UNIVERSITE DE TUNIS EL MANAR

المدرسة الوطنية للمهندسين بتونس
école nationale d'ingénieurs de Tunis

Unité de recherche en traitement du signal, traitement d'images et reconnaissance des formes
et
Laboratoire d'Informatique de Signaux et Systèmes de Sophia Antipolis

---

Mémoire

en vue de l'obtention du

Diplôme de Mastère

en

Automatique et Traitement du Signal

intitulé

Tissages Multiples d'Aspects d'Assemblage

Application à l'adaptation logicielle pour l'Informatique Ambiante

---

Élaboré par :

Sana FATHALLAH

Soutenu le 16/07/2009 devant le jury composé de **:**

M. Pr. GHAZEL Adel président
M. TIGLI Jean Yves membre
M. HAMROUNI Kamel membre
M. LAVIROTTE Stéphane membre

Résumé

L'Informatique Ambiante engage de nombreux dispositifs variés, intégrés aux objets de la vie quotidienne. Ces dispositifs peuvent collaborer pour faire émerger dynamiquement de nouvelles applications. Les applications logicielles s'adaptent alors en fonction des dispositifs disponibles dans l'infrastructure. Les travaux actuels proposent des mécanismes permettant de réaliser de telles adaptations pour l'informatique ambiante. Mais ceux-ci présentent des limitations soit sur les mécanismes de composition des adaptations, soit sur le modèle qu'ils utilisent pour spécifier ces adaptations.
Un problème complémentaire à traiter est l'étude des mécanismes permettant d'ajouter dynamiquement de nouvelles sémantiques dans la modification des applications. De plus, ces opérations doivent être indépendantes de tout langage utilisé pour l'adaptation, permettant ainsi d'ouvrir les possibilités d'adaptation des applications.

Abstract

Ambiant computing takes many various devices integrated to the object of everyday life. Those devices can collaborate to build dynamically new applications. Those applications adapt according to available devices in their execution context. The current works propose the mechanisms permitting to achieve such adaptations for the Ambiant computing. But they have limitations either on the mechanisms of composition changes, either on the model they use to specify these adaptations.

An additional problem to be addressed is the study of mechanisms to dynamically add new semantic changes in the applications. Moreover, these operations must be independent of the language used to adapt, thus opening the possibilities for adaptation of applications.

Dédicaces

Je dédie ce mémoire

A mes très chers parents qui ont toujours été là pour moi tout au long de mes études et qui m'ont donné un magnifique modèle de labeur et de persévérance. J'espère qu'ils trouveront dans ce travail toute ma reconnaissance et tout mon amour.

A mon mari pour sa patience et son soutien qu'il n'a cessé d'apporter au cours de ma formation

A ceux qui ont toujours cru en moi et qui n'ont cessé de me pousser vers L'avant.

Sana FATHALLAH

Remerciements

Sans l'aide et l'intervention de nombreuses personnes il m'aurait été difficile de réaliser ce travail ; je tiens donc à leur exprimer tous mes remerciements.

Je voudrais d'abord remercier M. Adel Ghazel, professeur à SUP'COM, pour avoir bien voulu accepter de présider le jury d'évaluation de mon mémoire de mastère.

Je tiens à remercier très chaleureusement Mr. Jean-Yves TIGLI et Mr. Stéphane LAVIROTTE pour leur encadrement sans faille et leur grande disponibilité malgré leurs nombreuses responsabilités. La réunion de chaque semaine entre nous m'a permis de continuer dans la bonne direction et de comprendre plus profondément le travail de la recherche.

Mes remerciements s'adressent aussi au Mr. Kamel HAMROUNI qui m'a donné l'opportunité de faire mon stage de mastère avec l'équipe RAINBOW.

Merci à tous les membres de l'équipe RAINBOW pour tous les moments que nous avons partagés, pour leur soutien et leur encouragement.

Sommaire

Liste de figures

Liste des abréviations

(AA)	Les Aspects d'Assemblage
AGG	Attributed Graph Grammar
AO4BPEL	Aspect-Oriented for Business Process Execution Language
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
BPEL	Business Process Execution Language
BSL	Behavior Specification Language
CAM/DAOP	Component and Aspect Model
CBSD	Component-based software engineering
DPO	Double-PushOut approach
EAOP	Event-based AOP
EJB	Entreprise Java Beans
GrGen.NET	Graph Rewrite GENerator,. NET
GXL	Graph eXchange Language
IAm	Informatique Ambiante
ISL	Interaction Specification Language
JPEG	Joint Photographic Experts Group
LCA	Lightweight Component Architecture
LHS	Left Hand Side
NAC	Negative Application Conditions
RHS	Right Hand Side
SAFRAN	Self-Adaptive Fractal CompoNents
SPO	Single-PushOut approach
TG	Transformation de Graphe
UDDI	Universal Description Discovery and Integration
XML	Extensible Markup Language

Introduction

Motivations

Dans notre vie quotidienne, les terminaux mobiles sont de plus en plus présents. Informatique, électronique et traitements informationnels deviennent ambiants, latents, omniprésents. Cette ubiquité se caractérise par le fait que l'informatique se « dissout » dans l'environnement qui nous entoure d'où la notion de l'**Informatique Ambiante** (IAm) qui a été introduite par MARK WEISER en 1991 [Weiser 91]. Son objectif était d'intégrer, de manière transparente, des dispositifs informatiques au monde physique et d'utiliser cette infrastructure dans les applications informatiques de la manière la plus transparente possible. L'informatique ambiante intègre donc les objets de la vie quotidienne auxquels l'utilisateur ne prête même plus attention en tant qu'entité numérique.

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 quotidienne (voiture, maison, etc.). Ces objets informatisés sont capables de construire un contexte pour l'application en fournissant à leur environnement des informations sur leur état tout en agissant également sur leur environnement.

L'IAm, de par sa nature même, impose alors certaines contraintes. En effet, un système informatique ambiant est utilisé dans un environnement qui se caractérise par de ***multiples*** dispositifs. Cette multiplicité implique par conséquent la prise en compte du problème ***d'hétérogénéité***, tant au niveau matériel que logiciel (langage de programmation, protocole de communication, etc.). Les dispositifs impliqués dans une application ne sont pas figés mais ils peuvent être ***mobiles*** dans un environnement fortement variable. À la diversité de dispositifs, s'ajoute l'apparition et la disparition dynamique qui ne peut être vu comme un problème, (cas de panne), mais bien comme une donnée du problème à traiter. Il est donc impossible de prévoir toutes les situations d'exécution d'une application. De ce faite, un système IAm doit supporter cette ***dynamicité***. La ***réactivité*** des systèmes à ces changements est un autre requis pour un système IAm.

Dans le domaine de l'informatique ambiante, nous nous intéressons donc à la capacité d'un système à ***s'adapter*** à son environnement. Les techniques actuelles qui assurent l'adaptation dynamique d'applications sont diverses et nous allons en faire une rapide présentation dans cette introduction.

Cadre de l'étude

Le cadre général de ce mémoire de master de recherche est l'adaptation dynamique des applications logicielles. L'adaptation des applications est actuellement un point important dans plusieurs domaines de l'informatique (IAm, mobile,…). Les mises à jour et l'évolution des applications ne peuvent être considérées au même titre que pour les applications traditionnelles car ces changements ne seront pas effectués pendant la conception, la

compilation ou le déploiement, mais bien lors de l'exécution même de l'application, sans redémarrage de celle-ci.

Il existe deux approches différentes pour l'adaptation logicielle. Elles sont illustrées dans la figure 0.1 par les lettre **A** et **B**. Dans la première boucle **A**, l'adaptation est déclenchée lorsque est atteint un point particulier dans l'exécution comme par exemple l'appel d'une méthode. Donc au moment de l'exécution nous obtenons une réponse à la question « *Où faire l'adaptation ?* » et le tisseur[1] fait le reste c'est-à-dire « *Comment faire cette adaptation ?* ». Plusieurs travaux dans la littérature font partie de cette approche. Nous les détaillerons dans le chapitre 1.

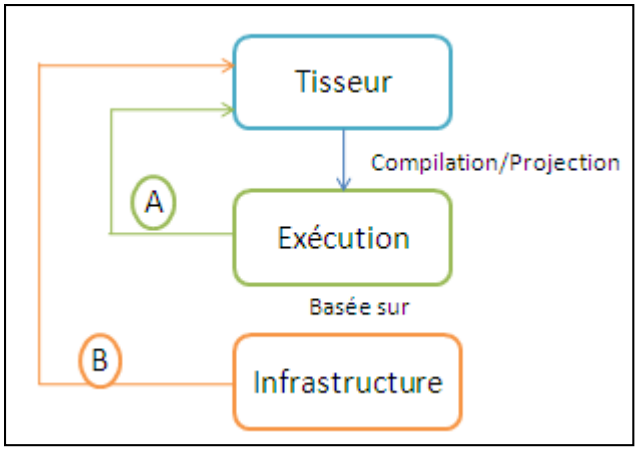


Figure 0.1 Modèles d'adaptation

Dans la boucle 2, marquée par la lettre **B**, l'adaptation est déclenchée par l'infrastructure c'est-à-dire suite à l'apparition d'un dispositif ou bien sa disparition par exemple. Il s'agit d'obtenir des applications qui s'adaptent à leur contexte d'exécution et se reconfigurent en fonction des changements du contexte. L'infrastructure doit donc déterminer « *Quand l'adaptation sera exécutée ?* ». Nous pourrons conserver les mécanismes décrits précédemment sur le « *Où ?* » et « *Comment ?* ».

La grande différence entre l'approche A et B réside dans le fait que dans l'approche A, l'application doit s'adapter en prenant en compte uniquement sa propre dynamique. Dans le cas B, elle doit réagir aux variations de l'infrastructure, donc à des sollicitations qui ont leur propre dynamique. L'application subit alors les modifications dues aux variations de l'infrastructure et doit s'y adapter tout en assurant une réactivité suffisante permettant l'utilisation de celle-ci. Cette dernière approche est la plus adaptée au domaine de l'informatique ambiante où nous avons une forte variation de l'environnement et une nécessité que les applications en tire parti.

Les adaptations des applications ambiantes s'inscrivent donc dans le cas **B** et plus précisément à l'intérieur de la boite « Tisseur ». En effet le mécanisme de tissage permet de calculer l'adaptation à mettre en place. La figure 0.2 montre ce mécanisme d'une manière générale.

[1] Tisseur : sont rôle est de calculer les modifications à effectuer

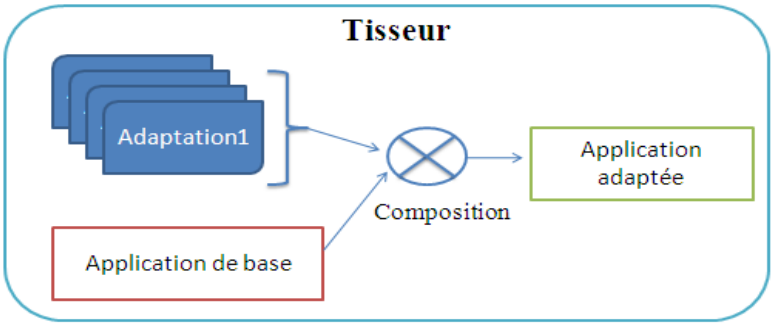


Figure 0.2 Tisseur d'adaptations

Une application en informatique ambiante est construite à base de composants. Elle est décrite sous la forme d'un assemblage de composants logiciels.
Généralement, nous avons besoin d'exécuter plusieurs adaptations en même temps car l'environnement est ouvert. Pour cette raison, le tisseur est doté d'un mécanisme de composition qui prend en entrée les modifications à appliquer et en utilisant l'application (assemblage de composant), il calcule l'adaptation finale qui représente l'application modifiée. Les adaptations à exécuter peuvent être en conflit entre elles, c'est-à-dire que les adaptations modifient l'application au même endroit (portant sur un même sous-ensemble de composants logiciels). Le mécanisme de composition est combiné avec un mécanisme de résolution de conflit. Mes travaux s'articulent autour de ces mécanismes de composition.

Problématique

Les adaptations, qui sont l'entrée du mécanisme de composition, sont basées sur un langage. Dans les travaux de l'équipe les plus récents, la composition se base sur des arbres syntaxiques mais le résultat de la composition est un assemblage de composants qui va être ajouté à l'assemblage initial. Donc nous sommes obligés de passer d'un langage (représentation arborescente) à un assemblage de composants (représentation de type graphe).
D'autre part, nous savons faire la composition avec des composants de sémantique connue. Mais si nous souhaitons introduire des ***nouvelles sémantiques*** pour la composition dans l'adaptation des applications, les difficultés rencontrées pour réaliser cette tâche sont multiples.
Dans le cadre de ce stage de Mastère, nous proposerons une modélisation des transformations par des graphes pour la spécifier les applications et leurs adaptations. Une description des architectures basées sur les graphes présente, outre l'aspect formel rigoureux, l'avantage d'offrir une description visuelle et facilement compréhensible qui ne nécessite aucun pré-requis concernant les langages formels. Tout nouveau besoin sera modélisé par des composants de sémantique nouvelle l'ajout de nouvelles règles dans le mécanisme de transformation de graphes.

Plan

Ce mémoire est organisé en trois parties :

- *Chapitre 1 :* Dans ce chapitre nous reprenons en détail le cadre général de nos travaux. Nous détaillons les travaux sur l'adaptation logicielle en utilisant le concept d'aspect

et nous nous intéressons en particulier aux mécanismes de composition des adaptations.

- *Chapitre 2 :* L'étude bibliographique effectuée dans le chapitre 1 nous amene naturellement au domaine de transformation de graphe. Dans ce chapitre, nous nous intéressons aux travaux qui traitent le sujet de l'adaptation par transformation de graphe mais plus précisément, nous vérifions s'il existe des mécanismes pour la composition des adaptations.
- *Chapitre 3 :* Dans ce chapitre, nous présentons notre contribution pour la composition des adaptations avec des composants de nouvelle sémantique, non imposée par un langage a priori. Nous présentons également les résultats obtenus et évoquerons l'implémentation réalisée.
- *Conclusion et perspectives*. Dans ce chapitre, nous identifions tout d'abord les apports essentiels de notre travail, ainsi que ses limites. Nous y exposons également les perspectives ouvertes par nos travaux.

Afin de faciliter la lecture de ce rapport et en particulier l'étude bibliographique, nous proposons des conclusions et synthèses (sous la forme de paragraphes encadrés en pointillé) des travaux étudiés les plus importants et représentatifs du domaine.

Chapitre 1 Cadre Général
Systèmes adaptatifs dynamiques

Comme nous avons pu le démontrer dans l'introduction, la capacité d'adaptation à l'exécution représente une exigence nouvelle et importante des logiciels modernes. Une manière très répandue d'adapter les applications est l'utilisation du concept d'aspect issu de l'AOP (*Aspect-Oriented Programming*). L'AOP définit une séparation des préoccupations techniques des descriptions métier dans une application. Cette séparation a permis l'utilisation de l'aspect dans les travaux sur l'adaptation logicielle en considérant l'aspect comme une unité d'adaptation. Ceci a permis de rendre les adaptations indépendantes de l'application elle-même. C'est la propriété qui permet aux programmeurs de prévoir des adaptations sans connaître de manière spécifique les applications qui seront déployées.

Dans ce chapitre nous dressons une cartographie de l'utilisation du concept d'aspect pour l'adaptation dans les différents paradigmes. Pour chaque approche, nous nous intéressons particulièrement aux mécanismes de composition s'ils existent. Cette étude nous amènera vers l'approche la plus développée dans ce domaine. Nous projetterons donc notre problématique dans cette approche pour dégager les limites, ce qui nous permettra de soulever notre problématique.

1.1 AOP (Aspect-Oriented Programming)

Le but de cette section est d'introduire la programmation orientée aspect telle qu'elle a été définie par Kiczales [Kiczales et al. 01]. Pour ce faire, nous commençons par une présentation générale de cette approche, puis nous proposons un survol du langage AspectJ, qui est devenu une référence pour l'AOP.

1.1.1 Concepts généraux

Les concepts de la programmation orientée aspect ont été formulés initialement par Kiczales et son équipe en 1997. L'AOP est une méthode de programmation qui permet de séparer l'implémentation de toutes les exigences, fonctionnelles ou non, d'un logiciel [Kiczales et al. 01]. Le principe est donc de programmer chaque problématique séparément et de définir leurs règles d'intégration en vue de former le système final. Cette disposition favorise la réutilisation et permet ainsi la réduction des délais de développement et de maintenance.
Cependant, la réutilisation n'est pas toujours aisée [Bouraquadi et al. 01]. C'est notamment le cas pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de services donnés, mais nécessite également la prise en compte de différentes propriétés non-fonctionnelles telles que la distribution, la persistance, etc. Dans de telles applications, l'implémentation des services réalisés et celle des différentes propriétés non-

fonctionnelles se trouvent intimement enchevêtrées (figure 1.1.a). De ce fait, la réutilisation se trouve compromise.

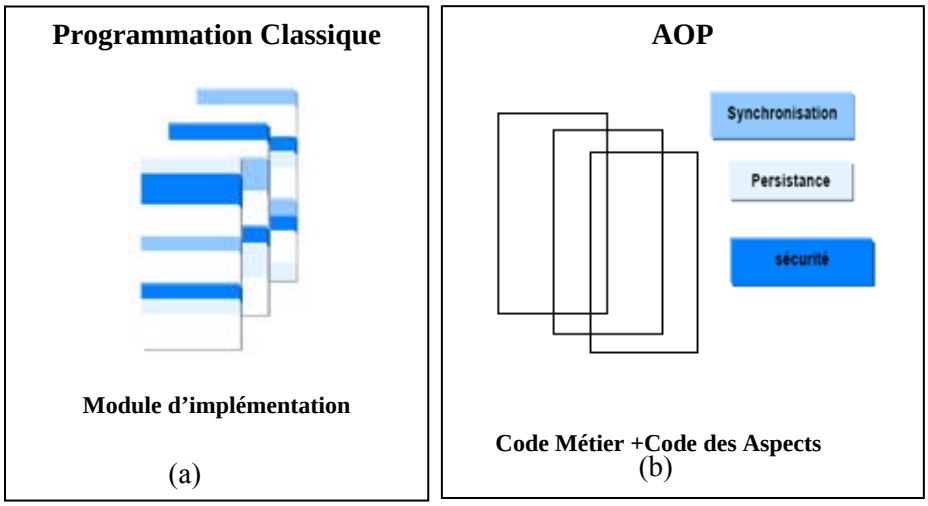


Figure 1.1 programmation classique vis AOP

Dans le but de permettre une meilleure réutilisation et de minimiser la dispersion du code, le paradigme de l'AOP propose de structurer les applications sur la base du concept d'aspect (figure 1.1.b).

