وزارة التعليم العالي و البحث العلمي

Université Badji Mokhtar –Annaba-

Badji Mokhtar –Annaba- University

جامعة باجي مختار- عنابة-

Année**: 2014**

Faculté des sciences de l'ingéniorat
Département d'informatique

T H È S E

Pour obtenir le diplôme de
Docteur 3éme cycle

Gestion des Préoccupations dans le Contexte de Maintenance

Filière : Informatique
Spécialité : Ingénierie des Logiciels Complexes

Préparée par

Hanene Cherait

Jury :

Président: Mr Salim Ghanemi	Pr.	Université Badji Mokhtar Annaba
Directeur de thèse : Mme Nora Bounour-Zeghida	MCA.	Université Badji Mokhtar Annaba
Examinateur : Mr Djamel Meslati	Pr.	Université Badji Mokhtar Annaba
Examinateur : Mr Abdelkrim Amirat	MCA.	Université Med Chérif Messaidia-Souk Ahras
Examinatrice : Melle Fadila Atil	MCA.	Université Badji Mokhtar Annaba

REMERCIEMENTS

Tout d'abord, je remercie notre clément Dieu qui m'a donné la puissance, le courage et la détermination nécessaire pour finaliser ce travail de thèse.

Je remercie très particulièrement mon directeur de thèse Dr Nora Bounour qui n'a réservé aucun effort pour m'aider et me mettre toujours sur le bon chemin. Ses critiques constructives, et la confiance qu'elle me donne pendant toutes les années de cette formation ont conduit à l'achèvement de ce travail. Qu'elle trouve ici l'expression de mon très grand respect et du plaisir que j'ai à travailler avec elle.

Je tiens également à remercier tous les enseignants qui ont participé à ma formation depuis ma première année universitaire, et particulièrement l'équipe de formation ILC pour leur volonté de donner.

Je souhaite exprimer toute ma reconnaissance aux membres de jury: Pr Salim Ghanemi, Pr Djamel Meslati, Dr Abdelkrim Amirat, et Dr Fadila Atil pour l'honneur qu'ils me font en acceptant d'évaluer mon travail.

Enfin, je tiens à remercier les membres de ma famille qui m'ont couvert par le soutien et les encouragements, surtout ma mère qui était vraiment à côté de moi durant les moments de stress.

Vraiment merci à tous.

Hanene Cherait

Décembre 2014

Je dédie ce travail à ma très chère mère.

“Be not afraid of going slowly; be afraid only of standing still.”

Chinese proverb

Résumé

La décennie passée a vu l'utilisation augmentée des techniques du développement du logiciel Orienté Aspect (OA) comme un moyen pour modulariser les préoccupations transversales dans les systèmes logiciels. Les grands projets industriels qui utilisent le paradigme OA mettent en valeur des applications notables de ce paradigme. Vu que ces systèmes sont devenus de plus en plus populaires, ils seront les logiciels hérités du futur. Un des défis principaux de ces systèmes logiciels réside dans leur évolution. Notre dissertation se focalise sur ce problème, où nous proposons un Framework d'évolution pour traiter l'évolution des programmes OA. Ce Framework d'évolution permet d'une part de modéliser et de valider l'évolution du logiciel OA. D'une autre part, il permet de garder l'historique de cette évolution dans un dépôt formel supportant les propres caractéristiques du paradigme OA. Ce dépôt peut être analysé par la suite pour comprendre l'évolution du logiciel OA, et prédire son futur développement. Le formalisme de la transformation algébrique de graphes est utilisé comme un support formel et rigoureux de notre Framework d'évolution proposé.

Mots-clés : Programmation orientée aspect, évolution du logiciel, systèmes de contrôle de version, transformation algébrique de graphes, détection des patrons de changement.

Abstract

The past decade has seen the increased use of Aspect Oriented (AO) software development techniques as a means to modularize crosscutting concerns in software systems. The major industrial projects using AO paradigm highlights its notable applications. Since these systems are becomes more and more popular, they will be the legacy software of the future. One of the main challenges of these software systems lies in their evolution. Our dissertation focuses on this problem, where we propose an evolution Framework to treat the evolution of AO programs. This evolution Framework permits in one side to model and validate the AO software evolution. In the other side, it permits to keep the history of this evolution in a formal repository, supporting the own features of the AO paradigm. This repository can be analyzed further to understand the AO software evolution, and predict its future development. The algebraic graph transformation formalism is used as a formal and rigorous support of our proposed evolution Framework.

Keywords: Aspect oriented programming, software evolution, version control systems, algebraic graph transformation, change pattern detection.

ملخص

شهد العقد الماضي زيادة استخدام تقنيات تطوير البرمجيات جانبية المنحى كوسيلة لتغليف الاهتمامات الشاملة في أنظمة البرمجيات. المشاريع الصناعية الكبرى باستخدام النموذج جانبي المنحى تبرز تطبيقاتها البارزة. بما أن هذه البرمجيات أصبحت أكثر فأكثر شعبية، فأنها ستكون البرامج الموروثة في المستقبل. أحد التحديات الرئيسية لهذه النظم للبرمجيات يكمن في تطورها. أطروحتنا تركز على هذه المشكلة، حيث أننا نقترح إطارا للتطور لعلاج تطور البرامج جانبية المنحى. يسمح إطار التطور هذا بنمدجة و التحقق من صحة تطور البرمجيات جانبية المنحى من جهة. من جهة أخرى يسمح بالحفاظ على تاريخ هذا التطور في مستودع قطعي يدعم الميزات الخاصة للنمط جانبي المنحى. يمكن تحليل هذا المستودع لفهم تطور البرمجيات، وتوقع تطورها المستقبلي. استعمل التحويل الجبري للمخططات كدعم قطعي ودقيق لإطار التطور المقترح.

كلمات مرشدة: البرمجة جانبية المنحى، تطور البرمجيات، أنظمة التحكم في الإصدار، التحويل الجبري للمخططات، كشف أنماط التغيير.

TABLE DES MATIERES

TABLE DES ILLUSTRATIONS

TABLE DES PROGRAMMES

LISTE DES TABLEAUX

ABREVIATIONS

AC	Attribute Condition
AGG	Attributed Graph Grammar
AJDT	AspectJ Developpement Tools
AST	Abstract Syntax Tree
CVS	Concurrent Versioning System
GGX	Graph Grammar eXchange
GTS	Graph Transformation System
GXL	Graph Exchange Language
LHS	Left Hand Side
NAC	Negative Application Condition
POA	Programmation Orientée Aspect
RHS	Right Hand Side
SCM	Software Configuration Management
SCV	Système de Contrôle de Version
UML	Unified Modeling Language
XML	eXtended Markup Language
XSLT	XML Stylesheet Language Transformation

CHAPITRE 1

Introduction Générale

*"A successful and used software product must be subject to **evolution**;*
else it becomes progressively less satisfactory."
M.M. Lehman, 1976 [LEH76]

Ce chapitre présente le contexte de notre travail qui s'inscrit dans l'évolution des systèmes logiciels, et plus spécifiquement l'évolution des systèmes Orientés Aspect (OA). Nous expliquons notre problématique et nos motivations en montrant que le paradigme OA manque encore de mécanismes adéquats pour gérer et entreposer l'évolution. Nous présentons les problèmes associés à l'évolution des logiciels OA. Ensuite, nous donnons les objectifs de notre travail et nos contributions principales. Enfin, nous expliquons le plan général de notre thèse.

1.1 Contexte de Recherche

De nos jours le logiciel est touché par des changements fréquents pour qu'il puisse réagir avec ces nouveaux besoins. Le processus d'extension et d'adaptation continue est connu sous le nom «l'évolution du logiciel». Le problème principal des systèmes logiciels est que le logiciel est soumis de façon continue à l'évolution ou au changement. L'évolution du logiciel est menée par plusieurs facteurs, y compris le changement des besoins, technologies, plates-formes et la maintenance corrective et perfective (changements pour supprimer des bogues et améliorer des fonctionnalités). La compréhension, le changement, la correction, et l'adaptation des systèmes logiciels sont des activités essentielles pendant le développement du logiciel. Donc, la recherche sur le génie logiciel doit fournir des méthodes et des outils pour supporter ces différentes tâches d'évolution du logiciel.

Par conséquent, l'évolution des systèmes logiciels est devenue une matière vitale dans l'industrie du logiciel. Le terme évolution du logiciel a paru premièrement dans la littérature du génie logiciel en 1970s par une étude menée par Lehman et al. [LEH76]. Dans cette étude les auteurs ont mesuré la complexité, la taille, le coût et la maintenance de 20 versions du système d'exploitation OS/360, basé sur son code source. Toutes les 20 versions de ce système logiciel ont montré une tendance croissante dans toutes les mesures. Cette étude a montré que l'évolution des systèmes logiciels est une opération très importante et très coûteuse [ERL00].

Donc, puisque le développement de logiciel doit traiter de plus en plus la compréhension et la modification des systèmes hérités *(legacy systems*), qui ont évolué avec le temps, au lieu de reproduire de nouveaux systèmes logiciels à partir de zéro. La plupart des projets logiciels essaient de garder l'infrastructure et la modifier pour satisfaire de nouveaux besoins. En résultat, un montant énorme du code doit être maintenu et actualisé chaque année. Les études sur l'effort de développement de logiciel prouvent que la portion dépensée pour l'évolution du logiciel a augmenté dramatiquement de 49% en 1977 [LIE96] à plus que 90% en 1995 [AKS92, REE95].

Une étude industrielle organisée par Grady Booch en 2005 estime le nombre total de lignes du code dans la maintenance d'être autour de 800 milliard [BOO04]. Tel que, 30 milliard de lignes du code sont nouvelles ou doivent être modifiés chaque année

par approximativement 15 millions de programmeurs. Cela exige un montant énorme de ressources. Les études industrielles estiment les coûts de maintenance pour être autour de 80-90% [ERL00] des coûts du logiciel totaux, et le personnel de maintenance 60-80% [COL94] du personnel du projet total. Des études sur le coût de compréhension du logiciel, tel que ceux organisées par Standish [STA84] et Corbi [COR89], montrent que cette activité prend la moitié de l'effort total du développement.

Par conséquent, dans les dernières années un montant substantiel de recherche a été fait, qui adresse plusieurs aspects du problème de l'évolution du logiciel. La recherche sur l'évolution du logiciel s'est concentrée sur deux axes principaux pour faciliter le processus de l'évolution du logiciel, et réduire le coût/effort dépensé dans cette évolution: la modélisation de l'évolution et la prédiction du changement.

La modélisation de l'évolution: La recherche autour de cet axe a construit des processus et des structures de logiciel qui peuvent faciliter et valider l'évolution du logiciel; nous les appelons modèles de l'évolution. Cette ligne de recherche introduit des méthodes et des outils qui automatisent l'implémentation de changements. Ils essaient de traiter le problème de développeurs qui introduisent des erreurs au logiciel pendant l'implémentation du changement en élevant le niveau d'abstraction. Généralement, ces modèles transforment le logiciel à un état évolué en utilisant des transformations prédéfinies. Le développeur décrit comment le changement va être implémenté à travers ces transformations, et les outils fournis avec ces modèles font les changements au logiciel. Beaucoup de techniques et plus spécifiquement des modèles ont été proposés pour gérer l'évolution du logiciel dans une direction fiable et organisée le long de son cycle de vie [CHE10].

La prédiction du changement: Le principe de la prédiction du changement est de fournir des prédictions au sujet des futurs changements. Généralement, les chercheurs utilisent l'historique de l'évolution du logiciel pour fournir ces prédictions. Les systèmes de contrôle de version comme CVS [GRU86] sont des outils largement adoptés pour suivre les changements faits au logiciel. Les dépôts de ces systèmes tiennent l'historique entier de l'évolution du logiciel, et l'analyse de ces dépôts dérive des métadonnées qui peuvent être utilisées pour prédire de futurs changements basés sur l'historique d'évolution.

Ces deux axes sont largement considérés et traités comme deux processus indépendants, mais ils convergent au même but "l'évolution du logiciel", et se sont basés sur le même concept "le changement". Donc, la modélisation de l'évolution et la prédiction du changement sont complémentaires. Par conséquent, pour bien supporter le phénomène d'évolution nous suggérons de les intégrer.

1.2 Problématique et Motivations

La Programmation Orienté Aspect (POA) [KIC97] est une technique utilisée pour la modularisation des préoccupations transversales. Ces dernières sont implémentées dans une seule unité appelée "Aspect". Malgré les différents bénéfices de la POA, les programmes OA font face à beaucoup de défis durant leur évolution. La POA permet de modulariser les préoccupations transversales. Par ailleurs, elle produit de nouvelles dépendances entre eux; ce qui restreint l'évolutivité du système logiciel.

Une caractéristique importante de la POA est qu'elle donne naissance à un grand nombre d'unités supplémentaires de système (d'une grande à une fine granularité) prêt à être composé à l'application finale. Avec ce nombre croissant d'unités de système, les dépendances entre elles deviennent vastes et enchevêtrées (les relations transversales sont difficiles à représenter efficacement). Les aspects ne sont pas invoqués explicitement, mais ils sont invoqués implicitement [XUR04]. Ainsi, les changements introduits avec la POA ne sont pas directement visibles dans le code source du système de base, rendant alors la compréhension du programme plus difficile. Les aspects sont entreposés habituellement dans des fichiers séparés; mais les effets de ce code peuvent influencer le système entier [VOL02]. Pour résumer, la modélisation de l'évolution du programme OA implique des relations plus complexes que celles des programmes traditionnels (orienté objet, procédural...etc.).

D'un autre côté, certaines dépendances intéressantes ne peuvent pas être détectées directement à partir du code source OA. De telles dépendances *logiques* peuvent être extraites en analysant l'historique de l'évolution d'un système OA. Malheureusement, l'information évolutionnaire contenue dans les systèmes de version actuels pour le logiciel OA est incomplète et de qualité basse, limiter d'où le domaine d'analyse de l'évolution du logiciel OA. Le paradigme OA est caractérisé par la quantification et l'inconscience (*obliviousness*) [FIL01]: les aspects devraient être une partie de places

multiples dans un programme, et le code du système devrait être inconscient de tous aspects respectivement. Aucun des outils de contrôle de version actuels ne traite les besoins de contrôle et d'entreposage de l'évolution du logiciel OA, ils ne fonctionnent pas bien avec l'inconscience et la quantification trouvées dans le code OA. Par conséquent, leurs dépôts fournissent une information incomplète pour l'analyse de l'évolution du logiciel OA.

Pour bien traiter l'évolution du logiciel OA en prenant en compte les problèmes précités, nous proposons un Framework d'évolution. Ce dernier permet de modéliser et de valider l'évolution du logiciel OA, et en parallèle il permet de garder l'historique de cette évolution dans un dépôt spécifique au logiciel OA. Ce dépôt est consacré pour manier les propres caractéristiques du paradigme OA. L'analyse de ce dépôt riche peut être plus exacte et effective pour comprendre l'évolution du logiciel OA, prédire des futurs changements, identifier des fautes potentielles, détecter de nouvelles préoccupations transversales…etc.

Nous avons dédié notre dissertation pour répondre à la question suivante: ***Comment modéliser l'évolution du logiciel orienté aspect, et comment entreposer en même temps cette évolution d'une manière correcte pour prédire efficacement son futur changement?***

1.3 Objectifs et Contributions

Afin de pallier les problèmes de l'évolution des logiciels OA, nous proposons un Framework d'évolution. Ce dernier permet de modéliser et de valider l'évolution du logiciel OA, et en parallèle il permet de garder l'historique de cette évolution dans un dépôt spécifique au logiciel OA. Donc, nous intégrons la modélisation de l'évolution et l'entreposage de l'historique de cette évolution dans un même Framework. Cette intégration permet de traiter le changement du logiciel comme une entité de première classe. Par contraste avec le principe basée-fichier des systèmes de version classiques, nous nous basons sur le principe basé-changement pour présenter correctement le phénomène complet de l'évolution du logiciel OA.

Nous suggérons l'utilisation de la réécriture algébrique de graphes [BAR02] comme un support pour notre Framework d'évolution proposé. La théorie de la transformation de graphes a été développée pendant les trois dernières décennies comme un ensemble

de techniques et d'outils pour la modélisation formelle et la programmation visuelle de haut niveau. Elle autorise la représentation de transformation complexe dans une manière visuelle compacte. De plus, la théorie de la transformation de graphes fournit une fondation formelle pour l'analyse et l'application automatique et interactive des transformations de modèles.

La notation graphique fournit une approche intuitive et flexible pour décrire l'information structurelle. Comparée avec d'autres méthodes formelles, la transformation de graphes est convenable pour décrire l'évolution du logiciel. Elle est bien fondée théoriquement et beaucoup d'environnements et d'outils maturés [BUR04, SCH95, ZHA01] sont disponibles pour supporter sa conception et son implémentation. En outre, depuis que le code source OA peut être représenté comme un graphe, un processus de l'évolution peut être simulé avec l'application des règles de transformation sur ce graphe.

Dans notre proposition, le code source OA est modélisé dans un format abstrait et formel comme un graphe coloré attribué, où les différentes dépendances dans le système logiciel sont bien définies. Donc, les requêtes du changement seront présentées comme des règles de réécriture sur le graphe coloré. En plus, nous proposons un dépôt basé-règle où chaque règle de réécriture appliquée est entreposée *directement* dans ce dépôt. Dans ce dernier, chaque version du logiciel OA consiste en l'ensemble de séquences des règles de réécriture. Chaque séquence de règles de réécriture présente une requête spécifique du changement.

Nous proposons une approche d'analyse du dépôt proposé. L'approche nous offre l'extraction des patrons de changement durant l'évolution des programmes OA. Donc, le dépôt basé-règle est analysé pour détecter des patrons de règle en utilisant l'algorithme Apriori [AGR94]. Vu que ces règles sont la formulation des changements du code source.

Par conséquent, les contributions principales de notre travail peuvent être résumées comme suit:

— La modélisation du code source OA: Dans notre proposition, le code source OA est modélisé comme un graphe coloré attribué. Notre modèle représente un système OA en termes de ses composants et les différentes relations entre eux.

— La modélisation du changement: Définir des règles pour modéliser les changements dans un système logiciel OA. Décrire quel genre de changements peut être fait sur le modèle. Ces changements sont formalisés comme des règles de réécriture sur le graphe coloré attribué.

— un dépôt basé-règle de réécriture du logiciel OA pour entreposer une quantité maximale d'information au sujet de son évolution, en entreposant chaque règle de réécriture quand elle est appliquée i.e. traiter le changement comme une entité de première classe.

— Une approche pour l'analyse du dépôt basé-règle de réécriture: le dépôt proposé est analysé pour extraire des métadonnées. Ces derniers consistent en les patrons de changements, et les relations logiques entre les entités du logiciel OA.

1.4 Plan de la Thèse

Outre le chapitre Introduction Générale, ce mémoire de thèse est structuré dans les sept chapitres suivants:

Chapitre 2: Il est consacré à l'évolution des logiciels OA. Nous allons décrire dans ce chapitre le paradigme de la programmation OA, son principe, ses bénéfices,…etc. Ainsi que le langage AspectJ, qui est le langage le plus populaire parmi les langages OA. Ensuite nous nous intéressons aux challenges que puisse poser l'évolution du logiciel OA. Et nous passons brièvement sur les travaux existants qui traitent les différentes tâches de l'évolution du logiciel OA.

Chapitre 3: Il présente un état de l'art examinant l'analyse de l'historique d'évolution du logiciel de façon générale. Ensuite nous nous concentrons sur les limites des dépôts actuels dans l'entreposage de l'historique de l'évolution des logiciels OA.

Chapitre 4:dans ce chapitre nous discutons la puissance et l'intuitivité que fournisse le formalisme de la réécriture algébrique de graphes (transformation de graphes) dans le but d'assister et de valider l'évolution du logiciel. Ensuite, et en basant sur ce formalisme nous présentons la description globale de notre proposition, qui est le Framework d'évolution pour les logiciels OA.

Chapitre 5: Nous nous concentrons dans ce chapitre sur la modélisation de l'évolution du logiciel OA comme un système de réécriture de graphes. Nous donnons ici, les détails du modèle proposé ainsi que son implémentation. Ensuite, nous fournissons une évaluation empirique pour prouver l'efficacité de notre proposition.

Chapitre 6: Il présente notre dépôt basé-règle de réécriture, ainsi que l'approche de détection des patrons de changement et de relations logiques pour les programmes OA.

Chapitre 7: Il synthétise les principales contributions de notre travail. De plus, nous comparons nos propositions avec les travaux existants dans cette zone de recherche. Et nous discutons les éventuelles perspectives futures.

CHAPITRE 2

L'Evolution des Logiciels Orientés Aspects: Challenges et Approches

"Given that AOP has set out to modularize crosscutting concerns (its methodological claim), but by its very nature (its mechanics) breaks modularity…." Friedrich Steimann, 2006 [STE06]

Depuis 1997, la POA provoque l'engouement de la communauté scientifique s'intéressant au génie logiciel. Cette technique offre des nouvelles perspectives à la séparation avancée des préoccupations. Le but principal de ce nouveau paradigme est d'améliorer la séparation des préoccupations lors du développement de logiciel. On cherche à isoler chaque problématique pour pouvoir la coder séparément, et définir les règles d'intégration afin de les combiner pour pouvoir former un système final.

La séparation des préoccupations est bonne théoriquement. Quand nous utilisons la POA, nous obtenons beaucoup d'avantages, mais nous devons aussi traiter leurs problèmes. Si on modifie un logiciel OA pour le faire évoluer ou si on transforme un système existant en un système OA, nous serons confrontés aux problèmes évolutionnaires. Nous allons découvrir ce paradigme de programmation dans ce chapitre, son principe, ses bénéfices…etc. Ainsi que le langage AspectJ, qui est le langage le plus populaire parmi les langages OA. D'un autre côté, nous discuterons les différents challenges de l'évolution du logiciel OA, et nous présentons brièvement les travaux existants dans cette zone de recherche.

2.1 La Séparation des Préoccupations

La séparation des préoccupations (*Separation of Concerns*) [LOP95, PAR72] est un principe commun, largement utilisé dans le génie logiciel. Elle est considérée comme l'une des approches les plus prometteuses du génie logiciel. Elle suggère qu'un problème complexe doit être divisé en une série de plus petits problèmes qui sont moins complexes et plus facile à comprendre. Ces plus petits problèmes peuvent être résolus séparément, et finalement ils peuvent être intégrés ensemble pour résoudre le grand problème. Le développement du logiciel avec cette manière offre une plus grande compréhensibilité, maintenabilité, adaptabilité, et réutilisabilité aux programmes. Puisque les problèmes sont réduits en des unités dont la taille est perceptible par les esprits humains, et cela peut être généralisé pour adapter à plusieurs besoins.

Au niveau du logiciel, chaque unité représente ou implémente une préoccupation du système. La définition communément utilisée d'une préoccupation à partir de l'IEEE 1471: *Une préoccupation est un intérêt qui a rapport au développement du système, son opération ou tout autre aspect qui est critique ou autrement important à un ou plusieurs utilisateurs.*

Depuis une trentaine d'années, l'approche orientée objet a procuré des bénéfices au niveau de la séparation des préoccupations. En effet, elle permet de modulariser de larges systèmes d'une manière intuitive, car elle offre un modèle de programmation riche très proche du monde réel. Elle représente des objets individuels dans un modèle du domaine du problème. Aujourd'hui, l'orienté objet bénéficie d'un excellent background notamment grâce à la popularisation d'UML. En plus, une multitude de langages de programmation (Java, C++, Smalltalk, ...) ont adopté ce paradigme de manière native. Cependant, l'approche orientée objet montre ses limites et échoue face à la modularisation des préoccupations transversales (*crosscutting concerns*) au système. Parmi les préoccupations transversales les plus courantes, on trouve la sécurité, la gestion transactionnelle de la persistance, la synchronisation, le logging…etc. Le domaine de recherche qui est survenu pour traiter ce problème était connu par la séparation ***avancée*** des préoccupations.

2.2 La Séparation Avancée des Préoccupations

Cet axe de recherche prend en charge les préoccupations transversales du système. Ces dernières sont les fonctionnalités dites non métiers. Un développeur est souvent confronté à ce genre de fonctionnalités lorsqu'il développe une application de grande taille. Dans ce cas, même si on applique une bonne modularisation verticale des préoccupations métiers (avec un langage orienté objet), on aura toujours un problème de préoccupations horizontales qui transverses l'ensemble des modules métiers. Ces préoccupations transversales ne peuvent pas être modularisées dans la décomposition sélectionnée.

Il existe en fait deux principaux symptômes liés aux préoccupations transversales: (1) ces préoccupations ont la particularité d'un côté, d'être *dispersées* à travers plusieurs modules, (2) et d'un autre côté, d'être *enchevêtrées* avec les modules métiers du système.

Enchevêtrement du code (*code tangling*): L'enchevêtrement du code est provoqué quand un module est implémenté pour traiter plusieurs préoccupations en même temps. Un développeur a souvent affaire, pendant qu'il développe un module, à des préoccupations telles que la logique métier, la gestion transactionnelle de la persistance, le logging, la sécurité…etc. Cela conduit à la présence simultanée d'éléments issus de chaque préoccupation, et il en résulte en un enchevêtrement du code (Figure 2.1).

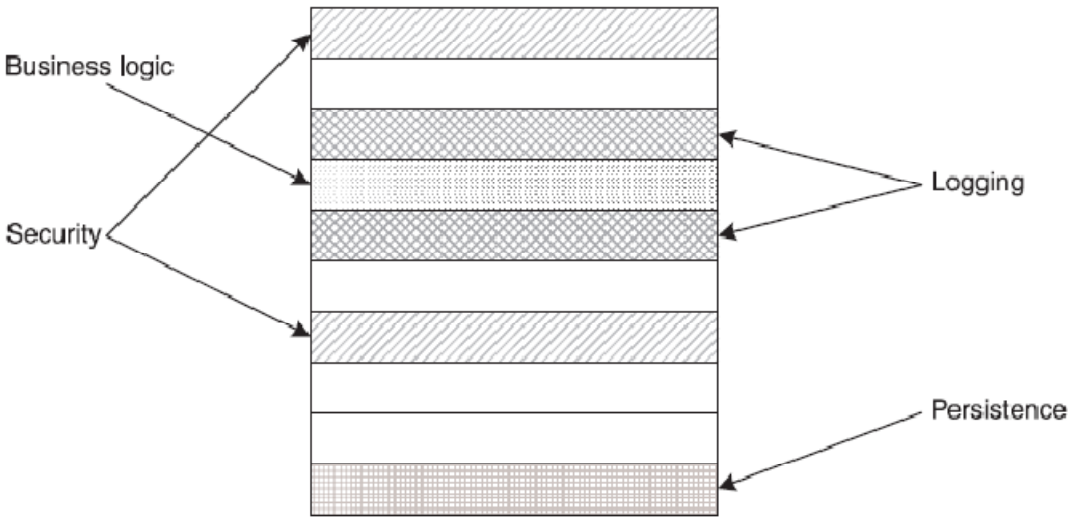


Figure 2.1. Enchevêtrement du code [LAD10].

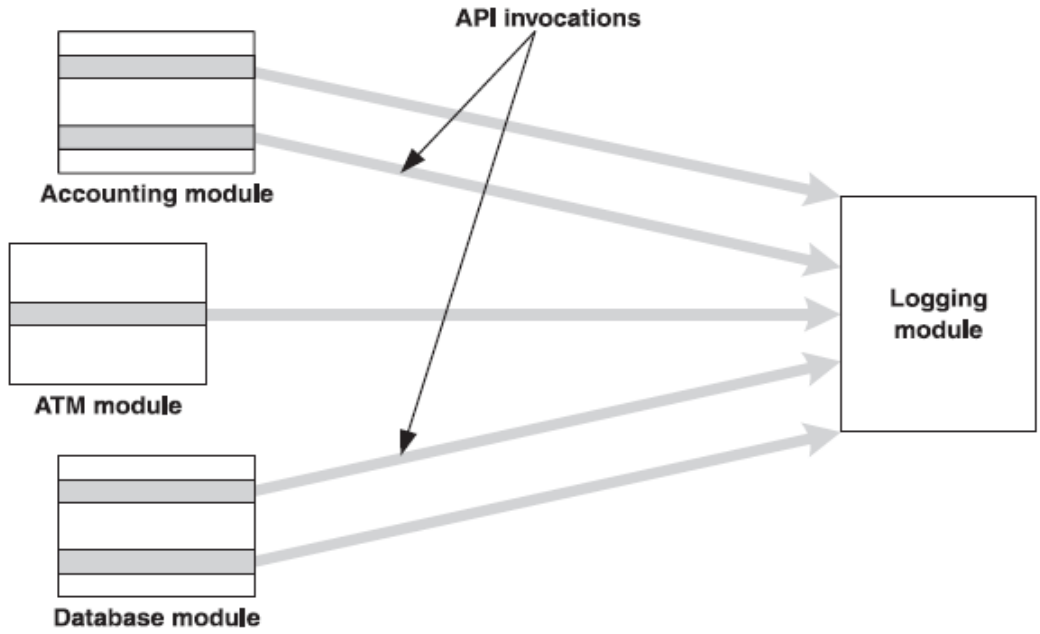


Figure 2.2. Éparpillement du code de la préoccupation logging [LAD10].

Éparpillement du code (*code scattering*): L'éparpillement du code survient quand une préoccupation est implémentée dans plusieurs modules. Les préoccupations transversales sont, par définition, dispersées à travers plusieurs modules. Par exemple, dans un système utilisant une base de données, le logging est une préoccupation implémentée dans tous les modules qui accèdent à la base de données (Figure 2.2).

Ces deux phénomènes dégradent considérablement le maintien, la compréhension et l'évolutivité du code. Plusieurs approches ont été proposées pour traiter la séparation avancée des préoccupations [OSS99], tel que la programmation orientée aspect (Aspect Oriented Programming) [KIC97], la programmation flexible (Adaptive Programming) [LIE96], la composition des filtres (Composition Filters) [AKS92], la modélisation de rôle (Role Modeling) [REE95], et la programmation orienté sujet (Subject Oriented Programming) [HAR93]. Nous nous intéressons dans cette thèse à la programmation orientée aspect (POA) qui est une solution efficace de la séparation avancée des préoccupations.

2.3 La Programmation Orientée Aspect

La POA permet de modulariser le code en fournissant des mécanismes explicites pour capturer la structure des préoccupations transversales dans les systèmes logiciels, telle que le traitement des exceptions, la synchronisation, les optimisations de la performance, et le partage de ressource. Cela est habituellement difficile à exprimer proprement dans le code source par les techniques de programmation existantes (ex. orientée objet). Les langages OA peuvent contrôler un tel code enchevêtré et éparpillé pour rendre les préoccupations transversales plus apparentes.

En effet, l'orienté aspect procure une solution élégante aux problèmes *d'enchevêtrement* et *d'éparpillement* du code. Cette technique d'ingénierie du logiciel offre une nouvelle dimension pour la modularisation avec la notion d'«Aspect». Par conséquent, parallèlement aux classes qui sont un support idéal pour modulariser les préoccupations métiers du système, les aspects sont un support pour capturer les préoccupations transversales, où chaque aspect se concentre sur une fonctionnalité transversale spécifique. Par exemple, une préoccupation de logging peut être séparée d'un programme de base en spécifiant tous les points du programme qui doivent être enregistrés. Si l'architecture de logging devrait être changée, par exemple en utilisant un Framework plus sophistiqué à la place des appels println, tous les changements sont localisés dans l'aspect logging.

Dans une démarche OA, les préoccupations transversales peuvent évoluer indépendamment des préoccupations métier et vice-versa. Et afin que l'application finale prennent en compte toutes les préoccupations, le système passe par une étape dite de tissage d'aspects (Aspect weaving). Durant cette étape (Figure 2.3), les préoccupations transversales encapsulées dans les aspects vont être tissées ou intégrées dans les préoccupations métiers. Par conséquent, la POA aide pour créer des applications qui sont plus faciles à concevoir, à implémenter et à maintenir.

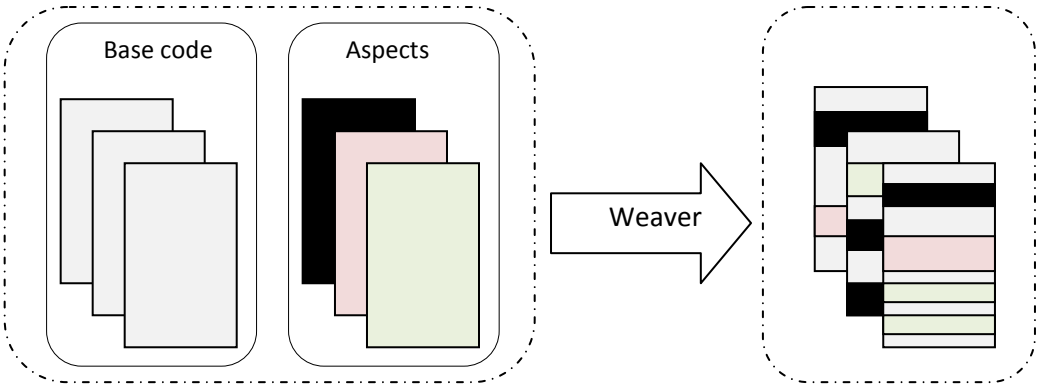


Figure 2.3. La programmation orienté aspect.

2.3.1 Les Bénéfices de la POA

Les différents bénéfices de la POA peuvent être résumés dans les points suivants:

Modularisation plus élevée: La POA fournit un mécanisme pour adresser chaque préoccupation séparément avec l'accouplement minimal. Elle permet à un module de prendre seulement la responsabilité de sa préoccupation. Donc, un module n'est plus responsable d'autres préoccupations qui se recoupent. Par exemple, un aspect qui implémente le logging n'implémente pas les fonctionnalités de sécurité. Cela, résulte en des affectations plus propres de responsabilités, réduire le fouillis du code et évite la reproduction. Il améliore aussi la traçabilité de besoins à leur implémentation, et vice versa. Ceci rend l'implémentation bien modulée même en présence des préoccupations qui se recoupent. Ceci améliore par conséquent le système où le code est beaucoup moins redondant.

Implémentation plus améliorée**:** La réduction du code enchevêtré le rend plus simple à tester, détecter les problèmes potentiels, et exécuter des révisions du code. Examiner le code d'un module qui implémente seulement une préoccupation exige la participation juste d'un expert dans la fonctionnalité implémentée par ce module. Un tel processus simplifié produit un code d'une qualité haute. La réduction du code éparpillé évite la modification de plusieurs modules pour implémenter une préoccupation transversale. Donc, la POA réduit le coût de l'implémentation d'une préoccupation transversale. D'un autre côté,

l'encapsulation des préoccupations transversales, aide le développeur à se concentrer sur les préoccupations métiers.

***Conception plus simplifiée*:** Avec la POA, si nous avons besoin d'un type particulier de fonctionnalités plus tard, nous pouvons l'implémenter sans devoir faire des changements larges sur le système. Donc, la POA aide à ajouter de façon incrémentale des caractéristiques à travers l'introduction d'aspects, souvent sans modifier le reste du code.

***Code plus réutilisable*:** le concept clé pour une meilleure réutilisation du code est la bonne modularisation de l'implémentation. Si un module implémente des préoccupations multiples, les autres systèmes qui ont besoin de fonctionnalité semblable ne peuvent pas utiliser ce module. Avec la POA, puisque chaque préoccupation transversale est implémentée comme un aspect, les modules de base sont inconscients des fonctionnalités transversales. En changeant simplement les spécifications de tissage au lieu des modules multiples de noyau, ou en modifiant les aspects, nous pouvons changer la configuration du système. Par exemple, une couche de service peut être sécurisée dans un projet avec un plan de sécurité, dans un autre projet avec un autre plan, ou dans un autre projet elle reste sans protection en incluant ou excluant les aspects appropriés. Sans la POA, une couche du service attachée avec une implémentation spécifique de sécurité ne peut pas être réutilisée dans un autre projet [LAD10]. De plus, les études montrent que la surcharge introduite par les approches OA est relativement faible. Enfin, les implémentations OA ont des niveaux d'adaptabilité et de réutilisabilité plus élevés que les implémentations uniquement objet.

Evolution plus facile: Avec une solution basée sur la POA, nous serons capable de combiner plusieurs modules, appelés aspects, pour créer un système qui fournit juste les services que nous avons besoin, ni plus ni moins. Nous pouvons ensuite modifier chaque aspect séparément, ou même le remplacer sans affecter de façon défavorable les autres parties du système. L'ajout d'une nouvelle fonctionnalité est une question d'inclure un nouvel aspect séparé et n'exige aucun changement dans les modules de base. Malgré que ceci soit juste théoriquement, il n'est pas toujours le cas pratiquement. La POA présente différents problèmes

au niveau de l'évolution, nous présentons ces problèmes dans les prochaines sections.

Si on résume, la POA permet de résoudre les problèmes dus à l'enchevêtrement et l'éparpillement du code. Elle permet ainsi de modulariser l'implémentation des problématiques transversales, de créer des systèmes plus évolutifs, de réduire le coût et le temps de développement et d'assurer une meilleure réutilisation du code.

2.3.2 Les Différentes Applications de la POA

Les concepts OA ont été appliqués aux différentes étapes du génie logiciel comme la spécification des besoins et la conception [CHI05], la vérification et les approches formelles [DOU05, KAT05], aussi bien que de nouvelles plateformes et outils surviennent [BRI05]. Actuellement, une panoplie de langages et de plateformes OA existe: AspectJ [LAD10], AspectWerkz [VAS04], Josh [CHI04], JMangler [KNI04], JAC [PAW01], ou encore HyperJ [IBM01], AspectC++ [SPI02], JBoss [PAW05]…etc.

Le succès de l'OA atteint aussi les middlewares utilisés pour les grands systèmes distribués [COL03, COL04, LOU05], où les aspects sont utilisés pour abstraire les préoccupations transversales telles que la persistance, la communication transactionnelle, la sécurité, la qualité de service, ou la synchronisation. La POA a été appliquée sur les bases de données aussi [RAS04]...etc.

Aujourd'hui, grâce à des outils robustes et éprouvés, la POA intéresse de plus en plus de services informatiques de grands groupes industriels. Il existe une variété de Framework commerciaux qui inclut des caractéristiques OA. Le Tableau 2.1 présente les applications industrielles qui utilisent la POA.

Tableau 2.1. Les projets industriels qui utilisent la POA [RAS10].

Application	Description
IBM WebSphere application server	C'est un serveur d'application Java qui supporte Java Enterprise Edition (EE) et Web services. WebSphere est distribué dans plusieurs éditions qui supportent des caractéristiques différentes, l'AspectJ est utilisé pour isoler ces caractéristiques.
JBoss Application	C'est un serveur d'application Java libre, open source. Il supporte Java EE. Le noyau de JBoss AS utilise JBoss AOP pour déployer des services tels

Server (AS)	que la sécurité et la gestion de transaction.
Oracle TopLink	C'est un Framework de persistance (object-to-relational) Java qui est intégré avec le serveur d'application Spring (www.springsource.org). TopLink accomplit de hauts niveaux de transparence de la persistance en utilisant le Spring AOP.
Sun Microsystems	utilise AspectJ pour simplifier le développement d'application mobile pour la plate-forme Java Micro Edition (ME). Les aspects sont utilisés pour simplifier le développement d'applications mobiles pour le déploiement à différentes interfaces de la communauté du jeu du portable.
Siemens' Soarian	C'est un système informatique de santé qui supporte l'accès souple aux enregistrements médicaux des patients et la définition de flots du travail pour les organisations de santé. Soarian utilise AspectJ et fastAOP (http://sourceforge.filet/projets/fastaop) pour intégrer les fonctionnalités transversales dans un processus du développement agile.
Motorola's wi4	C'est un système d'infrastructure cellulaire qui fournit le support pour la norme WiMax de la bande large sans fil. le logiciel de contrôle wi4 est développé en utilisant WEAVR (une extension OA au standard UML 2.0) pour le débogage et le test.
ASML	Il concerne l'industrie semi-conducteur, utilise Mirjam, une extension OA à C, pour modulariser les préoccupations suivantes: tracing, profiling, et error-handling.
Glassbox	C'est un agent de localisation de pannes pour les applications Java qui automatiquement diagnostique les problèmes communs. L'inspecteur Glassbox utilise AspectJ pour diriger l'activité de la machine virtuelle Java.
MySQL	C'est un système de gestion de base de données relationnelle largement utilisé. La fonctionnalité de logging dans MySQL est implémentée en utilisant AspectJ.

