Université de Montréal

Application d'algorithmes de bio-informatique à la recherche de patrons de conception

par
Olivier Kaczor

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

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Application d'algorithmes de bio-informatique à la recherche de patrons de conception

présenté par :

Olivier Kaczor

a été évalué par un jury composé des personnes suivantes :

Julie Vachon
président-rapporteur

Yann-Gaël Guéhéneuc
directeur de recherche

Sylvie Hamel
codirecteur

Petko Valtchev
membre du jury

Mémoire accepté le

RÉSUMÉ

La maintenance de programmes orientés objets est une activité très coûteuse et la compréhension des programmes est essentielle pour les mainteneurs. L'identification de micro-architectures similaires aux motifs de conception dans un programme aide à comprendre les problèmes rencontrés lors de sa conception ainsi que les solutions apportées. Les techniques existantes utilisées pour la recherche de motifs de conception présentent toutes, cependant, un problème de performance.

Ce travail de recherche propose une solution à ce problème en adaptant des algorithmes efficaces de comparaisons et d'alignements de chaînes de caractères de bio-informatique. Des chaînes de caractères représentant les programmes et les patrons de conception sont d'abord construites et ensuite comparées à l'aide d'algorithmes de programmation dynamique, de simulation d'automates ou de vecteurs de bits.

Nous appliquons notre approche à plusieurs programmes de tailles différentes et comparons nos résultats avec ceux de deux outils utilisant la programmation par contraintes avec explications. Contrairement aux deux outils comparés, notre approche s'avère très efficace et permet une analyse relativement rapide de programmes de grandes tailles.

Mots clés : Maintenance, patrons de conception, analyse statique, vecteurs de bits.

ABSTRACT

Maintenance of object-oriented programs is a time- and ressource-consuming activity and program comprehension is essential to maintainers. Design patterns identification can help in designing, in understanding and in re-engineering programs. Most previous approaches of design patterns identification are limited because of their performance.

This research work offers a solution to the efficiency problem by using adaptation of string matching algorithms from bio-informatics. First, we generate string representations of the design motifs and of the programs and then apply bit-vector, automata simulation or dynamic programming algorithms to identify exacts and approximates occurrences of design motifs.

We apply our algorithm on several different size programs and compare its performance and results with two existing constraint-based approaches. Unlike the two compared approaches, ours is very efficient and allows big programs analysis.

Keywords: Maintenance, design patterns, static analysis, bit-vectors.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

Je tiens à remercier sincèrement mes deux directeurs de recherche Sylvie Hamel et Yann-Gaël Guéhéneuc pour l'aide compétente qu'ils m'ont apportée, pour leurs précieux conseils, pour leur démarche scientifique mais aussi pour leur bonne humeur, leur gentillesse et leur patience à mon égard. Ce fut un privilège de travailler avec eux !

Je tiens également à exprimer mes remerciements aux membres de l'équipe PTIDEJ avec qui j'ai pu discuter de très divers sujets fort intéressants reliés de près ou de loin à ma recherche.

Merci aussi à mes parents, Diane et Jean-Claude et à mon frère Joël pour leurs encouragements et soutien.

Je termine avec une pensée pour les membres de l'équipes GEODES qui ont contribué, à travers les nombreux séminaires ou discussions de couloir, à enrichir mes connaissances et à me donner le goût de la recherche.

Olivier Kaczor

mai 2006

CHAPITRE 1

INTRODUCTION

La maintenance de programmes orientés objets est une activité très coûteuse contribuant souvent à plus de 50% du coût total d'un programme [Kos04, LS81]. La complexité des programmes souvent combinée à l'absence de documentation récente contribue à la difficulté de leur compréhension. Le code source est souvent la seule source d'information à jour d'un programme. Le recouvrement de sa conception et de son architecture facilite l'identification des choix réalisés lors de sa conception et fait donc partie des tâches importantes d'un mainteneur. Les patrons de conception [GHJV94] sont des documents décrivant des solutions générales à des problèmes récurrents en orienté objets. L'identification des patrons de conception utilisés dans un programme contribue donc à réduire la complexité apparente de celui-ci en renseignant le mainteneur sur les problèmes rencontrés lors de la conception du programme et les solutions apportées. L'utilisation d'outils de détection permet de réduire le temps nécessaire à la réalisation de ces tâches longues et difficiles. Les outils doivent toutefois être efficaces pour pouvoir être utilisés régulièrement par les mainteneurs.

1.1 Patrons de conception

Les patrons de conception décrivent des solutions à des problèmes architecturaux récurrents en orienté objets. Ces solutions (appelées motifs de conception) sont indépendantes du contexte et du langage de programmation utilisé. Le terme patron de conception (*design pattern*) a été introduit dans le domaine de l'architec-

ture des bâtiments par Christopher Alexander : “Chaque patron décrit un problème qui se manifeste constamment dans notre environnement et la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière” [AIS77]. Les patrons de conception capitalisent un savoir précieux né du savoir d'experts [BMR+96]. Ils proposent des architectures de qualité, auto-documentent le code, encouragent la réutilisation de code avec des solutions adaptables, facilitent la maintenance et créent un vocabulaire améliorant la communication. Les patrons de conception les plus répandus sont ceux catalogués dans [GHJV94]. Chaque patron de ce catalogue est décrit en quatre sections :

1. Un nom unique.
2. Une description du problème à résoudre.
3. Une solution au problème.
4. Les conséquences de l'application de la solution.

Afin d'organiser les différents patrons de conception, Gamma et al. proposent une classification en trois familles selon leur utilisation (ci-dessous). Le tableau 1.1 présente les 23 patrons catalogués dans [GHJV94] selon leur famille.

- **créationnels** : patrons décrivant des processus de création d'objets ;
- **structuraux** : patrons s'occupant de la composition d'objets ou de classes ;
- **comportementaux** : patrons décrivant les interactions entre objets ou classes et la distribution de responsabilités.

Le patron de conception Composite, qui sert souvent d'exemple dans ce mémoire, est un patron structurel dont le but est de traiter uniformément un objet ou un groupe d'objets. La figure 1.1 présente la structure générale du motif de conception

Patrons de conception		
Créationnels	Structuraux	Comportementaux
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Facade	Iterator
	Proxy	Mediator
		Memento
		Flyweight
		Observer
		State
		Strategy
		Visitor

TAB. 1.1 – La classification des 23 patrons selon leur famille

Composite. Une classe abstraite *Component* déclare l'interface pour la composition d'objets. Les opérations peuvent être appliquées indifféremment sur un objet de type *Leaf* ou sur une composition d'objets de type *Composite*. La classe *Composite* définit par ailleurs, un comportement pour les composants ayant des enfants.

1.2 Détection de patrons de conception

Comme mentionné plus haut, l'identification de patrons de conception dans un programme contribue à réduire sa complexité apparente et aide à documenter le code source. La production d'un code source de qualité est une préoccupation importante chez les programmeurs. En orienté objets, la qualité du code dépend aussi bien de son efficacité à répondre à un problème que de sa structure générale et de ses relations avec les autres portions de code. Un code source de qualité facilite généralement la maintenance et l'ajout de fonctionnalités. L'identification de patrons de conception peut aussi servir à mesurer la qualité de conception d'un

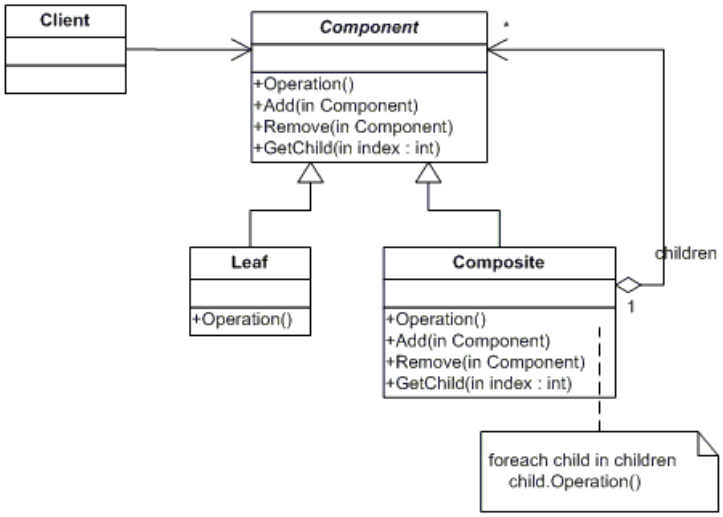


FIG. 1.1 – Structure du patron de conception Composite (illustration tirée de [GHJV94]

système.

Chaque patron de conception se caractérise par un ensemble de rôles avec des particularités propres (structurelles ou autres). La détection d'un motif de conception dans un programme consiste à associer aux différents rôles, des entités du programme respectant ces particularités. Un rôle d'un motif peut également être joué par une entité fantôme (*ghost entity*), une entité non présente dans le programme analysé mais connue par référence, comme par exemple les entités d'une bibliothèque externe.

Les patrons de conception ne sont pas toujours connus des développeurs et des problèmes peuvent avoir été résolus de manières différentes de celles proposées par les patrons de conception. Certaines contraintes imposées par l'architecture d'un programme obligent aussi parfois des modifications à la micro-architecture (structure d'un sous-ensemble des classes d'un programme orienté objets [GJ01a]) d'un motif de conception. La détection de motifs "dégradés" est donc aussi importante

et ces micro-architectures pourraient possiblement être améliorées en appliquant des corrections basées sur les motifs de conception.

Plusieurs techniques de détection ont été proposées comme la programmation logique [Wuy98,KP96,HHL02], la programmation par contraintes [QYW97,GJ01b], la transformation de graphes [SvG98, NSW+02, SK00] ou la logique floue [JZ97, NWW01, NMW04], etc. Ces approches ont toutes un problème de performance en temps. Les approches fondées sur les métriques [AFC98, GSZ04] sont prometteuses mais encore trop lentes pour l'analyse de programmes de grandes tailles ou pour être utilisées quotidiennement par des mainteneurs.

1.3 Algorithmes de bio-informatique

Les bio-informaticiens sont confrontés à des problèmes similaires à la recherche de motifs de conception dans des programmes. En effet, la comparaison et l'alignement de séquences d'ADN (Acide DésoxyriboNucléique) sont essentiels en biologie moléculaire. Le concept d'approximation est aussi extrêmement 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 [Gus97]. 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 la détection de formes approchées de motifs de conception dans un programme de grande taille. Les séquences d'ADN peuvent atteindre des longueurs de plusieurs millions (voire milliards) de caractères. Les algorithmes de comparaison de chaînes se doivent donc d'être performants. Plusieurs des algorithmes développés ou utilisés sont très efficaces en temps et pourraient

possiblement être adaptés pour résoudre le problème d'identification de motifs de conception dans des programmes de grandes tailles.

1.4 Notre approche

Pour remédier au problème de performance, nous tentons d'adapter les algorithmes efficaces utilisés en bio-informatique pour la recherche de micro-architectures similaires aux motifs de conception dans un programme. Comme ces algorithmes fonctionnent sur des chaînes de caractères, la première étape consiste à construire une représentation de la structure des programmes et des motifs de conception sous la forme de chaînes de caractères. Nous y parvenons en parcourant des graphes orientés représentant le programme et le motif. Cette solution est inspirée de techniques d'assemblage utilisées en bio-informatique.

Nous avons développé trois algorithmes de recherche de motifs de conception inspirés de ceux utilisés en bio-informatique. Le premier utilise la programmation dynamique, le deuxième la simulation d'automates et le dernier manipule des vecteurs de bits. Un outil utilisant ces algorithmes (`EPI`) a été développé et une étude de cas approfondie a été effectuée. L'approche avec vecteurs de bits s'avère plus efficace en temps que les approches existantes avec une qualité de résultats similaire.

1.5 Organisation du mémoire

Ce mémoire est divisé en 5 chapitres. La chapitre 2 passe en revue les différentes techniques utilisées pour l'identification d'occurrences de motifs de conception dans un programme. Les qualités que doit posséder un outil de détection y sont également discutées. Le chapitre 3 introduit les algorithmes utilisés en bio-

informatique ainsi que l'utilité qu'ils peuvent avoir en génie logiciel. Le chapitre 4 détaille les algorithmes d'identification développés inspirés de la bio-informatique. Enfin, le chapitre 5 présente un survol de l'implémentation en Java de notre outil ainsi qu'une étude de cas pour évaluer les résultats obtenus. Nous démontrons que notre approche permet une très bonne performance sans sacrifier la précision et le rappel. Finalement, nous concluons par un récapitulatif des points importants de la recherche, des contributions apportées par ce travail et des limites de l'approche. Certaines idées de travaux futurs sont également présentées.

CHAPITRE 2

ÉTAT DE L'ART

2.1 Principales techniques

Plusieurs techniques différentes ont été utilisées afin de détecter des occurrences de motifs de conception dans un programme. Dans cette section, les principales techniques sont expliquées et discutées.

2.1.1 Programmation logique

L'une des premières techniques utilisées pour identifier automatiquement les occurrences de patrons de conception dans un programme est la programmation logique [Wuy98], ou plus tard renommée la métaprogrammation déclarative [Bri00], qui consiste à manipuler du code source avec la programmation logique.

La programmation logique est une forme de programmation (simple et) déclarative où les instructions exécutées par l'ordinateur sont remplacées par des règles de logique mathématique. L'objectif étant de *considérer des formules comme des programmes et la construction de leurs preuves comme l'exécution de ces programmes*[1]. Ce paradigme de programmation est largement utilisé en intelligence artificielle, notamment pour simuler l'expertise humaine dans les systèmes experts ou pour faciliter le traitement du langage naturel. En programmation logique, les données (ensembles de faits et de règles) sont représentées par des clauses de Horn et la résolution d'un problème se fait à l'aide d'un algorithme d'unification. L'unification

[1] http ://www.irisa.fr/lande/ridoux/LPAZ/node10.html

est une opération binaire qui consiste à rendre deux termes égaux en attribuant des valeurs aux variables. Le système manipule donc les faits et les règles pour dériver de nouveaux faits.

La programmation logique a été largement utilisée pour concevoir des outils de détection automatique de patrons de conception [Wuy98, KP96, HHL02]. Sa simplicité, réduisant la tâche du développeur à décrire les connaissances et le problème à résoudre est son principal avantage.

En général, les outils de détection de patrons de conception basés sur la programmation logique construisent un ensemble de règles (ou prédicats) à partir des motifs de conception et une banque de faits à partir des artefacts du programme (classes, méthodes, relations, etc.). Un moteur d'inférence recherche ensuite les entités pouvant jouer un rôle dans un motif de conception. Par exemple, SOUL (**S**malltalk **O**pen **U**nification **L**anguage) [Wuy98] est un langage de programmation logique basé sur PROLOG et implémenté dans VisualWorks SMALLTALK. Les entités jouant un rôle dans un patron sont obtenus grâce aux algorithmes d'unification de PROLOG.

