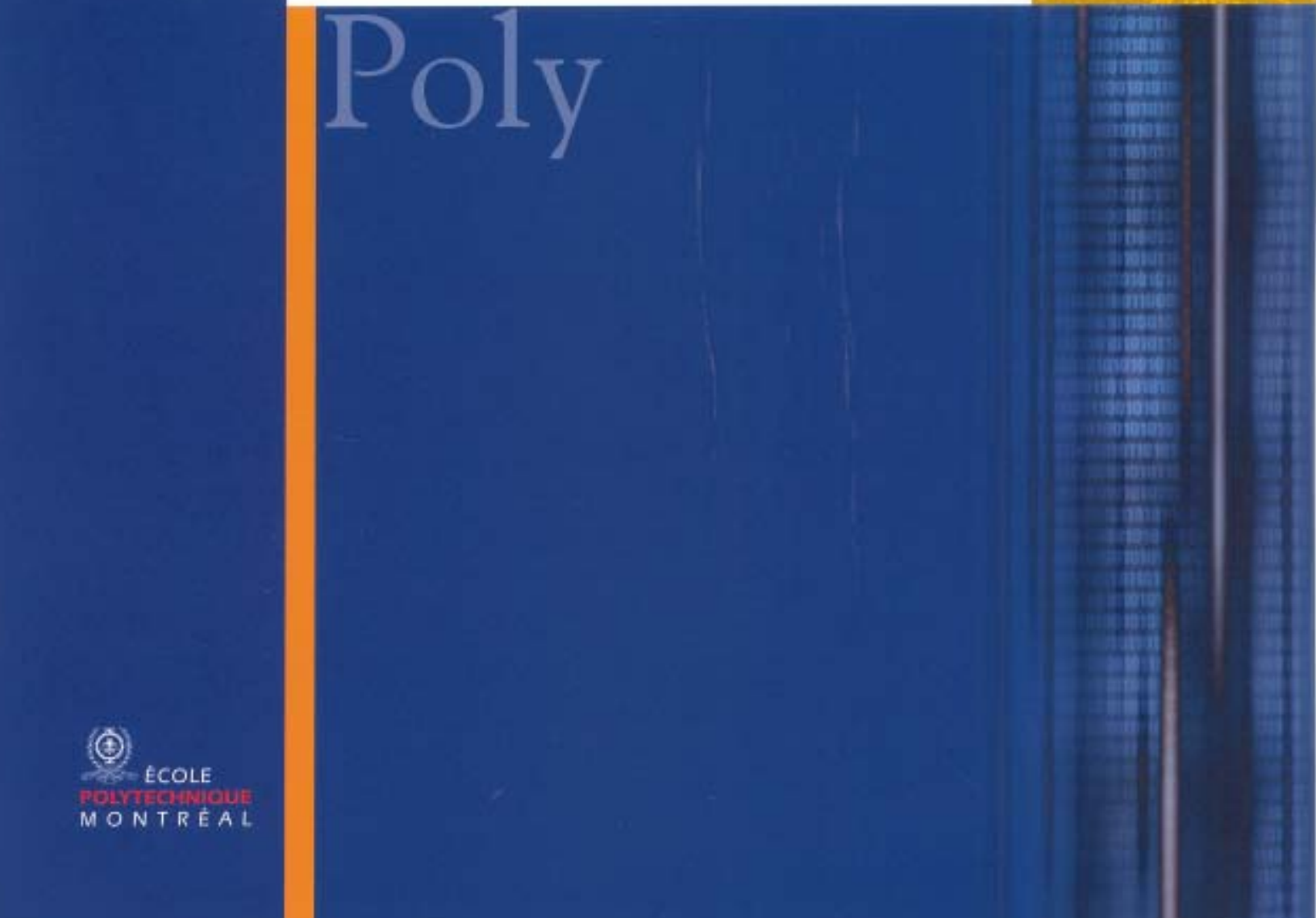
Titre: Title:	Évolution biologique et applications en génie logiciel
Auteurs: Authors:	Salima Hassaine
Date:	2010
Type:	Rapport / Report
Référence: Citation:	Hassaine, Salima (2010). Évolution biologique et applications en génie logiciel. Rapport technique. EPM–RT–2010-03.

EPM–RT–2010-03

ÉVOLUTION BIOLOGIQUE ET APPLICATIONS EN GÉNIE LOGICIEL

Salima Hassaine
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Août 2010



EPM-RT-2010-03

Évolution biologique et applications en génie logiciel

Salima Hassaine
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Août 2010

Dépôt légal :
Bibliothèque nationale du Québec, 2009
Bibliothèque nationale du Canada, 2009

EPM-RT-2010-03

Évolution biologique et applications en génie logiciel
par : Salima Hassaine
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Université de Montréal

Évolution biologique et applications en génie logiciel

par
Salima Hassaine

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Partie orale
de l'examen pré-doctoral

Décembre, 2009

Université de Montréal
Faculté des études supérieures

Évolution biologique et applications en génie logiciel

présenté par:

Salima Hassaine

a été évalué par un jury composé des personnes suivantes:

Nadia El Mabrouk
président-rapporteur

Yann-Gaël Guéhéneuc
directeur de recherche

Sylvie Hamel
codirecteur

Stefan Monnier
membre du jury

RÉSUMÉ

La maintenance des programmes orientés objets est une activité très coûteuse et la compréhension des programmes par les mainteneurs est essentielle pour la réaliser. L'analyse de l'évolution de l'architecture d'un programme aide à comprendre les problèmes rencontrés lors de sa conception ainsi que les solutions apportées. Les techniques existantes utilisées pour l'analyse de l'évolution de logiciels présentent toutes, cependant, un problème de performance.

Le but de notre travail de recherche est d'étudier l'évolution de logiciels d'un point de vue biologique. Nous proposons un mapping entre l'évolution biologique et l'évolution logicielle. Ce mapping propose une solution aux problèmes de performance, en adaptant des algorithmes efficaces de comparaisons et d'alignements de chaînes de caractères de bio-informatique, pour bénéficier de leurs performances sur des programmes de grande taille. Parmi nos résultats actuels, nous rapportons l'évidence de l'utilité de notre mapping en identifiant les classes en évolution qui maintiennent une structure stable des relations (utilisation, association, agrégation, création, héritage) avec d'autres classes et constituent ainsi probablement les épines dorsales des programmes. Nous rapportons aussi l'application de notre approche sur plusieurs programmes différents et comparons son exécution et résultats avec l'approche la plus récente de la littérature.

Mots clés : Évolution de logiciels ; changement de l'architecture ; stabilité de l'architecture ; évolution biologique ; homologie ; algorithmes de bio-informatique.

ABSTRACT

The maintenance of large programs is a costly activity because their evolution often leads to two problems: an increase in their complexity and an erosion of their design. These problems impede the comprehension and the future evolution of the programs. Most previous approaches of software evolution analysis are limited because of their performance.

The goal of our research work is to study the evolution of programs from a biological point of view by presenting a mapping between biological evolution and software evolution. This mapping offers a solution to the efficiency problem by using adaptation of string matching algorithms from bio-informatics to benefit from their performance on large programs. We report evidence of the usefulness of this mapping by identifying evolving classes that maintain a stable structure of relations (use, association, aggregation, inheritance, creation) with other classes and thus likely constitute the backbones of the programs. We apply our approach on several different size programs and compare its performance and results with the most recent approach from the literature.

Keywords: Software evolution; Design change; Design stability; Biological evolution; Homology; Bioinformatics Algorithms

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE 1

INTRODUCTION

Nous décrivons brièvement, dans ce chapitre, les problèmes qui ont motivé cette recherche, le but de notre travail ainsi que les solutions que nous proposons (réalisées et futures). Nous terminons en donnant un aperçu des différents chapitres du rapport.

1.1 Contexte : maintenance et évolution des programmes

Les systèmes industriels actuels nécessitent le développement de programmes de plus en plus complexes et exigent de leur part qu'ils soient de très bonne qualité du point de vue de la *maintenabilité* et de la *réutilisabilité*. Dans l'industrie, le coût de maintenance d'un programme est estimé entre 50% et 75% de son coût total [35]. On estime également que plus de la moitié de cette maintenance est consacrée à la compréhension du programme lui-même [10]. La maîtrise de la compréhension des programmes s'avère donc indispensable.

Un programme doit constamment évoluer pour rester utile. Ceci est principalement dû aux changements des besoins de ses utilisateurs et l'accroissement de leurs exigences. Faire évoluer un programme est un réel défi. L'expérience pratique avec de grands projets a montré que les développeurs font toujours face à des difficultés dans la maintenance de leur programmes [31]. De plus, depuis l'avènement des programmes à code source libre (*open-source*), le développement peut être réalisé par des milliers de développeurs répartis sur toute la planète, ce qui aggrave les difficultés de maintenance. La communauté du génie logiciel, qui s'intéresse à la compréhension de programmes, fait face à des défis de taille : Comment aider les développeurs à saisir toute la complexité de ces programmes ? Comment les aider à découvrir quelles composantes du programme sont concernées par leurs

changements ? Comment les aider à réaliser leurs changements sans introduire de bogues ou aller à l'encontre des choix de conception pris avant leur contribution ? De nouvelles méthodes et de nouveaux outils d'évaluation et de compréhension de l'évolution des programmes sont nécessaires pour aider les développeurs dans leurs activités de maintenance.

Si on se tourne maintenant vers un tout autre domaine, l'étude de l'évolution des êtres vivants est identifiée, par les biologistes, comme la clé de départ de la compréhension du phénomène vivant, de son origine et de son avenir. Le fait de disposer du génome complet d'un grand nombre d'organismes ouvre des perspectives nouvelles concernant l'analyse de l'évolution biologique. Même si les biologistes sont loin de connaître la fonction de tous les gènes, la génomique comparative permet déjà de poser des questions sous un angle nouveau. Les biologistes peuvent étudier non seulement la présence mais aussi l'absence de gènes pour détecter des groupes de gènes co-occurrents et analyser l'obsolescence des gènes ou leur "érosion" progressive dans certaines espèces où leur fonction n'est plus requise.

Les bio-informaticiens sont donc confrontés à des problèmes similaires à la recherche de patrons récurrents dans l'évolution des programmes. En effet, localiser des gènes mutés ou des protéines modifiées, dans de longues séquences d'ADN (de plusieurs millions de caractères), sont des problèmes essentiels en génomique comparative. L'élaboration d'algorithmes efficaces pour l'alignement local ou global de séquences d'ADN est donc une problématique naturelle. De nombreux chercheurs ont donc travaillé à l'amélioration d'algorithmes de recherche de patrons (motifs) classiques [30, 34] dans le but de les accélérer, que ce soit par l'utilisation d'algorithmes parallèles ou d'outils combinatoires [5, 7, 29]. Plusieurs des algorithmes développés sont très efficaces en temps et pourraient possiblement être adaptés pour résoudre le problème d'identification de motifs récurrents de conception dans des programmes de grandes tailles.

