(call \oplus del(b)) \oplus nop =$	$del(b)$	$call \oplus (del(b) \oplus nop) =$	$del(b)$	OK
268	$(nop \oplus del(b)) \oplus nop =$	$del(b)$	$nop \oplus (del(b) \oplus nop) =$	$del(b)$	OK
269	$(op(a) \oplus del(b)) \oplus nop =$	$del(b)$	$op(a) \oplus (del(b) \oplus nop) =$	$del(b)$	OK
270	$(op(b) \oplus del(b)) \oplus nop =$	$del(b)$	$op(b) \oplus (del(b) \oplus nop) =$	$del(b)$	OK
271	$(op(c) \oplus del(b)) \oplus nop =$	$del(b)$	$op(c) \oplus (del(b) \oplus nop) =$	$del(b)$	OK
272	$(del(a) \oplus del(c)) \oplus nop =$	$error$	$del(a) \oplus (del(c) \oplus nop) =$	$error$	OK
273	$(del(b) \oplus del(c)) \oplus nop =$	$error$	$del(b) \oplus (del(c) \oplus nop) =$	$error$	OK
274	$(del(c) \oplus del(c)) \oplus nop =$	$del(c)$	$del(c) \oplus (del(c) \oplus nop) =$	$del(c)$	OK
275	$(call \oplus del(c)) \oplus nop =$	$del(c)$	$call \oplus (del(c) \oplus nop) =$	$del(c)$	OK
276	$(nop \oplus del(c)) \oplus nop =$	$del(c)$	$nop \oplus (del(c) \oplus nop) =$	$del(c)$	OK
277	$(op(a) \oplus del(c)) \oplus nop =$	$del(c)$	$op(a) \oplus (del(c) \oplus nop) =$	$del(c)$	OK
278	$(op(b) \oplus del(c)) \oplus nop =$	$del(c)$	$op(b) \oplus (del(c) \oplus nop) =$	$del(c)$	OK
279	$(op(c) \oplus del(c)) \oplus nop =$	$del(c)$	$op(c) \oplus (del(c) \oplus nop) =$	$del(c)$	OK
280	$(del(a) \oplus call) \oplus nop =$	$del(a)$	$del(a) \oplus (call \oplus nop) =$	$del(a)$	OK
281	$(del(b) \oplus call) \oplus nop =$	$del(b)$	$del(b) \oplus (call \oplus nop) =$	$del(b)$	OK
282	$(del(c) \oplus call) \oplus nop =$	$del(c)$	$del(c) \oplus (call \oplus nop) =$	$del(c)$	OK
283	$(call \oplus call) \oplus nop =$	nop	$call \oplus (call \oplus nop) =$	nop	OK
284	$(nop \oplus call) \oplus nop =$	nop	$nop \oplus (call \oplus nop) =$	nop	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
285	$(op(a) \oplus call) \oplus nop =$	$op(a)$	$op(a) \oplus (call \oplus nop) =$	$op(a)$	OK
286	$(op(b) \oplus call) \oplus nop =$	$op(b)$	$op(b) \oplus (call \oplus nop) =$	$op(b)$	OK
287	$(op(c) \oplus call) \oplus nop =$	$op(c)$	$op(c) \oplus (call \oplus nop) =$	$op(c)$	OK
288	$(del(a) \oplus nop) \oplus nop =$	$del(a)$	$del(a) \oplus (nop \oplus nop) =$	$del(a)$	OK
289	$(del(b) \oplus nop) \oplus nop =$	$del(b)$	$del(b) \oplus (nop \oplus nop) =$	$del(b)$	OK
290	$(del(c) \oplus nop) \oplus nop =$	$del(c)$	$del(c) \oplus (nop \oplus nop) =$	$del(c)$	OK
291	$(call \oplus nop) \oplus nop =$	nop	$call \oplus (nop \oplus nop) =$	nop	OK
292	$(nop \oplus nop) \oplus nop =$	nop	$nop \oplus (nop \oplus nop) =$	nop	OK
293	$(op(a) \oplus nop) \oplus nop =$	$op(a)$	$op(a) \oplus (nop \oplus nop) =$	$op(a)$	OK
294	$(op(b) \oplus nop) \oplus nop =$	$op(b)$	$op(b) \oplus (nop \oplus nop) =$	$op(b)$	OK
295	$(op(c) \oplus nop) \oplus nop =$	$op(c)$	$op(c) \oplus (nop \oplus nop) =$	$op(c)$	OK
296	$(del(a) \oplus op(a)) \oplus nop =$	$del(a)$	$del(a) \oplus (op(a) \oplus nop) =$	$del(a)$	OK
297	$(del(b) \oplus op(a)) \oplus nop =$	$del(b)$	$del(b) \oplus (op(a) \oplus nop) =$	$del(b)$	OK