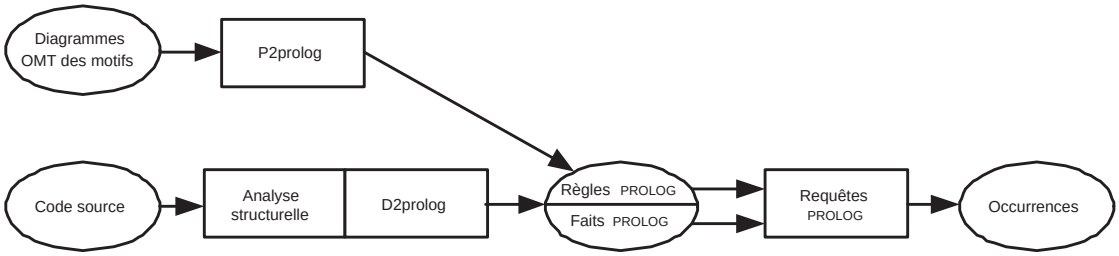


FIG. 2.1 – Architecture de PAT

L'outil PAT [KP96] est également basé sur PROLOG et permet de détecter des occurrences de patrons de conception dans des programmes C++. Un outil d'analyse structurelle (PARADIGM PLUS) est utilisé pour extraire des entêtes de fichiers

C++, l'information structurelle nécessaire à la détection d'occurrences de patrons. Les motifs de conception, issus d'une banque de patrons de conception, et l'information extraite du code source sont représentés avec la notation OMT et traduits en une représentation PROLOG grâce aux programmes P2PROLOG et D2PROLOG. Ils deviennent les faits et les règles que le moteur d'inférence de PROLOG manipulera pour rechercher les micro-architectures satisfaisant l'ensemble des règles issues d'un motif. La figure 2.1 présente l'architecture de PAT et la figure 2.2, la règle utilisée pour détecter les occurrences du patron Composite.

```
composite(Cpnt,Leaf,Compos) :-
   class(-,Cpnt),
   class(concrete,Leaf),
   class(concrete ,Compos),
   operation(-,-, Cpnt ,Op ,-,-,-),
   operation(-,-, Leaf ,Op ,-,-,-),
   operation(-,-, Compos,Op ,-,- ,-),
   operation(-,- ,Cpnt ,Add ,-,- ,-),
   operation(-, -, Cpnt ,Remove,-, -, -) ,
   operation(-,-,Cpnt,GetCh,-,-,-),
   operation(-,-, Compos,Add,-, -, -) ,
   operation(-,-,Compos,Remove,-,-,-),
   operation(- ,- ,Compos,GetCh ,-,-,-),
   inheritance(Cpnt ,Leaf),
   inheritance(Cpnt ,Compos),
   aggregation(Compos,exactlyone,Cpnt,many).
```

FIG. 2.2 – Exemple d'une règle dans PAT pour la détection du patron Composite

Deux inconvénients majeurs limitent le succès de la programmation logique dans le cadre de la détection de patrons de conception. Premièrement, les algorithmes d'unification ne permettent pas de détecter des micro-architectures similaires aux motifs. En effet, chacune des variantes d'un motif nécessite de nouvelles règles. Toutes les formes approchées doivent donc être prévues à l'avance.

Deuxièmement, l'efficacité de la technique est limitée notamment à cause de la complexité combinatoire à identifier toutes les entités pouvant jouer un rôle dans un motif, qui correspond au problème d'isomorphisme de sous-graphes. De plus, la lenteur d'exécution des algorithmes d'unification est reconnue et *essentiellement due à l'inadaptation du style logique vis-à-vis du modèle de Von-Neumann* [TL00].

2.1.2 Programmation par contraintes (avec explications)

Proche de la programmation logique, la programmation par contraintes permet de résoudre des problèmes combinatoires définis sur des domaines finis. Une contrainte est une relation logique entre variables prenant leur valeur dans des domaines donnés.

Un problème de satisfaction de contraintes (CSP) est défini par un ensemble de variables, et des ensembles de domaines et de contraintes pour ces variables. Une solution à un problème de satisfaction de contraintes est *une instanciation complète* qui satisfait toutes les contraintes, c'est-à-dire qu'à chaque variable est affectée une valeur par réduction de son domaine à un singleton. À chaque réduction du domaine d'une variable, le solveur examine s'il est possible de réduire ceux des autres variables impliquées avec elle dans une contrainte. On converge ainsi vers les solutions du problème. Si le domaine d'une variable se retrouve vide, il n'y a pas de solution (*contradiction*) et il faut remonter dans l'arbre de recherche (backtracking).

De nombreux problèmes dans différents domaines se prêtent bien à la programmation par contraintes. Elle est utilisée, par exemple, dans des logiciels d'optimisation pour la gestion du trafic aérien, de rotation de personnel, de simulation de circuits intégrés ou d'ordonnancement [Jus03].

Dans leurs travaux précurseurs, Quilici *et al.* [QYW97] traduisent la détection

de patrons de conception dans un programme en un problème de satisfaction de contraintes. À chaque classe d'un motif de conception est associée une variable dont le domaine est l'ensemble des classes du programme. Les relations entre les classes (association, agrégation, etc.) deviennent les contraintes du système.

PTIDEJ (Pattern Trace Identification, Detection and Enhancement in Java) [GJ01b, Gué05] est une suite d'outils dédiée à l'analyse et à la maintenance de programmes orientés objets permettant la détection d'occurrences de patrons de conception à l'aide de la programmation par contraintes avec explications. Les explications servent à justifier les solutions ou l'absence de solution en indiquant les contraintes satisfaites et celles ne pouvant l'être. Les contraintes amenant une contradiction (absence de solution) peuvent être relaxées et ainsi permettre de trouver des architectures similaires aux motifs de conception. Cette action est laissée au développeur car il est en mesure de choisir quelles contraintes peuvent être relaxées sans perdre le principe du patron de conception recherché. Cette action peut toutefois être automatisée en attribuant des poids a priori à chacune des contraintes et en laissant le système relaxer ces contraintes dans l'ordre défini par ces poids.

Variables :
Client
Component
Composite
Leaf

Contraintes :
association(client, component)
inheritance(component, composite)
inheritance(component, leaf)
composition(composite, component)

FIG. 2.3 – Exemple de système de contraintes pour la détection du Composite

Comme la programmation logique, la programmation par contraintes a l'avantage d'être simple pour le développeur. Toutefois, malgré un nombre important de travaux pour améliorer son efficacité (propagation de contraintes, algorithmes de filtrages), celle-ci demeure limitée (temps d'exécution et espace mémoire requis). Par contre, contrairement aux règles de programmation logique, les contraintes peuvent être relaxées ou retirées afin de permettre la détection de micro-architectures similaires aux motifs de conception.

2.1.3 Réduction de l'espace de recherche à l'aide de métriques

Rechercher les occurrences d'un motif de conception par l'approche de la "force brute" demande de tester toutes les configurations possibles des classes d'un programme pour les rôles du motif. Il faut donc tester, pour un motif de k classes dans un programme en contenant n, n^k configurations différentes. Les approches par programmation logique et par programmation par contraintes avec explications ne règlent pas ce problème de complexité combinatoire.

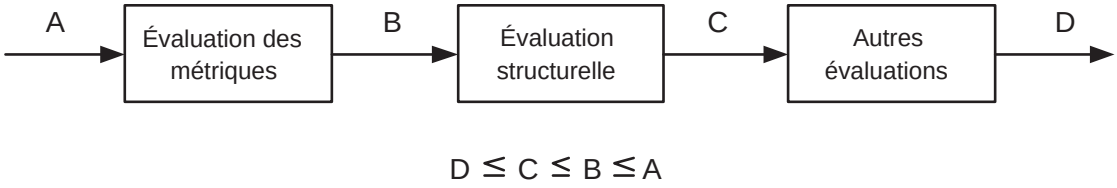


FIG. 2.4 – "Multi-stage filtering"

Afin de réduire l'espace de recherche et ainsi améliorer l'efficacité des outils de détection, Antoniol *et al.* [AFC98] proposent d'utiliser les métriques. Les métriques sont des mesures de certaines propriétés d'une partie d'un logiciel ou de sa spécification. Antoniol *et al.* proposent de séparer la détection en trois processus d'évaluation (voir Figure 2.4) : l'évaluation des métriques, l'évaluation structurelle

et l'évaluation des délégations. L'évaluation des métriques consiste à réduire le nombre de classes possibles pour chacun des rôles d'un motif. Lors de cette étape d'évaluation, certaines métriques des classes du système étudié (nombre d'attributs, d'opérations ou d'associations) sont comparées avec les valeurs espérées pour les rôles du motif. Ces valeurs espérées proviennent de la description théorique des patrons de conception. Cependant, les occurrences de motifs de conception ne reflètent pas toujours exactement la théorie.

Récemment, Guéhéneuc *et al.* [GSZ04] ont combiné la programmation par contraintes avec explications et l'étude empirique de métriques pour identifier efficacement des micro-architectures similaires aux motifs de conception. Contrairement aux travaux d'Antoniol *et al.*, ils ne se servent pas de la description théorique des patrons de conception mais d'une "empreinte" de chacun des rôles d'un motif obtenue en étudiant les attributs internes de classes (taille, filiation, cohésion, couplage, etc.) jouant ce rôle. Ils utilisent les métriques pour réduire la taille des domaines des variables utilisées par leur solveur de contraintes.

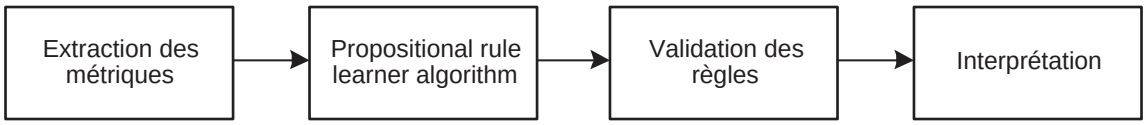


FIG. 2.5 – Création des empreintes des rôles des motifs de conception

Les empreintes sont construites (voir figure 2.5) à partir de données prises dans P-MART [GSZ04], une base de micro-architectures similaires aux motifs de conception recueillies manuellement à partir de onze programmes de différentes tailles. Les métriques calculées sur ces micro-architectures sont données à JRIP [WF99], un *propositional rule learner algorithm* qui en déduit un ensemble de règles pour chacun des rôles d'un motif de conception. Par exemple, voici une règle de trois

empreintes générée pour le rôle de feuille du patron Composite où NMI représente son nombre de méthodes héritées, DIT sa profondeur dans l'arbre d'héritage, NMO son nombre de méthodes surchargées et NM son nombre de méthodes :

(NMI >= 26 et DIT >= 5) ou (NMI >= 25 et NMO <= 2) ou (DIT >= 3 et NM <= 12).

Une classe peut jouer le rôle de feuille du patron Composite si et seulement si son empreinte correspond à celle de la règle. Par exemple, son nombre de méthodes héritées (NMI) doit être plus grand ou égal à 26 et sa profondeur dans l'arbre d'héritage (DIT) plus grande ou égale à 5. Dans un programme de taille moyenne utilisé pour valider l'approche, ces empreintes ont permis une réduction de l'espace de recherche allant de 69.00% à 89.15%.

L'étude des attributs internes des classes d'un programme ne permet pas, à elle seule, d'identifier des occurrences de motifs de conception. Deux rôles peuvent exiger les mêmes attributs internes et deux classes peuvent avoir les mêmes valeurs pour un ensemble de métriques. Par contre, combinée à une autre approche comme la programmation par contraintes, l'évaluation de métriques permet d'améliorer la performance de détection en réduisant considérablement l'espace de recherche. L'évaluation des métriques améliore également la précision des résultats en enlevant les occurrences avec des classes ne pouvant pas jouer de rôle dans le motif de conception d'après les valeurs empiriques des métriques. L'efficacité de l'outil développé dans [GSZ04] souffre toutefois de la mauvaise performance de la programmation par contraintes avec explications.

2.1.4 Transformation de graphes

Les graphes sont souvent utilisés pour représenter diverses données. De nombreux algorithmes efficaces de manipulations et transformations de graphes permettent de les utiliser pour résoudre plusieurs types de problèmes tels les problèmes combinatoires. Les motifs de conception et les programmes peuvent facilement être représentés par des graphes. Les diagrammes UML le sont d'ailleurs souvent. La recherche de motifs de conception peut alors être étudiée comme un problème de recherche de sous-graphes dans un graphe. Toutefois, ce problème d'isomorphisme de sous-graphes est, dans le cas général, NP-complet [Epp95].

Plusieurs travaux d'identification de motifs de conception [SvG98, NSW+02, SK00] représentent les motifs et les programmes comme des graphes orientés. Les classes, méthodes et attributs représentent généralement les nœuds et les relations inter et intra-classes, les arcs du graphe. Dans leurs travaux, Seemann *et al.* parcourent le code source d'un programme afin de construire un graphe préliminaire contenant ses entités (classes, interfaces, etc.) et ses relations élémentaires (héritage, instanciation, etc.). Ils utilisent ensuite des règles de transformation de graphes pour obtenir de nouveaux nœuds (motifs) et arcs (associations, agrégations, délégations, etc.). Une règle de transformation est composée de deux parties. La partie de gauche représente un sous-graphe à identifer dans le graphe tandis que la partie de droite représente la modification à faire aux sous-graphes trouvés. Par exemple, un arc agrégation entre deux classes est ajouté si et seulement si des arcs association et création existent entre ces deux classes :

$$c1 \text{ agrège } c2 \leftrightarrow c1 \text{ est associée à } c2 \wedge c1 \text{ crée } c2$$

L'identification d'occurrences du motif de conception Composite est réalisée

grâce la transformation suivante :

$$\textit{Soit SUB(C)} = \{D | \textit{D sous-classe transitive de C}\}.$$
$$\textit{Si } \exists\ D \in \textit{SUB(C)} | \textit{D agrège C} \wedge \textit{C délègue à D} \Rightarrow \textit{composite(C,D,A)}$$
$$\textit{où D = SUB(C) et A = SUB(C) - SUB(D).}$$

Les occurrences du motif de conception Composite se retrouvent alors dans le graphe sous forme de nœuds.

Niere *et al.* [NSW+02] ont également développé un outil de détection basé sur les transformations de graphes. Comme Seemann *et al.* [SvG98], les occurrences de motifs de conception sont représentées par des nœuds ajoutés à un graphe représentant un programme. Leur graphe préliminaire est un graphe syntaxique abstrait (ASG) généré par le parseur de code source Java JavaCC. Afin de réduire le temps nécessaire à l'obtention des premiers résultats et d'éviter les longs calculs inutiles en cas d'erreur au début du processus de détection, leur algorithme de détection alterne entre deux modes d'analyse, ascendant et descendant, et permet une interaction avec le mainteneur. Les annotations au graphe servent de point de départ à la recherche de sous-graphes.