Un *aspect* est une abstraction qui définit une structure et un comportement qui se superposent à l'application. Les propriétés non fonctionnelles (sécurité, persistance,…) peuvent être définies par des aspects dont la structure est présentée par la figure 1.2.

Figure 1.2 Modèle général d'un Aspect

Point de coupe (angl *Pointcut*) désigne un ensemble de points de jonction du programme de base à adapter où l'aspect va intervenir.

Point de jonction. (angl *Join Points*) La composition des aspects avec le programme de base se traduit par l'établissement d'une jonction entre ces unités de modularisation [Hachni 02]. Un point de jonction est donc 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.). À un point de coupe correspond un ou plusieurs points de jonction.

Greffon (angl. *advice*) permet de décrire les actions à effectuer lorsqu'un point de coupe est identifié.

La programmation orientée aspect consiste donc en un ensemble de points de coupe (qui produisent des points de jonction), de greffons et un mécanisme de *tissage* dont le rôle est de calculer les modifications à effectuer et ceci en injectant le code du greffon dans les points de jonction identifiés.

Ayant présenté en détail la programmation par aspects, nous présentons dans ce qui suit, AspectJ qui est un exemple de langage de programmation par aspects.

1.1.2 AspectJ

AspectJ est aujourd'hui une référence pour la programmation par aspects telle que définie par Kiczales. C'est un langage qui étend Java et permet la définition des aspects. Un tisseur est chargé de « mélanger » le code du programme de base écrit en Java avec celui des aspects afin de générer une application complète [Pawlak et al. 05]. Ce langage représente l'application des aspects dans le paradigme de l'Orienté Objet.

- **Modèle d'Aspect**

La définition d'un nouvel aspect en AspectJ est très semblable à la création d'une nouvelle classe [Pawlak et al. 05]. Le mot-clef utilisé est ***aspect***. Dans un aspect nous définissons les points de coupes et les codes du greffon (advice) qui induiront les fonctionnalités liées à cet aspect. La représentation d'un aspect est donnée par la figure 1.3. La gestion des aspects (instanciation et destruction) se fait automatiquement selon le besoin de l'application (le programmeur n'a pas le droit d'instancier un aspect).

```
public aspect Nom_Aspect {
  pointcut Nom_Method() : // code
// Greffon
before():Nom_Method() {
  // code
  } }
```

Figure 1.3 Modèle Général d'un Aspect en AspectJ

Le mot clé ***pointcut*** permet de définir une coupe qui regroupe un ou plusieurs points de jonction. AspectJ supporte différents types de points de jonction qui sont [Pawlak et al. 05] :

- Appel d'une méthode, exécution d'une méthode
- Appel d'un constructeur, exécution d'un constructeur
- Exécution d'un greffon
- Lecture ou écriture d'un attribut
- Exécution d'un bloc de codes statiques particulier
- Initialisation, exception

Chaque action définie par le greffon doit être précédée par la définition du moment de son exécution : avant l'événement (**before)**, après (**after)** ou autour de l'événement (**around)**. Le mot-clef **around** permet soit de remplacer l'instruction en elle-même, soit d'exécuter du code avant et après le point de jonction.

- **Mécanisme de tissage**

Le tisseur AspectJ prend en entrée des aspects, l'application et les directives de tissage et produit dans la sortie l'application augmentée des aspects [Levy 06].

Le mécanisme de tissage en AspectJ est statique. Les aspects sont tissés au moment de la compilation directement dans le code de l'exécutable ce qui génère soit un fichier source .java ou directement un fichier bytecode .class. Les tisseurs statiques ont moins de contraintes de

temps d'exécution que les tisseurs dynamiques [Levy 06], mais il y a un risque d'allonger la phase de compilation. L'inconvénient majeur du tissage statique est qu'une fois le tissage terminé, nous retrouvons la même application que dans l'approche OO classique (dispersion du code). Par conséquence, l'apport de l'utilisation d'aspect est perdu au moment de l'exécution.

Kiczales démontre qu'il est possible de minimiser la dispersion du code en regroupant les préoccupations transverses (sécurité, persistance, etc.) dans des entités réutilisables appelées *Aspect*. Ces dernières sont intégrées dans l'application par le mécanisme du tissage. Plus tard, Kiczales a développé une implémentation des concepts nécessaires pour la programmation orientée aspect qui est le langage AspectJ. L'inconvénient dans cette implémentation est que le mécanisme de tissage est statique, nous perdons alors au moment de l'exécution la séparation des préoccupations de l'application de base. Une autre limite de ce modèle est le manque de mécanisme de traitement des interactions entre les aspects. Il n'y a aucun mécanisme pour la composition des aspects. Les aspects sont appliqués d'une manière séquentielle. De ce faite, l'application tissée peut avoir des comportements non souhaités dans le cas où nous avons plusieurs aspects qu'ils s'appliquent au même endroit.

1.2 EAOP (Event-based AOP)

Les travaux de Douence [Douence et al. 02] [Douence et al. 03] forment une extension de l'AOP classique afin de pallier aux limites identifiées précédemment. En effet, Douence propose à travers EAOP une approche qui est basée sur l'observation des événements d'exécution [Douence et al. 02]. Les aspects peuvent être en effet vus comme des propriétés sur les traces d'exécution du programme et sont exprimés au moyen de ces événements. Ils n'ont pas le pouvoir de changer les variables de l'application de base et peuvent seulement sélectionner ou couper une trace d'exécution de l'application.

Un autre apport est démontré dans EAOP : la résolution de conflit entre aspects. EAOP implémente deux types de commandes de résolution de conflits : d'abord, la composition explicite des greffons au même point de 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 (aspect d'aspect).

- **Modèle d'Aspect**

Un aspect est vu comme un transformateur d'événement. En fait, un aspect est un programme Java qui prend un événement comme paramètre d'entrée, exécute un certain calcul (qui peut modifier l'événement), et il reste en attente de prochains événements.

Un aspect est défini par héritage de la classe abstraite *Aspect* et en définissant la méthode *définition*. Cette méthode n'impose pas de contraintes à la structure du code et utilise la méthode *nextEvent* afin d'obtenir l'événement suivant. Cette dernière méthode bloque l'aspect et attend que le moniteur l'active avec l'évènement actuel.

```
class Nom_Class extends Aspect {
//définition du point de coupe
Type Mthode(object c) {
//code
}
public void definition(){
}}
```

Figure 1.4 Modèle Général d'un Aspect selon EAOP

Dans EAOP, les points de coupes sont définis en utilisant les événements Java déclenchés au moment de l'exécution [Brichau et Haupt 05]. De ce faite, les points de jonctions sont les événements d'exécution qui peuvent être, par exemple, des appels de méthode, l'écriture ou la lecture d'une variable, des points d'entrée dans une branche conditionnelle, etc.

- **Modèle de tissage**

Le tissage d'aspect est dynamique et réalisé à l'aide d'un *moniteur d'exécution* [Douence et al. 02] qui permet d'évaluer les points de coupe et exécuter par la suite le greffon correspondant.
Le moniteur d'exécution fonctionne simultanément au programme de base. Il observe son exécution et intercepte les événements de l'exécution définis par les aspects. Il tisse dynamiquement les aspects correspondants à l'événement en cours sur la séquence d'événements qui sont définis par les points de coupe et exécute le greffon relatif.

La contribution essentielle dans EAOP est la prise en compte de la dynamicité pour les aspects en se basant sur un mécanisme de tissage dynamique confié à un moniteur d'exécution. Ce dernier capte les événements émis durant l'exécution et déclenche les aspects appropriés. Les point de coupes sont donc plus évolués et exprimés en termes d'événements. Une deuxième contribution proposée par Douence, est ***la résolution de conflit*** entre les aspects par l'application d'aspect sur des aspects ou bien par la composition explicites des aspects. Malgré que dans ces travaux Douance à traité le problème de composition des aspects, la solution qu'il propose reste encore limitée car en informatique ambiante nous ne savons pas à l'avance les adaptations qui vont être en conflit. Nous ne pouvons pas donc utilisé la composition explicite des aspects.

1.3 AO4BPEL (Aspect-Oriented for Business Process Execution Language)

Dans les sections précédentes, nous avons présenté des modèles proposants des apports différents en termes d'architecture. Mais dans ces travaux, le sujet de l'adaptation logicielle n'est pas traité. A partir de cette section, nous présentons l'aspect comme unité d'adaptation dynamique et nous verrons si le sujet de compositions des adaptations est traité ou non.
AO4BPEL [Charfi et al. 04] emprunte la notion d'aspect pour l'appliquer au domaine des Web service [Alonso et al. 04] qui est devenu la technologie pour intégrer des applications distribuées et hétérogènes sur l'Internet. AO4BPEL vient pour corriger les limites des

approches d'orchestration [Peltz 03] classiques qui sont : (a) manque de moyens pour la représentation de préoccupations transverses et (b) pas d'adaptation dynamique de la composition de service. D'où le besoin d'avoir un langage plus flexible pour la composition des Web services.

AO4BPEL fournit des mécanismes pour la composition dynamique des Web services : les aspects peuvent être branchés ou débranchés dans le processus de composition des services au moment de l'exécution. Dans ce qui suit nous présentons l'aspect selon AO4BPEL.

1.3.1 Modèle d'Aspect

La séparation de l'activité principale de la logique de composition de service est réalisée grâce aux aspects. Ces derniers sont définis sous la forme d'un document XML (Extensible Markup Language) (figure 1.5).

```
<aspect name="Nom_Aspect">
<pointcutandadvice type="after">
 <pointcut name="nom_coupe">
// Définition de la coupe
  </pointcut>
  <advice>
// Définition du greffon
   </advice>
</pointcutandadvice>
</aspect>
```

Figure 1.5 Modèle Général d'un Aspect en AO4BPEL

Dans AO4BPEL chaque processus est composé d'un ensemble d'activités. Un processus peut être représenté sous la forme d'un graphe (figure 1.6). Dans ce formalise les nœuds représentent des activités et les liens représentent les flux entre les activités [Charfi 04]. Ces activités constituent donc l'ensemble des points de jonction qui sont décrits en utilisant le type et les attributs de l'activité [Brichau et Haupt 05].

Les *points de coupe* permettent de référencer l'ensemble des points de jonction qui couvrent plusieurs processus métiers dans lesquels des fonctionnalités transversales doivent être exécutés. Le langage utilisé pour définir les points de coupe est XPath [Clark et DeRose 99]. Nous présentons un exemple de point de coupe permettant la sélection des invocations qui font un appel à l'opération *foo* et dont le type de port est *ws1PT* depuis tous les processus déployés dans le moteur BPEL :

```
//invoke[@operation=foo and @portType=ws1PT]
```

Un *advice* en AO4BPEL est une activité spécifié dans BPEL [Curbera et al. 03] qui doit être exécuté avant, après ou au lieu d'une autre activité [Charfi et al. 04]. La figure 1.6 illustre l'application d'un *advice* de type around. Le nœud f, qui est sélectionné comme un point de jonction, sera précédé par les activités i et j et suivi par l'activité k.

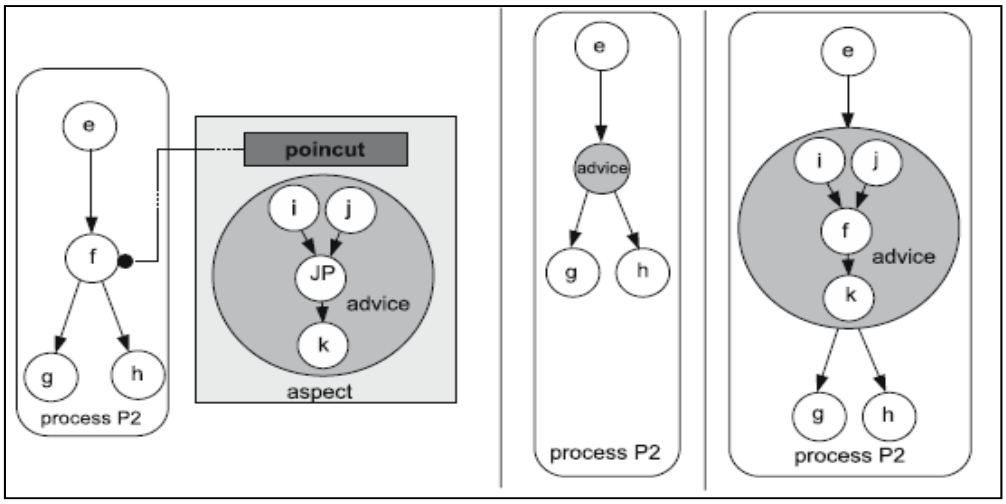


Figure 1.6 Exemple d'application de greffon « around »

La spécification AO4BPEL prend en compte les erreurs qui peuvent apparaître au cours de l'exécution du greffon. De ce faite, le greffon en AO4PBEL est intégré dans une activité de type *scope* qui définit une exception. Cette disposition garantie qu'aucune erreur ne peut se propager d'un greffon vers un processus [Charfi 04].

1.3.2 Modèle du tissage d'aspect

La spécification AO4BPEL définit une plate-forme Java non distribuée de services ainsi que des moyens permettant de réaliser le déploiement des fournisseurs et des demandeurs de services à l'intérieur de la plate-forme. L'extension apportée par AO4BPEL est le composant ***« Aspect Runtime »*** qui est le tisseur dynamique [Charfi 04].

Chaque activité possède un cycle de vie : *activate-enable-run-complete*. La vérification des aspects, qui correspond au point de jonction de l'activité en cours, se réalise avant l'état *Run* et après l'état *complete* de l'activité. À cet effet, l'interprète de BPEL passe des métadonnées au sujet de l'activité courante (nom de processus, nom d'activité, activité de parent, type d'activité, attributs d'activité, variables, etc.) au « *Aspect Runtime* ». Ce dernier tient une liste de tous les aspects actuellement déployés et vérifie s'il y a d'autres aspects pour l'activité courante exécutée par l'interprète BPEL.

Charfi avec AO4BPEL combine les notions d'aspect et de service dans le but de répondre à la séparation de l'activité principale de la logique de composition du service. Les aspects peuvent être utilisés au moment de l'exécution ce qui garanti la dynamicité qui est renforcée par l'utilisation d'un mécanisme de tissage dynamique (confié à un composant qui est ***Aspect Runtime***). Le problème de composition des aspects n'a pas été traité dans les travaux de Charfi. Dans l'écriture de ces aspects il a utilisé le mécanisme classique de l'AOP : Before, After. Le conflit qui peut être déclenché entre les aspects n'est pas résolu.

1.4 SAFRAN (Self-Adaptive Fractal CompoNents)

SAFRAN est une plate-forme 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 qui est appelé *aspect d'adaptation* [David 05]. Ce dernier est sous la forme de politiques réactives qui détectent les évolutions du contexte d'exécution et adaptent le programme de base en le reconfigurant. Ce dernier doit être développé séparément du reste de l'application en utilisant un formalisme spécifique. Ainsi, il peut être intégré ou retiré dynamiquement du système.

SAFRAN est constitué (a) d'un langage dédié permettant de programmer l'aspect d'adaptation sous la forme de politiques réactives, et (b) du support d'exécution nécessaire au tissage et à l'exécution de ces politiques (aspects) dans les composants Fractal (programme de base).

1.4.1 Modèle d'Aspect

SAFRAN [David 05] reprend le schéma générique d'un aspect (point de coupe et greffon). L'aspect d'adaptation est le regroupement composé des coupes (correspondant à la notification d'événements endogènes ou exogènes), des actions (restreintes à la reconfiguration de l'architecture) et des politiques d'adaptation modulaires qui sont structurées comme un ensemble de règles réactives de la forme : when <event> if <condition> do <action>.

Une règle d'adaptation indique que lorsqu'un événement correspondant à l'expression <event> se produit, si l'expression <condition> est vraie, alors la reconfiguration <action> est exécutée, adaptant ainsi le système à la nouvelle situation résultant de l'occurrence de l'événement.

Dans le système SAFRAN, les politiques d'adaptation associées dynamiquement aux composants Fractal adaptatifs sont constituées de séquences (ordonnées) de règles d'adaptation :

```
Policy example {
rule { when <event1> if <condition1> do <action1>}
rule { when <event2> if <condition2> do <action2>}
… …
}
```

Figure 1.7 Modèle Général d'un Aspect (politique adaptative) SAFRAN

Les notifications d'événement décrites dans une politique constituent les points de jonctions. Le modèle SAFRAN fait la distinction de deux types d'événements : *endogène* et *exogène*. Un évènement endogène correspond à un point d'exécution dans le programme de base qui est un ensemble de composants Fractal. Il 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 évènement exogène est une réification des informations implicites du contexte d'exécution de l'application par l'utilisation de la plate-forme WildCAT [David 05] pour le développement d'applications sensibles au contexte.

Une coupe est la description de la notification d'un évènement 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'évènements provenant du contexte ou de l'application (c'est une extension du modèle EAOP). Suite à la notification d'un événement endogène ou exogène, une ou plusieurs actions de reconfiguration sont exécutées. Une action de reconfiguration est un programme écrit dans le langage procédural simple FScript [Van Deursen et al. 00] 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.

1.4.2 Modèle du tissage

La plate-forme Fractal a du être étendue afin de pouvoir associer dynamiquement des politiques d'adaptation aux composants d'une application. Cette extension est traduite par la spécification d'une nouvelle interface de contrôle : le *contrôleur d'adaptation* [David 05]. Ce mécanisme d'association dynamique constitue le mécanisme de tissage de l'AOP.

SAFRAN permet d'associer les politiques d'adaptation aux composants métiers d'une application d'une façon dynamique. Le tissage est donc confié au contrôleur d'adaptation. Un composant SAFRAN intègre grâce à cette interface le code d'adaptation lui-même et devient ainsi adaptatif, c'est-à-dire autonome et acteur de sa propre adaptation.

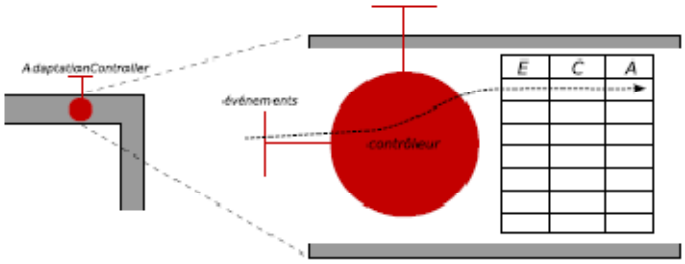


Figure 1.8 Le contrôleur d'adaptation

A la réception d'un événement par le contrôleur, ce dernier détermine la réaction appropriée en fonction des différentes règles puis il exécute cette réaction pour adapter le composant aux nouvelles circonstances.