1.2 Problématique : étude de l'histoire de grands programmes

La plupart des outils développés pour étudier l'évolution des programmes sont basés sur des comparateurs, comme par exemple *diff* [16] qui permet de comparer le contenu de deux fichiers et de montrer les différences entre ces fichiers en termes de lignes ajoutées, supprimées ou modifiées. Bien que fort utiles dans certains cas, l'utilisation de ces outils n'est pas adaptée aux programmes de grandes tailles pour les raisons suivantes :

1. Ils sont basés sur une représentation textuelle du code source du programme ; Ces outils opèrent donc à un niveau d'abstraction faible. Ils indiquent les numéros de lignes modifiées sans pour autant donner d'indication sur les entités logicielles ayant été effectivement modifiées.
2. Toutes les modifications n'ont pas la même importance. Changer une ligne dans un commentaire n'a pas le même impact que changer la signature d'une méthode. Pourtant, dans les deux cas, un outil comme *diff* rapportera une différence au niveau textuel.
3. Au niveau du code source, le nombre de modifications peut être très important. Des milliers de lignes peuvent être changées, ajoutées et supprimées chaque jour. Les résultats rapportés par ces outils sont bien trop détaillés pour être exploitables. En réalité, de tels outils sont utiles pour analyser l'évolution d'un fichier à la fois.

Ainsi, utiliser un outil comme *diff* n'est pas d'une grande utilité pour la compréhension de grands programmes, car les modifications sont en général trop nombreuses et sont, de toutes façon, exprimées à un niveau d'abstraction beaucoup trop bas.

De nombreux travaux [1, 3, 17, 20, 21, 28, 37, 39–41, 43] s'intéressent à l'extraction de motifs récurrents dans l'histoire d'un programme pour offrir aux développeurs une vision précise et simplifiée du programme dans le temps. Ces motifs ont pour

but de faciliter la compréhension et de diminuer les erreurs lors de l'évolution du programme. Par exemple, dans ses travaux précurseurs, Zimmermann [42] a défini le patron de changement par l'ensemble d'éléments (classes et/ou méthodes) d'un programme qui ont changé ensemble au cours de leur évolution. Ce type de patron permet aux développeurs de détecter les changements incohérents appliqués à une version. Dans un autre travail, Yann-Gaël *et al.* [9] ont généralisé le travail de Zimmermann en introduisant le concept de patrons de changements—un ensemble de modifications du programme qui occurrent régulièrement pendant l'évolution du programme—et en utilisant la technique de Dynamic TimeWarping [22] pour améliorer grandement la précision et le rappel de la détection des motifs d'évolution. Plus récemment, Kopjedo *et al.* [21] ont proposé d'identifier l'ensemble des classes qui ne changent pas dans l'histoire d'un programme, en utilisant un algorithme d'appariement de graphes approchés, basé sur des métaheuristiques (ETGM : Error Tolerant Graph Matching).

Cependant, toutes les approches existantes sont limitées dans leurs performances en temps et en mémoire et dans leurs résultats en précision et en rappel, lorsqu'appliquées sur de grands programmes. Ainsi, les deux principales limites à l'étude de l'histoire de grands programmes sont :

- le manque de patrons (d'évolution et de changements) connus décrivant des situations récurrentes de changement dans l'évolution d'un programme, et facilitant la compréhension de cette évolution ;
- le trop grand volume de données à analyser pour comprendre l'évolution de programmes de grande taille. Par exemple, le navigateur Web Mozilla a plus de 68 versions (à ce jour), dont la première version ("Milestone 3" du 19/03/99) contient 1 398 991 lignes de codes alors que l'une des dernières (v1.7.13) en inclue 3 612 838 (en ne comptant que les lignes de code source C/C++).

1.3 Objectifs

L'objectif de ce travail est d'étudier l'historique de grands programme d'un point de vue biologique. Ce projet fait suite aux travaux de Kaczor *et al* [18], qui transforment, de façon automatique, un programme orienté-objet et le patron recherché en chaînes de caractères. Cela permet d'appliquer l'algorithme vectoriel [7], utilisé pour la recherche de séquences biologiques dans les génomes, à la recherche de patrons de conception dans de grands programmes. Nous proposons d'adapter leurs idées à l'analyse de l'évolution des programmes. Dans un premier temps, nous allons extraire des motifs récurrents dans l'histoire d'un programme pour répondre, par exemple, aux questions suivantes :

1. Quels éléments sont stables (changent rarement) ?
2. Quels éléments sont instables (changent fréquemment) ?
3. Combien d'éléments ont été ajoutés, modifiés, effacés ?
4. Pourquoi le changement a-t-il été fait ?
5. Quels éléments changent fréquemment ensemble ?

Ensuite, nous allons étudier l'évolution de ces motifs pour faire apparaître des patrons de changements caractéristiques d'une bonne architecture, nécessaires au développement de bonnes pratiques pour réduire les coûts de développement et de maintenance.

1.4 Contributions

Pour remédier au problème de performance, nous tentons d'adapter les algorithmes efficaces développés dans la communauté bio-informatique pour la recherche de motifs récurents dans l'histoire de grands programmes. Nous proposons les contributions suivantes :

- **Étape 1 : représentation du génome d'un programme**. Définir une représentation de l'architecture d'un programme sous forme d'une séquence.
- **Étape 2 : enrichissement de la représentation du génome**. Enrichir la représentation du génome, par des informations structuelles plus détaillées.
- **Étape 3 : définition du mapping**. Définir un mapping entre les concepts de l'évolution biologique et l'évolution logicielle.
- **Étape 4 : alignement des versions d'un programme**. Adapter les algorithmes bio-informatiques à l'alignement de plusieurs versions d'un programme.
- **Étape 5 : définition des patrons de changements**. Analyser les différences (ou similarités) entre deux versions d'un même programme, pour identifier des patrons de changements.
- **Étape 6 : comparaison systématique et exhaustive des outils**. Construire un ensemble de critères objectifs et d'un corpus d'évaluation des outils de détection des patrons de changement.

1.5 Organisation du rapport

Le Chapitre 2, introduit les différents travaux préalables dans les domaines reliés à notre recherche. L'évolution biologique, les algorithmes d'alignements de séquences d'ADN, et enfin, les travaux précédents concernant l'évolution de programmes. Le Chapitre 3, présente les étapes de notre méthodologie. Chapitre 4 présente nos premiers résultats sur la détection des micro-architectures stables. Finalement, le Chapitre 5 présente à la fois les perspectives de nos travaux futurs et notre échéancier.

CHAPITRE 2

ÉTAT DE L'ART

Nous nous intéressons dans ce chapitre aux différents travaux préalables dans les domaines reliés à notre recherche. Ce chapitre est articulé en deux parties. La première partie est dédiée à l'évolution logicielle, tandis que la seconde effleure les efforts de la communauté bio-informatique pour étudier l'évolution des séquences d'ADN.

2.1 Évolution logicielle

2.1.1 Maintenance

Plusieurs recherches ont été conduites pour vérifier empiriquement l'importance de la maintenance. Selon [19, 26, 33], la maintenance est la phase la plus coûteuse du développement logiciel. Les chiffres montrent que plus de 50% des coûts du développement total d'un système est lié à la maintenance. Des études empiriques [19] ont vérifié que plus de 80% de l'effort global de maintenance est lié à faire évoluer le programme avec les besoins de ses utilisateurs. Swanson [36] a défini une taxonomie sur les catégories associées à la phase de maintenance :

- **la maintenance corrective** est ce qui est généralement appelée la maintenance classique, qui consiste à corriger des erreurs ;
- **la maintenance perfective** est la maintenance qui cherche à ajouter et à modifier des fonctionnalités. Ceci est généralement fait pour garder le système pertinent pour ses utilisateurs ;
- **la maintenance adaptative** est l'adaptation du programme pour un nouvel environnement d'opérations. On parle souvent de « port » sur une nouvelle plateforme matériel ou à un nouveau système d'exploitation. Selon la

taxonomie originale, les différentes sortes de maintenances se doivent d'être exclusives ;

À ces catégories, on ajoute également :

– **La maintenance préventive** qui est l'effort pour prévenir les problèmes futurs.

Selon [6], l'évolution logicielle comprend la maintenance perfective et adaptative, mais pas la maintenance corrective.

2.1.2 Évolution logicielle

Le terme "évolution logicielle" a été utilisé pour la première fois par Lehman [25] lorsqu'il a mené une étude empirique sur l'évolution des programmes, en commençant par analyser les changements au sein du système d'exploitation pour les gros systèmes OS/390 d'IBM. Démarrée en 1969 et encore poursuivie de nos jours, cette étude fait émerger huit lois [24], applicables à l'évolution des programmes. Les six premières lois sont :

– **changement continuel** : un programme utilisé dans un environnement du monde réel doit nécessairement changer sinon il devient progressivement de moins en moins utile dans cet environnement ;
– **complexité grandissante** : lorsqu'un programme change, sa structure tend à devenir plus complexe. Des ressources additionnelles doivent être consacrées à maintenir et à préserver sa structure ;
– **évolution des grands programmes** : l'évolution des grands programmes est un processus auto-régulateur. Les attributs comme la taille, le temps entre versions et le nombre d'erreurs signalées sont approximativement invariants pour chaque version du programme ;
– **conservation de l'organisation** : l'effort dédié à faire évoluer un programme demeure constant au cours du temps ;

- **conservation de la familiarité** : tous les gens concernés par l'évolution doivent à tout moment comprendre le programme sinon l'évolution sera mauvaise ;
- **croissance continue** : le nombre de fonctionnalités d'un programme augmente avec le temps, afin de satisfaire ses utilisateurs.
- **qualité en baisse** : la qualité d'un programme sera dégradée avec le temps, à moins que sa conception ne soit soigneusement maintenue et adaptée aux nouvelles contraintes opérationnelles.

Une des conclusions majeures des lois de Lehman est qu'un programme est fortement dépendant de son environnement. Ses évolutions sont dictées par celui-ci selon le principe du *feed-back* : tout changement dans l'environnement extérieur envoie un signal au sein du programme, dont les évolutions constituent la réponse renvoyée à l'extérieur. Cette réponse peut générer elle-même des changements, engendrant ainsi une boucle de rétro-action.