298	$(del(c) \oplus op(a)) \oplus nop =$	$del(c)$	$del(c) \oplus (op(a) \oplus nop) =$	$del(c)$	OK
299	$(call \oplus op(a)) \oplus nop =$	$op(a)$	$call \oplus (op(a) \oplus nop) =$	$op(a)$	OK
300	$(nop \oplus op(a)) \oplus nop =$	$op(a)$	$nop \oplus (op(a) \oplus nop) =$	$op(a)$	OK
301	$(op(a) \oplus op(a)) \oplus nop =$	$op(a)$	$op(a) \oplus (op(a) \oplus nop) =$	$op(a)$	OK
302	$(op(b) \oplus op(a)) \oplus nop =$	$error$	$op(b) \oplus (op(a) \oplus nop) =$	$error$	OK
303	$(op(c) \oplus op(a)) \oplus nop =$	$error$	$op(c) \oplus (op(a) \oplus nop) =$	$error$	OK
304	$(del(a) \oplus op(b)) \oplus nop =$	$del(a)$	$del(a) \oplus (op(b) \oplus nop) =$	$del(a)$	OK
305	$(del(b) \oplus op(b)) \oplus nop =$	$del(b)$	$del(b) \oplus (op(b) \oplus nop) =$	$del(b)$	OK
306	$(del(c) \oplus op(b)) \oplus nop =$	$del(c)$	$del(c) \oplus (op(b) \oplus nop) =$	$del(c)$	OK
307	$(call \oplus op(b)) \oplus nop =$	$op(b)$	$call \oplus (op(b) \oplus nop) =$	$op(b)$	OK
308	$(nop \oplus op(b)) \oplus nop =$	$op(b)$	$nop \oplus (op(b) \oplus nop) =$	$op(b)$	OK
309	$(op(a) \oplus op(b)) \oplus nop =$	$error$	$op(a) \oplus (op(b) \oplus nop) =$	$error$	OK
310	$(op(b) \oplus op(b)) \oplus nop =$	$op(b)$	$op(b) \oplus (op(b) \oplus nop) =$	$op(b)$	OK
311	$(op(c) \oplus op(b)) \oplus nop =$	$error$	$op(c) \oplus (op(b) \oplus nop) =$	$error$	OK
312	$(del(a) \oplus op(c)) \oplus nop =$	$del(a)$	$del(a) \oplus (op(c) \oplus nop) =$	$del(a)$	OK
313	$(del(b) \oplus op(c)) \oplus nop =$	$del(b)$	$del(b) \oplus (op(c) \oplus nop) =$	$del(b)$	OK
314	$(del(c) \oplus op(c)) \oplus nop =$	$del(c)$	$del(c) \oplus (op(c) \oplus nop) =$	$del(c)$	OK
315	$(call \oplus op(c)) \oplus nop =$	$op(c)$	$call \oplus (op(c) \oplus nop) =$	$op(c)$	OK
316	$(nop \oplus op(c)) \oplus nop =$	$op(c)$	$nop \oplus (op(c) \oplus nop) =$	$op(c)$	OK
317	$(op(a) \oplus op(c)) \oplus nop =$	$error$	$op(a) \oplus (op(c) \oplus nop) =$	$error$	OK
318	$(op(b) \oplus op(c)) \oplus nop =$	$error$	$op(b) \oplus (op(c) \oplus nop) =$	$error$	OK
319	$(op(c) \oplus op(c)) \oplus nop =$	$op(c)$	$op(c) \oplus (op(c) \oplus nop) =$	$op(c)$	OK
320	$(del(a) \oplus del(a)) \oplus op(a) =$	$del(a)$	$del(a) \oplus (del(a) \oplus op(a)) =$	$del(a)$	OK
321	$(del(b) \oplus del(a)) \oplus op(a) =$	$error$	$del(b) \oplus (del(a) \oplus op(a)) =$	$error$	OK
322	$(del(c) \oplus del(a)) \oplus op(a) =$	$error$	$del(c) \oplus (del(a) \oplus op(a)) =$	$error$	OK
323	$(call \oplus del(a)) \oplus op(a) =$	$del(a)$	$call \oplus (del(a) \oplus op(a)) =$	$del(a)$	OK
324	$(nop \oplus del(a)) \oplus op(a) =$	$del(a)$	$nop \oplus (del(a) \oplus op(a)) =$	$del(a)$	OK
325	$(op(a) \oplus del(a)) \oplus op(a) =$	$del(a)$	$op(a) \oplus (del(a) \oplus op(a)) =$	$del(a)$	OK
326	$(op(b) \oplus del(a)) \oplus op(a) =$	$del(a)$	$op(b) \oplus (del(a) \oplus op(a)) =$	$del(a)$	OK
327	$(op(c) \oplus del(a)) \oplus op(a) =$	$del(a)$	$op(c) \oplus (del(a) \oplus op(a)) =$	$del(a)$	OK
328	$(del(a) \oplus del(b)) \oplus op(a) =$	$error$	$del(a) \oplus (del(b) \oplus op(a)) =$	$error$	OK