SAFRAN est un modèle d'adaptation de composant Fractal. L'utilisation de composants offre une modularité sans laquelle toute adaptation est difficile. David définit explicitement l'aspect comme une entité d'adaptation d'où l'appellation *Aspect d'adaptation.* L'adaptation est considérée donc comme préoccupation transverse à l'application. Le déclenchement des aspects est réalisé par les événements endogènes et exogènes, ce qui permet de prendre en compte l'évolution du contexte d'exécution dans des environnements variés. Le tissage dans SAFRAN est dynamique, ceci assurant la dynamicité de l'adaptation. Chaque composant intègre un contrôleur d'adaptation qui contient une politique d'adaptation réactive (événement- condition- action). Dans ce modèle d'adaptation nous pouvons avoir un conflit entre les actions à exécuter. Ce type de problème n'a pas été posé dans SAFRAN.

1.5 CAM/DAOP : Component and Aspect Model

CAM/DAOP [Pinto et al. 05] est une plateforme qui propose un modèle d'adaptation logicielle en combinant CBSD [2]et AOSD (Aspect-Oriented Software Development), qui sont deux technologies complémentaires, afin d'augmenter la modularité et l'évolution des systèmes. L'ensemble de composants et d'aspects (qui sont les entités de base) d'une application sont indépendants les uns des autres c'est-à-dire sans aucun mécanisme de composition entre eux (augmenter la réutilisabilité de ces entités). Le modèle CAM/DAOP décrit la structure d'une application en termes d'aspects, de composants et de mécanismes de communication qui définit la relation entre les aspects et les composants (stockée au niveau de la plateforme). Cette information est consultée au moment de l'exécution pour établir le lien entre aspect et composant et peut être modifié au cours de l'exécution ce qui constitue l'adaptation dynamique de l'application et modifier donc la sémantique de l'application.

L'aspect selon CAM/DAOP est différent de l'aspect de l'AOP classique. Il existe une séparation entre la définition de greffon et la spécification de point de coupe qui le lie avec les composants. De ce faite, Ils proposent une description de modèle de point de coupe séparément de celui de l'aspect.

1.5.1 Modèle de Point de coupe

Dans CAM les composants sont considérés comme des boîtes noires. Les points de coupe interceptent donc le comportement interne d'un composant à travers son interface publique. Les points de coupe sont définis dans un fichier XML en dehors de l'aspect et du composant. CAM définit quatre points de coupe qui sont : création du composant, suppression d'un composant, envoie de message (aussi réception) et réception d'un événement. Les points de jonction associés à ces coupes sont: BEFORE_NEW, AFTER_NEW, BEFORE_DESTROY, AFTER_DESTROY, BEFORE_SEND, BEFORE_ RECEIVE, AFTER_THROW, SEND_EVENT.

La spécification d'une coupe (selon le type) nécessite la définition du composant source et son rôle, la signature de message, le composant cible et son rôle, signature d'événements

1.5.2 Modèle d'Aspect

Un aspect en CAM/DAOP est une classe JAVA qui contient (1) deux attributs qui sont le rôle (ligne 2) que joue l'aspect et la référence (ligne 3) vers la plateforme (2) un constructeur avec deux paramètres (ligne 4). Le comportement de l'aspect est défini par des méthodes qui constituent la partie greffon selon l'AOP classique (ligne 8 et 9).

```
1. public class MyAspect{
2. String role;
3. DAOPPlatform daop;
4. MyAspect(String role, DAOPPlatform daop){
5. this.role = role;
6. this.daop = daop;
7. }
8. public eval(PointCutIntrospection pc){…}
9. public evalBEFORE_SEND(PointCutIntrospection pc) {...}}
```

[2] CBSD(Component-based software engineering) est une branche du génie logiciel qui définit un système par décomposition en des composants fonctionnel et des composants logique.

Le greffon dans CAM/DAOP définit une méthode pour chaque type de point de jonctions même si l'aspect est indépendant de cette jonction (dans ce cas la méthode sera vide). Les signature des méthodes varient et peuvent prendre plusieurs formes qui sont : eval <pointdejonction>(), evalBEFORE_SEND, evalAFTER_NEW(), etc. Quand un point de jonction est intercepté, la méthode de greffon correspondante est exécutée. Le greffon eval est toujours exécuté quelque soit le type de la jonction.

1.5.3 Modèle de tissage

CAM/DAOP offre un mécanisme de tissage dynamique qui se décompose en deux étapes : (1) chargement du fichier XML qui contient la spécification des points de coupe qui va être appliquée aux éléments de l'application instanciée (2) consultation et utilisation de cette spécification durant l'exécution pour appliquer les adaptations associés.

Le modèle de tissage utilise le mécanisme de réflexion de JAVA. Les composants sont instanciés et communiquent entre eux par appel de service offert par la plateforme DAOP. A chaque instant un de ces services est utilisé et (1) une nouvelle instance de composant est créé (2) une instance est supprimée (3) un message est envoyé ou bien reçu (4) un événement est déclenché (5) une exception est levé, la plateforme DAOP intercepte le point de jonction et consulte les informations pour savoir quels aspects sont concerné par cette jonction et exécute le traitement associé.

Contrairement au modèle SAFRAN, qui intègre l'adaptation dans le composant, CAM/DAOP met en évidence un modèle d'adaptation dynamique qui sépare le composant de l'aspect. L'adaptation est paramétrée en interceptant des communications entrantes et sortantes entre composants via les aspects.
L'aspect selon CAM/DAOP contient seulement la définition de greffon, la coupe est définie dans un fichier XML en dehors de l'aspect ce qui augmente la réutilisabilité de cette entité. L'adaptation consiste donc à modifier la définition de la coupe c'est à dire la composition des aspects et des composants au moment de l'exécution ce qui permet de changer la sémantique de l'application (ajouter ou supprimer des propriétés). Dans le modèle CAM/DAOP, le problème de conflit entre aspect n'est pas traité.

1.6 Les Aspects d'Assemblage (AA)

Daniel [Cheung-Foo-Wo 09] propose un modèle d'adaptation utilisant un concept original appelé Aspect d'Assemblage (AA). Les notions des aspects classiques sont toujours valables pour un AA mais avec quelques nuances. Un AA modifie soit la structure d'un assemblage de composants représentant une application locale, soit la structure d'un assemblage encapsulée dans un service composite. 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.

L'utilisation des AA permet de construire entièrement une application. Les greffons ne sont plus rattachés à une application de base car en informatique ambiante, l'application de base

n'existe pas mais elle est construite en fonction de l'environnement. Ainsi, l'expression des adaptations doit être la plus générale possible.

Comme un aspect classique, un AA est formé d'un point de coupe et d'un greffon. Un point de coupe permet de désigner des endroits spécifiques de l'assemblage. Daniel Cheung a utilisé les expressions régulières dans la définition de la coupe. Un point de coupe est donc une succession de filtres sur les composants de l'application [Cheung-Foo-Wo 09]. Ces filtres prennent en entrée la liste de tous les composants de l'application et produisent une ou plusieurs listes de composants.

<variable>:=(nom du filtre) parametres ; [| autre filtre ...]
isalon := /i*/

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.

Le greffon d'un aspect d'assemblage selon Daniel Cheung désigne la structure d'une application. Il a définit deux langages pour le greffon : ISL (Interaction Specification Language) et BSL (Behavior Specification Language). Le premier permet de gérer la concurrence au niveau de la diffusion d'événements et de la réécriture de méthodes. La structure du langage est classique : un membre gauche spécifiant le port d'un composant virtuel et un membre droit spécifiant un assemblage de composants. BSL permet de gérer la concurrence au niveau des appels de méthodes. La structure s'inverse : le membre droit est l'entrée d'un composant et le membre de gauche un assemblage. Un conflit est détecté dans le premier cas lorsque deux membres gauches sont associés à un même port d'un composant de l'assemblage de base et dans le second cas, lorsque deux membres droits sont associés à la même entrée d'un composant de l'assemblage de base.

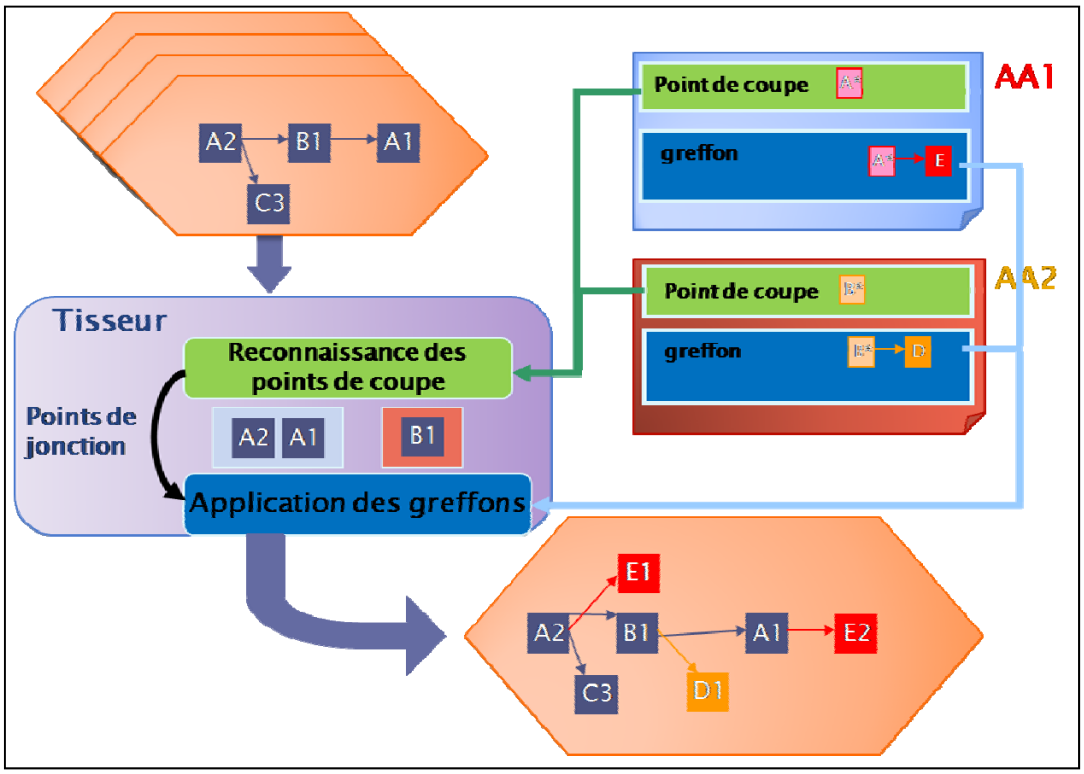


Figure 1.9 Tissage des Aspects d'assemblage

[Cheung-Foo-Wo 09] propose un mécanisme de tissage (figure 1.9) des AA qui 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. Le tissage d'un aspect consiste en une première étape de reconnaissance des points de coupe. Il s'agit d'une fonction qui, à partir de l'ensemble de composants de l'assemblage initial, produira un ensemble de points de jonctions. Lorsque ces points de jonctions sont identifiés, à l'aide du greffon de l'aspect d'assemblage, qui est un ensemble de règles portant sur ceux-ci, des instances de greffons peuvent être générées. Ceci en remplaçant les composants décrits dans ces greffons par les points de jonctions, s'il ne s'agit pas de composants instanciés par le greffon lui-même.
Parallèlement à cela, l'assemblage initial A0 est également transformé en une instance de greffon en vue d'être superposée avec celles créées par la fabrique. Le mécanisme de superposition considère donc un ensemble d'instances de greffon en vue d'en générer un unique, représentant l'assemblage final. Il s'agit de prendre indépendamment chaque instance de greffon et de les superposer les unes après les autres avec l'instance correspondant à l'assemblage initiale A0. Il est important de noter que l'assemblage obtenu après superposition ne servira pas par la suite d'assemblage initial. La superposition permet de respecter les propriétés d'associativité, de commutativité et d'idempotence lors de l'application des aspects d'assemblages.

La construction d'une application peut alors se voir comme l'application successive d'AAs construisant au fur et à mesure l'application finale à partir d'un assemblage de départ (l'apparition de nouveaux composants déclenchant le tissage de nouveaux aspects).

Plusieurs AA peuvent être applicables en même temps. De ce fait, il peut se produire un conflit entre les instances de greffon (qui veulent s'appliquer sur les mêmes instances dans l'assemblage des composants). Comme solution à ce type de problème, [Cheung-Foo-Wo 09] propose un mécanisme de résolution de conflit basé sur le mécanisme de fusion inspiré des règles de fusion ISL [Berger01]. Il a construit, à partir du modèle de Berger, un nouveau modèle de fusion d'interactions qui s'applique au modèle de composant. Il a proposé également la fusion pour le greffon écrit en BSL.

La contribution essentielle apportée par [Cheung-Foo-Wo 09] est la définition d'un mécanisme de composition des adaptations avec la résolution de conflits. Ce mécanisme repose sur la sémantique des opérateurs définis dans le langage et est indépendant des AA qui vont être appliqués pendant une adaptation.

1.7 Synthèse

Le concept d'aspect d'assemblage illustre les travaux les plus évolués en utilisant l'aspect pour l'adaptation. Le problème de composition des adaptations est posé clairement dans les travaux sur les AA. Le tisseur des AA est doté d'un mécanisme de composition qui travaille sur les instances de greffon et l'assemblage initial pour produire l'assemblage final. Il intègre

aussi un mécanisme de résolution de conflits dans le cas où nous avons plusieurs adaptations en un même point de jonctions.

La problématique de notre travail identifiée dans l'introduction serait donc d'étendre le modèle des AA pour supporter l'ajout des nouvelles sémantiques dans la modification des assemblages de composants. Or le mécanisme de composition réalisé est limité au langage défini et utilisé par les AA. Il est donc très difficile de l'étendre à de nouveaux composants de sémantique connue.

De plus, nous pouvons identifier un autre problème lié aux AA : la représentation des AA n'est pas homogène avec la représentation et la construction de l'application. Les AA sont spécifiés dans un langage alors que nous travaillons sur des assemblages de composants qui sont proches d'une représentation de graphes. Il est donc nécessaire d'effectuer des transformations d'un graphe vers un langage (ou arbre de syntaxe abstraite) puis du langage vers une modélisation sous forme de graphe. Nous pensons qu'il serait pertinent de rester plus proche du modèle d'exécution, donc au niveau d'un modèle de graphes (figure 1.10). Il pourrait alors être intéressant d'expliciter les modifications (adaptations) à apporter à l'application sous forme de graphes et la composition aurait alors comme entrée des graphes.

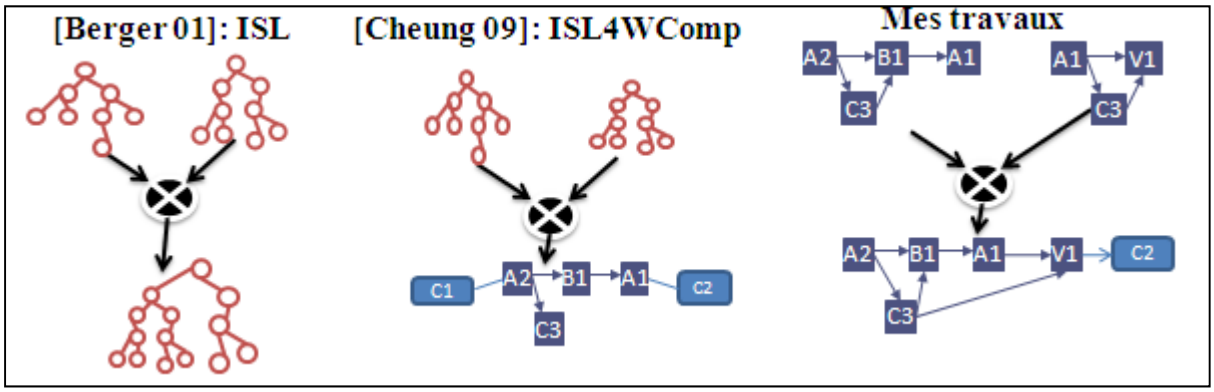


Figure 1.10 Evolution de mécanisme de composition

Nous détaillerons donc dans le chapitre suivant les travaux qui ont été réalisés dans le domaine de l'adaptation par transformation de graphe (TG) et nous serons particulièrement attentifs aux mécanismes de composition des adaptations.

Chapitre 2

Adaptation dynamique par TG

L'adaptation des applications par transformation de graphe est une approche orientée-modèle pour une adaptation structurelle automatique pendant l'exécution. Cette approche est basée sur des techniques orientées-règles pour la reconfiguration architecturale dynamique. Ainsi, chaque étape élémentaire de l'évolution de l'architecture est décrite par une combinaison de règles de transformation de l'architecture. Chacune de ces règles spécifie les actions de transformation élémentaires.

Nous exposons, dans ce chapitre dédié à l'état de l'art, différents travaux de la littérature qui se situent dans le domaine de l'adaptation dynamique par transformation de graphe. Nous nous intéresserons plus précisément au mécanisme de compositions et de résolution de conflits entre les règles d'adaptations. Pour situer le cadre formel de notre contribution, nous introduisons, ici, le modèle formel des transformations de graphe.

2.1 La théorie de la transformation de graphe

La réécriture de graphes [Hartmut 99] est un mécanisme pour transformer des graphes d'une manière mathématique rigoureuse. Ce formalisme a été introduit à la fin des années 60 pour traiter des problématiques variées telles que la construction de compilateurs, la reconnaissance de formes, l'intelligence artificielle, etc. Cette variété de domaine d'application est dû au fait que les graphes sont un outil puissant pour modéliser des situations complexes à un niveau intuitif.

Il existe plusieurs approches pour la transformation de graphe [Ehrig 99] parmi lesquelles:

- **Node label replacement approach** : principalement développé par Rozenberg, Engelfriet et Janssens, permet de remplacer un seul nœud (le côté gauche L est un nœud) par un graphe arbitraire R. La règle d'intégration est déterminée en fonction de l'étiquette du nœud [Hartmut] . L'adaptation dynamique que nous visons ne se limite pas au remplacement d'un seul nœud. Cette approche ne permet donc pas de couvrir tous types de problème vu le faite que la transformation est très limitée. Nous ne nous intéressons donc pas à cette catégorie de transformation puisque elle ne permet pas d'exprimer les adaptations souhaitées.
- **Algebraic approach:** L'approche algébrique est basée sur la dérivation (*pushout*) des graphes. En fait, il existe deux grandes variantes de l'approche algébrique : le double et le simple *pushout*.
- **Programmed graph replacement approach :** Développé par Schurr sous la forme d'un programme permettant de contrôler le choix non déterministe de la règle de

transformation à appliquer. Dans cette approche, Schurr utilise un programme pour tracer le flux d'application des règles de transformation.