2.1.3 Travaux antérieurs

De nombreux travaux existent pour aider les développeurs à comprendre l'évolution de leur programme. On distingue les travaux qui s'appuient sur les changements statistiques ou encore les métriques d'évolution [12,13,26,27] et ceux qui s'appuient sur les algorithmes d'appariement de graphes [1,3,17,20,21,28,37,39–41,43].

Par exemple, Antoniol *et al.* [3] ont proposé une approche automatique basée sur les techniques de recherche d'information, pour tracer, identifier et documenter les discontinuités de l'évolution des classes. Cette approche a été utilisée pour identifier les restructurations possibles du code. Cependent, ce travail est limité aux restructurations des classes seulement. Zimmermann [42] a proposé d'identifier les co-changements dans l'histoire d'un programme, il a utilisé des techniques de fouille de données, à base des règles d'associations, pour détecter des relations entre

les changements, pour découvrir quels sont les fichiers, les classes, les méthodes et les champs qui changent fréquemment ensemble. Ce qui permet de prévoir les changements futurs suite au changement d'un élément, pour prévenir les changements incomplèts. Dans un autre travail, Yann-Gaël *et al.* [9] ont généralisé le travail de Zimmermann en introduisant le concept de patrons de changements (voir figure 2.1) et en utilisant la technique de Dynamic Time Warping [22] pour améliorer grandement la précision et le rappel. La figure 2.1 montre l'histoire de 6 éléments d'un

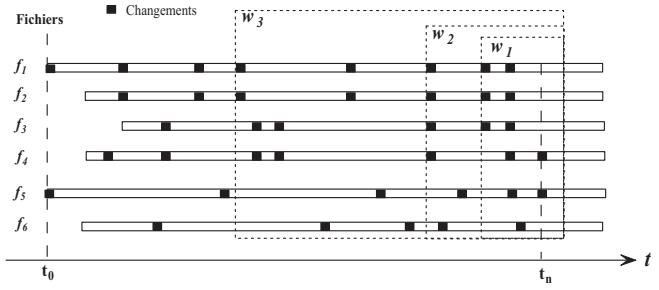


FIG. 2.1 – Changements d'éléments f dans le temps (source [9])

programme f_1–f_6 avec leurs changements dans le temps et 3 fenêtres temporelles de différentes tailles. Suivant la fenêtre choisie, différents éléments seront identifiés comme changeant ensemble : avec w_1, $\{f_1, f_2, f_3\}$ semblent changer ensemble régulièrement, avec w_2, ce sont toujours $\{f_1, f_2, f_3\}$ alors qu'avec w_3 ce sont seulement $\{f_1, f_2\}$. Ainsi, suivant le motif recherché et sa "longueur" dans le temps, plusieurs réponses sont possibles. Ces travaux soulignent la nécessité de définir des motifs récurrents plus précis, complexes et représentatifs de situations intéressantes et le besoin en algorithmes plus performants pour leur recherche. Un travail récent [38] a posé 10 questions :

1. Qui a travaillé sur ces éléments ?
2. Quel(s) développeur(s) travaillent sur les mêmes éléments ?
3. Quelle(s) sorte(s) de changement fait un développeur ?
4. Quand le changement a-t-il été fait ?

5. Pourquoi le changement a-t-il été fait ?
6. Quels éléments changent fréquemment ensemble ?
7. Quels éléments sont stables (changent rarement) ?
8. Quels éléments sont instables (changent fréquemment) ?
9. Combien de développeurs ont travaillé sur une version du programme ?
10. Combien d'éléments ont été ajoutés, modifiés, effacés ?

Les réponses à ces questions sont jugées essentielles lorsque des développeurs doivent faire évoluer un programme en modifiant ses éléments (classes, méthodes...).

Kim *et al.* [20] ont présenté une approche automatisée *LSDiff* (*Logical Structural Diff*) pour détecter les changements structurels de haut-niveau, par exemple, les changements dans les en-têtes de méthodes. Leur outil *LSDiff* est basé sur la programmation logique et construit un ensemble de règles de changements (par des prédicats) à partir des différents changements structurels, mais sans prendre en compte les changements de relations entre les classes. Dagenais et Robillard [11] ont proposé d'analyser les changements dans un *cadriciel* et de recommander des changements similaires à ses clients. *SemDiff* et *LSDiff* sont semblables, ils permettent d'identifier les ajouts et les suppressions de méthodes et des appels de méthodes. *SemDiff* effectue un appariement des graphes d'appel de fonctions, pour trouver des changements dans les appels d'une *API* supprimée, au cours de l'évolution du *cadriciel*. Godfrey *et al.* [37] propose une approche destinée à analyser l'évolution structurelle de programmes. L'outil *BEAGLE* proposé travaille au niveau fichier pour déterminer les fusions ou spécialisations de classes. Cet outil comporte un algorithme basé sur "origin analysis", pour déterminer l'origine des changements structurels. La figure 2.2 illustre un exemple d'analyse de dépendances. La méthode A est nouvellement ajoutée à la version $v_{2.0}$, et on veut retrouver son origine dans la version précédente. Pour ce faire, l'outil *BEAGLE* analyse les méthodes appellées par A et les méthodes appelantes de A dans les deux versions, ce qui permet de

déduire que la méthode A est l'une des anciennes méthodes de $v_{1.0}$ qui a été renommée.

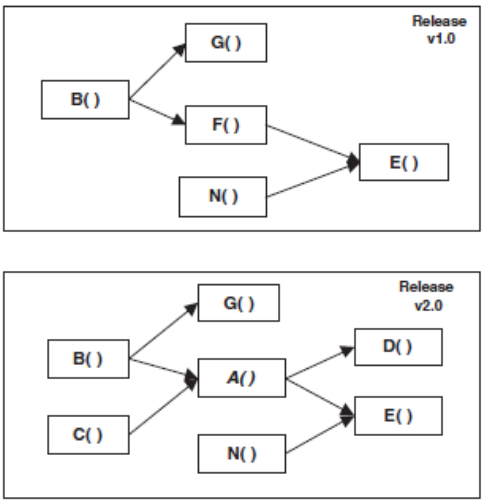


FIG. 2.2 – Un exemple de changement des appels de méthodes (source [37])

Xing *et al.* [41] proposent un algorithme nommé *UML-Diff* pour apparier différentes versions d'un programme en utilisant plusieurs métriques de classes. À partir de deux diagrammes de classes obtenus à partir du code source par rétro-ingénierie, *UML-Diff* produit un arbre de changements structurels qui rapporte les différences depuis le niveau d'abstraction "package" jusqu'au niveau des attributs ou méthodes. *UML-Diff* construit une solution sur la base de classes conservant le même nom d'une version à une autre. Cet algorithme est donc, comme le reconnaissent ses auteurs, très vulnérable à un "renommage" important entre deux versions. Son adaptation à d'autres types d'artefacts logiciels apparaît assez peu évidente, tellement il est spécifique aux diagrammes de classes. *UML-Diff* a été également employé pour étudier l'évolution de l'architecture [40] pour comprendre des phases et des types d'évolution [39].

Dans un autre travail [4], chaque version du programme est considérée comme un ensemble de classes. Des mesures de similarité sont calculées entre les classes de deux versions en tenant compte de la distance d'édition entre les noms de classes,

d'attributs et de méthodes. Des seuils sont appliqués pour déterminer la présence d'arcs valués entre noeuds des deux versions. Un algorithme d'appariement maximum est ensuite appliqué au graphe biparti résultant. Cette approche propose la modélisation la plus rapprochée d'un problème AGM (Approximate Graph Matching). Contrairement aux autres travaux, un problème d'optimisation y est défini. Toutefois, les relations entre classes ne sont pas modélisées. Lorsqu'on retrouve par rétro-ingénierie les diagrammes de classes de deux versions successives, l'application d'un algorithme AGM paraît pertinente pour retrouver les changements intervenus entre deux versions.

Kopjedo *et al.* [21] ont proposé l'approche Error Correcting Graph Matching (ECGM) qui utilise les graphes pour analyser les diagrammes de classes de versions successives d'un même programme. Partant du diagramme de classes d'une version donnée, ils procèdent par appariements successifs avec des diagrammes des versions successives. Ils définissent ainsi un coût d'évolution (EC) qui quantifie la somme de changements survenus dans une classe d'une version de départ choisie jusqu'à la version courante du projet programme. Ils retrouvent aussi le tunnel du programme, *i.e.*,l'ensemble des classes appariées pendent toute la durée de vie du programme. Le tunnel identifie les classes les plus stables d'un programme et donc les plus susceptibles d'en représenter l'épine dorsale. Cependent, l'algorithme ECGM proposé est basé sur des méta-heuristiques, et donc l'ensemble des classes appariées peut ne pas être optimal.

2.1.4 Discussion

Les différentes techniques utilisées jusqu'à maintenant pour l'analyse du niveau conceptuel d'un programme ont une efficacité limitée. La plupart obtiennent des résultats satisfaisants sur des programmes de petites tailles mais ne permettent pas l'analyse de gros programmes. L'identification de micro-architectures dans un artefact logiciel tel qu'un diagramme de classes est un champ de recherche reconnu

du génie logiciel et souvent une étape nécessaire dans l'identification de patrons de changements dans l'historique des programmes. De manière générale, nous cherchons à identifier les micro-architectures stables/instables, pour trouver des relations entre les choix de conception pris au départ et les changements, et donc pour mieux comprendre l'évolution de programmes.

Les bio-informaticiens sont confrontés à des problèmes similaires à la recherche de micro-architectures dans des programmes de grandes tailles. En effet, la comparaison et l'alignement de séquences d'ADN (Acide DésoxyriboNucléique) sont essentiels en biologie moléculaire. Le concept de changement est aussi important en bio-informatique car la duplication avec modifications est un processus central dans l'évolution de protéines. La mutation de gènes est aussi très fréquente en biologie [14]. Localiser des gènes mutés dans de longues séquences d'ADN ou des protéines modifiées dans de longues séquences d'acides aminés sont donc des problèmes similaires à celui de détection de formes approchées de micro architectures dans un grand programme. À part quelques travaux sur la détection des patrons de conception [18], peu de travail a été effectué pour adapter les algorithmes de bio-informatique à l'analyse de l'évolution de programmes. La section suivante introduit différentes techniques de recherche exacte et approximative en bio-informatique.