329	$(del(b) \oplus del(b)) \oplus op(a) =$	$del(b)$	$del(b) \oplus (del(b) \oplus op(a)) =$	$del(b)$	OK
330	$(del(c) \oplus del(b)) \oplus op(a) =$	$error$	$del(c) \oplus (del(b) \oplus op(a)) =$	$error$	OK
331	$(call \oplus del(b)) \oplus op(a) =$	$del(b)$	$call \oplus (del(b) \oplus op(a)) =$	$del(b)$	OK
332	$(nop \oplus del(b)) \oplus op(a) =$	$del(b)$	$nop \oplus (del(b) \oplus op(a)) =$	$del(b)$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
333	$(op(a) \oplus del(b)) \oplus op(a) =$	$del(b)$	$op(a) \oplus (del(b) \oplus op(a)) =$	$del(b)$	OK
334	$(op(b) \oplus del(b)) \oplus op(a) =$	$del(b)$	$op(b) \oplus (del(b) \oplus op(a)) =$	$del(b)$	OK
335	$(op(c) \oplus del(b)) \oplus op(a) =$	$del(b)$	$op(c) \oplus (del(b) \oplus op(a)) =$	$del(b)$	OK
336	$(del(a) \oplus del(c)) \oplus op(a) =$	$error$	$del(a) \oplus (del(c) \oplus op(a)) =$	$error$	OK
337	$(del(b) \oplus del(c)) \oplus op(a) =$	$error$	$del(b) \oplus (del(c) \oplus op(a)) =$	$error$	OK
338	$(del(c) \oplus del(c)) \oplus op(a) =$	$del(c)$	$del(c) \oplus (del(c) \oplus op(a)) =$	$del(c)$	OK
339	$(call \oplus del(c)) \oplus op(a) =$	$del(c)$	$call \oplus (del(c) \oplus op(a)) =$	$del(c)$	OK
340	$(nop \oplus del(c)) \oplus op(a) =$	$del(c)$	$nop \oplus (del(c) \oplus op(a)) =$	$del(c)$	OK
341	$(op(a) \oplus del(c)) \oplus op(a) =$	$del(c)$	$op(a) \oplus (del(c) \oplus op(a)) =$	$del(c)$	OK
342	$(op(b) \oplus del(c)) \oplus op(a) =$	$del(c)$	$op(b) \oplus (del(c) \oplus op(a)) =$	$del(c)$	OK
343	$(op(c) \oplus del(c)) \oplus op(a) =$	$del(c)$	$op(c) \oplus (del(c) \oplus op(a)) =$	$del(c)$	OK
344	$(del(a) \oplus call) \oplus op(a) =$	$del(a)$	$del(a) \oplus (call \oplus op(a)) =$	$del(a)$	OK
345	$(del(b) \oplus call) \oplus op(a) =$	$del(b)$	$del(b) \oplus (call \oplus op(a)) =$	$del(b)$	OK
346	$(del(c) \oplus call) \oplus op(a) =$	$del(c)$	$del(c) \oplus (call \oplus op(a)) =$	$del(c)$	OK
347	$(call \oplus call) \oplus op(a) =$	$op(a)$	$call \oplus (call \oplus op(a)) =$	$op(a)$	OK
348	$(nop \oplus call) \oplus op(a) =$	$op(a)$	$nop \oplus (call \oplus op(a)) =$	$op(a)$	OK
349	$(op(a) \oplus call) \oplus op(a) =$	$op(a)$	$op(a) \oplus (call \oplus op(a)) =$	$op(a)$	OK
350	$(op(b) \oplus call) \oplus op(a) =$	$error$	$op(b) \oplus (call \oplus op(a)) =$	$error$	OK
351	$(op(c) \oplus call) \oplus op(a) =$	$error$	$op(c) \oplus (call \oplus op(a)) =$	$error$	OK
352	$(del(a) \oplus nop) \oplus op(a) =$	$del(a)$	$del(a) \oplus (nop \oplus op(a)) =$	$del(a)$	OK
353	$(del(b) \oplus nop) \oplus op(a) =$	$del(b)$	$del(b) \oplus (nop \oplus op(a)) =$	$del(b)$	OK
354	$(del(c) \oplus nop) \oplus op(a) =$	$del(c)$	$del(c) \oplus (nop \oplus op(a)) =$	$del(c)$	OK
355	$(call \oplus nop) \oplus op(a) =$	$op(a)$	$call \oplus (nop \oplus op(a)) =$	$op(a)$	OK
356	$(nop \oplus nop) \oplus op(a) =$	$op(a)$	$nop \oplus (nop \oplus op(a)) =$	$op(a)$	OK
357	$(op(a) \oplus nop) \oplus op(a) =$	$op(a)$	$op(a) \oplus (nop \oplus op(a)) =$	$op(a)$	OK
358	$(op(b) \oplus nop) \oplus op(a) =$	$error$	$op(b) \oplus (nop \oplus op(a)) =$	$error$	OK