D'après la description des différentes catégories de transformation de graphe, nous optons pour l'étude de l'approche algébrique. Ce choix est justifié par le faite que cette approche ne se limite pas à des graphes particuliers, mais elle a été généralisée à une grande variété des différents types de graphes et des types de structures de haut niveau, comme les graphes typés, les hypergraphes, les graphes avec attributs, etc.

2.2 L'approche algébrique

L'approche algébrique est basée sur la théorie des catégories [Gradit 99]. L'appellation algébrique a comme origine le collage des graphes qui se fait par un « push out » dans une catégorie appropriée. Cette approche est divisée, principalement, en deux sous approches qui sont SPO (single-pushout approach) et DPO (double-pushout approach). Nous détaillons dans cette section ces deux approches toute en présentant les avantages et les limites de chacune d'elles. Nous présentons aussi le NAC qui consiste à ajouter une condition avant l'application d'une règle.

2.2.1 SPO (Single-PushOut)

L'approche SPO est l'une des approches les plus classiques. La réécrire d'un graphe G en un graphe G' revient à remplacer un sous graphe L de G par un graphe R. G' est le graphe résultant de ces deux opérations. G est appelé graphe hôte ("host graph"), L est appelé graphe mère ("mother graph") et R est appelé graphe fille ("daughter graph"). Dans cet esprit, une règle de réécriture est sous la forme p: L →R. Ce type de règle est applicable à un graphe G s'il existe une occurrence de L dans G [Guennoun 07]. Son application a pour conséquence la suppression de l'occurrence de L du graphe G et son remplacement par une copie (isomorphe) de R. La figure 2.1 représente un exemple d'application de transformation selon SPO.

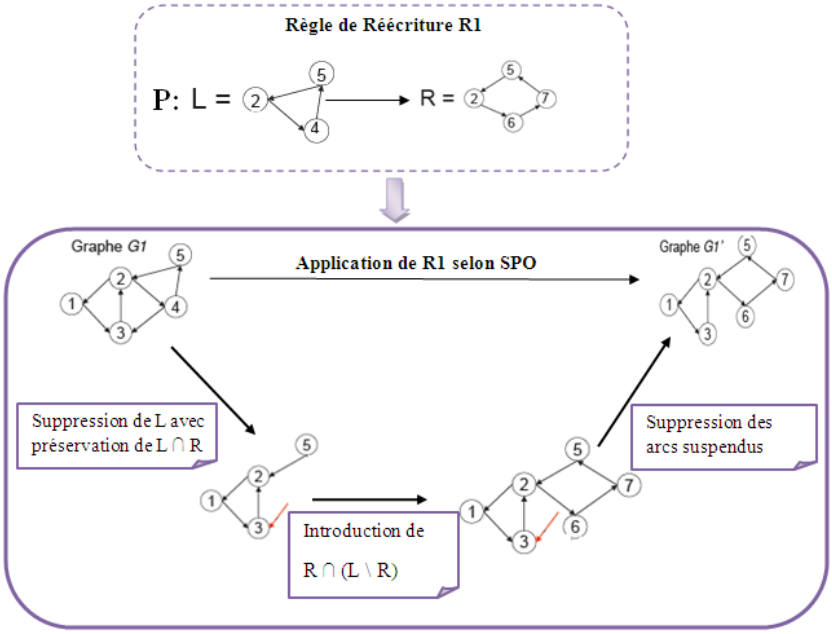


Figure 2.1 Exemple d'application d'une règle de transformation selon SPO

L'application de la règle implique la suppression du graphe correspondant à Supp = (L ∩ (L\R)) et le rajout du graphe correspondant à Aj = (R ∩ (L \ R)). Selon l'approche SPO, les arcs suspendus (arcs sans nœud de départ ou sans nœud d'arrivée ou sans les deux) sont supprimés. L'application de la règle de réécriture est décomposée en deux étapes concernant : (1) la mise à jour du graphe en supprimant l'occurrence du graphe *Supp* (2) l'introduction d'une copie du graphe *Aj*, et (3) la suppression de l'arc suspendu (qui reliait le nœud 4 au nœud 3) dont l'apparition est due à la suppression du nœud 4.
La figure 2.1 est une représentation simplifiée de l'approche SPO. La notation mathématique rigoureuse [Hartmut 99] est de la forme suivante :

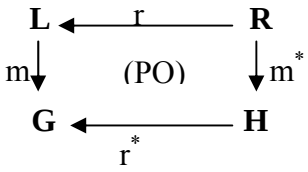


Figure 2.2 Notation mathématique d'une règle de transformation SPO

Dans cette notation, r est un graphe morphisme qui est injectif (entre L et R). La règle de transformation est appliquée, s'il existe un morphisme m entre L et G. H est obtenu en supprimant L de G et ajoutant R. H est unique car r est injective [Wermelinger 99] .
Le problème posé dans cette approche est l'apparition des arcs suspendus ("dangling edges"). Comme solution, SPO procède par la suppression de ce type d'arc. De ce faite SPO est considérée comme une approche non préservative, avec risque d'effet de bord. En effet, l'application d'une transformation SPO peut entrainer l'apparition d'un nœud sans aucun arc vers le reste des nœuds de l'application. Par contre, SPO est plus intuitive et plus facile à implémenter.

2.2.2 DPO (Double -PushOut)

La règle de transformation DPO, est de la forme L←K→R. K permet de spécifier explicitement la partie à préserver après l'application de la règle (au lieu de la déduire par l'opération L \R). Une règle de type DPO est applicable à un graphe G s'il y'a une occurrence de L dans G. La différence principale avec l'approche SPO est que l'application de cette règle est subordonnée à une condition supplémentaire appelée la condition de suspension ("Dangling Condition"). Cette condition stipule que la règle ne peut être appliquée que si son application ne va pas entraîner l'apparition d'arcs suspendus. Si les deux conditions d'existence de l'occurrence et d'absence d'arcs suspendus sont réunies, l'application de la règle implique la suppression du graphe correspondant à l'occurrence de *Supp* = (L\K) et le rajout d'une copie du graphe *Aj* = (R \ K).

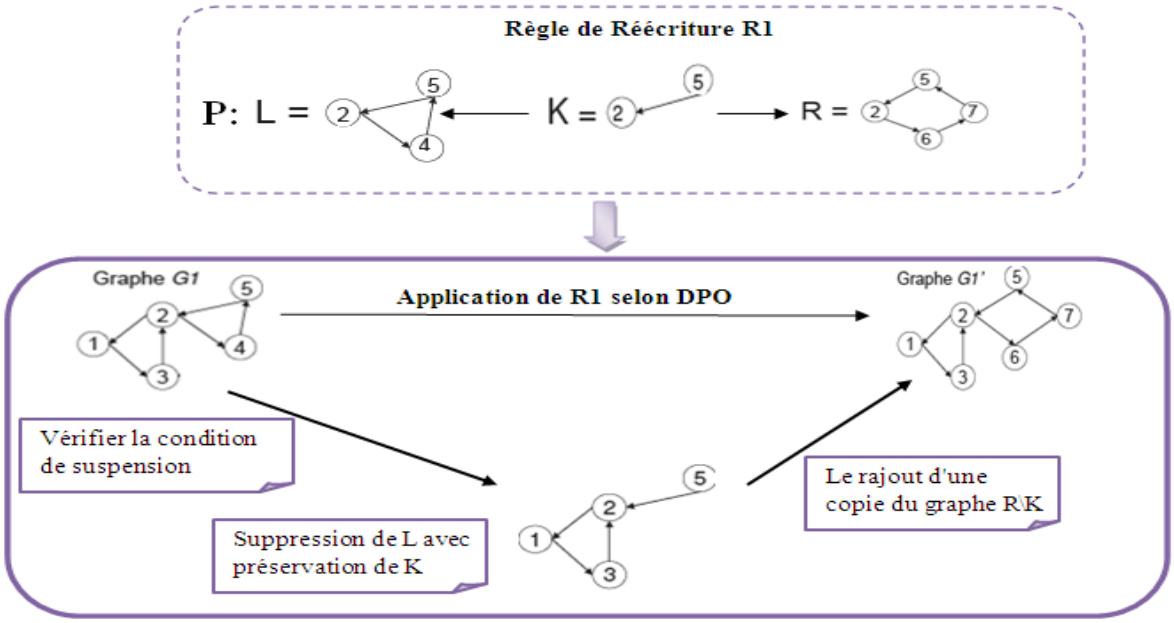


Figure 2.3 Exemple d'application d'une règle de transformation selon DPO

La figure 2.3 présente un exemple d'application d'une règle DPO. Nous avons choisit un autre graphe hôte car cette règle n'est pas applicable si nous l'appliquons au même graphe de la figure 2.1 (violerait la condition de suspension). La différence réside dans le fait que le graphe G1 dans la figure 2.3 ne contient plus l'arc reliant les nœuds 4 et 3 dans le graphe G1 de la figure 2.1.

A cette représentation graphique correspond une notation formelle donnée par la figure suivante :

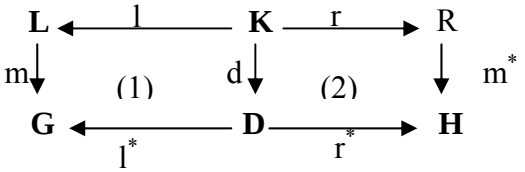


Figure 2.4 Notation mathématique d'une règle de transformation SPO

DPO est basé sur deux morphismes de graphe r et l (qui sont injectives) [Wermelinger 99]. Chaque carré correspond à un *pushout*. D est obtenu en supprimant de G tous les nœuds et des arcs qui apparaissent en L, mais pas dans K. Alors H est obtenu en ajoutant à D, tous les nœuds et des arcs apparaissent dans R mais pas dans K. Le fait que l et r sont injectives implique que H est unique.

La transformation de graphe selon DPO ajoute une condition à vérifier avant d'appliquer une règle ; c'est la condition de suspension. Le problème d'arc suspendu n'est plus posé. Cette approche est donc conservative sans risque d'effet de bord.

Les deux approches se basent sur la vérification de l'existence du graphe mère dans le graphe hôte : c'est la recherche d'un homomorphisme entre ces deux graphes. Mais pour DPO nous avons un double morphisme de graphe. Pour cette raison la complexité de DPO est plus

grande que celle de SPO. Une autre différence importante entre les deux approches est que la dérivation directe est inversible en DPO. Cette propriété est non vérifiée pour SPO.

2.2.3 Les conditions d'application négatives (NAC)

Pour déterminer l'applicabilité d'une règle de transformation, les approches présentées précédemment se basent sur l'existence d'une occurrence du sous graphe *L*. Dans certains cas, il est nécessaire de pouvoir exprimer des conditions supplémentaires permettant de spécifier des conditions relatives à l'absence d'une occurrence dans le graphe hôte. Ce type de conditions est appelé conditions d'application négatives (Negative Application Conditions ou NACs). L'intégration de conditions de type NAC ajoute donc une nouvelle zone dans la structure des règles de réécriture. Une règle SPO aura la structure (L; R; NAC) et sera applicable à un graphe G s'il y'a une occurrence de *L* dans *G* et s'il n'y a pas d'occurrence de NAC dans *G*. Dans le même esprit, une règle selon l'approche DPO aura la structure (L ;K ;R ;NAC). Dans tous les cas, le motif NAC n'intervient qu'au niveau de l'applicabilité de la règle et non dans l'étape de transformation qui reste la même que dans le cas sans.

Dans cette section nous avons introduit le cadre formel de la transformation de graphe. Nous analysons dans la section suivante les travaux sur l'adaptation logicielle d'assemblages de composants en utilisant la théorie des transformations de graphe et nous serons particulièrement attentifs aux mécanismes de composition des adaptations.

2.3 Modèles d'adaptation dynamique avec TG

2.3.1 K-Component

K-Component [Dowling et al 01] est un modèle qui permet de construire des architectures logicielles dynamiques pour des systèmes fonctionnant dans un environnement décentralisé. Il utilise les architectures dynamiques et la réflexion structurelle dans la construction de systèmes adaptatifs. L'intégrité et la sûreté de l'adaptation dynamique du logiciel est garantie par l'utilisation des transformations de graphe sur une architecture de logiciel. Dans ce modèle, il y a une séparation du code d'adaptation du code fonctionnel en l'encapsulant dans des programmes réflexifs appelés le *contact d'adaptation* [Dowling et al 02].

La reconfiguration dynamique est modélisée comme des transformations de graphe conditionnelles spécifiées dans des contrats d'adaptation. Le nœud d'un graphe selon K-component représente une interface étiquetée par l'implémentation du composant et les arcs modélisent des connecteurs libellés avec ses propriétés. L'étiquette de l'arc représente les propriétés reconfigurables du connecteur telles que la capacité de changer le protocole de transmission ou l'ensemble d'intercepteurs installés. Une transformation de graphe est une manipulation basée sur les règles de configuration de graphe. Un exemple de transformation est donné dans la figure 2.5. Ces règles définissent comment et quand un graphe doit être transformé. Cette transformation consiste à remplacer des composants logiciels appartenant à une configuration particulière d'un système et à modifier les stratégies des connecteurs en passant d local vers Remote.

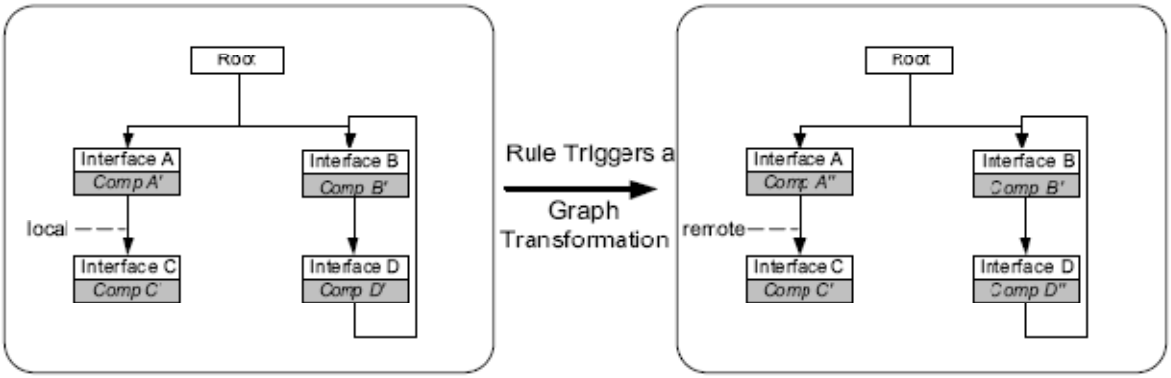


Figure 2.5 Exemple d'une règle de transformation en K-Component

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 [Dowling et al 04].

- **Le protocole de reconfiguration**

Pour s'assurer que seuls les sommets qui sont affectés par la transformation doivent être dans un *état sûr,* un *protocole de reconfiguration* [Dowling et al 01] est employé. Ce dernier réduit la durée de la phase de reconfiguration et permettent les invocations concurrentes pendant la phase de reconfiguration sur les composants qui ne sont pas affectés par la transformation. Le protocole de reconfiguration déroule les étapes suivantes :

1. L'opération de reconfiguration est appelée pour transformer le graphe
2. Identifier la nouvelle configuration cible et envoie un message de « blocage » seulement à ces composants et connecteurs dans la configuration originale qui seront mis à jour dans la nouvelle configuration.
3. Exécuter la transformation du graphe, c.-à-d. détacher, créer, enlever et lier les nouveaux composants et/ou changer la stratégie de connecteur.
4. Refaire le traitement pour les autres composants et connecteurs bloqués.

- **Contrats d'adaptation**

Un contrat d'adaptation contient une série de règles de transformation de graphe pour la configuration d'architecture de logiciel. Les contrats d'adaptation exigent un mécanisme pour accéder aux propriétés architecturales [Dowling et al 04]. Ils fournissent des *événements d'adaptation* comme mécanisme pour permettre à des contrats d'adaptation de récupérer les contraintes des configurations architecturale. Les événements d'adaptation ont également l'avantage de découpler le contrat d'adaptation au niveau des composants et des connecteurs. En effet, ils fournissent une séparation entre le code d'adaptation et le code fonctionnel. Ceci permet que les contrats d'adaptation soient dynamiquement chargés et déchargés, puisque le code fonctionnel n'a pas de dépendances directes sur elles.

- **Le modèle de K-Component**

Le méta-modèle d'architecture de K-Component fournit un gestionnaire de configuration qui stocke le méta-modèle d'architecture et applique les opérations de reconfiguration (réécriture de graphe) sur de l'architecture. Le gestionnaire de configuration est également un récipient d'exécution pour le déploiement, l'établissement du programme et l'exécution des contrats d'adaptation et peut sur option fournir une interface procédurale pour le chargement/le déchargement des contrats d'adaptation au temps d'exécution. Il est mis en application comme objet actif.

K-Component apporte une contribution originale à l'adaptation en utilisant comme outil la transformation de graphe. Le principe du contrat d'adaptation est équivalent à celui de l'AA avec quelques différences. En effet, un contrat d'adaptation contient la définition des interfaces, composants, événement, intercepteurs, opérateur de reconfiguration et règles de TG.

Dans le modèle k-component l'adaptation est limitée au changement des propriétés sur les liens entre les nœuds. Il est impossible d'ajouter ou de supprimer les nœuds ou les arcs. Ceci constitue un défit pour une application en IAm car nous ne pouvons pas adapter une application au changement de son environnement suite à l'apparition ou la disparition des dispositifs.

L'ordre d'application des règles de transformation influence le résultat d'adaptation. De plus, nous pouvons nous trouver dans une situation où il y a un conflit entre les règles de transformation. Ces problèmes ne sont pas traités dans le modèle k-component.

2.3.2 Raffinement de modèle de graphe

Dans ces travaux [Baresi et al 04] [Baresi et al 06], Luciano avait comme objectif le raffinement des modèles des architectures dynamiques. Même si l'adaptation dynamique n'était pas parmi ces objectifs, les modèles proposés englobent l'évolution des architectures. Ces systèmes ne se composent pas simplement d'une structure fixe et statique, mais peuvent réagir à certaines conditions ou événements par la reconfiguration d'exécution de ses composants et leurs connexions.

Baresi définit un système de transformation de graphe comme (1) une composition d'un *type de graphe* pour définir les éléments architecturaux, (2) un ensemble de *contraintes* pour prouvé les modèles valides, et (3) un ensemble de *règles de transformation de graphe.* Une architecture de système qui se conforme à un modèle donné est représentée comme une instance du type graphe. La reconfiguration dynamique est spécifiée par des règles de transformation de graphe modelée au niveau des instances d'architecture [Baresi et al 04].