Dans notre thèse, nous nous intéressons particulièrement au langage AspectJ qui est à notre connaissance l'expérimentation la plus aboutie des langages OA. Ce n'est pas un langage à part entière, mais une extension du langage Java qui permet à ce dernier de supporter la programmation OA.

2.4 Le Langage AspectJ

L'histoire d'AspectJ est étroitement liée à celle de la POA. En effet, Ce langage a été développé par la même équipe à l'origine de la POA. Un premier prototype d'AspectJ a été réalisé en 1998. Et depuis, plusieurs versions d'AspectJ ont vu le jour, et chacune apportait de nouvelles fonctionnalités et corrigeait les bogues de la précédente. La première version officielle d'AspectJ, désignée AspectJ 1.0, a été réalisée en novembre 2001, Durant cette année, la POA a été complètement reconnue par la communauté informatique mondiale. En décembre 2002, le

projet AspectJ a quitté XEROX PARC et a rejoint la communauté open-source Eclipse (éclipse.org). Et depuis, le plugin Eclipse *AspectJ Developpement Tools (AJDT)* est développé. Ce plugin intègre AspectJ et permet d'écrire, de compiler et d'exécuter des programmes OA dans l'environnement de développement Eclipse.

L'AspectJ est l'implémentation originale et reste la meilleure solution de la POA. Ce langage est un choix populaire pour plusieurs bonnes raisons. Une de ses forces est (et toujours a été) son approche pragmatique à la conception de langage. Au lieu de permettre au langage de s'embourber en théorie, les développeurs d'AspectJ ont commencé avec le support de base de la POA et ont ajouté seulement de nouvelles caractéristiques après que les gens dans le domaine ont discuté leur usage pratique largement. Le résultat était la création d'un langage simple qui était assez puissant pour résoudre des problèmes réels. AspectJ est utilisé dans les projets réels pour améliorer des plates-formes middleware, contrôler et améliorer la performance, ajouter la sécurité aux applications existantes, parmi d'autres. Tous ces projets ont vu des résultats impressionnants dans la réduction du montant de code et le temps exigé pour créer les produits. La section précédente montre qu'AspectJ est le langage le plus utilisé dans les grands systèmes industriels (Tableau 2.1).

Avec AspectJ, les programmeurs peuvent encapsuler les préoccupations qui entrecoupent la hiérarchie dominante de classe dans des unités séparées appelées «Aspects». Ces unités résument brièvement le code de telles préoccupations dans une seule place, et explicitement définit les points auxquels le code sera inséré dans la hiérarchie de classe dominante. Le pré-compilateur d'AspectJ génère ensuite un code Java ordinaire qui se rend compte des préoccupations selon les règles définies dans les unités. Ce processus est appelé le tissage (aspect weaving).

AspectJ ajoute les concepts de la POA (Figure 2.4) à Java, en créant un langage puissant qui facilite la modularisation des préoccupations en retenant les avantages de Java, tel que l'indépendance de plate-forme.

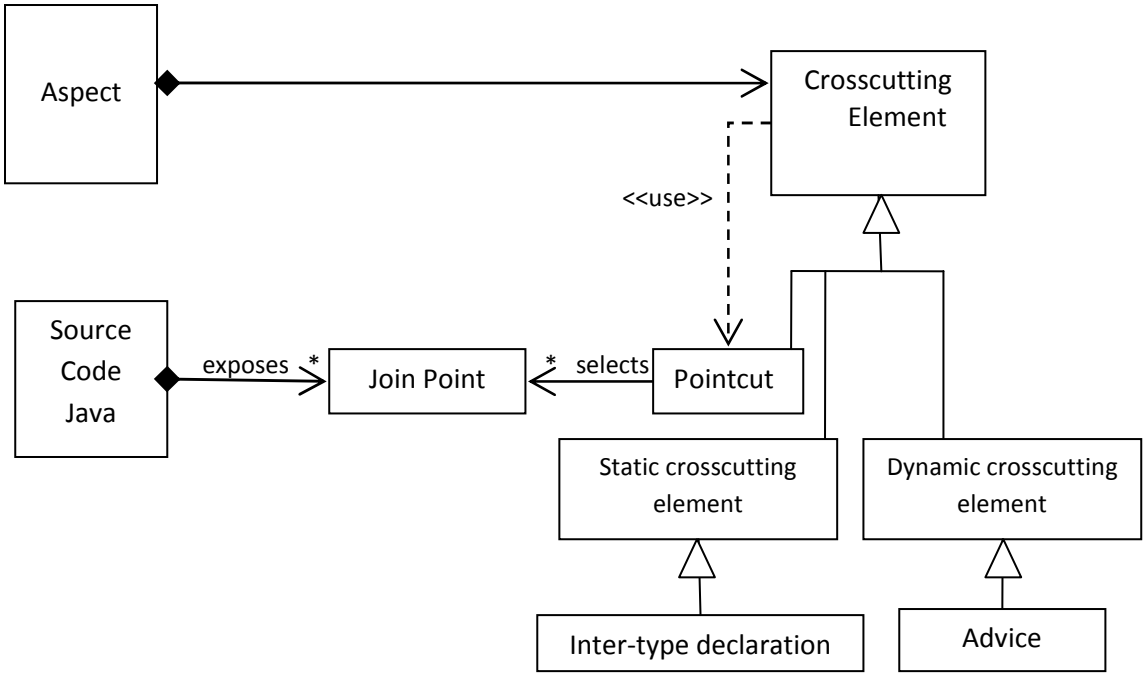


Figure 2.4. Le modèle AspectJ.

Nous pouvons ajouter de nouvelles fonctionnalités sans changer le code dans les modules de noyau, et ces modules sont inconscients de ce que nous avons fait [LAD10]. Nous pouvons classer les concepts transversaux dans le modèle AspectJ comme suit: les concepts transversaux dynamiques, et les concepts transversaux statiques. Ces concepts sont tous encapsulés dans une seule unité appelé Aspect.

2.4.1 Concepts Transversaux Dynamiques

a. Point de Jointure (Join-Point)

La notion de point de jointure est un concept fondamental dans la POA. Il correspond aux événements dans le flux de contrôle d'un programme (l'exécution d'un programme) [LAD10]. Par exemple, dans les langages orientés objet, ils peuvent faire référence au passage de messages ou la modification de variables. Dans le code du programme, il y a des instructions qui correspondent à ces événements. Par exemple, une instruction d'appel correspond à l'événement de passage de message; une instruction d'affectation correspond à la modification de variable…etc. Une instruction dans le code qui correspond à un point de jointure est appelée l'ombre du point de jointure (*shadow point)* dans la littérature OA.

Les points de jointure disposent d'un contexte qui leur est associé. Par exemple, un point de jointure d'appel à une méthode aura dans son contexte l'objet cible, ainsi que les éventuels arguments qui lui sont passés. Les points de jointure sont présents dans tous les systèmes, même ceux qui n'utilisent pas la POA; puisqu'ils sont des points pendant l'exécution d'un système. La POA identifie et catégorise simplement ces points. Bien que n'importe quel point d'exécution dans le programme puisse être défini comme un point de jointure, AspectJ limite les points de jointure à ceux qui peuvent être utilisés de manière systématique. Les points de jointure définis par AspectJ sont présentés dans le Tableau 2.2. Après avoir identifié les points de jointure utiles pour une fonctionnalité transversale, nous devons les sélectionner en utilisant les points de coupure (Pointcuts).

Tableau 2.2. Les types de point de jointure dans AspectJ.

Point de jointure	Description
Method call	quand une méthode est appelée
Method execution	quand le corps de la méthode est exécuté
Constructor call	quand un constructeur est appelé
Constructor execution	quand le corps du constructeur est exécuté
Static initializer execution	quand l'initialisation statique d'une classe est exécutée
Object pre-initialization	avant l'initialisation d'un objet
Object initialization	quand l'initialisation d'un objet est exécutée
Field reference	quand un attribut non-constant d'une classe est référencé
Field set	quand un attribut d'une classe est modifié
Handler execution	quand un traitement d'une exception est exécuté
Advice execution	quand le code d'un advice (consigne) est exécuté

b. Point de Coupure (Pointcut)

L'implémentation d'une préoccupation transversale exige la sélection d'un ensemble spécifique de points de jointure. Un point de coupure est un élément du programme qui choisit des points de jointure et expose des données à partir du contexte de l'exécution de ces points. Il sélectionne tous les points de jointure qui satisfaites les critères. Une expression du point de coupure est en fait un mécanisme de quantification sur les événements et/ou la syntaxe d'un programme [NAG06].

L'expression d'un point de coupure consiste en des points de coupure de base ou des points de coupure composés (un point de coupure peut utiliser un autre point de coupure pour former une sélection complexe). Un point de coupure composé

peut être développé à partir des points de coupures primitives et/ou composés avec l'aide des opérateurs de composition (opérateurs booléens). Pour traiter la variabilité dans les événements sélectionnés, l'expression d'un point de coupure peut être paramétrée avec les propriétés des points de jointure désignés. Par exemple, un point de coupure peut collectionner les arguments d'une méthode comme contexte. Le concept de point de jointure et de point de coupure ensemble forment un modèle de point de jointure du système OA (*Join point Model*). Les points de coupure de base fournis par le langage AspectJ sont présentés dans le Tableau 2.3.

Tableau 2.3. Les expressions des points de coupure.

Point de coupure	Description
call(*MethodPattern*)	sélectionne chaque point de jointure de type «appel de méthode» dont sa signature égale à *MethodPattern*.
execution(*MethodPattern*)	sélectionne chaque point de jointure de type «exécution de méthode» dont sa signature égale à *MethodPattern*.
get(*FieldPattern*)	sélectionne chaque point de jointure de type «référence de champ» dont sa signature égale à FieldPattern.
set(*FieldPattern*)	sélectionne chaque point de jointure de type «mis à jour de champ» dont la signature égale à FieldPattern.
call(*ConstructorPattern*)	sélectionne chaque point de jointure de type «appel du constructeur» dont la signature égale à ConstructorPattern.
execution(*ConstructorPattern*)	sélectionne chaque point de jointure de type «exécution du constructeur» dont la signature égale à ConstructorPattern.
initialization(*ConstructorPattern*)	sélectionne chaque point de jointure de type «initialisation d'objet» dont la signature égale à ConstructorPattern.
preinitialization(*ConstructorPattern*)	sélectionne chaque point de jointure de type «pré-initialisation d'objet» dont la signature égale à ConstructorPattern.
staticinitialization(*TypePattern*)	sélectionne chaque point de jointure de type «staticinitialization» dont la signature égale à TypePattern.
handler(*TypePattern*)	sélectionne chaque point de jointure de type «exception handler» dont la signature égale à TypePattern.
adviceexecution()	sélectionne toute les points de jointure de type «adviceexecution».
within(*TypePattern*)	sélectionne chaque point de jointure où le code exécutant est défini dans un type égale à TypePattern.
withincode(*MethodPattern*)	sélectionne chaque point de jointure où le code exécutant est défini dans une méthode dont sa signature égale à MethodPattern.
withincode(*ConstructorPattern*)	sélectionne chaque point de jointure où le code exécutant est défini dans un constructeur dont sa signature égale à ConstructorPattern.
cflow(*Pointcut*)	sélectionne chaque point de jointure dans le flux de contrôle de n'importe quel point de jointure P sélectionné par *Pointcut*, y compris P lui-même.
cflowbelow(*Pointcut*)	sélectionne chaque point de jointure dans le flux de contrôle de n'importe quel point de jointure P sélectionné par *Pointcut*, mais pas P lui-même.

this(*Type* or *Id*)	sélectionne chaque point de jointure où l'objet qui s'exécute actuellement (l'objet relié à *this*) est une instance de *Type*, ou du type de l'identificateur *Id*.
target(*Type* or *Id*)	sélectionne chaque point de jointure où l'objet cible (l'objet sur lequel un appel ou l'opération de champ est appliquée à) est une instance de *Type*, ou du type de l'identificateur *Id*.
args(*Type* or *Id*, ...)	sélectionne chaque point de jointure où les arguments sont des instances du type approprié (ou le type de l'identificateur si en utilisant cette forme). Un argument nulle est sélectionné si le type statique de l'argument (type paramètre de type ou type de champ déclarés) est le même, ou un sous-type de, le type d'*args* spécifié.
PointcutId(*TypePattern* or *Id*, ...)	sélectionne chaque point de jointure qui est sélectionné par *PointcutId*.
if(*BooleanExpression*)	sélectionne chaque point de jointure où l'expression booléenne évalue à vrai. L'expression booléenne utilisée peut accéder seulement aux membres statiques, les paramètres exposés par le point de coupure ou la consigne, et les forme de type *thisJoinPoint*. En particulier, il ne peut pas appeler des méthodes non-statiques dans l'aspect ou utiliser les valeurs retournées ou les exceptions exposés par la consigne *after*.
!*Pointcut*	sélectionne chaque point de jointure qui n'est pas sélectionné par *Pointcut*.
Pointcut0&&*Pointcut1*	sélectionne chaque point de jointure qui est sélectionné par *Pointcut0* et *Pointcut1*.
Pointcut0 || *Pointcut1*	sélectionne chaque point de jointure qui est sélectionné par *Pointcut0 ou Pointcut1*.
(*Pointcut*)	sélectionne chaque point de jointure sélectionné par *Pointcut*.

c. Consigne (Advice)

Le support de comportement transversal dynamique d'AspectJ vient sous forme de consigne [LAD10]. Cette dernière affecte l'exécution du système. Après la sélection des points de jointure par le point de coupure, nous devons augmenter ces points de jointure avec un comportement supplémentaire ou alternatif. La consigne dans la POA fournit une facilité pour faire ça. Une consigne ajoute le comportement avant (*Before*), après (*After*), ou autour (*Around*) les points de jointure sélectionnés i.e. la consigne *autour* entoure l'exécution du point de jointure et peut l'exécuter zéro ou plusieurs fois.

Pendant que la consigne *avant* est relativement sans problèmes, il peut y avoir trois interprétations de la consigne *après*: après l'exécution normale d'un point de jointure (*after returning*), après qu'elle lance une exception (*after handling*), ou après qu'elle fasse l'un ou l'autre (*after*). L'AspectJ traite toutes ces situations avec la consigne appropriée.

Le corps d'une consigne est semblable au corps d'une méthode. Comparées aux méthodes de langages orientés objets traditionnels, les consignes ne sont pas appelées explicitement. Mais, l'exécution d'une consigne est déclenchée automatiquement quand le flux de contrôle atteint le point de jointure désigné. Cette propriété peut être exprimée comme suit: la consigne est inconsciente (*obvious*) du point de jointure. Par conséquent, les modules du programme dans lesquels les événements dans leur flux de contrôle sont désignés, sont aussi inconscients aux consignes correspondantes [NAG06].

2.4.2 Concept Transversal Statique (Introduction)

L'introduction *(inter-type declaration)* [LAD10] est une construction transversale statique qui modifie la structure statique des classes, des interfaces et des aspects dans le système. Par exemple, nous pouvons ajouter une méthode ou un champ à une classe, ou déclarer un type pour implémenter une interface. Dans une introduction, un seul type (un aspect) déclare la structure pour les autres types (classes, interfaces et aussi les aspects). Une autre forme d'introduction «*member introduction*» offre une façon pour ajouter de nouvelles méthodes et champs à d'autres types.

2.4.3 Aspects

Parce que le but final de la POA est d'avoir un module qui rassemble la logique transversale, nous avons besoin d'une place pour exprimer cette logique. L'aspect fournit une telle place. Les aspects sont des unités modulaires de l'implémentation transversale. Ils définissent précisément comment les fonctionnalités transversales dynamiques et statiques doivent avoir lieu au moyen de points de coupure, des consignes et d'introductions [LAD10].

L'aspect est l'unité principale dans le langage AspectJ, de la même façon qu'une classe est l'unité principale à Java (La déclaration d'aspect est semblable à la déclaration de classe, ainsi il définit un type et une implémentation pour ce type). Il peut être associé avec d'autres aspects dans une manière semblable aux associations entre les classes. Ainsi, les aspects peuvent contenir des données, des méthodes et des membres de classe, juste comme une classe Java normale.

2.5 Les Challenges de l'Evolution du Logiciel Orienté Aspect

L'adoption industrielle de la programmation orientée objet dans les premières années quatre-vingt-dix a déclenché une variété de recherche sur l'évolution du logiciel, la rétro-ingénierie, la réingénierie et la restructuration. La même chose se passe actuellement pour le paradigme OA. Puisque tous les systèmes logiciels sont soumis à l'évolution, et puisque la recherche dans la programmation OA a atteint la maturité avec plusieurs produits de recherche industriels et académiques (voire la section 2.3.2), les systèmes OA eux-mêmes doivent évoluer aussi (c'est nécessaire de comprendre et de traiter son évolution).

Nous définissons l'évolution du logiciel OA comme *la modification progressive des éléments d'un système logiciel OA pour améliorer ou maintenir sa qualité avec le temps, par le changement de ses contextes et ses besoins*.

Le paradigme OA fournit beaucoup d'avantage au niveau de la modularisation des préoccupations transversales. Chaque préoccupation du programme devrait être localisée dans un seul module. Donc, si les besoins concernant cette préoccupation changent, seulement ce module doit être modifié. Chaque module devrait adresser seulement une préoccupation, afin que le développeur de ce module puisse se concentrer exclusivement sur cette préoccupation. Malheureusement ce paradigme vient avec son propre ensemble de problèmes. Il y a des limites des langages OA concernant l'évolutivité du logiciel que nous adressons dans cette section. Il y a deux concepts majeurs qui caractérisent le paradigme OA, et en même temps ils sont la cause de la plupart des défis de l'évolution de logiciel OA: ***la quantification*** et l'***inconscience***.

2.5.1 La Quantification & l'Inconscience

Le paradigme OA est définie par Filman et Friedman [FIL01] comme: la quantification + l'inconscience (*obliviousness*):

— **L'inconscience:** Les points de coupure spécifient les places dans le code source ou son exécution, où le code des consignes doit être tissé. Cela veut dire que les aspects ne sont pas invoqués explicitement par le programme. Le

programme de base (i.e. le programme sans les aspects) n'est pas conscient des aspects appliqués sur lui. Au contraire, c'est les aspects eux-mêmes qui spécifient quand et où ils agissent sur le programme. Cette propriété qui a été connue sous le nom de la propriété d'inconscience (*obliviousness*), est une des caractéristiques les plus essentielles d'un langage de programmation OA.

— **La quantification:** C'est l'idée qu'on peut écrire un aspect qui affecte arbitrairement différentes places non-locales dans un programme.

La quantification et l'inconscience causent des problèmes tels que les difficultés dans la compréhension et l'évolution du logiciel OA. Donc, la POA, en prévenant l'enchevêtrement et l'éparpillement du code, améliore la qualité du code d'une part, et en même temps, en introduisant la quantification et l'inconscience, elle la décroît d'une autre part [PRZ10].

2.5.2 Les Défis de l'Evolution

Les différents challenges de l'évolution des logiciels OA peuvent être résumés dans les points suivants:

Compréhension difficile: une limitation importante provient de la relation entre les classes et les aspects. Avec la POA, les aspects devraient être capables de faire des changements radicaux au comportement de différentes parties d'une application, sans le savoir des développeurs des classes et des méthodes dont le comportement est affecté. Cependant, les développeurs d'une méthode peuvent être inconscients des aspects qui affectent son comportement, mais le développeur d'un aspect ne doit pas être inconscient des méthodes qu'il affecte. Également, le développeur d'un aspect peut quantifier sur plusieurs caractéristiques des modules du programme, mais les développeurs de ces modules ne peuvent pas influencer la quantification i.e. quelles parties de quels modules sont quantifiées, depuis qu'ils sont inconscients à la quantification. Seiter [SEI08] affirme que "*l'inconscience mène aux programmes qui sont difficiles à développer, à comprendre et à maintenir"*. Clifton et Leavens [CLI03] aussi assurent que "*l'inconscience le rend difficile pour un développeur pour comprendre l'effet de maintenir une classe ou un aspect sur le système total*".

Couplage fort: les aspects peuvent être couplés fortement aux classes du programme, puisque il n'existe pas une interface explicite entre eux. Donc, ils ne peuvent pas être bien réutilisables ou maintenables. D'un autre côté, et dans la plupart des langages OA, les concepts du langage qui expriment les aspects, les consignes et les spécifications des points de coupure sont aussi couplés fortement les uns avec les autres. Par le couplage fort nous signifions que ces concepts ne peuvent pas être séparés syntaxiquement les uns des autres. Ceci peut restreindre l'évolutivité des spécifications du tissage. Par conséquent, le code résultant exige une maintenance intensive pendant le développement d'un système [NAG06].

Un flux complexe du programme: l'abstraction masque les détails inessentiels et donc réduit la complexité du système sous-jacent. La bonne abstraction mène à la création de modules bien-isolés. En effet, chaque module représente un sous-système beaucoup plus petit, l'abstraction offre une façon de contourner la complexité à un niveau où on peut la bien traiter. Mais l'abstraction introduite par la POA n'est pas sans coûts. L'abstraction, par sa nature, masque les détails. Dans les systèmes logiciels, les plus hauts niveaux d'abstraction toujours signifient que moins d'information est disponible au niveau du code. L'analyse d'un segment du code OA ne nous donne pas l'histoire entière qui dépliera pendant l'exécution du système. On ne peut pas savoir (sauf à travers un bon support d'outils) qu'une action transversale aura lieu dans une certaine partie du code [LAD10].

Maintenance intensive: l'association déclarative d'aspects avec le code de base pour la décomposition principale est une technique puissante pour éliminer la dispersion du code. Elle peut résulter en modules qui sont plus concentrés et plus concis. Cependant, la technique peut aussi être utilisée pour écrire un code qui est difficile de comprendre et de maintenir. Par exemple, la manière déclarative dans laquelle les aspects sont introduits signifie qu'un programmeur doit, en général, comprendre le programme entier pour raisonner au sujet de toute opération dans la décomposition principale. Dans les systèmes logiciels OA, les chercheurs ont découvert que les changements sont propagés aux modules du système qui ne sont pas en relation avec ces changements. Cela a été causé par les interdépendances, créées par les points de coupure et les introductions, entre le code de base et les aspects. La séparation avancée des préoccupations dans les versions OA rend les changements moins

visibles puisque des modules inattendus ont été affectés [RAS10].Ce qui exige une maintenance intensive pendant la modification du système logiciel.

La fragilité des points de coupure: les points de coupure qui simplement énumèrent les signatures de toutes les méthodes qu'ils doivent capturer sont appelés points de coupure basés sur une énumération extensible. De tels points de coupure sont spécifiques et ils peuvent être violés facilement quand le programme de base évolue. Des définitions plus robustes du point de coupure peuvent être obtenues en mentionnant explicitement seulement l'information qui est exigée absolument et en utilisant des patrons (*wildcard*) ou d'autres mécanismes pour abstraire sur certains détails de l'implémentation. Les tels points de coupure sont connus sous le nom de points de coupure intentionnels ou basés-patron [MENk07].

Quand le code de base évolue la sémantique des points de coupure est altérée silencieusement. Il y a peut-être plusieurs changements non-locaux du code de base qui modifient la sémantique du point de coupure (les deux types de point de coupure) en terme des points de jointure sélectionnés, comme la modification du nom des: classes, des méthodes ou des champs, le déplacement de méthode ou de classe…etc. Donc, actuellement l'évolution du système est sujette à l'erreur due à la fragilité des points de coupure. Finalement, les résultats de plusieurs recherche [FIG08, KAS07, MUN08, TOU03] indiquent que la POA mène à un logiciel qui est dur, ou peut-être même plus dur, à évoluer et à réutiliser qu'avant i.e. par rapport aux logiciels implémentés par les autres paradigmes (procédurale, orienté objet…).

2.5.3 Des Etudes Empiriques

Un ensemble de travaux de recherche a été concentré principalement sur la spécification des différents défis dans l'évolution du logiciel OA [ALD04, ALE04, STE06]. Par exemple, une recherche antérieure a indiqué que l'utilisation de certains mécanismes de la POA peut violer l'encapsulation du module [ALD04] et même introduit de nouveaux types de fautes [ALE04]. En particulier, certains chercheurs affirment que ces fautes seront probablement amplifiées dans la présence de changements évolutionnaires [KAS07]. Par exemple, quand les points de coupure permettent à certains changements d'être absorbés et donc augmentent la stabilité

d'une conception, ils sont aussi une source d'effets secondaires qui réduisent la stabilité [RAS10].

D'autres ont prouvé avec une évidence empirique les problèmes d'évolution de la POA qui se produisent dans la pratique. Leur analyse confirme que le manque de conscience entre les modules de base et les modules aspectuels (*l'inconscience*) a tendance à mener aux implémentations inexactes. Ferrari et al. [FER10], par exemple, ont examiné comment l'inconscience influence la présence de fautes dans l'évolution des programmes OA. Ils ont trouvé que l'inconscience facilite l'apparition de fautes sous les conditions de l'évolution du logiciel. Ils ont montré que 40% de fautes rapportées étaient dû au manque de conscience entre le code de base et les aspects. Et ils ont indiqué que les mécanismes de la POA présentent la même tendance aux fautes quand nous considérons le système total et les implémentations spécifiques des préoccupations. Les résultats ont révélé l'impact négatif de l'inconscience sur la tendance aux fautes des programmes implémentés avec AspectJ.

Par conséquent, des dépendances logiques existent dans le logiciel OA qui rendent son évolution de plus en plus difficile i.e. par exemple, un changement dans une classe spécifique peut exiger un changement dans les autres classes ou dans les aspects, bien que; il n'existe pas de dépendances traditionnelles (ex., flux de données ou flux de contrôle) entre ces entités du logiciel OA. Des dépendances cachées (logique) existent dans n'importe quel paradigme de programmation, mais d'après les effets négatifs de l'inconscience sur l'évolution du logiciel OA présentés au-dessus, l'existence de telles dépendances dans le logiciel OA est plus vaste.

2.6 Les Approches supportant l'Evolution du Logiciel OA

L'évolution du logiciel est un sujet discuté largement dans les trente dernières années. Plusieurs propositions ont été présentées, chacune avec ses mérites et ses limites. L'évolution du logiciel OA est un sujet plus récent néanmoins plusieurs travaux ont été déjà faits. Les langages OA sont différents des langages procéduraux ou orientés objet. Pour bien supporter les tâches de compréhension du programme et de la maintenance du logiciel, de nouvelles techniques pour l'analyse et le test du logiciel

OA sont fortement exigées. Nous présentons dans cette section les travaux les plus intéressants qui traitent les différentes tâches de l'évolution du logiciel OA.

2.6.1 L'analyse du Flux de Contrôle et du Flux de Données

La question principale de la recherche qui devrait être adressée pour supporter la compréhension et la maintenance du logiciel OA est: développer une représentation exacte et effective du programme pour modéliser les flux de contrôle et les flux de données dans les programmes OA.

Dans [ZHA08a] les auteurs suggèrent que beaucoup de techniques de la compréhension du programme et de la maintenance du logiciel qui ont été bien développées pour les langages de programmation procéduraux et orientés objet sont applicables aux langages OA, avec plus ou moins de changement. Par exemple, beaucoup de techniques pour les langages de programmation procéduraux ou orientés objet se basent sur les informations des flux de contrôle et des flux de données qui sont rassemblées typiquement par l'analyse du flux de contrôle et du flux des données et sont représenté par un graphe d'appel (*call graph)*, un graphe du flux de contrôle (*control-flow graph*) ou un graphe de dépendance du programme (*program dependence graph*).

Récemment, plusieurs chercheurs considèrent l'analyse du flux de contrôle et du flux de données pour les programmes OA. Sereni et de Moor [SER05] proposent un graphe d'appel simple pour les programme OA sans considérer une analyse du flux de contrôle plus détaillée. Rinard et al. [RIN04] proposent un graphe du flux de contrôle dans le système de classification pour les programmes OA. D'autres approches qui fournissent des solutions partielles pour l'analyse des flux de contrôle dans les programmes OA ont été proposées par Zhao [ZHA04], Xu et Rountev [XUR07], et Bernardi et Lucca [BER07]. De plus, Zhao et Rinard [ZHA02a, ZHA03a] définissent les graphs de dépendance du système pour les programmes OA pour le slicing, et Xu et Rountev [XUR08] étendent ce travail en s'occupant de situations plus complexes telles que les consignes multiples avec un point de jointure, et les consignes dynamiques dans les programmes OA.

2.6.2 L'Analyse de l'Impact du Changement

L'analyse de l'impact du changement est une technique utile pour la maintenance du logiciel. Zhao [ZHA02b] présente une approche pour supporter l'analyse de l'impact du changement des programmes OA basée sur le slicing du programme. Stoerzer et Graf [STO05] se focalisent sur la modification sémantique du code de base et introduisent une analyse de delta pour négocier avec le problème de la fragilité des points de coupure. Cette analyse est basée sur une comparaison des ensembles des points de jointure sélectionnés pour deux versions du programme. Shinomi et Tamai [SHI05] discutent l'analyse de l'impact du tissage d'aspect et sa propagation à travers le programme de base et les aspects. Ils se concentrent sur l'impact du changement causé par l'aspect du tissage. Récemment, Zhang et al.[ZHA08b] présentent les changements atomiques pour les différents concepts de la POA pour capturer les changements sémantiques du programme dans le niveau du code source. Leur modèle de l'impact du changement peut être utilisé pour déterminer les parties du programme affectées, les tests affectés et leurs changements responsables effectivement.

2.6.3 Le Test du Logiciel OA

Le test de régression (*Regression testing*) est une autre technique utile pour la maintenance du logiciel. Le test de régression des programmes OA a été étudié par Xu et Rountev [XUR07] et Zhao et al. [ZHA06].

2.6.4 Résoudre le Problème de la Fragilité des Points de Coupure

Dans [ALJ09] les auteurs proposent ParaAJ (Parametric Aspects), comme une extension à AspectJ. ParaAJ permet aux classes de spécifier quels aspects devraient être appliqués, et permet aux applications de spécifier quels aspects à appliquer à quelles classes avec quelle manière. Cela rend plus facile pour les classes et les aspects d'être développés et maintenus indépendamment, et encourage la réutilisation des deux. Nous pensons que cette approche viole le principe de la POA, leur langage ne supporte pas l'inconscience qui est la caractéristique principale du paradigme OA. Si nous voulons améliorer l'évolution des programmes OA, nous ne devons pas toucher le programme mais il faut chercher un modèle pour supporter cette évolution i.e. c'est évident qu'appeler directement des aspects à partir du code de base n'est pas du tout POA.

Tourwé et al. [TOU04] ont proposé un environnement avancé de la gestion de point de coupure, basé sur les techniques de l'apprentissage automatique (*machine learning techniques*). Ils identifient trois problèmes principaux pour les langages OA de nos jours: le premier est que le langage du point de coupure est très primitif et n'est pas assez expressif, le second est que ces points de coupure sont couplés fortement à la structure d'une application. Le dernier est que les développeurs sont forcés de traiter les points de coupure à un niveau très faible. Comme une solution aux problèmes précités ils proposent d'inclure la notion de points de coupure générés de façon inductive dans le langage OA, dans cette direction les développeurs peuvent spécifier les points de coupure en utilisant une interface graphique et le Framework générera automatiquement les points de coupure correspondants.

Gybels et al. [GYB03] ont traité le problème du patron arrangé. Les langages OA utilisent la sélection des patrons (*pattern matching*) pour capturer les points de jointure, c'est une bonne technique pour décrire la sémantique désirée d'une préoccupation. Mais c'est encore dépendant de la convention de nommage. Les auteurs ont proposé un mécanisme linguistique plus flexible pour implémenter la transversalité comme des patrons, et par conséquent éviter le problème exposé de la sélection des patrons.

2.7 Bilan

Nous avons présenté dans ce chapitre le paradigme OA, et ses bénéfices par rapport aux autres paradigmes de programmation. Nous avons mis l'accent sur le langage AspectJ qui constitue la meilleure implémentation de la POA.

Avec la POA, les préoccupations transversales sont invisibles pour les classes. En d'autres termes, les classes sont inconscientes des aspects qui modifient leur comportement. L'inconscience de classes est une des décisions prises pendant la conception de la POA. Bien qu'elle ait prouvé pour être utile dans beaucoup de cas (le développement du logiciel), elle a un impact sur l'évolutivité du système OA. Nous avons discuté dans ce chapitre les différents challenges qui doivent être adressés dans l'évolution du logiciel OA. Et nous avons donné une brève présentation des travaux en cours dans cette zone de recherche. Mais beaucoup de travail doit encore être fait pour supporter l'évolution du logiciel OA.

Nous constatons que les dépendances entre les classes et les aspects doivent être plus explicites pour bien supporter l'évolution des logiciels OA. Par conséquent, l'utilisation d'un modèle d'évolution augmente la compréhension et réduit le temps exigé pour l'évolution du logiciel OA.

CHAPITRE 3

L'Analyse de l'Evolution du Logiciel Orienté Aspect

"Go and collect data; analyze it; and finally, learn from it."

Andreas Zeller, 2010 [TIC10]

Pour comprendre pourquoi les systèmes logiciels deviennent moins maintenables quand ils sont changés de façon continue et prédire leurs futurs changements; nous devons étudier leurs dépôts de versions. Cet axe de recherche est connu sous le nom «analyse de l'évolution du logiciel». C'est l'analyse rétrospective de l'évolution. Donc, on analyse les données disponibles dans les dépôts du logiciel pour découvrir l'information intéressante au sujet de son évolution et son futur développement. L'analyse de l'historique de l'évolution peut aider pour identifier les changements nécessaires, comprendre l'impact du changement et fournir une facilité pour suivre les changements et déduire des relations logiques entre les entités changées. Nous allons découvrir cette zone de recherche dans ce chapitre, les systèmes de contrôle de version actuels, les approches et les techniques d'analyse, …etc. Et nous explorons l'analyse de l'historique des logiciels OA, où nous démontrerons que les systèmes de contrôle de version actuels n'entreposent pas l'information complète au sujet de l'évolution de ces logiciels.

3.1 La Gestion de la Configuration Logicielle

Le logiciel évolue constamment en subissant de nouveaux changements. Pour contrôler de tels changements, il est nécessaire d'établir un contrôle efficace et systématique des versions produites et délivrées aux utilisateurs. Cela peut être accompli à travers la gestion de la configuration logicielle (*SCM: Software Configuration Management)*. C'est un ensemble d'activités qui a l'intention de gérer les changements produits pendant tous le cycle de vie du logiciel, et aider aussi pour garantir sa qualité [PRE09]. La gestion de la configuration logicielle concerne le suivi et le contrôle des items qui sont importants et peuvent être changés dans le logiciel. Dans le génie logiciel il signifie habituellement le suivi des changements faits avec le temps à plusieurs fichiers du système logiciel [MAL10].

La gestion de la configuration logicielle est une façon standardisée pour suivre et contrôler les changements causés par plusieurs développeurs qui travaillent ensemble dans des projets logiciels. L'idée est de suivre les changements dans les fichiers pour voir s'il y a deux (ou plusieurs) développeurs qui essaient de changer un fichier en même temps, et de gérer cet événement. S'il n'y aurait aucun contrôle, un développeur peut annuler les changements d'un autre. En plus, c'est très important pour trouver la cause des bogues quand un fichier a été changé. Avec la gestion de la configuration logicielle on est capable de revenir dans les révisions du fichier. C'est indispensable pour reproduire ce qui a été fait ou le supprimer.

La gestion de la configuration logicielle est composée de cinq tâches: (1) l'identification des changements, (2) le contrôle de version, (3) le contrôle des changements, (4) la vérification de la configuration, et (5) les rapports (pour informer ce qui est arrivé, qui a fait les changements, quand le changement a été fait, qui sera influencé par le changement…etc.). La tâche de *contrôle de version* est la plus connue et, en même temps, elle a plus de responsabilités, parce qu'elle traite le stockage et l'extraction de différentes versions des artefacts générées pendant le processus logiciel.

3.2 Les Systèmes de Contrôle de Versions

Depuis le début des années 1970s, le versioning du logiciel était constamment un sujet actif de recherche dans l'ingénierie du logiciel. Les Systèmes de Contrôle de Versions (SCVs) ont évolué d'être quelque chose de non entendu il y a quatre décennies, à quelque chose qui est utilisée par presque tous les développeurs de logiciels.

C'est très commun que des développeurs multiples sont impliqués dans le développement d'un grand projet logiciel. Ces développeurs qui sont probablement dispersés dans différents bureaux ou même éparpillés partout dans le monde, ont besoin de modifier simultanément le logiciel. Les SCVs sont utilisés largement de nos jours pour faciliter le développement distribué du logiciel. Il y a deux fonctions principales des SCVs:

1. La possibilité de naviguer dans les versions d'un projet. Aller vers un état spécifique d'un projet entier, ou un seul fichier à tout point dans le temps. Cette fonction fournit le support clé pour corriger des erreurs.

2. Coordonner le travail d'un nombre illimité de développeurs sur le même projet ou le même fichier, et gérer les conflits possibles qui se produisent.

3.2.1 Les Approches de Versioning

Il y a deux approches du versioning. La plupart des SCVs actuels [COS04] offrent les deux approches, en laissant la décision à l'utilisateur pour choisir l'approche à utiliser.

a. Le Versioning Pessimiste

Les SCVs pessimistes suivent le paradigme *lock-modify-unlock* [COS04] pour traiter le problème de partage d'artefact. Dans les telles approches, seule une personne est autorisée à changer un artefact à la fois. Un utilisateur doit verrouiller (*lock*) un artefact avant qu'il puisse commencer à faire des changements sur celui ci. Pendant ce temps, d'autres utilisateurs ne peuvent pas appliquer des changements à l'artefact (ils peuvent seulement accéder en

lecture). Uniquement après que l'utilisateur entrepose sa version dans le dépôt et déverrouille l'artefact (*unlock*), d'autres utilisateurs peuvent faire un verrouillage, pour modifier la version la plus récente. L'approche pessimiste interdit le travail en parallèle sur les artefacts et par conséquent peut causer une sérialisation inutile. En plus, le verrouillage peut exiger l'intervention d'un administrateur, si un utilisateur a oublié de déverrouiller les artefacts [ALT09].

b. Le Versioning Optimiste

Une alternative au versioning pessimiste est l'approche optimiste qui est aussi appelée la solution *copy-modify-merge* [COS04, SHE02a]. Chaque utilisateur du SCV contacte le dépôt du projet et crée une copie de travail personnelle "*working copy*". Cette dernière est une réflexion locale de l'artefact du dépôt. Les utilisateurs peuvent travailler alors en parallèle, en modifiant leurs copies privées. Si un utilisateur enregistre ses changements sur le dépôt, quand un autre utilisateur essaie d'enregistrer ses changements, le SCV l'informe que son artefact est périmé (il n'a pas été actualisé). En d'autres termes, l'artefact dans le dépôt a été changé d'une façon ou d'une autre depuis que l'utilisateur l'a récupéré dernièrement à partir du dépôt. Donc, il doit intégrer sa version avec l'artefact modifié dans le dépôt par une commande de mise à jour (*update)*, ce qui déclenche les phases: comparaison, détection de conflit, résolution de conflit et fusion. Après avoir accompli cette tâche, l'artefact fusionné peut être entreposé dans le dépôt du SCV [ALT09]. Pratiquement, le montant du temps pris pour résoudre les conflits dans le versioning optimiste est petit par rapport au temps perdu par l'approche pessimiste [COS04].

3.2.2 Le Principe des SCVs

Le flux d'activités pour contrôler les versions dans les SCVs actuels implique la création d'un dépôt, l'importation des fichiers d'un projet, et ensuite les tâches *check-out, merge,* et *commit* sont appliquées récursivement pour ajouter des révisions au dépôt de version:

Check-out: les SCVs utilisent des dépôts de fichiers qui contiennent tous les fichiers d'un projet particulier qui est sous contrôle de version, et toutes les versions de ces fichiers. Cependant, aucune activité n'est appliquée directement

sur les fichiers du dépôt -au lieu que- chaque utilisateur a une copie d'une version particulière des fichiers, habituellement la dernière. Il peut accomplir cette copie locale à travers la commande *check-out.* Cette opération crée une copie de la révision pour être modifiée et la réserve pour l'utilisateur. Elle relie aussi la nouvelle copie à son originale avec la relation *revision-of.* L'utilisateur peut modifier alors la copie avec un éditeur arbitraire. Toute opération *check-out* ultérieure de la même révision originale cause un *branchement*, avec un avertissement qui déclare qu'une opération de fusion sera nécessaire plus tard.