359	$(op(c) \oplus nop) \oplus op(a) =$	$error$	$op(c) \oplus (nop \oplus op(a)) =$	$error$	OK
360	$(del(a) \oplus op(a)) \oplus op(a) =$	$del(a)$	$del(a) \oplus (op(a) \oplus op(a)) =$	$del(a)$	OK
361	$(del(b) \oplus op(a)) \oplus op(a) =$	$del(b)$	$del(b) \oplus (op(a) \oplus op(a)) =$	$del(b)$	OK
362	$(del(c) \oplus op(a)) \oplus op(a) =$	$del(c)$	$del(c) \oplus (op(a) \oplus op(a)) =$	$del(c)$	OK
363	$(call \oplus op(a)) \oplus op(a) =$	$op(a)$	$call \oplus (op(a) \oplus op(a)) =$	$op(a)$	OK
364	$(nop \oplus op(a)) \oplus op(a) =$	$op(a)$	$nop \oplus (op(a) \oplus op(a)) =$	$op(a)$	OK
365	$(op(a) \oplus op(a)) \oplus op(a) =$	$op(a)$	$op(a) \oplus (op(a) \oplus op(a)) =$	$op(a)$	OK
366	$(op(b) \oplus op(a)) \oplus op(a) =$	$error$	$op(b) \oplus (op(a) \oplus op(a)) =$	$error$	OK
367	$(op(c) \oplus op(a)) \oplus op(a) =$	$error$	$op(c) \oplus (op(a) \oplus op(a)) =$	$error$	OK
368	$(del(a) \oplus op(b)) \oplus op(a) =$	$del(a)$	$del(a) \oplus (op(b) \oplus op(a)) =$	$del(a)$	OK
369	$(del(b) \oplus op(b)) \oplus op(a) =$	$del(b)$	$del(b) \oplus (op(b) \oplus op(a)) =$	$del(b)$	OK
370	$(del(c) \oplus op(b)) \oplus op(a) =$	$del(c)$	$del(c) \oplus (op(b) \oplus op(a)) =$	$del(c)$	OK
371	$(call \oplus op(b)) \oplus op(a) =$	$error$	$call \oplus (op(b) \oplus op(a)) =$	$error$	OK
372	$(nop \oplus op(b)) \oplus op(a) =$	$error$	$nop \oplus (op(b) \oplus op(a)) =$	$error$	OK
373	$(op(a) \oplus op(b)) \oplus op(a) =$	$error$	$op(a) \oplus (op(b) \oplus op(a)) =$	$error$	OK
374	$(op(b) \oplus op(b)) \oplus op(a) =$	$error$	$op(b) \oplus (op(b) \oplus op(a)) =$	$error$	OK
375	$(op(c) \oplus op(b)) \oplus op(a) =$	$error$	$op(c) \oplus (op(b) \oplus op(a)) =$	$error$	OK
376	$(del(a) \oplus op(c)) \oplus op(a) =$	$del(a)$	$del(a) \oplus (op(c) \oplus op(a)) =$	$del(a)$	OK
377	$(del(b) \oplus op(c)) \oplus op(a) =$	$del(b)$	$del(b) \oplus (op(c) \oplus op(a)) =$	$del(b)$	OK
378	$(del(c) \oplus op(c)) \oplus op(a) =$	$del(c)$	$del(c) \oplus (op(c) \oplus op(a)) =$	$del(c)$	OK
379	$(call \oplus op(c)) \oplus op(a) =$	$error$	$call \oplus (op(c) \oplus op(a)) =$	$error$	OK
380	$(nop \oplus op(c)) \oplus op(a) =$	$error$	$nop \oplus (op(c) \oplus op(a)) =$	$error$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
381	$(op(a) \oplus op(c)) \oplus op(a) =$	$error$	$op(a) \oplus (op(c) \oplus op(a)) =$	$error$	OK
382	$(op(b) \oplus op(c)) \oplus op(a) =$	$error$	$op(b) \oplus (op(c) \oplus op(a)) =$	$error$	OK
383	$(op(c) \oplus op(c)) \oplus op(a) =$	$error$	$op(c) \oplus (op(c) \oplus op(a)) =$	$error$	OK
384	$(del(a) \oplus del(a)) \oplus op(b) =$	$del(a)$	$del(a) \oplus (del(a) \oplus op(b)) =$	$del(a)$	OK
385	$(del(b) \oplus del(a)) \oplus op(b) =$	$error$	$del(b) \oplus (del(a) \oplus op(b)) =$	$error$	OK
386	$(del(c) \oplus del(a)) \oplus op(b) =$	$error$	$del(c) \oplus (del(a) \oplus op(b)) =$	$error$	OK
387	$(call \oplus del(a)) \oplus op(b) =$	$del(a)$	$call \oplus (del(a) \oplus op(b)) =$	$del(a)$	OK