- **Type de graphe**

Baresi définit deux modèles d'architecture. Le premier à base de composant et le deuxième à base de service. Au début, il définit un modèle architectural générique qui peut être employé pour modéliser les architectures de l'application. Ensuite, les transformations seront explicitées en des instances conformes aux types de graphe. Dans le cas d'un système à base

de composant (figure 2.6), Il distingue les composants et les connecteurs en tant qu'éléments architecturaux de première classe. Les composants ont des ports qui sont décrits par les interfaces fournies et exigées. La communication est basée sur l'échange de messages, et la création et la modification des liens sont les seules opérations tolérées pour la reconfiguration.

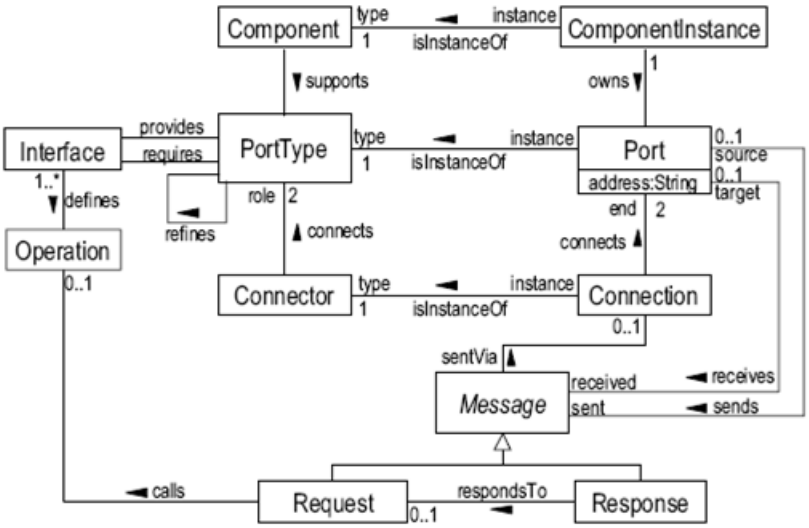


Figure 2.6 Type de Graphe d'une application à base de composent

- **Contraintes**

Avec le type le graphe vient un ensemble *C de contraintes* qui limitent l'ensemble de l'instance de graphes valides. Un exemple simple de contraintes est les cardinalités qui limitent la multiplicité de liens entre les éléments.

- **Règle de transformation de graphe**

L'adaptation dynamique de la structure d'une application est donnée par l'application des *règles de transformation de graphe.* Une règle doit être conforme au type de graphe. La figure 2.7 représente un exemple de règle de transformation de graphe. Cette règle ne peut être appliquée que si les instances des composants ne sont pas encore reliés *(état négatif d'application)* et s'ils possèdent les ports dont les types peuvent être reliés. Le résultat de l'application de règle est la création d'un nouveau lien entre les deux ports.

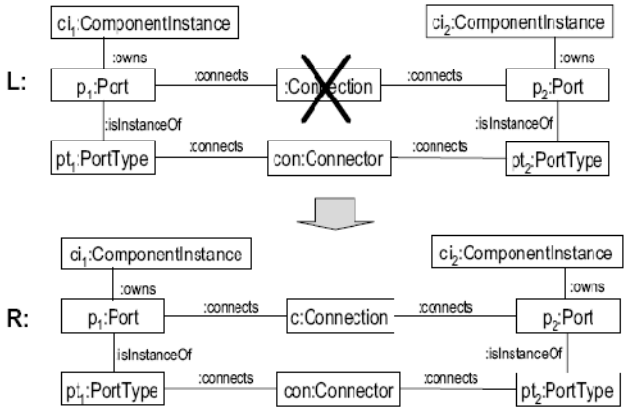


Figure 2.7 Règle de transformation de Graphe d'une application à base de composent

En respectant le même principe, Baresi définit un autre type de graphe en utilisant l'approche service. Les règles de transformation permettent donc de modifier une architecture définie autour des services [Baresi et al 06].

Pour simuler ces transformations, Baresi utilise des outils classiques de transformation de graphe. Il a opté pour l'outil PROGRES puisqu'il supporte la recherche en profondeur et le backtraking. PROGRES lui permet de définir un type de graphe, l'ensemble de règles de transformation, et le graphe hôte sur lequel les transformations seront appliquées.

La contribution essentielle des travaux de [Baresi et al 06] est le raffinement des modèles architecturaux. L'apport principal de ces travaux est la définition d'un type de graphe qui permet de vérifier la validité des règles de transformation. Ce type de graphe est équivalent au modèle LCA ((Lightweight Component Architecture) dans les travaux de [Cheung-Foo-Wo 09]. Les règles de transformation définies par Luciano sont de type DPO avec la condition négative NAC. Ces règles sont limitées à la création des liens entre les ports (connecteurs). De ce faite l'adaptation reste encore limitée et il n'est plus possible de prendre en compte des changements de l'environnement. Un autre problème qui n'a pas été traité dans cette approche est le cas où nous avons des règles de transformation en conflit les unes avec les autres.

2.3.3 Transformation de graphe pour la reconfiguration de l'architecture logicielle

La capacité de modifier des architectures de logiciel afin de les adapter à de nouvelles conditions ou à un environnement en cours d'évolution a été l'objectif des travaux de [Wermeling 99]. Ils proposent une approche algébrique dans laquelle: (1) les composants sont écrits dans un langage de programmation et de conception de haut niveau avec la notion d'état, (2) l'approche traite des problèmes typiques tels que garantir que de nouveaux composants sont ajoutés dans l'état correct et que l'architecture en résultant se conforme à certaines contraintes structurales et (3) le code d'adaptation et le code fonctionnel sont explicitement séparés.

Ils définissent les architectures comme des graphes dont les nœuds sont des programmes écrits dans COMMUNITY (langage de conception de programme parallèle avec la notion état) et les arcs dénotent des relations de superposition. La reconfiguration est spécifiée par les règles conditionnelles de réécriture de graphe qui dépendent de l'état des composants impliqués.

L'utilisation de la transformation algébrique de graphe offre la possibilité de définir des règles de transformation comme un ensemble de productions conditionnelles de graphe en employant seulement des graphes et des morphismes de graphe. Le programmeur a la charge de spécifier l'état du composant qui sera ajouté par une règle de récriture. La figure 2.8 représente un exemple de règle de transformation.

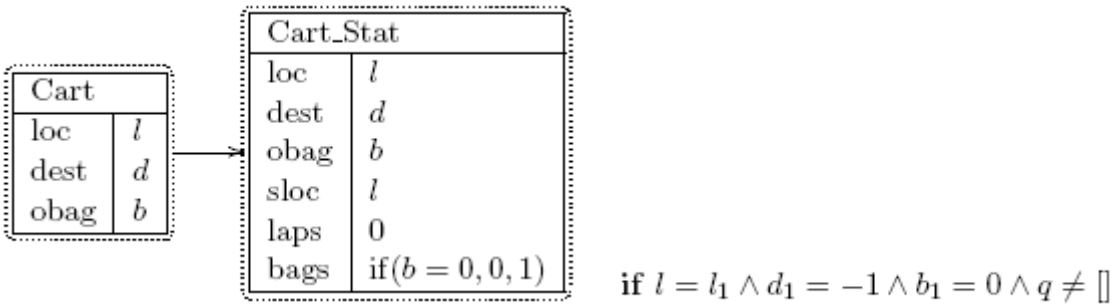


Figure 2.8 Exemple de règle de Transformation

Cette règle définit comme adaptation le changement d'un composants Cart par le composant Cart_Stat avec une vision des variables local du nouveau composant.

L'adaptation dynamique est confiée à un gestionnaire de reconfiguration qui applique la dérivation d'une instance de graphe donné G vers une instance H. En effet, il n'y a aucune restriction aux graphes obtenus, mais dans la reconfiguration ils vérifient que le résultat est en effet une instance d'architecture, sinon la règle ne s'applique pas. Le problème de récursivité infinie d'application des règles de transformation (le côté à gauche est un sous-graphe du côté droit) est résolu par la définition des dérivations permises.

Par rapport aux autres travaux qui utilisent les transformations de graphe comme outils d'adaptation dynamique, ce modèle résout le problème de récursivité d'application des règles par la définition des dérivations permises. Les règles de transformation considèrent que les composants sont des boites blanches. Ils ont une vision sur les variables locales. Ceci est contradictoire par rapport aux applications en IAm où nous considérons les composants comme des boites noires et ne sont accessibles que par leurs interfaces. Deplus, le problème de conflit entre les règles n'est une fois de plus pas étudié.

2.3.4 Architectures Dynamiques dans le contexte des Applications à Base de Composants et Orientées Services

L'objectif de [Guennoun 07] était d'adapter l'application pour répondre à un changement de son contexte (e.g. alarmes de sécurité, changements au niveau des ressources disponibles, pannes réseau), à de nouveaux objectifs applicatifs (e.g. passage d'une phase d'exécution vers une autre), ou à une modification du nombre ou du rôle applicatif de ses composants (e.g. intégration ou exclusion de participants dans une application coopérative). Il propose une approche orientée-modèle pour une adaptation structurelle automatique des applications pendant l'exécution. Cette approche est basée sur des techniques orientées-règles pour la reconfiguration architecturale dynamique. La transformation peut impliquer l'introduction ou la suppression de composants et/ou la modification des interdépendances entre ces composants.

Pour passer l'architecture d'une configuration donnée à une autre, il propose un système permettant de transformer l'instance courante de l'architecture en produisant une nouvelle instance afin de répondre à la production d'un événement de reconfiguration. Les règles de

reconfiguration de l'architecture sont une combinaison des règles de transformation de graphe. Elles traitent des nœuds multi libellés et des arcs multi-étiquetés. Pour les mêmes besoins de généricité, les règles de reconfiguration peuvent être paramétrées et considérer des labels variables. Une règle de transformation est donc spécifiée par un quadruplet (L;K;R;N;C). Les graphes L, K et R sont les même que l'approche DPO alors que N désigne la condition d'application négative et C contient les instructions de connexion. Les instructions de connexion permettent de rediriger les interdépendances impliquant les composants à supprimer vers les composants à introduire. Ce dispositif se révèle très utile dans le cas d'un composant pouvant avoir un nombre indéterminé de composants voisins. Les règles de transformations sont une combinaison des approches SPO et DPO. DPO est exploitée car elle permet de désigner de manière explicite la partie du graphe mère qui sera maintenu après l'application de la production. SPO est utilisé pour le traitement des arcs suspendus.
Les règles de transformation sont donc décrites par des graphes étendus où les composants (ou les services) sont représentés par des nœuds et les interdépendances (e.g. les connexions, les relations de contrôle ...etc.) sont décrites par des arcs (figure 2.9).

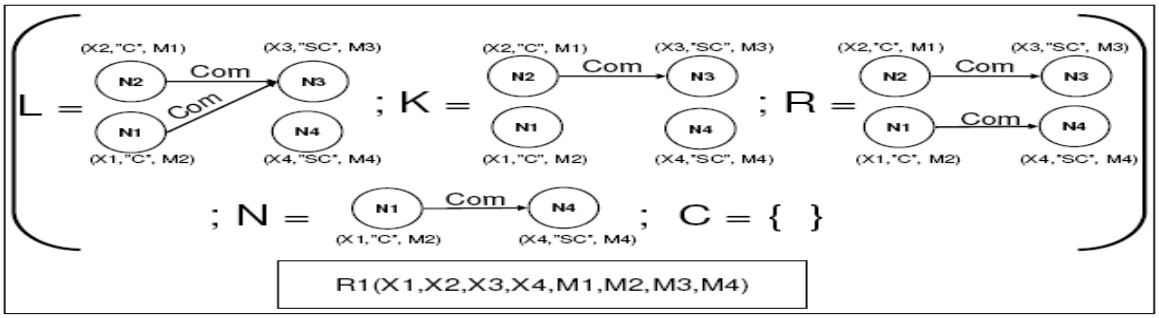


Figure 2.9 Exemple de règle de transformation

Les nœuds d'un graphe représentent les composants logiciels de l'application. Ces composants peuvent correspondre à des unités de calcul indépendantes telles que des composants EJB (Entreprise Java Beans), ou des Services Web. Ils peuvent aussi désigner d'autres entités logicielles telles que des registres UDDI (Universal Description Discovery and Integration) ou des bases de données, ou même un service composite. Les paramètres nécessaires pour la description d'un composant peuvent varier d'un type de composants à un autre et d'un contexte de description à un autre. Les nœuds des graphes sont, donc, multi-libellés où chaque label correspond à un paramètre du composant. Les arcs du graphe sont étiquetés par le type et les propriétés des liens qu'ils représentent.
Pour chaque composant, [Guennoun 07] considère un premier paramètre correspondant à l'identifiant du composant. Le deuxième paramètre au type du composant. Il s'intéresse aussi à la localisation du composant en introduisant un paramètre relatif à la machine sur laquelle il est déployé. Pour les composants de type service simple il expose, en plus, un paramètre de QdS indiquant le temps de réponse par le service. Pour les liens, il définit deux types : 1) le premier concerne les liens de communication qui lient un consommateur à un service simple ou composite (Com), alors que 2) le second concerne les liens de composition liant un service composite aux services simples ou composites qu'il contrôle (Comp).
L'application de la règle de la figure 2.9 donne comme résultat la figure 2.10. Il s'agit de la redirection d'un client vers un autre service.

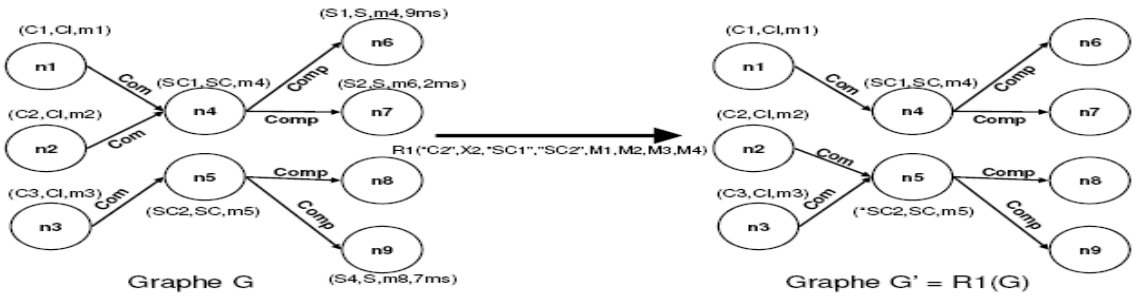


Figure 2.10 Application d'une règle de transformation d'un graphe

Dans son méta modèle [Guennoun 07] utilise la grammaire de graphe pour garantir que le graphe est valide. Ces grammaires de graphes sont étendues pour augmenter le pouvoir d'expression du méta-modèle. Les extensions concernent, par exemple, l'introduction de variables typées dans les productions de grammaires et la notion de restrictions à l'applicabilité de ces productions. Ce méta-modèle traite aussi la description des contraintes et des propriétés architecturales d'une application (e.g. structure architecturale de l'application pouvant être par exemple arborescente ou en anneau, nombre maximal d'imbrications pour la composition des services, cardinalité des liens entre différents types de composants...etc). Ce type de contraintes est spécifié par l'absence ou la présence de motifs dans les graphes décrivant l'architecture. Leur vérification automatique est réalisée à l'aide d'algorithmes de recherche d'homomorphismes.

L'adaptation dynamique dans cette approche est confiée au protocole de gestion dynamique de l'architecture qui est défini par l'ensemble des événements impliquant une reconfiguration et la combinaison des règles de transformation à appliquer pour exécuter les actions correspondantes. Les événements d'adaptation peuvent être générés par les différents composants du système ou par son environnement. Ils sont décrits par un type et peuvent contenir des paramètres qui amènent une spécialisation des actions à réaliser.

Dans ses travaux, [Guennoun 07] a traité le problème de composition des règles de transformation. Il a définit trois opérateurs de compositions qui sont :

- L'opérateur d'application séquentielle (noté " ;") où une combinaison c=r1;r2 implique : 1) si r1 est applicable alors appliquer r1 puis r2, 2) sinon la combinaison c n'est pas applicable.
- L'opérateur de disjonction (noté "||") où une combinaison c=r1||r2 implique : 1) si r1 (respectivement r2) est applicable alors appliquer r1 (respectivement r2) sinon, 2) appliquer r2 (respectivement r1) si cette règle est applicable. Si ni r1 ni r2 ne sont applicables alors c n'est pas applicable.
- L'opérateur d'application multiple (noté "*") où l'application d'une combinaison c=r* implique d'appliquer la règle r tant que cette règle est applicable20. La combinaison c est applicable si et seulement si r est applicable une fois.

A travers son méta-modèle, [Guennoun 07] a proposé une transformation de graphe qui englobe composants, services et services composites. Par rapport aux autres travaux sur l'adaptation utilisant les transformations de graphe, les travaux de [Guennoun 07] sont les plus évolués. Il a proposé une solution pour gérer le problème de conflit entre les règles. La solution proposée est de définir d'une manière explicite des opérateurs de composition de règle. Mais ce mécanisme reste limité car en IAm, nous avons besoin d'une composition implicite puisque nous ne savons pas d'avance quels sont les règles en conflit. D'autre part, la solution proposée pour la gestion des conflits repose sur un mécanisme simple de priorité entre les règles.

2.4 Conclusion

Les travaux qui portent sur l'adaptation dynamique en utilisant les transformations de graphe sont moins évolués que ceux qui se basent sur les Aspects. Le sujet de la composition des règles d'adaptation se limite à la définition de priorités entre les règles dans les travaux de [Guennoun 07] ce qui reste limité par rapport ce qui a été réalisé pour les AA. En partant de l'ensemble de ces limites, tant dans le domaine des adaptations par aspects ou par transformations de graphe, nous définissons dans le chapitre suivant notre contribution.

Chapitre 3

Contribution et Illustrations

L'adaptation par transformation de graphe suscite un certain intérêt, mais soulève des problèmes non encore résolus. Le conflit qui peut être générer entre les règles de transformation reste encore non traité. Pour ce faire, nous nous appuyons sur le mécanisme qui a été développé pour les AA pour mettre en place un mécanisme plus général qui sera indépendant d'un langage spécifique. Ceci nous permettra d'atteindre notre objectif pour l'ajout de la prise en compte de composants avec de nouvelles sémantiques, non prédéfini.

Dans ce chapitre nous détaillons notre contribution à savoir le mécanisme général pour le déroulement de la composition. Nous explicitons les adaptations sous forme de règles de transformation qui seront appliquées sur l'assemblage de composants qui est vu comme un graphe. Nous commencerons donc par la définition du modèle de graphe qui illustre un assemblage de composants.

Suite à la définition de notre contribution, nous détaillons l'implémentation réalisée qui inclus le choix de l'outil pour les transformations de graphe et l'expérimentation de notre approche sur des exemples réels. Pour terminer ce chapitre, nous analyserons le travail réalisé et nous dégagerons alors ces apports et ces limites.