Merge: pour rendre le développement parallèle possible qui est d'importance particulière dans le développement du logiciel complexe, une approche commune est d'autoriser en même temps: la modification d'un artefact, et de combiner les différentes modifications dans une seule nouvelle version de l'artefact (le versioning optimiste). Par conséquent, les SCVs doivent supporter un processus de fusion (*merge*) pour les artefacts ayant évolués simultanément. Le processus de fusion constitue des phases de: la comparaison, la détection du conflit, la résolution du conflit et la fusion [ALT09]. Pour faire ceci, le développeur utilise la commande *update* ou une autre commande semblable.

Commit: finalement, après que tous les changements soient faits, l'opération *commit* (ou *check-in*) détecte et enregistre les changements qu'un utilisateur a fait à ses fichiers sur sa copie privée dans un dépôt. Pour chaque fichier dans le logiciel, le dépôt enregistre des détails tels que la date de création du fichier, les changements au fichier avec le temps, la taille et une description des lignes affectées par le changement.

En outre, le dépôt associe pour chaque changement la date exacte de son occurrence, un commentaire tapé par le développeur pour indiquer la raison du changement, et dans quelques cas une liste d'autres fichiers qui faisaient partie du changement décrite par le commentaire du développeur. De tels enregistrements détaillés autorisent le retour en arrière du code à tout point dans le temps, pour récupérer une version ancienne du code. Ils peuvent être utilisés aussi pour abandonner des nouveaux changements qui ont été trouvés sans rapport ou avec bogue [HAS03]. Cette opération rend la copie (modifiée) visible à d'autres utilisateurs. Avant qu'une révision soit entreposée, elle devrait

satisfaire quelque critère de contrôle de qualité, tel qu'un test réussi, pour s'assurer que c'est utilisable par d'autres développeurs [TIC88].

3.2.3 Les Types des SCVs

a. Les SCVs centralisés

Principe

Ils suivent le principe client-serveur. Les utilisateurs accèdent à un dépôt central à travers un client. Typiquement, leurs machines locales tiennent seulement une copie du travail de l'arbre du projet. Les changements en une seule copie du travail doivent être entreposés dans le dépôt central avant qu'ils soient propagés à d'autres utilisateurs. L'historique de version est seulement disponible dans le dépôt central [ALT09].

Donc, le serveur est toujours en ligne et entrepose toutes les données. Il répond aux demandes de tous les utilisateurs, et il joue un rôle intermédiaire entre eux. Le flux de communication de base est présenté dans la Figure 3.1. Un développeur transfère les données à un serveur qui suit les changements, marque les versions et gère les conflits. Tous les autres développeurs reçoivent les changements à partir du serveur [MUK11]. Si un fichier est changé tous les autres qui ont l'accès à ce projet peuvent voir le changement. Si quelqu'un déclenche un nouveau branchement, le test sera possible pour tout le monde pour le voir. C'est-à-dire, si quelque chose change tout le monde le verra, surtout si quelque chose était mal faite ou fausse comme cela pourrait interrompre les suites de test d'un projet.

Un inconvénient évident à ce principe de communication, est qu'un serveur présente un seul point de panne. Dans le cas d'une panne du serveur à cause d'une surcharge, d'une panne d'alimentation, ou une attaque, la plateforme de collaboration complète échoue et les développeurs sont entravés dans leur travail sur un projet. En plus, il représente aussi un seul point de contrôle. Si un tel SCV est utilisé en même temps par un nombre énorme de développeurs, il montre évidemment le problème de la scalabilité.

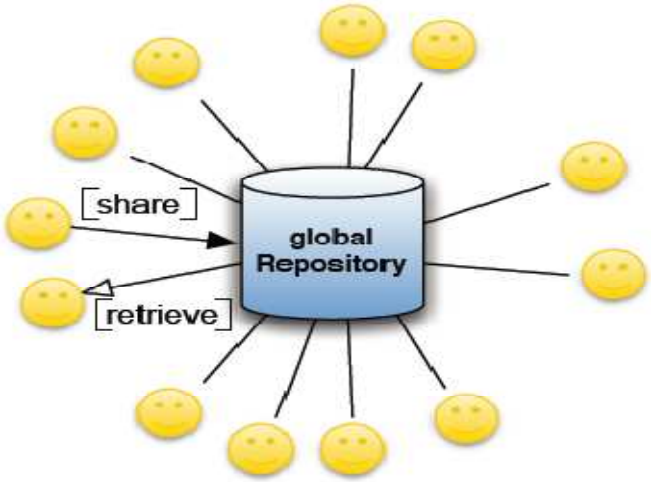


Figure 3.1. Le principe de base des SCVs centralisés [MUK11].

Un serveur peut fonctionner hors de ressources qui mènent à sa panne ou, dans les meilleurs cas, une performance faible.

Outils

Le premier SCV est le *Source Code Control System* (SCCS) [ROC75] qui a été développé en 1972. Pour supporter les variantes, le *Revision Control System* (RCS) [TIC85] a été développé en 1982, en remplaçant SCCS. Il tient compte de la création d'une version qui existe en parallèle avec une autre version dans un branchement. Par ailleurs, cet outil peut contrôler des fichiers seuls, il n'a pas pu contrôler les projets entiers qui ont consisté en plusieurs fichiers. RCS et SCCS permettent seulement de contrôler les versions d'un seul artefact.

RCS a été remplacé finalement par le *Concurrent Versions System* (CVS) [GRU86]. Ce dernier a commencé comme une collection de scripts qui opèrent sur RCS au milieu de l'année 1984. CVS a été publié dans la fin des années 90, et il est encore utilisé dans beaucoup de projets. Il ne compte plus sur RCS, comme il est devenu un projet logiciel indépendant. CVS a facilité le travail sur un ensemble d'artefacts. C'est le plus célèbre et il constitue un outil principal de choix pour beaucoup de développeurs, il permet de contrôler le code source ou la documentation pour une longue durée.

L'outil subversion (SVN) [NAG05] a été développé pour vaincre la limitation de CVS, —contrôler l'historique des fichiers seuls—, par la compagnie CollabNet

qui a originairement développé le CVS. Il a été publié dans la fin de l'année 2000. Bien qu'il ait été adopté par un nombre énorme de projets, il n'a jamais complètement remplacé le CVS.

b. Les SCVs distribués

Principe

La configuration d'un SCV distribué est illustrée dans la Figure 3.2. Par contraste avec les dépôts centraux, dans les SCVs distribués, chaque développeur a son propre dépôt sur sa machine locale en plus de leurs copies de travail. Il peut synchroniser facultativement leurs dépôts avec les autres. Chaque développeur possède son branchement du projet (une copie), et personne ne peut voir ce que les autres développeurs font comme c'est une copie locale du projet. Il n'y a aucun emplacement central où les développeurs travaillent. Si un développeur crée un branchement de son projet, personne ne verra ce branchement. L'historique de version est distribué sur les différents dépôts. Donc, les changements peuvent être tracés localement sans établir une connexion à un serveur. En plus, des opérations telles que le commit, voire l'historique, et la récupération des changements sont plus rapides, puisque on n'a pas besoin de communiquer avec un serveur central sur un réseau [ALT09].

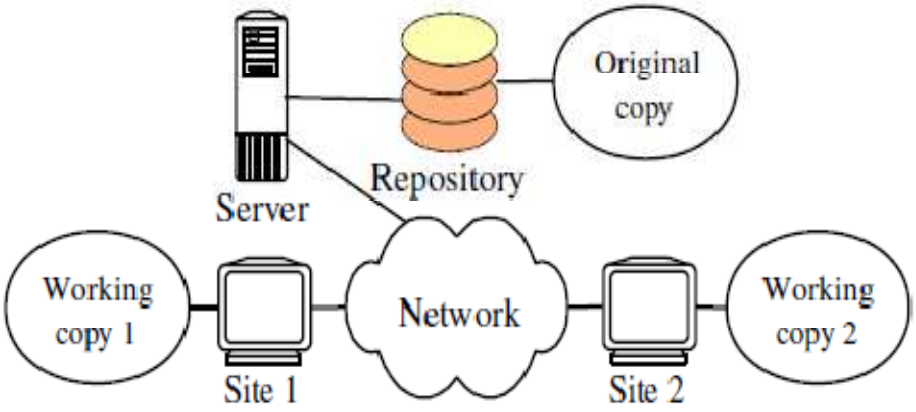


Figure 3.2. Le principe de base des SCVs distribués [SHE02b].

Outils

Le premier SCV distribué était BitKeeper (BK) [KRO02]. Le développement a commencé en 1997 sur cet outil, et il est encore progressif. Les autres outils qui donnent suite à ce modèle sont par exemple Git [HAM05]. Il est une nouvelle approche pour manier les branchements. Linus Torvalds a initialisé le développement de Git en avril 2005. Aujourd'hui c'est le SCV distribué le plus populaire, utilisé par beaucoup de projets. Git a été étendu et amélioré pour créer Mercurial [MAC05], Bazar [BAZ14]…etc.

3.3 L'Analyse de l'Historique de l'Evolution

Comprendre comment les programmes évoluent?, ou comment ils continuent à changer? est un besoin clé avant d'entreprendre toute tâche dans le génie logiciel, ou la maintenance du logiciel. Les équipes du logiciel consistent en différents membres avec des rôles distincts dans leurs projets. Un *développeur* est intéressé à savoir comment les artefacts en rapport ont été changé dans le passé et pourquoi ces changements se sont produits. Un *ingénieur* veut considérer comment un système a évolué afin qu'il puisse apprendre d'expériences antérieures avant de reconcevoir le système. Un *gestionnaire* est intéressé à comprendre le développement progressif et le travail antérieur d'un programmeur, avant d'assigner un nouveau travail. Un *chercheur* veut étudier comment les grands projets ont évolué afin que les leçons apprises puissent être appliquées aux nouveaux projets. Finalement, un *testeur* souhaite savoir quelles parties du programme à tester, et parler avec qui s'il a des questions ou des problèmes à signaler [GER05].

Les diverses questions que ces différents membres posent d'un projet logiciel peuvent souvent être répondues par l'analyse de l'historique de l'évolution. Cette section présente le principe générale de l'analyse d'évolution, et donne une étude sur quelques travaux intéressants dans cette zone de recherche.

3.3.1 Le Principe Générale

Les dépôts du code source contiennent de l'information historique riche, qui peut être utilisée pour obtenir une image claire du processus de l'évolution du logiciel.

En basant sur notre étude dans la zone de recherche de l'analyse d'évolution [CHE12, CHE11], nous suggérons que la majorité des techniques pour analyser les dépôts d'évolution suit le même principe. La Figure 3.3 montre les différentes tâches nécessaires pour analyser les dépôts du logiciel. Dans cette figure nous identifions trois étapes fondamentales pour l'analyse des données:

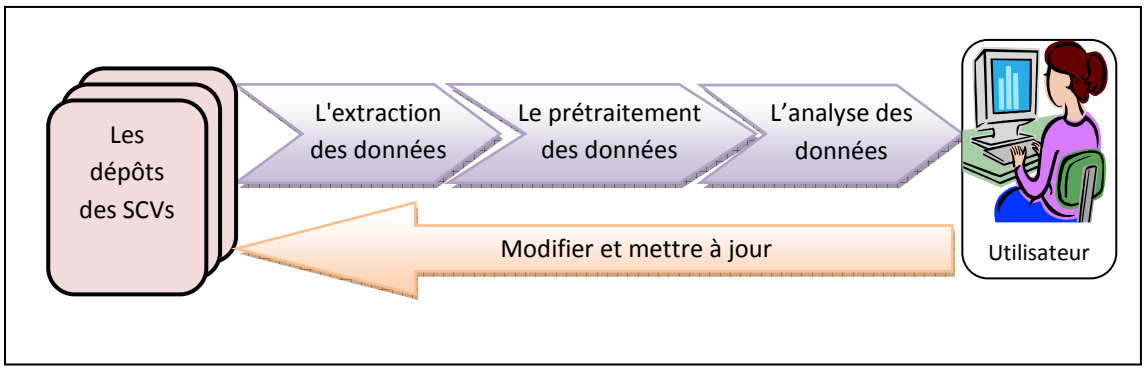


Figure 3.3. L'analyse de l'évolution du logiciel.

(1) L'extraction des données: La première étape nécessaire pour analyser l'évolution d'un système logiciel est d'avoir l'accès aux données entreposées dans les dépôts des SCVs. Bien que cette étape soit critique pour obtenir les bonnes données pour un traitement efficace, cette opération n'est pas supportée à un niveau complètement approprié dans la pratique. Par conséquent, l'extraction des données exige un effort considérable et elle est souvent spécifique à un système donné i.e. beaucoup de recherches visent des dépôts d'un SCV donné, à cause de sa grande popularité et disponibilité sur le marché.

(2) Le prétraitement des données: Les données obtenues dans la première étape doivent être traitées pour extraire des faits pertinents, et pour filtrer les grands ensembles des données. Ces derniers sont convenables comme entrée à une technique d'analyse d'évolution. En plus, des modèles de données peuvent être formulés pour être une base à toute analyse. Le prétraitement peut inclure l'analyse syntaxique des données (ex. le code source, les fichiers log, les rapports du bogue…etc.), l'application des techniques pour lier les différentes sources des données (ex. les artefacts d'un SCV avec les rapports de bogue), la reconstitution d'informations non

enregistrées (ex. la reconstruction des informations à partir des fichiers log).

(3) ***L'analyse des données:*** dans cette étape les données extraites et modélisées sont utilisées pour résoudre un problème d'évolution du logiciel, ou un ensemble de problèmes à travers différentes techniques et approches. Donc, nous pouvons appliquer une technique d'analyse appropriée pour extraire les méta-informations au sujet du logiciel ayant évolué. Par exemple, les entités du logiciel qui sont changées habituellement ensemble.

3.3.2 Les Approches d'Analyse de l'Evolution

Dans les années récentes beaucoup d'effort de recherche a été consacré à l'analyse de l'historique de l'évolution, en montrant l'importance croissante du domaine. Les différents travaux dans cette zone de recherche peuvent être classés dans les axes suivants:

La visualisation de l'historique de l'évolution: Il y a trois types de techniques de la visualisation. Le premier type comporte les techniques qui visualisent les métriques (l'âge du code source, le nombre de bogue…etc.) sur une représentation textuelle du logiciel [TAY02]. Le second type montre l'information structurelle [GAL99]. Le troisième ce sont des techniques qui extraient des patrons récurrents à partir de l'historique du logiciel en utilisant des techniques visuelles de Mining [RYS04, VOI07].

L'extraction des dépendances logiques: les dépendances logiques peuvent être découvertes en analysant l'historique de l'évolution d'un système logiciel. Par exemple, dans [RAT05] les auteurs exploitent des données historiques extraites à partir des dépôts et se concentrent sur les couplages du changement (change couplings). Ying et al. [YIN05] ont proposé une technique pour déterminer l'impact du changement en se basant sur les règles d'association. Dans [WON10], les auteurs formalisent le couplage logique comme un processus stochastique en utilisant un modèle de chaîne de Markov. Dans [ZIM05], le Data Mining est appliqué aux historiques de version pour extraire les patrons du changement. Ces derniers sont utilisés pour guider les programmeurs dans le développement et la

maintenance du logiciel. Weidl et Gall [WEI98] ont détecté les hot-spots possibles de maintenabilité en analysant les relations du Co-changement des modules qui pointent aux dépendances cachées, et aux points faibles de la structure d'un système logiciel.

Les langages d'extraction des données: d'autres travaux ont proposé des langages pour extraire les méta-informations à partir des dépôts du logiciel. Par exemple dans [HIN05] les auteurs proposent un modèle basé sur les graphes. Dans lequel les différentes entités entreposées dans le dépôt sont présentées comme des nœuds, et leurs relations sont des arcs. Ils définissent ensuite le SCQL, un langage de requête basé sur la logique temporelle de premier ordre pour interroger les dépôts du code source.

L'étude de la complexité du développement: dans [HAS03] en utilisant les concepts mathématiques sonores de la théorie de l'information (l'Entropie de Shannon), les auteurs étudient la complexité de processus du développement. Ils examinent les fichiers log du dépôt du code source pour les grands projets logiciel.

3.3.3 L'analyse de l'Evolution des Logiciels OA

Bien qu'il y ait un grand nombre de techniques de l'analyse d'évolution pour les différents paradigmes de programmation (procédural, orienté objet…etc.), peu d'effort a été consacré aux logiciels OA. Dans [QIA08], les auteurs proposent une approche pour extraire les patrons du changement dans l'évolution du logiciel AspectJ. Ils analysent premièrement les versions consécutives d'un programme AspectJ, puis ils décomposent leurs différences dans un ensemble de changements atomiques. Finalement, ils emploient l'algorithme Apriori de Data Mining pour générer les ensembles des items les plus fréquents. Ces patrons du changement peuvent être utilisés pour la prédiction de faute et l'analyse de l'impact du changement dans l'évolution du logiciel AspectJ.

Nous pensons qu'un dépôt riche et précis de l'historique d'évolution pour ces systèmes logiciels est la première étape vers une analyse efficace de leur évolution. La prochaine section présente les limites des SCVs actuels dans l'entreposage et le contrôle de l'évolution des logiciels OA.

3.4 La POA Versus les SCVs Actuels

Nous présentons dans cette section les différentes limites des SCVs actuels de façon générale. Ensuite, nous expliquons les conséquences de ces limites sur l'évolution des logiciels OA spécifiquement.

3.4.1 Les Limites des SCVs Actuels

Malgré que les dépôts du logiciel basé sur les SCVs actuels soient utiles pour gérer l'évolution d'un système logiciel, l'information qu'ils contiennent est limitée de plusieurs façons. Dans une étude antérieure [ROB05], les auteurs ont montré que la plupart des SCVs utilisés aujourd'hui perdent en effet beaucoup d'information au sujet du système logiciel. Donc, ils ne sont pas satisfaisants pour la recherche de l'évolution. Il y a trois points faibles qui ont des conséquences majeures, et sont la cause de la plupart des autres. La plupart des SCVs sont:

(1) basés-fichier à la place d'être basés-entité: Robbes et Lanza affirment que la vision tenue communément d'un logiciel comme un ensemble de fichiers, et son historique comme un ensemble de versions ne représente pas correctement le phénomène d'évolution du logiciel *"le développement du logiciel est un processus incrémental plus complexe que simplement l'écriture des lignes de texte"* [ROB05]. Donc, nous ne pouvons pas suivre l'évolution de chaque entité dans le logiciel i.e. la granularité de contrôle de version est une "ligne" et la différence est générée ligne par ligne. Par conséquent, l'analyse de l'évolution est plus difficile et elle n'est pas assez efficace pour la compréhension du programme ou la rétro-ingénierie.

(2) basés-instant (*Snapshot*), à la place d'être basés-changement: Le programme est entreposé comme une instance avec un tampon de temps particulier, sans enregistrer les changements réels qui arrivent entre deux instances successives. La séquence exacte de changements entre deux versions est dure à régénérer, en menant à une perte d'informations. L'ordre du temps de changements est perdu, et il ne peut pas être dérivé parfaitement. Pour la compréhension des changements, l'ordre du temps peut être important. De plus, l'ordre du temps est utile pour la détection de conflit et le fusionnement

[KOE09]. Les groupements de changements (changements composites) sont perdus. Par exemple, les opérations de Refactoring causent beaucoup de changements qui peuvent être groupés. Cela réduit le nombre de changements et le représente à un plus haut niveau d'abstraction. La dérivation des changements composites est difficile, et dans quelques cas même impossible dû aux problèmes de masquage [KOE09].

Ces inconvénients réduiront la capacité des utilisateurs pour comprendre le changement. En plus, la plupart des SCVs tiennent une copie pour chaque version du produit et pas les différences entre eux. Donc, un grand montant d'espace dans le dépôt est gaspillé.

Ainsi, la proposition d'un nouveau SCV où le changement est traité comme une entité de première classe peut être une source fondamentale pour l'analyse de l'évolution du logiciel. Beaucoup de travaux prouvent que l'évolution basée changement est plus satisfaisante que l'évolution basée état [HAT11, KOE09]. Dans [HAT11] les auteurs prouvent que pour les tâches qui ont eu besoin d'information à une fine granularité sur le changement, ou dans lesquelles l'ordre chronologique était important, un SCV basé changement est nécessaire pour remplacer les SCVs actuels.

(3) basés-code source, à la place d'être basés-modèle: Les SCVs classiques ne peuvent pas gérer les modèles comme une unité logique, parce que ces systèmes traitent principalement les codes sources. Ils ne peuvent pas manipuler des artefacts avec un modèle de données interne complexe i.e. la plupart des SCVs actuels sont basés sur les structures de système basés-fichier, pendant que les langages de modélisation sont basés sur des structures de plus haut niveau. La conversion de ces structures complexes utilisées par les langages de modélisation à des structures basées fichier est dangereuse dû à la divergence des concepts [MUR08]. Ces limitations incluent l'incapacité d'accomplir un versioning à une fine granularité des éléments d'un modèle, ou de fusionner régulièrement et de détecter des conflits parmi les éléments du modèle. Cependant, l'évolution du système logiciel est traitée généralement dans une représentation plus abstraite du code source (modèle conceptuel). Ce modèle aide pour faciliter les tâches de l'évolution (ex. l'analyse de l'impact du changement.). Ainsi, l'entreposage des

changements dans un format plus abstrait (le format de modélisation) est une idée intéressante pour améliorer la compréhension et le contrôle du changement.

3.4.2 L'Evolution du Logiciel OA et les Dépôts de Version Actuels

Les SCVs actuels causent beaucoup de problèmes comme présenté dans la section précédente. Cependant, le contrôle de version dans le développement du logiciel OA est plus complexe et problématique que pour les systèmes logiciels classiques. La POA améliore la qualité du code en prévenant le code enchevêtré et éparpillé. En même temps, en introduisant la quantification et l'inconscience [FIL01], elle rend son versioning plus difficile. Les systèmes de version actuels sont incapables de manier la nature transversale de la POA. Par conséquent, leurs dépôts ne sont pas une bonne source d'information pour une analyse efficace de l'évolution du logiciel OA.

La propriété d'inconscience de la POA implique que les développeurs des fonctionnalités du noyau ne doivent pas être informés du code anticipé ou conçu qui doit être touché par les aspects [FIL01]. Puisque, le corps d'une consigne est beaucoup plus comme le corps d'une méthode i.e. il encapsule la logique qui doit être exécutée quand on atteint un point de jointure spécifique. Comparées avec les méthodes des langages orientés objets traditionnels, les consignes ne sont pas appelés explicitement. L'exécution d'une consigne est déclenchée automatiquement quand le flux de contrôle atteint le point de jointure qui a été désigné. Par conséquent, les modules du programme dans lesquels les événements sur leurs flux de contrôles sont désignés, sont aussi inconscients aux consignes correspondantes. Cela restreint l'évolutivité du logiciel OA, et il rend son versioning plus difficile.

A cause des caractéristiques spécifiques à la POA, CVS, Subversion…etc., aucun de ces outils ne traite le besoin de contrôler l'évolution du logiciel OA. Ils n'ont jamais été adaptés complètement au paradigme OA i.e. les systèmes de version ne fonctionne pas bien avec l'inconscience et la quantification trouvées dans le code OA. Puisque les classes sont inconscientes aux aspects, ainsi, l'effet transversal d'aspects n'est pas suivi par le système de version [IFR12].

La plupart des systèmes de version actuels sont *basés-fichier*, à la place d'être *basé-entité*. Ils gèrent les révisions de programmes comme des documents texte organisés dans des fichiers. Cependant, la POA par sa nature défie ce principe [IFR12]. Premièrement, les préoccupations entrecoupent la structure du fichier. Donc, ce n'est pas possible de présenter et de suivre les effets de changements dans le code de base ou les aspects i.e. les systèmes de version sont associés à l'entreposage et la récupération de fichiers détissés et ils ignorent toute information du tissage (dépendances transversales). Deuxièmement, l'inconscience laisse certains effets transversaux non détectés dans l'affichage (textuel) de fichiers et de changements [IFR12].

Pour résumer, les systèmes de contrôle de version actuels n'arrivent pas à: gérer, entreposer, ou afficher l'information transversale. Donc, leurs dépôts ne sont pas assez complets pour une analyse efficace de l'évolution du logiciel OA.

3.5 Bilan

Le but de la recherche de l'évolution du logiciel est d'utiliser l'historique d'évolution d'un système logiciel pour analyser son état présent et prédire son futur développement. Donc, aussi bien que les modèles d'évolution, les systèmes de contrôle de version sont une partie indivisible de l'évolution du logiciel. Dans ce chapitre nous avons présenté le domaine de l'analyse de l'historique d'évolution en commençant par un état de l'art sur les systèmes de contrôle de version actuels, leur approches, principes, types…etc. Ensuite, nous avons étudié les approches et les techniques d'analyse d'évolution. Nous avons exploré l'analyse de l'historique des logiciels OA, où nous avons démontré que les SCVs actuels n'entreposent pas l'information complète au sujet de l'évolution des systèmes OA.

Nous pensons que pour une analyse efficace de l'évolution du logiciel OA, les éléments logiques dans un système logiciel tel que Classe, Aspect et Méthode…etc. devraient être les unités principales dans le contrôle de version. Cela peut aider pour suivre l'évolution de chaque entité dans le logiciel. Et par conséquent, il peut conserver les dépendances entre les entités du logiciel OA indépendamment des fichiers qui les contiennent.

Pour résumer, seulement un dépôt basé-changement permet de faire une analyse efficace sur l'évolution, depuis qu'il fournit toute l'information requise. En plus, prendre en charge le modèle d'évolution du système logiciel comme une unité logique pour le contrôle de version est une première étape vers un Framework efficace de l'évolution. Ce Framework basé-changement ouvre de nouvelles directions pour les développeurs et les chercheurs pour explorer et faire évoluer des systèmes logiciels complexes. Nous nous sommes basé sur cette idée dans notre thèse, où nous proposons alors un Framework d'évolution basé-changement pour les logiciels OA.

CHAPITRE 4

Vers un Framework d'Evolution du Logiciel Orienté Aspect basé sur la Transformation de Graphes

"The formal technique helps to make the problems explicit. It directs the developer to the problematic parts of a model. It helps in understanding aspect-oriented compositions and it helps in reasoning effectively about the crosscutting" Mehner et al. 2006 [MEH06]

Puisque tous les systèmes logiciels sont soumis à l'évolution, les systèmes orientés aspects (OA) aussi doivent évoluer. Donc, au-delà de l'identification et de la définition des préoccupations transversales, un problème supplémentaire survient: Comment traiter leur évolution? Ce chapitre présente l'idée globale de notre Framework d'évolution pour les logiciels OA. Nous utilisons la transformation algébrique de graphes comme un support pour notre Framework d'évolution proposé. Vue que la structure du logiciel peut être vue facilement comme un graphe, la transformation de graphes est impliquée pour spécifier comment ces graphes devraient être construits et devraient être interprétés, et comment ils évoluent dans le temps.

Dans ce chapitre nous présentons la transformation algébrique de graphes qui est la fondation théorique de notre travail, en spécifiant comment ce paradigme peut être utilisé pour supporter l'évolution du logiciel de façon général. Ensuite, nous donnons l'idée globale de notre Framework proposé. Ce Framework permet de modéliser et de valider l'évolution du logiciel OA, et de garder l'historique de cette évolution dans un dépôt formel précis. Ce dernier peut être analysé par la suite pour extraire des informations intéressantes au sujet de l'évolution du logiciel OA.

4.1 La Transformation Algébrique de Graphes

4.1.1 Les Grammaires de Graphes

L'approche algébrique des grammaires de graphes [EHR73] a été inventée à l'université technique de Berlin dans les premières années soixante-dix par Ehrig, Pfender et Schneider. Cette approche permet de généraliser les grammaires de Chomsky des chaines de caractères (String) aux graphes. L'idée principale était de généraliser la concaténation des chaines de caractères à une construction collant (gluing) pour les graphes i.e. différent de la grammaire de chaîne qui exprime des phrases dans des séquences de caractères, les grammaires de graphes sont convenables pour spécifier l'information visuelle dans un mode multidimensionnel. L'approche a été appelée "algébrique" parce que les graphes sont considérés comme des genres spéciaux d'algèbres, et le collage pour les graphes est défini par une "construction algébrique", appelée *pushout*, dans la catégorie de graphes et les morphismes total de graphes. Cette idée de base permet d'appliquer des résultats généraux à partir de l'algèbre et la théorie de la catégorie dans la théorie algébrique des grammaires de graphes [COR97].

Une grammaire de graphes permet de décrire de façon limitée une collection de graphes, i.e. ces graphes qui peuvent être obtenus d'un graphe initial à travers des applications répétées des productions de graphes. Formellement, une grammaire de graphes peut être définie comme suit:

Définition 1 (grammaire de graphes) [EHR73]**:** Une grammaire de graphes est une paire: GG = (G_0, P), G_0 est appelé le graphe initial et P est un ensemble de règles de production. L(GG) est l'ensemble de graphes qui peuvent être dérivés à partir de G_0 en appliquant les règles dans P.

Etant Donné que les graphes sont un moyen utile pour modéliser beaucoup de systèmes complexes, les grammaires de graphes était, dés le début motivées fortement par ses applications dans l'informatique. Les zones importantes d'applications incluent actuellement les systèmes logiciels appliqués et la modélisation des systèmes, beaucoup d'aspects de parallélisme, les systèmes distribués et les mécanismes de synchronisation, la sémantique opérationnelle de la programmation fonctionnelle (et

logique), la sémantique opérationnelle des langages orientés objet, et beaucoup d'autres aspects des langages de programmation.

La transformation de graphes (réécriture de graphes) [EHR06] est un calcul défini formellement basé sur la théorie des ensembles, l'algèbre ou la théorie de la catégorie. Elle généralise la notion des grammaires de graphes. Plus précisément, la transformation de graphes est décrite par l'application des productions de graphes qui modélisent les actions autorisées sur les graphes. Elle peut par exemple représenter les états d'un système logiciel ou les structures de données. De plus, la transformation de graphes définit une relation sur les graphes qui peuvent être itérés arbitrairement pour produire un processus de transformations. De cette manière un ensemble de productions obtient une sémantique opérationnelle [BAR97]. L'utilisation explicite des règles de réécriture de graphes offre plusieurs avantages. La réécriture de graphes fournit une représentation abstraite et de haut niveau d'une solution à un problème computationnel (elle est très significative pour la modélisation et la méta-modélisation dans le génie logiciel et les langages visuels). Aussi, les fondations théoriques de la réécriture de graphes facilitent la tâche de preuve des propriétés d'exactitude et de convergence.

4.1.2 Graphe coloré attribué et Graphe Type

L'idée principale de l'approche algébrique pour la réécriture de graphes est de donner une caractérisation algébrique abstraite des graphes colorés attribués [HEC02]. Un graphe coloré attribué peut être formalisé comme suit:

Définition 2 (graphe coloré attribué) [HEC02]: formellement un graphe coloré attribué, par exemple G, est représenté par six éléments:$G = \{N_G; A_G;\ s_G; t_G; m_1; m_2\}$. Ici, $N_G$dénote l'ensemble des nœuds, A_G dénote l'ensemble d'arcs; s_G est une projection topographique (mapping) qui fait correspondre les arcs à leurs sources et t_G les fait correspondre à leurs cibles. m_1 et m_2 sont des projections topographiques qui font correspondre les nœuds et les arcs dans le graphe aux alphabets fixes de couleurs de nœuds et d'arcs respectivement.

Les couleurs sont très importantes pour donner une description sémantique de l'élément (nœud, arc). Sans noms (couleurs) ce serait dur pour identifier les éléments et leurs corrélations.

Avant de transformer un graphe coloré attribué nous devons présenter le graphe type [COR96] du graphe proposé. Semblable à la définition d'un Méta modèle d'un langage visuel, un graphe type spécifie la structure générale dans l'approche des grammaires de graphes. Il garantit la consistance du graphe à chaque transformation. Ceci spécifie tous ce qui est nécessaire à un graphe pour être valide. Donc, ce graphe type spécifie comment créer un graphe coloré attribué *bien formé*. Tous les graphes instances doivent être conformes à ce graphe type. Les graphes sont typés sur leur graphe type par un morphisme de graphes (Définition 3) qui fait correspondre chaque élément à son élément type dans le graphe type, i.e. chaque nœud à un nœud et chaque arc à un arc dans le graphe type (Définition 4).

Définition 3 (Morphisme de graphes) [EHR06]: un morphisme de graphes f: $G \to G'$ est une paire de fonctions $f =< f_N : N \to N', f_E : E \to E' >$, qui préservent les sources et les cibles, tel que $f_N \circ s = s' \circ f_E$ et $f_N \circ t = t' \circ f_E$.

Définition 4 (Graphe type) [COR96]: un graphe type est un graphe distingué TG. Une paire $(G, type_G)$ d'un graphe G avec un morphisme de graphes $type_G : G \to TG$, est appelé un graph typé. Donné les graphes typés $G = (G, type_G)$ et $H = (H, type_H)$, un morphisme de graphes typé f est un morphisme de graphes $f : G \to H$, tel que $type_H \circ f = type_G$.

4.1.3 La Réécriture de Graphes

Une transformation de graphes est composée d'un ensemble de règles de réécriture, appelées productions. À partir de la Figure 4.1, une règle de réécriture de graphes est constituée de deux éléments $L \to R$. Où L est le côté gauche (Left Hand Side "LHS") de la règle est appelé le patron du graphe et R le côté droit (Right Hand Side "RHS") de la règle est le graphe de remplacement.

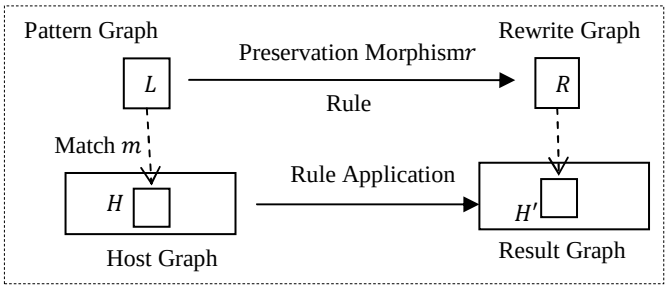


Figure 4.1. L'idée de base de la réécriture de graphes [GEI07].

De plus, on doit identifier les éléments du graphe (nœuds ou arcs) de L et R pour les conserver pendant la réécriture. Cela est fait par un morphisme (Définition 3) de conservation (preservation morphism) r qui faire correspondre des éléments de L à R.

La transformation est faite par l'application d'une règle à un graphe hôte (host graph). Pour faire ceci, on doit trouver une occurrence du patron de graphes dans le graphe hôte. Parler mathématiquement, un tel égal (match) m est un isomorphisme de L à un sous-graphe de H. Ce morphisme ne peut pas être unique, i.e. il peut y avoir plusieurs égaux.

Ensuite, on change le point d'égalité (matched spot) $m(L)$du graphe hôte, tel qu'il devient un sous-graphe isomorphique du graphe de remplacement R. Les éléments de L qui n'ont pas été fait correspondre par r sont supprimés de $m(L)$ pendant la réécriture. Les éléments de R qui n'ont pas d'image par r sont insérés dans H. Tous les autres (les éléments qui ont une correspondance par r) sont retenus. En d'autre terme, $L \cap R$ est la partie du graphe qui n'a été pas changée, et l'union $L \cup R$ devrait former un graphe encore. Le graphe $L \setminus (L \cap R)$ définit la partie qui sera supprimée, et le graphe $R \setminus (L \cap R)$ définit la partie qui doit être créée. Le résultat de ces étapes est le graphe résultant H'.

4.1.4 Les Conditions d'Application des Règles de Réécriture

Potentiellement, des contraintes supplémentaires comme «quand une règle doit ou ne devrait pas être appliquée» peuvent être spécifiées. Les conditions les plus intéressantes sont: les conditions d'attribut (Attribute Conditions «AC»), et les conditions d'application négatives (Negative Application Conditions «NACs»).

Un NAC est un patron du graphe qui ne doit pas être présent dans le graphe hôte pour que la règle soit considérée applicable. Par exemple, la Figure 4.2 présente une règle de réécriture qui ajoute un nouvel point de coupure "P" à l'aspect "A". Le NAC présenté dans le côté gauche de cette figure, est utilisé ici pour éviter l'existence d'un autre Pointcut avec le même nom dans l'aspect "A".

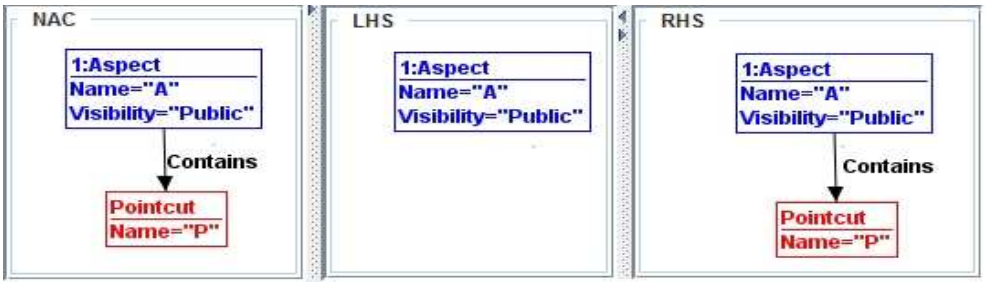


Figure 4.2. Créer un Aspect publique "A".

Note: dans une règle de réécriture, les libellés qui préfixent les nœuds et les arcs doivent être utilisés pour des buts d'identification, c.à.d. un nœud (ou arc) qui porte le même libellé dans LHS et RHS (et peut-être NAC) fait référence au même nœud (ou arc) dans le graphe hôte. Par exemple dans la Figure 4.2, le libellé 1 qui préfixe l'aspect A dans le LHS, RHS, et le NAC signifie qu'on fait référence au même aspect.

Les ACs sont des expressions Booléennes qui peuvent paraître pour exprimer des contraintes sur les valeurs d'attribut. Donc, l'évaluation d'une condition d'attribut est dépendante des instanciations de variables. En plus, les règles peuvent avoir des paramètres qui sont utiles pour déterminer par exemples les attributs de nouveaux objets du graphe spécifiés par l'utilisateur.

Un ensemble de règles de réécriture de graphes, avec un graphe type [COR96], est appelé un système de transformation de graphes (graph transformation system «GTS»). Une des facilités principales de l'analyse statique pour les GTSs est la vérification des conflits et des dépendances entre les règles et les transformations.

4.1.5 Les Approches de la Transformation de Graphes

L'idée de base dans l'approche algébrique pour la transformation de graphes est celle du collage (gluing) des graphes. L'action de collage de deux graphes est une construction, dans la catégorie des graphes et des morphismes de graphe appelée

pushout. Il y a deux approches principales de la réécriture algébrique de graphes: double-pushout et single-pushout.

L'approche double-pushout: Historiquement la première approche algébrique proposée dans les premières années soixante-dix par Ehrig, Pfender et Schneider [EHR73], et examinée par Ehrig [EHR79] est l'approche double-pushout. Le nom double-pushout a été donné à cette approche, puisque l'application d'une production à un graphe est définie via deux diagrammes du pushout dans la catégorie de graphes et des morphismes totaux de graphes. Le premier pushout modélise la suppression du côté gauche de la production à partir du graphe, et le deuxième présente l'addition du côté droit. Donné un graphe G, une production *p*: $L \overset{1}{\leftarrow} K \overset{2}{\rightarrow} R$ et un morphisme de graphes μ: $L \rightarrow G$:

- Premier pushout: supprime de G tous les éléments images d'éléments dans L mais pas d'éléments dans K.
- Deuxième pushout: ajoute au graphe tous les éléments qui n'ont pas de pré-image dans K.

L'approche Single-pushout: c'est une approche algébrique alternative pour la transformation de graphes [EHR97]. Dans cette approche, une étape de réécriture est définie par un seul diagramme de pushout dans la catégorie de graphes et des morphismes partiels. Le format d'une règle de réécriture est: $p : L \overset{r}{\rightarrow} R$ où r est un morphisme partiel de graphes.

La différence entre les deux approches est que l'approche double-pushout exige deux conditions supplémentaires pour une étape de réécriture: la condition de pendage (dangling) et la condition d'identification. Une règle de transformation ne s'applique pas si elle mène aux arcs pendants dans le graphe résultant. C'est une façon correcte de définir la transformation, depuis qu'aucune suppression n'est appliquée sans la déclarer explicitement. L'approche single-pushout n'exige pas de telles conditions; les arcs potentiellement pendants seront supprimés implicitement. Il a été prouvé que les règles de l'approche single-pushout peuvent modéliser plus de situations que celles de l'approche double-pushout.

