Information Systems and Technologies

ICIST'2011

Edited by

Mohamed Ridda LAOUAR

Information Systems and Technologies

Edited by
Mohamed Ridda LAOUAR

ISBN : 978-9931-9004-0-5

Conference Committees

Conference Chairs

- Mohamed Ridda LAOUAR, University of Tebessa, Algeria.
- Farid MEZIANE, Salford University, United Kingdom.

Technical Program Committee

- Abdallah BOUKERRAM, Setif University, Algeria
- Abdelaziz BOURAS, Lyon 2 University, France
- Abdelazziz KHARDAOUI, Geneva University, Switzerland
- Abdelkarim AMIRAT, Souk Ahras University, Algeria
- Abdelmalik Taleb-Ahmed, University of Valenciennes, France
- Adil Al-Yasiry, University of Salford, UK
- Ahmed AIT-BOUZIAD, Boumerdès University , Algeria
- Amar SIABDELHADI, Oran University, Algeria
- Amine ABDELMALEK, University of Saida, Algeria
- Amit Mitra, the west of England University, UK
- Aris M. OUKSEL, University of Illinois at Chicago, USA
- Azzeddine BILAMI, University of Batna, Algeria
- Baghdad ATMANI, Oran University, Algeria
- Bornia TIGUIOUART, University of Annaba, Algeria
- Bouziane BELDJILALI, USTO, Oran, Algeria
- Camille SALINESI, Paris I Panthéon-Sorbonne, France
- Cherif FOUDIL, University of Biskra, Algeria
- Colette ROLLAND, Paris I Panthéon-Sorbonne, France
- Djamel BOUCHAFFRA, State University of New York at Buffalo, USA
- Djamel Eddine SAIDOUNI, University of Constantine, Algeria
- Djamel MESLATI, University of Annaba, Algeria
- Djemel ZIOU, Sherbrook university, Canada
- Djamila HAMDADOU, Oran University, Algeria
- Driss ABOUTAJDINE, (IEEE Morocco section, Rabat University), Morocco
- Emilia MENDEZ, Auckland University, New Zeland
- Epaminondas Kapetanios, university of Westminster, UK
- El Hassan ABDELWAHED, Cadi Ayyad, Morocco
- Faiez GARGOURI, ISIMS, University of Sfax, Tunisia

- Farid MEZIANE, Salford University, United Kingdom.
- Farid MOUKHATI, University of Oum el Bouaghi, Algeria
- Farida SEMMAK, Paris 12, France
- Farouk YALAOUI, Troyes University , France
- Fatiha SADAT, UQAM, Canada
- Fatima BENDELLA, Oran University, Algeria
- Ghalem BELALEM, Oran University, Algeria
- hafida BELBACHIR, USTO, Oran, Algeria
- Haikal El Abed, ICT, Braunschweig University, Germany
- Hakim BENDJENNA, University of Tebessa, Algeria
- Hassina SERIDI, University of Annaba, Algeria
- Hayet MEROUANI, University of Annaba, Algeria
- Henri PIERREVAL, IFMA, France
- Imed KACEM, Metz, France
- Jamel FEKKI, Miracle Laboratory, University of Sfax, Tunisia
- Jean-Aymon MASSIE, President of AFGE, Paris, France
- Jérôme DARMONT, Lyon 2, France
- Karim BOUAMRANE, Oran Université, Algeria
- Kameleddine MELKEMI, University of Biskra, Algeria
- Kamal BECHKOUM, Northampton University, UK
- Krishnaprasad Thirunarayan, Wright State University, USA
- Labib TERESSA, University of Biskra, Algeria
- Labiba SOUICI-MESLATI, University of Annaba, Algeria
- Ladjel BELLATRECHE, ENSMA (Poitiers), France
- Latifa BABAHAMED, Oran University, Algeria
- Laure BERTI-EQUILLE, Rennes 1 University, France
- Lionel AMODEO, Troyes University, France
- Lynda ZAOUI, Oran University, Algeria
- Maamar BETTAYEB, University Of Sharjah, United Arab Emirates
- Madjid MERABTI, Liverpool John Moores University, United Kingdom
- Mahieddine DJOUDI, Poitiers University, France
- Mahmoud BOUFAIDA, University of Constantine, Algeria
- Messabih BELHADRI, Oran University, Algeria
- Malik SI-MOHAMMED, University of Tizi-Ouzou, Algeria
- Michel SIMONET, IMAG, France
- Mohammed Amine Chikh, University of Tlemcen, Algeria
- Mohamed Bachir MENAI, King Saud University, Saudi Arabia
- Mohamed BATOUCHE, King Saud University, Saudi Arabia

- Smaine MAZOUZI, Université of Skikda, Algeria
- Smail NIAR, UVHC, Valenciennes, France
- Souham MESHOUL, King Saud University, Arabie Saoudite
- Tahar BOUHADADA, University of Annaba, Algeria
- Thouraya TEBIBEL, ESI, Algeria
- Vijay Sugumaran, Oakland University, USA
- Wassim JAZIRI, ISIMS, Sfax, Tunisia
- Yagoubi BELABBES, Oran University, Algeria
- Yahya Mohamed Elhadj, University AL Imam Muhammad Ibn Saud, Saudi Arabia
- Yahya SLIMANI, Tunis Faculty of Sciences, Tunisia
- Zakaria ELBERRICHI, Sidi Bellabbes University, Algeria
- Zizette BOUFAIDA, University of Constantine, Algeria