3.1 Contribution : composition par TG

Nous rappelons que notre objectif est d'ajouter des nouvelles sémantiques dans la modification dynamique d'assemblage de composants logiciels. L'étude des travaux de recherche qui traitent le sujet de l'adaptation dynamique nous amène à l'adoption de concept des AA qui permet la modification structurelle des applications pendant l'exécution. Nous ne nous intéressons pas à l'adaptation dynamique uniquement, mais nous souhaitons aussi ajouter de nouvelles sémantiques, c'est-à-dire des règles sur des composants de sémantique connue mais qui ne sont pas prédéfinies dans un langage comme celui utilisé par les AA. Nous avons prouvé que les AA ne permettent pas l'ajout de nouvelles sémantiques puisque le mécanisme de composition est lié à un langage. Nous avons donc besoin d'un autre mécanisme pour aboutir à notre objectif.

Pendant la composition des AA, le moteur de fusion produit comme résultat un assemblage de composants qui s'exprime sous la forme d'un graphe. Ceci nous a donc conduit à la théorie des graphes qui englobe un mécanisme de transformation rigoureux des graphes qui peut être vue comme un outil pour faire l'adaptation dynamique en partant du faite qu'une adaptation n'est qu'une modification structurelle d'un sous-assemblage qui représente un sous-graphe du graphe original (c'est à dire l'application de départ).

Notre approche est donc basée sur les apports qui ont été obtenus par les AA en étudiant leur transposition dans le domaine de la transformation de graphe. Dans notre approche nous considérons donc qu'un assemblage de composants est un graphe. Les AA qui modélisent la modification structurelle à apporter seront représentés comme des règles de TG. Dans le cas où nous avons plusieurs règles à appliquer nous utilisons le TG pour les fusionner et résoudre ensuite le conflit qui peut être généré. Nous avons besoin pour cela de définir en premier lieu le modèle de graphe qui permettra de représenter notre application. Ensuite nous détaillons nos règles de TG ainsi que le mécanisme général pour l'adaptation. Enfin nous détaillons le mécanisme de composition des règles avec l'ajout des nouvelles sémantiques.

3.1.2 Un modèle de graphe représentant un assemblage de composants

L'adaptation par transformation des graphes s'appuie sur l'utilisation d'un graph Host qui représente l'application qui doit être modifiée suite aux changements dans l'environnement. Nous devons donc exprimer les contraintes des informations à représenter.

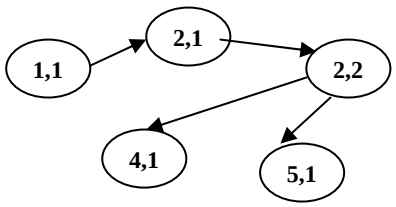
Notre graphe devra être un graphe orienté car les liens entre les composants modélisent l'émission d'un événement partant d'un composant qui déclenche l'appel d'une méthode d'un composant. D'une manière générale, un graphe G est définit par :

G=(S, A)

S: ensemble de sommet

A: ensemble d'arc orienté: {(s1, s2), (s2, s3), …}

Nous commençons par la définition de l'ensemble S. En effet nous avons le choix entre deux possibilités. La première est de considérer que S représente l'ensemble des ports des composants c'est-à-dire les événements qu'ils peuvent émettre ou les méthodes en entrée. Ceci nous amène à un graphe bipolaire (d'un ensemble d'entrées vers un ensemble de sorties). Nous obtenons donc le graphe suivant :



Dans ce cas nous, dupliquons le nombre de nœud du graphe. Nous obtenons autant de nœuds que nous avons des ports au niveau des composants. Cette possibilité est à éliminer car nous avons potentiellement un nombre important de port pour chacun des composants ce qui alourdirait la représentation de nos graphes.

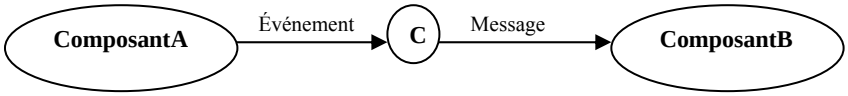
La deuxième possibilité est de considérer qu'un nœud est un composant logiciel et qu'un port est un composant avec un arc. Nous avons donc la représentation suivante :

Composant : 

port : 

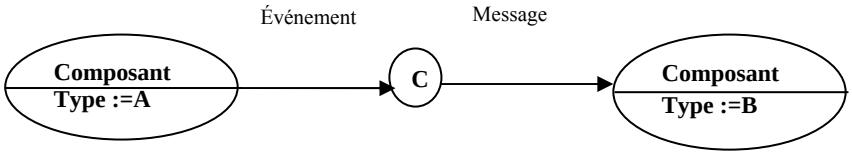
Avec cette représentation nous n'obtenons pas un graphe ; il est nécessaire de modéliser le liaison entre les ports. La solution retenue a été d'ajouter un nœud intermédiaire qui joue le

rôle de connecteur. Son rôle est de lier un événement d'un composant à un message d'un autre. Ceci se traduit par la figure suivante :



Ce modèle a pour avantage de permettre de spécifier dans des travaux futurs l'expression de différents types de liaisons en spécifiant un type pour le composant connecteur C.

Sur les nœuds nous spécifions le nom du composant et sur les arcs les noms des événements ou bien les méthodes. Toutefois, avec cette représentation, les règles de transformation seront liées aux noms de composants c'est-à-dire que nous devons dupliquer les règles pour chaque instance de composant. Pour cette raison nous utiliserons des graphes attribués avec ubn attribut spécifiant le type de composant sur lequel la règle s'appliquera et non sur l'instance sur laquelle celle-ci s'appliquera. Nous définissons alors la représentation finale de notre graphe.



Nous distinguons deux types de nœud : (1) les nœuds ordinaire qui sont les composants logiciels et (2) les nœuds qui représente les opérateurs ou bien des nouvelles sémantiques. Cette séparation offre l'avantage de pouvoir traiter les nouvelles sémantiques d'une manière particulière.

Nous définitions donc notre modèle de graphe qui sert pour la validation de résultat de transformation qui doit être conforme au modèle identifié. La figure 3.1 représente le Type Graphe que nous avons utilisé. Dans ce modèle nous définissons les nœuds et les liaisons entre eux. Nous avons deux types de nœuds les composants ordinaires et les composants de sémantique connue (tel que PAR, IF et SEQ). Dans notre modèle nous interdisons donc d'avoir des liens d'un composant vers un autre sans passé par un connecteur. Si une transformation ne respecte pas cette modélisation elle devra être refusée. Au niveau d'un nœud nous spécifions son type et pour les arcs le nom de port.

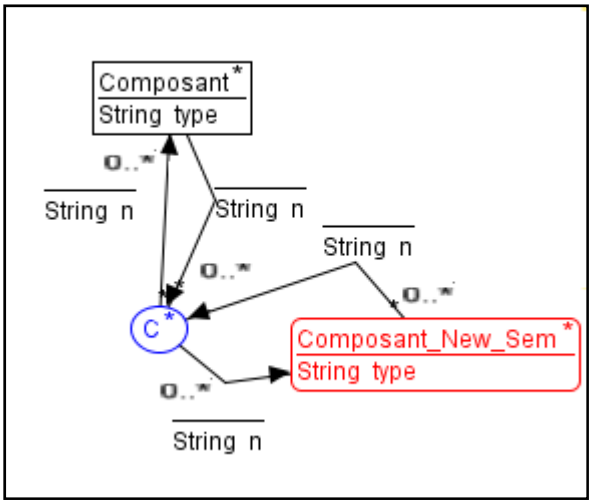


Figure 3.1 Modèle de graphe de Type

L'instanciation de ce modèle représente notre application. Ceci est illustré par la figure 3.2 suivante.

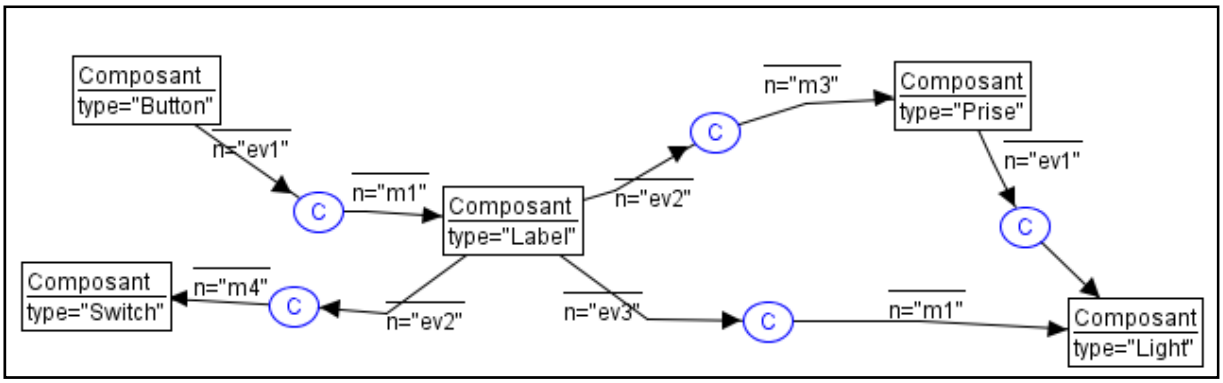


Figure 3.2 Assemblage de composant selon notre modèle de graphe

Nous avons donc défini l'assemblage de composants sous forme d'un graphe en explicitant le modèle de graphe que nous considèrerons par la suite. Nous allons maintenant présenter la spécification de nos règles de transformation qui vont agir sur le graphe de l'application afin de réaliser l'adaptation.

3.1.3 Les règles de TG

Comme nous avons pu le voir dans le chapitre 1, les AA sont définis par deux parties qui sont le point de coupe et le greffon. Par définition un point de coupe sélectionne les endroits qui seront modifiés au niveau d'assemblage de composant. Nous l'utilisons donc pour formuler la partie gauche (LHS) dans la règle de Transformation de graphe. Le greffon, quant à lui, spécifie la modification à mettre en place. Il est donc isomorphe à la spécification de la partie droite (RHS) d'une règle.

Les règles de transformation que nous avons utilisées sont des règles avec la partie NAC et ceci dans le but d'éliminer l'application récursive de nos règles.

Nous avons deux types de règles : les règles statiques, définies indépendamment des adaptations, et des règles dynamiques. Les règles statiques sont les règles pour la résolution de conflit entre les modifications à mettre en place ou bien pour le traitement des opérateurs de sémantique connue. Les figure 3.3 et 3.4 montre deux règles statiques.

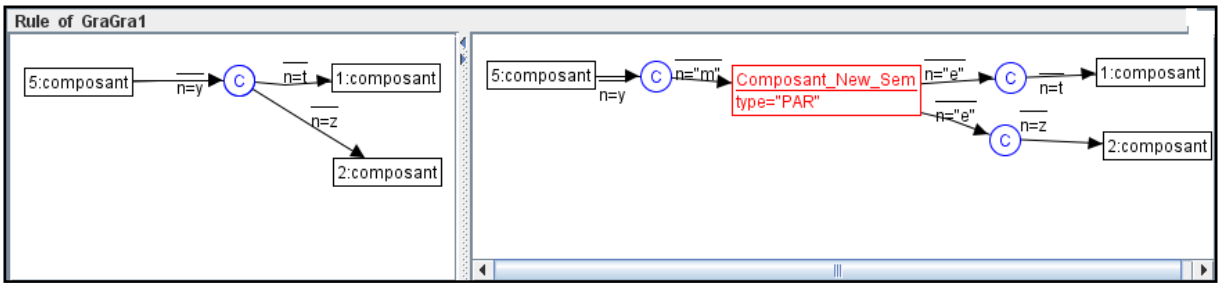


Figure 3.3 Règle de transformation de graphe de type statique : Résolution de conflit

Dans cette règle nous avons utilisé les variables pour qu'elle soit applicable quelque soit les noms d'événement et des méthodes. Le conflit est identifié dès que nous avons, dans notre graphe, un événement d'un composant qui déclenche deux opérations différentes. Dans ce cas nous avons deux appels à deux méthodes. La résolution de ce type de conflit, comme défini dans [Cheung-Foo-Woo 09], revient à la création d'un composant « PAR » et le transfert de deux messages vers les sorties du composant PAR.

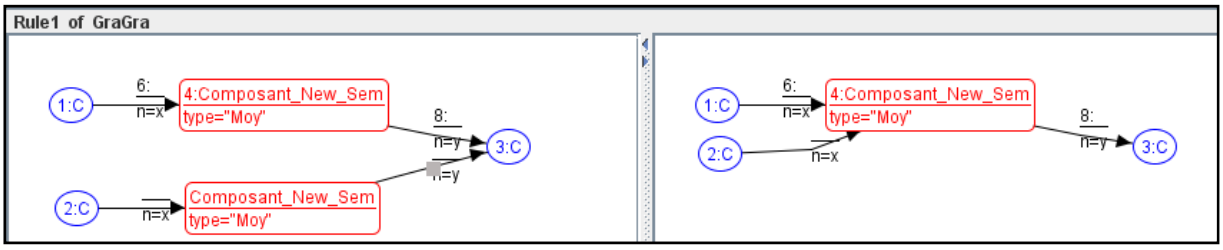


Figure 3.4 Règle de transformation pour composants de nouvelle sémantique

Cette règle traite le composant de sémantique connue « Moy ». Si nous avons deux composants qui ont deux événements liés à la même méthode de « Moy » et qu'ils ont le même événement en sortie alors il s'agit du même composant moyenne qui doit être partagé par ces deux composants. Etant donné que nous connaissons la sémantique de l'opérateur « Moy », l'adaptation spécifiée permettra de composer ce comportement avec pour conséquence une modification des liens comme illustré par la figure 3.4.

3.1.4 Mécanisme général

Le mécanisme général sera représenté par la figure 3.5. En se basant sur l'assemblage de composant initial, un ensemble de règles de transformation est sélectionné. Ces règles peuvent être en conflit c'est-à-dire qu'elles accèdent au même endroit pour le modifier. Pour résoudre ce problème nous avons défini un mécanisme de composition des règles de TG.

Les LHS des règles seront superposées pour construire le LHS Final et la même opération est faite pour la partie RHS des règles. Le mécanisme de composition des règles est combiné à un

mécanisme de résolution des conflits, c'est-à-dire si nous avons deux opérations différentes qui doivent s'appliquer en un même endroit.

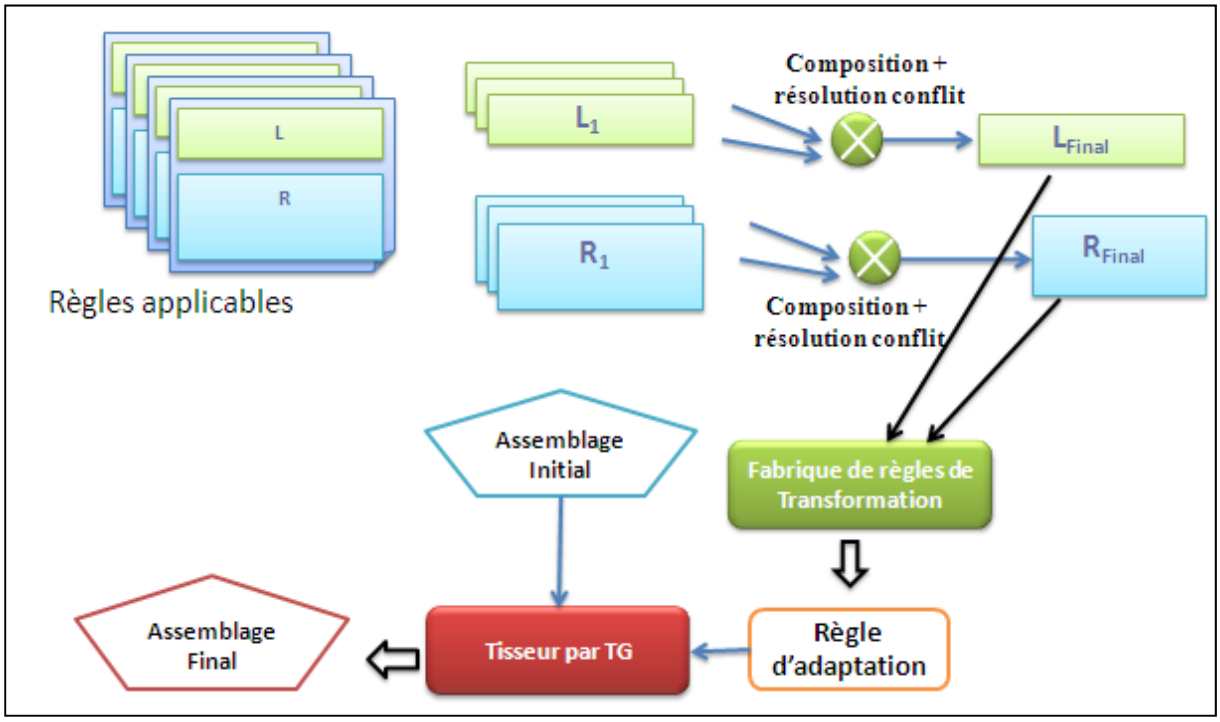


Figure 3.5 Composition des règles de transformation de graphe

Les parties L et R obtenues seront l'entrée de la fabrique de règles de transformation qui produira la règle finale spécifiant la modification à exécuter par le moteur de transformation.

En utilisant l'assemblage de base, la règle de transformation finale et les règles relatives aux composants de sémantique connue, le moteur de transformation applique ces règles sur l'assemblage initial et produira le graphe final qui représente l'application adaptée.

3.1.5 Composition des règles de TG

Le diagramme 3.6 montre la fonction de composition des règles de transformation. Elle prend en entrée les graphes qui représentent la partie gauche ou bien droite des règles à superposer.

L'algorithme pour la composition des règles de transformation peut être décrit de la manière suivante. En première étape, on initialise G au premier élément de la liste. G contiendra le résultat de chaque étape de transformation. Il sera modifié automatiquement. Nous configurons ensuite la règle de transformation c'est-à-dire les parties L, R et le NAC. Nous notons que nous utilisons le NAC pour éviter la récursivité de la transformation. Nous mettons L au graphe résultant de l'intersection entre G et Le graphe courant de la liste c'est l'élément Li. Ensuite, nous mettons R à L_i et R à L_i. A ce stade nous exécutons la transformation

À cette étape nous avons ajouté L_i aux éléments déjà superposés. La dernière opération consiste à résoudre le conflit qui peut être identifié c'est à dire si nous avons deux actions

différentes pour un même événement d'un composant. Ces étapes se répètent tant que nous avons des graphes à superposer.