388	$(nop \oplus del(a)) \oplus op(b) =$	$del(a)$	$nop \oplus (del(a) \oplus op(b)) =$	$del(a)$	OK
389	$(op(a) \oplus del(a)) \oplus op(b) =$	$del(a)$	$op(a) \oplus (del(a) \oplus op(b)) =$	$del(a)$	OK
390	$(op(b) \oplus del(a)) \oplus op(b) =$	$del(a)$	$op(b) \oplus (del(a) \oplus op(b)) =$	$del(a)$	OK
391	$(op(c) \oplus del(a)) \oplus op(b) =$	$del(a)$	$op(c) \oplus (del(a) \oplus op(b)) =$	$del(a)$	OK
392	$(del(a) \oplus del(b)) \oplus op(b) =$	$error$	$del(a) \oplus (del(b) \oplus op(b)) =$	$error$	OK
393	$(del(b) \oplus del(b)) \oplus op(b) =$	$del(b)$	$del(b) \oplus (del(b) \oplus op(b)) =$	$del(b)$	OK
394	$(del(c) \oplus del(b)) \oplus op(b) =$	$error$	$del(c) \oplus (del(b) \oplus op(b)) =$	$error$	OK
395	$(call \oplus del(b)) \oplus op(b) =$	$del(b)$	$call \oplus (del(b) \oplus op(b)) =$	$del(b)$	OK
396	$(nop \oplus del(b)) \oplus op(b) =$	$del(b)$	$nop \oplus (del(b) \oplus op(b)) =$	$del(b)$	OK
397	$(op(a) \oplus del(b)) \oplus op(b) =$	$del(b)$	$op(a) \oplus (del(b) \oplus op(b)) =$	$del(b)$	OK
398	$(op(b) \oplus del(b)) \oplus op(b) =$	$del(b)$	$op(b) \oplus (del(b) \oplus op(b)) =$	$del(b)$	OK
399	$(op(c) \oplus del(b)) \oplus op(b) =$	$del(b)$	$op(c) \oplus (del(b) \oplus op(b)) =$	$del(b)$	OK
400	$(del(a) \oplus del(c)) \oplus op(b) =$	$error$	$del(a) \oplus (del(c) \oplus op(b)) =$	$error$	OK
401	$(del(b) \oplus del(c)) \oplus op(b) =$	$error$	$del(b) \oplus (del(c) \oplus op(b)) =$	$error$	OK
402	$(del(c) \oplus del(c)) \oplus op(b) =$	$del(c)$	$del(c) \oplus (del(c) \oplus op(b)) =$	$del(c)$	OK
403	$(call \oplus del(c)) \oplus op(b) =$	$del(c)$	$call \oplus (del(c) \oplus op(b)) =$	$del(c)$	OK
404	$(nop \oplus del(c)) \oplus op(b) =$	$del(c)$	$nop \oplus (del(c) \oplus op(b)) =$	$del(c)$	OK
405	$(op(a) \oplus del(c)) \oplus op(b) =$	$del(c)$	$op(a) \oplus (del(c) \oplus op(b)) =$	$del(c)$	OK
406	$(op(b) \oplus del(c)) \oplus op(b) =$	$del(c)$	$op(b) \oplus (del(c) \oplus op(b)) =$	$del(c)$	OK
407	$(op(c) \oplus del(c)) \oplus op(b) =$	$del(c)$	$op(c) \oplus (del(c) \oplus op(b)) =$	$del(c)$	OK
408	$(del(a) \oplus call) \oplus op(b) =$	$del(a)$	$del(a) \oplus (call \oplus op(b)) =$	$del(a)$	OK
409	$(del(b) \oplus call) \oplus op(b) =$	$del(b)$	$del(b) \oplus (call \oplus op(b)) =$	$del(b)$	OK
410	$(del(c) \oplus call) \oplus op(b) =$	$del(c)$	$del(c) \oplus (call \oplus op(b)) =$	$del(c)$	OK
411	$(call \oplus call) \oplus op(b) =$	$op(b)$	$call \oplus (call \oplus op(b)) =$	$op(b)$	OK
412	$(nop \oplus call) \oplus op(b) =$	$op(b)$	$nop \oplus (call \oplus op(b)) =$	$op(b)$	OK
413	$(op(a) \oplus call) \oplus op(b) =$	$error$	$op(a) \oplus (call \oplus op(b)) =$	$error$	OK
414	$(op(b) \oplus call) \oplus op(b) =$	$op(b)$	$op(b) \oplus (call \oplus op(b)) =$	$op(b)$	OK
415	$(op(c) \oplus call) \oplus op(b) =$	$error$	$op(c) \oplus (call \oplus op(b)) =$	$error$	OK
416	$(del(a) \oplus nop) \oplus op(b) =$	$del(a)$	$del(a) \oplus (nop \oplus op(b)) =$	$del(a)$	OK
417	$(del(b) \oplus nop) \oplus op(b) =$	$del(b)$	$del(b) \oplus (nop \oplus op(b)) =$	$del(b)$	OK