Organization Committee

- Louardi BRADJI, University of Tebessa, Algeria (President)
- **Kamal HAOUAM**, University of Tebessa, Algeria (Vice-President)
- Mohamed Yassine HAOUAM
- Mohamed AMROUNE
- Halim CHAABANE
- Rouh Allah BENABOUD
- Nouzha HARATHI
- Issam BENDIB
- Chawki DJEDDI
- Abdelgafour AZZEDINE
- Akram BENOUR
- GATTAL Abdeldjalil
- Samir TAG
- Mekhazenia TAHAR
- Abdelmoumen ZEBDI
- Abdelmalek MATROUH
- Abdelhalim CHABANE
- Houda AZAZ
- Amar ZEMMAR
- Nabil OUAZENE
- Salima BOUROUGAA
- Rachid MAHMOUDI

- Abderrahim SIAM
- Samir KHEDAIRIA
- Lotfi AMIAR

Conference Secretariat

- NOUIOUA Tarek, University of Tebessa, Algeria (General secretary)
- GOUGHA Hamza
- DJABRI Mustafa Rédouane (Webmaster)
- BELAID Narjess
- BOUGUESSA Brahim
- BOUGHANEM Madjid
- BOUHAMLA Soumaia
- BENOUR Abderazzak
- DJEDDI Abdelhakim

ATTESTATION

Hereby we confirm the participation of **CHERIET HANANE**

At the 1st International Conference on Information Systems and Technologies – ICIST 2011

Which took place on April 24th-26th at the University of Tebessa - Algeria

Title to an poster presentation: **Vers un Modèle d'Evolution des Programmes Orientés Aspect**

Author (s): **hanane cheriet and nora bounour**

Date: April 24th 2011

Chair of ICIST'11
Mohamed Ridda LAOUAR

Vers un Modèle d'Evolution des Programmes Orientés Aspect

Hanane Cheriet[1], Nora Bounour[2]
Département d'Informatique
Laboratoire de Recherche en Informatique (LRI)
University Badji Mokhtar Annaba, BP.12, 23000, Annaba
Algérie
[1]*cheriet_hanane@yahoo.fr*
[2]*nora_bounour@yahoo.fr*

Résumé

La programmation orienté objet est l'approche de choix pour la plupart des projets logiciels actuels. Elle est généralement efficace dans l'expression des fonctionnalités dites verticales. Par contre, elle s'avère particulièrement limitée dans l'expression des fonctionnalités dites horizontales ou transversales, celles exprimant les aspects techniques de l'application. Les chercheurs se sont penchés sur ce sujet et ont conçu la programmation orientée aspect afin de pallier cette faiblesse. Comme tout autre système, le logiciel orienté aspect a besoin d'évoluer. Mais il n'existe pas jusqu'à maintenant un modèle explicite d'évolution de ces logiciels. D'un autre coté, pour bien traiter le phénomène d'évolution on doit premièrement modéliser cette dernière, puis garder l'historique pour analyser l'évolution. Généralement, les approches actuelles d'évolution traitent séparément ces deux concepts. Pour faire face à ces problèmes nous allons, analyser les besoins des systèmes orientés Aspect et proposer un modèle assurant leur évolution (modéliser l'évolution et garder l'historique en même temps).

Mots clés— AOP, Evolution de logiciel, Réécriture de graphe, Systèmes de contrôle de versions.

1. Introduction

Les systèmes logiciels réels exigent le changement continu et l'amélioration pour satisfaire les nouveaux besoins de l'utilisateur, et aussi éviter la dégradation de la structure du logiciel [3]. L'évolution de logiciel consiste en une maintenance continue couvrant le cycle de vie entier du système logiciel. Différents travaux ont été menés dans le cadre de la maîtrise de l'évolution de logiciels, nous avons proposé une classification de ces travaux ([1], [2]).

D'après l'analyse de ces différents travaux, nous avons constaté que tous les modèles d'évolution basés sur le code source ne traitent pas les systèmes orientés aspect. Dans ce papier, nous allons analyser les besoins des systèmes orientés aspect afin d'introduire un modèle général pour assurer leur évolution. Nous visons par « général » le fait que le modèle ne doit pas seulement modéliser l'évolution, mais il doit garder l'historique en même temps. Notre modèle est constitue l'hybridation de deux approches existantes pour assurer l'évolution des systèmes orientés objets ; une pour la modélisation formelle d'évolution « ***la réécriture de graphe algébrique*** [5] » et une autre « ***modèle basé changements*** [7]» pour garder l'historique d'une façon fiable comme nous montrons par la suite. Nous allons donner brièvement le principe de chaque approche ainsi que les raisons pour lesquelles nous les avons choisies.

Depuis plusieurs années, de nombreux travaux concernant la modélisation et la gestion des artefacts logiciels ont donné naissance à différentes représentations que nous regroupons en trois catégories

dépendant de la façon dont le logiciel est perçu, c'est-à-dire comme un graphe, une base de données ou un hyper-document. D'après ces représentations, les graphes sont des structures de données d'une grande importance pour de nombreuses applications de l'ingénierie des logiciels [4].