L'analyse débute en mode ascendant avec les règles dépendant seulement d'objets du ASG. Ceci permet d'éviter les nombreuses contradictions possibles avec les approches descendantes dues à un manque d'information en début d'analyse. Les règles dépendant de règles activées sont déclenchées. Si ces règles ne peuvent pas être traitées immédiatement par manque d'information, elles sont ajoutées à une liste d'attente pour le mode descendant qui débute lorsque le mode ascendant ne peut plus progresser. Dans le mode descendant, les règles en attente déclenchent les règles dont elles dépendent. L'analyse retourne au mode ascendant lorsque la liste

d'attente pour le mode descendant ne contient que des règles ne pouvant s'activer. L'analyse se termine lorsque la liste d'attente du mode ascendant est vide ou que les règles en attente ne peuvent être activées.

Par exemple, la figure 2.6 (tirée de [NSW+02]) montre l'exécution de l'algorithme pour la recherche d'occurrences de motifs de conception Composite. L'ovale noir représente un nœud ajouté par le mode ascendant tandis que les ovales gris représentent les nœuds ajoutés par le mode descendant. La règle *Généralisation* déclenche la règle *Composite* en mode ascendant qui ne peut s'activer car elle requiert les annotations *association* et *1N_Délégation*. Une fois en mode descendant, elle déclenche les règles de délégation et d'association.

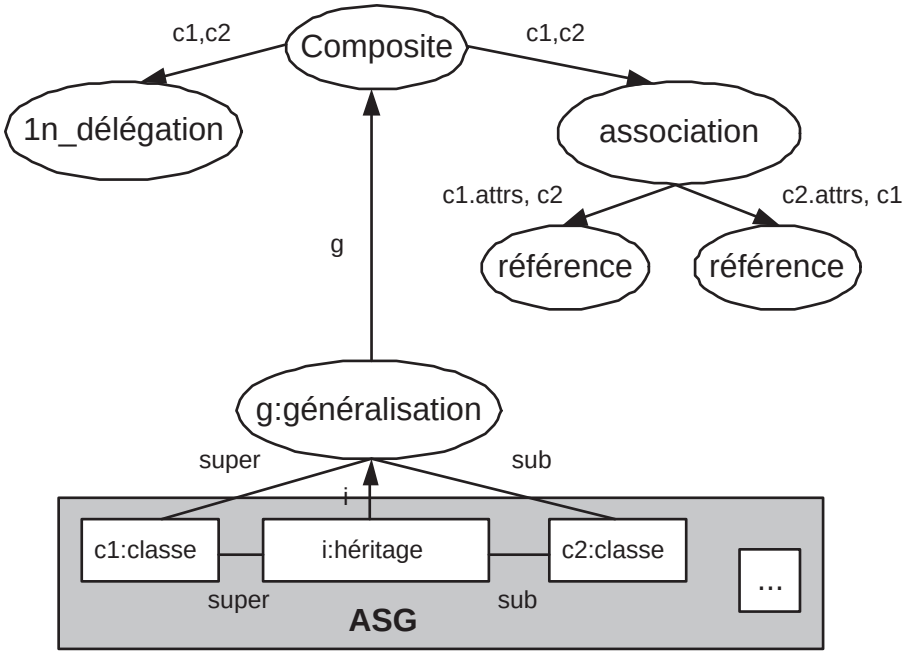


FIG. 2.6 – Exécution de l'algorithme pour la recherche de motifs de conception Composite

Chacune des annotations représente un résultat intermédiaire et les règles de transformation peuvent être modifiées en tout temps par le mainteneur. Des anno-

tations peuvent également être ajoutées manuellement par celui-ci.

L'algorithme d'inférence derrière les transformations de graphes ressemble beaucoup aux algorithmes de programmation logique. Les graphes sont cependant très pertinents pour représenter les motifs de conception et les modèles de programmes car la transformation est très simple et leur ressemblance avec les diagrammes UML facilite leur compréhension. Toutefois l'identification de formes similaires aux motifs de conception n'est pas possible sans modifications importantes du moteur d'inférence. Par exemple, Niere *et al.* ont récemment modifié leur approche pour traiter la logique floue.

2.1.5 Logique floue

La programmation logique a beaucoup été utilisée pour la recherche d'occurrences de motifs de conception notamment à cause de sa facilité à représenter le problème. Cependant, elle ne permet pas la détection de micro-architectures similaires aux motifs de conception puisque les prédicats sont soit vrais soit faux. La logique floue est une généralisation de la logique classique qui permet différents degrés de satisfaction d'une condition. Le raisonnement en logique floue est largement utilisé en intelligence artificielle et représente davantage le raisonnement humain. Cette technique est d'ailleurs largement utilisée en robotique et en médecine. La logique floue a également été utilisée pour la détection d'occurrences de motifs de conception [JZ97,NWW01,NMW04]. Jahnke *et al.* ont développé des réseaux génériques de raisonnements flous (GFRN) [JSZ97] pour la rétro-ingénierie de bases de données relationnelles. Leur moteur d'inférence effectue des recherches en permettant une information incomplète ou inconsistante en transformant les GFRN en réseaux de Pétri flous (FPN). Ils ont également utilisé leur approche pour détec-

ter des micro-architectures similaires aux motifs de conception et les corriger selon les spécifications des patrons de conception. Toutefois, aucune étude ne montre la précision de leurs résultats ou les temps d'exécution sur de gros programmes.

Niere *et al.* ont modifié le moteur d'inférence de Jahkne *et al.* afin de détecter les micro-architectures similaires aux motifs de conception en remplaçant les règles des GFRN basées sur l'algèbre relationnelle par des règles de transformation de graphes floues. À chacune des règles est associée une valeur de crédibilité estimée par le mainteneur. Pour chaque annotation créée par une règle de transformation, une valeur de confiance est calculée selon le niveau de respect des conditions de la règle. Les résultats peuvent donc être triés et ainsi permettre au mainteneur d'examiner seulement les occurrences qu'ils jugent problématiques ou intéressantes. Toutefois, établir les premières valeurs de crédibilité peut poser problème.

Le calcul des valeurs de confiance se fait lorsque le moteur d'inférence a terminé ses opérations ou lorsque celui-ci est interrompu par le mainteneur. Elles sont calculées à l'aide d'un réseau de Petri flou obtenu à partir du graphe annoté. À chaque annotation et lien du graphe correspondent, respectivement, une place et une transition dans le réseau de Petri.

Soit un réseau de Petri flou $RPF = (P, T, F, cr, th, uac, cac, ac)$ *avec*

P *un ensemble fini de places*

T *un ensemble fini de transitions*

$F \subset (P \times T) \cup (T \times P)$ *un ensemble d'arcs*

$cr : P \rightarrow [0..1]$ *une valeur de crédibilité*

$th : P \rightarrow [0..1]$ *un seuil*

$uac : P \rightarrow [0..1]$ *une valeur de confiance définie par le mainteneur*

$cac : P \rightarrow [0..1]$ *une valeur de confiance calculée*

$$ac : P \rightarrow [0..1] \text{ avec } \forall p \in P,\ ac(p) = \begin{cases} uac(p) & \textit{si définie} \\ cac(p) & \textit{sinon} \end{cases}$$

Soient les fonctions $ftt : T \rightarrow [0...1]$ et m_t(p) définies comme

$ftt(t) = max(\{ac(p) | (p,t) \in F\})$ *et*

$m_t(p) = min(\{ftt(t) | (t,p) \in F\})$,

la valeur de confiance calculée *cac* pour une place est $\forall$p $\in$ P,

$$cac(p) = \begin{cases} cr(p) & \neg\exists t \in T \textit{ avec } (t,p) \in F \\ min(\{cr(p), m_t(p)\}) & \textit{sinon} \end{cases}$$

Le seuil pourrait servir à éviter la création d'annotations avec une valeur de confiance trop basse en ne permettant pas les annotations avec une valeur de confiance plus petite que leur seuil d'être utilisées pour créer de nouvelles annotations. Le seuil n'est toutefois pas utilisé dans la définition donnée plus haut.

L'approche de la logique floue est intéressante car elle permet la détection de micro-architectures similaires aux motifs de conception sans définir chacune des variantes possibles. De plus, les résultats peuvent être triés selon leur valeur de confiance. Par contre, comprendre comment les valeurs de confiance sont calculées peut s'avérer difficile et ainsi limiter leur utilité.

2.1.6 Synthèse

D'autres techniques d'identification d'occurrences de motifs de conception dans un programme (requêtes sur des modèles de programmes [KSRP99], études linguistiques et stylistiques [Bac98]) ont été étudiées mais les principales sont celles présentées dans ce chapitre. Le tableau 2.1 résume les caractéristiques des ces différentes techniques.

	PL	PPC	Transformations de graphes	Logique floue
Identification automatique d'occurrences approchées	non	oui	non	oui
Performance	mauvaise	mauvaise	mauvaise	N/A

TAB. 2.1 – Caractéristiques des différentes techniques d'identification

Une technique d'identification de motifs de conception doit permettre l'identification d'occurrences similaires aux motifs de conception dans de gros programmes en un temps raisonnable. Aucune des techniques présentées ne répond parfaitement à ces besoins. La performance de ces techniques peut toutefois être améliorée en réduisant l'espace de recherche par l'étude des métriques.

La précision des résultats est également importante dans un algorithme de recherche. L'étude de la structure d'un programme et de son code source seulement amène de nombreux faux-positifs (fausses détections) et n'assure pas la détection de motifs de conception autres que structuraux. Combinée avec une technique d'identification, l'étude empiriques des métriques permet d'améliorer nettement la précision. L'étude du comportement d'un programme grâce à l'enregistrement d'informations lors de son exécution sous forme de traces combinée à l'étude du code source semble également être une idée prometteuse pour améliorer la préci-

sion [Wen03, HZCD05, PC00].

Notre outil de détection doit donc être rapide, permettre l'identification d'occurrences similaires aux motifs de conception et pourrait utiliser l'étude empirique des métriques pour obtenir une bonne précision.

CHAPITRE 3

LA BIO-INFORMATIQUE AU SERVICE DU GÉNIE LOGICIEL

Les différentes techniques utilisées jusqu'à maintenant pour la détection de motifs de conception 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 (plusieurs centaines de classes). Par exemple, PTIDEJ avec la programmation par contraintes nécessite plusieurs heures de calculs pour analyser QuickUML, un programme d'environ 400 classes.

Les bio-informaticiens sont confrontés à des problèmes similaires à la recherche de motifs de conception dans des programmes de grande taille. En effet, la comparaison et l'alignement de séquences d'ADN (Acide DésoxyriboNucléique) sont essentiels en biologie moléculaire. Le concept d'approximation est aussi extrêmement 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 [Gus97]. 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 motifs de conception dans un grand programme.

Une séquence d'ADN est formée d'une succession de nucléotides (caractères). Les quatre nucléotides possibles sont l'adénine (A), la cytosine (C), la guanine (G) et la thymine (T). Les séquences d'ADN peuvent atteindre des longueurs de plusieurs millions (voire milliards) de caractères. Les algorithmes de comparaison de chaînes se doivent donc d'être performants. Plusieurs des algorithmes développés

ou utilisés sont très efficaces en temps et pourraient possiblement être adaptés pour résoudre le problème d'identification de motifs de conception dans des programmes de grandes tailles.

Ce chapitre introduit différentes techniques de recherche exacte et approximative en bio-informatique ainsi qu'une méthode pour transformer les motifs de conception et programmes en chaînes de caractères inspirée des méthodes d'assemblage en bio-informatique.

3.1 Algorithmes de recherche en bio-informatique

3.1.1 Recherche exacte

Étant donné un mot P de longueur p et un texte T de longueur t sur un alphabet commun Σ, le problème de recherche exacte consiste à trouver toutes les occurrences de P dans T. L'idée la plus simple consiste à parcourir le texte avec une fenêtre glissante de longueur p, à comparer les caractères de la fenêtre à ceux de P et à avancer la fenêtre d'un caractère lorsque deux caractères ne sont pas égaux. Cet algorithme naïf ne s'avère toutefois pas très efficace et nécessite, dans le pire des cas, $p \times t$ comparaisons.

Approches avec comparaisons de caractères

Plusieurs types d'algorithmes ont été proposés pour optimiser le problème de recherche exacte. Le premier algorithme à résoudre le problème en temps linéaire est celui de Pratt et Morris [MP70]. Cet algorithme ressemble beaucoup à celui de la force brute mais exploite la réussite partielle avant une erreur afin d'optimiser le décalage de la fenêtre.

Soient une tentative à la position j dans T $(0 < j < t - p)$ et u le mot formé

par la réussite partielle avant l'inégalité à la position $P[i]$ et $T[i+j]$ $(0 < i \leq p)$. Il est possible qu'un préfixe v de u soit égal à un suffixe de u. On appelle ce mot, à la fois préfixe et suffixe, bord de u. Il est donc possible d'effectuer un décalage (voir figure 3.1) de manière à ce que les comparaisons reprennent entre le caractère qui suit le bord maximal de u dans P et le caractère à la position $i+j$ dans T.

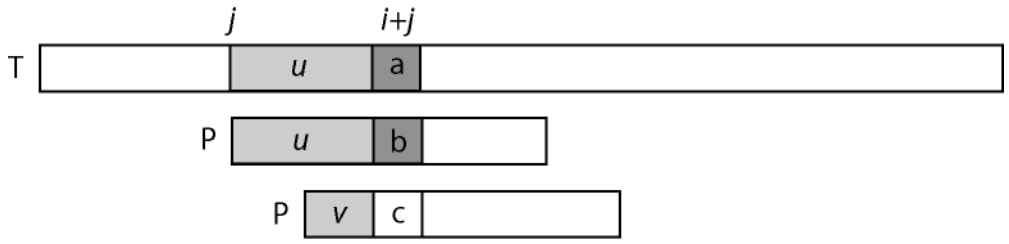


FIG. 3.1 – Décalage de l'algorithme Pratt–Morris

Comme les décalages ne dépendent que de u (donc de P), ils peuvent être précalculés. La complexité de cet algorithme est en $O(p+t)$.

Plusieurs autres algorithmes différents ont été développés avec fenêtres glissantes. Le plus utilisé est celui de BOYER ET MOORE [BM77] qui possède comme particularité d'effectuer les comparaisons de droite à gauche. Les décalages dépendent aussi du caractère dans T provoquant la différence. La position ou l'absence de ce caractère dans P permet de plus ou moins grands décalages. L'opération de base de ces algorithmes est la comparaison de caractères.