418	$(del(c) \oplus nop) \oplus op(b) =$	$del(c)$	$del(c) \oplus (nop \oplus op(b)) =$	$del(c)$	OK
419	$(call \oplus nop) \oplus op(b) =$	$op(b)$	$call \oplus (nop \oplus op(b)) =$	$op(b)$	OK
420	$(nop \oplus nop) \oplus op(b) =$	$op(b)$	$nop \oplus (nop \oplus op(b)) =$	$op(b)$	OK
421	$(op(a) \oplus nop) \oplus op(b) =$	$error$	$op(a) \oplus (nop \oplus op(b)) =$	$error$	OK
422	$(op(b) \oplus nop) \oplus op(b) =$	$op(b)$	$op(b) \oplus (nop \oplus op(b)) =$	$op(b)$	OK
423	$(op(c) \oplus nop) \oplus op(b) =$	$error$	$op(c) \oplus (nop \oplus op(b)) =$	$error$	OK
424	$(del(a) \oplus op(a)) \oplus op(b) =$	$del(a)$	$del(a) \oplus (op(a) \oplus op(b)) =$	$del(a)$	OK
425	$(del(b) \oplus op(a)) \oplus op(b) =$	$del(b)$	$del(b) \oplus (op(a) \oplus op(b)) =$	$del(b)$	OK
426	$(del(c) \oplus op(a)) \oplus op(b) =$	$del(c)$	$del(c) \oplus (op(a) \oplus op(b)) =$	$del(c)$	OK
427	$(call \oplus op(a)) \oplus op(b) =$	$error$	$call \oplus (op(a) \oplus op(b)) =$	$error$	OK
428	$(nop \oplus op(a)) \oplus op(b) =$	$error$	$nop \oplus (op(a) \oplus op(b)) =$	$error$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
429	$(op(a) \oplus op(a)) \oplus op(b) =$	$error$	$op(a) \oplus (op(a) \oplus op(b)) =$	$error$	OK
430	$(op(b) \oplus op(a)) \oplus op(b) =$	$error$	$op(b) \oplus (op(a) \oplus op(b)) =$	$error$	OK
431	$(op(c) \oplus op(a)) \oplus op(b) =$	$error$	$op(c) \oplus (op(a) \oplus op(b)) =$	$error$	OK
432	$(del(a) \oplus op(b)) \oplus op(b) =$	$del(a)$	$del(a) \oplus (op(b) \oplus op(b)) =$	$del(a)$	OK
433	$(del(b) \oplus op(b)) \oplus op(b) =$	$del(b)$	$del(b) \oplus (op(b) \oplus op(b)) =$	$del(b)$	OK
434	$(del(c) \oplus op(b)) \oplus op(b) =$	$del(c)$	$del(c) \oplus (op(b) \oplus op(b)) =$	$del(c)$	OK
435	$(call \oplus op(b)) \oplus op(b) =$	$op(b)$	$call \oplus (op(b) \oplus op(b)) =$	$op(b)$	OK
436	$(nop \oplus op(b)) \oplus op(b) =$	$op(b)$	$nop \oplus (op(b) \oplus op(b)) =$	$op(b)$	OK
437	$(op(a) \oplus op(b)) \oplus op(b) =$	$error$	$op(a) \oplus (op(b) \oplus op(b)) =$	$error$	OK
438	$(op(b) \oplus op(b)) \oplus op(b) =$	$op(b)$	$op(b) \oplus (op(b) \oplus op(b)) =$	$op(b)$	OK
439	$(op(c) \oplus op(b)) \oplus op(b) =$	$error$	$op(c) \oplus (op(b) \oplus op(b)) =$	$error$	OK
440	$(del(a) \oplus op(c)) \oplus op(b) =$	$del(a)$	$del(a) \oplus (op(c) \oplus op(b)) =$	$del(a)$	OK
441	$(del(b) \oplus op(c)) \oplus op(b) =$	$del(b)$	$del(b) \oplus (op(c) \oplus op(b)) =$	$del(b)$	OK
442	$(del(c) \oplus op(c)) \oplus op(b) =$	$del(c)$	$del(c) \oplus (op(c) \oplus op(b)) =$	$del(c)$	OK
443	$(call \oplus op(c)) \oplus op(b) =$	$error$	$call \oplus (op(c) \oplus op(b)) =$	$error$	OK
444	$(nop \oplus op(c)) \oplus op(b) =$	$error$	$nop \oplus (op(c) \oplus op(b)) =$	$error$	OK
445	$(op(a) \oplus op(c)) \oplus op(b) =$	$error$	$op(a) \oplus (op(c) \oplus op(b)) =$	$error$	OK
446	$(op(b) \oplus op(c)) \oplus op(b) =$	$error$	$op(b) \oplus (op(c) \oplus op(b)) =$	$error$	OK
447	$(op(c) \oplus op(c)) \oplus op(b) =$	$error$	$op(c) \oplus (op(c) \oplus op(b)) =$	$error$	OK