4.1.6 Les Outils de la Transformation de Graphes

La transformation de graphes est une zone de recherche active, motivée par une grande gamme d'applications. Un outil basé sur la transformation de graphes (appelé aussi une machine de transformation de graphes) contient habituellement un éditeur graphique et un programme d'interprétation. L'éditeur graphique autorise respectivement la définition des productions de graphes qui consistent en un côté gauche et un côté droit (LHS et LRH respectivement), où les deux côtés sont représentés par des graphes. Le programme d'interprétation ensuite interprète les productions de graphes pour être appliquer par l'utilisateur sur le graphe existant.

Un éditeur supporte habituellement la représentation graphique des règles et du graphe. Une interface graphique est offerte pour afficher les différentes vues sur les graphes, comme des sections différentes d'un graphe, ou des niveaux différents d'abstraction. L'interprétation des règles de graphes peut être considérée comme une simulation par épreuve. Si la simulation fonctionne bien c'est concevable de générer un prototype à partir des règles définies [BAR97]. Différents outils de la transformation de graphes existent, nous pouvons citer parmi d'autres:

- ***PROGRES*** [SCH95]: est un environnement intégré pour un langage de programmation de plus haut niveau. Il a des sémantiques définies formellement basées sur les systèmes de réécriture de graphes programmés (PROgrammed Graph Rewriting Systems).
- ***AGG*** [LOW93]: est un langage visuel basé-règle qui supporte l'approche single-pushout pour la transformation de graphes. Il vise la spécification et l'implémentation prototypique d'applications avec des données structurées de graphes complexes.
- ***AToM^3*** [DEL02]: c'est un outil pour le multi-formalisme et la méta-modélisation. Il supporte la conception des langages visuels spécifiques aux domaines. Il autorise de définir la syntaxe abstraite et concrète du langage visuel au moyen de la méta-modélisation. Et il permet d'exprimer la manipulation de modèle au moyen de la transformation de graphes. Avec l'information du méta-modèle, AToM^3 génère un environnement de modélisation personnalisé pour le langage décrit.

4.2 La Transformation de Graphes comme Support de l'Evolution

4.2.1 Les Bénéfices d'une Fondation Formelle

Les approches de l'évolution du logiciel se préoccupent des changements du logiciel, leurs causes et leurs effets. Donc, le processus de l'évolution dépend de la complexité du logiciel, le nombre des entités du logiciel et les relations entre elles. Si le logiciel a beaucoup de relations (dépendances), le processus d'évolution est plus complexe, et l'analyse de l'impact du changement est plus difficile. Par conséquent, l'évolution du logiciel peut être traitée plus facilement si elle est supportée par des outils et des techniques efficaces. Pour développer des outils efficaces, le problème doit être compris et modélisé correctement. Cette modélisation est mieux formulée à travers une fondation formelle.

Une fondation formelle pour l'évolution du logiciel nous permet de donner une définition précise et non ambigüe des concepts impliqués dans l'évolution du logiciel. Par exemple, un artefact logiciel, un conflit d'évolution…etc. Le manque d'une fondation formelle peut conduire à une évolution incomplète ou inconsistante à différents niveaux [MEN99]. Par conséquent, une fondation formelle est bénéfique pour beaucoup de raisons, on peut citer parmi d'autres:

- Un modèle informel est souvent incomplet, donc les développeurs doivent deviner ou inventer les caractéristiques manquantes.
- les parties d'un modèle informel peuvent être ambiguës, en autorisant des interprétations différentes possibles.
- Nous pouvons utiliser les théorèmes et les résultats existants du formalisme formel pour prouver des propriétés intéressantes. Par exemple, l'exactitude et la consistance, l'équivalence entre différents modèles…etc.
- Un modèle formel facilite aussi le support d'outil. Par exemple, il peut être utilisé pour la vérification de modèles, pour la simulation de modèles, pour la génération automatique du code source…etc.

4.2.2 Le Logiciel Vu Comme un Graphe

Les graphes sont un formalisme intuitif, visuellement attirant, général et mathématiquement bien traité (la théorie de graphes). Ils comprennent un nombre énorme de concepts, de méthodes et d'algorithmes. Cela les rend très intéressants d'un point de vue formel. D'un point de vue pratique, les graphes sont aussi très utiles, depuis qu'ils sont souvent utilisés comme une représentation sous-jacente des artefacts logiciels et de leurs dépendances.

Les graphes sont des structures de données d'une grande importance pour de nombreuses applications de l'ingénierie du logiciel. Ils sont utilisés pour représenter des objets (concepts) complexes pour lesquels, les relations entre les composantes sont primordiales. Les nœuds représentent les artefacts logiciels et les arcs les différentes relations entre ces composants. Les graphes de flux de données, les graphes de flux 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. La représentation la plus communément employée même par les compilateurs est celle de l'AST (Abstract Syntax Tree), de ce fait le logiciel est vu au départ comme un graphe.

Par conséquent, l'analyse du logiciel et plus spécifiquement celle du code source, a conduit depuis longtemps à privilégier la structure de graphes comme moyen simple et adapté à la modélisation et à la manipulation des logiciels. Ainsi, la transformation de graphes semble la meilleure solution pour supporter l'évolution de ces logiciels.

4.2.3 L'Evolution basée sur la Transformation de Graphes

Depuis que les graphes sont utilisés pour représenter les différents artefacts du logiciel, la transformation de graphes est un choix naturel pour représenter l'évolution de ces artefacts. Comme les graphes, la zone de recherche de la transformation de graphes a une grande base mathématique solide. Fondamentalement, ce formalisme est convenable pour la modélisation de l'évolution du logiciel grâce aux caractéristiques suivantes [MEN99]:

— *Naturel*: la plupart des structures du logiciel peuvent être représentées avec les graphes.

— *Visuel*: basé sur les notations graphiques qui facilitent la compréhension du logiciel.
— *Déclaratif*: se concentre sur "quoi faire" et pas "Comment faire".
— *Intuitif*: aucune méthode standard pour suivre.
— *Haut niveau*: traite le niveau modèle.
— *Formel*: possède des fondations mathématiques basés sur la théorie de la catégorie. Il devient possible d'utiliser beaucoup de propriétés et de théorèmes qui ont déjà été prouvés pour la transformation de graphes.
— Supporté par des outils.

La modélisation de l'évolution du logiciel par la transformation de graphes signifie de prendre le graphe d'un système logiciel, et de le transformer selon certaines règles de transformation. Le résultat est le graphe du système logiciel évolué. Donc, les grammaires de graphes peuvent être utilisées comme un formalisme pour décrire d'un côté la structure des systèmes logiciels, et d'un autre coté pour décrire tous les genres d'opérations sur eux. Dans ce cas, la transformation de graphes définit l'évolution *dynamique* de ces structures. Un modèle d'évolution peut être défini précisément par un système de transformation de graphes (T, R) consistant en un graphe type T et un ensemble de règles de transformation R. Le graphe du programme logiciel peut être spécifié comme un graphe instance d'un graphe type T qui est le méta-modèle du langage de programmation de ce programme.

Tom Mens [MEN00] était l'un des premiers chercheurs ayant proposé d'utiliser le formalisme de la transformation de graphes pour modéliser et valider l'évolution du logiciel. Il a présenté une approche formelle basée sur la transformation de graphes pour gérer l'évolution du logiciel orienté objet. Plus spécifiquement, il a expliqué comment le formalisme des contrats de réutilisation pourrait être défini à travers la réécriture conditionnelle de graphes. Cela l'a rendu possible de traiter l'évolution d'artefacts du logiciel avec une manière intuitive et efficace.

Ensuite, l'utilisation de la transformation de graphes dans l'évolution du logiciel a été devenue une zone de recherche active. Ce formalisme a été appliqué dans plusieurs tâches d'évolution aussi. Par exemple, dans [RAJ97] les auteurs présentent un modèle pour la propagation du changement pendant la maintenance et l'évolution du logiciel.

Le formalisme pour ce processus est basé sur la réécriture de graphes. La propagation du changement est modélisée comme une séquence d'états, où chaque état représente un moment particulier dans le processus, avec quelques dépendances du logiciel qui sont logiques et d'autres qui sont contradictoires. Un état (graphe) est changé dans le prochain par un changement dans une entité logiciel et les dépendances en rapport avec elle.

La transformation de graphes a été appliquée aussi pour le refactoring du logiciel [BOT03, MEN02a, MENt07, PER10]. Ces différents travaux ont adopté généralement, un graphe type pour représenter les codes source et ont formalisé les refactorings comme des règles de réécriture. Le résultat a montré que les règles de réécriture de graphes pourraient spécifier la transformation du code source impliqué par un refactoring, et le formalisme garantit la conservation du comportement.

Le formalisme de la transformation de graphes n'a pas encore été appliqué pour supporter l'évolution du logiciel OA. Mais, il est utilisé principalement pour la vérification et la détection des conflits dans de tels programmes [ABM99, AKS09, ARI08, HAV07, MEH06, STA06]. La section suivante présente comment nous allons utiliser ce formalisme intéressant pour traiter l'évolution des logiciels OA.

4.3 Notre Framework d'Evolution du Logiciel Orienté Aspect

4.3.1 L'Idée de Base

Notre étude faite sur les modèles d'évolutions, nous a permis de constater que les modèles d'évolution et les dépôts de version peuvent être utilisés d'une manière complémentaire [CHE12, CHE10]. D'un côté, l'historique du logiciel peut être utilisé pour valider les modèles suggérés pour l'évolution. Et d'un autre côté, ils peuvent être utilisés pour les découvrir. Les approches d'évolution actuelles traitent séparément ces deux concepts. Par conséquent, l'intégration de ces deux concepts en un seul processus d'évolution est très intéressante pour bien gérer l'évolution du logiciel OA. Cette intégration est possible en traitant le changement du logiciel comme *une entité de première classe*. Le dépôt du changement produit par cette intégration est plus exact et contient une richesse d'information au sujet de l'évolution du logiciel OA. Il suffit juste de l'analyser et de découvrir l'information.

En se basant sur cette idée, nous proposons un Framework d'évolution complet pour gérer l'évolution du logiciel OA. Nous visons par "complet" que le Framework ne doit pas gérer l'évolution seulement, mais il doit garder son historique en même temps. Donc, un tel Framework d'évolution est utile pour les développeurs du système pour gérer l'évolution du code source OA. L'historique d'évolution peut être utilisé plus tard pour améliorer d'autres tâches d'analyse de l'évolution. Il peut être utilisé pour exécuter des évaluations sur les systèmes OA. Nous pouvons répondre à plusieurs questions intéressantes, telles que: lequel des aspects contient un point de coupure donné? quel est l'aspect qui change fréquemment (hotspot)? quelles sont les préoccupations couplées dans le système (couplage logique)? i.e. si on change une préoccupation (aspect) on doit changer l'autre, et ainsi de suite. Par conséquent, cela peut aider pour comprendre l'évolution du logiciel OA, prédire de futurs changements, identifier des fautes potentielles, détecter de nouvelles préoccupations transversales, et développer de nouveaux algorithmes de refactoring pour le code source OA.

4.3.2 Présentation Globale du Framework d'Evolution

L'évolution du logiciel OA peut être définie comme le processus de modification progressive des éléments du système OA pour améliorer ou maintenir sa qualité avec le temps. L'idée principale de notre proposition est de modéliser correctement comment ce logiciel évolue en utilisant un format abstrait et formel. Et d'entreposer cette évolution dans un dépôt basé changement. Nous utilisons la transformation de graphes (réécriture algébrique de graphes) comme une technique formelle pour donner une sémantique formelle à notre Framework d'évolution, et pour l'analyser rigoureusement.

L'analogie entre l'évolution du logiciel OA et la transformation de graphes est assez naturelle: un système logiciel OA peut être exprimé comme un graphe qui contient un ensemble de composants mises en corrélation par des connecteurs. Les transformations de graphes nous permettent d'exprimer l'évolution de ces graphes du logiciel d'une manière précise. De plus, ils permettent l'analyse formelle et le raisonnement au sujet de l'évolution du logiciel OA. La Figure 4.3 montre la présentation globale de notre Framework d'évolution, qui peut être divisée en trois composantes principales:

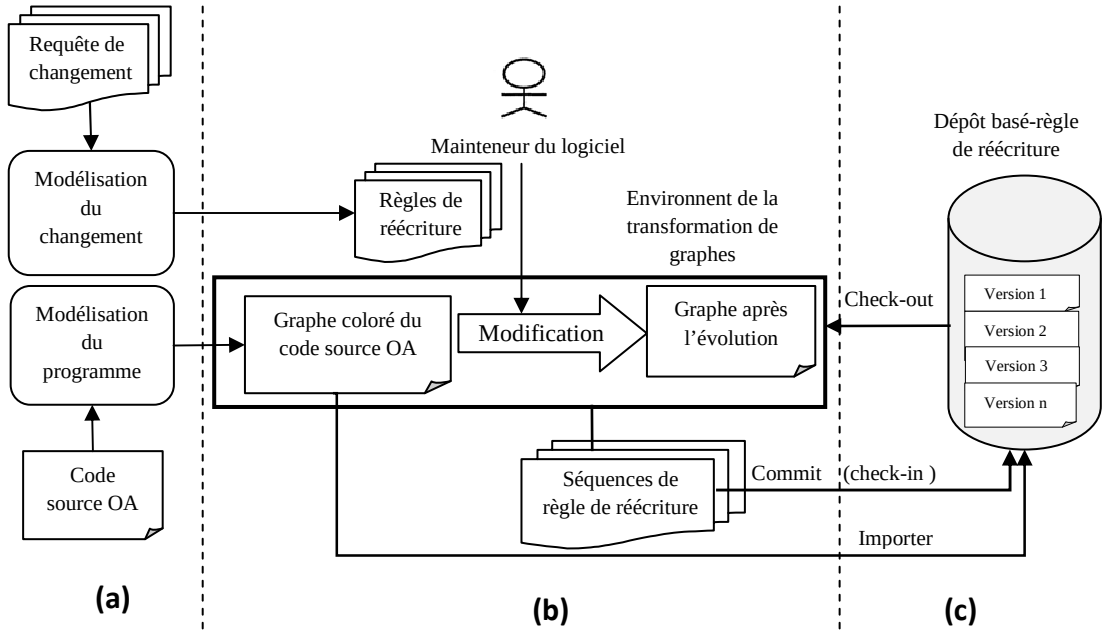


Figure 4.3. Le Framework d'évolution du logiciel OA.

(1) Abstraction: comme il été présenté dans la Figure 4.3.a, le code source OA évolué est modélisé par un graphe coloré attribué.

(2) Gestion du changement: les changements au logiciel sont formalisés comme des règles de réécriture qui transforment le graphe G en un graphe G', pour accomplir les requêtes d'évolution. Donc, le mainteneur du logiciel modifie le graphe coloré attribué du code source OA en appliquant des séquences de règles de réécriture dans un certain ordre (Figure 4.3.b).

(3) Gestion de l'historique: le graphe coloré attribué est importé vers le dépôt, et la version du logiciel est entreposée (checked-in) en enregistrant les séquences de règles de réécriture appliquées par le mainteneur. Donc, toute version peut être récupérée (checked-out) seulement en appliquant les séquences de règles en rapport sur le graphe coloré attribué du programme OA (Figure 4.3.c).

Dans notre approche, nous avons créé un dépôt du logiciel conçu pour entreposer une quantité maximale d'informations au sujet de l'évolution du logiciel OA. En particulier, nous n'avons pas utilisé un système de version, mais nous avons construit indépendamment un dépôt de logiciel basé-règle. Comparés avec les systèmes de version actuels, les changements au système logiciel sont entreposés directement dans le dépôt. Donc, on ne considère pas l'historique d'un système logiciel OA comme une

séquence de versions (versions de fichiers), mais comme la somme des changements qui ont apporté le système à son état actuel. La réalisation typique d'un changement du logiciel est une modification au code source OA. Donc, une nouvelle version est créée quand un changement du code source se produit.

L'utilisation de la réécriture de graphes aide pour bien suivre et contrôler les changements dans l'évolution du logiciel OA. Les changements sont présentés dans un format formel comme des règles de réécriture. Comparée avec le format texte du changement, la règle de réécriture est plus exacte et significative, parce qu'elle contient l'information complète au sujet du changement:

- **Où elle est appliquée:** la pré-condition de la règle présente les parties du graphe qui seraient sujet au changement i.e. les éléments changés et leurs entités en rapport;

- **Comment elle est appliquée**: la post-condition de la règle montre les parties modifiées du graphe après l'application de la règle;

- **Quand elle est appliquée**: les différentes conditions que nous pouvons formuler pour l'applicabilité de la règle (NAC, AC…etc.). Si une des conditions n'est pas vérifiée, la règle (changement) ne s'appliquera pas.

Ce format riche facilite la compréhension du changement (Où, Comment et Quand), et par la suite il facilite l'évolution du logiciel OA aussi.

4.4 Bilan

Dans ce chapitre, nous avons présenté le paradigme de la transformation de graphes. Nous avons examiné une variété de notations et de mécanismes qui ont été proposées pour la transformation de graphes. Ce dernier est un formalisme prometteur, bien traité théoriquement, avec la possibilité d'être pratiquement utile dans une variété de zones d'application.

En basant sur ce formalisme, nous avons proposé un Framework d'évolution pour le code source OA. Notre approche permet de modéliser le code source OA comme un graphe coloré attribué; en représentant les différentes entités du système et les relations entre elles. Les requêtes d'évolution sont formalisées comme des règles de

réécriture sur le graphe du système. En plus, nous avons proposé un dépôt basé-règle de réécriture pour entreposer l'historique de l'évolution en évitant les limites des dépôts actuels. Notre dépôt est basé-changement (spécifiquement basé-règle de réécriture); par conséquent nous pouvons reproduire chaque version du système, et nous pouvons suivre l'évolution de chaque entité du système à tout point dans le temps

La première étape évidente vers notre but pour représenter l'évolution du logiciel OA par la transformation de graphes, est l'introduction d'une représentation convenable du graphe pour les programmes OA. On doit décider quelles entités doivent être représentées par des nœuds, quelles relations doivent être représentées par des arcs, et quelle information est représentée par des propriétés de nœuds et d'arcs. Cela est discuté dans le chapitre suivant.

CHAPITRE 5

La Modélisation de l'Evolution des Logiciels Orientés Aspect

"The ability to simplify means to eliminate the unnecessary so that the necessary may speak." Hans Hoffmann (1880-1966)

Bien que l'ingénierie du logiciel Orienté Aspect (OA) soit le sujet d'une recherche intense. La programmation OA est une technologie mûre qui modularise les préoccupations transversales. Par ailleurs, elle produit de nouvelles dépendances; ce qui restreint l'évolutivité du système logiciel. Pour faire face aux différents types des dépendances dans un programme OA, nous proposons un modèle formelle pour son évolution. Dans ce chapitre, nous donnons la présentation de notre modèle d'évolution proposé pour le code source AspectJ. Nous présentons ici, les détails de notre approche aussi bien que son implémentation. Et, nous fournissons une évaluation empirique pour prouver l'efficacité de notre proposition.

5.1 La Modélisation du Programme

À la première étape, notre modèle vise à définir une représentation abstraite du code source AspectJ. Comparée avec la représentation texte du code source, l'abstraction donne une présentation plus compréhensible et globale de tous les éléments du logiciel et leurs relations, en éliminant les détails à une fine granularité. Cela facilite les tâches de l'évolution: l'analyse de l'impact du changement, la propagation du changement...etc.

Dans notre approche, le code source AspectJ est modélisé par un graphe coloré attribué [HEC02]. Ce dernier est généré directement à partir du code source AspectJ. Nous pensons que pour notre but, l'approche basée-graphe est très convenable. Vu l'acceptation large des représentations à travers des graphes dans la modélisation du logiciel, il paraît naturel et intéressant d'utiliser la réécriture de graphes comme une base pour notre modèle d'évolution.

Les graphes sont basés sur une fondation mathématique bien située "la théorie des graphes" [GOD01]. Cela les rend très intéressants d'un point de vue formel. D'un point de vue pratique, les graphes sont aussi très utiles, depuis qu'ils sont souvent utilisés comme une représentation sous-jacente des éléments du logiciel arbitrairement complexes et leurs corrélations [MEN01].

Pour spécifier formellement le code source AspectJ comme un graphe coloré attribué, nous devons présenter le Méta-modèle du graphe proposé. Ce Méta-modèle garantie la consistance du modèle (graphe) à chaque transformation. Ceci spécifie tous ce qui est nécessaire à un modèle pour être valide (bien formé). Nous utilisons par conséquent; un graphe type [COR96], qui joue le rôle d'un Méta-modèle. Ce graphe type (diagramme de classe), présenté dans la Figure 5.1, spécifie comment créer un graphe coloré attribué bien-formé d'un logiciel AspectJ. Tout code source AspectJ bien-formé peut être représenté comme un graphe qui se conforme à ce graphe type.

Nous considérons que ce graphe type est l'union de deux parties: le sous-graphe du code de base et les sous-graphes des Aspects. Ces deux parties sont liées avec les différents arcs de dépendance. Dans ce qui suit nous donnons les détails des différents éléments du graphe type proposé pour le langage AspectJ.

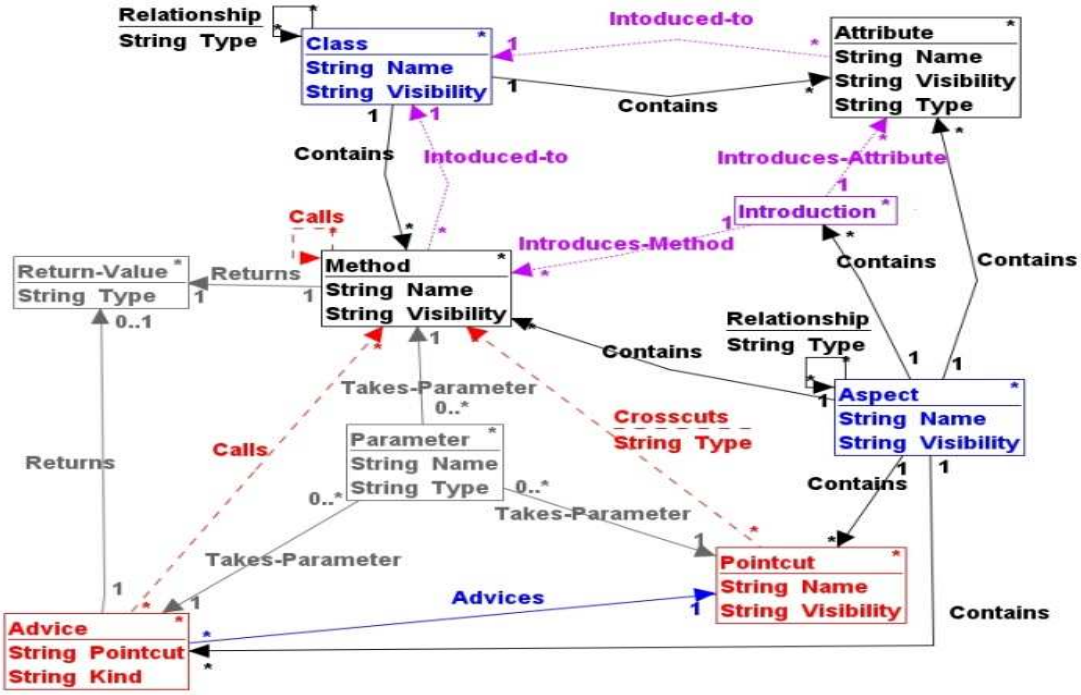


Figure 5.1. Le graphe type d'un programme AspectJ.

5.1.1 Le Sous-Graphe du Code de Base

Le code de base du code source AspectJ est un programme orienté objet. Ce dernier consiste en entités du code source et leurs dépendances. Dans notre représentation, les nœuds représentent les entités du programme et les arcs sont leurs dépendances. Un graphe coloré attribué possède une paire d'alphabets de couleur, une pour colorer les arcs et une pour colorer les nœuds. Les éléments de l'alphabet de couleur du nœud pour le code de base sont les suivants:

- **Class:** «Name», «Visibility» (c'est l'un de: *abstract*, *private*, *protected*, ou *public*).
- **Attribute:** «Name», «Visibility», «Type»
- **Method:** «Name», «Visibility»
- **Parameter:** «Name», «Type»
- **Return value:** «Type»

En outre, les arcs décrivent les relations (dépendances) entre ces entités. Ces arcs sont présentés dans le Tableau 5.1. Les relations peuvent être classées dans trois classes:

Tableau 5.1. Les couleurs des arcs pour le sous-graphe du code de base.

Nœud source	Nœud cible	Couleur de l'arc	Attribut
Class	Attribute	*Contains*	-
	Method	*Contains*	-
	Class	*Relationship*	Type
Parameter	Method	*Takes-parameter*	-
Method	Return value	*Returns*	-
	Method	*Calls*	-

— La première classe de relations représente quels attributs et méthodes appartiennent à une classe. C'est aussi important pour ce modèle de montrer les paramètres et les valeurs retournées que les méthodes prennent ou retour.

— La deuxième classe de relations montre la connexion entre les classes. Ici, la relation *Relationship* peut être une de: *Association*, *Aggregation*, *Generalization*, ou *Composition*.

— La troisième classe de relations capture les relations d'objet (*Calls*). Ce type de relation est important à inclure dans notre modèle parce qu'il spécifie quelles méthodes et paramètres sont touchés par les changements causés sur les demandes d'évolution.

5.1.2 Le Sous-Graphe d'Aspect

Un Aspect est une unité d'encapsulation. Il consiste aussi en des entités et des dépendances. Chaque aspect du système est aussi modélisé avec un sous-graphe coloré attribué. Ce dernier est semblable au sous-graphe qui modélise le programme du code de base, mais nous devons ajouter d'autres concepts propres au code source AspectJ. Les éléments de l'alphabet de couleur du nœud sont:

— **Aspect:** «Name», «Visibility»

— **Attribute:** «Name»**,** «Visibility», «Type»

— **Method:** «Name», «Visibility»

— **Parameter:** «Name», «Type»

— **Return value:** «Type »

— **Pointcut:** «Name», «Visibility»

— **Advice:** « Pointcut[1]», « Kind» (c'est l'un de: before / after returning / after throwing/after /around)

— **Introduction**

Le Tableau 5.2 regroupe les différentes dépendances dans le sous-graphe d'aspect. Pour chaque arc, nous définissons son nœud source/cible, aussi bien que la couleur utilisée et le ou les attributs nécessaires.

Tableau 5.2. Les couleurs des arcs pour le sous-graphe d'Aspect.

Nœud source	Nœud cible	Couleur de l'arc	Attribut
Aspect	Attribute	*Contains*	-
	Method	*Contains*	-
	Pointcut	*Contains*	-
	Advice	*Contains*	-
	Introduction	*Contains*	-
	Aspect	*Relationship*	*Type*
Method	Method	*Calls*	-
	Return value	*Returns*	-
Parameter	Method	*Takes-parameter*	-
	Pointcut	*Takes-parameter*	-
	Advice	*Takes-parameter*	-
Advice	Pointcut	*Advices*	-
	Advice around →Return value	*Returns*	-
	Method	*Calls*	-
Introduction	Attribute	*Introduces-Attribute*	-
	Method	*Introduces-Method*	-

5.1.3 La Modélisation du Système Globale

Pour modéliser le code source AspectJ, nous devons donner une vue globale des différentes entités du programme et leurs dépendances. Pour intégrer les différents sous-graphes qui représentent les classes et les aspects, nous avons besoin de trois types d'arcs, le Tableau 5.3 décrit ces arcs:

— *Crosscuts:* ce type d'arcs relie le point de coupure de l'aspect avec son point(s) de jointure. Ils doivent montrer l'information au sujet du type de point de jointure: *Method call/ Method execution*…etc.

[1]Le nom du Pointcut où les consignes sont exécutées

Tableau 5.3. Les arcs de dépendance.

Couleur de l'arc	Nœud source	Nœud cible	Attribut
Calls	Une méthode d'un Aspect	Une méthode d'une classe	-
	Advice	Une méthode d'une classe	-
Crosscuts	Pointcut	Une méthode d'une classe	Type
Introduced-to	Une méthode ou un attribut introduit par un Aspect	Classe	-

— *Introduced-to:* ce type d'arcs relie les méthodes ou les attributs introduits par l'aspect avec le ou les classes où ils ont été introduits.

— *Calls:* ces arcs présentent les appels entre les méthodes ou les consignes d'aspect aux méthodes du code de base i.e. une méthode (ou une consigne) d'un aspect peut utiliser les méthodes d'une classe pour exécuter un traitement spécifique.

En plus des contraintes entre les nœuds et les arcs présentées avant, le graphe type (Figure 5.1) exprime les contraintes suivantes sur les graphes instanciés:

— **Contraintes de multiplicité sur les arcs:** Par exemple, une classe contient zéro ou plusieurs attributs et méthodes, la méthode a une seule valeur retournée et la valeur retournée est mise en rapport avec une seule méthode…etc.

— **Contraintes de multiplicité sur les nœuds:** Dans le graphe type considéré, nous n'avons pas défini les contraintes de multiplicité sur les nœuds. Cependant, nous pouvons décider d'autoriser les graphes qui contiennent au moins un nœud du type *Aspect* (en attachant la cardinalité 1..* *à l'Aspect),* et ainsi de suite.

— **Contraintes d'attributs:** Les spécificités des composants du programme, comme les types sont représentées principalement sur les arcs et les nœuds par des attributs qui sont principalement du type chaîne de caractères (String).

Un code source AspectJ peut maintenant être spécifié formellement comme un graphe qui se conforme au graphe type, avec toutes ses contraintes du graphe. Un exemple est donné dans la Figure 5.2. Il montre un graphe qui représente le programme AspectJ "*UpdateDisplay*" du Listing 5.1. Ce programme change le moniteur pour mettre à jour l'affichage des objets. Il contient un aspect *UpdateDisplay* qui actualise l'affichage quand les objets bougent, et une classe *Point*.

Listing 5.1. Le programme UpdateDisplay.

```
Class Point {
private int _x = 0,
            _y = 0;
int getX(){
return _x;
}
int getY(){
return _y;
}
void setX(int x){
 _x = x;
}
void setY(int y){
_y = y;
}
}
```

```
aspectUpdateDisplay{
 public String Point.name ;
 public void Point.setName (String name){
  this.name = name;}
 public String Point.getName(){
  return name;}
 pointcut move():
  call(void Point.setX(int))||
  call(void Point.setY(int));
 before() : move() {
  System.out.println("figure is going to
  be displaced");}
 after(): move (){
  Display.update();}
}
```

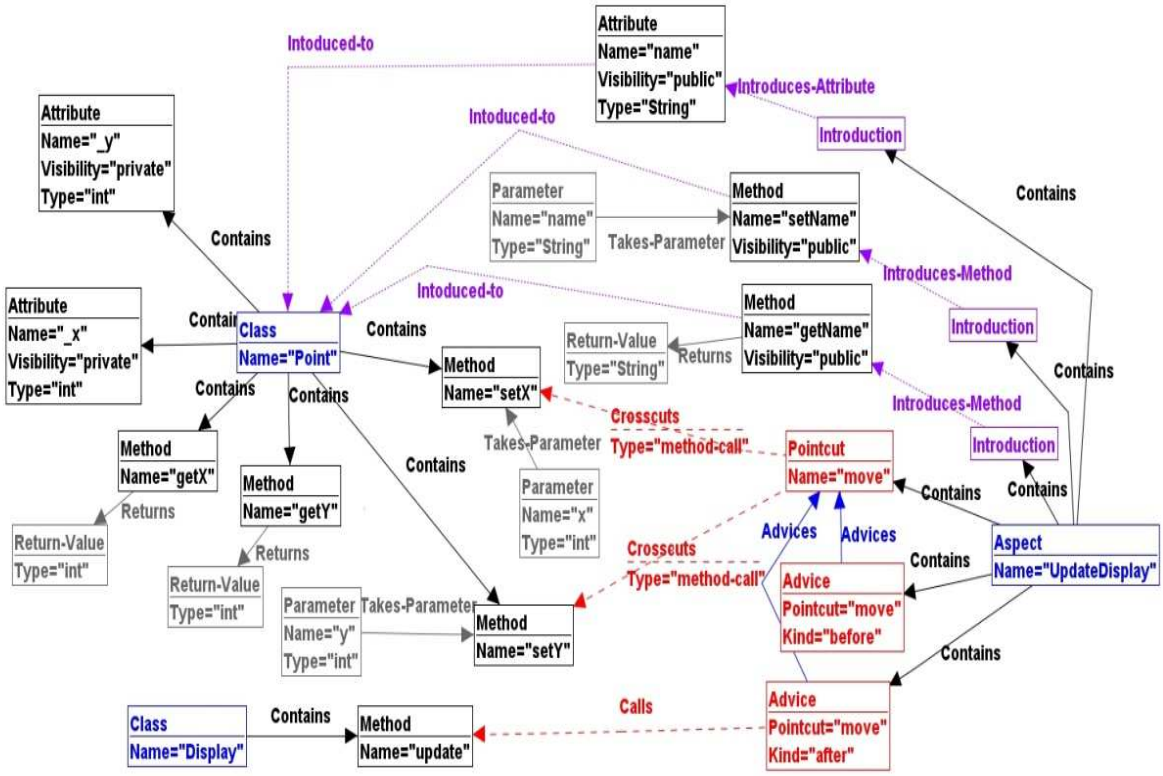


Figure 5.2. Le graphe coloré attribué du programme UpdateDisplay.

Dans cette figure nous pouvons constater trois sous-graphes: dans le côté gauche de la figure, on trouve le graphe coloré attribué de la classe *Point*, et dans le côté droit le graphe coloré attribué de l'Aspect *UpdateDisplay*. Les deux sous-graphes sont liés avec cinq arcs de dépendance: deux du type *crosscuts* et trois du type *introduced-to*. Au bas de la figure nous distinguons le sous-graphe de la classe "*Display*", sa méthode *Update* est appelée par l'advice *after move* (l'arc *Call*s).

5.1.4 Discussion

Nous devons noter ici que la représentation du code source AspectJ comme un seul graphe n'altère pas la modularité du code. C'est très important de conserver la séparation des préoccupations transversales pendant la modélisation du programme OA, ce qui est la motivation fondamentale derrière la POA. Notre proposition maintient une séparation stricte entre le code de base et les préoccupations transversales dans le modèle du code source proposé i.e. chaque préoccupation (Aspect) est modélisée comme un graphe coloré attribué. Le tissage est présenté comme des arcs de dépendance entre les sous-graphes du code de base et les sous-graphes des aspects. Ainsi, c'est très facile de distinguer le code de base et les préoccupations transversales (Aspects) du logiciel.

Si nous résumons, notre modèle proposé pour représenter le code source AspectJ conserve la modularité du paradigme OA, en même temps il décrit explicitement les dépendances entre les préoccupations transversales du système logiciel.

5.2 La Modélisation du Changement

L'objectif de notre travail n'est pas seulement de formaliser les opérations de l'évolution, mais d'automatiser leur application aussi. Puisque le code source AspectJ est formalisé graphiquement par un graphe coloré attribué, les requêtes d'évolution (changements) sont représentées comme des règles de réécriture. Dans cette section, nous présentons formellement comment le changement sur un graphe du code source OA doit être appliqué. Ensuite, nous expliquons comment de tels changements peuvent être appliqués automatiquement comme des règles de réécriture.

5.2.1 La Formalisation des Opérations d'Evolution

Nous pouvons distinguer deux types d'opérations du changement:

(1) Les opérations du changement atomiques: Ce sont les opérations d'évolution de base (changements) qui serviront comme des fondations à la création des opérations plus complexes. Les opérations du changement atomiques principales sont les opérations d'addition et de suppression des entités et des dépendances d'un programme OA. On note que l'opération de modification peut être modélisée par la

suppression de l'entité (ou de la dépendance) existante et l'ajout d'une nouvelle entité (dépendance).

(2) Les opérations du changement composites: les opérations du changement atomiques peuvent être combinées pour prendre en compte plusieurs requêtes d'évolution dans une manière consécutive (top-down) juste comme un algorithme. La combinaison de plusieurs opérations fondamentales sera capable de créer d'autres opérations d'évolution, ou à construire tout un processus d'évolution. Par exemple, une requête d'évolution qui exige le déplacement d'un point de coupure de l'aspect A à un aspect B consiste à le supprimer de A et l'ajouter à B. Nous devons en plus, déplacer le ou les consignes en rapport avec ce point de coupure à l'aspect B. Ces changements atomiques peuvent être groupés dans un seul changement «le déplacement d'un point de coupure».

Pour exprimer formellement les opérations d'évolution, nous utilisons les notations présentées dans le tableau 5.4.

Tableau 5.4. Notation formelle d'un programme orienté OA.

Notation	description
$\boldsymbol{P}$	Un programme OA.
$\boldsymbol{B}$	Un programme de base.
$\boldsymbol{A}$	Un Aspect.
$\boldsymbol{E_B}$	L'ensemble des entités du programme de base (classe, méthode…etc.).
$\boldsymbol{D_B}$	L'ensemble de dépendances entre les entités du programme de base. $D_B = \{D < a, b > \mid a, b \in E_B\}$, ici, une dépendance entre deux entités $a, b \in E_B$ est un couple étiqueté $D < a, b >$.
$\boldsymbol{E_A}$	L'ensemble des entités d'un aspect.
$\boldsymbol{D_A}$	L'ensemble de dépendances entre les entités d'un aspect. $D_A = \{D < a, b > \mid a, b \in E_A\}$, où, $D < a, b >$ est une dépendance entre deux entités a, b d'un aspect.
$\boldsymbol{E_P}$	L'ensemble d'entités d'un programme P, $E_P = \{E_B \cup E_{AS}\}$. C'est l'union des entités du programme de base et des aspects.
$\boldsymbol{E_{AS}}$	L'ensemble des entités de tous les aspects d'un programme. $E_{AS} = \bigcup_{i=1}^{n} E_{A_i} = \{E_{A_1} \cup E_{A_2} \cup \ldots \cup E_{A_n}\}$, où n est le nombre d'aspects dans le

	programme AspectJ.
$\boldsymbol{E_{JP}}$	L'ensemble des points de jointure d'un programme. $E_{JP} = \{e \mid e\ is\ a\ join\ point\}$. C'est un sous-ensemble des entités du programme de base, $E_{JP} \subseteq E_B$.
$\boldsymbol{D_C}$	L'ensemble de dépendances transversales (*crosscutting dependencies*) i.e. les dépendances entre les points de jointures (E_{JP}) et les entités des aspects du programme (E_{AS}), $D_C = \{ D < a, b > \mid a \in E_{AS}, b \in E_{JP}\}$.
$\boldsymbol{D_{call}}$	L'ensemble des dépendances d'appel (call) entre les entités des aspects et les entités du code de base i.e. les méthodes et les consignes d'un aspect peuvent faire appel aux méthodes du code de base (classes). $D_{call} = \{< a, b > \mid a \in E_{AS}, b \in E_M\}$, où E_M est l'ensemble des méthodes du code de base ($E_M \subset E_B$).
$\boldsymbol{D_P}$	L'ensemble de dépendances d'un programme P. C'est l'union de toutes les dépendances dans ce programme $D_P = \{D_B \cup D_{AS} \cup D_C \cup D_{call}\}$; où $D_{AS} = \bigcup_{i=1}^{n} D_{A_i} = \{D_{A_1} \cup D_{A_2} \cup ... \cup D_{A_n}\}$ (n est le nombre d'aspects dans le programme).

Les différentes opérations atomiques d'évolution peuvent être décrites comme suit, où P est un programme OA avant un changement, et P' le même programme après le changement:

L'opération d'addition d'une entité: pour l'addition d'une entité e à un programme P, nous avons trois possibilités:

- e est un Aspect
- e est une entité d'un aspect existant (A_i)
- e est une entité du code de base (B)

Les trois possibilités sont traitées dans l'algorithme suivant:

Listing 5.2. L'algorithme d'addition d'une entité.

```
Algorithm d'addition d'une entité (e: entité)
  Début
    si(e est un nouveau aspect)alors
      {
       céer(E_e) // l'ensemble des entités de e; E_e = ∅
       créer(D_e) // l'ensemble des dépendances de e; D_e = ∅
       }
       sinon
           // A_i est un aspect existant
           si(e est une entité de A_i)alors E_A_i' = {E_A_i ∪ {e}}
       sinon
           E_B' = {E_B ∪ {e}};
           E_p' = {E_p ∪ {e}}
Fin.
```

L'opération d'addition d'une dépendance: pour l'addition d'une dépendance $D < a, b >$ à un programme P, nous avons aussi trois possibilités:

- $D < a, b >$ est une dépendance dans un aspect existant A_i, où $a, b \in E_{A_i}$;
- $D < a, b >$ est une dépendance dans le code de base B , où $a, b \in E_B$;
- $D < a, b >$est une dépendance transversale où $a \in E_{A_j}, b \in E_B$ (A_j est un Aspect existant).