Les graphes sont utilisés pour représenter des objets (concepts) complexes pour lesquels, les relations entre les composants sont primordiales. L'analyse du logiciel et plus spécifiquement celle du code source, pour des besoins de compréhension et d'évolution a conduit depuis longtemps à privilégier la structure de graphe comme moyen simple et adapté à la modélisation et à la manipulation. Les graphes de flots de données, les graphes de flots de contrôle ainsi que les graphes d'héritage forment des exemples de structures communément utilisées dans le cadre de l'analyse des codes sources du logiciel. Au sein d'un code source, les artefacts sont liés entre eux à des niveaux abstro-granulaires différents [4].

A partir de la représentation de logiciel comme un graphe, ***la réécriture de graphe*** est l'approche la plus fiable pour garantir l'évolution. Les demandes d'évolution peuvent être formalisées en utilisant la réécriture du graphe algébrique (ex. [5]). Dans [5] les diagrammes de classe UML sont convertis en graphes colorés. Les nœuds dans le graphe représentent les composants du système ; les arêtes décrivent la relation entre les composants. L'approche fournit un ensemble de règles de réécriture du graphe algébrique qui formalise les changements qui peuvent être causés sur une demande d'évolution.

La réécriture de graphe algébrique est basée sur des fondements formels qui augmentent la sûreté et la validité d'évolution. Donc, c'est l'approche la plus convenable pour assurer l'évolution délicate des systèmes orienté aspect. De plus, la notion des graphes est très populaire dans le domaine d'évolution (techniques, outils,…). Mais pour bien analyser l'évènement d'évolution nous avons besoin d'un autre aspect (technique), qui garde l'historique de l'évolution « versioning systems». L'historique d'un système est très important dans beaucoup de technique d'évolution : analyser l'évolution, rétro-ingénierie, comparer différentes versions,…..etc. Cette nécessité est assurée avec les systèmes de contrôle de versions (ex. CVS, Subversion).

Les outils de contrôle de versions courants n'entreposent pas toute l'information générée par les développeurs. Ils n'enregistrent pas chaque version intermédiaire du système publié, mais seulement des versions instantanés prises quand un développeur entrepose le code source dans le dépôt. D'un autre coté, les approches traditionnelles modélisent l'historique d'un programme comme une séquence de versions, cela consomme la mémoire, puisque la plupart des parties du système ne changent pas et tous simplement dupliquées dans les versions [6].

A cause de ces limites, Romain Robbes et Michele Lanza proposent un modèle d'évolution qui est basé sur ***la modélisation du changement*** [7]. L'idée de ce modèle consiste à modéliser les activités du développement incrémentales de mainteneurs et développeurs (le changement) pour refléter attentivement ce qui arrive. Cette approche entrepose seulement les différences entre les versions au niveau du programme, et est capable de le reproduire à tout point dans le temps.

2. Inspiration

Nous allons nous inspiré des deux modèles présentés plus haut et bénéficier de leurs avantages pour proposer un modèle combinant les deux pour assurer l'évolution des codes source orientés aspect. Donc, notre modèle, présenté dans les prochaines sections, est basé sur les points suivants:

- Nous exploitons la puissance formelle des graphes pour modéliser notre système orienté aspect. Où nous nous inspirons de l'approche de réécriture de graphe algébrique [5].

- Nous modélisons les demandes d'évolution sur le système comme des règles de réécritures. La puissance formelle de ces dernières assure une évolution fiable des systèmes orientés aspect, surtout au niveau des points d'intégration entre le code de base et les différents aspects du système. Où nous pouvons éviter les conflits entre les aspects.

- Nous gardons l'historique d'évolution comme des séquences de règles de réécriture. Où nous proposons un nouveau type de système de contrôle de versions dans lequel le dépôt est basé-règle de réécriture (dépôt formel).

TABLEAU 1
CONCEPTS D'UN PROGRAMME ORIENTE ASPECT

Concept	Rôle
Point de Jointure (*joinpoint*)	Endroit précis dans l'exécution du programme. Par exemple, un appel à une méthode, à un constructeur … Les Consignes sont insérés au niveau des Points de Jointure.
Point de Coupure (*pointcut*)	Constitue le moyen de spécifier un ensemble de Points de Joiture particuliers. Une Coupe est souvent une expression régulière.
Consigne (*advice*)	Fragment de code à insérer au niveau des Points de Jointure. Implémente une préoccupation transversale.
Aspect	C'est une unité de regroupement : • D'une ou de plusieurs définitions de Points de Coupure • D'une ou de plusieurs définitions de Consignes. • D'une ou de plusieurs associations de Points de Coupure à des Consignes
Tisseur (*weaver*)	Est un outil spécial permettant d'appliquer les aspects au code de base.

3. Fondements des systèmes orientés aspect

La programmation orientée aspect (AOP : Aspect Oriented Programing) est une nouvelle méthodologie qui permet de séparer les préoccupations subsidiaires qui entrecoupent les fonctionnalités principales d'un système [8]. On pense par exemple aux fonctionnalités telles que l'authentification, l'autorisation ou la journalisation qui interviennent généralement avant ou après l'appel des fonctions principales. Jusqu'ici, le code associé aux requis subsidiaires se trouvait généralement dispersé un peu partout dans le programme principal. Ainsi disséminées, ces fonctions étaient inévitablement plus difficiles à maintenir. Grâce au paradigme aspect, le code des préoccupations transversales peut être regroupé au sein de modules spéciaux appelés « aspects », au lieu d'être dispersé dans les classes du système.