448	$(del(a) \oplus del(a)) \oplus op(c) =$	$del(a)$	$del(a) \oplus (del(a) \oplus op(c)) =$	$del(a)$	OK
449	$(del(b) \oplus del(a)) \oplus op(c) =$	$error$	$del(b) \oplus (del(a) \oplus op(c)) =$	$error$	OK
450	$(del(c) \oplus del(a)) \oplus op(c) =$	$error$	$del(c) \oplus (del(a) \oplus op(c)) =$	$error$	OK
451	$(call \oplus del(a)) \oplus op(c) =$	$del(a)$	$call \oplus (del(a) \oplus op(c)) =$	$del(a)$	OK
452	$(nop \oplus del(a)) \oplus op(c) =$	$del(a)$	$nop \oplus (del(a) \oplus op(c)) =$	$del(a)$	OK
453	$(op(a) \oplus del(a)) \oplus op(c) =$	$del(a)$	$op(a) \oplus (del(a) \oplus op(c)) =$	$del(a)$	OK
454	$(op(b) \oplus del(a)) \oplus op(c) =$	$del(a)$	$op(b) \oplus (del(a) \oplus op(c)) =$	$del(a)$	OK
455	$(op(c) \oplus del(a)) \oplus op(c) =$	$del(a)$	$op(c) \oplus (del(a) \oplus op(c)) =$	$del(a)$	OK
456	$(del(a) \oplus del(b)) \oplus op(c) =$	$error$	$del(a) \oplus (del(b) \oplus op(c)) =$	$error$	OK
457	$(del(b) \oplus del(b)) \oplus op(c) =$	$del(b)$	$del(b) \oplus (del(b) \oplus op(c)) =$	$del(b)$	OK
458	$(del(c) \oplus del(b)) \oplus op(c) =$	$error$	$del(c) \oplus (del(b) \oplus op(c)) =$	$error$	OK
459	$(call \oplus del(b)) \oplus op(c) =$	$del(b)$	$call \oplus (del(b) \oplus op(c)) =$	$del(b)$	OK
460	$(nop \oplus del(b)) \oplus op(c) =$	$del(b)$	$nop \oplus (del(b) \oplus op(c)) =$	$del(b)$	OK
461	$(op(a) \oplus del(b)) \oplus op(c) =$	$del(b)$	$op(a) \oplus (del(b) \oplus op(c)) =$	$del(b)$	OK
462	$(op(b) \oplus del(b)) \oplus op(c) =$	$del(b)$	$op(b) \oplus (del(b) \oplus op(c)) =$	$del(b)$	OK
463	$(op(c) \oplus del(b)) \oplus op(c) =$	$del(b)$	$op(c) \oplus (del(b) \oplus op(c)) =$	$del(b)$	OK
464	$(del(a) \oplus del(c)) \oplus op(c) =$	$error$	$del(a) \oplus (del(c) \oplus op(c)) =$	$error$	OK
465	$(del(b) \oplus del(c)) \oplus op(c) =$	$error$	$del(b) \oplus (del(c) \oplus op(c)) =$	$error$	OK
466	$(del(c) \oplus del(c)) \oplus op(c) =$	$del(c)$	$del(c) \oplus (del(c) \oplus op(c)) =$	$del(c)$	OK
467	$(call \oplus del(c)) \oplus op(c) =$	$del(c)$	$call \oplus (del(c) \oplus op(c)) =$	$del(c)$	OK
468	$(nop \oplus del(c)) \oplus op(c) =$	$del(c)$	$nop \oplus (del(c) \oplus op(c)) =$	$del(c)$	OK
469	$(op(a) \oplus del(c)) \oplus op(c) =$	$del(c)$	$op(a) \oplus (del(c) \oplus op(c)) =$	$del(c)$	OK
470	$(op(b) \oplus del(c)) \oplus op(c) =$	$del(c)$	$op(b) \oplus (del(c) \oplus op(c)) =$	$del(c)$	OK
471	$(op(c) \oplus del(c)) \oplus op(c) =$	$del(c)$	$op(c) \oplus (del(c) \oplus op(c)) =$	$del(c)$	OK
472	$(del(a) \oplus call) \oplus op(c) =$	$del(a)$	$del(a) \oplus (call \oplus op(c)) =$	$del(a)$	OK
473	$(del(b) \oplus call) \oplus op(c) =$	$del(b)$	$del(b) \oplus (call \oplus op(c)) =$	$del(b)$	OK
474	$(del(c) \oplus call) \oplus op(c) =$	$del(c)$	$del(c) \oplus (call \oplus op(c)) =$	$del(c)$	OK
475	$(call \oplus call) \oplus op(c) =$	$op(c)$	$call \oplus (call \oplus op(c)) =$	$op(c)$	OK
476	$(nop \oplus call) \oplus op(c) =$	$op(c)$	$nop \oplus (call \oplus op(c)) =$	$op(c)$	OK

Suite à la page suivante...