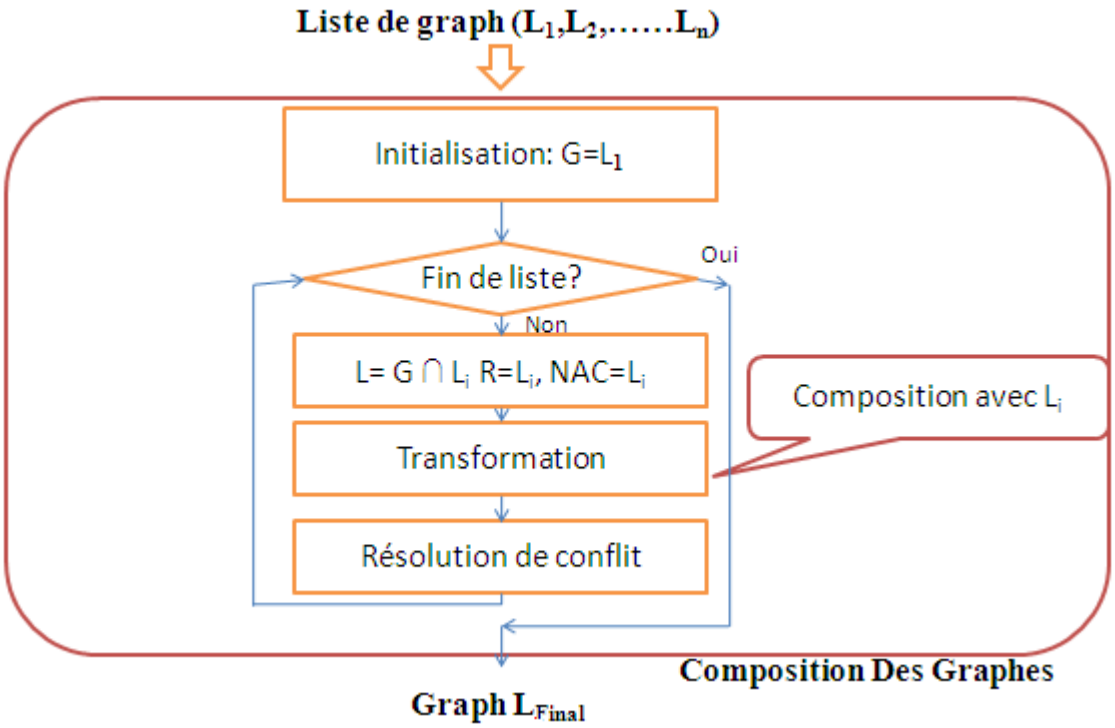


Figure 3.6 Organigramme de la fonction de composition des graphes

3.2 Implémentation de l'approche

3.2.1 Outils pour la transformation de graphe

A la fin des années 1960 et au début des années 1970, le concept de transformation de graphe est devenu intéressant. Les méthodes, les techniques et les résultats dans ce domaine de recherche ont été déjà appliqués dans plusieurs domaines de l'informatique. Ce concept peut servir aussi dans le cadre d'adaptation dynamique des applications en partant du fait qu'une application peut être modélisée sous forme d'un graphe qui sera transformé selon l'évolution de l'environnement.

Plusieurs implémentations ont été réalisées dans le domaine des transformations de graphe. Nous analysons dans cette section quelques outils et nous choisissons par la suite celui qui répond au mieux à nos besoins en termes d'adaptation dynamique d'application pour l'informatique ambiante.

- **GraphSynth**

GraphSynth est un interpréteur pour la création de grammaires de graphe en utilisant une interface graphique développé avec Microsoft Visual studio. Il est crée par le laboratoire de recherche "UT Automated Design Lab." Les règles et les graphes sont stockés en format XML et ils sont chargés dans le programme comme des objets qui sont de type arc ou nœud.

GraphSynth est basé sur l'approche SPO. Il classe les règles dans des groupes et donne le choix à l'utilisateur pour sélectionner le groupe de règles qui sera appliqué pendant la transformation. Il définit des graphes orientés, libellés et étiquetés.
Cet outil ne permet pas la définition des règles avec le NAC qui sera indispensable pour la résolution du problème d'application récursive des transformations. Un autre problème dans cet outil est qu'il se plante très souvent donc il n'est pas performant.
Il définit la transformation de graphe dans le cas le plus simple. Le « mapping » entre les graphes est limité aux nœuds qui sont définit c'est-à-dire il n'y a pas un mécanisme pour la définition des types de nœud et spécifier les règles de transformation en se basant sur les types et non pas les libellé des nœuds.

- **GROOVE**

GROOVE [Rensink 09] est un projet centré autour de l'utilisation de graphes simples pour la modélisation de la structure des applications pendant les phases de conception, de compilation et d'exécution. Il contient :

- Un éditeur graphique, capable de produire des graphes et des règles de transformation en format GXL (Graph eXchange Language);
- Un simulateur, capable de générer une transition à partir des règles : (1) grâce à la simulation manuel, y compris les retours en arrière (2) grâce à l'exploration automatique.

GROOVE [Rensink 09] combine G et R, L et NAC en un seul graphique et utilise un code de couleur pour distinguer les différentes parties d'une règle. En effet les nœuds qui vont être ajoutés sont précédés du mot clé *New* tant dis que les nœuds à supprimer sont spécifiés par le mot clé *Del*. La partie NAC dans une règle est déterminée par l'utilisation de mot clé *Not* ou bien *!*.

GROOVE est un outil puissant puisque qu'il supporte la notion de variable au niveau des règles de transformation avec la possibilité d'utilisation des expressions régulières pour les labels des arcs et des nœuds. Mais l'utilisation de cet outil pose deux problèmes :

1- La combinaison de tous les graphes en un seul pose un problème de visibilité et laisse la charge au programmeur de spécifier les nœuds et arcs à ajouter et/ou à supprimer au lieu de déterminer ceci d'une manière automatique ;
2- L'export du résultat de la transformation est limité au format JPEG (Joint Photographic Experts Group). Le résultat n'est pas donc exploitable en dehors de cet outil.

- **GrGen.NET**

GrGen.NET [Blomer 08] (Graph Rewrite GENerator,. NET) est un outil de transformation de graphe efficace qui génère un code C # (ou. NET-Assembly) à partir de la spécification de modèle de graph (*.grg) et des règles de transformation (*.gm). De ce fait GrGen utilise deux types de langage : (1) langage pour la définition du modèle de graphe (2) et un langage pour les règles. GrGen applique plusieurs nouvelles heuristiques d'optimisation. Pour accélérer l'étape de « mapping » il utilise le concept du « plan de recherche » pour représenter les

différentes stratégies de mapping en prenant en considération le graph de départ. Il s'appuie sur l'approche SPO.

Le résultat de la transformation de GrGen [Blomer 08] est la génération des class c# ou bien des Dll et en utilisant les fichiers générés, nous pouvons développer des applications qui illustre la transformation du graphe. Ceci nécessite donc de passer par une phase de compilation avant d'exécuter les transformations ce qui constitue un obstacle pour la l'adaptation dynamique des applications.

- **AGG (Attributed Graph Grammar)**

AGG est un outil qui illustre d'une manière générale l'approche algébrique y compris SPO, DPO, NAC, etc. Le grand apport dans cet outil est sa flexibilité en termes d'utilisation du concept d'attribution [Taentzer 04]. Les graphes (nœud et arc) peuvent être attribués par n'importe quel genre d'objet java et les transformations peuvent être basées sur ces objets décrits par des expressions de java. AGG permet d'appliquer des transformations à des niveaux d'abstraction très différents c'est-à-dire exprimer les règles de transformation indépendamment des attributs. Ceci est possible grâce à l'utilisation des variables pour remplacer les valeurs des attributs.

AGG définit une interface qui englobe des éditeurs graphique et d'un interprète qui est également utilisable sans interface graphique. AGG n'est pas spécialisé à un certain genre d'application mais il peut être utilisé par n'importe quel domaine [Taentzer 00].

3.2.2 Choix de l'outil

Le tableau suivant résume les différents outils de transformation de graphe que nous avons étudié pouvant satisfaire nos besoins pour l'implémentation. Selon ce comparatif, nous optons pour le choix d'AGG. En effet, c'est le seul outil complet réunissant l'ensemble des contraintes imposées par notre problématique.

	Règles	**Approche**	**NAC**	**Attribut**	**Condition sur Règles**	**Type Graph**	**Résolution Conflits**
AGG	graphique	Configurable par l'utilisateur	x	x	x	x	x
GrGen.NET	textuel	SPO	x	x	-	-	-
Groove	graphique	SPO	x	x	-	-	-
GraphSynth	graphique	SPO	-	-	-	-	-

3.2.3 Architecture du système implémenté

Le système de transformation de graphe selon AGG est représenté par la figure 3.7. La définition d'un type graphe est optionnelle. Dans nos travaux, nous utilisons un Type graph pour la validation des résultats de la transformation. Les types déclarés dans un type graphe seront utilisés pour la définition de G_{host} et des règles.

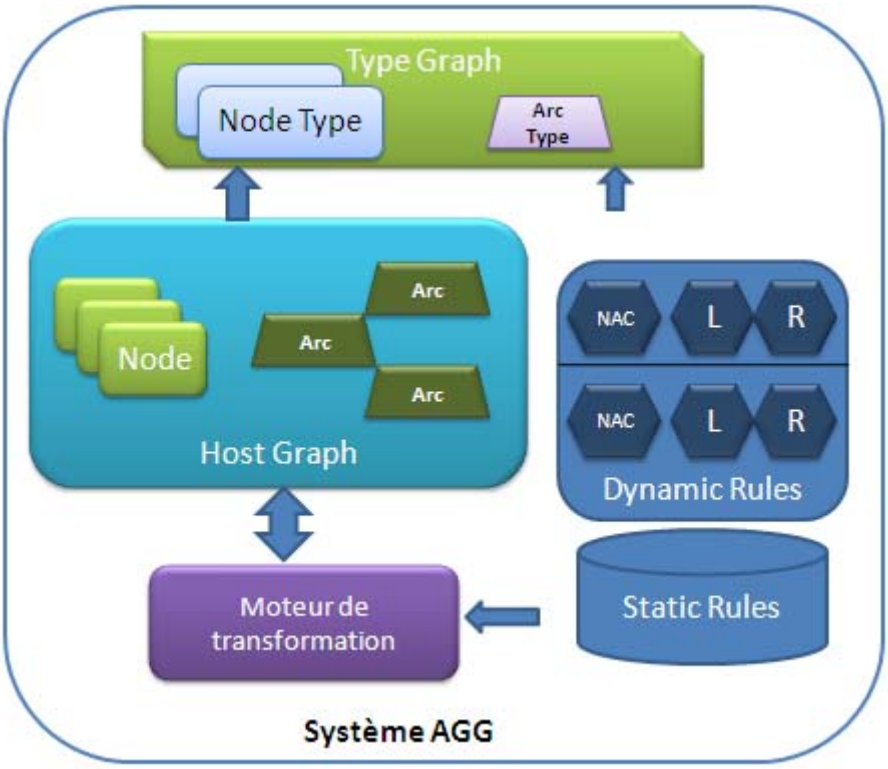


Figure 3.7 Architecture d'un système de transformation de graph AGG

G_{host} sera utilisé de trois manières différentes. En premier lieu il sert pour la fusion des L des règles de TG ensuite des R et enfin il représentera l'assemblage de composant qui sera adapté. Dans la base de règles nous avons deux types: (1) des règles statiques qui seront définies a priori et qui ne dépendrons pas de l'adaptation (2) et des règles dynamiques qui servent pour la fusion des L et R. Le moteur de transformation utilisera G_{host} et les règles selon la configuration en cours pour exécuter les transformations.

3.2.4 Exemples d'application de notre approche

➢ Exemple1 : La moyenne

Dans cet exemple, nous illustrons la composition des règles de transformation de graphe avec l'application des règles associées à un composant de sémantique connue. La sémantique que nous voulons ajouter est la moyenne de différentes températures. Pour ce faire nous utilisant deux AA qui sont :

- *Aspect d'assemblage 1 :*

Point de coupe :

s:=/Sonde*.^ev1/

Greffon :

schema pep_wcomp(s):

Create **moy**

s -> (**moy**.m1)

- *Aspect d'assemblage 2 :*

Point de coupe :

c:=/chauff*.m1/

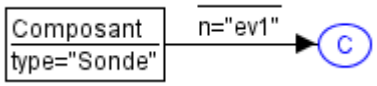
Greffon :

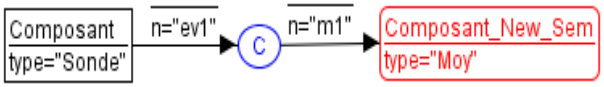
schema chauff_wcomp_2(c):

Create **moy**

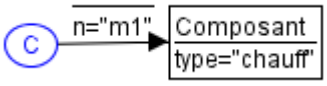
moy.^e1-> (c)

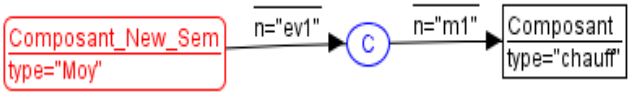
Nous écrivons donc nos règles de transformation. La première règle s'écrit comme suit :

$L_1 =$ 

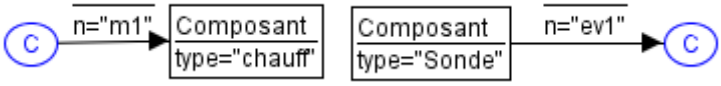
$R_1 =$ 

La deuxième règle est :

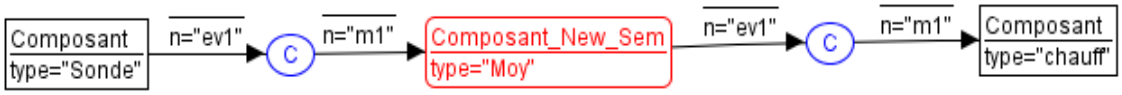
$L_2 =$ 

$R_2 =$ 

Le déroulement de mécanisme de superposition donne comme résultat la règle suivante :

$\mathbf{L_{Fnal}} =$ 

$\mathbf{R_{Fnal}} =$



L'application de cette règle sur l'assemblage de base représenté par la figure 3.8 donne le graphe de la figure 3.9.

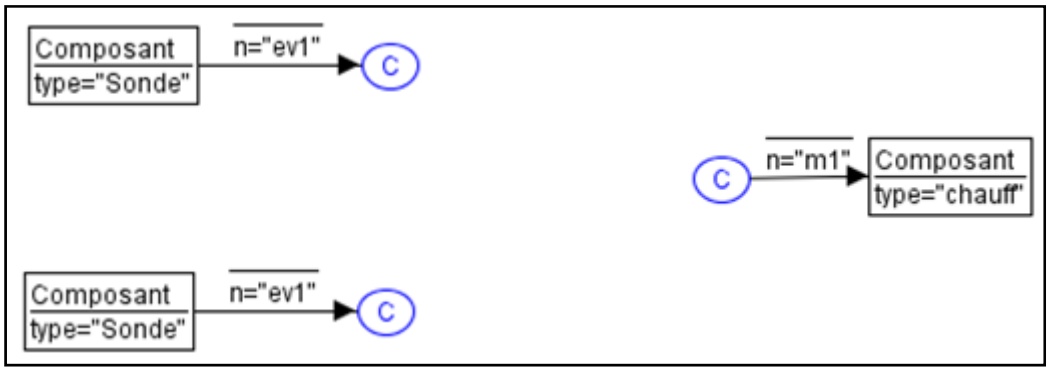


Figure 3.8 Graphe de l'application de départ

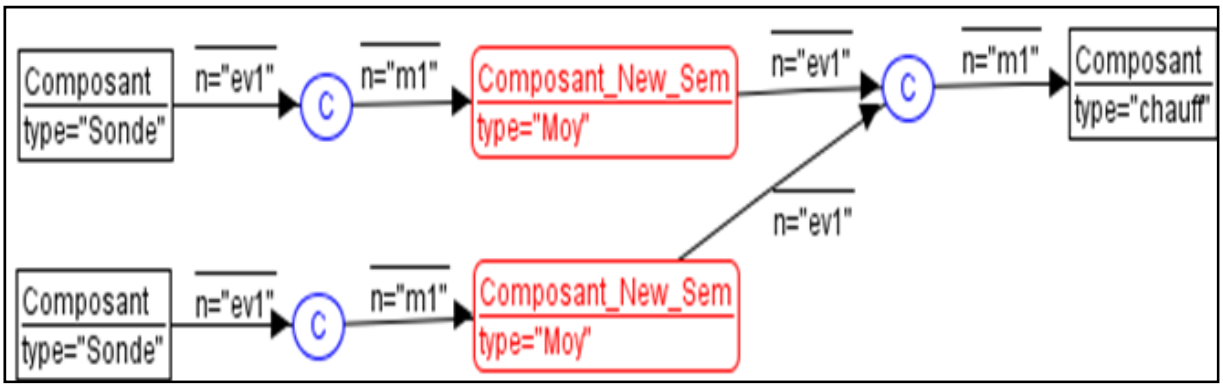


Figure 3.9 Graphe de l'application adaptée

La règle associée à cette sémantique (figure précédente **3.4**) est alors applicable. Il s'agit du même composant Moy et n'est pas deux composants différents. Nous obtenons donc comme résultat de l'adaptation l'assemblage représenté par la figure 3.10.

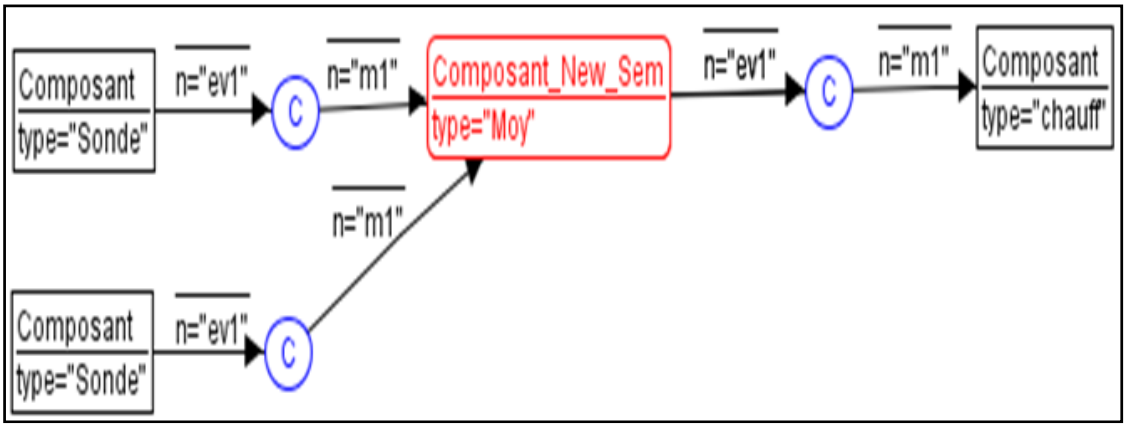


Figure 3.10 Graphe après l'application des règles associées à la sémantique moyenne.