2.2 Évolution biologique

En biologie, la génomique est une discipline consistant à faire l'inventaire de l'ensemble des gènes d'un organisme afin d'en déterminer leurs fonctions. Étudier les phénomènes évolutifs constitue un des enjeux majeurs de la génomique comparative. Cette dernière consiste à établir la correspondance entre les gènes de différentes espèces. Elle tente en effet, d'interpréter la multitude de données disponibles afin de proposer des modèles d'évolution, de retracer les processus responsables de la

conservation/divergence entre les génomes.

Par définition, l'ADN est une molécule que l'on retrouve dans toutes les cellules vivantes. Elle constitue le génome des êtres vivants et se transmet en totalité ou en partie lors des processus de reproduction. L'ADN est constitué de séquences de quatres nucléotides A, C, T, G. L'unité utilisée en génomique comparative est le *gène*, une séquence d'ADN qui code la fabrication des protéines. Au niveau le plus bas, les mutations, qui affectent directement les nucléotides du gène, sont étudiées (mutations ponctuelles). Finalement, à un niveau plus élevé, des parties de génomes et même des génomes entiers (ensembles de gènes) peuvent être dupliqués.

Selon la théorie de l'évolution, toutes les espèces vivantes sont issues d'un même ancêtre, des mutations interviennent au niveau de l'ADN, générant ainsi de nouvelles espèces. Ces mutations se produisent localement (suppressions, insertions ou substitutions d'un ou plusieurs nucléotides). Au sens de cette théorie, deux séquences peuvent donc être plus ou moins proches, selon le nombre de modifications ayant eu lieu. Lorsque leurs similarités sont suffisamment fortes, ou suffisamment significatives, on peut alors supposer leur homologie, complète ou de certaines régions de leur séquence.

L'alignement de séquences a pour but d'identifier des zones conservées entre séquences (voir figure 2.3). On peut distinguer trois principales catégories d'algorithmes d'alignements :

- Alignement local : utilisé pour trouver des zones des séquences qui sont significativement semblables (*e.g.*, algorithme de Smith-Waterman [34]).
- Alignement global : réalisé sur toute la longueur des séquences pour déterminer si elles sont homologues (*e.g.*, algorithme de Needleman-Wunsch [30]).
- Recherche de motifs : utilisé pour localiser les occurrences d'un motif dans une séquence (*e.g.*, algorithmes vectoriels [7]).

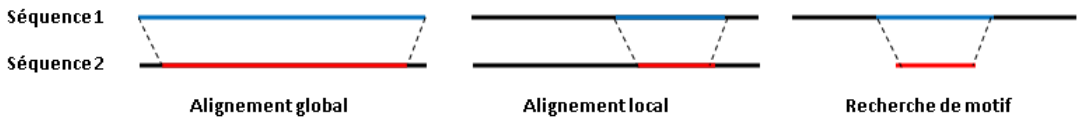


FIG. 2.3 – Les différents types d'alignement de séquences

Nous ne nous attarderons pas plus sur ces méthodes. Sauf pour dire, que pour notre projet, la recherche de motifs semble être la méthode la plus facilement adaptable à la recherche de micro-architectures dans un programme. Nous présenterons dans le Chapitre 4 notre adaptaion de la recherche de motifs basée sur les vecteurs de bits [7].

CHAPITRE 3

MÉTHODOLOGIE

Au cours de ce chapitre, nous présentons un aperçu général de notre méthodologie de recherche qui consiste principalement à faire un parallélisme entre l'évolution biologique et l'évolution logicielle. Notre méthodologie est constituée de plusieurs étapes :

- **Étape 1 : représentation du génome d'un programme**

  En bio-informatique, une séquence d'ADN est représentée par une chaîne de caractères composée d'une succession de quatres lettres (A, C, G, T) qui représentent chacune l'une des bases qui composent l'ADN *(Adénine, Cytosine, Guanine, Thynine)*. Donc, la plupart des algorithmes d'alignements de séquences d'ADN sont basés sur le traitement des chaînes de caractères. Pour pouvoir les adapter au contexte de génie logiciel, nous devons construire les génomes des programmes, sous forme de chaînes de caractères. Ces génomes doivent inclure tous les constituants qui permettent la compréhension des programmes.

  En génie logiciel, un programme peut être considéré à différents niveaux d'abstraction. Par exemple, à un niveau détaillé, nous trouvons le concept d'instructions ; à un niveau plus élevé, nous trouvons les classes et leurs relations. Avant de construire le génome, il faut préciser quel niveau d'abstraction du programme nous souhaitons étudier pour y inclure les informations du même niveau. Par exemple, si on s'intéresse à l'analyse de l'évolution de l'architecture du programme, le génome doit contenir des informations structurelles (*e.g.*, les classes et leurs relations).

– **Étape 2 : enrichissement de la représentation du génome**

Nous allons montré dans le chapitre suivant comment construire le génome d'un programme à un niveau d'abstraction élevé (niveau conceptuel ou niveau fonctionnel). Cependent, ces représentations sont limitées aux concepts d'un seul niveau à la fois. Par exemple, le génome qui représente l'architecture du programme n'inclut que les informations structurelles du programme (les noms des classes et leurs relations). Ces informations, parfois, ne sont pas suffisantes, parce que certains types d'analyses exigent d'autres détails issus d'un autre niveau d'abstraction, telle que la structure interne des classes (*e.g.*, les attributs, les signatures des méthodes, etc.).

À cette étape, nous allons définir une représentation qui inclut plusieurs niveau d'abstraction à la fois. La notion du niveau de détails permet une approche plus graduée. Elle permet d'amplifier dynamiquement les détails autour d'un centre d'intérêt tout en gardant le contexte global de l'analyse.

– **Étape 3 : définition du mapping**

L'étape fondamentale de l'analyse de l'évolution logicielle, du point de vue biologique, est d'établir la correspondance entre les concepts bio-informatiques et ceux du génie logiciel. Cette correspondance dépend du niveau d'abstraction du programme étudié et de l'objectif de l'analyse. Par exemple, si le génome du programme représente le graphe d'appels des méthodes, le *mapping* doit être fait entre les concepts des graphes d'appels et ceux des séquences biologiques. Le chapitre suivant présente un exemple de mapping entre les concepts d'homologie (en bio-informatique) et les concepts de micro-architectures stables (en génie logiciel).

– **Étape 4 : alignement des versions d'un programme**

Nous proposons d'adapter les algorithmes de recherche de motifs [8] à la comparaison des génomes des différentes versions d'un programme, dans le but de trouver leurs parties communes. Le chapitre suivant présente nos choix algorithmiques.

– **Étape 5 : définition des patrons de changements**

Dans le génome d'un programme, toutes les parties sont vitales au bon fonctionnement global du programme. Cependant, il est souvent nécessaire de s'intéresser uniquement à un ensemble de *fonctionnalités* d'un programme visibles par son utilisateur [2]. Les parties du génome d'un programme "encodant" ces fonctionnalités et l'étude de leurs évolutions pourraient faire apparaître des patrons de changements caractéristiques d'une bonne architecture. Ces derniers sont nécessaires au développement de bonnes pratiques pour réduire les coûts de développement et de maintenance. Étant donné les résultats des différences et des similarités entre les génomes de deux versions d'un même programme, nous pouvons facilement identifier l'ensemble des éléments qui changent et ceux qui ne changent pas. Par la suite, nous pourrons, pour chacun des ensembles trouvés, étudier leur stabilité dans les autres versions du programme. La figure 3.1 illustre un exemple d'alignements de versions successives d'un programme. Chaque version du programme est repésentée par un génome composé de plusieurs gènes ("encodant" des fonctionnalités différentes). L'étude de leur évolution nous montre la disparition du gène C et l'apparition d'un nouveau gène D dans la version 2. Dans la version 3, cette étude montre aussi la disparition du gène B et sa ré-apparition dans la version 4. Les autres gènes : A , E et F sont restés stables.

En bio-informatique, de nombreux algorithmes efficaces de fouille de graphes existent. Ils sont généralement utilisés pour l'analyse des réseaux d'interac-

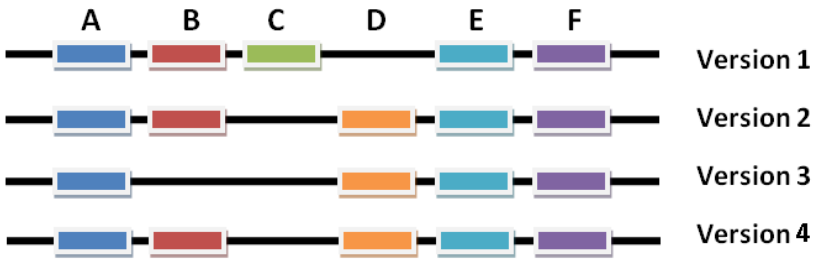


FIG. 3.1 – Un exemple d'alignement des versions successives d'un programme

tions moléculaires, pour chercher des motifs récurrents dans plusieurs parties du réseau. Les techniques utilisées sont souvent basées sur la recherche des sous-graphes les plus fréquents dans un graphe [15, 23]. En génie logiciel, les programmes peuvent facilement être représentés par des graphes. Les diagrammes UML le sont d'ailleurs souvent. Les classes, les méthodes et les attributs représentent généralement les noeuds du graphe, et les relations entre les classes représentent les arcs du graphe. La découverte de motifs fréquents dans un programme peut alors être étudiée comme un problème de recherche de sous-graphes fréquents dans un graphe. Nous appliquerons, par la suite, des règles d'association entre ces sous-graphes, pour extraire automatiquement des patrons de changements. La principale contribution de cette partie réside, d'une part, dans la proposition d'un algorithme qui extrait des sous-graphes fréquents, et d'autre part, dans le développement d'une méthode permettant la découverte de règles d'associations entre les motifs découverts.

– **Étape 6 : comparaison systématique et exhaustive des outils**

La mise en place d'une méthode d'évaluation des outils de détection des patrons de changements est nécessaire. Il existe plusieurs outils à différents niveaux d'abstraction du programme. La constitution d'une grille de test facilitant la comparaison de ces outils est apparue indispensable. Pour comparer ces outils, il faut avoir des données sur lesquelles il faut mener les évaluations

et avoir aussi des critères pour juger de leurs représentativités. Nous devons constituer un corpus, c'est-à-dire, un ensemble représentatif de logiciels de grandes tailles sur lesquels il faut tester les différents outils.

– **Étape 7 : évaluation**

  La mise en place de protocoles d'expérimentation sur divers programmes permettra de conduire des évaluations empiriques pour valider nos résultats. Ces études nous permettront d'analyser la corrélation entre les patrons de changements et l'apparition des défauts de conception dans les logiciels (des erreurs).