L'AOP permet de résoudre les problèmes dus à l'enchevêtrement et l'éparpillement du code. Elle permet aussi de modulariser l'implémentation des problématiques transversales, de créer des systèmes plus évolutifs et d'assurer une meilleure réutilisation du code. De plus, les études montrent que la surcharge introduite par les approches orientées aspect est relativement faible. Enfin, les implémentations orientées aspect ont des niveaux d'adaptabilité et de réutilisation plus élevés que les implémentations uniquement objet.

Avant de présenter notre modèle d'évolution pour les logiciels orientés aspect nous allons passer brièvement sur les concepts fondamentaux de ces systèmes. La programmation orientée aspects est une technique de structuration de programmes permettant la séparation des préoccupations dans les logiciels. Elle se fonde sur une séparation claire entre les préoccupations « métiers » (ou fonctionnelles) et non-fonctionnelles (ou techniques) présentées dans les applications. La décomposition d'une application fait apparaître :

-- Le code de base qui définit l'ensemble des services (i.e. fonctionnalités) réalisés par l'application. Autrement dit, l'aspect de base correspond au "Quoi" de l'application.

-- plusieurs aspects complémentaires qui précisent les mécanismes régissant l'exécution de l'application c'est-à-dire les aspects non-fonctionnels définissant le "Comment" (par exemple la synchronisation, la persistance ou la sécurité).

Les aspects sont utilisés pour regrouper des choix d'implémentation qui ont un impact sur l'ensemble du système et qui autrement seraient éparpillés à travers tout le code. Chaque aspect est destiné à être développé de façon indépendante puis intégré à une application par un processus dit de tissage d'aspects (aspect weaving). La construction d'une application à partir de différents aspects nécessite une étape "d'assemblage". En effet, les aspects étant des modules définis séparément les uns des autres, il faut définir leurs règles d'intégration pour les composer afin de "construire" l'application. D'où le besoin d'un mécanisme de composition pour réaliser cet assemblage.

Donc, de nouveaux concepts sont introduits avec l'AOP afin de permettre aux développeurs de spécifier et d'implémenter les préoccupations transversales. Le tableau_1 présente ces différents concepts. D'après ce tableau, le concept clés qui assure l'intégration entre le code de base et les aspects du système est « Les points de jointure ». Ils sont des points particuliers dans le graphe dynamique des appels. Il existe neuf sortes de

LISTING1
LE PROGRAMME POINT_SHADOW_PROTOCOL

```
public class Point {
protected int x, y;
public Point(int _x, int _y) {
x = _x;
y = _y;
}
public int getX() {
return x;
}
public int getY() {
return y;
}
public void setX(int _x) {
x = _x;
}
public void setY(int _y) {
y = _y;
}
public void printPosition() {
System.out.println("Point
at("+x+","+y+")");
}
public static void main(String[] args) {
Point p = new Point(1,1);
p.setX(2);
p.setY(2);
}
}
class Shadow {
public static final int offset = 10;
public int x, y;
Shadow(int x, int y) {
this.x = x;
this.y = y;
public void printPosition() {
System.outprintln("Shadow at
("+x+","+y+")");
}
}
```

```
aspect PointShadowProtocol {
private int shadowCount = 0;
public static int getShadowCount() {
return PointShadowProtocol.
aspectOf().shadowCount;
}
private Shadow Point.shadow;
public static void associate(Point p, Shadow s){
p.shadow = s;
}
public static Shadow getShadow(Point p) {
return p.shadow;
}
pointcut setting(int x, int y, Point p):
args(x,y) && call(Point.new(int,int));
pointcut settingX(Point p):
target(p) && call(void Point.setX(int));
pointcut settingY(Point p):
target(p) && call(void Point.setY(int));
after(int x, int y, Point p) returning :
setting(x, y, p) {
Shadow s = new Shadow(x,y);
associate(p,s);
shadowCount++;
}
after(Point p): settingX(p) {
Shadow s = new getShadow(p);
s.x = p.getX() + Shadow.offset;
p.printPosition();
s.printPosition();
}
after(Point p): settingY(p) {
Shadow s = getShadow(p);
s.y = p.getY() + Shadow.offset;
p.printPosition();
s.printPosition();
}
}
```

points de jointure :
– Points de jointure réception d'appel de méthode et constructeur
– Points de jointure appel de méthode et constructeur
– points de jointure exécution de méthode et constructeur
– Points de jointure d'accès à un champ (get et set)
– Points de jointure exécution d'un handler d'exception