Approches numériques

D'autres approches au problème de recherche exacte se basent sur des opérations sur des bits ou sur l'arithmétique. Les algorithmes SHIFT-OR [BYG92] et SHIFT-AND [WM92b] manipulent des vecteurs de bits et tirent leur efficacité de la capacité des langages de programmation à manipuler les mots machines. Soit M, un tableau de vecteurs binaires de taille p. Trouver les occurrences de P dans T

avec l'algorithme Shift-Or consiste à remplir ce tableau M définit pour $1 \leq i \leq p$ comme :

$$M_j[i] = \begin{cases} 0 & si\ P[1..i] = T[j-i+1..j] \\ 1 & sinon \end{cases}$$

$M_{j+1}[i] = 0$ si et seulement si $M_j[i-1] = 0$ et $T[j+1] = p[i]$. La transition entre M_j et M_{j+1} peut être calculée très efficacement à l'aide d'opérations binaires sur M_j et des vecteurs de bits obtenus préalablement. On définit pour chaque caractère x de P un vecteur S_x, indiquant les positions de x dans P comme :

$$S_x[i] = \begin{cases} 0 & si\ x = p_i \\ 1 & sinon \end{cases}$$

L'obtention de M_{j+1} se réduit alors à deux opérations, un décalage vers la gauche de une position en mettant un 1 dans le bit de poids faible ($\ll$) et un et logique ($\wedge$) :

$$M_{j+1} = (M_j \ll 1) \ \wedge \ S_{t_{j+1}}$$

Le premier vecteur du tableau représente les positions dans T où se trouve le premier caractère de P, le deuxième représente les positions dans T où se trouve le deuxième caractère de P précédé du premier caractère de P et ainsi de suite. Construire le dernier vecteur de M résout donc la recherche exacte. En effet, un zéro dans M_p indique la fin d'une occurrence dans T. Le tableau 3.1 est le tableau M pour la recherche du patron ABC dans le texte $ABEFAGDABCFABCE$. Les zéros aux positions 10 et 14 de M_3 indiquent que des occurrences se trouvent aux positions $10 - 3 + 1 = 8$ et $14 - 3 + 1 = 12$ de T. Un avantage de ces

algorithmes numériques, outre leur efficacité, est qu'ils sont facilement adaptables pour la recherche approximative. Une version de Shift-And est d'ailleurs utilisée dans le programme AGREP [WM92a] pour la recherche approximative de mots dans un texte.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		A	B	E	F	A	G	D	A	B	C	F	A	B	C	E
1	A	0	1	1	1	0	1	1	0	1	1	1	0	1	1	1
2	B	1	0	1	1	1	1	1	1	0	1	1	1	0	1	1
3	C	1	1	1	1	1	1	1	1	1	**0**	1	1	1	**0**	1

TAB. 3.1 – Tableau M pour la recherche exacte de *ABC* dans *ABEFAGDABCFABCE*

Certains algorithmes transforment le problème de recherche exacte en un problème arithmétique sur des nombres entiers. L'algorithme de KARP–RABIN [KR87] exploite cette idée en donnant des valeurs numériques à des chaînes de caractères à l'aide d'une fonction de hachage. Pour une fonction de hachage quelconque H, la recherche d'occurrences consiste à faire $t - p$ comparaisons entre $H(T[i..i + p])$ et $H(P[1..m])$ pour $1 \leq i \leq t - p$. Comme $H(P[1..m])$ est constant, le nombre d'évaluations de H à effectuer est $1 + t - p$.

Approches avec automates

Une autre approche utilisée pour la recherche d'occurrences d'un patron dans un texte consiste à construire un automate fini. Un automate peut être considéré comme un graphe orienté dont les sommets sont appelés états et les arêtes, transitions. Plus précisément, un automate fini est un quintuplet (S, Σ, T, q_0, A) où :

- Σ est un alphabet fini ;

- S est un ensemble fini d'états ;

- T est une fonction de transition ;

- $q_0 \in S$ est l'état initial ;

- $A \subseteq S$ est l'ensemble d'états finaux.

L'automate lit successivement les caractères du texte passant d'état(s) en état(s) et signale les positions du texte où une occurrence du patron se termine (états finaux). Un avantage des automates est qu'ils peuvent reconnaître tous les langages rationnels (Théorème de Kleene). Un automate permet donc aussi la recherche de patrons définis par des expressions régulières. La figure 3.2 représente un automate fini déterministe pour la recherche du patron *abba*. Lors de la recherche de *abba* dans le texte *abaabbabba*, l'automate passe successivement dans les états 0, 1, 2, 1, 2, 3, 4, 2, 3 et 4. Comme l'état final (4) a été atteint deux fois, le texte contient deux occurrences de *abba* (qui se terminent aux positions 7 et 10 du texte).

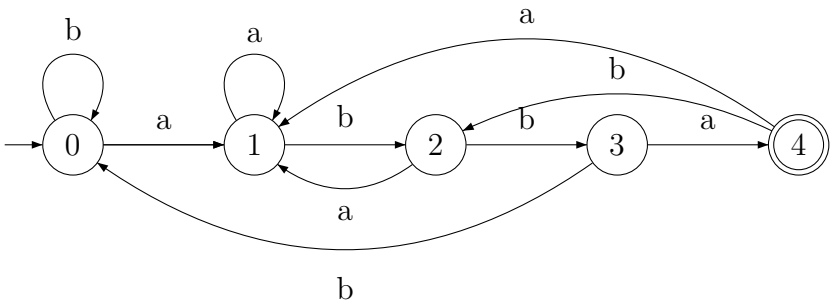


FIG. 3.2 – Automate fini déterministe pour la recherche du patron *abba*

3.1.2 Recherche approximative

La recherche approximative consiste à trouver toutes les occurrences d'un patron $P = \{p_1 p_2..p_m\}$ dans un texte $T = \{t_1 t_2..t_n\}$ à k erreurs près.

Plusieurs définitions de distance ont été proposées pour mesurer la différence entre deux chaînes de caractères. La distance de Hamming, par exemple, correspond

au nombre de caractères qui diffèrent entre les séquences. La distance de Hamming entre la chaîne *acbba* et la chaîne *abbba* est 1. La distance de Levenshtein (ou *edit distance*) est souvent utilisée en bio-informatique. Elle correspond au nombre minimum d'insertions, suppressions et substitutions nécessaires pour transformer une chaîne en une autre. Par exemple, la distance de Levenshtein entre *acbbab* et *abbba* est 2 car une façon optimale de passer de la chaîne 1 à la chaîne 2 est de substituer le c pour un b et de supprimer le dernier b.

Le problème de recherche approximative peut donc être résolu en trouvant tous les indices i de T où $d(T[i..j], P) \leq k$ avec $d(x, y)$ la distance entre les chaînes x et y. L'approche la plus utilisée pour résoudre ce problème est la programmation dynamique. La programmation dynamique est une méthode de résolution de problèmes d'optimisation satisfaisant le principe d'optimalité de Bellman. Une solution optimale d'un problème est obtenue à partir des solutions optimales de tous les sous-problèmes. La recherche approximative par programmation dynamique se base sur trois composantes essentielles, une relation de récurrence, une table de calcul et un chemin. L'approche descendante pour évaluer une récurrence est très simple à programmer mais n'est pas du tout efficace. Beaucoup de calculs effectués sont redondants. L'évaluation d'une relation de récurrence à deux variables requiert seulement $(n+1)(m+1)$ évaluations distinctes de la fonction. L'approche ascendante de la programmation dynamique effectue une seule fois chaque calcul et est ainsi beaucoup plus efficace.

L'approche consiste à remplir une table D de taille $m \times n$ dont les cases $D(i, j)$ prennent les valeurs $d(h..i, k..j)$ où $0 \leq h \leq i \leq n$ et $0 \leq k \leq j \leq m$. La valeur d'une case dépend des valeurs précédemment obtenues. La première ligne de la table $D(0, i)$ est initialisée à 0 pour permettre aux occurrences de débuter à

n'importe quelle position de T. La première colonne $D(j, 0)$ est initialisée à j et $D(i, j) = \min\{D(i-1, j-1) + w(P[i], T[j]),\ D(i-1, j) + 1,\ D(i, j-1) + 1\}$ où

$$w(i, j) = \begin{cases} 0 & si\ T[i] = P[j] \\ 1 & sinon \end{cases}$$

Le tableau 3.2 représente la table D pour la recherche approximative du mot *abcd* dans le texte *dabccdabcda* avec la distance de Levenshtein.

		d	a	b	c	c	d	a	b	c	d	a
	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	0	1	1	1	1	0	1	1	1	0
b	2	2	1	0	1	2	2	1	0	1	2	1
c	3	3	2	1	0	1	2	2	1	0	1	2
d	4	3	3	2	1	**1**	1	2	2	1	**0**	1

TAB. 3.2 – Table D pour la recherche approximative du patron *abcd* dans le texte *dabccdabcda* avec la distance de Levenshtein

La dernière ligne de la table indique la distance de l'alignement local optimal de P à chaque position de T. Par exemple, $D(10, 4) = 0$ signifie qu'un alignement local de distance 0 se trouve à $T[i..10]$ où i est la position où termine le chemin optimal débutant à la case $D(10, 4)$. Ce chemin (montré par les flèches dans le tableau 3.2) correspond aux valeurs utilisées pour calculer $D(10, 4)$. Le tableau 3.2 indique aussi qu'une occurrence du patron se trouve à $T[1..5]$ avec une distance de 1.

L'approche par programmation dynamique permet facilement de donner un poids différent aux substitutions dépendamment du caractère enlevé et ajouté. Elle permet aussi de spécifier un coût pour la présence de trou. Un trou est un espace permis dans une chaîne pour permettre un alignement. Par exemple, il est possible d'aligner *caaatcac* avec *ctcaaaac* en permettant des trous de longueur 3

$\binom{caaatca---c}{c---tcaaaac}$. Ces ajouts sont possibles en modifiant la relation de récurrence.

Comme mentionné dans la section 3.1.1, il existe également des approches numériques pour résoudre le problème de recherche approximative. Les automates sont aussi fréquemment utilisés pour ce type de problèmes [Ukk85, Hol97]. La figure 3.3 représente l'automate fini non déterministe pour la recherche approximative de *abba* avec une distance de Hamming maximale de 2. Lorsque l'état final (m, i) est atteint, cela indique une occurrence du mot à distance de Hamming i d'un sous-mot du texte. Plus l'automate permet d'erreurs, plus il sera grand. Le nombre nécessaire d'états est donné par la formule $(k+1)(m+1-\frac{k}{2})$. La construction de tels automates peut être coûteuse [Mel95]. Une solution intéressante consiste à simuler l'automate sans le construire.

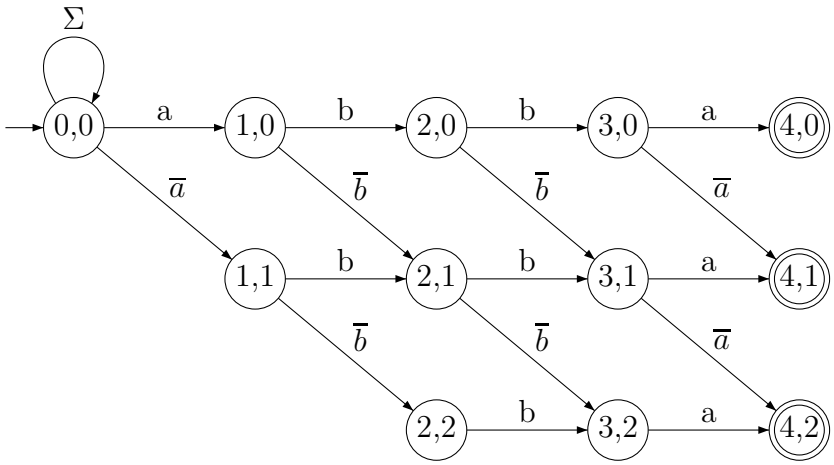


FIG. 3.3 – Automate fini non déterministe pour la recherche approximative de *abba* avec une distance de Hamming maximale de 2

3.2 Construction des chaînes de caractères d'un programme et d'un motif

Les algorithmes de bio-informatique présentés opèrent tous sur des chaînes de caractères. Les motifs de conception et les programmes dans lesquels nous voulons faire les recherches n'existent pas sous cette forme. Pour les utiliser pour la recherche de motifs, il faut convertir les programmes et motifs en chaînes de caractères.

Comme la complexité des algorithmes présentés dépend de la longueur des chaînes utilisées, il faut obtenir des représentations les plus petites possibles. Les programmes et les motifs de conception peuvent être représentés par leurs entités et les relations entre ces entités. Par exemple, un programme peut être représenté par un ensemble de triplets entité–relation–entité. Il suffit donc de trouver l'agencement optimal de tous ces triplets afin d'obtenir la plus petite chaîne possible.

3.2.1 Algorithmes de bio-informatique

Le problème d'assemblage de séquences lors de séquençages aléatoires (shotgun sequencing) en bio-informatique correspond au même problème d'agencement. Le but étant de déduire la chaîne ADN d'origine à partir d'un ensemble de fragments séquencés [KM95].

Définition : Étant donné un ensemble de k chaînes $P = \{s_1, s_2, ..., s_k\}$, une super-chaîne, de l'ensemble P est une chaîne contenant toutes les chaînes s_i.

Étant donné P un ensemble de fragments séquencés, la plus courte super-chaîne de P est une bonne candidate pour la chaîne ADN d'origine. Le problème de trouver la chaîne ADN d'origine peut alors être abordé comme celui de trouver la plus courte super-chaîne.

Par exemple, soit $P = \{ACCGT, CGTGC, TTAC, TACCGT\}$. La plus courte super-chaîne possible de P est $TTACCGTGC$. Trouver la plus courte super-chaîne est toutefois un problème NP-difficile [GJ79]. L'heuristique gloutonne est l'approche standard utilisée pour trouver une super-chaîne approchant la super-chaîne optimale. Cette super-chaîne est obtenue en joignant successivement les deux chaînes de l'ensemble possédant le plus grand chevauchement. Par exemple, étant donné un ensemble de chaînes $P = \{GTCAT, CTAGT, GCTA\}$, cet ensemble devient $P = \{GCTAGT, GTCAT\}$ et finalement $P = \{GCTAGTCAT\}$. Lorsque l'ensemble ne contient plus qu'un seul élément, cet élément est une super-chaîne ($GCTAGTCAT$). Il arrive que l'ensemble final contienne plus qu'une séquence sans chevauchement. Ces séquences peuvent être concaténées. Ceci n'est toutefois pas acceptable en bioinformatique où un séquençage supplémentaire est nécessaire. Avec cette approche, la super-chaîne obtenue ne dépasse jamais 4 fois la longueur de la super-chaîne optimale [KPS94].

Une autre approche consiste à construire un graphe orienté où les sommets représentent les k chaînes s_1 à s_k de P. Des arcs sont créés entre chaque paire de sommets (s_i, s_j) avec comme longueur, la longueur du plus long préfixe de s_j qui s'aligne avec un suffixe de s_i. Il suffit ensuite de trouver le chemin le plus court qui visite tout les sommets du graphe. Ce problème correspond au problème du postier chinois. Soit $P = \{ATC, CCA, CAG, TCC, AGT\}$, la figure 3.4 représente le graphe orienté et le chemin le plus court parcourant ce graphe permettant d'obtenir la super-chaîne de P, $ATCCAGT$ (tiré de bioalgorithms.info).