CHAPITRE 4

RÉSULTATS OBTENUS À CE JOUR

Lorsqu'un programme évolue fréquemment, il devient intéressant de comparer ses versions successives pour tirer des patrons de changements facilitant sa maintenance. Dans ce qui suit, nous présentons l'état d'avancement de nos travaux. Les étapes abordées du projet sont décrites, ainsi que les résultats obtenus à ce jour. Les étapes du projet n'ayant pas encore été abordées seront décrite dans le Chapitre 5.

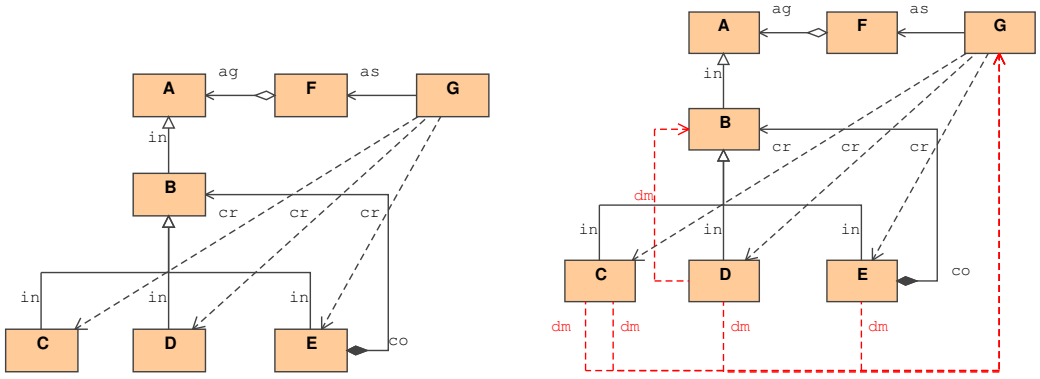
4.1 Étape 1 : représentation du génome d'un programme

Entrée : un modèle d'un programme décrivant les entités (classes, interfaces) et éléments (associations, héritages, etc) du programme à analyser.

Sortie : le génome du niveau conceptuel du programme sous forme d'une séquence. Cette dernière est composée des classes du programme et leurs relations.

Méthodologie : À cette étape, nous utilisons les travaux de Kaczor *et al.* [18] qui proposent une méthode de transformation du modèle d'un programme en une chaîne de caractères. Cette méthode consiste, tout d'abord, à parcourir le modèle du programme afin de construire un graphe préliminaire avec les entités comme sommets et les éléments comme arêtes. Les éléments considérés sont les relations binaires interclasses : création, spécialisation, implémentation, utilisation, association, agrégation et composition. Ces graphes sont orientés puisque les relations binaires interclasses le sont. Ensuite, la transformation d'un modèle de programme en une chaîne de caractères s'effectue en deux étapes. La première étape consiste à transformer le graphe du modèle en un graphe eulérien en ajoutant une liste optimale d'arêtes nulles, appellées *dm (dummy relationships)*, au graphe non-eulérien à l'aide de l'algorithme de *transportation simplex*. La deuxième étape consiste à

trouver le plus court chemin visitant chaque arête du graphe eulérien au moins une fois. Le cycle eulérien est, pour un graphe eulérien, la solution optimale au *problème du postier chinois*. La figure 4.1 représente un exemple de transformation du modèle d'un programme en un graphe eulérien, pour obtenir la représentation sous forme de chaîne de caractères (*i.e.*, génome du programme), qui est composée de la série des étiquettes des sommets et des arêtes parcourues par l'algorithme du postier chinois.



(a) Modèle du programme

(b) Modèle Eulerian

```
A in B in D dm B in E co B in C dm G cr C dm G cr D dm G cr E dm G as F ag A
```

(c) Transformation du Modèle Eulerian en une chaîne de caractères

FIG. 4.1 – Un exemple de génome du niveau conceptuel d'un program (source [18]).

Cette méthode nous permet de représenter de façon unique un programme sous forme d'une chaîne de caractères qui représente les classes du programme et les relations entre celles-ci. Cependent, cette repésentation est limitée aux informations du niveau conceptuel du programme, ce qui n'est parfois pas suffisant. En effet, certaines tâches d'analyse nécessitent la manipulation d'autres informations de bas niveau (code source) du programme, comme par exemple, l'analyse des changements des attributs et des signatures de méthodes d'une classe.

4.2 Étape 3 : définition du mapping

Entrée : un modèle de programme décrivant les entités (classes, interfaces) et éléments (associations, héritages, etc) du programme à analyser.

Sortie : une table de correspondance entre les concepts d'homologie bio-informatique et micro-architectures stables.

Méthodologie : nous avons présenté comment convertir un modèle de programme en une représentation unique sous forme d'une chaîne de caractères qui décrit les classes du programme et leurs relations. Ceci constitue le point de départ pour commencer à faire le parallélisme entre l'évolution biologique et l'évolution logicielle. En effet, cette représentation pourrait nous servir à définir le génome d'un programme. L'étude de l'évolution de ce programme consiste donc à aligner les génomes de ses différentes versions. La Table 4.1 présente un mapping entre les concepts des micro-architectures et les concepts bio-informatiques.

Concepts de l'évolution	
En bio-informatique	**En génie logiciel**
Individus	Versions d'un programme
Génome	Architecture du programme
Gènes	Micro-architectures
Acides Nucleiques (A, C, G, T)	Tokens (*e.g.*, noms des classes, relations d'aggregations)
Gènes homologues	Micro architectures stables
Spéciation	Réutilisation (*e.g.*, réutilisation d'une classe d'une version antérieure)
Duplication	Clonage
Perte de gène	Suppression (*e.g.*, suppression d'une classe et ses relations)

TAB. 4.1 – Mapping entre les concepts de l'évolution biologique et l'évolution logicielle.

Avec ce mapping, la recherche de gènes homologues en bio-informatique est équivalente à la recherche des micro-architectures stables d'un programme.

4.3 Étape 4 : alignement des versions d'un programme

Entrée : les génomes des différentes versions du programme à analyser.

Sortie : les similarités entre les différentes versions du programme.

Méthodologie : nous présentons dans ce qui suit deux catégories d'algorithmes d'alignements : les techniques basées sur la programmation dynamique, les techniques basées sur les vecteurs de bits.

1. Programmation dynamique : l'approche par programmation dynamique semble à priori appropriée pour l'alignement de génomes de différentes versions du programme. La possibilité de pouvoir attribuer des coûts différents aux substitutions dépendamment des caractères est intéressante. La recherche de similarité entre deux chaînes de caractères consiste à donner une valeur de similarité à la place d'une distance. Nous cherchons alors à maximiser la fonction objective plutôt qu'à la minimiser. L'approche par programmation dynamique est toutefois confrontée à deux problèmes importants. Premièrement, le génome d'une version de programme est composé de la série des étiquettes des sommets et des arêtes parcourues par l'algorithme du postier chinois. L'ordre du parcours de ces entités peut différer d'une version à une autre, *i.e.*, une séquence de triplets communs entre les deux versions ne se trouvent pas nécessairement de manière continue dans le génome de la deuxième version du programme. Aussi, les triplets (classe relation classe) où la relation est une "*dummy reltionship*" ne doivent pas être considérés dans l'alignement car ils ne représentent pas des entités importantes du programme. Un deuxième problème est celui de l'efficacité de ces algorithmes. La complexité en temps de l'alignement de deux séquences S et T, de tailles m et n est de l'ordre $O(nm)$, ce qui est très coûteux pour l'analyse des programmes de grandes tailles. Étant donné ces problèmes, l'approche par programmation dynamique n'est finalement pas vraiment appropriée pour l'alignement de différentes versions d'un programme.

2. Algorithmes vectoriels : nous proposons d'utiliser les algorithmes vectoriels qui sont reconnus pour leur efficacité, lorsque le motif à rechercher est petit. Afin d'obtenir un algorithme efficace, nous avons développé un algorithme vectoriel itératif s'inspirant des approches numériques de recherche exacte en bio-

informatique [8]. Cette approche consiste à chercher les triplets (classe relation classe) communs en effectuant des opérations binaires standard (ou-logique, et-logique, décalage à droite, décalage à gauche) sur des séquences de bits appelées vecteurs caractéristiques.

Le vecteur caractéristique d'un symbole s dans une chaîne de caractère $x = x_1...x_m$, est défini par le séquence de bits $\mathbf{s} = (s_1...s_m)$ d'une longueur égale à la longueur de la chaîne s, tel que :

$$s_i = \begin{cases} 1 & \text{si } x_i = s \\ 0 & \text{sinon.} \end{cases}$$

La valeur du vecteur caractéristique d'un symbole à la position i est égale à 1 si et seulement si ce symbole se trouve à la position i dans la chaîne du programme. Le décalage à droite ($\leftarrow$) d'un vecteur caractéristique $\mathbf{s} = (s_1...s_m)$ est défini comme $\mathbf{s} = (s_m s_1...s_{m-1})$. Tous les éléments ont été décalés d'une position vers la droite de manière circulaire. Le décalage à gauche ($\rightarrow$) de $\mathbf{s}$ est : $\mathbf{s} = (s_2...s_m s_1)$. Par exemple, dans le génome de la Figure 4.1(c), le vector caractéristique de la classe $\mathbf{A}$ est défini : $\mathbf{A} = \mathbf{1}\underbrace{000000000000000000000000000}_{29}\mathbf{1}$.

Nous proposons d'utiliser les vecteurs caractéristiques pour identifier les triplets communs entre deux versions. Notre algorithme bit-vector lit le génome de la première version triplets par triplets, et il applique des opérations binaires sur les vecteurs de bits. Par exemple, pour le triplet $T = \mathbf{A\ in\ B}$ l'algorithme vérifie si la conjonction du vecteur caractéristique de $\mathbf{B}$ dans la deuxième version avec la conjonction des vecteurs caractéristiques $\rightarrow$ **in** et $\rightarrow\rightarrow$ $\mathbf{A}$ n'est pas nulle. La présence de 1 dans le vecteur résultat signifie que le triplet $T = (\mathbf{A\ in\ B})$ est commun entre les deux versions du programme.