4. Modèle d'évolution pour les systèmes orientés aspect

4.1 Modélisation du code source orienté aspect

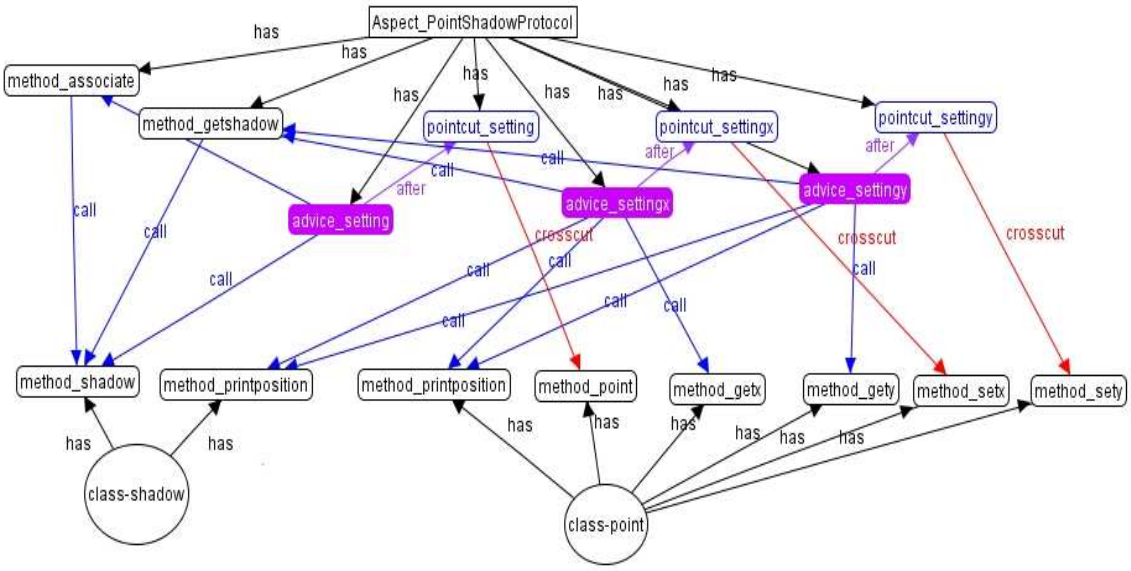
Notre approche consiste à modéliser le code source orienté aspect comme un graphe. Au lieu d'utiliser les diagrammes de classe et de séquence UML pour introduire le graphe (méthode de Ciraci [5]), on le génère directement à partir du code source orienté aspect (voir la section mise en œuvre).

Un système orienté aspect est composé de deux parties principales : le programme de base et les aspects du système. De plus, les points de coupure jouent un rôle fondamental dans l'intégration de ces deux parties.
D'après tout ça, notre modèle d'évolution est basé sur les points suivants :

a. la modélisation du code de base : le code de base du système orienté aspect est modélisé avec un graphe algébrique (ce code ni autre qu'un programme orienté objet). Les nœuds dans le graphe représentent les entités du système et les arêtes décrivent les relations entre ces entités : (1) les attributs et les méthodes qui appartiennent à une classe ; (2) la connexion entre les classes ; (3) les appels entre les différents entités (voir [5] pour plus de détails).

b. la modélisation des aspects : chaque aspect du système est aussi modélisé avec un graphe marqué. Ce dernier est semblable au graphe qui modélise un système orienté objet, mais en ajoutant d'autres concepts propres aux systèmes orientés aspect.

FIGURE 1
LE GRAPHE MODELISANT LE PROGRAMME POINTSHADOWPROTOCOL[1]



Un graphe marqué a deux alphabets de couleur, un pour colorer les arêtes et un autre pour colorer les nœuds. Dans notre modèle des systèmes orientés aspect les éléments de l'alphabet des couleurs sont les suivants :

-Aspect
-Attribute: « Type »
-Method
-Parameter: « Type»
-Return value: « Type »
- Pointcut
-Advice

Les arêtes dans notre modèle sont utilisés pour identifier les relations entres les différents composants du système. Nous avons trois classes de relations:

– la première spécifie quels attributs et méthodes appartiennent à un Aspect, ces arêtes sont annotés avec l'une des couleurs suivantes : *Has (private, public, protected) : attribute, method ; Takes parameter* ; ou la couleur *returns* pour la valeur retournée.

– La deuxième classe spécifie les points de coupure et les consignes de l'Aspect, ces arêtes sont annotés avec: *Has advice, Has pointcut*. De plus, les relations entre les consignes et les points de coupure, en utilisant les couleurs : *befor*, *after* et *around*.

– La troisième classe capture les relations entre les objets du système, i.e. les appels entre les méthodes ou entre les consignes et les méthodes. Ces arêtes sont annotées avec *call*.