Les trois possibilités sont traitées dans l'algorithme suivant:

Listing 5.3 L'algorithme d'addition d'une dépendance.

```
Agorithm d'addition d'une dépendance (D < a, b > )
Début
   si(D < a, b > est une dépendance dans A_i)alors
     D_A_i' = {D_A_i ∪ {D < a, b >}}
   sinon
     si(D < a, b > est une dépendance dans B)alors
         D_B' = {D_B ∪ {D < a, b >}}
     sinon
       si (D < a, b > est une dépendence d'appel)
         D_call' = {D_call ∪ {D < a, b >}}
       sinon
         // D < a, b > est une dépendance transversale
         {
           E_JP' = {E_JP ∪ {b}}; //b va être un point de jointure
           D_C' = {D_C ∪ {D < a, b >}};
         }
   D_p' = {D_p ∪ {D < a, b >}}
Fin.
```

L'opération de suppression d'une entité: la suppression d'une entité (e) a les mêmes précédentes possibilités. Mais, la suppression d'une entité nécessite la suppression de ses dépendances aussi. On doit préserver la consistance du système, donc, chaque dépendance $D < a, b >$, où $a = e \; ou \; b = e$ doit être supprimée dans tous les ensembles de dépendance.

Si e est un aspect existant:

- Supprimer toutes les dépendances transversales reliées à l'aspect e
- Supprimer l'ensemble des entités de e (E_e)
- Supprimer l'ensemble des dépendances de e (D_e)

Si e est une entité d'un aspect existant A_i:

- Supprimer toutes les dépendances reliées à l'entité e dans l'Aspect A_i
- Supprimer toutes les dépendances transversales en rapport avec l'entité e
- Supprimer l'entité e à partir de l'ensemble des entités de l'aspect A_i (E_{A_i})

Si e est une entité du code de base (B)

- Supprimer toutes les dépendances en rapport avec l'entité e dans le code de base B.
- Supprimer toutes les dépendances transversales reliées avec l'entité e
- Supprimer l'entité e de l'ensemble des entités du code de base (E_B)

Ici, l'ensemble de dépendances transversales (D_C) est le plus intéressant; la suppression dans cet ensemble est exprimée dans la procédure SDT (Suppression de Dépendance Transversale) dans l'algorithme en Listing 5.4.

L'opération de suppression d'une dépendance: la suppression d'une dépendance $D < a, b >$ a les mêmes précédentes possibilités (l'opération d'addition d'une dépendance). L'algorithme qui simule cette opération est présenté dans le Listing 5.5.

Listing 5.4. L'algorithme de suppression d'une entité.

Algorithme de suppression d'une entité (e: *entité*)
// Supprimer les dépendances transversales en rapport avec l'entité e
Procedure SDT $(\boldsymbol{D} < a, b >, E)$
Début
si $((\boldsymbol{a} \in \boldsymbol{E}) \vee (\boldsymbol{b} \in \boldsymbol{E}))$ ***alors***
{
//b n'est plus un point de jointure
$\boldsymbol{E_{JP'}} = \{\boldsymbol{E_{JP}} - \{\boldsymbol{b}\}\}$;
$\boldsymbol{D_{C'}} = \{\boldsymbol{D_C} - \{\boldsymbol{D} < \boldsymbol{a}, \boldsymbol{b} >\}\}$;
}
Fin
Début
// e est un aspect existant
$si(e$ est un aspect$)alors$
{
// n est la cardinalité de D_C
$pour(i = 1, i \leq n, i++)faire$
$SDT(D_i < a, b >, E_e)$;
//n est la cardinalité de D_{call}
$pour(i = 1, i \leq n, i++)$ faire
$si((D_i < a, b > \in D_{call}) \wedge (a \in E_e))alors$
$D_{call'} = \{D_{call} - \{D < a, b >\}\}$;
//supprimer l'ensemble des entités de e
$supprimer(E_e)$;
//supprimer l'ensemble des dépendances dans e
$supprimer(D_e)$;
}
//si e est une entité d'un aspect existant A_i
$sinon\ si\ (e\ est\ une\ entité\ de\ A_i)alors$
{
//n est le nombre de dépendances dans A_i
$pour(j = 1, j \leq n, j++)faire$
$si\ ((D_J < a, b > \in D_{A_i})$
$\wedge ((a == e)\ \vee\ (b == e)))$
$alors\ D_{A_{i'}} = \{D_{A_i} - \{D_j < a, b >\}\}$;
//si e est été en rapport avec une dépendance d'appel
$pour(i = 1, i \leq n, i++)$ faire
{
$si((D_i < a, b > \in D_{call}) \wedge (a == e))$
$alors\ D_{call'} = \{D_{call} - \{D < a, b >\}\}$;
//si e été en rapport avec une dépendance transversale
$pour(i = 1, i \leq n, i++)faire$
$SDT(D_i < a, b >, \{e\})$;
$E_{A_{i'}} = \{E_{A_i} - \{e\}\}$;
}
//si e est une entité de B
$sinon\{$
//supprimer les dépendances en rapport avec e dans le code de base
$pour(j = 1, j \leq n, j++)faire$
$si\ ((D_J < a, \in D_B) \wedge \begin{pmatrix}(a == e)\\ \vee (b == e)\end{pmatrix})$
$alors\ D_{B'} = \{D_B - \{D_j < a, b >\}\}$;
//si e est été en rapport avec une dépendance d'appel
$pour(i = 1, i \leq n, i++)$ faire
$si((D_i < a, b > \in D_{call}) \wedge (b == e))$
$alors\ D_{call'} = \{D_{call} - \{D < a, b >\}\}$;
//si e est été en rapport avec une dépendance transversale
$pour(i = 1, i \leq n, i++)faire$
$SDT(D_i < a, b >, \{e\})$;
$E_{B'} = \{E_B - \{e\}\}$;
}
$E_{p'} = \{E_p - \{e\}\}$
Fin.

Listing 5.5.L'algorithme de suppression d'une dépendance.

Algorithme de suppression d'une dépendance $(D < a, b >)$
Début
$si(D < a, b >\ est\ une\ dépendance\ de\ A_i)\ alors$
$D_{A_i'} = (D_{A_i} - \{D < a, b >\})$
$sinon$
$si(D < a, b >\ est\ une\ dépendance\ de\ B)\ alors$
$D_{B'} = (D_B - \{D < a, b >\})$
$sinon$
$si(D < a, b >\ est\ une\ dépendance\ d'appel)\ alors$
$D_{call'} = \{D_{call} - \{D < a, b >\}\}$;
$sinon$
$si(D < a, b >\ est\ une\ dépendance\ transversale)$
$\{E_{JP'} = \{E_{JP} - \{b\}\}$;//$b$ n'est plus un point de jointure
$D_{C'} = \{D_C - \{D < a, b >\}\};\}$
$D_{p'} = \{D_p - \{D < a, b >\}\}$
Fin.

5.2.2 Les Opérations d'Evolution comme des Règles de Réécriture

Une règle de réécriture change graphiquement (automatiquement) les ensembles d'entités et des dépendances d'un programme pour le faire évoluer; où les entités sont considérées comme des nœuds, et les dépendances sont les arcs entre les entités du programme. Comme les graphes, la réécriture de graphes est très intuitive dans l'utilisation. Néanmoins, elle a une base théorique solide. Ces fondations théoriques de la réécriture de graphes peuvent aider à prouver les propriétés d'exactitude et de convergence de l'évolution du logiciel OA. Nous représentons des changements au programme comme des règles de réécriture explicites à son graphe coloré attribué. Quand une règle de réécriture du changement est appliquée elle prend comme entrée un état du programme et rend un état modifié (évolué) de ce programme. Les règles de réécriture atomiques sont:

- **La règle d'addition:** ajoute un nouveau nœud (arc) au graphe du programme;
- **La règle de suppression:** supprime un nœud existant et toutes ses dépendances, ou la suppression d'un arc seulement.

Les opérations composites du changement sont appliquées pratiquement par le concept de la séquence de règle de récriture (*rewrite rule sequence*). Cette dernière est l'union de toutes les règles formulées pour une requête d'évolution spécifique, appliquées dans un certain ordre juste comme un algorithme.

Nous pouvons donner une signification précise et claire pour les concepts du système de réécriture de graphes qui décrive notre modèle d'évolution d'AspectJ. Le Tableau 5.5 donne les concepts de la transformation de graphes vis-à-vis l'évolution du logiciel AspectJ.

Dans ce qui suit, nous présentons un scénario d'évolution pour le graphe dans la Figure 5.2 en utilisant l'outil AGG [LOW93]. Nous changerons le programme *UpdateDisplay* (Listing 5.1) pour effectuer les requêtes d'évolution suivantes:

Tableau 5.5. L'évolution du logiciel AspectJ comme un système de transformation de graphes.

Concepts de la transformation de graphes	**Concepts de l'évolution d'AspectJ**
Graph hôte	Graph coloré attribué d'AspectJ
Règle de réécriture	L'opération d'évolution présentant le changement qui doit être appliqué sur le logiciel.
LHS de la règle	Le sous-graphe qui a été en rapport avec la requête du changement i.e. le ou les entités qui ont été en rapport avec le changement et leurs relations.
RHS de la règle	Le sous-graphe qui présente le LHS après l'évolution. Il modifie le LHS pour satisfaire la requête du changement.
LHS ∩ RHS	La partie du graphe qui doit être inchangée; elle n'a pas été touchée par la requête d'évolution.
LHS\ (LHS∩RHS)	La partie du graphe qui doit être supprimée. Elle représente les éléments touchés par la requête d'évolution.
RHS\ (LHS∩RHS)	La partie du graphe qui sera créée. Les éléments changés après l'évolution.
Séquence de règle	L'union de toutes les règles formulées pour une requête d'évolution spécifique, appliquées dans un certain ordre.
System de Transformation de Graphes GTS = (G_0, R)	Un Processus de l'évolution d'un code source AspectJ, où: G_0 est appelé le graphe initial (le graphe coloré attribué d'AspectJ avant l'évolution); et R est un ensemble d'opérations de l'évolution (règles de réécriture du graphe).

a) Ne pas afficher un message avant l'appel des méthodes *setX* et *setY* (supprimer l'advice *before () move*;

b) Contrôler la valeur de x "si x >10 x=x-1" (ajouter un nouveau pointcut "*control*" pour capturer la méthode *setX*:

```
pointcut control(int x):
 call (void Point.setX(x));
int around (int x): control(x){
 if (x>10) x=x-1;
 return x;}
```

c) Changer le nom de l'Aspect "*UpdateDisplay*" à "*DisplayAndControl*" (modifier l'attribut "*Name*" de l'aspect.

Les règles de réécriture pour ces requêtes sont les suivantes:

(a) La règle de suppression: La Figure 5.3 représente la règle de suppression. Elle supprime les consignes (advices) exécutées avant le pointcut *move*. La suppression d'une entité implique la suppression de toutes leurs dépendances en rapport (Listing 5.4). Les arcs entre l'advice *before* et le pointcut *move* et l'aspect *UpdateDisplay* sont supprimés aussi). Dans cette règle nous avons:

- LHS∩RHS={ node (Pointcut *move*, Aspect *UpdateDisplay*), edge (*Contains*)};
- LHS\ (LHS∩RHS) = { node (Advice *Before*), edge (*Advices*, *Contains*) }
- RHS\ (LHS∩RHS) = {}

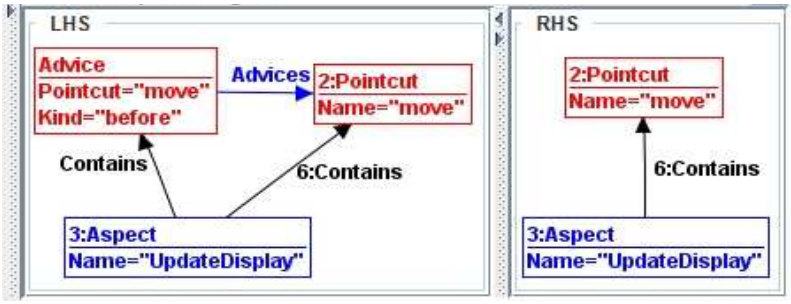


Figure 5.3. La règle de suppression de l'advice *before move*.

(b) La règle d'addition: la règle dans la Figure 5.4 ajoute le pointcut *control* qui a été déjà détaillé avant, à l'aspect *UpdateDisplay* "s'il n'existe pas déja". Cette condition est formulée en utilisant la condition d'application négative (NAC) représentée dans le côté gauche de la figure i.e. l'existence de deux Pointcuts avec le même nom dans le même Aspect est interdite. Le RHS de la règle montre que l'addition du pointcut implique l'addition de ses advices (advice *around* et sa valeur retournée). Les points de jointure sont spécifiés avec les arcs de type *crosscuts*. L'attribut Type de l'arc *crosscuts* spécifie que le point de jointure est de type *Method-call*.

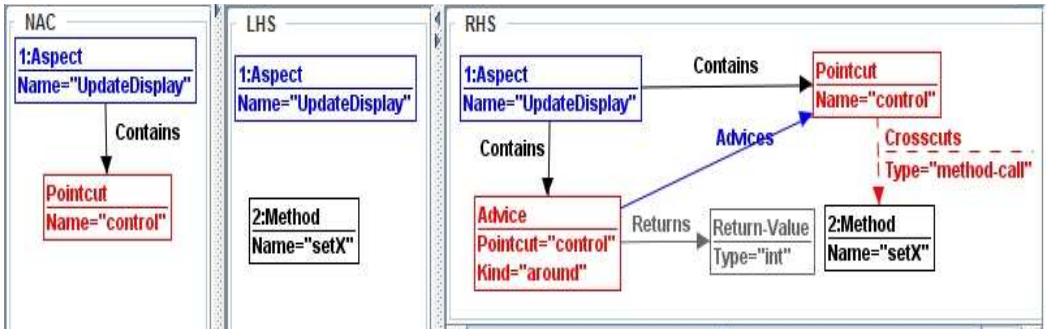


Figure 5.4. La règle d'ajout du pointcut control.

— LHS∩RHS= {node (Aspect *UpdateDisplay*, Method *SetX)*};

— LHS\ (LHS∩RHS) = {}

— RHS\ (LHS∩RHS) = {node (Pointcut *control*, Advice *Around, Return value)*, edge *(Contains, Contains, Crosscut, Advices, Returns)*}

(c) La règle de modification: la règle de modification est représentée dans le haut de la Figure 5.5. Cette règle substitue l'attribut *Name* existant de l'aspect par le nouvel attribut. Le NAC est utilisé ici pour éviter l'existence d'un autre aspect avec le même nom. Nous pouvons noter ici que la requête de modification est formulée comme la suppression de l'élément existant (nœud ou arc) et l'addition du nouveau.

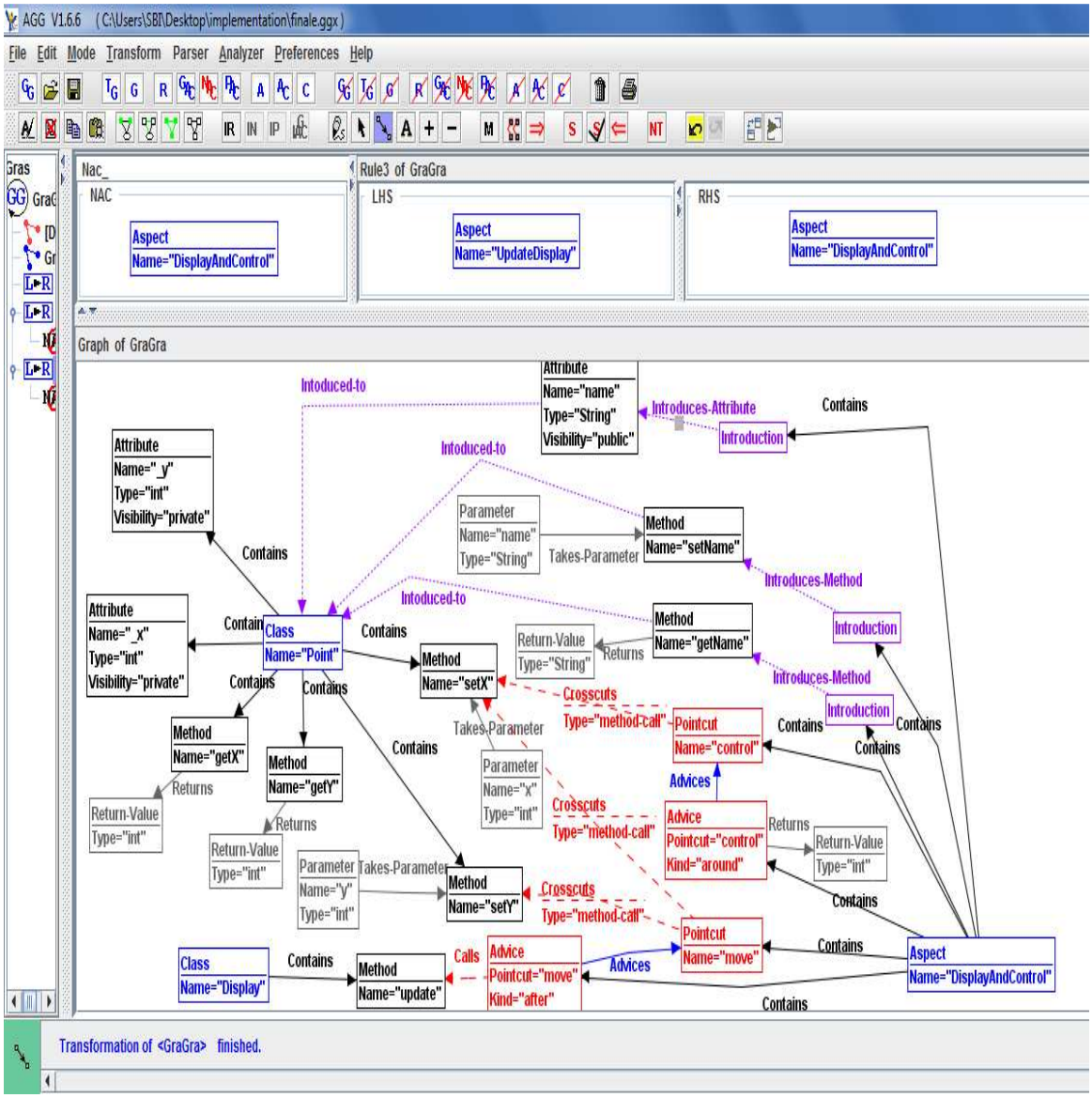


Figure 5.5. Le graphe *UpdateDisplay* après l'évolution.

— LHS∩RHS= {}

— LHS\ (LHS∩RHS) = {node (Aspect *UpdateDisplay*)}

— RHS\(LHS∩RHS)={node (Aspect *DisplayAndControl*)}

Après l'application de ces règles, la Figure 5.5 montre le graphe qui représente le programme *UpdateDisplay* après la transformation (l'évolution).

5.3 Mise en Œuvre du Modèle d'Evolution Proposé

Notre approche vise à construire une représentation plus abstraite du code source AspectJ comme un graphe coloré attribué (rétro-ingénierie). Les requêtes du changement sont formulées comme des règles de réécriture. Ces dernières sont appliquées au graphe AspectJ par un outil de transformation de graphes. Ainsi, le problème principal dans la validation de notre proposition est la conversion du code source AspectJ en un graphe coloré attribué. Nous avons complètement automatisé cette conversion grâce à l'outil présenté dans la sous-section suivante.

5.3.1 Présentation globale de l'outil

La Figure 5.6 montre la présentation globale de la validation de notre approche qui peut être résumée dans les parties suivantes:

1) L'outil de conversion (rétro-ingénierie): pour représenter le code source AspectJ comme un graphe coloré attribué, nous avons implémenté un outil de conversion. Ce dernier peut être divisé en trois sous-convertisseurs principaux:

- ***AspectJML***: C'est un outil open source existant proposé par Melo Junior et Mendonça [MEL05]. Il est un langage basé sur XML (*eXtended Markup Language*) [SUZ98] pour représenter un code source AspectJ. Le code source AspectJ est converti dans le format XML à travers la puissance d'AspectJML.

- ***le convertisseur XML-to-GXL:*** Nous avons implémenté ce convertisseur. Il convertit le document XML produit par AspectJML à un graphe GXL (*Graph Exchange Language*) [WIN01] en modélisant le code source AspectJ comme un graphe coloré attribué. Ce convertisseur est un document XSLT (*XML Stylesheet Language Transformation*) [CLA99]; où chaque élément AspectJ est traité par une Template XSLT spécifique.

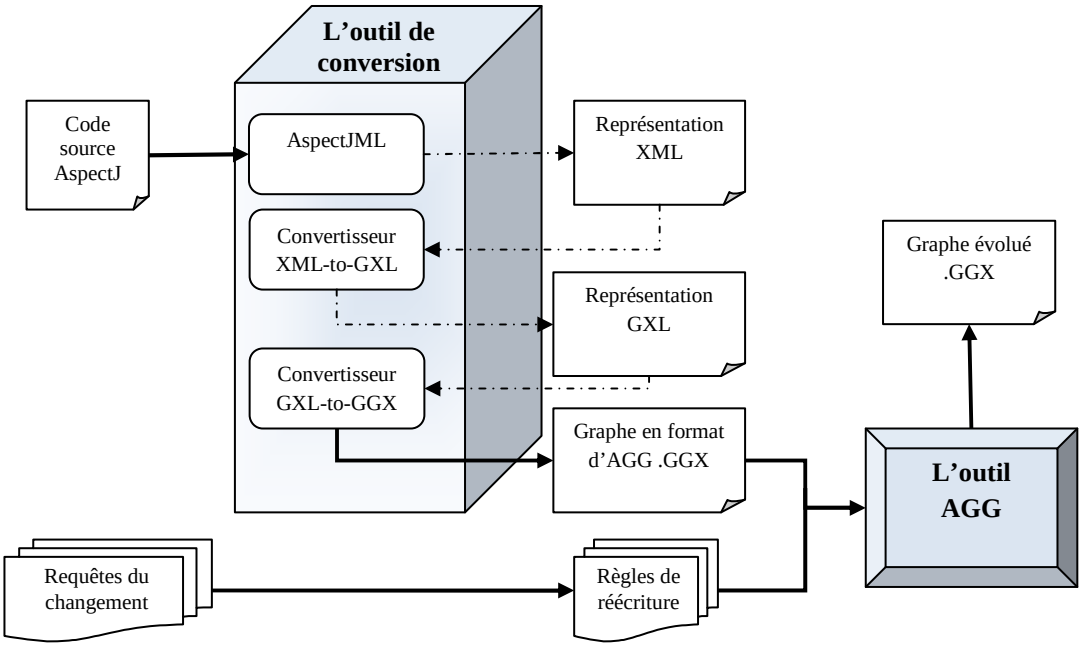


Figure 5.6. Mise en œuvre du modèle d'évolution.

La représentation basée-XML (ou GXL) fournit l'interchangeabilité d'information du modèle d'AspectJ entre plusieurs outils de développement tel que les outils CASE. Les outils logiciels utilisés habituellement emploient leurs propres formats pour décrire l'information du modèle (ex. les outils de visualisation). Un format d'application neutre permet à l'information du logiciel d'être interchangeable entre les outils de développement partout dans le cycle de vie de développement du logiciel. Une fois le code source AspectJ est converti à un graphe GXL, l'information du logiciel peut être réutilisable pour une grande gamme d'outils de développement différents. Cet interopérabilité souple de l'outil augmente notre productivité pour évoluer un code source AspectJ.

- ***le convertisseur GXL-to-GGX:*** Nous avons implémenté ce convertisseur pour convertir le document GXL produit par le XML-to-GXL convertisseur à un graphe GGX (Graph Grammare Xchange). Ce dernier est le format basé sur XML utilisé dans l'outil AGG [LOW93] pour représenter le graphe hôte. Ce format est un graphe GXL étendu avec des dispositions (layouts) pour les nœuds et les arcs. Ce convertisseur est aussi un document XSLT. Il est basé sur l'environnement de la transformation de graphes utilisé pour les

transformations. Ainsi, si nous voulons utiliser un autre environnement autrement qu'AGG, nous devons juste modifier le convertisseur pour générer le format du graphe approprié de cet environnement (en commençant à partir du graphe GXL).

2) L'outil AGG: Les requêtes du changement doivent être formalisées comme des règles de réécriture. Alors, nous utilisons un environnement de transformation de graphes pour appliquer ces règles sur le graphe coloré attribué du programme AspectJ. Dans notre validation nous avons utilisé l'outil AGG (Attributed Graph Grammar) [LOW93] qui est un outil puissant de transformation de graphes. Nous pouvons formuler des propriétés, des contraintes, analyser le graphe…etc. La transformation de graphe du logiciel produit une nouvelle version de celui-ci où toutes les requêtes du changement sont appliquées comme des règles de réécriture.

Note: Après la modification du graphe (l'évolution), nous pouvons régénérer le code source AspectJ en suivant la voie inverse: convertir le graphe GGX à GXL (par XSLT). Ce dernier doit être converti à XML par XSLT, ensuite à AspectJ par AspectJML. L'utilisation de XSLT est très intéressante pour notre proposition. Le traitement d'une Feuille de style XSLT est très rapide; nous pouvons générer le document GXL (ou GGX) d'une grande application dans moins de 10 secondes au moyenne. Ainsi, notre stratégie de validation est très efficace pour un petit code source AspectJ aussi bien qu'un grand code source.

5.3.2 Expérimentations

Pour évaluer la faisabilité et l'exactitude de notre approche, nous avons analysé des programmes AspectJ et nous les avons convertis à la représentation du graphe proposée en utilisant notre prototype. Notre étude a utilisé six programmes AspectJ montrés dans le Tableau 5.6. Cinq d'eux sont pris du package *Example* d'AspectJ. Nous avons choisi cette collection de programmes puisqu'ils ont été aussi utilisés comme des benchmarks par beaucoup de travaux de recherche comme des cas d'étude [DUF04, XUR07, XUR08]. Notre expérimentation inclut aussi un benchmark utilisé dans beaucoup de travaux de recherche "ProdLine[2] ".

[2] http://www.sable.mcgill.ca/benchmarks/.

Tableau 5.6. Les programmes expérimentés.

Programme	#Class	#Aspect	#Method	#Pointcut	#Advice	#Introduction
Telecom	10	3	39	6	6	3
Bean	2	1	16	1	2	1
Observer	6	2	9	1	1	11
Tjp	1	1	5	2	1	-
Introduction	1	3	13	-	-	6
ProdLine	9	11	10	15	15	84

Pour chaque programme le tableau donne le nombre d'aspects, de classes, de méthodes, de pointcuts, d'advices et d'introductions. Ces benchmarks nous donnent une variété de situations pour valider notre prototype. Par exemple, le benchmark Telecom contient 10 classes et 39 méthodes; ProdLine contient 11 aspects, 84 introductions, et 15 pointcuts, et ainsi de suite. L'applicabilité de notre approche à ces cas d'étude montre sa faisabilité dans la représentation et la validation de l'évolution du logiciel OA.

Nous avons vérifié les graphes colorés attribués (les documents GXL) générés par notre outil par une inspection manuelle du graphe et le code source analysé pour chacun des programmes susmentionnés. Nos expérimentations ont montré que les graphes générés par l'outil étaient corrects pour les plus petites benchmarks (ex. Tjp) aussi bien que les grands (ex. ProdLine).

Le temps d'exécution exigé pour la génération des graphes dépendent de la taille du code source AspectJ, aussi bien que le nombre de leurs modules (méthodes, pointcuts...). Le graphe dans la Figure 5.7 montre le temps d'exécution (en milliseconde) pour la génération des graphes GXL pour chaque Benchmark.

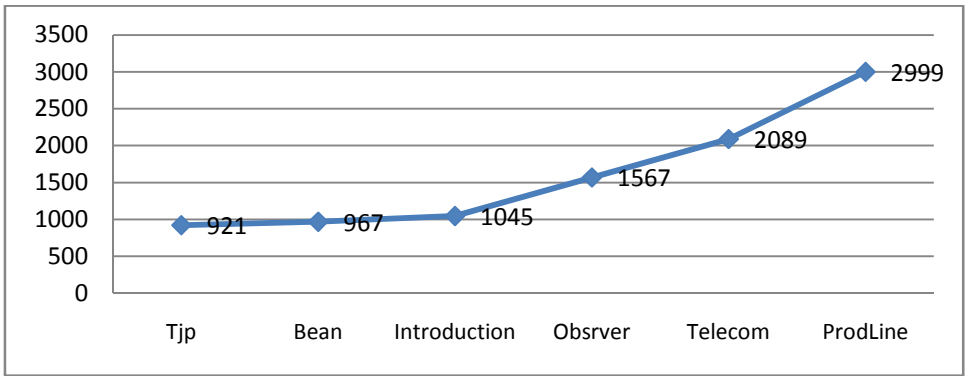


Figure 5.7. Le temps d'exécution en Milliseconde.

Ainsi, notre représentation du logiciel OA fournit un support utile pour gagner une meilleure connaissance de la structure interne de ces programmes compliqués, en réduisant l'effort exigé pour les obtenir dans une variété de tâches du génie logiciel, et surtout dans l'évolution du logiciel OA. Donc, nous pouvons utiliser notre prototype comme un outil de rétro-ingénierie du code source AspectJ.

5.4 Bilan

Nous avons proposé dans ce chapitre un modèle d'évolution pour le code source OA implémenté en AspectJ. Il est basé sur le formalisme de la réécriture algébrique de graphes qui lui donne une base formelle et une méthode d'implémentation automatique (en employant les outils de la réécriture de graphes). Notre approche commence par la rétro-ingénierie du code source en un graphe coloré attribué; en représentant les différentes entités du système et leurs dépendances. Les requêtes d'évolution sont formalisées en utilisant des règles de réécriture sur le graphe du système. Nous pouvons combiner plusieurs règles pour accomplir différentes requêtes d'évolution d'un système logiciel OA. Un outil prototype est construit et des cas d'étude sont expérimentés pour démontrer la faisabilité de notre approche.

Nous croyons que cette approche est assez générale pour être applicable à d'autres langages de programmation OA i.e. même si les langages OA peuvent exiger des genres spécifiques de nœuds et d'arcs, ils peuvent être tous exprimés en utilisant la même notation comme un graphe coloré attribué.

Pour bien traiter le phénomène d'évolution des programmes OA, l'historique de cette évolution doit être entreposé dans un dépôt dédié au paradigme OA. Cet historique peut être analysé par la suite pour comprendre l'évolution de ces systèmes, ainsi que pour prédire leur futur développement. Un tel dépôt est présenté dans le chapitre suivant.

CHAPITRE 6

Le Dépôt basé-Règles de Réécriture pour les Logiciels Orientés Aspect

> *"The fundamental unit of software evolution is the source code **change**, all other information is maintained to help understand, rationalize, and manage source code changes"* Kagdi et al, 2007 [KAG07]

Un succès durable d'une approche d'analyse de l'évolution dépend à une grande étendue du dépôt de version utilisé pour cette analyse. Notre intérêt de recherche est dans l'entreposage de l'évolution du logiciel Orienté Aspect (OA) et l'extraction des métadonnées à partir de son historique, pour faciliter son évolution et prédire son futur développement. Néanmoins, l'information évolutionnaire contenue dans les systèmes de version actuels pour les logiciels OA est incomplète et de qualité basse, ce qui limite les possibilités d'analyse de l'évolution de ces systèmes.

Dans ce chapitre, nous proposons un dépôt basé-règle de réécriture pour l'évolution du logiciel OA, et spécifiquement pour le langage de programmation AspectJ. Ce dépôt est consacré pour manier les caractéristiques propres du paradigme OA. Ensuite, nous analysons le dépôt proposé pour extraire des méta-informations intéressantes. Donc, à partir du dépôt proposé, nous présentons une approche pour la détection des patrons du changement, ainsi que pour l'extraction des relations logiques dans un code source AspectJ. Finalement, nous expliquons comment les telles relations logiques peuvent être utilisées pour détecter les défauts de modularité dans un programme AspectJ.

6.1 Le Dépôt basé-Règle de Réécriture

Un dépôt riche de l'évolution peut être le sujet d'une analyse efficace de l'évolution du logiciel OA. Par exemple, l'extraction des patrons de changement ou la découverte du couplage logique entre les entités du logiciel OA. Cependant, la zone de recherche du versioning pour les logiciels OA reste très limitée. Donc, il n'existe pas un dépôt convenable à l'évolution de ces logiciels i.e. il n'entrepose pas les changements des dépendances transversales dans le logiciel OA (voir la section 3.4). En se basant sur ces limites, nous proposons un dépôt basé-règle pour l'évolution du code source OA; où le changement est traité comme une entité de première classe.

6.1.1 Principe

Dans notre approche, nous ne considérons pas l'historique d'un système logiciel OA comme une séquence de versions, mais comme la somme des changements qui ont apporté le système à son état actuel. Donc, le principe de notre dépôt proposé est le suivant (Figure 6.1):

— Les opérations de changement qui servent pour naviguer entre les différentes versions du logiciel sont considérées comme des règles de réécriture sur le graphe du code source OA. La séquence des règles de réécriture qu'un développeur exécute est acquise dans le temps réel pendant son application en utilisant l'environnement de la transformation de graphes.

— Au lieu d'entreposer le graphe changé entier comme une version, on entrepose seulement les séquences de l'évolution (l'ensemble des règles de réécriture) sur ce graphe (Figure 6.1.a). Pour extraire et entreposer efficacement les changements, nous nous concentrons sur les activités du développeur qui transforment le graphe du code source en utilisant l'environnement de transformation de graphes. En d'autres termes, les changements sont entreposés directement quand les règles de réécriture sont appliquées.

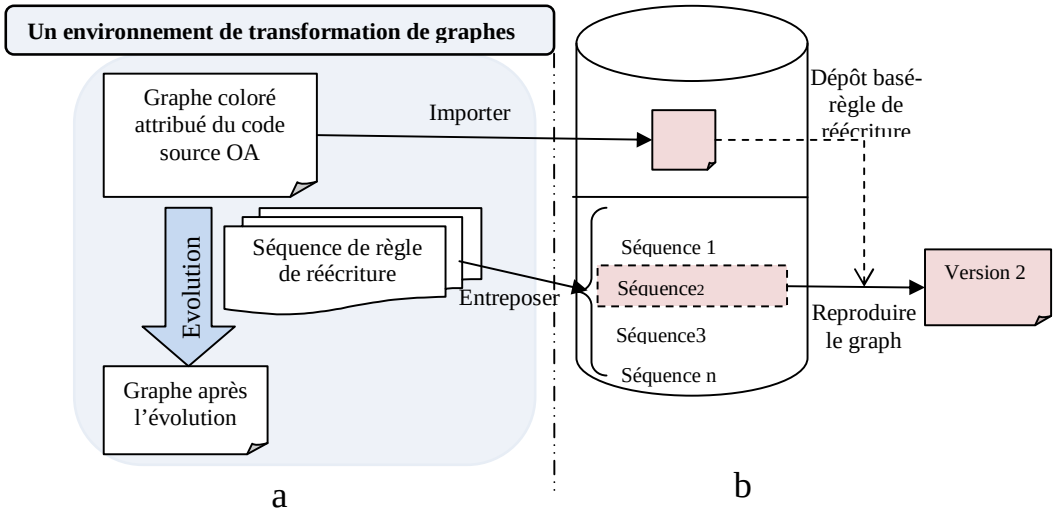


Figure 6.1. Principe du dépôt basé-règle de réécriture.

Supposons qu'un développeur crée un nouvel Aspect, l'exécution de la règle de réécriture "Crée l'Aspect A" déclenche l'entreposage de cette règle dans le dépôt basé-règle proposé.

- On peut reproduire chaque version du système par l'exécution de la séquence de règles de réécriture associée. Par exemple, dans la Figure 6.1.b si la version 2 est produite par l'application de la séquence 2 sur le graphe du logiciel. Pour extraire la version 2 du système, nous appliquons seulement cette séquence de règle de réécriture sur le graphe du système.

La Figure 6.2 montre la présentation de notre dépôt proposé. Au lieu d'entreposer tout le graphe changé comme une version, nous enregistrons seulement les séquences de règle de réécriture appliquées sur ce graphe. Donc, nous pouvons définir une Version comme: un groupe de séquences de règle de réécriture appliquées au graphe AspectJ formulant une requête d'évolution donnée.

Les changements sont entreposés dans un format formel comme des règles de réécriture qui rend le dépôt plus riche et fiable. Comparée avec le format texte du changement, la règle de réécriture est plus significative, parce qu'elle contient l'information complète au sujet du changement: pré-condition, post-condition, contraintes, actions…etc.

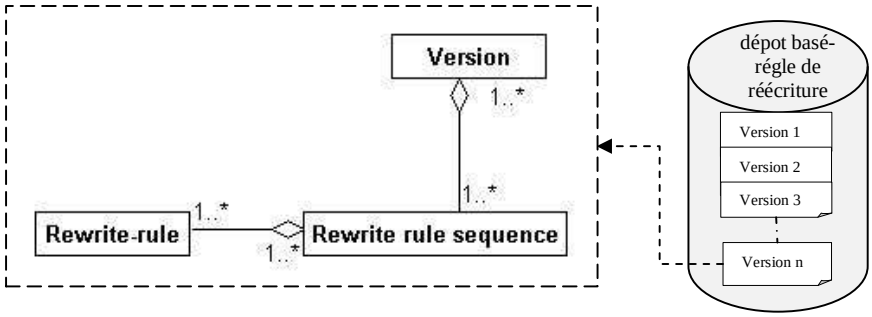


Figure 6.2. Le dépôt basé-règle de réécriture.

Cette information complète facilite la compréhension du changement, et par la suite l'entreposage du changement dans ce format rend le dépôt plus exact pour une analyse d'évolution de haute qualité.

6.1.2 Caractéristiques du Dépôt proposé

Comme présenté avant, chaque règle de réécriture est auto-explicative, elle contient autant d'information que possible pour formuler le changement et contrôler son application. Ce format va nous aider pour manier la nature transversale de la POA. La représentation et l'entreposage du changement comme des règles de réécriture peuvent rendre les dépendances du logiciel OA plus visibles dans le dépôt i.e. les effets d'inconscience dans le code source OA peuvent être entreposés explicitement.

Notre dépôt n'est pas basé-fichier mais il est basé-changement. Donc, il enlève les limites entre les fichiers du logiciel et entrepose le changement dans son format naturel i.e. il n'entrepose pas les changements comme des lignes de code seulement, mais comme des changements des entités du logiciel et de leurs dépendances.

Par exemple, si nous supprimons un point de coupure, nous devons supprimer leurs dépendances aussi. La règle de réécriture formulant ce changement est présentée dans la Figure 6.3. Ici, nous pouvons voir que la dépendance transversale entre le point de coupure P et la Méthode M est aussi supprimée.

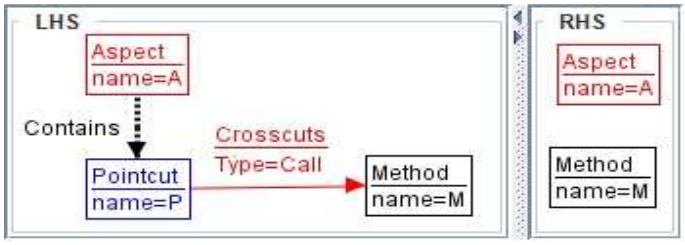


Figure 6.3. Suppression du pointcut P.

La Figure 6.3 a encore prouvé que trois entités qui appartiennent aux différents fichiers (la méthode M appartient au fichier de la classe C, l'aspect A et le point de coupure P appartiennent au fichier de l'aspect A) sont présentées et sont entreposés comme parties d'un seul changement, ce qui n'est pas possible dans les dépôts des systèmes de contrôle de version traditionnels.

Nous avons proposés deux approches d'analyse de l'évolution des programmes OA en se basant sur notre dépôt basé-règle de réécriture. Ces approches vont être présentées dans les sections suivantes.