Par exemple, pour le programme représenté par la figure 4.1(c), le vecteur caractéristique des entités A, in, B est :

$\mathbf{A} = \mathbf{1}\underbrace{0000000000000000000000000000}_{29}\mathbf{1}.$

$\mathbf{in} = 0\mathbf{1}0\mathbf{1}000\mathbf{1}0001\underbrace{0000000000000000000}_{19}.$

$\mathbf{B} = 00\mathbf{1}000\mathbf{1}0001\underbrace{00000000000000000000}_{20}.$

Pour chercher le triplet **A in B**, notre algorithme lit la chaîne par triplets. Il recherche les entités avant et après le symbole ***in*** dans la chaîne du programme avec des opérations binaires sur les vecteurs :

$0\mathbf{11}\underbrace{0000000000000000000}_{28} = (\rightarrow\rightarrow \mathbf{A}).$

$00\mathbf{1}0\mathbf{1}000\mathbf{1}0001\underbrace{000000000000000000}_{18} = (\rightarrow \mathbf{in}).$

$001\underbrace{0000000000000000000}_{28} = (\rightarrow\rightarrow \mathbf{A}) \wedge (\rightarrow \mathbf{in}).$

$00\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{00000000000000000000}_{20} = \mathbf{B}.$

$001\underbrace{0000000000000000000}_{28} = (\rightarrow\rightarrow \mathbf{A}) \wedge (\rightarrow \mathbf{in}) \wedge \mathbf{B}.$

4.4 Étape 5 : définition des patrons de changements

Le patron de stabilité décrit l'ensemble d'artéfacts qui ne changent pas au cours de l'évolution du programme.

4.4.1 Définition des micro-architectures stables

Entrée : les similarités entre les différentes versions du programme.

Sortie : les micro-architectures stables.

Méthodologie : l'identification de micro-architectures stables dans un programme est un champ de recherche reconnu du génie logiciel. Il est souvent une étape nécessaire dans l'identification de patrons de changements de l'architecture d'un logiciel. Un patron de changement peut être perçu comme un ensemble de micro-architectures qui maintiennent certaines propriétés au cours de l'évolution, comme la stabilité par exemple.

En génie logiciel, nous proposons de faire le parallélisme entre le concept d'homologie en biologie et le concept des micro-architectures stables en génie logiciel. Une micro-architecture définit l'ensemble des classes et leurs relations. Leur stabilité est l'équivalent de l'homologie en biologie, une micro-architecture stable entre deux versions signifie qu'elle est maintenue dans la seconde version sans aucun changement (voir figure 4.2), ce qui signifie qu'elle est susceptible de jouer un rôle important dans le programme. Pour détecter les micro-architectures stables, on effectue l'alignement des génomes de deux versions successives du programme.

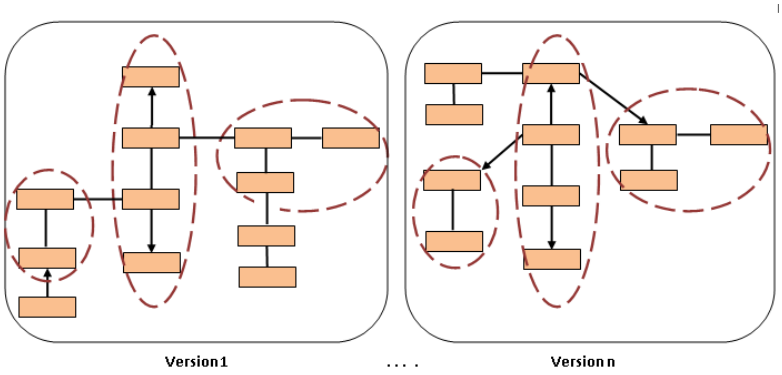


FIG. 4.2 – Un exemple d'homologie dans l'architecture d'une programme : il existe trois micro-architectures stables, elles sont encerclées par des ovales.

Nous proposons un algorithme de "clustering" incrémental pour regrouper les triplets connectés dans des ensembles appelés clusters, chacun définissant une micro-architecture stable. Cet algorithme nécessite un seul balayage de la liste des triplets. Chaque triplet lu sera soit affecté à l'un des clusters existants, soit utilisé pour créer un nouveau cluster. À chaque itération, deux clusters peuvent être fusionnés si le nouveau triplet ajouté les relie.

L'algorithme 1 traverse la liste de tous les triplets communs S, pour chaque triplet T dans S et pour chaque cluster C dans L, il teste s'il existe un triplet T^* dans C qui est en relation avec lui, alors le triplet T sera ajouté au cluster C qui sera marqué par *Cluster to be merged* (lignes 3-8). S'il existe un autre cluster contenant

Algorithm 1 Principe de notre algorithme de clustering.

```
1:  L ← EmptyList{Clusters}
2:  S ← List{Commontripletsbetweentwoprogramverions}
3:  for each Triplet T in S do
4:    for each Cluster C in L do
5:      if T is in relations with the existing triplet T* in C then
6:        if T is not added to any cluster then
7:          ADD T to C.
8:          ClusterToBeMerged ← C.
9:        else
10:         MERGE ClusterToBeMerged to C.
11:       end if
12:     end if
13:   end for
14:   if T is not added to any cluster then
15:     Create a new Cluster C*.
16:     ADD T to C*.
17:     ADD C* to L.
18:   end if
19: end for
```

un autre triplet T^* qui en relation avec le triplet T, alors le cluster courant sera fusionné avec le cluster qui est marqué (lignes 9-10). Après avoir vérifier tous les clusters C dans L, si le triplet T n'était pas assigné à aucun d'entre eux, alors un nouveau cluster C^* sera créé et le triplet T lui sera ajouté (lignes 14-17). Cet algorithme nous retourne, les micro-architectures stables représentée par chaque cluster.

4.4.2 Définition du tunnel

Entrée : Les micro-architectures stables.

Sortie : Les micro-architectures qui sont dans le tunnel.

Méthodologie : Le concept du tunnel identifie les micro-architectures les plus stables d'un programme et donc les plus susceptibles d'en représenter l'épine dorsale. Partant de la première version, nous procèdons par alignements successifs des génomes des différentes versions. Nous retrouvons l'ensemble des micro-architectures alignées tout le long de la durée de vie du logiciel.

4.5 Étape 7 : évaluation

Nous proposons de procéder par étude de cas des programmes *open sources* pour valider notre approche. Les résultats attendus vont au-delà de la performance de l'algorithme car ils permettent d'inférer des connaissances utiles quant à l'évolution du programme étudié, comme la stabilité, la complexité, etc.

4.5.1 Études de cas

Nous présentons, dans cette section, deux études de cas sur la détection des micro-architectures stables au cours de l'évolution de deux programmes. Étant donné les problèmes de performance des différentes techniques d'identification des patrons de changements existantes, le but de notre approche est d'améliorer le temps d'identification tout en obtenant une meilleur qualité de résultats. En effet, les points essentiels de notre approche résident dans l'exactitude des occurences des micro-architectures localisées et dans le concept du tunnel. Dans cette section, deux études de cas sur deux programmes (JFreeChart et Xerces-J) sont présentées et les résultats sont comparés avec ceux d'une approche récente [21].

4.5.1.1 Hypothèses

Nous allons étudier s'il est possible d'adapter un algorithme bio-informatique à l'étude de l'évolution des architectures de programmes. Ensuite, nous allons vérifier si les résultats de notre algorithme fournissent aux programmeurs une vision globale et claire sur l'évolution des programmes. Nous étudions donc les questions de recherche suivantes :

- **RQ1 :** Est-il possible d'appliquer nos algorithmes d'analyse de l'évolution de l'architecture des programmes $JFreeChart$ et $Xerces-J$?
- **RQ2 :** Est-il possible d'utiliser nos algorithmes pour trouver toutes les micro-architectures stables, ainsi que les tunnels dans $JFreeChart$ et $Xerces-J$?

Programmes	Entités (en classes)	Génomes (en tokens)	Versions	Dates
JFreeChart v1.0.13	1,335 9,105	87,227 1,089,345	51	25/11/00 20/04/09
Xerces-J v2.9.0	5,100 12,585	162,583 1,195,353	36	05/11/99 22/11/06

TAB. 4.2 – Descriptions statistiques des programmes $JFreeChart$ et $Xerces - J$.

4.5.1.2 Objets

Il existe une multitude de programmes *open sources* disponibles sur le Web. Les facteurs que nous avons utilisés pour juger de la représentativité d'un programme sont les suivants :

1. Taille intéressante : un programme composé de seulement quelques modules n'est pas intéressant parce qu'il est assez simple.
2. Évolution importante : un programme qui n'évolue pas n'est certainement pas intéressant. Nous cherchons un logiciel en évolution depuis plusieurs années et qui offre plusieurs versions ;

Notre approche a été testée sur deux logiciels libres de taille moyenne, 51 versions de $JFreeChart$ et 35 versions de $Xerces - J$. La table 4.2 présente quelques descriptions statistiques de ces programmes.

- $JFreeChart$[1] est une librairie gratuite d'édition de graphiques pour la plateforme Java. Elle est conçue pour générer et inclure des graphiques dans des applications.
- $Xerces - J$[2] est un parseur de documents XML, qui est utilisé dans de nombreux produits commerciaux. Il implémente plusieurs standards comme XML, XML schéma, DOM, SAX, et il est distribué avec le JDK de Sun.

[1] http ://www.jfree.org/jfreechart/
[2] http ://xerces.apache.org/

4.5.1.3 Résultats

Dans la figure 4.3, nous pouvons observer la croissance de Xerces-J. Dès les premières versions (1.0.1), Xerces montre une forte croissance des triplets (classe - relation - classe) communs entre deux versions successives. Cette courbe nous donne une idée sur la stabilité de Xerces-J. Lorsqu'on regroupe les triplets qui sont reliés on trouve les micro-architectures stables entre deux versions. La croissance des triplets signifie que l'architecture de Xerces-J devient de plus en plus stable. Cette croissance a une apparence assez linéaire jusqu'à la version 1.4.4 où on retrouve une grande réduction du nombre des triplets. Ceci est l'effet d'un changement de version majeure (1.4.4 vers 2.0.0). À ce moment, une partie du code a été déplacé dans des binaires distincts, distribués séparément. Lors de la deuxième version majeure, le nombre des triplets recommence à croître en "*escalier*". En lisant les notes de mise à jour, nous avons compris que les "*marches*" correspondent aux versions de maintenance corrective durant laquelle Xerces-J a été changé à cause de bogues. La deuxième courbe nous indique les triplets qui ont été conservés dès la première version de Xerces-J. Les micro-architectures stables composées par ces triplets constituent l'épine dorsale de Xerces-J. Nous remarquons que le nombre des triplets qui font partie du tunnel a connu une réduction à la version 2.0.0, ensuite il resté constant. Donc l'épine dorsale du programme est devenue plus stable.