➢ Exemple 2 : Fusion avec résolution de conflit

A travers cet exemple nous montrons le cas ou nous avons un conflit au niveau de nos règles de transformation. Nous partons de ces deux AA :

- ***Aspect d'assemblage 1***

Point de coupe :

l:=/Light*.SetStatus /

s=/Switch*.^Status_Ev/

Greffon :

schema pep_wcomp(l,s):

s -> l

- ***Aspect d'assemblage 2***

Point de coupe :

s=/Switch*.^Status_Ev/

p=/Prise*.SetStatus/

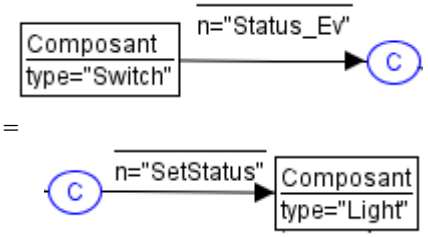
Greffon :

schema chauff_wcomp_2(s,p):

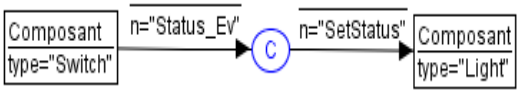
s-> p

Nos règles de transformation sont écrites de la manière suivante :

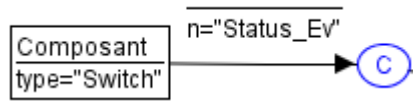
Règle1 :L_1 =



$R_{1=}$



Règle2 :L_2 =



R_2 =

Avant la résolution de conflit le mécanisme de composition pour la partie droite R donne le graphe suivant 3.11.

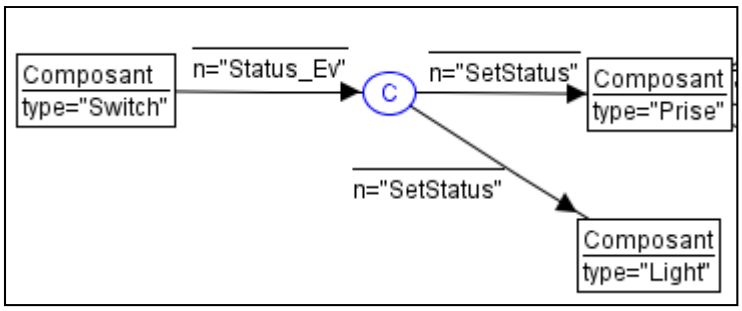


Figure 3.11 La partie droite de la règle de transformation finale

La règle de résolution est donc applicable et donne le graphe de la figure 3.12.

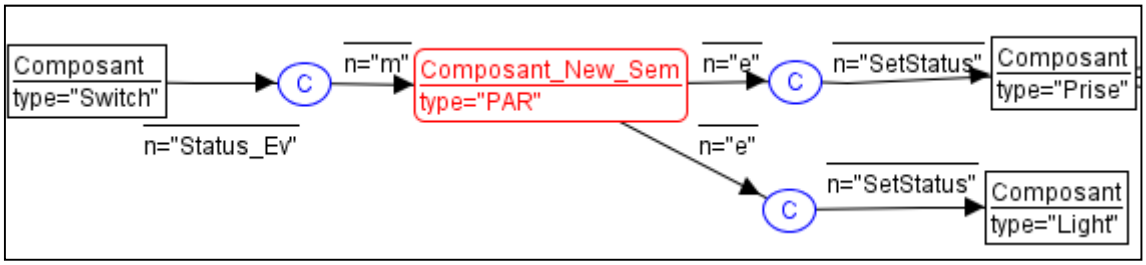


Figure 3.12 La partie droite de la règle de transformation finale

En effet, le conflit a été résolu par la création d'un composant PAR comme défini au niveau de notre règle. Nous obtenons donc la règle finale qui illustre l'adaptation à exécuter représentée par les figures 3.13 (partie gauche) et 3.12 (la partie droite).

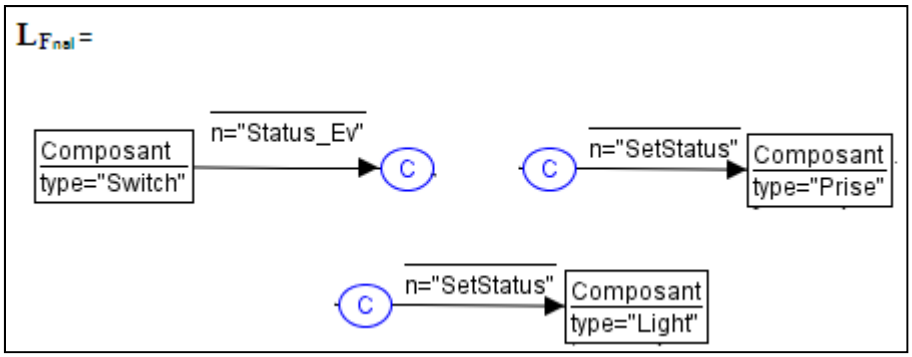


Figure 3.13 La partie gauche de la règle de transformation finale

Si nous partons d'un assemblage de base sur cette forme :

Figure 3.14 Graphe de départ de l'exemple 2

Nous obtenons comme adaptation le graphe représenté par la figure 3.15.

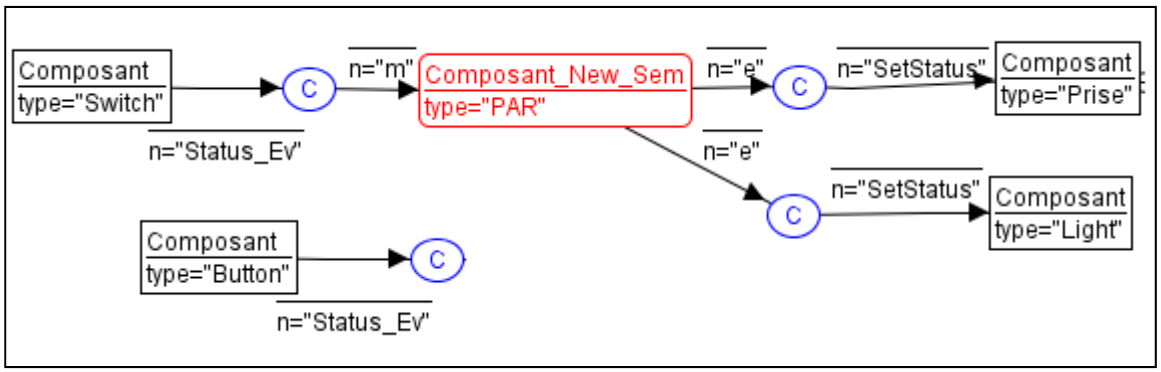


Figure 3.15 Assemblage de composants de l'application adaptée

3.3 Evaluation de notre approche

Cette étude pour la prise en compte de nouvelles sémantiques dans la modification d'assemblage de composants nous a permis de développer une nouvelle approche pour l'adaptation en utilisant comme outil les transformations de graphes.

Dans notre approche, nous avons implémenté un mécanisme de composition des règles de transformation pour résoudre le problème de conflit entre les adaptations à mettre en place. Nous associons des règles spécifiques pour le traitement des composants de nouvelles sémantique.
L'apport de notre approche est que ne sommes pas limités à un langage particulier pour exprimer les adaptations. Nous utilisons les graphes pour représenter l'assemblage de composant et des règles de transformation de graphes pour les modifications.

Toutefois, il existe un point que nous n'avons pas eu le temps d'explorer pour notre approche: s'il existe un conflit entre les règles de résolution de conflit. Dans les exemples implémentés nous avons un nombre limité de règles pour la résolution de conflit. Il sera donc intéressant de vérifier notre modèle pour des cas plus complexes d'adaptation.

3.4 Conclusion

Le modèle que nous avons présenté s'appuie sur deux principaux concepts développés pour le modèle des aspects d'assemblage. Il s'inspire des AA pour l'écriture des règles de transformations de graphe notamment la partie greffon qui illustre la modification à effectuer. Nous avons créé un nouveau mécanisme de tissage en utilisant le principe de transformations de graphe, scindé en trois opérations de composition et de résolution de conflits, renfermant le mécanisme de superposition des adaptations et notamment le traitement des composants de nouvelles sémantiques. Ces concepts s'enchaînent au moment de l'exécution de l'application pour produire des adaptations.

Conclusion et Perspectives

Les travaux réalisés dans le cadre de ce stage de Master s'inscrivent dans le domaine de l'Informatique Ambiante. Notre objectif était d'étudier et de concevoir un mécanisme d'adaptation des applications intégrant un mécanisme de résolution des conflits potentiels entre les adaptations et autorisant l'ajout de nouvelles règles pour la prise en compte des composants de nouvelle sémantique, non connue a priori. Ces sont des composants qui ne sont pas liées à un langage particulier mais qui seront rajoutés suite à l'expression d'un nouveau besoin.

Dans ce cadre nous avons présenté en premier lieu le domaine dans lequel nos travaux s'inscrivent. Avant d'aborder la problématique elle-même, il a été indispensable d'étudier les modèles d'adaptations existant et leur méthodologie dans l'utilisation des aspects. Nous avons trouvé que les travaux les plus évolués dans ce domaine, les AA, ne répondent pas à nos besoins car ils sont rattachés à un langage spécifiant les composants de sémantique connue. De ce faite le mécanisme de composition des AA ne peut pas supporter facilement l'introduction de nouvelles sémantiques, sauf à spécifier un nouveau langage d'opérateurs complets. L'étude approfondie des AA nous a permis de trouver qu'ils ne sont pas homogènes dans leur représentation. Il y a des transformations allant d'une représentation sous forme de graphe à une représentation de type langage (et vise versa). Nous avons donc identifié comme piste de recherche l'utilisation de graphes pour non seulement représenter l'assemblage de composants à modifier mais aussi toute adaptation sera vue comme une transformation de ce graphe. Il nous est donc apparu intéressant d'étudier et d'analyser si des travaux avaient été réalisés dans ce domaine toute en restant restreint à notre problématique : la composition des adaptations avec des nouvelles sémantiques.

L'étude bibliographique des travaux concernant l'adaptation par transformations de graphe nous a permis de montrer que le problème de composition des transformations n'est pas suffisamment traité et qu'il est limité à la définition explicite des priorités entre les règles ce qui est un mécanisme trop faible qui a été dépassé dans l'approche par aspects.

Nous avons donc proposé un mécanisme, inspiré de celui utilisé pour les AA, pour la composition des règles de transformation de graphes. Tout d'abord, nous avons modélisé l'assemblage de composants sous forme d'un graphe. Nous avons défini un modèle de graphe qui doit être respecté tout au long de l'adaptation afin d'assurer la cohérence de l'assemblage. Nous avons défini le format des règles de transformation. En effet, nous avons deux types de règles : statiques et dynamiques. Les règles statiques permettent de résoudre soit le conflit entre les adaptations, soit le traitement des composants de nouvelles sémantiques. Les règles dynamiques sont appliquées pendant la phase de tissage. Elles permettent de superposer les graphes qui représentent les parties gauches et droites des règles à appliquer. Nous avons

implémenté ce mécanisme de composition qui sera facilement extensible pour supporter les nouveaux besoins.

L'utilisation des règles de transformation de graphes a permis d'avoir une expressivité plus élaborée que dans les travaux précédents. En effet, les points de coupe dans les AA sont limités à la sélection de composants et les modifications seront appliquées à partir de ces points identifiés dans l'assemblage de composants. Dans la partie gauche L d'une règle de transformations nous pouvons sélectionner un sous-graphe pour le modifier, c'est-à-dire non seulement les composants mais aussi les liens entre eux. Nous passons donc de points de coupe syntaxiques à des points de coupe structurels. Nous proposons donc comme perspectives de poursuivre notre étude dans ces nouvelles problématiques qui ont pu être dégagées grâce aux résultats obtenus par ces recherches.

Bibliographie

[Alonso et al 04] **G. Alonso, F. Casati, H. Kuno,** and **V. Machiraju**, "*Web services: Concepts, Architecture, and Applications*", Springer New York, 2004.

[Baresi et al 04] **L. Baresi, R. Heckel, S. Thone** and **D. Varro**, "*Style-based refinement of dynamic software architectures*", Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture, pp. 155–166, 12-15 June, Oslo, Norway. IEEE Computer Society. 2004.

[Baresi et al 06] **L. Baresi, R. Heckel, S. Thöne,** and **D. Varro**, "*Style-based modeling and refinement of service-oriented architectures*," Software and Systems Modeling, vol. 5, no. 2, pp. 187–207, 2006.

[Berger 01] **L. Berger**, "*Mise en œuvre des Interactions en Environnements Distribués, Compilés et Fortement Typés : le Modèle MICADO*," Nice University phd, 2001.

[Blomer 07] **J. Blomer** and **R. Geiß**, "*The GrGen .NET User Manual*", University of Karlsruhe, ISSN, pp. 1432–7864, 2007.

[Bouraquadi et al. 01] **N. Bouraqadi-Saâdani** and **T. Ledoux**, "*Le point sur la programmation par aspect*,", Technique et Sciences Informatiques, vol. 20, no. 4, pp. 505–528, 2001.

[Brichau et Haupt 05] **J. Brichau** and **M. Haupt**, "*Survey of aspect-oriented languages and execution models*," European Network of Excellence in AOSD, 2005.

[Charfi 04] **A. Charfi** and **M. Mezini**, "*Aspect-oriented web service composition with AO4BPEL*", Lecture Notes in Computer Science, pp. 168–182, 2004.

[Charfi et al. 04] **A. Charfi** and **M. Mezini**, "*AO4BPEL : an aspect-oriented extension to BPEL*", World Wide Web, vol. 10, no. 3, pp. 309–344, 2004.

[Cheung-Foo-Wo 09] **D. Cheung-Foo-Wo**, "*Adaptation dynamique par tissage d'aspects*", PhD thesis, Université de Nice, 2009.

[Clark et al 99] **J. Clark, S. DeRose,** et **al.,** "*XML path language (XPath) version 1.0*", W3C recommendation, vol. 16, 1999.

[Curbera et al. 03] **F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte,** and **S. Weerawarana**, "*Business process execution language for web services*", 2002.

[David 05] **David, P.-C**. "*Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*". PhD thesis, 2005.

[David et al. 06] **P.-C. David** and **T. Ledoux**, "*Une approche par aspects pour le développement de composants Fractal adaptatifs*", RSTI-Série L'Objet (RSTI-Objet), vol. 12, no. 2-3, 2006.

[Douence et al. 02] **R. Douence** and **M. Sudholt**, "*A model and a tool for Event-based Aspect-Oriented Programming (EAOP)*", tech. rep., Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[Douence et al. 03] **R. Douence** and **J. Noyé**, "*Towards a concurrent model of event-based aspect-oriented programming*", 2006.

[Dowling et al 01] **J. Dowling** and **V. Cahill**, "*The k-component architecture meta-model for self-adaptive software*", Lecture notes in computer science, pp. 81–88, 2001.

[Dowling et al 02] **J. Dowling**, **V. Cahill**, and **S. Clarke**, "*Dynamic software evolution and the k-component model*", in Workshop on Software Evolution, OOPSLA, 2002.

[Dowling et al 04] **J. Dowling** and **V. Cahill**, "*Self-managed decentralised systems using K-components and collaborative reinforcement learning*", in Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004, Newport Beach, California, USA, October 31 - November 1, 2004.

[Ehrig 99] **H. Ehrig, G. Engels, H. Kreowski,** and **G. Rozenberg**, "*Handbook of graph grammars and computing by graph transformation: vol. 2 : applications, languages, and tools*", World Scientific Publishing Co., Inc. River Edge, NJ, USA, 1999.

[Gradit 99] **P. Gradit**, "*L'approche catégorique des grammaires de graphes*". Lecture Notes in Computer Science 2043 ,1999.

[Greenfield 06] **A. Greenfield**, "*Everyware: The dawning age of ubiquitous computing*". Peachpit Press Berkeley, CA, USA, 2006.

[Guennoun 07] **M. Guennoun**, "Architectures Dynamiques dans le Contexte des Applications à Base de Composants et Orientées Services", Thèse de doctorat, Université Toulouse III (Paul Sabatier), 2007.

[Hachni 02] **O. Hachni**, "*Apport de la programmation par aspects dans l'implantation des patrons de conception par objets*". PhD thesis, Université de Grenoble, 2002.

[Hartmut 99] **E. Hartmut** and **U. Prange**, "*Modeling with Graph Transformations*" Technical University of Berlin, Germany, 1999.

[Kiczales et al. 01] **G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm,** and **W. Griswold**, "*An overview of AspectJ*," Lecture Notes in Computer Science, pp. 327–353, 2001.

[Levy 06] **S. Levy**, "*A model and a tool for Event-based Aspect-Oriented Programming (EAOP)*", tech. rep., Technical Report 02/05/INFO, 2006.

[Pawlak et al. 05] **R. Pawlak, L. Seinturier,** and **J. Retaillé**, "*Foundations of AOP for J2EE Development*". Apress, 2005.

[Peltz 03] **C. Peltz**, "*Web services orchestration and choreography*," Computer, vol. 36, no. 10, pp. 46–52, 2003.

[Pérez et al. 05] **J. Perez, N. Ali, J. Carsi,** and **I. Ramos**, "*Dynamic evolution in aspect-oriented architectural models*", EWSA, pp. 59–76, 2005.

[Pinto et al. 05] **M. Pinto, L. Fuentes,** and **J. Troya**, "*A dynamic component and aspect-oriented platform*", The Computer Journal, vol. 48, no. 4, pp. 401–420, 2005.

[Rensink 09] **A. Rensink, I. Boneva, H. Kastenberg** and **T. Staijen**, "*User Manual for the GROOVE Tool Set*", User Manual, Department of Computer Science, University of Twente.

[Taentzer 00] **G. Taentzer**, "*AGG : A tool environment for algebraic graph transformation*", Lecture notes in computer science, pp. 481–488, 2000.

[Taentzer 04] **G. Taentzer**, "*AGG : A graph transformation environment for modeling and validation of software*", Lecture Notes in Computer Science, vol. 3062, pp. 446–453, 2004.

[Van Deursen et al. 00] **A. Van Deursen** and **J. Visser**, "*Domain-specific languages: An annotated bibliography*", ACM Sigplan Notices, vol. 35, no. 6, pp. 26–36, 2000.

[Wermelinger 99] **M. Wermelinger** and **J. Fiadeiro**, "*Algebraic software architecture reconfiguration*", ACM SIGSOFT Software Engineering Notes, vol. 24, no. 6, pp. 393–409, 1999.

[Weiser 91] **M. Weiser**, "*The Computer for the Twenty-First Century*". Scientific American, Vol. 1, pp. 94-104, 1991.