6.2 Première Approche de l'Analyse d'Evoluion des Programmes OA: La Détection des Patrons de Changement d'un Programme AspectJ

La compréhension de comment les programmes évoluent ou comment ils continuent à changer est un besoin clé avant d'entreprendre toute tâche dans le génie logiciel ou l'évolution du logiciel. L'extraction des patrons de changement est importante pendant l'évolution et la maintenance parce qu'ils fournissent le support nécessaire aux mainteneurs pour emporter des changements complets et logiques [BOU06]. Nous présentons dans cette section une approche de détection des patrons de changement pour le code source AspectJ à partir de notre dépôt basé-règle proposé précédemment. Nous définissons les patrons du changement pour répondre à la question suivante: donné un système logiciel AspectJ et un changement spécifique à appliquer, quels autres changements doivent s'appliquer au système pour rester cohérent?

6.2.1 L'Extraction du Changement

La plupart des approches de détection des patrons de changement utilisent des outils et des techniques sophistiqués pour analyser le dépôt de version. Ces techniques essaient d'extraire une représentation convenable des changements pour être l'entrée d'un algorithme spécifique de Mining. Cela est fait par la différentiation entre les versions consécutives entreposées dans le dépôt i.e. la différenciation de version (*version differencing*) [MEN02b].

Les deux étapes principales de ce processus sont: l'identification des groupes de changement atomique, et le groupage de ceux-ci comme des transactions. Le problème de trouver tous les changements atomiques et ensuite les différentes transactions n'est pas simple, parce que la performance peut être exponentielle par rapport au nombre de versions (le dépôt d'évolution). De cette façon, il exige plus d'effort; c'est une tâche coûteuse en terme de performance et d'espace mémoire. Elle représente approximativement 58% de temps d'exécution [TAE12]. Les chercheurs sont plus intéressés à gagner un accès commode aux données extraites dans un format facile à traiter [HAS08]. Donc, ça sera très intéressant d'éviter cette étape pour mieux améliorer l'extraction des patrons de changement (l'analyse d'évolution).

Dans notre contexte (dépôt basé-règle), les changements sont entreposés dans le dépôt quand ils se produisent, ce qu'élève le changement à un concept de première classe. Il n'y a aucun besoin pour la différenciation depuis que les changements sont enregistrés et entreposés, et donc ne doivent pas être dérivés plus tard. L'entreposage du changement est, en général, plus précis et potentiellement permet de rassembler plus d'information que la différenciation de versions. Comparées avec la différenciation de version, les séquences de changement entreposées incluent toutes les changements intermédiaires. En plus, la différenciation de version ne comprend pas un ordre de changements appliqués, ce qui est cependant, habituellement le cas avec les changements entreposés.

Donc, en utilisant notre dépôt basé-règle, la différenciation de version qui est la tâche la plus coûteuse et difficile dans l'analyse de l'évolution n'est pas exigée et elle est totalement *omise*.

Le Tableau 6.1 donne les concepts utilisés dans n'importe quelle technique d'extraction des patrons de changement, pour les approches traditionnelles. Et, il explique la présentation de ces concepts dans notre contexte. Ces concepts seront détaillés dans les prochaines sous-sections.

Tableau 6.1. Notre approche versus les approches traditionnelles.

Concept	Approches traditionnelles	Notre approche
Dépôt	Versions de fichiers du code source	Versions de séquences de règle de réécriture
Changements	Changements dans les lignes du code source.	Changements dans les entités du logiciel et leurs dépendances (règles de réécriture)
Changements atomiques	Addition, suppression, modification d'éléments du code source.	Création, suppression d'éléments du graphe (nœuds / arcs).
Transaction	L'ensemble de changements atomiques pour une requête spécifique du changement.	La séquence de règle de réécriture qui formule une requête spécifique du changement.
Patron de changement	Changements atomiques qui arrivent fréquemment parmi les transactions du changement atomique.	Opérations de graphe (création/suppression d'élément) qui sont assez dupliquées parmi les séquences de règle de réécriture.

6.2.2 Les Changements Atomiques

Dans notre proposition, nous représentons le changement comme une ou plusieurs règles de réécriture. D'après la définition d'une règle de réécriture, toute règle peut être divisée facilement en un ensemble de création et/ou de suppression d'éléments du code source (graphe). Par conséquent, chaque règle consiste en opérations atomiques i.e. création ou suppression d'éléments (nœuds) ou de dépendances (arcs). Par exemple, si nous décrivons la règle de réécriture dans la Figure 6.4, nous distinguons les changements atomiques suivants: suppression de la dépendance entre A et B, suppression du nœud B, création du nœud F, création du nœud E, et création d'une dépendance entre F et E.

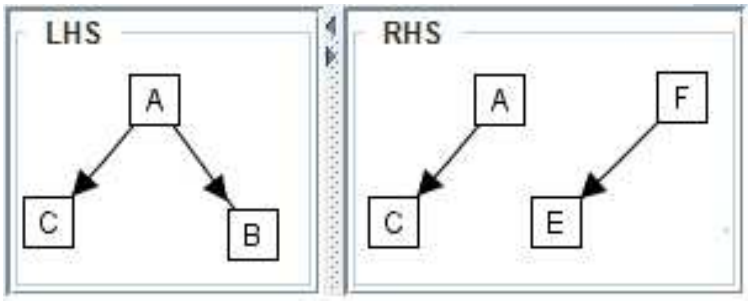


Figure 6.4. Exemple d'une règle de réécriture.

Par conséquent, nous ne devons pas, analyser le dépôt basé-règle pour générer les changements atomiques (opérations) comme le cas dans les techniques traditionnelles. Nous pouvons définir le changement atomique comme la création/suppression de tout élément de notre graphe (code source). Dans notre dépôt, chaque règle de réécriture est enregistrée directement quand elle est appliquée. Donc, nous ne devons pas utiliser un outil externe (ex. diff) pour comparer les différentes versions d'un programme afin de détecter de telles règles (changements). Les différentes règles de réécriture atomiques dans notre proposition sont montrées dans le Tableau 6.2.

Tableau 6.2. Les règles de réécriture atomiques.

	Les règles de création		**Les règles de suppression**	
	Abréviation	**Règle de réécriture atomique**	**Abréviation**	**Règle de réécriture atomique**
Nœud	CC	Create a Class	DC	Delete a Class
	CA	Create an Attribute	DA	Delete an Attribute
	CM	Create a Method	DM	Delete a Method
	CP	Create a Parameter	DP	Delete a Parameter
	CR	Create a Return value	DR	Delete a Return value
	CAS	Create an ASpect	DAS	Delete an ASpect
	CPO	Create a POintcut	DPO	Delete a POintcut
	CAD	Create an ADvice	DAD	Delete an ADvice
	CI	Create an Introduction	DI	Delete an Introduction
Arc	CECA	Create Edge CAlls	DECA	Delete Edge CAlls
	CEIA	Create Edge Introduces Attribute	DEIA	Delete Edge Introduces Attribute
	CEIM	Create Edge Introduces Method	DEIM	Delete Edge Introduces Method
	CECR	Create Edge CRosscuts	DECR	Delete Edge CRosscuts

6.2.3 Les Transactions de Changement Atomique

Une transaction de changement atomique inclut tous ce qui est nécessaire pour un changement spécifique i.e. c'est un ensemble des changements atomiques pour une requête du changement spécifique. Dans notre dépôt basé-règle, chaque version du programme est entreposée dans le dépôt comme un ensemble *de séquences de règle de réécriture* (des règles de réécriture enregistrées dans un certain ordre). Chaque séquence de règle de réécriture formule une requête de changement spécifique i.e. dans notre contexte, les séquences de règle présentent les transactions du changement. Donc, nous ne devons pas prétraiter le dépôt pour générer de telles transactions.

D'un autre côté, chaque séquence de règle est une liste ordonnée de règles. Donc, nous avons déjà les dépendances sémantiques entre les règles (changements). Une séquence de règle contient l'ensemble de règles pour un changement spécifique. Ainsi, les règles dans une séquence de règle sont toujours appliquées ensemble. Par conséquent, pour détecter les patrons de changement dans notre approche, nous devons ***juste*** analyser les différentes séquences de règle dans notre dépôt proposé.

6.2.4 Identification des Patrons de Changement

Dans cette étape, nous analysons le dépôt basé-règle de réécriture pour identifier les patrons de changement. Nous définissons les patrons de changement comme des modifications communes et récurrentes d'un système logiciel dans le temps, au cours de son évolution. Donc, nous extrayons des ensembles du changement atomique (règles de réécriture atomiques) qui s'appliquent fréquemment ensemble parmi les séquences de règle. Dans notre contexte, de tels ensembles, appelés patrons de règle (rule-patterns) ou patrons de changement, font référence aux changements atomiques (création/suppression) qui se produisent (toujours) ensemble.

Nous utilisons l'algorithme Apriori [AGR94] pour détecter les patrons de changement dans notre dépôt basé-règle. Soit $R = \{r_1,\ r_2, \dots, r_m\}$ un ensemble de règle atomique i.e. création ou suppression d'éléments ou de dépendances de graphe (Tableau 6.2), et $X \subseteq R$ un ensemble de règles. Nous définissons la base de données (dépôt) D comme

un ensemble de séquences de règle: $D = \{s_1,\ s_2, \dots, s_n\}$, où $s_i = \{s_{i1},\ s_{i2}, \dots, s_{ik}\}$ et $s_{ij} \in R$. Aussi, soit $s(X)$ un ensemble de séquences de règles qui contiennent l'ensemble de règle X, formellement $s(X) = \{Y \in D | Y \supseteq X\}$. Finalement, le support d'un ensemble de règle X est la fraction des séquences de règles dans la base de données qui contient X : $support(X) = |s(X)|/|D|$. Alors X est appelé un ensemble de règles fréquents quand son support est plus grand qu'un support minimum donné: $support(X) \geq minsupport$.

En autre terme, la force du patron $\{r_1, \dots, r_n\}$, où chaque r_i est une règle de réécriture atomique (changement), est mesurée par le support qui est le nombre (ou pourcentage) de séquences de règle contenants $r_1, \dots, r_n$. Un patron fréquent décrit un ensemble de règles atomiques qui ont le support plus grand qu'un seuil prédéterminé appelé min_support.

6.3 Deuxième Approche de l'Analyse d'Evoluion des Programmes OA : La Détection des Défauts de Modularité dans le Logiciel OA

Comme tout système logiciel, les systèmes OA sont modifiés de façon continue. Par conséquent, ils augmentent en taille et en complexité. Après beaucoup d'améliorations et d'autres activités de l'évolution, la modularité du logiciel OA peut être cassée, et les changements deviennent durs à faire. La modularité insuffisante des préoccupations transversales complique l'évolution du logiciel OA et réduit leur réutilisabilité. Par conséquent, des méthodes et des techniques sont exigées pour détecter les défauts de modularité dans le logiciel OA, pour améliorer sa décomposition et augmenter sa modularité. Cette section explique comment nous pouvons bénéficier des métadonnées extraites à partir d'un dépôt enregistrant l'historique d'évolution, pour détecter les défauts de modularité dans un logiciel OA.

6.3.1 Les Dépendances Logiques et les Défauts de Modularité

Sémantiquement, les entités du logiciel reliées logiquement ne peuvent pas être dépendantes structuralement les unes des autres i.e. les différentes entités d'un

système logiciel peuvent être mises en rapport les unes aux autres, bien que cette relation ne soit pas facilement détectable dans le code source du logiciel. Récemment, les chercheurs ont utilisé les historiques de version pour identifier plus efficacement les composants reliés sémantiquement (appelé le couplage logique ou évolutionnaire) en vérifiant comment les composants changent historiquement ensemble [HAS08, KAG07]. La détection des dépendances logiques extrait des dépendances intéressantes entre les entités du logiciel, ce qui n'est pas possible avec l'analyse d'une seule version.

Donc, en basant sur les données historiques nous pouvons détecter des dépendances logiques entre les entités d'un système logiciel. Dans ce dernier, deux entités sont dépendantes si un changement dans une entité A implique un changement dans une autre entité B, on dit que B dépend de A. L'analyse de ces dépendances peut aider dans la détection des défauts de modularité dans un système logiciel. L'idée de base est que nous pouvons distinguer deux types de dépendances logiques, comme montré dans la Figure 6.5: les dépendances logiques internes et externes. Une dépendance logique est une dépendance logique interne, si elle relie deux entités qui appartiennent au même module dans la décomposition du logiciel. D'un autre coté, une dépendance logique externe relie deux entités qui appartiennent à deux différents modules logiciels.

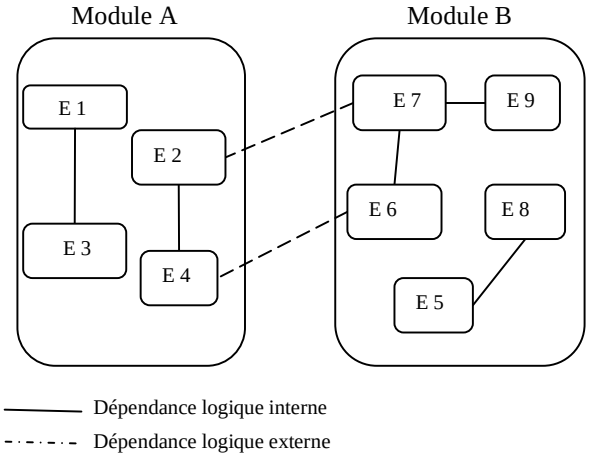


Figure 6.5. Les types de dépendances logiques.

Le dernier type est le plus important dans notre contexte; parce que l'existence de dépendances logiques externes dans un système logiciel présente des défauts possibles de modularité dans ce système. Deux modules qui sont supposés être changé indépendamment sont changés ensemble i.e. un changement dans une entité qui appartient à un module spécifique nécessitera des changements dans d'autre(s) entité(s) qui appartient à d'autres modules. Donc, nous appelons les telles dépendances logiques "dépendances logiques négatives" ou des "défauts de modularité".

Le paradigme OA vise à améliorer la modularité d'un système logiciel pour encapsuler chaque préoccupation transversale dans une seule unité, appelé Aspect. Notre question de recherche est: comment nous pouvons détecter efficacement des défauts de modularité dans le logiciel OA? Nous utilisons l'idée décrite ci-dessus pour accomplir notre but. Dans ce contexte, les modules logiciels (Figure 6.5) sont les préoccupations transversales (Aspects) du système, et les défauts de modularité sont considérés comme des dépendances logiques externes parmi ces aspects.

Pour résumer, l'existence des défauts de modularité (dépendances logiques externes) dans un système OA montre que la séparation des préoccupations transversales dans ce système n'est pas encore parfaite. Les préoccupations transversales dépendantes logiquement sont des candidates pour la restructuration ou le refactoring. Ici, les défauts de modularité détectés sont utilisés pour guider les efforts d'amélioration; pour obtenir une décomposition plus stable avec peu de dépendances i.e. une situation idéale autoriserait de changer chaque préoccupation transversale indépendamment des autres. C'est très utile pour construire une meilleure modularité pour le système logiciel OA, et une bonne réutilisabilité de leurs préoccupations transversales.

6.3.2 L'Approche de Détection des Défauts de Modularité

Le but de cette approche est de découvrir des défauts de modularité dans un logiciel OA en analysant son historique d'évolution. Comme montré dans la Figure 6.6, notre approche consiste en deux étapes complémentaires, qui forment une approche

intégrée pour la détection des défauts de modularité: 1) le dépôt du logiciel OA est analysé pour détecter les dépendances logiques entre les entités du logiciel qui appartiennent aux différents aspects du système; 2) les dépendances logiques extraites sont analysées ensuite par rapport à la décomposition du logiciel, pour détecter et localiser les défauts de modularité. En résumé, si deux entités x et y sont fréquemment changées ensemble, et elles appartiennent à deux aspects indépendants A et B respectivement, donc la dépendance logique (x, y) représente un défaut de modularité.

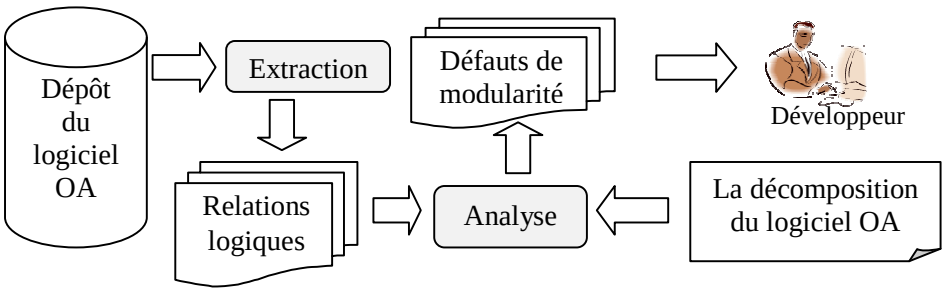


Figure 6.6. La détection des défauts de modularité.

Le but principal de tels défauts de modularité est d'évaluer la modularité d'un logiciel OA, et de guider les efforts de son amélioration i.e. ces dépendances logiques peuvent être utilisés pour guider le développeur du logiciel pendant les tâches de restructuration et derefactoring. Donc, c'est au développeur d'examiner le code correspondant (les entités qui sont mises en rapport avec un défaut de modularité) pour améliorer et perfectionner la modularité du système OA.

1) L'extraction des dépendances logiques

Dans cette étape, nous adressons les questions suivantes: Quelles sont les entités dépendantes logiquement dans le système OA? Et quelle sont les puissances de telles dépendances? Donc, les données sources de l'analyse constituent des différents modules des aspects du logiciel: champs, méthodes, points de coupure, consignes et introductions...etc. Nous utilisons la même approche de la détection des patrons de

changement présentée dans la section précédente, mais nous visons ici à analyser le dépôt basé-règle pour extraire les dépendances logiques dans un code source AspectJ.

L'algorithme Apriori [AGR94] est utilisé pour extraire les dépendances logiques à travers le principe suivant : soit $E = \{e_1,\ e_2, \dots, e_n\}$ un ensemble des entités d'aspects i.e. les entités qui appartiennent aux différents aspects du logiciel OA, et $X \subseteq E$ un sous ensemble d'entité. Nous définissons le dépôt R comme un ensemble de séquence de règle de réécriture $R = \{r_1,\ r_2, \dots, r_m\}$, où $r_i = \{e_{i1},\ e_{i2}, \dots, e_{ik}\}$ et $e_{ij} \in E$. Donc, soit $s(X)$ l'ensemble de séquence de règle de réécriture qui contient l'ensemble d'entité X, formellement $s(X) = \{Y \in R | Y \supseteq X\}$. Finalement, le support de l'ensemble d'entité X est la fraction de séquence de règle de réécriture dans le dépôt qui contient X : $support(X) = \frac{|s(X)|}{|R|}$. Donc, X est appelé une dépendance logique quand son support est plus grand qu'un support minimum donné: $support(X) \geq minsupport$. Dans d'autre terme, la force d'une dépendance logique $\{e_1, \dots, e_n\}$ où chaque e_i est une entité d'aspects, est mesurée par le support qui est le nombre (ou pourcentage) de séquence de règle de réécriture qui contiennent les entités $e_1, \dots, e_n$.

2) la détection des défauts de modularité

Les dépendances logiques extraites dans l'étape précédente, sont analysées alors pour détecter les défauts de modularité dans le système logiciel OA. Cette analyse est basée sur la décomposition du logiciel OA comme des préoccupations transversales (Aspects). Dans cette étape, les dépendances logiques détectées sont classés dans deux catégories: dépendances logiques internes et externes. Comme présenté dans la Figure 6.7, nous définissons la dépendance logique interne comme une relation entre deux entités qui appartiennent au même Aspect. Les dépendances entre les entités d'un Aspect et toutes les autres entités qui appartiennent à d'autres aspects sont considérées comme des dépendances logiques externes.

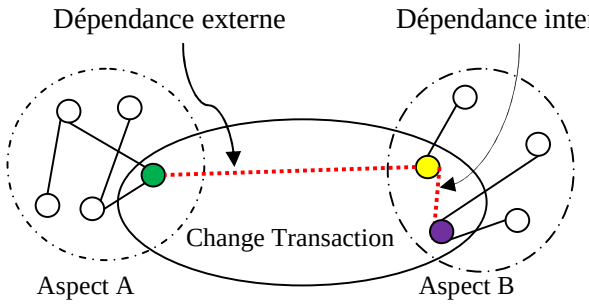


Figure 6.7. L'analyse des dépendances logiques.

Ces dépendances logiques externes sont considérées comme des défauts possibles dans la modularité du logiciel OA. Formellement, un ensemble de dépendances logiques externes DLE est défini comme DLE = $\{(e_i, e_j) | e_i \in A, e_j \in B\}$ où A et B sont deux aspects indépendants. Donc, l'ensemble de défauts de modularité DM dans un programme orienté aspect P est défini comme:

$$DM(P) = \sum_{i=1}^{|ELC|} DLE_i$$

Puisque les défauts de modularité sont des dépendances logiques, chaque défaut de modularité a une valeur du support: le nombre de transactions (séquence de règle de réécriture) qui contiennent la dépendance logique externe. Donc, nous pouvons dire que (x, y) est un défaut de modularité qui s'est produit une fois, (y, z) est un défaut de modularité qui s'est produit deux fois, et ainsi de suite.

6.3.4 Discussion

Les défauts de modularité détectés peuvent être utilisés pour aider les tâches de restructuration et du refactoring. Si certains aspects changent en même temps très souvent sur plusieurs versions, ils peuvent être utilisés comme des candidats pour le refactoring. Dans un autre coté, nos résultats peuvent être utilisés pour évaluer la modularité du logiciel OA. Nous pouvons définir par exemple une mesure de modularité en utilisant les dépendances logiques détectés (par exemple, elle peut être égal au nombre d'aspectsnon reliés, divisé par le nombre total d'aspects). Basé sur cette mesure nous pouvons évaluer la modularité du système OA.

Cette mesure peut être utilisée plus tard pour comparer différentes implémentations des systèmes logiciels OA. Donc, nous pouvons répondre aux questions intéressantes comme: Est-ce que le programme OA *P* est plus modulaire que le programme OA *P'*? Est-ce que l'implémentation de la préoccupation transversale C dans le programme P est beaucoup plus encapsulée que dans le programme P'? Si nous détectons des préoccupations transversales (aspects) qui n'ont aucune dépendance avec les autres préoccupations transversales, elles peuvent être des modules parfaits pour la réutilisation.

6.4 Mise en œuvre

Dans cette section, nous présentons la validation de nos propositions, le dépôt basé-règle de réécriture ainsi que les approches de détection des patrons de changement, et la détection des défauts de modularité.

6.4.1 Le Dépôt basé-règle de Réécriture

La présentation de notre validation est donnée dans la Figure 6.8 qui peut être résumée dans les parties suivantes:

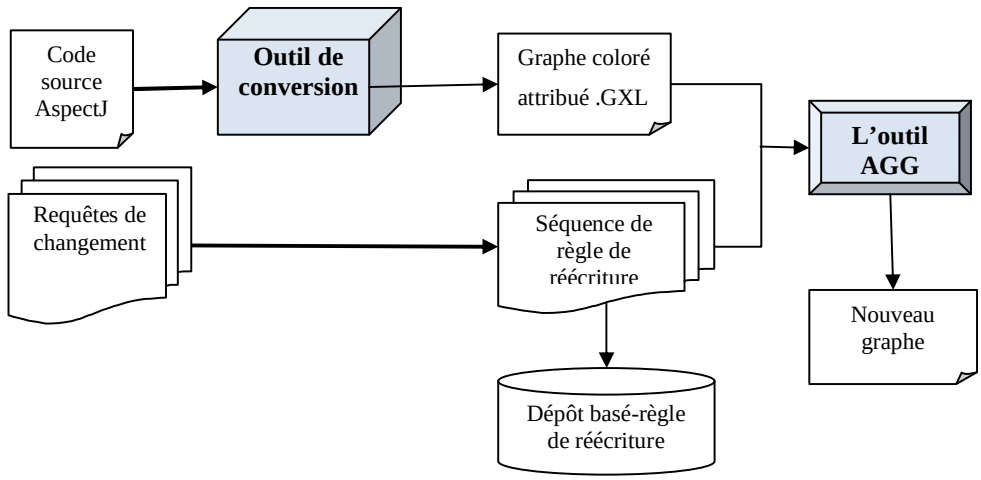


Figure 6.8. Dépôt-basé règle de réécriture.

(1) l'outil de conversion: pour représenter le code source AspectJ comme un graphe coloré attribué, nous avons implémenté un outil de conversion. Ce dernier a été présenté dans le chapitre précédent. Cet outil converti le code source AspectJ à un graphe coloré attribué dans le format GXL (Graph Exchange Language) [WIN01]. Ce graphe est importé dans l'outil AGG (Attributed Graph Grammar) [LOW93] pour appliquer les transformations nécessaires.

(2) les requêtes du changement: chaque requête de changement est formulée comme une séquence de règle de réécriture i.e. l'ensemble de règles de réécriture appliquées dans un certain ordre. Donc, nous utilisons AGG pour appliquer ces règles sur le graphe coloré attribué. Nous pouvons aussi formuler des propriétés, des contraintes; analyser le graphe…etc.

(3) Le dépôt basé-règle de réécriture: nous enregistrons chaque séquence de règle de réécriture dans le dépôt quand elle est appliquée. Notre dépôt peut être défini pratiquement, comme un ensemble de documents GXL. Chaque document GXL présente une version du programme (graphe). C'est l'ensemble des séquences de règle de réécriture appliquées dans une session d'évolution. La Figure 6.9 montre la structure d'une version. La structure d'une séquence de règle est représentée dans le côté droit de la Figure 6.9. Elle est constituée de règles de réécriture. Chaque règle est la combinaison de preserved, deleted, created et condition. Ils décrivent les éléments qui doivent être préservés, supprimés, et les conditions de la règle respectivement.

Note: les éléments (nœuds/arcs) du graphe, les règles de réécriture et les séquences de règle de réécriture contiennent un identificateur unique pour garder leurs identités dans le dépôt. Ceci aide à suivre le changement et à analyser le dépôt.

6.4.2 La Détection des Patrons de Changement

Puisque, notre dépôt est un ensemble de documents GXL, le problème de la détection des patrons de règle de réécriture à partir du dépôt basé-règle est converti en l'extraction des patrons à partir des documents GXL.

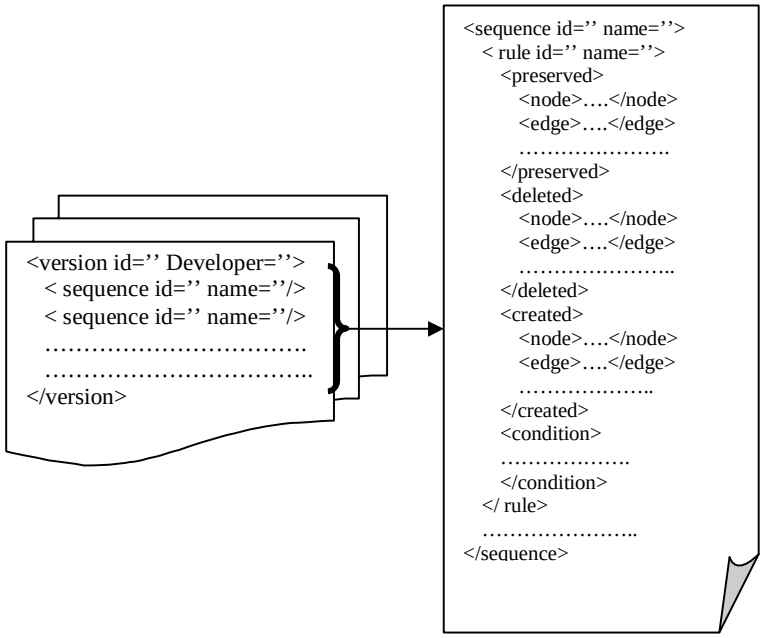


Figure 6.9. Structure d'une version.

Chaque document GXL est un document XML (*eXtended Markup Language*) [SUZ98]. Donc, nous utilisons l'implémentation XQuery de l'algorithme Apriori proposé par Wan et al. [WAN03] pour extraire de tels patrons. Ils proposent un ensemble de fonctions écrites seulement en XQuery qui implémente ensemble l'algorithme Apriori. Pour créer un document XML approprié pour être l'entrée du l'algorithme XQuery Apriori; nous suivons le pipeline dans la Figure 6.10. Ce dernier inclut 3 étapes:

Étape 1: notre dépôt basé-règle de réécriture est un ensemble de documents GXL (versions). Chaque document GXL est un ensemble de séquences de règle. Nous regroupons toutes les séquences de règle dans chaque version en seul un document GXL. Ce document représente les transactions qui doivent être analysées par l'algorithme de détection des patrons de changement.

Étape 2: par la puissance de XSLT (*XML Stylesheet Language Transformation*) [CLA99], nous transformons le document des séquences de règle produit dans la première étape à un document XML simple. Ce dernier doit être aussi simple que possible pour détecter efficacement les patrons de règle.

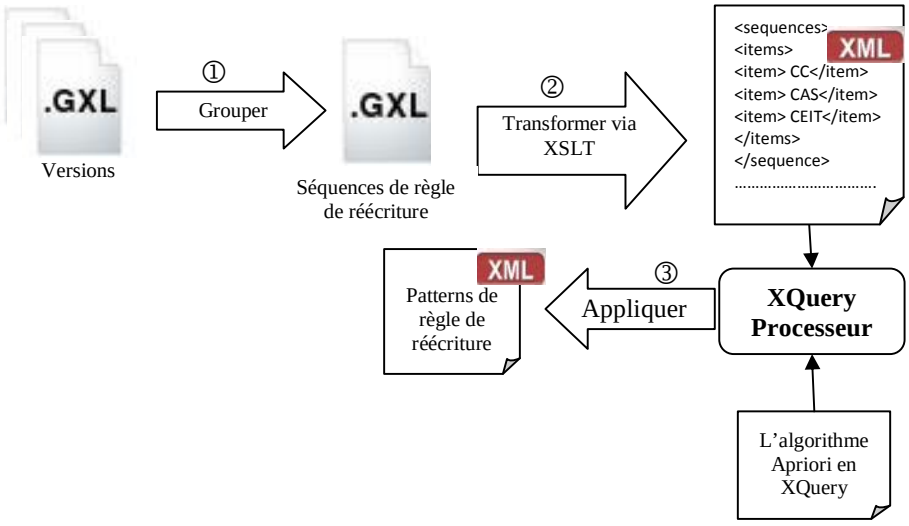


Figure 6.10. La détection des patrons de règle de réécriture.

Nous convertissons chaque règle de réécriture atomique en un tag simple avec l'abréviation de la règle de réécriture atomique comme valeur (Tableau 6.2). Par exemple, la règle de réécriture qui formule la création d'un nœud "Aspect" est représentée dans le format GXL comme suit:

```
<created>
 <node id="I173">
   <type name="Aspect"/>
   <attr name="Name">
     <string>A</string>
   </attr>
 </node>
</created>
```

Ce fragment est converti au tag simple suivant :

<item>CAS</item>

Étape 3: le document XML produit dans l'étape précitée est analysé avec un processeur XQuery; selon l'algorithme XQuery Apriori (nous devons fixer un support minimal spécifique). La sortie de cette étape est un document XML qui représente les patrons de règle de réécriture dans notre dépôt basé-règle. Ces règles représentent des changements au code source AspectJ. Par conséquent, notre approche permet de détecter les patrons de changement dans le code source AspectJ.

6.4.3 La Détection des défauts de modularité

Nous utilisons le même principe présenté dans la sous-section précédente pour extraire les dépendances logiques. Un outil de la détection des défauts de modularité est implémenté pour filtrer les dépendances logiques extraites. Ici, l'outil extrait les défauts de modularité en éliminant les dépendances logiques internes.Il utilise la représentation XML (obtenue à partir de l'outil AspectJML) du code source pour tester si les entités qui appartiennent à une dépendance logique spécifique existe dans le même aspect ou dans des aspects différents en utilisant leurs identificateurs (identifiers). Les résultats présentent des défauts de modularité possibles.

6.5 Expérimentations

Dans cette section nous visons à montrer la faisabilité de nos propositions. Donc, nous présentons un ensemble d'expérimentations où nous traitons différents cas d'études.

6.5.1 La Détection des Patrons de Changement

Pour valider notre approche, premièrement nous devons rassembler quelques données qui supportent notre proposition. Puisque aucun système de version basé-changement consacré à la POA n'existe déjà. En plus, aucun système n'a enregistré les changements dans le format de règle de réécriture; nous ne pouvons pas compter sur les dépôts du logiciel préexistant comme sources de données. Donc, nous avons choisi deux programmes AspectJ pour l'expérimentation: Figure Editor et Tracing. Alors, nous avons appliqué différents scénarios d'évolution sur ces programmes pour générer des différentes versions. Par conséquent, nous avons construit 5 versions du programme Figure Editor et 3 versions du programme Tracing. Des informations au sujet du nombre de Ligne de Code (LOC), nombre de versions et des séquences de règles pour ces programmes sont montrées dans le Tableau 6.3.

Tableau 6.3. Les programmes de l'expérimentation.

Programme	LOC	#version	# séquence de règle
Figure Editor	393	5	25
Tracing	1059	3	5

Le nombre des différentes règles de réécriture atomiques dans chaque version des programmes Figure-Editor et Tracing est montré dans la Figure 6.11 et 6.12 respectivement. Nous prédéfinissons le seuil du min_support à 20% pour les deux programmes. Après l'application de notre approche de détection des patrons de règle, le Tableau 6.4 résume les patrons de règle de réécriture générés.

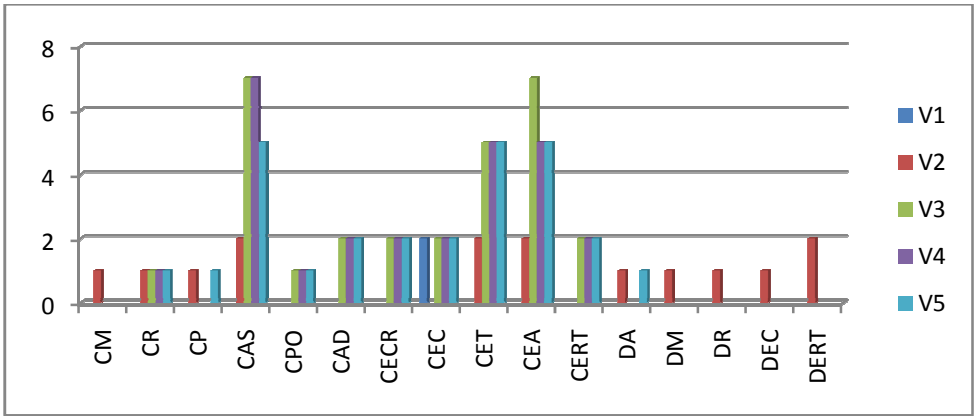


Figure 6.11. Les règles atomiques dans le programme Figure Editor.

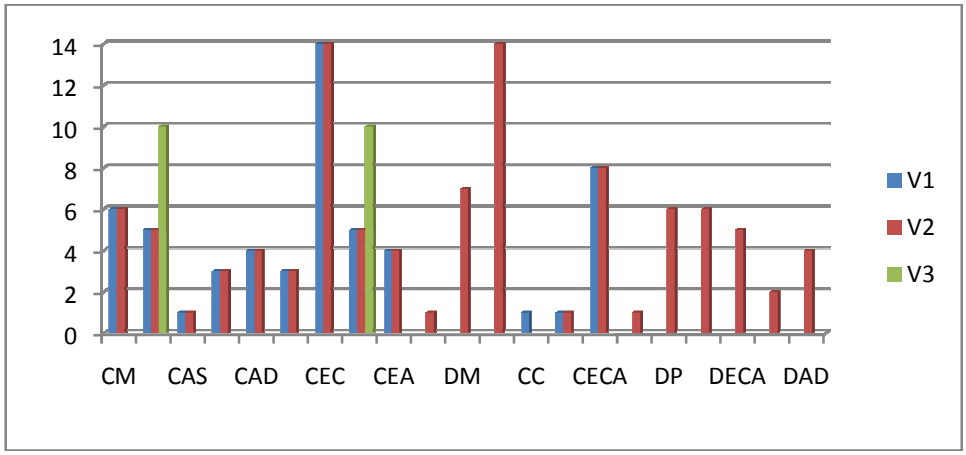


Figure 6.12. Les règles atomiques dans le programme Tracing.

Tableau6.4. Les patrons de règle de réécriture.

Pattern	Support	Programme
CM, CP, CECA	0.33	Tracing
CAS, CPO, CAD, CECR, CECA	0.33	Tracing
CAD, CP	0.24	Figure Editor
CPO, CP, CECR	0.20	Figure Editor

L'analyse du résultat. Nous avons détecté deux patrons de règle pour le programme Tracing. Le premier montre que fréquemment, la création des méthodes mène à la création des paramètres et des arcs du type *Calls* (appels à d'autres méthodes). Le deuxième montre que la création d'un Aspect mène à la création de points de coupure et de consignes. Et la création des arcs du type Crosscuts, pour spécifier les points de jointure. Cela mène aussi à la création des arcs de type *Calls* entre les consignes et les méthodes.

Pour le programme Figure-Editor, nous avons détecté aussi deux patrons. Les deux règles atomiques "creation of advice" et "creation of parameter" sont fréquemment appliquée ensemble. Et les règles atomiques: "creation of pointcut", "creation of parameter" et " creation of edge crosscuts" sont toujours appliquées ensemble. De tels patrons de règle peuvent aider le développeur à appliquer un changement complet i.e. il ne doit pas appliquer un changement spécifique dans un patron sans appliquer les autres. L'application de notre approche à d'autres cas d'étude avec un grand dépôt basé-règle peut détecter des patrons de changement plus intéressants.

6.5.2 La Détection des défauts de modularité

Pour prouver la faisabilité de notre approche, nous étudions 22 versions de 3 applications OA de taille moyenne. Ces dernières sont présentées dans le Tableau 6.5. Le premier, appelé Contract4J, nous avons considéré 5 versions de cette application dans notre étude. Le second est un logiciel pour dériver des applications qui manipulent des photos, vidéos et musique sur les appareils mobiles appelé Mobile-Media. Nous avons sélectionné 7 versions dans cette expérimentation. Le dernier système appelé Health-Watcher [SOA02], est un système d'information réel basé sur le Web qui permet aux citoyens d'enregistrer des réclamations au sujet de la santé dans les institutions publiques. Nous avons sélectionné 10 versions de Health-Watcher dans notre étude.

Tableau 6.5. Les programmes expérimentés.

Software	#version	#Aspect
Contract4J	5	8—14
Mobile Media	7	4—42
Health Watcher	10	11—23

Après l'application de notre approche sur ces systèmes, nous avons trouvé beaucoup de dépendances logiques internes comme: "fréquemment le changementd'un point de coupure implique le changementde ses consignes en rapport", "le changementd'un champ, implique le changementdes méthodes qui l'utilisent"…etc. Un ensemble de défauts de modularité (dépendances logiques externes) sont aussi détectés. Par exemple, dans le programme Contract4J, nous avons détecté que la méthode *doTest* et la méthode *doBeforeTest* sont fréquemment changés ensemble, malgré qu'ils appartiennent aux aspects indépendants *ConstructorBoundaryConditions* et *MethodBoundaryConditions* respectivement. Les consignes appliquées après le point de coupure *postCtor* dans l'aspect *ConstructorBoundaryConditions* est fréquemment changé avec les points de coupure *preNotInContract*, *postNotInContract* et *invarNotInContract* qui appartient à l'aspect *UsageEnforcement*.

Dans le programme Mobile-Media, nous avons détecté beaucoup plus de défauts de modularité par rapport à ceux détectés dans le programme Contract4J. Par exemple, les points de coupure *loadMediaDataFromRMS* et *readMediaAsByteArray* sont fréquemment changés ensemble, malgré qu'ils appartiennent aux aspects indépendants *DataModelAspectEH* et *UtilAspectEH* respectivement. La même chose pour les points de coupure *getMedias* et *getBytesFromMediaInfo*. Nous avons aussi détecté plusieurs duplications de point de coupure: les points de coupure *handleCommandAction* et *appendMedias* sont dupliqués dans les aspects *FavouritesAspect* et *SortingAspect*. En plus, le point de coupure *showImage* est défini dans les deux aspects *ControllerAspectEH* et *SortingAspect*. Donc tout changement dans les tels points de coupure implique des changements dans plusieurs autres aspects. En plus, les points de coupure *getMediaController* et *showImage*

(appartiennent aux aspects *CopyPhotoAspect* et *SortingAspect* respectivement) sont fréquemment changés ensemble.