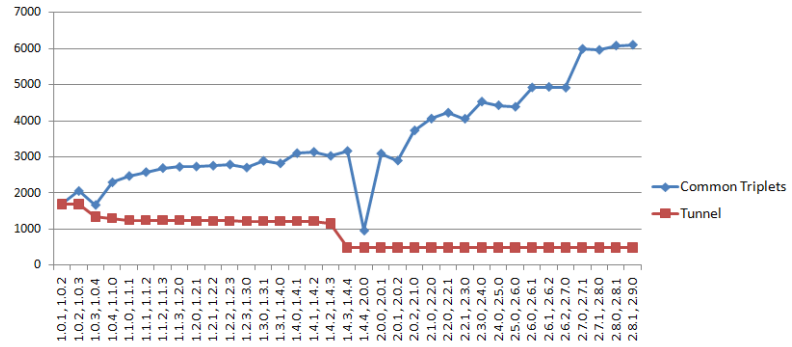


FIG. 4.3 – L'évolution des micro-architectures stables de Xerces-J.

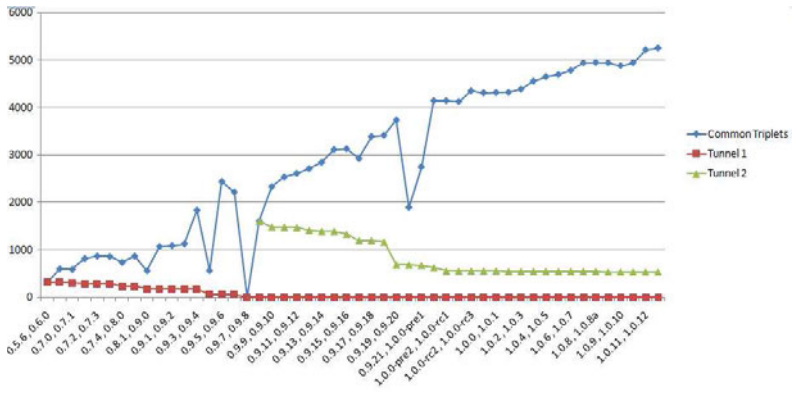


FIG. 4.4 – L'évolution des micro-architectures stables JFreeChart.

L'évolution de JFreeChart est illustrée dans la figure 4.4. En regardant la croissance du nombre des triplets communs entre deux versions successives, nous remarquons trois réductions majeures aux versions : 0.9.4, 0.9.8 et 0.9.20. En lisant les notes de mise à jour, nous avons compris qu'il avait des plusieurs renommages. Par exemple à la version 0.9.8, le *package* principal $com.jrefinery.*$ a été renommé par $org.jfree.*$, ce qui affecte le nom de beaucoup de classes dans les triplets de le génome représentant cette version. Lors de la comparaison des génomes des versions 0.9.7 et 0.9.8, aucun triplet n'a été apparié et le nombre de micro-architectures stables était nulle ; Donc, nous avons calculé un deuxième tunnel qui inclue les nouveaux noms dans les triplés. Ce tunnel a subi une décroisance du nombre des triplets à la version 0.9.20, ensuite il est devenu stable.

L'étude de l'évolution de l'architecture des programmes nous permet d'avoir une vision global sur :

- **La complexité des programmes :** la longueur des génomes est un bon indicateur sur la taille des programmes en terme de classes et leurs relations.
- **Les triplets ajoutés ou supprimés :** les triplets qui ont été ajoutés au programme peuvent être facilement détectés, car ils sont ajoutés au génome de cette version. Les triplets supprimés peuvent aussi être facilement détectées

par leur absence dans le génome de cette version.

- **Les phases de croissance et de stabilité dans l'évolution :** une phase de croissance est indiquée par une augmentation du nombre de micro-architectures stables, alors que pendant la phase de stabilité le nombre de micro-architectures stables restera le même.
- **Le tunnel d'un programme.** L'identification du tunnel doit permettre de détecter la partie stable du programme. Il existe deux types de tunnels. L'algorithme ECGM détecte le premier type, qui définit l'ensemble des classes qui a maintenu une structure stable de relations au cours de leur évolution Dans notre approche, nous découvrons le deuxième type de tunnel, qui définit l'ensemble des micro-architectures qui restent stables durant toute leur durée de vie.
- **Les micro-architectures stables :** l'algorithme ECGM ne permet pas d'identifier les micro-achitectures stables. Dans notre approche, nous identifions les micro-architectures stables comme étant des parties du programme les plus robustes aux changements.
- **La stabilité des résultats :** l'algorithme ECGM est basé sur des méta-heuristiques, et donc l'ensemble des classes appariées peut ne pas être optimal. Avec notre approche, on trouve toujours les mêmes résultats concernant les parties stables d'un programme.

4.5.2 Limites de notre approche

Les changements entre deux versions d'un même logiciel incluent des *renommages* de classes qui rendent le sous-problème non trivial. L'alignement de deux versions basé uniquement sur les noms des classes a de bonnes chances de conduire à des appariements profondément erronés. Les classes, ayant été renommées ou déplacée dans un autre package, ne pouvant être retracées.

CHAPITRE 5

TRAVAUX FUTURS ET ÉCHÉANCIER

Dans ce rapport, nous avons cherché à mieux comprendre le phénomène de l'évolution des programmes. Pour ce faire, nous avons fait un premier *mapping* entre l'évolution biologique et l'évolution logicielle. Pour commencer, nous avons appliqué un algorithme bio-informatique pour détecter les micro-architectures stables au cours de l'évolution d'un programme. Partant du génome d'une version donnée, on a procédé par appariements successifs avec les génomes des versions successives. On retrouve aussi le tunnel de l'application qui est l'ensemble des classes appariées tout le long de la durée de vie du programme. Le tunnel identifie les classes les plus stables d'une application et donc les plus susceptibles d'en représenter l'épine dorsale.

5.1 Travaux futurs

5.1.1 Étape 1 : représentation du génome d'un programme

Nous avons montré dans le chapitre 4 comment construire le *génome du niveau conceptuel* d'un programme, par transformation du modèle de programme en une chaîne de caractères. Il nous reste à définir les génomes du niveau fonctionnel par transformation du graphe d'appels de fonctions en une chaîne de caractères, en utilisant le même algorithme utilisé par Kaczor *et al.* [18].

5.1.2 Étape 2 : enrichissement de la représentation du génome

Nous proposons de définir un génome incluant les informations du niveau d'implémentation du programme, comme suit :

– La traduction des diagrammes de classes dans un format XMI équivalent.

XMI (XML Metadata Interchange) qui est un standard très souvent utilisé pour décrire des diagrammes UML [32]. Pour chaque fichier XMI, une structure sous forme d'arbre est construite. Chaque noeud possède : un ensemble d'informations de définition, comme le nom d'une classe ou la signature d'une méthode, un ensemble de propriétés atomiques, comme la visibilité, et plusieurs ensembles d'éléments fils.

– La génération d'une chaîne de caractères qui décrit représente de façon unique une classe. Cette chaîne est composées des caractéristiques internes de la classe (les signatures des méthodes, les attributs, etc.) et ses relations avec les autres classes du programme. Cette représentation pourrait nous servir à définir le génome d'un programme. L'étude de l'évolution de ce programme consiste donc à aligner les génomes de ses différentes versions.

5.1.3 Étape 4 : alignement des versions d'un programme

L'algorithme d'alignment présenté dans le chapitre 4 est basé sur la comparaison des noms d'entités qui composent le génome du programme (*e.g.*, noms des classes et noms des relations). Il est donc très vulnérable à un "renommage" important entre deux versions. Ceci justifie de faire des comparaisons à un plus bas niveau de granularité, dans le cas où la classe recherchée n'est pas trouvée dans la deuxième version. Nous proposons donc d'ajouter des mesures calculant la similarité de cette classe recherchée avec toutes les autres classes de la version 2. Ces mesures seront fondées sur les caractéristiques internes des classes, comme par exemple, le pourcentage des types d'attributs communs et des signatures des méthodes communes, et/ou les caractéristiques structurelles, comme le pourcentage des relations communes.

L'approche par programmation dynamique (*e.g.*, algorithme de Needleman-Wunsch [30]) semble appropriée pour l'alignement de génomes du niveau d'implémentation (incluant les caractéristiques internes des classes), pour la détection du renom-

mage des classes entre les différentes versions du programme. La possibilité de pouvoir attribuer des coûts différents aux substitutions dépendamment des caractères est intéressante. La recherche de similarité entre deux chaînes de caractères (représentant deux classes de différentes versions) consiste à donner une valeur de similarité à la place d'une distance. Nous cherchons alors à maximiser la fonction objective plutôt qu'à la minimiser.

5.1.4 Étape 5 : définition des patrons de changements

5.1.4.1 Étude de l'évolution des fonctionnalités

Nous proposons d'utiliser l'approche décrite dans [2] pour selectionner les parties du génome concernées par certaines fonctionnalités. Ensuite, nous étudions leurs évolutions pour faire apparaître des patrons de changements.

5.1.4.2 Étude de l'évolution des motifs fréquents

Nous proposons de représenter les programmes par des graphes, pour pouvoir adapter les techniques de recherche de motifs dans les réseaux d'intéractions de protéines [15, 23] à la recherche de motifs de conception dans un programme. Ensuite, pour extraire les patrons de changements, nous appliquerons des règles d'association entre ces motifs trouvés (sous-graphes).

5.1.4.3 Étude d'une généalogie des changements

Nous proposons d'étudier une généalogie des changements qui décrit comment des groupes d'artéfacts changent ensemble sur plusieurs versions du programme. Dans cette généalogie, nous allons définir un modèle qui décrit l'évolution des groupes d'artefacts (*e.g.*, classes) qui proviennent du même groupe ancêtre. Ce modèle représente la façon dont chaque élément dans un groupe a changé par rapport à d'autres éléments dans le même groupe. Nous allons définir les différents

types de patrons de changements, comme suit :

- **Patron d'ajout** : il décrit l'ajout d'un nouvel artefact (classe, méthode, etc.) à la nouvelle version.
- **Patron de supression** : il décrit la suppression d'un artefact dans la nouvelle version.
- **Patron de changement cohérent** : il décrit l'ensemble des artefacts qui changent toujours ensemble (voir la figure 5.1).
- **Patron de changement incohérent** : il décrit l'ensemble des changements qui devaient être appliqués à un artéfact qui n'a pas changé avec son groupe (voir la figure 5.1).
- **Patron de stabilité** : il décrit l'ensemble d'artéfacts qui ne changent pas (restent stables) au cours de l'évolution du programme.

5.1.5 Étape 6 : comparaison systématique et exhaustive des outils

Nous proposons de constituer une grille de test et un corpus d'évaluation qui facilitent la comparaison des outils de détection de changements.