Tableau D.1: Calculs de l'associativité

n	$(a \oplus b) \oplus c$		$a \oplus (b \oplus c)$		
477	$(op(a) \oplus call) \oplus op(c) =$	$error$	$op(a) \oplus (call \oplus op(c)) =$	$error$	OK
478	$(op(b) \oplus call) \oplus op(c) =$	$error$	$op(b) \oplus (call \oplus op(c)) =$	$error$	OK
479	$(op(c) \oplus call) \oplus op(c) =$	$op(c)$	$op(c) \oplus (call \oplus op(c)) =$	$op(c)$	OK
480	$(del(a) \oplus nop) \oplus op(c) =$	$del(a)$	$del(a) \oplus (nop \oplus op(c)) =$	$del(a)$	OK
481	$(del(b) \oplus nop) \oplus op(c) =$	$del(b)$	$del(b) \oplus (nop \oplus op(c)) =$	$del(b)$	OK
482	$(del(c) \oplus nop) \oplus op(c) =$	$del(c)$	$del(c) \oplus (nop \oplus op(c)) =$	$del(c)$	OK
483	$(call \oplus nop) \oplus op(c) =$	$op(c)$	$call \oplus (nop \oplus op(c)) =$	$op(c)$	OK
484	$(nop \oplus nop) \oplus op(c) =$	$op(c)$	$nop \oplus (nop \oplus op(c)) =$	$op(c)$	OK
485	$(op(a) \oplus nop) \oplus op(c) =$	$error$	$op(a) \oplus (nop \oplus op(c)) =$	$error$	OK
486	$(op(b) \oplus nop) \oplus op(c) =$	$error$	$op(b) \oplus (nop \oplus op(c)) =$	$error$	OK
487	$(op(c) \oplus nop) \oplus op(c) =$	$op(c)$	$op(c) \oplus (nop \oplus op(c)) =$	$op(c)$	OK
488	$(del(a) \oplus op(a)) \oplus op(c) =$	$del(a)$	$del(a) \oplus (op(a) \oplus op(c)) =$	$del(a)$	OK
489	$(del(b) \oplus op(a)) \oplus op(c) =$	$del(b)$	$del(b) \oplus (op(a) \oplus op(c)) =$	$del(b)$	OK
490	$(del(c) \oplus op(a)) \oplus op(c) =$	$del(c)$	$del(c) \oplus (op(a) \oplus op(c)) =$	$del(c)$	OK
491	$(call \oplus op(a)) \oplus op(c) =$	$error$	$call \oplus (op(a) \oplus op(c)) =$	$error$	OK
492	$(nop \oplus op(a)) \oplus op(c) =$	$error$	$nop \oplus (op(a) \oplus op(c)) =$	$error$	OK
493	$(op(a) \oplus op(a)) \oplus op(c) =$	$error$	$op(a) \oplus (op(a) \oplus op(c)) =$	$error$	OK
494	$(op(b) \oplus op(a)) \oplus op(c) =$	$error$	$op(b) \oplus (op(a) \oplus op(c)) =$	$error$	OK
495	$(op(c) \oplus op(a)) \oplus op(c) =$	$error$	$op(c) \oplus (op(a) \oplus op(c)) =$	$error$	OK
496	$(del(a) \oplus op(b)) \oplus op(c) =$	$del(a)$	$del(a) \oplus (op(b) \oplus op(c)) =$	$del(a)$	OK
497	$(del(b) \oplus op(b)) \oplus op(c) =$	$del(b)$	$del(b) \oplus (op(b) \oplus op(c)) =$	$del(b)$	OK
498	$(del(c) \oplus op(b)) \oplus op(c) =$	$del(c)$	$del(c) \oplus (op(b) \oplus op(c)) =$	$del(c)$	OK
499	$(call \oplus op(b)) \oplus op(c) =$	$error$	$call \oplus (op(b) \oplus op(c)) =$	$error$	OK
500	$(nop \oplus op(b)) \oplus op(c) =$	$error$	$nop \oplus (op(b) \oplus op(c)) =$	$error$	OK
501	$(op(a) \oplus op(b)) \oplus op(c) =$	$error$	$op(a) \oplus (op(b) \oplus op(c)) =$	$error$	OK
502	$(op(b) \oplus op(b)) \oplus op(c) =$	$error$	$op(b) \oplus (op(b) \oplus op(c)) =$	$error$	OK
503	$(op(c) \oplus op(b)) \oplus op(c) =$	$error$	$op(c) \oplus (op(b) \oplus op(c)) =$	$error$	OK
504	$(del(a) \oplus op(c)) \oplus op(c) =$	$del(a)$	$del(a) \oplus (op(c) \oplus op(c)) =$	$del(a)$	OK
505	$(del(b) \oplus op(c)) \oplus op(c) =$	$del(b)$	$del(b) \oplus (op(c) \oplus op(c)) =$	$del(b)$	OK
506	$(del(c) \oplus op(c)) \oplus op(c) =$	$del(c)$	$del(c) \oplus (op(c) \oplus op(c)) =$	$del(c)$	OK
507	$(call \oplus op(c)) \oplus op(c) =$	$op(c)$	$call \oplus (op(c) \oplus op(c)) =$	$op(c)$	OK
508	$(nop \oplus op(c)) \oplus op(c) =$	$op(c)$	$nop \oplus (op(c) \oplus op(c)) =$	$op(c)$	OK
509	$(op(a) \oplus op(c)) \oplus op(c) =$	$error$	$op(a) \oplus (op(c) \oplus op(c)) =$	$error$	OK
510	$(op(b) \oplus op(c)) \oplus op(c) =$	$error$	$op(b) \oplus (op(c) \oplus op(c)) =$	$error$	OK
511	$(op(c) \oplus op(c)) \oplus op(c) =$	$op(c)$	$op(c) \oplus (op(c) \oplus op(c)) =$	$op(c)$	OK

Bibliographie

[1] A. M. Dias, « Interfaces gráficas em CxProlog (usando as APIs do Java e do wxwidgets) », Rapport technique, CITI (Portugal), may 2006.

[2] A. M. Dias, « Ligação entre CxProlog e wxWidgets », Rapport technique, CITI (Portugal), apr 2005.

[3] D. H. D. Warren, « An abstract prolog instruction set », Rapport technique 309, SRI International, oct 1983.