Finalement, nous avons détecté beaucoup de dépendances logiques internes dans l'application Health-Watcher, mais nous n'avons pas détecté des défauts sérieux de modularité dans ce système (sauf quelques dépendances logiques externes détectés une seule fois). Donc, nous pouvons dire que: comparée avec les applications précitées (Contract4J et Mobile-Media), l'application Health-Watcher a une bonne modularité, et ses préoccupations transversales (Aspects) sont de bons modules réutilisables.

6.6 Bilan

Dans ce chapitre nous avons présenté les principes derrière un dépôt basé-changement pour le logiciel OA, et comment ils peuvent traiter quelques problèmes d'évolution de logiciel OA. Nous avons proposé un dépôt basé-règle de réécriture pour les programmes AspectJ. Ce dernier est représenté comme un graphe coloré attribué, et les changements sont formulés comme des règles de réécriture. Chaque règle de réécriture est entreposée directement dans le dépôt quand elle est appliquée. Notre approche est consacrée pour manier la caractéristique d'inconscience dans la POA. Elle aide pour améliorer la compréhension du programme en rendant l'interaction entre les aspects et le code de base plus explicite. Cela ne réduit pas l'inconscience parmi les modules de système, parce que chaque module (aspect, classe) peut être observé facilement comme un module indépendant avec notre représentation.

En plus, la représentation et l'entreposage des changements comme des règles de réécriture conservent l'information complète au sujet du changement (les entités changées, leurs dépendances, les contraintes…etc.). Ce format aide pour rendre le dépôt de l'évolution plus adéquat à la nature transversale du logiciel OA, en évitant les limites des systèmes de version actuels basés-fichier i.e. comparée avec le principe basé-fichier des outils de versioning classiques, notre proposition entrepose les changements dans les entités du logiciel et leurs dépendances indépendamment des

fichiers qu'ils contiennent. Donc, les changements dans les dépendances transversales sont bien entreposés dans notre dépôt.

En plus, nous avons proposé une approche de détection des patrons de changement pour le code source AspectJ. Cette approche est basée sur notre dépôt basé-règle pour extraire les patrons de règle de réécriture atomiques. Ces patrons sont considérés comme des patrons de changement dans le code source AspectJ. Ils peuvent être utilisés comme des mesures d'aide et des prédicateurs de faute pour l'évolution du logiciel AspectJ. C'est très important pour prédire l'évolution future du logiciel AspectJ, d'améliorer la compréhensibilité du système logiciel et par conséquent de diminuer le coût de l'évolution.

En utilisant notre approche de détection des patrons de changement proposée, nous avons prouvé que c'est plus facile d'extraire les changements à partir de notre dépôt, parce qu'ils sont entreposés avec une manière explicite. Cela améliore la qualité des résultats des efforts de Mining. Donc, nous pensons que notre dépôt peut être une source intéressante pour une analyse d'évolution de haute qualité.

Nous nous sommes basé sur le même principe pour extraire les dépendances logiques dans un code source OA. Ces dépendances sont ensuite utilisées pour détecter les défauts de modularité dans le tel code source. Plusieurs cas d'étude sont expérimentés pour démontrer la faisabilité de notre approche. Les résultats montrent que l'approche est capable de détecter les dépendances logiques entre les entités des aspects, aussi bien que les défauts de modularité. L'approche mène naturellement à une évaluation de la modularité d'un système OA. Nos résultats peuvent être utilisés par la suite pour la restructuration et le refactoring du système logiciel OA.

Finalement, nous croyons que l'approche fondamentale présentée dans ce chapitre est assez générique pour être adaptée à d'autres langages de programmation orientés objet ou OA.

CHAPITRE 7

Conclusion & Perspectives

"Research is to see what everybody else has seen; and to think what nobody else has thought" Albert Szent-Gyorgyi (1893 – 1986)

Malgré les différents bénéfices de la programmation OA, les programmes OA font face à beaucoup de problèmes lors de leur évolution. Pour bien traiter l'évolution de ces logiciels, nous avons proposé un Framework d'évolution. Ce dernier permet de modéliser et de valider l'évolution du logiciel OA, et en parallèle il permet de garder l'historique de cette évolution dans un dépôt spécifique à ce paradigme. Nous avons sélectionné la transformation de graphes comme formalisme sous-jacent pour notre Framework proposé. Ce formalisme constitue le meilleur choix pour notre proposition, parce qu'il est concentré naturellement sur la description et la manipulation d'information structurelle, et il autorise la modélisation des conditions de changement aussi. Ce chapitre conclut notre dissertation en résumant les contributions de la thèse, et en indiquant les zones potentielles pour une future recherche.

7.1 Contributions

Dans cette thèse, nous avons analysé les besoins des systèmes OA afin de proposer un Framework pour assurer leur évolution. Ce Framework vérifie deux objectif principaux: 1) il permet de modéliser l'évolution des programmes OA; 2) et en même temps, il permet de garder cette évolution dans un dépôt basé-changement. Afin d'atteindre ces objectifs, nous nous sommes basé sur le paradigme de la transformation algébrique de graphes. Ce dernier est fondé sur une bonne base théorique et mathématique. De plus, pratiquement il est supporté par une large gamme d'outils et d'environnements.

A travers la transformation de graphes, le code source OA est modélisé comme un graphe coloré attribué. Avec une telle représentation, l'évolution est spécifiée comme des modifications sur le graphe du code source. Donc, les requêtes de l'évolution peuvent être capturées dans des règles de réécriture de graphes. Une collection de règles qui regroupent toutes les possibilités évolutionnaires. Elle est appelée un système de production de graphes. Le graphe qui représente l'état initial du système, constitue un modèle formel de l'évolution du système OA.

Nous nous sommes basé sur ce modèle formel d'évolution pour proposer un dépôt de changement pour les systèmes OA i.e. le changement est traité comme une entité de première classe. Nous adoptons une approche pour entreposer les changements sur le code source OA (quand ils se produisent) dans un dépôt basé-règle de réécriture. Dans ce dernier, les changements sont entreposés comme des règles de réécriture. Le dépôt proposé prend en charge les propres caractéristiques des systèmes OA. Nous pensons que ce dépôt peut être une source fondamentale d'information pour l'analyse de l'évolution du logiciel OA. Et il peut ouvrir de nouvelles directions pour les développeurs et les chercheurs pour mieux comprendre et explorer l'évolution du logiciel OA.

Pour analyser notre dépôt, nous avons proposé une approche de détection des patrons de changement des programmes AspectJ. Dans cette approche, l'algorithme Apriori est utilisé pour extraire les ensembles des règles de réécriture atomiques qui s'appliquent fréquemment ensemble parmi les séquences de règle de réécriture. Dans ce contexte, de tels ensembles, sont appelés patrons de changement. Ils font référence

aux changements qui se produisent fréquemment ensemble. Ces patrons peuvent être utilisés pour comprendre l'évolution des programmes OA, prévoir des changements futurs, détecter de nouvelles préoccupations, …etc.

En plus, —et pour mieux prouver l'utilité de notre dépôt basé-règle de réécriture— nous avons proposé de détecter les défauts de modularité dans un logiciel OA en analysant ses dépendances logiques. Nous avons suivi le même principe de la détection des patrons de changement pour extraire les dépendances logiques dans le logiciel OA. Ensuite, ces dépendances sont analysées pour détecter les défauts de modularité (les dépendances logiques externes). Les résultats de cette analyse peuvent être ensuite utilisés pour améliorer la modularité du système OA.

7.2 Les Applications de Notre Framework d'Évolution du Logiciel OA

D'une perspective théorique, la force de cette proposition est la présentation complète de l'évolution du logiciel OA avec le formalisme de la transformation de graphes. Le Framework représente complètement le code source OA comme un graphe coloré attribué. Donc, il ne compte pas sur des descriptions textuelles ou des annotations qui doivent être analysés pour l'application ou la vérification du modèle d'évolution. Cette représentation basé-graphe peut être manipulée en utilisant des grammaires de graphes qui sont des représentations naturelles, formelles, visuelles, déclaratives et de haut niveau pour les graphes.

D'une perspective d'un praticien, l'utilisation des mécanismes de la réécriture de graphes rend le Framework d'évolution du logiciel OA théoriquement riche, et pratiquement utile comme un outil assistant l'évolution. Les fondations théoriques des systèmes de réécriture de graphes peuvent aider dans la vérification et la validation des propriétés d'exactitude et de convergence de la transformation. Notre Framework d'évolution peut être utilisé comme un support pour:

L'analyse et la compréhension: la compréhension du programme est une partie essentielle dans l'évolution du logiciel i.e. les logiciels qui ne sont pas bien compris ne peuvent pas être changés. Une fois qu'un modèle du logiciel OA a été développé, il peut être considéré comme un support de documentation. Un tel modèle peut être passé aux utilisateurs consécutifs dans une étape de développement donnée, tel

qu'entre des générations consécutives de développeurs. Par exemple, ils peuvent visualiser le modèle du logiciel en utilisant l'outil de visualisation.

Un historique complet de l'évolution: notre approche crée un dépôt fiable de l'évolution pour le futur développement. Notre Framework d'évolution évite les limites des systèmes de contrôle de version actuels. Ces derniers n'entreposent pas toute l'information générée par les développeurs. Ils n'enregistrent pas chaque version intermédiaire du système logiciel, mais seulement des versions instantanées prises quand un développeur entrepose le code source dans le dépôt.

Une modélisation formelle: la majorité des techniques utilisées pour analyser les dépôts d'évolution sont des approches formelles. Ils modélisent les éléments du dépôt dans une représentation formelle. Donc, le dépôt formel est le meilleur choix pour faciliter les tâches d'analyse de l'historique (la détection du couplage logique, le calcul des métriques…etc.). Nous concluons que ce formalisme est en effet convenable pour la spécification de l'évolution du logiciel OA, parce que (i) les graphes peuvent être utilisés comme une représentation transparente du code source; (ii) les règles de réécriture de graphes sont une façon concise et élégante pour spécifier les transformations du code source impliquée par une requête d'évolution.

Une évolution Visuelle: notre Framework présente une traduction du processus de l'évolution d'un langage OA aux systèmes de transformation de graphes. L'idée était de coder la structure du logiciel OA graphiquement, et d'utiliser des règles pour simuler son évolution. Une notation graphique aide les développeurs pour analyser et comprendre les programmes OA. Elle peut les aider aussi pour estimer les effets transversaux des aspects sur leurs classes de base. En plus; avec ce modèle nous pouvons extraire de petites vues pour une proposition spécifique (les voisins directs d'un élément nœuds + arcs). Par exemple, tous les points de coupure qui entrecoupent un point de jointure donné.

Des patrons de l'évolution: un des buts les plus importants de la recherche de l'évolution est la réutilisation des opérations de l'évolution, pour faciliter le processus de l'évolution d'une famille du logiciel. Dans notre travail les opérations d'évolution sont formulées comme des règles de réécriture. Une séquence de règles de réécriture pour une requête spécifique d'évolution peut être considérée comme un patron

d'évolution. Ce dernier peut être utilisé plus tard pour les requêtes d'évolution du même type. Par conséquent, cela peut augmenter la réutilisation des opérations d'évolution.

7.3 Comparaison aux Travaux Proches et Similaires

Cette section présente les travaux connexes en discutant les avantages de notre proposition par rapport aux autres. Notre travail touche les zones de recherche suivantes:

La modélisation basé-changement de l'évolution: en se basant sur notre étude du domaine, il y a peu de travaux dans cette zone de recherche. Le premier travail qui traite cette idée pour le logiciel orienté objet est celui de Lanza et al. [LAN07]. Ils représentent un état d'un programme comme un arbre de syntaxe abstrait (abstract syntax tree «AST») de son code source. Donc, les changements au programme sont représentés comme des opérations explicites du changement à son AST. Hattori et al. [HAT10] étendent ce modèle d'évolution du logiciel basé-changement dans un contexte multi-développeur. Ils modélisent l'évolution d'un système comme un ensemble contenant des séquences de changements. Bien que, l'idée de ces travaux est semblable à notre proposition où le changement est traité comme une entité de première classe, mais l'utilisation de l'AST n'est pas un bon choix pour l'analyse de l'évolution du logiciel. L'AST capture la structure du code source mais elle ne capture pas sa sémantique. Donc, le dépôt du changement n'est pas suffisant pour l'analyse de l'évolution. Par contre, notre proposition est prometteuse pour mieux améliorer et perfectionner le dépôt de changement. Nous pouvons capturer les informations structurelles et sémantiques au sujet du changement à travers les règles de réécriture.

La modélisation basée-graphe pour le logiciel OA. Une variété de modèles basés-graphe ont été proposés pour représenter les différentes caractéristiques des programmes OA [BER07, LEM07, PAR08, ZHA02a, ZHA03b]. Chacun de ces modèles met l'accent sur quelques caractéristiques spécifiques d'un programme OA pour des buts de slicing ou de test, mais pas pour l'évolution. La plupart d'eux sont des représentations à une granularité fine qui représentent un programme au niveau des "instructions" i.e. les nœuds représentent les instructions et les arcs représentent les différentes dépendances entre eux. Cependant, la tendance dans ces techniques de

l'évolution est d'utiliser des représentations à une grande granularité, qui représentent les différents composants de l'artefact (ex. aspects, consignes, méthodes) et les dépendances entre eux (ex membership, crosscutting). Notre approche suit ce principe pour modéliser l'évolution du logiciel OA.

Nagy et al. [NAG03] ont proposé une représentation basée-graphe des programmes OA pour raisonner au sujet des mécanismes de composition. Comparé avec notre représentation, leur graphe est trop complexe i.e. les attributs d'un nœud sont aussi représentés comme des nœuds (ex. le nom d'une classe, la visibilité, le type d'un attribut...etc.). Ce qui exige des arcs plus sophistiqués. En plus, les relations importantes sont représentées juste comme des attributs littéraux, par exemple: les appels entre les méthodes, les introductions…etc. Donc, leur modèle n'est pas convenable pour les propositions de l'évolution i.e. les dépendances entre les entités sont le centre du processus de l'évolution.

L'évolution du logiciel OA: Plusieurs techniques [CHA06, GRI06, KEL06, PIR11] existent pour traiter le problème de la fragilité des points de coupure. Par exemple, Kellens et al. [KEL06] proposent un modèle basé sur les points de coupure. Ils séparent les définitions du point de coupure de la structure actuelle du programme de base et les définissent à travers d'un modèle conceptuel du logiciel. Nous pensons que notre représentation abstraite du code source OA offerte par notre approche peut être utilisée complémentairement avec ces techniques pour alléger ce problème. Notre modèle représente d'une manière plus explicite et claire les dépendances entre les préoccupations transversales du système, aussi bien que toutes les dépendances entre les aspects et le code de base i.e. si nous changeons n'importe quel point de jointure, nous pouvons détecter les préoccupations transversales (Aspects), ou plus spécifiquement le ou les points de coupure en rapport avec ce point de jointure. En utilisant une des techniques précédentes, nous pouvons vérifier que les points de coupure sont bien implémentés dans notre modèle, et par conséquent, dans le code source OA.

Le formalisme de la réécriture de graphes n'est pas encore utilisé pour l'évolution des logiciels OA, mais plusieurs travaux prouvent son efficacité dans la vérification de ces logiciels [ABM99, AKS09, GLA97, HAV07, MEH06, MON05, STA06]. C'est

pourquoi; nous croyons qu'un modèle basé sur la réécriture de graphes est le meilleur choix pour l'évolution du logiciel OA.

Dépôts de version OA: les dépôts de version des systèmes de contrôle de version actuels ne sont pas satisfaisants pour l'analyse de l'évolution du logiciel OA (section 3.4). C'est pourquoi quelques travaux de recherche essaient d'adapter les systèmes de version actuels pour manier les caractéristiques du paradigme OA. Par exemple, le travail dans [IFR12] propose un mécanisme qui entrepose (checks-in) avec les versions du code source OA des métadonnées transversales pour suivre l'effet les aspects. Dans [ARI08] l'outil TOFRA est présenté pour traiter le problème de gestion de la configuration dans le contexte du Framework transversal (Crosscutting Frameworks «CFs») [DEC08]. Cependant, ces travaux gardent le mécanisme traditionnel du versioning classique (section 3.4.1). Ils n'enregistrent pas l'information complète au sujet de l'évolution du logiciel OA. Dans l'autre côté, l'analyse de leurs dépôts devient un défi de recherche parce que les données sont non structurées, non étiquetées. Par contre, notre travail fournit un dépôt basé-changement qui entrepose le processus complet d'évolution, ce qui facilite l'extraction du changement et améliore l'analyse de l'évolution.

La détection des patrons de changement: Il y a beaucoup de recherches faites sur l'extraction des patrons de changement pour les programmes procéduraux ou orientés objet [HAS08, KAG07]. Peu d'effort est consacré aux programmes OA. Qian et al. [QIA08] traitent la détection des patrons de changement dans les programmes AspectJ. Ils analysent en premier les versions consécutives d'un programme AspectJ, puis ils décomposent leurs différences dans un ensemble de changements atomiques. Finalement, ils emploient l'algorithme Apriori de Mining pour générer les ensembles d'items les plus fréquents. Cependant, ils se sont basés sur les dépôts des systèmes de version actuels qui ne sont pas complètement adaptés aux caractéristiques de la POA. Et ils ont besoin d'un processus sophistiqué pour extraire les changements atomiques et les transactions. Notre proposition évite ces problèmes où nous sommes basés sur un dépôt basé-règle. Ce dernier entrepose les changements quand ils se produisent dans un format plus précis et formel comme des règles de réécriture.

La modélisation du logiciel OA: Bien que ce ne soit pas le but de notre dissertation, nous pouvons mettre notre travail dans la zone de recherche qui traite la modélisation

du logiciel OA. Généralement tous les travaux pour la modélisation des systèmes OA se sont basés sur le langage UML; nous pouvons mentionner ici les travaux de Stein, Hanenberg et Unland [STEI02a, STEI02b, STEI06]. Avec la représentation UML chaque aspect du système est modélisé par un seul diagramme, par contre notre modèle donne une vue générale où tous les aspects du système sont modélisés ensemble. Donc, au lieu d'utiliser plusieurs modèles différents, nous proposons d'utiliser un seul modèle qui est semblable à un diagramme de classe mais aussi inclut les relations à partir des diagrammes d'interaction. En plus, c'est difficile de gérer l'évolution dans les diagrammes UML.

7.4 Perspectives de Recherche

Le travail de cette thèse consiste en la mise en place d'un Framework d'évolution pour le logiciel OA. Le travail réalisé peut se prolonger vers les perspectives suivantes:

Une perspective directe constituant une continuité de notre travail est l'évaluation du notre Framework sur plusieurs codes source OA de taille importante. Ainsi que, l'utilisation de notre approche de détection des patrons de changement et de relations logiques pour étudier le phénomène d'évolution de grandes applications OA.

Une autre perspective intéressante est la création des techniques de la détection de l'impact du changement, de la propagation, ou du Refactoring dédiées au logiciel OA en basant sur notre Framework proposé. Plus spécifiquement, les métadonnées extraites à partir de l'analyse de notre dépôt-basé règle de réécriture peuvent être employées pour supporter les différentes tâches de l'évolution du logiciel OA (ex. Refactoring).

D'un autre coté, des outils d'évolution basés sur notre Framework peuvent être conçus. Notre Framework peut être intégré à un outil CASE pour effectuer plusieurs tâches, par exemple pour visualiser le dépôt de règle de réécriture.

La perspective majeure de notre travail est de rendre notre Framework d'évolution beaucoup plus complet pour bien aider le mainteneur du logiciel OA. Nous pouvons considérer notre Framework d'évolution comme un assistant d'un mainteneur. L'idée est comme suit: le mainteneur initialise un changement, pendant que l'assistant fournit un aide à la décision et permet de rendre le système dans un état cohérent. Par

exemple, l'assistant détecte toutes les places (ex. préoccupations) qui sont affectées par un changement donné et les présentent au programmeur pour une mise à jour. Il propose des corrections et peut-être même dérive des corrections en observant le programmeur.

Finalement, une perspective intéressante de notre travail est d'étendre les mécanismes des systèmes de contrôle de versions classiques de façon à ce qu'ils gèrent l'évolution par une approche similaire à la notre. Les différentes tâches de versioning peuvent être traitées par le formalisme de la transformation de graphes. Nous parlons ici de fusionnement de versions, la détection de conflits…etc.

Bibliographie

[ABM99] U. Aßmann, and A. Ludwig, “Aspect weaving with graph rewriting”, *In Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE’99),* London, UK, 1999, Springer-Verlag, pp. 24–36.

[AGR94] R. Agrawal, and R. Srikant, “Fast algorithms for mining association rules in large databases”, *In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, Proceedings of 20th International Conference on Very Large Data Bases*, Santiago, Chile, September 12-15, 1994, pp. 487–499.

[AKS09] M. Aksit, A. Rensink, and T. Staijen, “A graph-transformation-based simulation approach for analysing aspect interference on shared join points”, *In Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD),* 2009, pp. 39-50.

[AKS92] M. Aksit, L. Bergmans, and S. Vural, “An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach”, *In Proceedings of the ECOOP 1992*, Utrecht, The Netherlands, June 29-July 3, 1992, Springer, 1992, LNCS 615, pp. 372-395.

[ALD04] J. Aldrich, “Open Modules: Reconciling Extensibility and Information Hiding”, *In Proceedings of Software Engineering Properties of Languages for Aspect Technologies, AOSD’04 Workshop*, 2004.

[ALE04] R.T. Alexander, J.M. Bieman, and A.A. Andrews, “Towards the Systematic Testing of Aspect-Oriented Programs”, Report CS-04-105, Colorado State University, Fort Collins-USA, March 2004.

[ALJ09] K. Aljasser, and P. Schachte, “ParaAJ: toward Reusable and Maintainable Aspect Oriented Programs”, *In Proceedings of the 32nd Australasian Computer Science Conference (ACSC’09),* Wellington, New Zealand, January 2009, pp. 53-62.

[ALL95] L. Allen et al., “ClearCase MultiSite:Supporting geographically-distributed software development”, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5*

Workshops, Springer-Verlag, October 1995, LNCS 1005, pp. 194-214.

[ALT09] K. Altmanninger, M. Seidl, and M. Wimmer, "A Survey on Model Versioning Approaches", *International Journal of Web Information Systems (IJWIS)*, 2009, Vol. 5(3), pp. 271 - 304.

[ARI08] M.M. Arimoto, M.I. Cagnin, and V.V. de Camargo, "Version control in crosscutting framework-based development", *In Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08),* Fortaleza, Ceara, Brazil, 2008, pp. 753–758.

[BAR02] L. Baresi, and R. Heckel, "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective", *In Proceedings of the International Conference on Graph Transformation,* 2002, Springer, LNCS 2505, pp. 402-439.

[BAR97] R. Bardohl, "Application of Graph Transformation to Visual Languages: State of the Art and Further Ideas", The Pennsylvania State University, March 1997.

[BAZ14] Bazaar Tutorial, doc.bazaar.canonical.com/beta/en/tutorials/ tutorial.html, [Accessed January 2014].

[BER07] M.L. Bernardi, and G.A.D. Lucca, "An interprocedural aspect control flow graph to support the maintenance of aspect oriented systems", *In Proceedings of the International Conference of Software Maintenance (ICSM'07),* 2007, pp. 435–444.

[BOO04] G. Booch, "Software Architecture Presentation", *Rational User Conference*, 2004, [online] http://www.booch.com/ architecture/blog/artifacts/Software Architecture. ppt.

[BOT03] P. Bottoni, F. Parisi-Presicce, G. Taenzer, "Specifying Integrated Refactoring with Distributed Graph Transformations," *In Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE'03),* Charlottesville, VA, USA, September 27-October 1 2003, Springer, LNCS 3062, pp. 220-235.

[BOU06] S. Bouktif, Y.G Guéhéneuc, and G. Antoniol, "Extracting Change-patterns from CVS Repositories", *In Proceedings of 13th Working*

Conference on Reverse Engineering (WCRE '06), 2006, pp. 221 – 230.

[BRI05] J. Brichau, and M. Haupt, "Survey of aspect-oriented languages and execution models", Technical Report AOSD-Europe- VUB-01, AOSD-Europe, May 2005.

[BUR04] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zundorf, "Tool Integration at the Meta-Model Level: the Fujaba Approach", *International Journal on Software Tools for Technology Transfer*, *Special section on tool integration applications and frameworks*, 2004, Vol 6(3), pp. 203-218.

[CHA06] C. Chavez, A. Garcia, T. Batista, M. Oliveira, C. Sant'Anna, and A. Rashid, "Composing Architectural Aspects Based on Style Semantics", *In Proceedings of the 6th international conference on Aspect-oriented software development (AOSD'09)*, 2006, pp. 111-122.

[CHE12] H. Cherait, and N. Bounour, "Modeling Software Evolution through Version Control System", *In Proceedings of Colloque Africain sur la Recherche en Informatique et Mathématiques Appliquées (CARI'12)*, Alger, Algérie, 13-16 Octobre 2012.

[CHE11] H. Cherait, and N. Bounour, "Toward a Version Control System for Aspect Oriented Software", *In Proceedings of 1st International Conference on Model & Data Engineering*, Portugal, September 28-30, 2011, Springer-Verlag, LNCS 6918, pp. 110–121, DOI: 10.1007/978-3-642-24443-8_13.

[CHE10] H. Cherait, and N. Bounour, "Software Evolution: Models and Challenges", *In Proceedings of International Conference on Machine and Web Intelligence (ICMWI'10)*, Algiers, Algeria, 2010, pp. 479-481.

[CHI04] S. Chiba, and K. Nakagawa, "Josh: An Open AspectJ-like Language", *In Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, March 2004, pp. 102–112.

[CHI05] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto Alarcon, J. Bakker, B. Tekinerdoğan, S. Clarke, and A. Jackson, "Survey of aspect-

oriented analysis and design approaches", Technical Report AOSD-Europe-ULANC-9, AOSD-Europe, May 2005.

[CLA99] J. Clark, XSL Transformations (XSLT) Version 1.0. Recommandation 16, november edition, 1999, [online] http://www.w3.org/TR/xslt.

[CLI03] C. Clifton, and G.T. Leavens, "Obliviousness, modular reasoning, and the behavioral subtyping analogy", *In Proceedings of the AOSD'03 workshop on Software engineering Properties of Languages for Aspect Technologies Conference (SPLAT)*, Boston, Massachusetts, March 18, 2003.

[COL04] A. Colyer, and A. Clement, "Largescale AOSD for middleware", *In Proceedings of the 3rd international conference on Aspect-oriented software development*, 2004, pp 56-65.

[COL03] A. Colyer, G. Blair, and A. Rashid, "Managing complexity in middleware", *In Proceedings of the Workshop on Aspects, Components and Patterns for Infrastructure Software (held with AOSD'03)*, Boston, USA, 2003, pp. 21-26.

[COL94] D.M. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability", *IEEE Computer Journal*, 1994, Vol 27(8), pp. 44–49.

[COR97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Lowe, "Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach", in G. Rozenberg editor. Handbook of Graph Grammars and Computing by Graph transformation, Vol 1: Foundations, World Scientific, Singapore, 1997, chapter 3, pp. 163-245.

[COR96] A. Corradini, U. Montanari, and F. Rossi, "Graph processes", *Fundamenta Informaticae*, 1996, Vol. 26**(**3/4), pp. 241–265.

[COR89] T. Corbi, "Program understanding: challenge for the 1990's", *IBM System Journal*, 1989, Vol 28(2), pp. 294–306.

[COS04] B. Collins-Sussman, B.W. Fitzpatrick, and C.M. Pilato, "Version Control with Subversion", O'Reilly Media, 2004.

[COU89] W. Courington, “The Network Software Environment”, Sun Technical Report FE197-0, Sun Microsystems Inc, February 1989.

[DEC08] V.V. De Camargo, and P.C. Masiero, “A pattern to design crosscutting frameworks”, *In Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC’08*), Fortaleza, Ceara, Brazil, 2008, pp. 759–764.

[DEL02] J. de Lara, and H. Vangheluwe, “AToM3: A Tool for Multi-Formalism Modelling and Meta-Modelling”, *In Proceedings of 5th International Conference, FASE’02 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS’02*, Grenoble, France, April 8–12, 2002, Springer, LNCS 2306, pp. 174 – 188.

[DOU05] R. Douence, and D. Le Botlan, “Towards a Taxonomy of AOP Semantics”, Technical Report AOSD-Europe, July 2005.

[DUF04] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge, “Measuring the dynamic behavior of AspectJ programs”, *In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA’04),* 2004, pp. 150-169.

[EHR06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, “Fundamentals of Algebraic Graph Transformation”, *EATCS Monographs in Theoretical Computer Science*, Springer, 2006, ISBN 978-3-540-31187-4.

[EHR97] H. Ehrig, R. Heckel, M. Korff, M. Lowe, L. Ribeiro, A. Wagner, and A. Corradini, “Algebraic Approaches to Graph Transformation II: Single Pushout Approach and comparison with Double Pushout Approach”, In G. Rozenberg, editor, Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations, World Scientific, 1997.

[EHR79] H. Ehrig, “Introduction to the algebraic theory of graph grammars (a survey)”, *In V. Claus, H. Ehrig and G. Rozenberg, eds., Graph grammars and their application to Computer Science and Biology*, Springer, Berlin, 1979, LNCS 73, pp. 1-69.

[EHR73] H. Ehrig, M. Pfender, and H.J. Schneider, “Graph grammars: An

algebraic approach", *In Proceedings of IEEE Conference on Automata and Switching Theory, SWAT '73,* 1973, pp. 167-180.

[ERL00] L. Erlikh, "Leveraging legacy system dollars for e-business", *IT Professional Journal,* 2000, Vol 2(3), pp. 17-23.

[FER10] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado, "An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs", *In Proceedings of International Conference on Software Engineering (ICSE'10),* Cape Town, South Africa, May 2-8 2010, pp. 65 – 74.

[FIG08] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F.C. Filho, and F. Dantas, "Evolving software product lines with aspects: An empirical study on design stability", *In Proceedings of the 30th International Conference on Software Engineering (ICSE'08),* Leipzig, Germany, 2008, pp. 261-270.

[FIL01] R.E. Filman, and D.P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", RIACS Technical Report 01.12, May 2001.

[GAL99] H. Gall, M. Jazayeri, and C. Riva, "Visualizing software release histories: The use of color and third dimension", *In Proceedings of International Conference on Software Maintenance (ICSM),* Oxford, 1999, pp. 99—108.

[GEI07] R. Geiß, M. Kroll, "GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool", *In Proceedings of Third International Symposium of Applications of Graph Transformations with Industrial Relevance (AGTIVE'07),* Kassel, Germany, October 10-12 2007, LNCS 5088, pp 568-569, [online] www.grgen.net.

[GER05] D.M. German, D. Cubranic, and M.A.D. Storey, "A Framework for Describing and Understanding Mining Tools in Software Development", *In Proceedings of the International Workshop on*

Mining software repositories (MSR '05), 2005, pp. 1-5.

[GLA97] J.R.W. Glauert, R. Kennaway, A.G. Papadopoulos, and R. Sleep, "Dactl: an experimental graph rewriting language", *Journal of Programming Languages*, 1997, Vol. 5, pp. 85–108.

[GOD01] C. Godsil, and G.F. Royle, "Algebraic Graph Theory", *Graduate Texts in Mathematics*, Springer, 2001, Vol. 207, ISBN 978-0-387-95220-8.

[GRI06] W.G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, H. Rajan, "Modular software design with crosscutting interfaces", *IEEE Journal of Software*, 2006, Vol. 23(1), pp. 51-60.

[GRU86] D. Grune, "Concurrent Versions System, A Method for Independent Cooperation", Vrije University, Amsterdam, Rapport R-114, 1986, [online] http://savannah.nongnu.org/projects/cvs.

[GYB03] K. Gybels, and J. Brichau, "Arranging Language Features for More Robust Pattern-Based Crosscuts", *In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, April 2003, pp. 60–69.

[HAM05] J. Hamano, and L. Torvalds, "Git - Fast Version Control System", 2005, [online] http://git-scm.com/.

[HAR93] W.H. Harrison, and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *In Proceedings of the OOPSLA*, Washington, DC, USA, 26 September- 1 October 1993, pp. 411-428, In SIGPLAN Notices, 1993, Vol 28(10).

[HAS08] A.E. Hassan, "The road ahead for mining software repositories", *In Proceedings of Frontiers of Software Maintenance***,** Beijing, 2008**,** pp. 48–57.

[HAS03] A.E. Hassan, and R.C. Holt, "The Chaos of Software Development", *In Proceedings of the International Workshop on Principles of Software Evolution (IWPSE'03)*, Helsinki, Finland, Sept. 2003, pp. 84 – 94.

[HAT11] L. Hattori, M. D'Ambros, M. Lanza and M. Lungu, "Software Evolution Comprehension: Replay to the Rescue", *In Proceedings of*

IEEE 19th International Conference on Program Comprehension (ICPC), 2011, pp. 161 – 170.

[HAT10] L. Hattori, and M. Lanza, "Syde: A tool for collaborative software development", *In Proceedings of 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 235–238.

[HAV07] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit, "A Graph-Based Approach to Modeling and Detecting Composition Conflicts Related to Introductions", *In Proceedings of the 6th international conference on Aspect-Oriented Software Development (AOSD '07)*, Vancouver, Canada, 2007, pp. 85-95.

[HEC02] R. Heckel, J.M. Kuster, and G. Taentzer, "Confluence of Typed Attributed Graph Transformation Systems", *In Proceedings of ICGT'02*, Barcelona, Spain, 2002, Springer-Verlag, LNCS 2505, pp. 161-176.

[HIN05] A.J. Hindle, and D.M. German, "SCQL: A Formal Model and a Query Language for Source Control Repositories", *In Proceedings of the International workshop on Mining software repositories*, 2005, pp. 100-104.

[IBM01] IBM, HyperJ: Multi-Dimensional Separation of Concerns for Java, October 2001, [online] http://www.alphaworks.ibm.com/tech/hyperj.

[IFR12] S. Ifrah, and D.H. Lorenz, "Crosscutting Revision Control System", *In Proceedings of International Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 321-330.

[KAG07] H. Kagdi, M.L Collard and J.I. Maletic, "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution", *Journal of Software Maintenance and Evolution: Research and Practice*, 2007, vol. 19, no. 2, pp. 77-131.

[KAS07] C. Kästner, S. Apel, and D. Batory, "A Case Study Implementing Features using AspectJ", *In Proceedings of the 11th International Conference of Software Product Line (SPLC'07)*, Kyoto, Japan, 2007, pp. 223-232.

[KAT05] S. Katz, “A Survey of Verification and Static Analysis for Aspects”, Technical Report AOSD-Europe, July 2005.

[KEL06] A. Kellens, K. Mens, J. Brichau, and K. Gybels, “Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts”, *In Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP’06)*, Nantes, France, Springer-Verlag, 2006, LNCS 4067, pp. 501-525.

[KIC97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, “Aspect-oriented programming”, *In Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP’97)*, 1997, Springer-Verlag, LNCS 1241, pp. 220–242.

[KNI04] G. Kniesel, P. Costanza, and M. Austermann, “JMangler - A Powerful Back-End for Aspect-Oriented Programming”, *In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, Aspect-oriented Software Development*, chapter 9, Prentice Hall, 2004.

[KOE09] M. Koegel, M. Herrmannsdoerfer, J. Helming, and Y. Li, “State-based vs. Operation-based Change Tracking”, *In Proceedings of MODELS ’09 MoDSE-MCCM Workshop*, Denver, USA, 2009.

[KRO02] G. Kroah-Hartman, “Kernel korner: the kernel hacker's guide to source code control”, *Linux Journal*, 2002, Vol. 101, pp. 11.

[LAD10] R. Laddad, “AspectJ in Action”, Second Edition 2010.

[LAN07] M. Lanza, and R. Robbes, “A Change-based Approach to Software Evolution”, *Electronic Notes in Theoretical Computer Science (ENTCS).* Elsevier, 2007, Vol. 166, pp. 93-109.

[LEH76] M.M. Lehman, and L.A. Belady, “A model of large program development”, *IBM Sysem Journal*, 1976, Vol 15(1), pp. 225-252.

[LEM07] O.A.L. Lemos, A.M.R. Vincenzi, J.C. Maldonado, and P.C. Masiero, “Control and data flow structural testing criteria for aspect-oriented programs”, *Journal of Systems and Software,* Elsevier, 2007, Vol. 80(6), pp. 862-882.

[LIE96] K.J. Lieberherr, “Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns”, PWS Publishing Company, 1996, ISBN. 053494602X.

[LOP95] C.V. Lopes, and W.L. Hursch, "Separation of Concerns", Technical Report UN-CCS-95-03, Northeastern University, Boston, February 1995.

[LOU05] N. Loughran, N. Parlavantzas, M. Pinto, P. Sánchez, M. Webster, and A. Colyer, “Survey of Aspect-Oriented Middleware”, AOSD-Europe Project Deliverable No: AOSD-Europe-ULANC-10, Editor(s): N. Loughran, M. Pinto, June 2005.

[LOW93] M. Lowe, and M. Beyer, “AGG- An Implementation of Algebraic Graph Rewriting”, *In C. Kirchner, editor, Proceedings of Rewriting Techniques and applications*, springer, 1993, LNCS 690, pp. 451-456.

[MAC05] M. Mackall, Mercurial, April 2005, [online] http://www.mercurial.selenic.com/.

[MAL10] C.F. Malmsten, “Evolution of Version Control Systems: Comparing Centralized against Distributed Version Control models”, Report No. 2010:017, University of Gothenburg, Department of Applied Information Technology, Gothenburg, Sweden, May 2010, ISSN: 1651-4769.

[MEH06] K. Mehner, M. Monga, and G. Taentzer, “Interaction Analysis in Aspect-Oriented Models”, *In Proceedings of the 14th IEEE International Requirements Engineering Conference (RE’06)*, 2006, pp. 66-75.

[MEL05] L.S. Melo Junior, and N.C. Mendonça, “AspectJML: A Markup Language for AspectJ”, *In Proceedings of the 2nd Brazilian Workshop on Aspect Oriented Software Development (WASP'05),*Uberlândia, MG, Brazil, 2005.

[MEN02a] T. Mens, S. Demeyer, and D. Janssens, “Formalizing behavior preserving program transformations”, *In Proceedings of International Conference on Graph-transformation*, 2002, pp. 286–301.

[MEN02b] T. Mens, “A state-of-the-art survey on software merging”, *IEEE Transaction of Softw*are *Engineering*, 2002, Vol. 28(5), pp. 449–462.

[MEN01] T. Mens, “A Formal Foundation for Object Oriented Software Evolution”, *In Proceedings of* 17th IEEE *International Conference on Software Maintenance (ICSM’01)*, Florence, Italy, 2001, pp. 549 – 552.

[MEN00] T. Mens, “Conditional graph rewriting as a domain-independent formalism for software evolution”, *In Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Springer-Verlag, 2000, LNCS 1779, pp. 127-143.

[MEN99] T. Mens, “A Formal Foundation for Object Oriented Software Evolution”, PH.D. Dissertation, Vrije Universiteit Brussel, Faculty Of Sciences, Department Of Computer Science, Programming Technology Lab, August 27, 1999.

[MENk07] K. Mens, and T. Tourwé, “Evolutionary Problems in Aspect-Oriented Software Development”, *In Proceedings of the Third International ERCIM Symposium on Software Evolution*, 2007.

[MENt07] T. Mens, G. Taentzer, and O. Runge, “Analyzing refactoring dependencies using graph transformation”, *Journal of Software and Systems Modeling*, Springer, 2007, pp. 269-285.

[MON05] M. Monga, “Aspect-oriented programming as model driven evolution”, *In Proceedings of the linking aspect technology and evolution workshop* (*LATE’05),* Chicago, IL USA, 2005.

[MUK11] P Mukherjee, “A Fully Decentralized, Peer-To-Peer Based Vers Ion Control System”, PhD dissertation, Technische Universität Berlin, Germany, 2011.

[MUN08] F. Munoz, B. Baudry, and O. Barais, “A classification of invasive patterns in AOP”, *In Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM’08)*, Beijing, China, 2008.

[MUR08] L. Murta, C. Corrêa, J. G. Prudêncio, and C. Werner, “Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System”, *In Proceedings of the 2008 international workshop on*

Comparison and versioning of software models (CVSM'08), Leipzig, Germany, May 17, 2008, pp. 25-30.

[NAG06] I. Nagy, "On the Design of Aspect-Oriented Composition Models for Software Evolution", PhD Thesis, University of Twente, 2006, ISBN 90-365-2368-0.