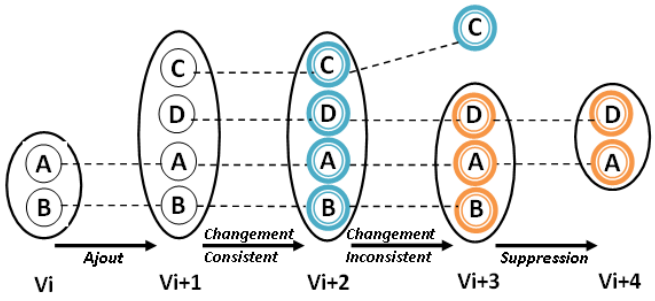


FIG. 5.1 – Un exemple d'un changement cohérent entre la version V_{i+1} et V_{i+2} ; Les classes (A, B, C, D), appartiennent à un même groupe de changement (*i.e.*, elles changent toujours ensemble). Un exemple d'un changement incohérent entre la version V_{i+2} et V_{i+3} ; La classe C n'a pas changé avec son groupe.

5.2 Échéancier

La figure 5.2 et le tableau présentent notre plan de recherche ainsi que nos principales phases de publications.

Session	Date	Conférence	Contribution
H10	02 - 2010	**ICPC'10** : Conf. Internationale sur la compréhension de programmes	La détection des micro-architectures stables
H10	04 - 2010	**ICSM'10** : Conf. Internationale sur la maintenance de programmes	L'alignement amélioré par des mesures de similarités stucturelles
E10	07 - 2010	**WCRE'10** : Conf. Internationale sur la ré-ingénierie des programmes	Étude de l'évolution des fonctionnalités
A10	12 - 2010	**Journal IST** : *Information and Software Technology*	Détection des motifs fréquents
H11	02 - 2011	**ICPC'11** : Conf. Internationale sur la compréhension de programmes	Étude de la stabilité des motifs
H11	04 - 2011	**ICSM'11** : Conf. Internationale sur la maintenance de programmes	Étude de la taxonomie des changements
E11	06 - 2011		Rédaction de la thèse
E11	07 - 2011	**WCRE'11** : Conf. Internationale sur la ré-ingénierie des programmes	Comparaison des outils de détection de changements
A11	11 - 2011	**Journal TSE** : *IEEE Transactions of Software Engineering*	Une Approche complète, méthodologie, et résultats
A11	12 - 2011		Dépot de la thèse

Tab. 5.1 – Plan de publications.

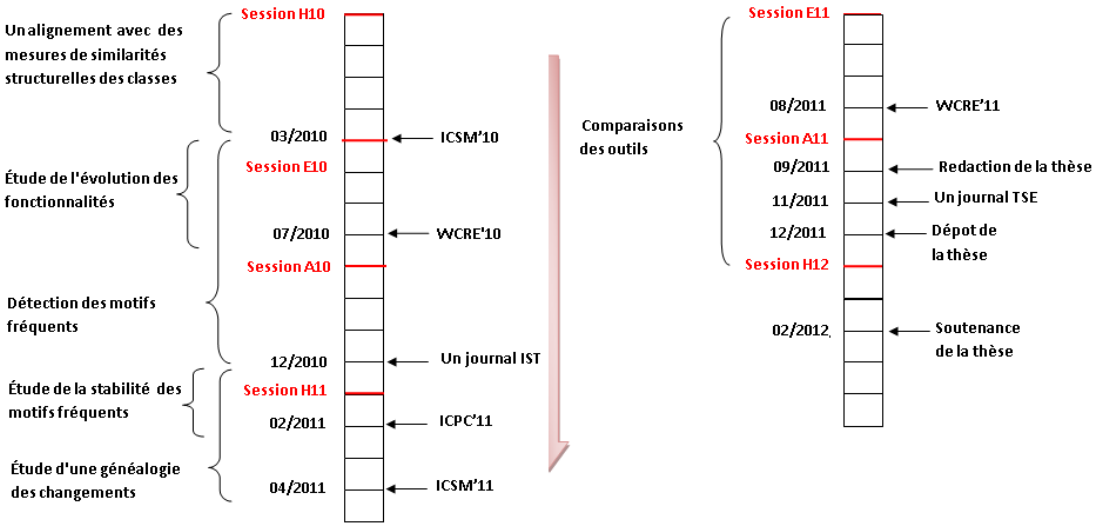


FIG. 5.2 – Plan de recherche.

BIBLIOGRAPHIE

[1] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Maintaining traceability links during object-oriented software evolution. *Software : Practice and Experience*, 31(4) :331–355, 2001.

[2] Giuliano Antoniol, Bruno Caprile, Alessandra Potrich, and Paolo Tonella. Design-code traceability for object-oriented systems. *Ann. Software Engineering*, 9(1-4) :35–58, 2000.

[3] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification : An epidemiological metaphor. *IEEE Transactions on Software Engineering (TSE)*, 32(9) :627–641, September 2006. 15 pages.

[4] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. *Principles of Software Evolution, International Workshop on*, 0 :31–40, 2004.

[5] Ricardo A. Baeza-Yates and Gonzalo Navarro. A faster algorithm for approximate string matching. In *Proceedings of the* 7^{th} *Symposium on Combinatorial Pattern Matching*, pages 1–23. Springer–Verlag LNCS, 1996.

[6] Laszio A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3) :225–252, 1976.

[7] Anne Bergeron and Sylvie Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1) :53–66, 2002.

[8] Anne Bergeron and Sylvie Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1) :53–65, 2002.

[9] Salah Bouktif, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Extracting change-patterns from CVS repositories. In Susan Elliott Sim and Massimiliano

Di Penta, editors, *Proceedings of the* 13^{th} *Working Conference on Reverse Engineering (WCRE)*, pages 221–230. IEEE Computer Society Press, October 2006. 10 pages.

[10] T. A. Corbi. Program understanding : challenge for the 1990's. *IBM Syst. J.*, 28(2) :294–306, 1989.

[11] Barthélémy Dagenais and Martin P. Robillard. Semdiff : Analysis and recommendation support for api evolution. In *ICSE'09 : International Conference on Software Engineering*, pages 599–602, 2009.

[12] Harald Gall, Mehdi Jazayeri, Rene Kloesch, and Georg Trausmuth. Software evolution observations based on product release history. *Software Maintenance, IEEE International Conference on*, 0 :160, 1997.

[13] Michael W. Godfrey and Qiang Tu. Evolution in open source software : A case study. In *ICSM '00 : Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[14] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK, 1997.

[15] J. Huan, W. Wang, A. Washington, J. Prins, R. Shah, and A. Tropsha. Accurate classification of protein structural families using coherent subgraph analysis. In *In Proc. Pacific Symposium on Biocomputing*, pages 411–422, 2004.

[16] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[17] Mehdi Jazayeri. On architectural stability and evolution. In *da-Europe '02 : Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, pages 13–23, London, UK, 2002. Springer-Verlag.

[18] Olivier Kaczor, Yann-Gael Gueheneuc, and Sylvie Hamel. Efficient identification of design patterns with bit-vector algorithm. In *CSMR '06 : Proceedings*

of the Conference on Software Maintenance and Reengineering, pages 175–184, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering.*, 25(4) :493–509, 1999.

[20] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07 : Proceedings of the 29th international conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, and Giuliano Antoniol. Recovering the evolution stable part using an ecgm algorithm : Is there a tunnel in mozilla ? In *CSMR '09 : Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 179–188, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Joseph B. Kruskal and Mark Liberman. The symmetric time-warping problem : From continuous to discrete. In David Sankoff and Joseph B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules : The Theory and Practice of Sequence Copmparison*. Addison-Wesley, 1983.

[23] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *ICDM '01 : Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 313–320, Washington, DC, USA, 2001. IEEE Computer Society.

[24] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96 : Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[25] M. M. Lehman and L. A. Belady, editors. *Program evolution : processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[26] Manny M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. volume 68, pages 1060–1076, 1980.

[27] Meir M. Lehman, Juan F. Ramil, P. D. Wernick, Dewayne E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. IEEE Symp. Software Metrics*, pages 20–32. IEEE Computer Society, 1997.

[28] David Mandelin, Doug Kimelman, and Daniel Yellin. A bayesian approach to diagram matching with application to architectural models. In *ICSE '06 : Proceedings of the 28th international conference on Software engineering*, pages 222–231, New York, NY, USA, 2006. ACM.

[29] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3) :395–415, 1999.

[30] S.B. Needlman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. . *J. Mol. Biol.*, 48 :443–453, 1970.

[31] Thomas M. Pigoski. *Practical Software Maintenance : Best Practices for Managing Your Software Investment.* Wiley, New York, 1996.

[32] Nicholas Routledge, Linda Bird, and Andrew Goodchild. Uml and xml schema. In *ADC '02 : Proceedings of the 13th Australasian database conference*, pages 157–166, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[33] N. F. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering.*, 13(3) :303–310, 1987.

[34] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147 :195–197, 1981.

[35] Ian Sommerville. *Software Engineering.* Addison-Wesley, sixth edition, 2000.

[36] Burton E. Swanson. The dimensions of maintenance. In *Intl. Conf. on Software Engineering*, pages 492–497, San Francisco, California, 1976. IEEE Computer Society.

[37] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *IWPC '02 : Proceedings of the 10th International Workshop on Program Comprehension*, page 127, Washington, DC, USA, 2002. IEEE Computer Society.

[38] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. Visualization of CVS repository information. In Susan Elliott Sim and Massimiliano Di Penta, editors, *Proceedings of the* 13^{th} *Working Conference on Reverse Engineering*. IEEE Computer Society Press, Octobre 2006.

[39] Zhenchang Xing. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering.*, 31(10) :850–868, 2005. Member-Stroulia, Eleni.

[40] Zhenchang Xing and Eleni Stroulia. Understanding class evolution in object-oriented software. *International Conference on Program Comprehension*, 0 :34, 2004.

[41] Zhenchang Xing and Eleni Stroulia. Umldiff : an algorithm for object-oriented design differencing. In *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[42] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, May 2004.

[43] Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2) :166–181, 2005.

Member-Godfrey, Michael W.

L'École Polytechnique se spécialise dans la formation d'ingénieurs et la recherche en ingénierie depuis 1873

École Polytechnique de Montréal

École affiliée à l'Université de Montréal

Campus de l'Université de Montréal
C.P. 6079, succ. Centre-ville
Montréal (Québec)
Canada H3C 3A7

www.polymtl.ca

ÉCOLE
POLYTECHNIQUE
MONTRÉAL