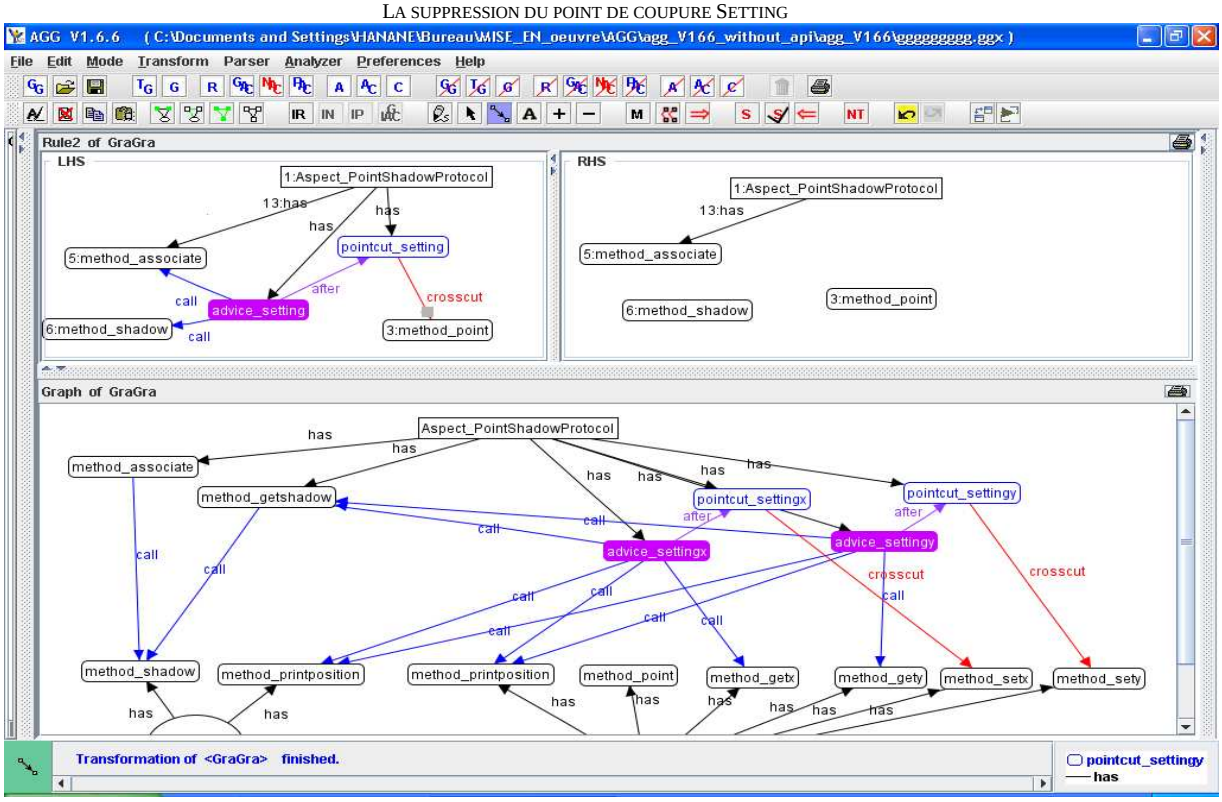
c. la modélisation du système global : pour intégrer les différents sous-graphes qui représentent les classes et les aspects, nous avons besoin de deux types d'arête pour relier les sous-graphes (nous pouvons les appeler aussi les arêtes de dépendances):

– Arêtes d'appel depuis les méthodes ou les consignes des aspects vers les méthodes des classes : annotées avec *call* ;
– Des arêtes qui relient chaque point de coupure avec ses points de jointure dans le code de base: annotées avec *crosscut* plus un des opérateurs logiques « *and, or* », s'il y a plus d'un point de jointure pour un point de coupure.

La figure 1 montre un exemple d'un graphe représentant le programme orienté aspect du Listing 1 : le programme associé Shadow Points avec chaque Point objet, il contient un aspect PointShadowProtocol qui entrepose un objet de l'ombre (shadow Point) dans chaque Point, et deux classes Point et Shadow. Nous avons éliminé les éléments non affectés par les arêtes de dépendance pour simplifier le graphe.

[1] Ce graphe a été fait par l'outil AGG : Attributed Graph Grammar. Il est basé sur la notion de règles de transformation de graphe.

FIGURE 2
LA SUPPRESSION DU POINT DE COUPURE SETTING



4.2 Formalisation des demandes d'évolution

Notre approche fournit un ensemble de règles de réécriture du graphe: ajout, suppression, modification des différents éléments du graphe (classes, aspects, méthodes, points de coupures,…etc) qui formalise les changements qui peuvent être causés par une demande d'évolution. Alors ces règles peuvent être combinées pour se rendre compte juste de plusieurs demandes d'évolution dans une manière du top-down comme un algorithme. Par exemple, une demande d'évolution qui exige l'addition d'un Aspect peut être formulée en utilisant la règle de l'addition de l'Aspect et les règles de l'addition de ses différents éléments (attributs, méthodes, points de coupures, consignes,…).

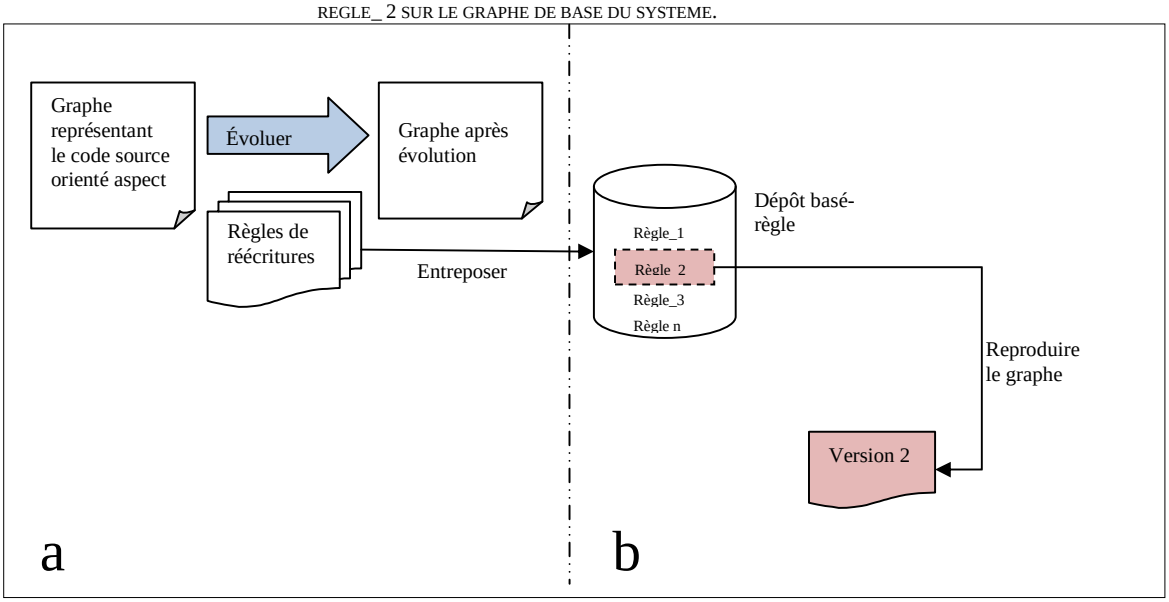
Les points clés dans les systèmes orientés aspect sont les points de coupure qui relient les différents modules (aspects, classes) du système. Donc, les demandes de changement sur ces points sont les plus intéressants. Par exemple, la suppression d'un point de coupure exige la suppression de toutes ses relations (les arcs qui le relie avec les autres entités du système), plus que la suppression de l'advice associé à ce point de coupure (figure 2).