Une autre approche est utilisée pour un cas particulier du problème de super-chaîne, lors de séquençages par hybridation [DLBC89]. Lors de ce type de séquençage, les fragments ont tous une longueur fixe k et deux fragments consécutifs se

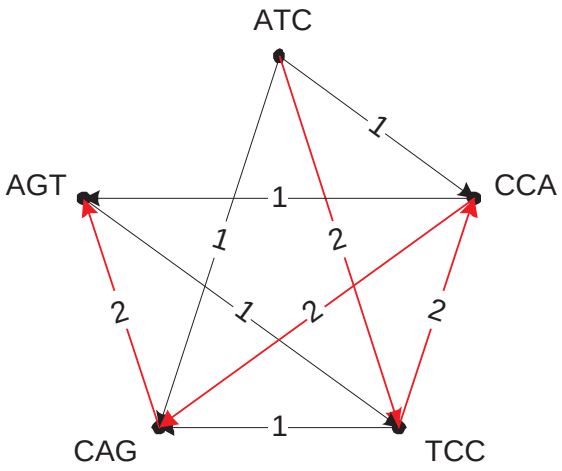


FIG. 3.4 – Graphe orienté pour P et le chemin le plus court

chevauchent par $k-1$ caractères. On construit un graphe orienté dont les sommets représentent les différentes chaînes ADN distinctes de longueur $k-1$. À chaque fragment correspond une arête entre les sommets représentant ses $k-1$ caractères de gauche et ses $k-1$ caractères de droite. Une super-chaîne candidate à la chaîne ADN peut alors être obtenue en parcourant chacune des arêtes du graphe. Afin d'obtenir la plus petite super-chaîne, il faut trouver un chemin, s'il en existe un, qui emprunte une fois et une seule, chacune des arêtes. Le mathématicien Euler résolut ce problème et énonça qu'un graphe admet ce type de chemins (dits eulériens) si et seulement s'il est connexe et ne possède pas plus de deux sommets de degré impair et si tous ses sommets sont de degré pair s'il est orienté. Le degré d'un sommet dans un graphe orienté correspond à la somme du nombre d'arêtes entrantes et du nombre d'arêtes sortantes. La chaîne ADN est donc obtenue en trouvant et en parcourant un chemin eulérien dans le graphe orienté. La figure 3.5 montre le graphe orienté (tiré de [Gus97]) correspondant à l'ensemble de fragments $P = \{AAA, AAC, ACA, CAC, CAA, ACG, CGC, GCA, ACT, CTT, TTA, TAA\}$.

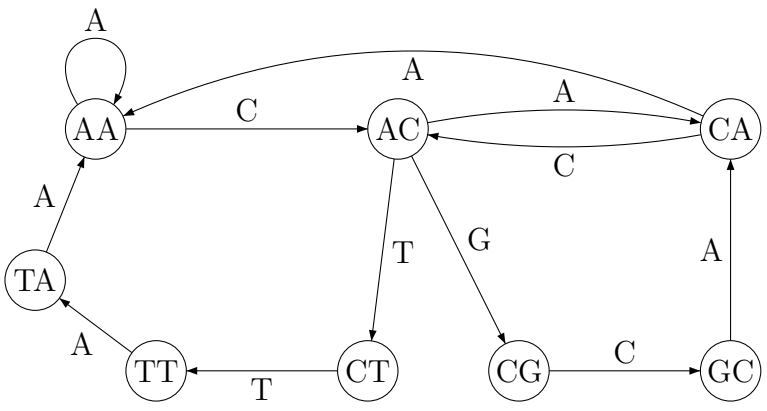


FIG. 3.5 – Graphe orienté pour l'ensemble de fragments $P = \{AAA, AAC, ACA, CAC, CAA, ACG, CGC, GCA, ACT, CTT, TTA, TAA\}$

Un chemin eulérien passe successivement par les sommets AC, CA, AC, CG, GC, CA, AA, AC, CT, TT, TA, AA et AA. pour obtenir la super-chaîne $ACACG$-$CAACTTAAA$.

La notion de séquence circulaire (le dernier caractère précède le premier) est aussi importante en bio-informatique. En effet, l'ADN des bactéries et des mitochondries est généralement circulaire. L'ADN des virus possède parfois aussi des propriétés circulaires. Dans certaines populations virales, l'ordre linéaire de l'ADN d'un individu est une rotation circulaire de celui d'un autre individu [Gus97]. Des cycles eulériens sont utilisés plutôt que des chemins pour pouvoir manipuler les chaînes comme des séquences circulaires.

3.2.2 Notre approche

Un programme conçu selon le paradigme orienté objets est représenté statiquement par son code source décrivant les entités et éléments interagissant pour exécuter les fonctionnalités. Un métamodèle est utilisé pour décrire les entités et les élé-

ments des motifs de conception et des programmes et ainsi créer leurs modèles. Ces modèles sont des graphes avec les entités comme sommets et les éléments comme arêtes. Les éléments considérés sont les relations binaires interclasses : création (instantiation), spécialisation, implémentation, utilisation, association, agrégation et composition. Les graphes sont orientés puisque les relations binaires interclasses le sont. Si plus d'une relation identique existe entre deux entités, une seule est conservée puisque les autres ne procurent aucune information supplémentaire. La figure 3.6 représente le graphe du modèle du motif de conception Composite.

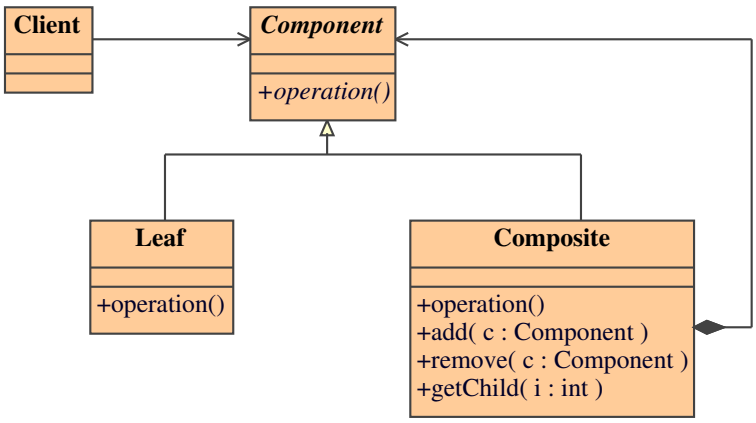


FIG. 3.6 – Graphe du modèle du motif de conception Composite

Comme les programmes et les motifs de conception peuvent être représentés sous la forme de graphes orientés, notre approche s'inspire des deux dernières approches présentées dans la section précédente pour créer les super-chaînes. Les chaînes de caractères sont construites en parcourant des cycles eulériens des graphes représentant les motifs et les programmes. L'importance d'avoir des chaînes circulaires sera expliquée à la fin de la section.

Les graphes orientés représentant les modèles de motifs et de programmes ne sont pas nécessairement eulériens. Il faut donc les rendre eulériens en ajoutant, par

exemple, des arêtes entre les sommets avec un nombre différent d'arêtes entrantes et sortantes. Ces sommets sont identifiés à l'aide de la matrice d'adjacence représentant le graphe. La matrice d'adjacence A d'un graphe fini G à n sommets est une matrice $n \times n$ où a_{ij} est égale au nombre d'arêtes allant du sommet i au sommet j. La figure 3.3 montre la matrice d'adjacence du graphe du motif de conception Composite sans prendre en considération le sommet *Client*. Le graphe n'est pas eulérien puisque ses sommets `Leaf` et `Component` possèdent des degrés impairs ($1 + 0 = 1$ pour $Leaf$ et $2 + 1 = 3$ pour $Component$).

	Component	Leaf	Composite	Degré sortant
Component	0	1	1	2
Leaf	0	0	0	0
Composite	1	0	0	1
Degré entrant	1	1	1	

TAB. 3.3 – Matrice d'adjacence du graphe du motif de conception Composite

Une liste optimale d'arêtes à ajouter au graphe non eulérien est obtenue à l'aide du *transportation simplex*. L'algorithme du simplex permet de résoudre des problèmes d'optimisation dont la fonction objective à optimiser et les contraintes à respecter sont linéaires (programmation linéaire). Le *transportation simplex* résout le problème de transport qui consiste à minimiser le coût total d'envoi de marchandises entre m origines (chacune avec un approvisionnement propre a_i) et n destinations (chacune avec une demande propre b_i). Ce problème consiste à minimiser la fonction de coût $\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$ où $\sum_{j=1}^{n} x_{ij} \leq a_i \ \forall i \leq m$, $\sum_{i=1}^{m} x_{ij} = b_j$ $\forall j \leq n$, $x_{ij} \geq 0 \ \forall i, j$ et c_{ij} est le coût de transport d'une unité de marchandise entre l'origine i et la destination j.

Les sommets avec plus d'arêtes entrantes que sortantes sont considérés comme origines et ceux avec plus d'arêtes sortantes que entrantes comme destinations. La

fonction des coûts c entre origines et destinations est considérée uniforme (c'est à dire à valeur unique). La figure 3.7 représente le graphe eulérien du modèle Composite. Une arête "dummy" a été ajoutée entre les sommets `Leaf` et `Component`.

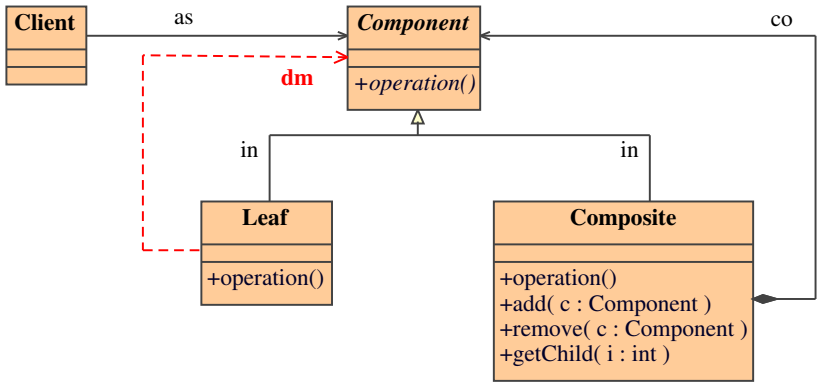


FIG. 3.7 – Graphe eulérien du modèle du motif de conception Composite

Tout graphe eulérien possède un cycle eulérien. Les représentations sous formes de chaînes de caractères sont obtenues en parcourant les cycles eulériens. Ceux-ci sont obtenus en résolvant le problème du postier chinois qui consiste à trouver le plus court chemin visitant chaque arête d'un graphe au moins une fois. Le cycle eulérien est, pour un graphe eulérien, la solution optimale au problème du postier chinois. La figure 3.8 montre un cycle eulérien pour le graphe du motif de conception Composite sans considérer le rôle `Client`.

Étant donné un sommet de départ s_d, la solution au problème du postier chinois est une liste unique d'arêtes commençant et se terminant par s_d et contenant tous les arêtes une seule fois. La représentation sous forme de chaîne de caractères est la série des étiquettes des sommets et des arêtes parcourues. Par exemple, pour le motif de conception Composite, la chaîne obtenue est :

Component in Leaf dm Component in Composite co Component.

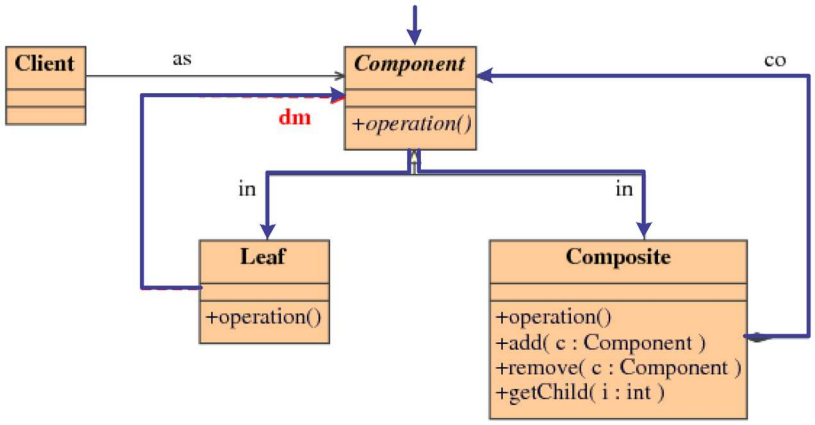


FIG. 3.8 – Cycle eulérien pour le graphe du modèle du motif de conception Composite

Le choix du sommet de départ influence le résultat. En effet, la chaîne obtenue à partir du sommet *Composite* pour le graphe du motif de conception Composite est : *Composite co Component in Leaf dm Component in Composite*. Les deux chaînes sont différentes mais sont toutefois équivalentes lorsque considérées comme des séquences circulaires. En effet, l'implémentation de l'algorithme du postier chinois utilisée choisit toujours la même arête pour sortir d'un sommet. L'arête utilisée dépend de son poids et de sa valeur lexicographique en cas d'égalité de poids entre deux arêtes.

CHAPITRE 4

TECHNIQUES DE DÉTECTION

Le problème de recherche de motifs de conception dans un programme en est un de recherche approximative. Toutefois, les distances utilisées en bio-informatique ne sont pas applicables. En effet, les approximations tolérées, dans notre cas, sont limitées. Par exemple, la chaîne *Component in Leaf dm ClassX in Composite co Component* a une distance de Hamming de seulement 1 avec la chaîne représentant le motif de conception Composite. Pourtant, cette micro-architecture est très loin de celle du motif de conception Composite. Dans ce chapitre, les différentes approximations souhaitables sont discutées. Trois techniques de détection de motifs de conception inspirées de la bio-informatique sont ensuite détaillées. Finalement, deux approches pour améliorer la précision des résultats sont présentées.

4.1 Approximations

Les patrons de conception ne sont pas toujours connus des développeurs et certains problèmes peuvent avoir été résolus de manières différentes de celles proposées par les patrons de conception. Certaines contraintes imposées par l'architecture d'un programme obligent parfois des modifications à la micro-architecture d'un motif de conception. Des micro-architectures identiques à un motif de conception peuvent également *être remaniées dans leurs structures et organisations* lors de l'évolution d'un programme [MHG06]. Les techniques de détection doivent permettre la détection de ces formes approchées des motifs de conception. Ces occurrences doivent toutefois être examinées manuellement par le mainteneur pour

s'assurer qu'elles sont véritables. Dans certains cas, des corrections basées sur les motifs de conception peuvent être appliquées afin d'améliorer le code.

Trois types d'approximation sont relativement fréquents. Premièrement (1), la relation entre deux entités peut être plus ou moins forte. Par exemple, la relation d'agrégation d'un motif de conception peut être remplacée par une relation plus forte (composition) ou plus faible (association, utilisation) dans le programme [GAA04]. Deuxièmement (2), tous les rôles d'un motif de conception ne doivent pas nécessairement être joués par des entités du programme. Parfois, certains rôles peuvent être absents sans compromettre la solution (ex : motif Composite sans *Leaf*). Troisièmement (3), la hiérarchie des entités peut être approchée. Une entité peut être insérée ou retirée de la hiérarchie suggérée par un motif de conception.