[NAG05] W. Nagel, "Subversion Version Control: Using The Subversion Version Control System in Development Projects", Prentice Hall Professional Technical Reference, 14 April 2005.

[NAG03] I. Nagy, M. Aksit, and L. Bergmans, "Composition Graphs: a Foundation for Reasoning about Aspect-Oriented Composition", *In Proceedings of Foundations of Aspect-Oriented Langauges Workshop (FOAL'03),* Boston, MA, USA, 2003.

[NIC00] U. Nickel, J. Niere, A. Zündorf, "The FUJABA environment**",** *In Proceedings of the 22nd International Conference on Software Engineering (ICSE '00),* 2000, pp. 742-745, [online] http://www.fujaba.de.

[OSS99] H. Ossher, and P.L Tarr, "Multi-Dimensional Separation of Concerns in Hyperspaces", *Position Paper at the AOP Workshop at ECOOP 1999,* Lisbon, Portugal, June 14-18, 1999.

[PAR08] R.M. Parizi, and A.A. Abdul Ghani, "AJcFgraph - AspectJ Control Flow Graph Builder for Aspect-Oriented Software", *International Journal of Electrical and Computer Engineering*, 2008, Vol. 3(3), pp. 170-181.

[PAR72] D.L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, December 1972, Vol 15(12), pp. 1053-1058.

[PAW05] R. Pawlak, L. Seinturier, J.P. Retaillé, "JBoss AOP", In Book "Foundations of AOP for J2EE Development", Apress, 2005, pp 91-112, [online] http://www. jbossaop.jboss.org.

[PAW01] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier, " JAC: A Flexible Solution for Aspect-Oriented Programming in Java", *In Proceedings of*

Third International Conference REFLECTION, Kyoto, Japan, September 25–28, 2001, Springer, LNCS 2192, 2001, pp 1-24, [online] http://jac.aopsys.com/.

[PER10] J. Pérez, Y. Crespo, B. Hoffmann, and T. Mens, "A Case Study to Evaluate the Suitability of Graph Transformation Tools for Program Refactoring", *International Journal on Software Tools for Technology Transfer*, Vol. 12(3-4) , 2010, pp 183-199.

[PIR11] P.F. Pires, F.C. Delicato, M. Pinto, L. Fuentes, and É. Marinho, "Software Evolution in AOSD: A MDA-based Approach", *In Proceedings of 14th international ACM Sigsoft symposium on Component based software engineering CBSE'11*, Boulder, Colorado, USA, 2011, pp. 193-197.

[PRE09] R. Pressman, "Software Engineering: A Practitioner's Approach", McGraw-Hill, 7th edition, 2009.

[PRZ10] A. Przybylek, "What is Wrong with AOP*?"*, *In Proceedings of 5th International Conference on Software and Data Technologies*, Athens, Piraeus, 2010.

[QIA08] Y. Qian, S. Zhang, and Z. Qi, "Mining Change Patterns in AspectJ Software Evolution", *In Proceedings of International Conference on Computer Science and Software Engineering*, 2008, pp. 108-111.

[RAJ97] V. Rajlich, "A Model for Change Propagation Based on Graph Rewriting*"*, *In Proceedings of ICSM'97*, Bari, Italy, September 28 - October 2, 1997, pp. 84-91.

[RAS10] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Südholt, and W. Joosen, "Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe", *Computer Journal*, IEEE Computer Society, February 2010, Vol 43(2), pp. 19–26.

[RAS04] A. Rashid, and N. Leidenfrost, "Supporting Flexible Object Database Evolution with Aspects", *In Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE'04)*, Vancouver, Canada, 2004, pp. 75-94.

[RAT05] J. Ratzinger, M. Fischer, and H. Gall, “Improving Evolvability through Refactoring”, *In Proceedings of international workshop on Mining software repositories*, Saint Louis, Missouri, USA, 2005, pp. 1-5.

[REE95] T. Reenskaug, P. Wold, and O.A. Lehne, “Working with Objects: The OORam Software Engineering Method”, Manning/Prentice Hall, Upper Saddle River, New Jersey, 1995.

[RIN04] M. Rinard, A. Salcianu, and S. Bugrara, “A classification system and analysis for aspect-oriented programs”, *In ACM SIGSOFT Software Engineering Notes*, 2004, Vol. 29(6), pp. 147–158.

[ROB05] R. Robbes, and M. Lanza, “Versioning systems for evolution research”, *In Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE’05)*, IEEE Computer Society, 2005, pp. 155–164.

[ROC75] M.J. Rochkind, “The Source Code Control System”, *IEEE Transactions on Software Engineering*, 1975, Vol. 1(4), pp. 364–370.

[RYS04] F.V. Rysselberghe, and S. Demeyer, “Studying software evolution information by visualizing the change history”, *In Proceedings of International Conference on Software Maintenance*, 2004, pp. 328—337.

[SCH95] A. Schurr, A. Winter, and A. Zundorf, “Graph Grammar Engineering with PROGRES”, *In Proceedings of the Europe Software Engineering Conference*, 1995, pp. 219-234.

[SEI08] L.M. Seiter, “Balancing quantification and obliviousness in the design of aspect-oriented frameworks”, *In Proceedings of the 10th international conference on Software Reuse: High Confidence Software Reuse in Large Systems*, Springer, 2008, LNCS 5030, pp. 318-329.

[SER05] D. Sereni, and O. de Moor, “Static analysis of aspects”, *In Proceedings of the Aspect Oriented Software Development Conference (AOSD’05)*, 2005, pp. 30–39.

[SHE02a] H. Shen, and C. Sun, “Flexible merging for asynchronous collaborative systems”, *In Proceedings of Confederated International Conferences*

CoopIS, DOA, and ODBASE, 2002, Springer, LNCS 2519, pp. 304–321.

[SHE02b] H. Shen and C. Sun, "A log compression algorithm for operation-based version control systems", *In Proceedings of 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, 2002, pp. 867 – 872.

[SHI05] H. Shinomi, and T. Tamai, "Impact analysis of weaving in aspect-oriented programming", *In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 657–660.

[SOA02] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with AspectJ", *In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 2002, pp. 174 – 190.

[SPI02] O. Spinczyk , A. Gal , W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language", *In Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications CRPIT '02*, Australia, 2002, pp. 53-60, [online] http://www.aspectc.org/.

[STA06] T. Staijen, and A. Rensink, "A GraphTransformation-Based Semantics for Analysing Aspect Interference", *In Proceedings of Workshop on Graph Computation Models*, Natal, Brazil, 18 Sept 2006.

[STA84] T. Standish, "An essay on software reuse", *IEEE Transactions on Software Engineering*, September 1984, Vol 10(5), pp. 494–497.

[STE06] F. Steimann, "The Paradoxical Success of Aspect-Oriented Programming", *In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA'06)*, 2006, pp. 481-497.

[STEI06] D. Stein, S. Hanenberg, and R. Unland, "Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design", *In Proceedings of the 5th international conference on Aspect-oriented software development (AOSD'06)*, Bonn, Germany, March 20-24, 2006, pp. 15 – 26.

[STEI02a] D. Stein, S. Hanenberg, and R. Unland, “Designing Aspect-Oriented Crosscutting in UML”, *In Proceedins of Workshop on Aspect Oriented Modeling with UML (AOSD’02)*, 2002.

[STEI02b] D. Stein, S. Hanenberg, and R. Unland, “An UML-based Aspect-Oriented Design Notation For AspectJ”, *In Proceeding of AOSD 2002*, Enschede, The Netherlands, 2002, pp. 106-112.

[STO05] M. Stoerzer, and J. Graf, “Using pointcut delta analysis to support evolution of aspect-oriented software”, *In Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM’05)*, Washington, DC, USA, 2005, pp. 653–656.

[SUZ98] J. Suzuki, and Y. Yamamoto, “Managing the software design documents with xml”, *In Proceedings of the 16th annual international conference on Computer documentation*, ACM Press: New York, 1998, pp. 127-136.

[TAE12] G. Taentzer, C. Ermel, P. Langer and M. Wimmer, “A fundamental approach to model versioning based on graph modifications: from theory to implementation”, *Journal of Software and Systems Modeling*, Springer-Verlag, 2012, Vol. 13(1), pp. 239-272.

[TAY02] C.M.B. Taylor and M. Munro, “Revision towers”, *In Proceedings of Workshop on Visualizing Software for Understanding and Analysis VISSSOFT*, Paris, France, 2002, pp. 43-50.

[TIC10] W. Tichy, “An Interview with Prof. Andreas Zeller : Mining your way to software reliability”, Ubiquity, an ACM publication, November 2010.

[TIC88] W. Tichy, “Software Configuration Management Overview”, 1988.

[TIC85] W.F. Ticky, “RCS: a system for version control”, *Journal of Software Practice and Experience*, 1985, Vol. 15(7), pp. 637-654.

[TOU04] T. Tourwé, A. Kellens, W. Vanderperren, and F. Vannieuwenhuyse, “Inductively Generated Pointcuts to Support Refactoring to Aspects”, *In Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT’04)*, Lancaster, UK, March 2004.

[TOU03] T. Tourwé, J. Brichau, and K. Gybels, “On the Existence of the AOSD-Evolution Paradox”, *In Proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT’03) at AOSD’03*, Boston, Massachusetts, 2003.

[VAS04] A. Vasseur, “Dynamic AOP and Runtime Weaving for Java- How Does AspectWerkz Address It?”, *In R. E. Filman, M. Haupt, K. Mehner, and M. Mezini, editors, Proceedings of the 2004 Dynamic Aspect Workshop (DAW’04)*, Lancaster, England, March 2004, pp. 135–145.

[VOI07] L. Voinea and A. Telea, “Visual data mining and analysis of software repositories”, *Journal of Computers & Graphics*, 2007, Vol 31, pp. 410–428.

[VOL02] D. Vollmann, “Visibility of join-points in aop and implementation languages”, *In Proceedings of the Second Workshop on Aspect-Oriented Software Developement (AOSD’02)*, Bonn, Germany, 2002, pp. 65–69.

[WAN03] J.W.W. Wan, and G. Dobbie, “Extracting Association Rules from XML Documents using XQuery”, *In Proceedings of the 5th ACM international workshop on Web information and data management (WIDM’03)*, New Orleans, Louisiana, USA, November 7–8, 2003, pp. 94-97.

[WEI98] J. Weidl, and H. C Gall, “Binding object models to source code: An approach to object-oriented re-architecting”, *In Proceedings of the 22nd International Computer Software and Applications Conference*, 1998, pp. 26–31.

[WIN01] A. Winter, B. Kullbach, and V. Riediger, “An overview of the GXL graph exchange language”, *In Proceedings of International Seminar Dagstuhl Castle*, Germany, 2001, Springer-Verlag, LNCS 2269, pp. 324-336.

[WON10] S. Wong, Y. Cai, and M. Dalton, “Change Impact Analysis with Stochastic Dependencies”, Department of Computer Science, Drexel University, Technical Report DU-CS-10-07, October 2010.

[XUR08] G. Xu, and A. Rountev, “AJANA: A general framework for source-

code-level interprocedural dataflow analysis of AspectJ software", *In Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD'08)*, Brussels, Belgium, 2008, pp. 36–47.

[XUR07] G. Xu, and A. Rountev, "Regression test selection for AspectJ software", *In Proceedings of the International Conference of Software Engineering (ICSE'07)*, 2007, pp. 65–74.

[XUR04] J. Xu, H. Rajan, and K. Sullivan, "Understanding aspects via implicit invocation", *In Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*, 2004, pp. 332-335.

[YIN05] A.T.T. Ying, J.L. Wright, and S. Abrams, "Source code that talks: an exploration of Eclipse task comments and their implication to repository mining", *In Proceedings of International Workshop on Mining Software Repositories (MSR)*, Saint Louis, Missouri, USA, 2005, pp. 1-5.

[ZHA08a] J. Zhao, "Maintenance Support for Aspect-Oriented Programs: Opportinuties and Challenges", *In Proceedings of the International Conference of Software Maintenance (ICSM'08)*, 2008, pp. 482-483.

[ZHA08b] S. Zhang, Z. Gu, Y. Lin, and J. Zhao, "Change impact analysis for AspectJ programs", *In Proceedings of the 24st IEEE International Conference on Software Maintenance*, Beijing, China, 2008, pp. 87 – 96.

[ZHA06] J. Zhao, T. Xie, and N. Li, "Towards regression test selection for AspectJ programs", *In Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, 2006, pp. 21 – 26.

[ZHA04] J. Zhao, "Control-flow analysis and representation for aspect oriented programs", *In Proceedings of the Sixth International Conference on Quality Software*, 2004, pp. 38–48.

[ZHA03a] J. Zhao, and M. Rinard, "System dependence graph construction for aspect-oriented programs", Technical Report MIT-LCS-TR-891, Laboratory for Computer Science, MIT, March 2003.

[ZHA03b] J. Zhao, "Data-Flow-Based Unit Testing of Aspect-Oriented Programs", *In Proceedings of 27th Annual IEEE International Computer Software*

and Applications Conference (COMPSAC'03), Dallas, Texas, 2003, pp. 188-197.

[ZHA02a] J. Zhao, "Slicing aspect-oriented software", *In Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC'02)*, Paris, French, 2002, pp. 251–260.

[ZHA02b] J. Zhao, "Change impact analysis for aspect-oriented software evolution", *In Proceedings of the 5th International Workshop on Principles of Software Evolution*, May 2002, pp. 108–112.

[ZHA01] K. Zhang, D.Q. Zhang, and J. Cao, "Design, Construction, and Application of a Generic Visual Language Generation Environment", *IEEE Transaction on Software Engineering*, 2001, Vol 27(4), pp. 289-307.

[ZIM05] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes", *IEEE Transactions on Software Engineering*, 2005, Vol 31(6), pp. 429–445.

ANNEXES

ANNEXE A

La Représentation XML de la Classe Point du Listing 7.1

```
<class id="I_Point" idkind="type" name="Point">
   <field id="I_x" idkind="field" name="_x">
      <modifiers><modifier name="private"/></modifiers>
      <type name="int" primitive="true"/>
      <var-initializer>
         <literal-number kind="integer" value="0"/>
      </var-initializer>
   </field>
   <field id="I_y" idkind="field" name="_y">
      <modifiers><modifier name="private"/></modifiers>
      <type name="int" primitive="true"/>
      <var-initializer>
         <literal-number kind="integer" value="0"/>
      </var-initializer>
   </field>
   <method id="I_getX()" idkind="method" name="getX">
      <type name="int" primitive="true"/>
      <block>
         <return>
            <field-ref idkind="field" idref="I_x" name="_x"/>
         </return>
      </block>
   </method>
   <method id="I_getY()" idkind="method" name="getY">
      <type name="int" primitive="true"/>
      <block>
         <return>
            <field-ref idkind="field" idref="I_y" name="_y"/>
         </return>
      </block>
   </method>
   <method id="I_setX()" idkind="method" name="setX">
      <type name="void" primitive="true"/>
      <formal-arguments>
         <formal-argument id="I_arg1" idkind="formal" name="x">
            <type name="int" primitive="true"/>
         </formal-argument>
      </formal-arguments>
      <block>
         <assignment-expr op="=">
            <lvalue>
```

```
                <field-set idkind="field" idref="I_x" name="_x"/>
            </lvalue>
            <formal-ref idkind="formal" idref="I_arg1" name="x"/>
        </assignment-expr>
    </block>
</method>
<method id="I_setY()" idkind="method" name="setY">
    <type name="void" primitive="true"/>
    <formal-arguments>
        <formal-argument id="I_arg2" idkind="formal" name="y">
            <type name="int" primitive="true"/>
        </formal-argument>
    </formal-arguments>
    <block>
        <assignment-expr op="=">
            <lvalue>
                <field-set idkind="field" idref="I_y" name="_y"/>
            </lvalue>
            <formal-ref idkind="formal" idref="I_arg2" name="y"/>
        </assignment-expr>
    </block>
    </method>
</class>
```

ANNEXE B

La Représentation XML de l'Aspect UpdateDisplay du Listing 7.1

```
<aspect id="I_UpdateDisplay;" idkind="type" name="UpdateDisplay">
   <constructor id="LUpdateDisplay" idkind="constructor"
   name="UpdateDisplay">
      <block>
         <super-call>
            <arguments/>
         </super-call>
      </block>
   </constructor>
   <intertype-member-declaration id="I_interField1" idkind="intertype-field-
   declaration" name="interField1">
      <type idkind="type" idref="I_Point" name="Point"/>
      <field id="I_name" idkind="type" name="name">
         <modifiers>
            <modifier name="public"/>
            <modifier name="static"/>
         </modifiers>
         <type name="String" primitive="true"/>
      </field>
   </intertype-member-declaration>
   <intertype-member-declaration id="I_interMethod1" idkind="intertype-
   method-declaration" name="interMethod1">
      <type idkind="type" idref="I_Point" name="Point"/>
      <method id="I_setName()" idkind="type" name="setName">
         <modifiers>
            <modifier name="public"/>
            <modifier name="static"/>
         </modifiers>
         <type name="void" primitive="true"/>
         <formal-arguments>
            <formal-argument id="I_arg3" idkind="formal" name="name">
               <type idkind="type" idref="Ljava/lang/String;" name="String"/>
            </formal-argument>
         </formal-arguments>
         <block>
            <assignment-expr op="=">
               <lvalue>
                  <field-ref idkind="field" idref="I_name" name="name">
                     <this/>
                  </field-ref>
               </lvalue>
               <formal-ref idkind="formal" idref="I_arg3" name="name"/>
```

```
        </assignment-expr></block></method>
</intertype-member-declaration>
<intertype-member-declaration id="I_interMethod2" idkind="intertype-
method-declaration" name="interMethod2">
   <type idkind="type" idref="I_Point" name="Point"/>
   <method id="I_getName()" idkind="type" name="getName">
      <modifiers>
         <modifier name="public"/>
         <modifier name="static"/>
      </modifiers>
      <type idkind="type" idref="Ljava/lang/String;" name="String"/>
      <formal-arguments/>
      <block>
         <return>
            <field-ref idkind="field" idref="I_name" name="name"/>
         </return>
      </block>
   </method>
</intertype-member-declaration>
<pointcut id="I_move" idkind="pointcut" name="move">
   <formal-arguments/>
   <binary-pointcut-expridkind="pointcut-expr" op="||">
      <method-constructor-pointcut-expridkind="pointcut-expr"
      type="method-call">
         <method-patt>
            <type idkind="type" name="void"/>
            <class-type-dot-id simple-name-patt="setX">
               <type idkind="type" name="Point"/>
            </class-type-dot-id>
            <formal-arguments-patt>
               <formal-argument-patt>
                  <type idkind="type" name="int"/>
               </formal-argument-patt>
            </formal-arguments-patt>
         </method-patt>
      </method-constructor-pointcut-expr>
      <method-constructor-pointcut-expridkind="pointcut-expr"
      type="method-call">
         <method-patt>
            <type idkind="type" name="void"/>
            <class-type-dot-id simple-name-patt="setY">
               <type idkind="type" name="Point"/>
            </class-type-dot-id>
            <formal-arguments-patt>
               <formal-argument-patt>
                  <type idkind="type" name="int"/>
               </formal-argument-patt>
            </formal-arguments-patt>
         </method-patt>
```

```
            </method-constructor-pointcut-expr>
        </binary-pointcut-expr></pointcut>
        <advice id="I_before" idkind="advice" kind="before" name="before1">
            <formal-arguments-pointcut-expridkind="pointcut-expr"
            type="pointcut-ref">
                <pointcut-ref idref="I_move"/>
            </formal-arguments-pointcut-expr>
            <modifiers>
                <modifier name="public"/>
            </modifiers>
            <type name="void" primitive="true"/>
            <formal-arguments/>
            <block>
                <send idkind="method"
                idref="Ljava/io/PrintStream;println(Ljava/lang/String;)V"
                message="println">
                    <target>
                        <field-ref idkind="field" idref="Ljava/lang/System;out"
                        name="out"/>
                    </target>
                    <arguments>
                        <literal-string value="figure is going to be displaced"/>
                    </arguments>
                </send>
            </block>
        </advice>
        <advice id="I_after" idkind="advice" kind="after" name="after1">
            <formal-arguments-pointcut-expridkind="pointcut-expr"
            type="pointcut-ref">
                <pointcut-ref idref="I_move"/>
            </formal-arguments-pointcut-expr>
            <modifiers>
                <modifier name="public"/>
            </modifiers>
            <type name="void" primitive="true"/>
            <formal-arguments/>
            <block>
                <send idkind="method" idref="I_update()" message="update">
                    <target>
                        <type idkind="type" idref="I_Display;" name="Display"/>
                    </target>
                    <arguments/>
                </send>
            </block>
        </advice>
    </aspect>
```

ANNEXE C

La Représentation GXL du Listing 7.1

```
<gxl>
   <graph id="C:\AspectJML\Point.java">
      <node value="Class" id="I_Point">
         <attrname="name"><String>Point</String></attr>
      </node>
      <node value="Attribute" id="I_x">
         <attr name="name"><String>_x</String></attr>
         <attr name="Visibility"><String>private</String></attr>
         <attr name="Type"><String>int</String></attr>
      </node>
      <edge value="Has Attribute" from="LPoint;" to="I_x" />
      <node value="Attribute" id="I_y">
         <attr name="name"><String>_y</String></attr>
         <attr name="Visibility"><String>private</String></attr>
         <attr name="Type"><String>int</String></attr>
      </node>
      <edge value="Has Attribute" from="LPoint;" to="I_y" />
      <node value="Method" id="I_getX()">
         <attr name="name"><String>getX</String></attr>
      </node>
      <edge value="Has Method" from="I_Point" to="I_getX()" />
      <node value="Return value" id="IDATJUU">
         <attr name="Type"><String>int</String></attr>
      </node>
      <edge value="Returns" from="I_getX()" to="IDATJUU" />
      <node value="Method" id="I_getY()">
         <attr name="name"><String>getY</String></attr>
      </node>
      <edge value="Has Method" from="I_Point" to="I_getY()" />
      <node value="Return value" id="IDAAKUU">
         <attr name="Type"><String>int</String></attr>
      </node>
      <edge value="Returns" from="I_getY()" to="IDAAKUU" />
      <node value="Method" id="I_setX()">
         <attr name="name"><String>setX</String></attr>
      </node>
      <edge value="Has Method" from="I_Point" to="I_setX()" />
      <node value="Parameter" id="I_arg1">
         <attr name="name"><String>x</String></attr>
         <attr name="Type"><String>int</String></attr>
      </node>
      <edge value="Takes parameter" from="I_arg1" to="I_setX()" />
      <node value="Method" id="I_setY()">
         <attr name="name"><String>setY</String></attr>
```

```
</node>
<edge value="Has Method" from="I_Point" to="I_setY()" />
<node value="Parameter" id="I_arg2">
   <attr name="name"><String>y</String></attr>
   <attr name="Type"><String>int</String></attr>
</node>
<edge value="Takes parameter" from="I_arg2" to="I_setY()" />
<node value="Class" id="I_Display">
   <attr name="name"><String>Display</String></attr>
</node>
<node value="Method" id="I_update()">
   <attr name="name"><String>update</String></attr>
</node>
<edge value="Has Method" from="I_Display" to="I_update()" />
<node value="Aspect" id="I_UpdateDisplay">
   <attrname="name"><String>UpdateDisplay</String></attr>
</node>
<node value="Attribute" id="I_name">
   <attr name="name"><String>name</String></attr>
   <attr name="Visibility"><String>public</String></attr>
   <attr name="Type"><String>String</String></attr>
</node>
<edge value="Introduce_Attribute" from="I_UpdateDisplay" to="I_name"
/>
<edge value="Introduced_to" from="I_name" to="I_Point" />
<node value="Method" id="I_setName()">
   <attr name="name"><String>setName</String></attr>
   <attr name="Visibility"><String>public</String></attr>
</node>
<edge value="Introduce_Method" from="I_UpdateDisplay"
to="I_setName()" />
<edge value="Introduced_to" from="I_setName()" to="I_Point" />
<node value="Parameter" id="I_arg3">
   <attr name="name"><String>name</String></attr>
   <attr name="Type"><String>String</String></attr>
</node>
<edge value="Takes parameter" from="I_arg3" to="I_setName()" />
<node value="Method" id="I_getName()">
   <attr name="name"><String>getName</String></attr>
   <attr name="Visibility"><String>public</String></attr>
</node>
<edge value="Introduce_Method" from="I_UpdateDisplay"
to="I_getName()" />
<edge value="Introduced_to" from="I_getName()" to="I_Point" />
<node value="Return value" id="IDAHPUU">
   <attr name="Type"><String>String</String></attr>
</node>
<edge value="Returns" from="I_getName()" to="IDAHPUU" />
<node value="Pointcut" id="I_move">
```

```
            <attr name="name"><String>move</String></attr>
        </node>
        <edge value="Has Pointcut" from="I_UpdateDisplay" to="I_move"/>
        <edge value="Crosscut" from="I_move" to="I_setX()">
            <attr name="Type"><String>method-call</String></attr>
        </edge>
        <edge value="Crosscut" from="I_move" to="I_setY()">
            <attr name="Type"><String>method-call</String></attr>
        </edge>
        <node value="Advice" id="I_before">
            <attr name="Kind"><String>before</String></attr>
            <attr name="Pointcut"><String>move</String></attr>
        </node>
        <edge value="Advices" from="I_before" to="I_move" />
        <node value="Advice" id="I_after">
            <attr name="Kind"><String>after</String></attr>
            <attr name="Pointcut"><String>move</String></attr>
        </node>
        <edge value="Advices" from="I_after" to="I_move" />
        <edge value="Call" from="I_after" to="I_update()" />
    </graph>
</gxl>
```

ANNEXE D

L'Outil AGG

AGG est un environnement de programmation visuelle basée sur les règles, et s'appuyant sur la transformation de graphes. AGG vise la spécification et la mise en œuvre des applications constituées de données complexes et structurées en graphes. Il contient un moteur à usage général de la transformation de graphes implémenté en langage Java.

En AGG, les règles de production (ou transformation) sont stockés dans le cadre d'une grammaire en graphes attribués. AGG prend aussi en charge la spécification des graphes types avec multiplicités et attributs. En AGG, les attributs d'un arc ou un nœud agissent en tant que variables Java ordinaires auxquelles une valeur peut être assignée.

Par le biais d'expressions Java, les règles de transformation peuvent spécifier comment les valeurs d'attributs doivent être mises à jour lors d'une transformation. Les règles peuvent également contenir des NACs et des contraintes supplémentaires (conditions de contexte) qui doivent être satisfaites lorsqu'elles sont appliquées sur un graphe hôte. Ceci est très intéressant puisque un graphe type et des NACs ne sont pas toujours suffisamment expressifs.

Globalement, les caractéristiques d'AGG sont:

- Les structures de données complexes sont modélisées sous forme de *graphes* qui peuvent être typés par un *graphe type*.
- Le comportement du système est spécifié par un ensemble de *règles de transformation* sous la forme d'un style de description si-alors.
- En outre, AGG permet de rattacher des *conditions négatives d'application NACs* aux règles afin d'exprimer des contraintes de non-existence de sous-structures.
- Les graphes peuvent être attribués en *types et classes* disponibles dans les bibliothèques Java (String, int…etc.). En outre, de nouvelles classes Java peuvent être aussi incluses.
- Les règles de transformation peuvent aussi être *attribuées par des expressions Java* qui seront évalués lors de l'application de ces dernières. De plus, les

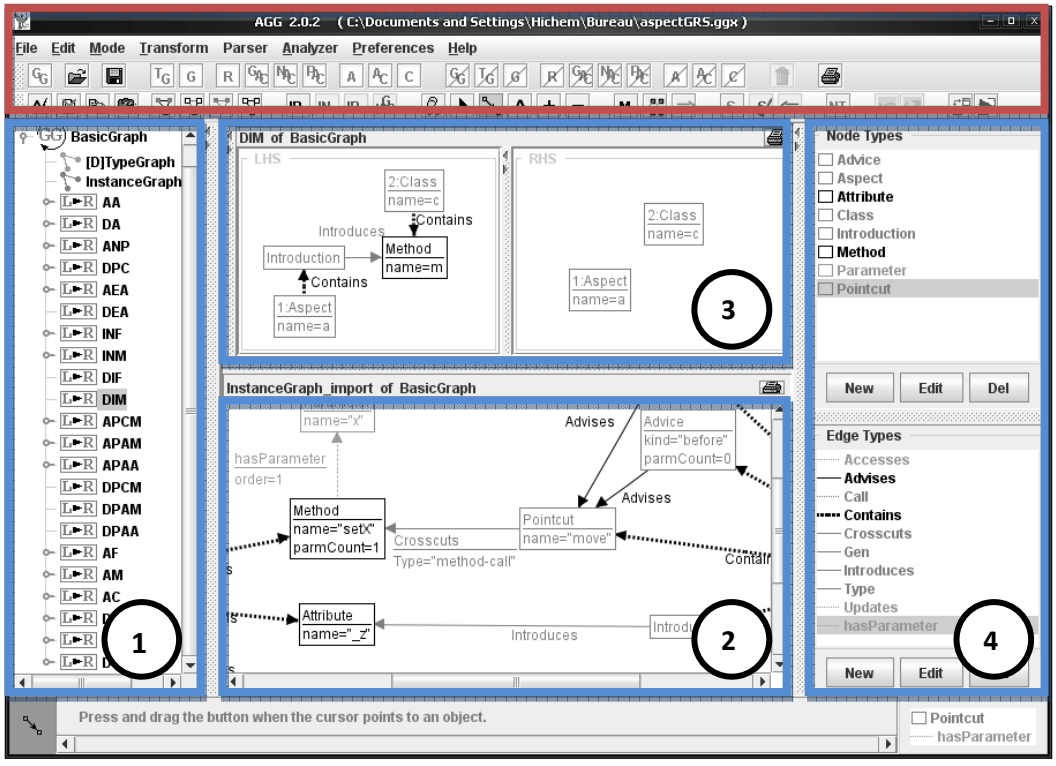
règles peuvent avoir des *conditions d'attributs* sous forme d'expressions Java booléennes.

— AGG propose un format de stockage de systèmes de transformation de graphes appelé *GGX*. Ce dernier, est constitué de graphes en *GXL* en plus d'un ensemble de règles de transformations.

— Il permet aussi d'*importer des graphes* externes en *GXL*.

Interface de l'Environnement AGG:

L'environnement AGG fournit une interprétation visuelle fluide en termes d'éditeurs graphiques pour les graphes et les règles, en plus d'un éditeur textuel intégré pour les expressions Java. La figure suivante montre une capture d'écran détaillée sur AGG.



L'interface d'AGG qui apparaît ci-dessus est principalement constituée de quatre principales vues (encadrées en bleu):

— Une première vue qui contient l'arborescence du système de transformation de graphes courant, où il est constitué d'un Graphe Type, un Graphe Hôte (ou Instance), et un ensemble de règles de transformations.

— Une deuxième vue sur l'un des graphes que contienne le système de transformation (ici apparait le graphe hôte).

— Une troisième vue affichant la règle de transformation sélectionnée. L'exemple courant ne montre qu'une règle en LHS et RHS, mais il est aussi possible d'afficher une règle qui contient une condition NAC.
— Une quatrième vue où apparaît l'ensemble de type d'arcs et de nœud que puisse gérer le système courant.

En plus des vues ci-dessus, la barre d'outils (encadré en rouge) que fournisse AGG, permet de manipuler facilement les différents composants du système de transformation de graphes (Importer un graphe, ajouter des nœuds, appliquer une règles …etc.).

ANNEXE E

L'Algorithme Apriori

L'algorithme Apriori est un algorithme d'exploration de données conçu en 1994, par Agrawal et Sikrant, dans le domaine de l'apprentissage des règles d'association. Il sert à reconnaitre des propriétés qui reviennent fréquemment dans un ensemble de données et d'en déduire une catégorisation. Apriori est conçu pour opérer sur les bases de données qui contiennent des transactions. Donné un ensemble d'items *"itemsets"*, l'algorithme essaie de trouver des sous-ensembles qui sont communs à au moins un nombre minimum C d'itemsets. Apriori utilise une approche "bottom up" où les sous-ensembles fréquents sont étendus un item à la fois, (une étape connu par la génération du candidat) et les groupes de candidats sont testés contre les données. L'algorithme termine quand aucune extension prospère supplémentaire n'est trouvée.

L_1 = {large 1 − itemsets};
for ($k = 2$; $L_{k-1} \neq \emptyset; k++$)do begin
  C_k = apriori − gen(L_{k-1}); // New candidates
  forall transactions $t \in D$ do begin
    C_t = subset(C_k , t); // Candidates contained in t
    forall candidates $c \in C_t$ do $c.count++$;
  end
  $L_k = \{c \in C_k \mid c.count \geq minsupg\}$
end
Answer = $\bigcup_k L_k$;

ANNEXE F

L'Algorithme Apriori en XQuery

Les trois fonctions (*join*, *commonIts*, *removeIts*) suivantes actent comme les opérateurs classiques des ensembles: l'union, l'intersection, et l'exception respectivement:

```
define function join(element $X, element $Y) returns element {
   let $items := (for $item in $Y where every $i in $X satisfies $i != $item return
   $item)
      return $X union $items
}
```

```
define function commonIts(element $X, element $Y) returns element {
   for $item in $X where some $i in $Y satisfies $i = $item return $item
}
```

```
define function removeIts(element $X, element $Y) returns element {
   for $item in $X where every $i in $Y satisfies $i != $item return $item
}
```

Les trois fonctions *candidateGen*, *prune*, et *removeDuplicate* sont utilisées pour générer l'ensemble de candidats. Les fonctions *candidateGen* et *prune* fonctionnent comme suit: Pour chaque passe *k*, la fonction *candidateGen* génère *k-itemsets* en joignant l'ensemble *large (k-1) itemsets* avec lui même. À l'intérieur de cette fonction, elle fait appel à la fonction *prune* pour supprimer l'*itemset* qui n'est pas un *large-itemset* potentiel. De la génération de candidats, il peut exister des *itemsets* dupliqués. Par conséquent, la fonction *removeDuplicate* a été définie pour éliminer la duplication.

```
define function candidateGen(element $l) returns element {
   for $freqSet1 in $l
   let $items1 := $freqSet1//items/*
      for $freqSet2 in $l
      let $items2 := $freqSet2//items/*
      where $freqSet2 >> $freqSet1 and
         count($items1)+1 =count($items1 union $items2)
         and prune(join($items1,$items2), $l)
```

```
        return <items> {join($items1,$items2)}</items>
}
```

```
define function prune(element $X, element $Y) returns boolean{
   every $item in $X satisfies
   some $items in $Y//items satisfies
   count(commonIts(removeIts($X,$item),$items/*))
   = count($X) - 1
}
```

```
define function removeDuplicate(element $C) returns element{
   for $itemset1 in $C
   let $items1 := $itemset1/*
   let $items :=(for $itemset2 in $C
      let $items2 := $itemset2/*
      where $itemset2>>$itemset1 and
      count($items1) =
      count(commonIts($items1, $items2))
      return $items2)
   where count($items) = 0
   return $itemset1
}
```

Une fois que les *itemsets* de candidats sont disponibles, on doit sélectionner le *large-itemsets* de l'ensemble de candidats. La fonction *getLargeItemsets* prend quatre arguments, l'ensemble de candidats *C*, le support minimum *minsup*, le nombre total de transactions *total* et la source de données *src*. Pour chaque itemset *c* dans *C*, la fonction parcoure à travers le document des transactions *src* pour obtenir la valeur du support pour *c* et rendre l'itemsets avec la valeur du support plus grand que le *minsup*.

```
define function getLargeItemsets(element $C, element $minsup, element $total,
element $src) returns element {
    for $items in $C
    let $trans := (for $tran in $src where every $item1 in $items/* satisfies
        some $item2 in $tran/* satisfies $item1 = $item2 return $tran)
    let $sup := (count($trans) * 1.00) div $total
    where $sup >= $minsup
    return <largeItemset> {$items}<support> {$sup} </support> </largeItemset>
}
```

```
define function apriori(element $l, element $L, element $minsup, element $total, element $src) returns element {
    let $C := removeDuplicate(candidateGen($l))
    let $l := getLargeItemsets($C, $minsup, $total, $src)
    let $L := $l union $L
    return if (empty($l)) then $L else apriori($l, $L, $minsup, $total, $src)
}
```

Donc l'expression XQuery qui manipule les différentes fonctions précédentes va être comme suite:

```
let $src := document("/transactions.xml")//items
let $minsup := 0.4
let $total := count($src) * 1.00
let $C := distinct-values($src/*)
let $l :=(for $itemset in $C
    let $items := (for $item in $src/*
        where $itemset = $item return $item)
    let $sup := (count($items) * 1.00) div $total
        where $sup >= $minsup
            return <largeItemset>
                        <items> {$itemset} </items>
                        <support> {$sup} </support>
                    </largeItemset>)
    let $L := $l
        return <largeItemsets> { apriori($l, $L,$minsup, $total, $src) }</largeItemsets>
```

A Propos de l'Auteur

Biographie

Hanene Cherait a obtenu son baccalauréat en science de la nature et de la vie en 2004. Elle rejoignit l'université de Badji Mokhtar – Annaba (UBMA) la même année, pour suivre une formation en informatique et obtenir une licence en 2007. Elle obtient son Master option Ingénierie des Logiciels Complexes (ILC) en 2009. Depuis cette date, elle a commencé de préparer sa thèse sous la direction de Dr Nora. Bounour, afin d'obtenir le diplôme de Doctorat 3ème cycle.

Actuellement, elle est membre du Laboratoire d'Ingénierie des Systèmes COmplexes LISCO, qui s'active autour les thèmes de l'évolution et de la réutilisation du logiciel. Ses intérêts de recherche comprennent: la séparation avancé des préoccupations, le paradigme orienté aspect, la maintenance et l'évolution du logiciel, la rétro-ingénierie du logiciel, et la transformation de graphes.

Courriel: hanene_cherait@yahoo.fr

Adresse: Laboratoire LISCO, Département d'informatique, Université Badji Mokhtar – Annaba, P.O. Box 12, 23000 Annaba, Algeria.

Publications Internationales

- H. Cherait, and N. Bounour, "Rewriting Rule-based Model for Aspect Oriented Software Evolution", *International Journal of Computer Applications in Technology (IJCAT), Special Issue on Current Trends and Improvements in Software Engineering Practices*, (in press), to appear (2015). www.inderscience.com/ijcat

- H. Cherait, and N. Bounour, "Detecting Change Patterns in Aspect Oriented Software Evolution: Rule-based Repository Analysis", *International Journal of Software Engineering and Its Applications (IJSEIA)*, January 2014, Vol 8(1), pp. 247-266. www.sersc.org/journals/IJSEIA/vol8_no1_2014/22.pdf

Communications Internationales

- H. Cherait, and N. Bounour, "Modeling Software Evolution through Version Control System", *In Proceedings of Colloque Africain sur la Recherche en Informatique et Mathématiques Appliquées (CARI'12),* Alger, Algérie, 13-16 Octobre 2012. www.cari-info.org/cari-2012/session%203/3C2.pdf
- H. Cherait, and N. Bounour, "Toward a Version Control System for Aspect Oriented Software", *In Proceedings of 1st International Conference on Model & Data Engineering*, Portugal, September 28-30, 2011, Springer-Verlag, LNCS 6918, pp. 110–121, DOI: 10.1007/978-3-642-24443-8_13. www.medi2011.ensma.fr *or* http://link.springer.com/chapter/10.1007%2F978-3-642-24443-8_13
- H. Cherait, and N. Bounour, "Vers un Modèle d'Evolution des Programmes Orientés Aspect", *In Proceedings of 1st International Conference on Information Systems and Technologies,* Tébessa, Algérie, 24-26 Avril 2011, pp. 35-45. http://icist2011.eu5.org/index.php
- H. Cherait, and N. 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, pp. 247-254. http://majecstic2010.labri.fr/
- H. Cherait, and N. Bounour, "Software Evolution: Models and Challenges", *In Proceedings of International Conference on Machine and Web Intelligence (ICMWI'10),* Algiers, Algeria, October 3-5, 2010, pp. 458-460, IEEE Computer Society, DOI: 10.1109/ICMWI.2010.5647967. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5647967

Communications Nationales

- H. Cherait, and N. Bounour, "Une classification comparative des modèles de l'évolution de logiciel", *In Proceedings of Journées d'Etudes Doctorales (JED'10),* Annaba, Algérie, 14-15 Juin, 2010, pp. 132-137. www.labged.net/index.php?rubrique=mapage102