Une règle de réécriture décrit la transformation d'un terme à un autre et peut être gardée par une condition. Il a la forme ***R: l -> r*** où ***R*** est le nom de la règle, ***l*** le pattern gauche (pré-condition) et ***r*** le pattern droit (post-condition) de la règle. Le pattern gauche est comparé à un terme et s'ils sont égaux, le pattern droit est créé pour construire le nouveau terme. Des règles multiples peuvent avoir le même nom, et les règles sont toujours invoquées par nom [9].

La figure 2 montre la règle qui supprime le point de coupure ***Setting***. Ici la règle est constituée de deux partie, la partie gauche présente la pré-condition de la règle et la droite sa post-condition. L'application de cette règle sur le graphe du système nous donne le nouveau graphe représenté dans le bas de la figure.

L'addition et la modification d'un élément du système sont réalisées avec la même manière. A travers l'application de différentes règles sur le système orienté aspect nous pouvons effectuer des demandes de changement complexes. De plus des règles de

FIGURE.3. SYSTEME DE VERSION BASE REGLE DE REECRITURE. ICI POUR EXTRAIRE LA VERSION_2 DU SYSTEME, ON APPLIQUE SEULEMENT LA REGLE_ 2 SUR LE GRAPHE DE BASE DU SYSTEME.



réécriture, nous pouvons aussi définir les différentes propriétés et les conditions qui doivent êtres préservées dans le graphe.

4.3 Garder l'historique

Comme nous avons montré dans la section 1, nous avons besoin de garder l'historique de l'évolution d'un système pour l'utiliser plus tard dans l'analyse et la validation de cette évolution. Pour intégrer ce concept dans notre modèle d'évolution pour les systèmes orientés aspect nous allons suivre le même principe de l'approche de modélisation du changement [7] où :

– Les opérations exécutables de changement qui sert à naviguer entre les différentes versions du système sont les règles de réécriture sur le graphe du code source.

– Nous proposons un dépôt basé règle comme un système de version : au lieu d'enregistrer tous le graphe changé comme une version, nous enregistrons seulement les séquences d'évolution (l'ensemble de règles de réécriture) sur ce graphe (figure 3.a). Nous pouvons reproduire chaque version du système par l'exécution de la séquence de règles de réécriture associées (figure 3.b).

Donc, tous les bénéfices de l'approche [7] sont gardés dans notre modèle. De plus, on gagne l'indépendance du langage, car le même modèle peut être appliqué à différents langages de programmation où nous utilisons différents concepts reliés au programme modélisé.

4.4 Mise en œuvre de l'approche

La figure 4 représente la mise en œuvre de notre approche (modèle) où:

1 : le code source AspectJ est convertit en format XML [11] à travers la puissance de AspectJML [14].

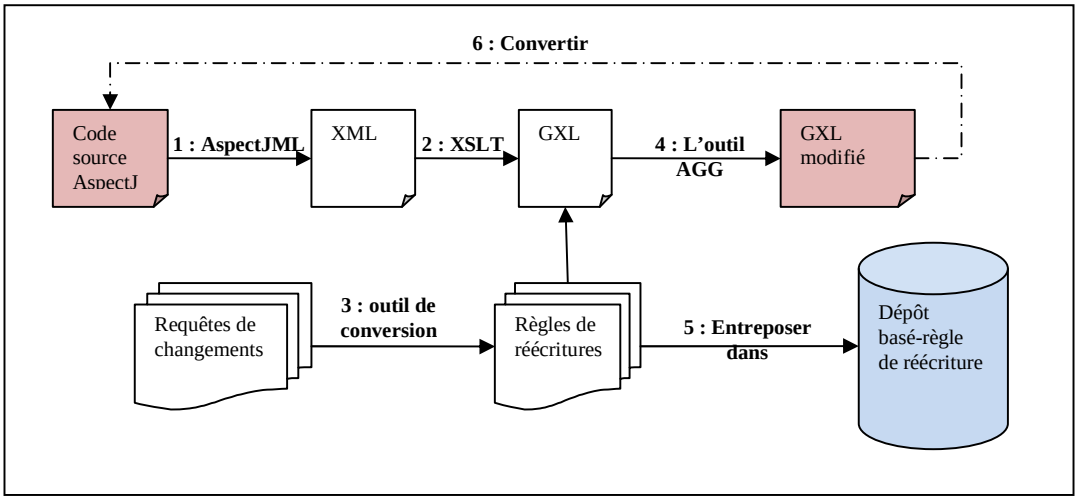
2 : à travers les outils XSLT [12] le document XML est convertit en GXL [13], pour représenter le code source comme un graphe. Le graphe obtenue sera marqué avec des étiquettes en utilisant XSLT.