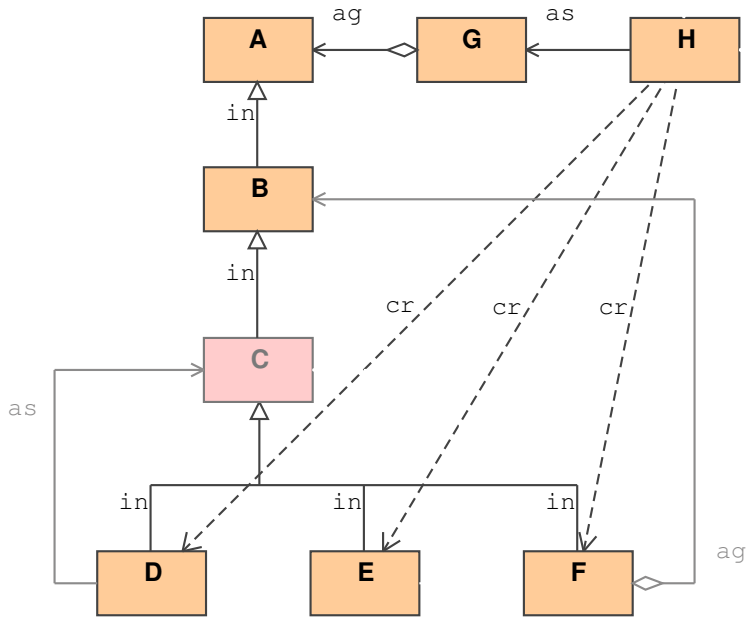


FIG. 4.1 – Programme avec deux occurrences approchées du motif de conception Composite

La figure 4.1 représente un exemple de programme avec deux occurrences approchées du motif de conception Composite. L'occurrence $\{Component = B, Composite = F, Leaf = D, E\}$ est approchée avec les approximations de type 1 et 3. La relation de *composition* est remplacée par une relation d'*agrégation* et la classe emphC est insérée entre la classe jouant le rôle de *Component* et la classe jouant le rôle de *Composite*. L'occurrence $\{Component = C, Composite = D, Leaf = E, F\}$ est également approchée puisque la relation entre la classe jouant le rôle de *Composite* et la classe jouant le rôle de *Component* est une relation de type *association* (approximation de type 1).

4.2 Approche avec programmation dynamique

L'approche par programmation dynamique semble à priori appropriée pour la recherche d'occurrences de motifs. La possibilité de pouvoir attribuer des coûts différents aux substitutions dépendamment des caractères est intéressante. Une manière de comparer deux chaînes de caractères consiste à donner une valeur de similarité à la place d'une distance. On cherche alors à maximiser la fonction objective plutôt qu'à la minimiser. Cette approche est beaucoup plus utilisée en bioinformatique. Dans le cas de la recherche de motifs, l'approximation de type 1 peut facilement être prise en compte. Il suffit, par exemple, de donner une valeur de similarité élevée entre deux relations identiques, une valeur moins grande pour deux relations de même type mais non identique (ex : composition et agrégation) et une valeur nulle entre deux relations différentes (ex : composition et héritage). Comme les occurrences peuvent se trouver n'importe où dans la chaîne représentant le programme, il faut aussi permettre l'insertion d'espaces au début et à la

fin de l'alignement. Le patron *A association B* aurait donc une valeur de similarité élevée avec la chaîne *C création A agrégation B dummyRelationship D*.

L'approche par programmation dynamique est toutefois confrontée à deux problèmes importants. Premièrement, une occurrence (approchée ou non) d'un motif de conception ne se trouve pas nécessairement de manière continue dans la chaîne du programme. Des espaces doivent pouvoir être insérés entre les triplets. Une façon de contourner ce problème pourrait être de considérer les triplets comme des caractères et modifier en conséquence la fonction de similarité. Le deuxième problème vient du fait que la chaîne représentant un motif de conception correspond plus à une expression régulière qu'à une chaîne de caractères. Par exemple, on ne cherche pas les caractères *Component*, *Leaf* ou *Composite* dans la chaîne du programme lorsqu'on recherche le motif Composite mais plutôt des entités pouvant jouer ces rôles. *Component*, *Leaf* et *Composite* peuvent être substitués par n'importe quelle entité du programme. Ceci ne serait pas vraiment un problème si chaque occurrence de ces caractères dans la chaîne du motif pouvait être remplacée par n'importe quelle entité du programme. Toutefois, cela amènerait un nombre trop important de fausses occurrences. Chaque occurrence d'un rôle dans la chaîne du motif doit être remplacée par la même entité. Comme ce problème est difficilement gérable avec la programmation dynamique et que cette technique n'est pas reconnue pour son efficacité, cette approche n'est finalement pas vraiment appropriée pour la recherche d'occurrences de motifs de conception.

4.3 Approche avec automates

La simulation d'automates est un moyen très utilisé pour rechercher les occurrences d'une expression régulière dans un texte. Il existe deux classes d'automates pour décrire un mot défini par une expression régulière, les *automates finis non déterministes* (`AFND`) et les *automates finis déterministes* (`AFD`). La différence entre ces deux classes d'automates est que tout état d'un `AFD` possède une seule transition sortante pour chaque symbole de l'alphabet tandis qu'un `AFND` peut avoir plusieurs états suivants pour un même symbole. Ces deux types d'automates reconnaissent toutefois les mêmes langages. Un `AFD` peut être construit à partir d'un `AFND` et vice-versa. La génération d'un automate à partir d'une expression régulière peut facilement être automatisée [Glu61, Tho68].

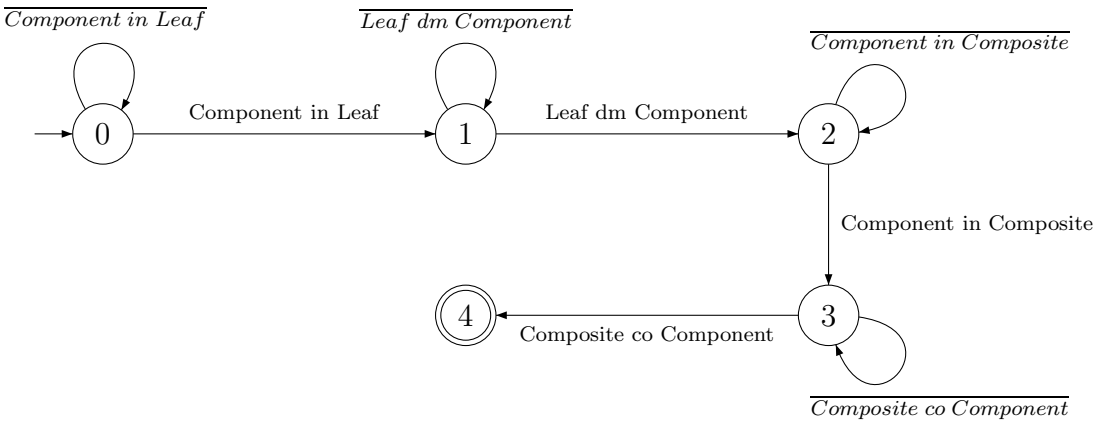
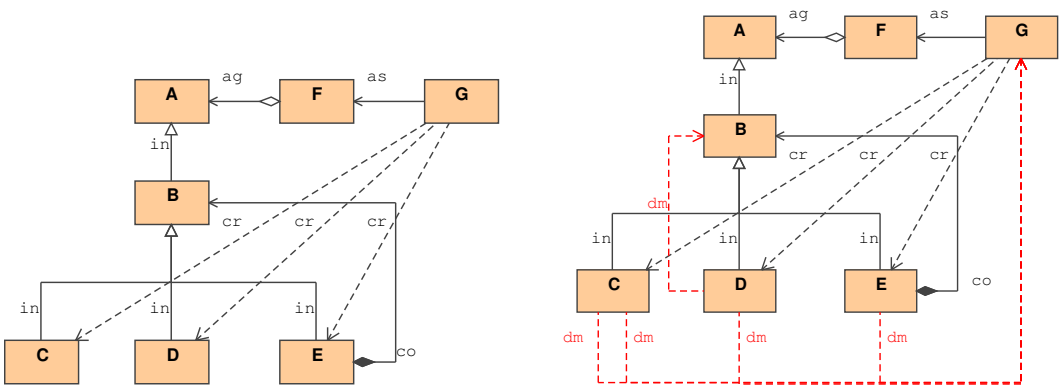


FIG. 4.2 – `AFND` pour la recherche du motif de conception Composite

La figure 4.2 représente l'`AFND` pour la recherche du motif de conception Composite. L'automate `AFND` possède $\lfloor m/2 \rfloor + 1$ états où m est le nombre de symboles dans la chaîne de caractères. Une transition avec conditions est ajoutée pour chaque triplet. Par exemple, le premier triplet correspond à la transition *Component in Leaf* entre l'état 0 et l'état 1. Des transitions en boucle sont également ajoutées à chaque

état, sauf le final, pour permettre des trous entre les triplets. Lors de la simulation de l'automate, une liste d'occurrence est créée et validée. Les conditions sur les transitions servent à s'assurer que chaque occurrence d'un rôle est remplacée par la même entité. Soit le programme représenté par la figure 4.3, l'automate lit le premier triplet *A in B* et les états 0 et 1 s'activent. L'état 1 s'active en créant une occurrence avec *A* comme *Component* et *B* comme *Leaf*. L'état 0 reste activé grâce à la boucle et permettra d'obtenir des occurrences avec des entités différentes pour les rôles *Component* et *Leaf*. Par exemple, en lisant le prochain triplet, une occurrence $\{Component = B, Leaf = D\}$ est créée. L'automate trouve l'occurrence $\{Component = B, Leaf = D, Composite = E\}$ en passant successivement par les états 0, 0, 1, 2, 3 et 4. Lorsque l'état 2 est activé, la transition *Component in Composite* peut être empruntée si et seulement si le premier symbole lu correspond à l'entité associée au rôle *Component*. Dans le cas de l'exemple, la transition a pu être prise lorsque l'automate a lu le triplet *B in E*.



(a) Modèle du programme

(b) Modèle eulérien

```
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) Représentation sous la forme de chaîne de caractères du modèle eulérien

FIG. 4.3 – Exemple simple de programme

Contrairement aux automates servant à vérifier si un mot appartient au langage défini par des expressions régulières, le chemin parcouru par les automates pour la recherche d'occurrences de motif est important. La simulation d'un `AFND` fonctionne avec une liste de chemins et par retours en arrière. Pour chaque nouveau triplet lu, les transitions suivantes possibles de tous les chemins actifs sont testées. Chaque chemin se terminant à l'état final correspond à une occurrence. La figure 4.4 représente le pseudo-code simplifié de l'algorithme de simulation. La simulation se fait en lançant la fonction *match* avec la chaîne du programme, l'état initial de l'automate et une occurrence vide comme paramètres.

```
match(input, currentState, currentOccurrence)
occurrences := {}
IF currentState IS finalState
   ADD currentOccurrence IN occurrences
   RETURN occurrences
ENDIF
IF (NOT END OF input)
   FOR ALL transitions x IN currentState.transitions
      IF x.conditionRespected(input.getTriplet, currentOccurrence)
         UPDATE currentOccurrence
         ADD match(input.removeTriplet, x.destinationState, currentOccurrence) IN occurrences
      ENDIF
   ENDFOR
ENDIF
RETURN occurrences
```

FIG. 4.4 – Pseudo-code simplifié pour la simulation de l'`AFND`

Le nombre de chemins, se rendant à l'état final ou non, peut devenir rapidement très grand et rend cette technique moyennement efficace. L'utilisation d'`AFD` règle ce problème puisqu'un seul chemin permet de trouver les différentes occurrences. La construction de ce type d'automates pour la recherche de motifs est par contre plus complexe. La figure 4.5 représente l'`AFD` pour la recherche du motif de conception Composite. À chaque transition est associée une condition. Par exemple, la transi-

tion C_2 *in* C_3 entre les états 1 et 2 peut être prise si et seulement si C_2 représente l'entité associée au rôle *Component* (C_2 de la transition C_1 *co* C_2 entre les états 0 et 1). La transition * d'un état est prise lorsque les autres transitions de l'état ne peuvent pas l'être. Elle correspond à tous les autres triplets possibles non spécifiés par les transitions existantes. Les transitions contenant la relation *dm* peuvent être considérées comme des ϵ–transitions dans les `AFND` et comme des transitions sans condition dans les `AFD`.

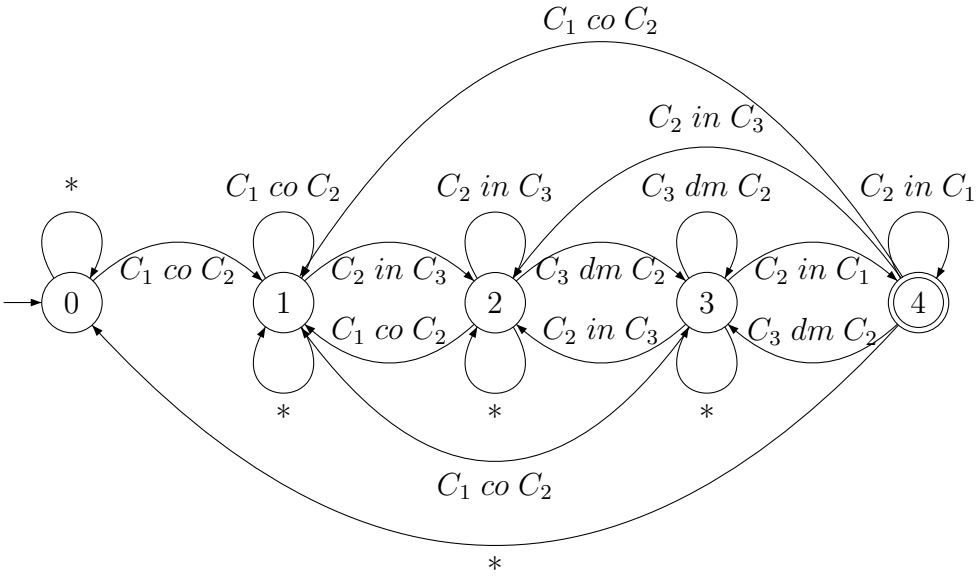


FIG. 4.5 – `AFD` pour la recherche du motif de conception Composite

Les occurrences approchées peuvent facilement être obtenues avec l'approche par automates. L'approximation de type 1 s'obtient en ajoutant simplement des transitions. Par exemple, ajouter des transitions C_x *ag* C_y là où il y a des transitions C_x *co* C_y dans l'`AFD` permet d'avoir des occurrences avec une relation d'*agrégation* à la place de la relation de *composition* (par exemple, C_1 *ag* C_2 entre les états 0 et 1). L'approximation de type 2 s'obtient également en ajoutant des transitions. Ajouter une transition sans condition entre l'état 2 et l'état 4 de l'`AFD` permet de trouver

des occurrences sans entité associée au rôle *Leaf*. Finalement, l'approximation de type 3 s'obtient en modifiant les conditions sur les transitions en permettant, par exemple, aux enfants ou aux parents d'une entité de jouer son rôle.

L'approche par automates présentée possède toutefois un problème. En effet, elle utilise l'hypothèse que les triplets d'une occurrence se retrouvent toujours dans le même ordre. L'approche utilisée pour créer les chaînes de caractères en maximise les chances mais il est possible que des occurrences soient oubliées par l'approche par automates. Ce problème pourrait être contourné en construisant des automates avec des arcs entre chaque paire d'états et en ajoutant la condition qu'une occurrence existe si et seulement si tous les états de l'automate ont été parcourus.

4.4 Approche avec vecteurs de bits

Les algorithmes vectoriels sont reconnus pour leur efficacité lorsque le mot à rechercher est petit. Afin d'obtenir un algorithme efficace et régler les différents problèmes rencontrés avec les deux autres approches, nous avons développé un algorithme vectoriel itératif s'inspirant des approches numériques de recherche exacte en bio-informatique.

Les entités jouant des rôles dans un motif de conception sont trouvées en effectuant des opérations binaires standard (ou–logique, et–logique, décalage à droite, décalage à gauche...) sur des séquences de bits appelées *vecteurs caractéristiques*.

Définition : Soit un symbole l dans une chaîne de caractères x, le vecteur caractéristique de l, noté **l**, est défini comme :

$$\mathbf{l}_i = \begin{cases} 1 & \text{si } x_i = l \\ 0 & \text{sinon.} \end{cases}$$

Chaque symbole de la chaîne du programme a son propre vecteur caractéristique d'une longueur égale à la longueur de la chaîne du programme. 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. Les vecteurs caractéristiques correspondent, en quelque sorte, à la première ligne du tableau M des algorithmes `Shift-And` et `Shift-Or` (3.1). Les vecteurs caractéristiques sont circulaires puisque les chaînes de caractères le sont. Le décalage à droite d'un vecteur caractéristique $\mathbf{x} = x_1 \dots x_m$ est défini comme $\rightarrow \mathbf{x} = x_m x_1 \dots x_{m-1}$. Tous les éléments ont été décalés d'une position vers la droite de manière circulaire. Le décalage à gauche de $\mathbf{x}$ est $\leftarrow \mathbf{x} = x_2 \dots x_m x_1$.

Par exemple, pour le programme représenté par la figure 4.3, le vecteur caractéristique de la classe G est :

$$\mathbf{G} = \underbrace{00000000000000}_{14} 10001000100010000$$

tandis que celui de la relation d'héritage *in* est :

$$\mathbf{in} = 010100010001 \underbrace{0000000000000000000}_{19}.$$

Notre algorithme lit la chaîne du motif par triplets et associe des entités du programme aux rôles. Par exemple, pour le motif Composite, l'algorithme débute en lisant le triplet *Component in Leaf*. Il recherche les entités avant et après le symbole *in* dans la chaîne du programme avec des opérations binaires sur les vec-

teurs caractéristiques pour les associer respectivement aux rôles *Component* et *Leaf* (opérations détaillées dans le pseudo-code de la figure 4.6).

```
before := {}
after := {}
→token
FOR EACH ENTITY X IN THE STRING
   conjunctionX := X ∧ token
   IF conjunctionX IS NOT NULL
      ADD X IN after
      ←←conjunctionX
      FOR EACH ENTITY Y IN THE STRING
         conjunctionY := Y ∧ conjunctionX
         IF conjunctionY IS NOT NULL
            ADD Y IN before
         ENDIF
      ENDFOR
   ENDIF
ENDFOR
```

FIG. 4.6 – Pseudo-code pour récupérer les entités avant et après un symbole dans la chaîne représentant le programme

Chaque couple (*Component*, *Leaf*) représente une occurrence potentielle qui sera validée ou mise-à-jour après la lecture de chaque triplet de la chaîne du motif. Le prochain triplet de la chaîne du motif Composite représente une relation *dm* et est ignoré. L'opération est répétée pour le troisième triplet (*Component in Composite*). Le rôle *Component* a déjà été rencontré et des entités du programme y ont été associées. L'algorithme essaie donc d'associer des entités au rôle *Composite* respectant la contrainte d'héritage avec les entités jouant le rôle de *Component* en vérifiant si la conjonction du vecteur caractéristique de chaque entité du programme avec la conjonction des vecteurs caractéristiques → **in** et →→ **Component** n'est pas nulle. La présence de 1 dans le vecteur résultat signifie que l'entité est enfant de l'entité jouant le rôle de *Component* (se trouve directement après le symbole de l'entité jouant le rôle de *Component* et le symbole *in*). Par exemple, les opérations

suivantes sont effectuées sur les vecteurs caractéristiques pour savoir si l'entité E peut jouer le rôle de *Composite* dans l'occurrence partielle potentielle $\{$*Component* $= B$, *Leaf* $= C\}$:

$$
\begin{aligned}
\rightarrow\rightarrow \mathbf{B} &= 0000100010001\underbrace{0\ldots0}_{18} \\
\rightarrow \mathbf{in} &= 0010100010001\underbrace{0\ldots0}_{18} \\
(\rightarrow\rightarrow \mathbf{B}) \wedge (\rightarrow \mathbf{in}) &= 0000100010001\underbrace{0\ldots0}_{18} \\
\mathbf{E} &= \underbrace{00000000}_{8}1\underbrace{0\ldots0}_{15}1\underbrace{000000}_{6} \\
(\rightarrow\rightarrow \mathbf{B}) \wedge (\rightarrow \mathbf{in}) \wedge \mathbf{E} &= \underbrace{00000000}_{8}\mathbf{1}\underbrace{0\ldots0}_{22}.
\end{aligned}
$$

L'occurrence $\{$*Component* $= B$, *Leaf* $= C$, *Composite* $= E\}$ est ajoutée à la liste d'occurrence potentielle parce que le vecteur $(\rightarrow\rightarrow \mathbf{B}) \wedge (\rightarrow \mathbf{in}) \wedge \mathbf{E}$ contient un 1. L'entité E se trouve après les symboles B *in* dans la chaîne du programme. La table 4.1 montre les occurrences après la lecture du premier, troisième et quatrième triplet. L'ordre dans lequel les triplets sont lus influence le temps d'identification. Il est préférable de traiter les triplets représentant des relations plus rares en premier pour ainsi réduire le nombre d'occurrences potentielles en début d'analyse. Ceci peut être fait en donnant des poids différents aux arêtes du graphe représentant le motif lors de la résolution du problème du postier chinois, en faisant des décalages circulaires ou en effectuant un post-traitement sur la chaîne du motif. Par exemple, la chaîne du motif de conception Composite (voir 3.2.2) peut être décalée de manière circulaire pour obtenir la chaîne *Composite co Component in Leaf dm Component in Composite* et ainsi traiter la relation plus rare de *composition* au début. La table

4.2 montre que le nombre d'occurrences potentielles est ainsi diminué.

Triplets							
First (*in*)		Third (*in*)			Fourth (*co*)		
Component	Leaf	Component	Leaf	Composite	Component	Leaf	Composite
A	B	A	B	B	B	C	E
B	C	B	C	C	B	D	E
B	D	B	C	D	B	E	E
B	E	B	C	E			
		B	D	C			
		B	D	D			
		B	D	E			
		B	E	C			
		B	E	D			
		B	E	E			

TAB. 4.1 – Occurrences après la lecture du premier, troisième et quatrième triplet.

Les approximations de type 1 et 2 sont traitées différemment qu'avec les automates. L'approximation de type 1 s'obtient en utilisant la conjonction des vecteurs caractéristiques des relations équivalentes plutôt que celui de la relation. L'approximation de type 2 s'obtient en permettant d'ignorer des relations (triplets). Un rôle peut être absent en ignorant toutes les relations qui lui sont reliées.

4.5 Amélioration de la précision

La précision des résultats dépend en grande partie des données analysées. L'étude seule de la structure d'un motif et d'un programme ne permet pas toujours une grande précision. L'ajout d'approximations dans la recherche permet la détection de motifs approchés mais augmente aussi beaucoup le nombre de fausses occurrences. Deux approches pour améliorer la précision des résultats ont été implémentées.

Triplets							
First (*co*)		Second (*in*)			Fourth (*in*)		
Composite	Component	Composite	Component	Leaf	Composite	Component	Leaf
E	B	E	B	C	E	B	C
		E	B	D	E	B	D
		E	B	E	E	B	E

TAB. 4.2 – Occurrences après la lecture du premier, deuxième et quatrième triplet.

Premièrement, nos chaînes de caractères représentant les motifs de conception contiennent ce qui doit être trouvé. Certains patrons de conception précisent aussi ce qui ne doit pas l'être. Par exemple, l'entité jouant le rôle d'*Adaptee* dans le patron de conception Adapter ne doit pas connaître l'entité jouant le rôle d'*Adapter* (4.7). Le nombre de fausses occurrences peut donc être réduit en ajoutant des relations de type *ignorance* dans les chaînes de caractères. Une relation d'*ignorance* peut être considérée comme un ensemble de relations négatives (pas de *composition*, pas d'*agrégation*, pas d'*association*, etc). Cet ajout est facilement possible avec l'approche par vecteurs de bits en utilisant l'opération de négation. Les triplets représentant les relations d'*ignorance* sont ajoutés à la fin des chaînes des motifs. Ils jouent, en quelque sorte, le rôle de filtres.

La réduction de l'espace de recherche par l'étude empirique de métriques améliore la performance d'un outil de détection mais aussi sa précision. L'analyse des métriques a donc été ajoutée à nos approches de détection. Une entité peut jouer un rôle dans une occurrence si et seulement si son empreinte respecte celle du rôle en question. Une liste d'entités possibles, pour chacun des rôles d'un motif, est calculée au début de l'analyse. Une entité peut être associée à un rôle si et seule-

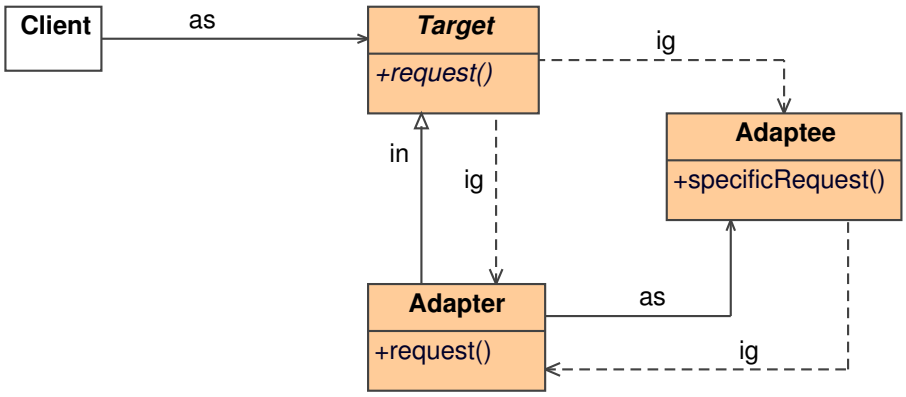


FIG. 4.7 – Diagramme représentant le motif de conception `Adapter` avec relations d'ignorance

ment si elle figure dans la liste de ce rôle. Les métriques sont calculées par POM (Primitives, Operators, Metrics) sur le modèle du programme, obtenu avec le métamodèle PADL (Pattern and Abstract-level Description Language). Le processus d'extraction des métriques, la création des empreintes des rôles et l'approche en général est détaillée dans [GSZ04].

4.6 Synthèse

Un bon algorithme de détection de motifs de conception doit être efficace, précis et permettre la détection de formes approchées. Les trois approches abordées dans ce chapitre permettre la détection de formes approchées des motifs de conception. L'approche par programmation dynamique le gère très facilement mais n'est ni efficace ni précise. Le fait que la chaîne du motif correspond plus à une expression régulière qu'à une chaîne de caractères n'est pas efficacement gérable avec la programmation dynamique. Les expressions régulières sont fortement reliées à la théorie des automates. L'approche avec automates est donc plus efficace et plus

précise mais elle suppose que les triplets d'une occurrence se retrouvent toujours dans le même ordre, ce qui n'est pas nécessairement le cas. Certaines occurrences peuvent donc être oubliées en utilisant cette approche. L'approche avec vecteurs de bits s'avère la plus efficace et la plus précise. La manipulation de vecteurs de bits est d'ailleurs reconnue pour être très efficace.

CHAPITRE 5

IMPLÉMENTATION ET EXPÉRIMENTATIONS

Afin de valider notre approche, de trouver ses limites et de comparer son efficacité en temps par rapport aux autres approches de détection de motifs de conception, nous avons développé EPI (Efficient Patterns Identification), un outil permettant la détection d'occurrences de motifs de conception exactes et approchées avec les techniques du chapitre 4. La première section traite de l'implémentation de l'outil et de la façon de l'utiliser. Une étude de cas est ensuite présentée et les résultats sont comparés avec ceux de deux autres outils similaires.

5.1 Implémentation

EPI a été développé en JAVA avec ECLIPSE. Plusieurs outils et algorithmes existants ont aussi été empruntés. Dans cette section sont présentés les différents composants de EPI.

PADL. Nous utilisons le métamodèle PADL [Gué05] pour décrire les motifs de conception et les programmes. Il définit leur structure statique ainsi qu'une partie de leur comportement. Les relations binaires inter-classes ainsi que l'envoi de messages sont, entre autre, définis. PADL est associé à plusieurs analyseurs syntaxiques pour construire des modèles à partir de fichiers AOL, C++ et Java. Il contient également une banque de motifs de conception. Nous parcourons les modèles obtenus pour construire les chaînes de caractères les représentant.

Transportation Simplex. Afin de rendre les graphes eulériens, nous utilisons une implémentation de l'algorithme du *transportation simplex* de Park[1].

Problème du postier chinois. La construction des chaînes de caractères est faite avec l'implémentation efficace de Thimbleby [Thi03] d'un algorithme résolvant le problème du postier chinois. L'implémentation utilise plusieurs algorithmes connus pour leur efficacité comme celui de Floyd-Warshall pour obtenir les chemins les plus courts.

Ptidej. EPI peut être utilisé seul mais a aussi été intégré à PTIDEJ pour visualiser les résultats (voir la figure 5.3). PTIDEJ (Pattern Trace Identification, Detection, and Enhancement in Java) [Gué05] est une suite d'outils dédiée à l'analyse et la maintenance de programmes orientés objets. L'interface est toutefois la même avec ou sans PTIDEJ (voir les figures 5.1 et 5.2).

L'interface principale permet le choix de l'analyseur (*solver*), du motif de conception, du type d'approximation et l'utilisation ou non de la réduction du domaine de recherche avec les métriques. La chaîne du motif est générée suite au choix du motif de conception et peut être modifiée par le mainteneur. Le choix des approximations se fait à l'aide de l'interface 5.2(a) en choisissant les relations équivalentes de chaque relation. La relation *pathInheritance* permet l'approximation de type 3 tandis que la relation *null* permet d'ignorer un type de relation afin d'obtenir l'approximation de type 2.

Metrical Ptidej Solver. Les listes d'entités potentielles pour un rôle d'un motif de conception d'après l'étude empirique de leurs métriques sont calculées avec

[1]Voir `www.orlab.org`.

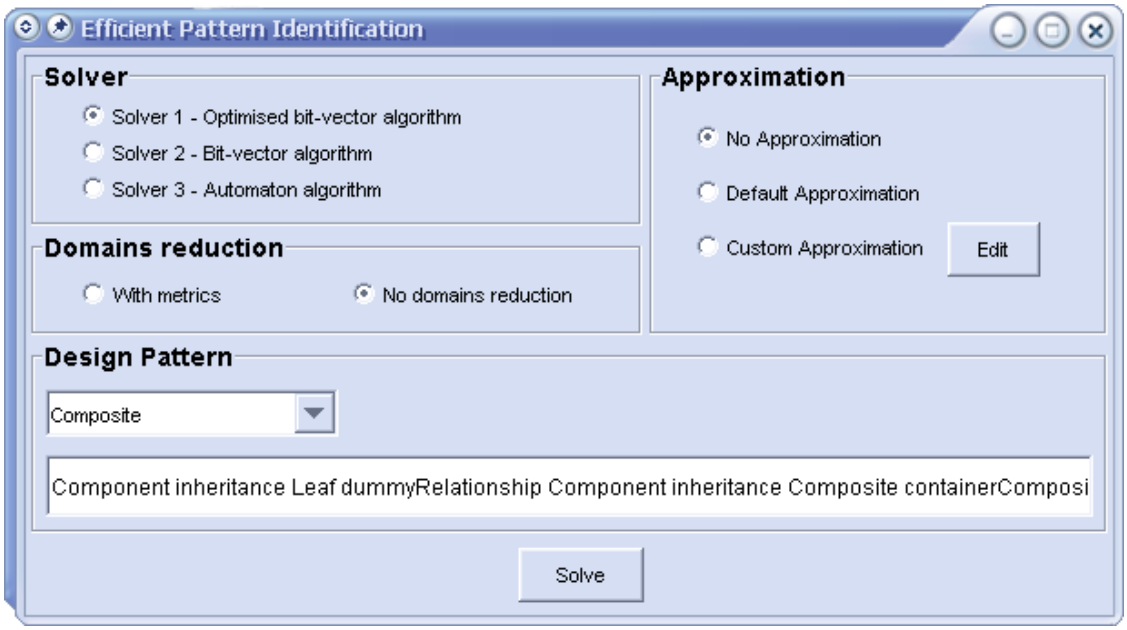
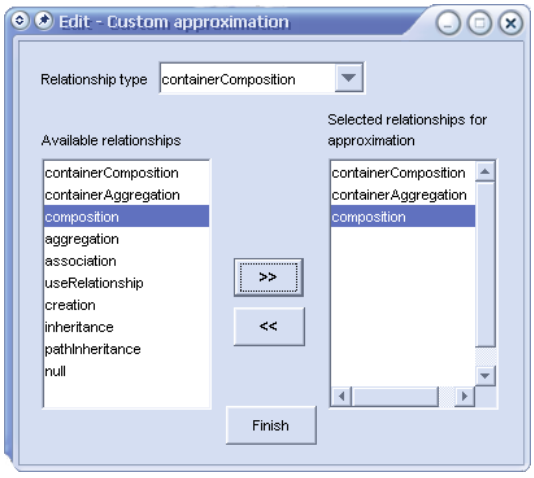
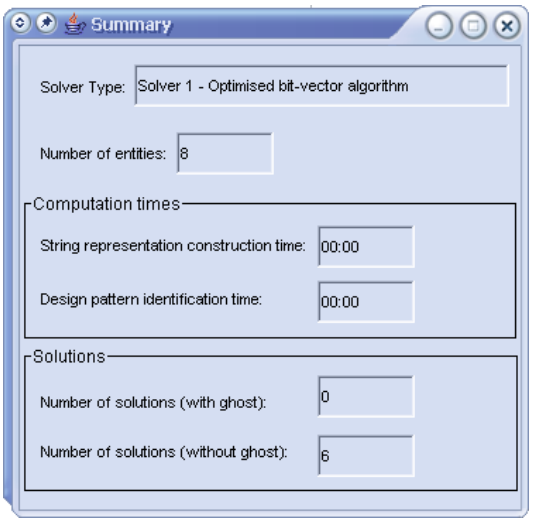


FIG. 5.1 – Interface principale de EPI



(a) Interface pour le choix des approximations



(b) Interface pour les résultats

FIG. 5.2 – Interfaces de EPI pour les approximations et les résultats

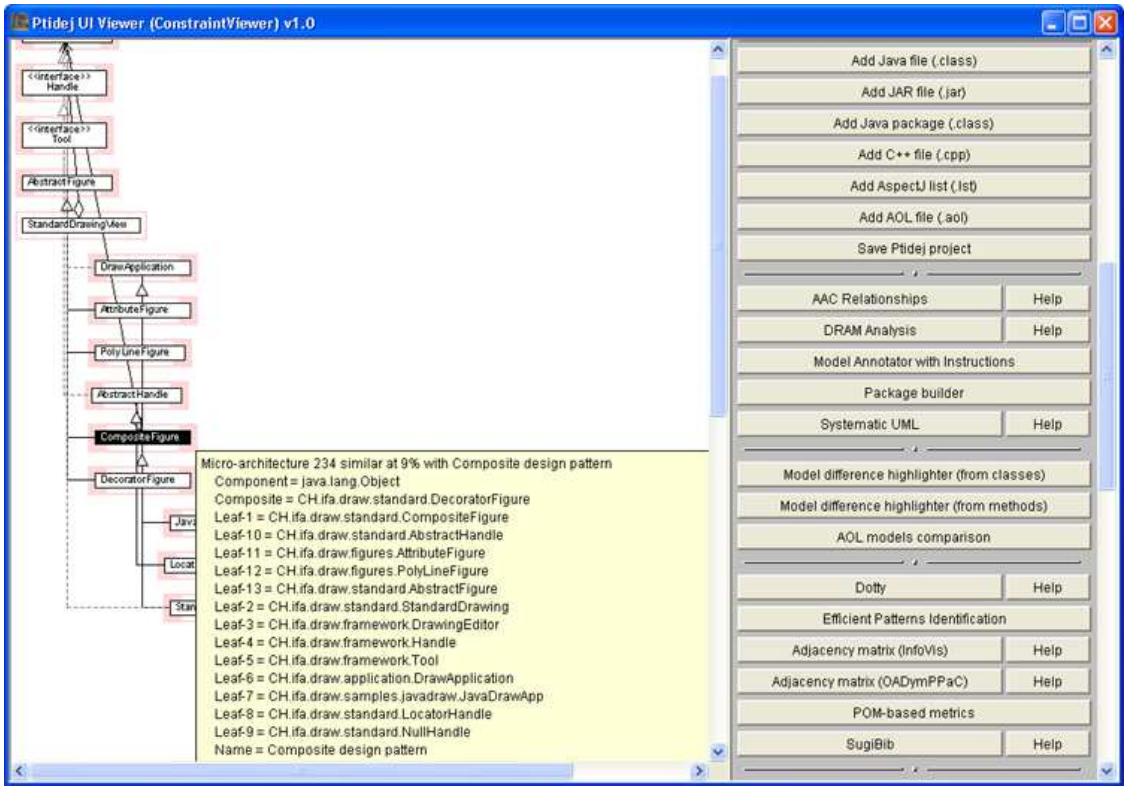


FIG. 5.3 – Visualisation des résultats dans PTIDEJ

Metrical Ptidej Solver, un des analyseurs inclus dans Ptidej. Cet analyseur utilise la bibliothèque de métriques logiciels POM pour réduire l'espace de recherche.

EPI. Les deux principales tâches de EPI sont de convertir en chaînes de caractères les modèles du motif et du programme et de rechercher les occurrences de l'une dans l'autre. La figure 5.4 représente l'architecture simplifiée de EPI avec les principaux paquetages et principales classes. Le paquetage principal *solver* contient les classes permettant d'effectuer les deux tâches mentionnées. La classe `StringBuilder` permet la construction d'une chaîne de caractères à partir d'un modèle PADL ou d'un ensemble de fichiers sources. La hiérarchie de classes `EPISolver` représente les différents analyseurs possibles pour la recherche de motifs. Chaque sous-classe implémente la fonction *computeSolutions*() permettant de lancer la recherche et retournant la liste des occurrences trouvées. Un fichier de résultats pouvant être lu par Ptidej est généré. La figure 5.5 représente un exemple du contenu d'un fichier de résultats. Les analyseurs à base de vecteurs de bits utilisent des *SparseBitSet* pour limiter l'espace mémoire utilisé. Un *sparse bitset* est une collection de valeurs booléennes optimisée pour lorsque la majorité des valeurs sont fausses.

Les étapes à effectuer pour la recherche d'un motif de conception dans un programme avec Ptidej sont :

1. Charger un programme dans Ptidej.
2. Lancer l'outil EPI qui se trouve dans l'onglet *Tools*.
3. Choisir les options de la recherche.
4. Lancer l'analyse en appuyant sur le bouton *Solve*.
5. Charger le fichier de résultats afin de les visualiser.

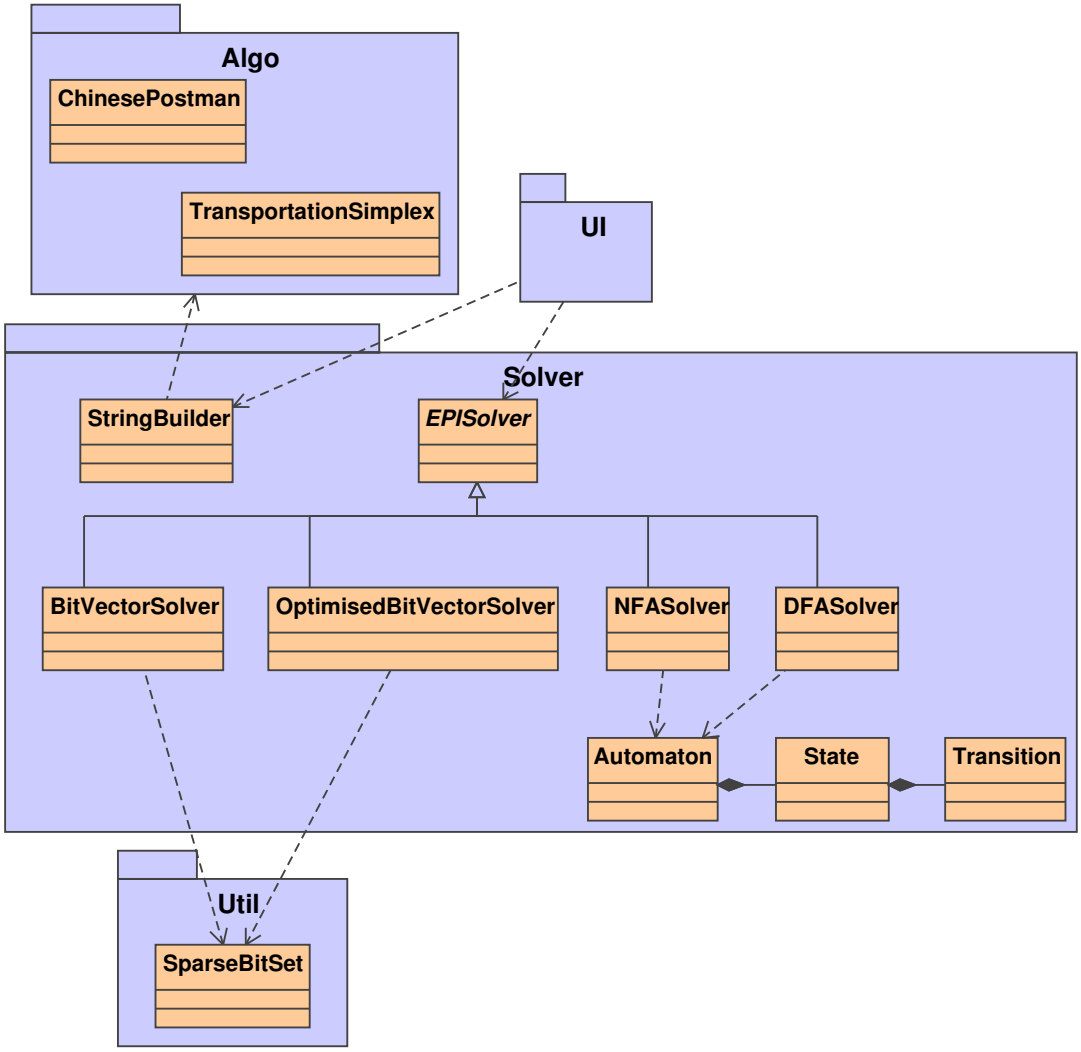


FIG. 5.4 – Architecture simplifiée de EPI