3 : nous proposons d'utiliser un outil pour automatiser la conversion des demandes de changements en règles de réécriture. Cette étape est très importante pour ne pas exiger du mainteneur à apprendre les concepts des règles de réécriture.

4 : nous utilisons l'outil AGG [10], qui est un outil puissant de transformation des graphes. Il a été utilisé dans beaucoup de travaux de recherche. Il est capable de manipuler des graphes GXL ou autres. nous pouvons aussi formuler des propriétés, des contraintes, analyser le graphe, calculer les métriques, …etc. donc nous avons tous pour assurer l'évolution (en combinant les règles de réécritures de base).

5 : à chaque demande d'évolution nous enregistrons la règle qui cause cette évolution dans un dépôt basé-règle pour l'utiliser comme un système de contrôle de versions (nous pouvons reproduire le graphe (version du programme) à travers la règle appliquée.

FIGURE. 4
MISE EN ŒUVRE DE L'APPROCHE



6 : après la modification du graphe, nous pouvons régénérer le code source aspectJ en suivant le chemin inverse : convertir le graphe GXL en XML via XSLT, puis en AspectJ via AspectJML.

5. Conclusion

Nous avons proposé dans ce papier un modèle d'évolution pour les codes source orientés aspect. Notre approche consiste à modéliser le code source comme un graphe représentant les différentes entités du système et les relations entre eux. Les demandes d'évolution sont formalisées en utilisant des règles de réécriture sur le graphe du système. Nous pouvons combiner plusieurs règles pour effectuer les différentes demandes d'évolution d'un système orienté aspect. D'un autre coté, nous avons proposé un système de contrôle de versions basé-règle pour garder l'historique de l'évolution en éliminant les limites des systèmes de contrôle de versions actuelles. Ce papier présente l'idée générale de notre modèle d'évolution des systèmes orientés aspect.

6. Références

[1] Hanane cheriet, Nora Bounour, "Software Evolution: Models and Challenges", *In Proceedings of International Conference on Machine and Web Intelligence (ICMWI'2010),* Algiers, Algeria, October 3-5 2010, pp.458-460.

[2] Hanane cheriet, Nora Bounour, "Une classification des modèles d'évolution de logiciels", *In Proceedings* of *MAnifestation des JEunes Chercheurs en STIC (MajecSTIC'10*), Bordeaux, France, 13- 15 Octobre 2010

[3] Lehman MM and Belady L, *"Program Evolution – Processes of Software Change"*, Academic Press, 1985

[4] PM. Oum Oum Sack , M.Bouneffa, Y. Maweed. "Expérimentation de GXL pour l'interopérabilité des outils de réingénierie du logiciel". 2005

[5] S. Ciraci, P.M. van den Broek, "Modeling Software Evolution using Algebraic Graph Rewriting", *In Proceedings of Workshop on Architecture-Centric Evolution (ACE'06)*, Nantes, France, *3-7 July* 2006

[6] R. Robbes and M. Lanza. "Versioning systems for evolution research". *In Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, IEEE Computer Society, 2005, pp.155–164

[7] M. Lanza, R. Robbes, "A Change-based Approach to Software Evolution", *In Proceedings of ENTCS'07*, 2007, Volume 166, ISSN: 1571-0661, pp: 93-109,

[8] Lopes C. V. and Hursch W. L., "Separation of Concerns", College of Computer Science, Northeastern University, Boston, February 1995

[9] Karl Trygve Kalleberg, Eelco Visser. "Combining Aspect-Oriented and Strategic Programming". Electronic Notes in Theoretical Computer Science. 2005

[10] Thorsten Schultzke Claudia Ermel. "AGG Environnement: A Short Manual". http://tfs.cs.tuberlin.de/agg/ShortManual.ps, short manual edition.User Manual.

[11] Junichi Suzuki and Yoshikazu Yamamoto. "Managing the software design documents with xml ". In *Proceedings of the 16th annual international conference on Computer documentation*, ACM Press, 1998, pp. 127-136

[12] James Clark. "*XSL Transformations (XSLT)Version 1.0*". http://www.w3.org/TR/xslt, w3c recommendation16 november 1999 edition, Nov 1999. Recommandation.

[13] Andreas Winter, Bernt Kullbach, and Volker Riediger. "An overview of the gxl graph exchange language". In

Revised Lectures on Software Visualization, International Seminar, Springer-Verlag, 2002, pp. 324-336
[14] MELO JUNIOR, L. S., MENDONÇA, N. C. "AspectJML: A Markup Language for AspectJ". *In: Proc. of the 2nd Brazilian Workshop on Aspect Oriented Software Development (WASP'05)*, 2005, Uberlândia, MG, Brazil. In Portugues.