```
 1.  # Micro-architecture 1 similar at 100% with Composite
 2.  1.100.Component = C
 3.  1.100.Leaf = D
 4.  1.100.Composite = D
 5.
 6.  # Micro-architecture 2 similar at 100% with Composite
 7.  2.100.Component = C
 8.  2.100.Leaf = F
 9.  2.100.Composite = D
10.
11.  # Micro-architecture 3 similar at 100% with Composite
12.  3.100.Component = C
13.  3.100.Leaf = E
14.  3.100.Composite = D
```

FIG. 5.5 – Exemple du contenu d'un fichier de résultats généré par EPI

5.2 Étude de cas

Étant donné les problèmes de performance des différentes techniques de détection de motifs de conception existantes, le but de EPI est d'améliorer les temps d'identification tout en obtenant au moins la même qualité de résultats. En effet, un outil extrêmement rapide mais oubliant un nombre important d'occurrences n'est pas très utile. Dans cette section, une étude des temps d'identification sur des programmes de différentes tailles est présentée et les résultats sont comparés avec ceux de deux outils similaires.

Plusieurs travaux montrent que la programmation par contraintes obtient des résultats de bonne qualité et que les métriques réduisent grandement les temps d'identification. Nous avons donc comparé notre analyseur à base de vecteurs de bits à deux analyseurs inclus dans PTIDEJ utilisant la programmation par contraintes avec explications. Le premier n'utilise aucune technique de réduction de domaines

tandis que le second implémente l'approche avec métriques. Il aurait été intéressant d'inclure d'autres outils dans l'étude mais plusieurs outils existants ne permettent pas les mêmes analyses que `EPI`. Par exemple, `PAT` analyse seulement des programmes C++, `SOUL` que des programmes `SMALLTALK` tandis que `Fujaba`, un outil utilisant la logique floue, ne permet pas la détection d'un seul patron à la fois. De plus, `PAT` et `SOUL` ne permettent pas la détection d'occurrences approchées. Les autres outils ne sont tout simplement pas disponibles.

Les trois techniques ont été testées sur trois logiciels libres de taille moyenne, Juzzle v0.5, JHotDraw v5.1 et QuickUML 2001. Juzzle est un petit jeu de puzzles, JHotDraw un programme de dessins vectoriels et QuickUML un éditeur de diagrammes de classes UML. Les trois logiciels ont été développés en Java et sont composés de respectivement 99, 261 et 373 classes. Les trois techniques de recherche ont été comparées avec les patrons de conception Abstract Factory et Composite, deux patrons connus avec des buts et solutions différents. Les temps d'identification ont été récupérés à l'aide du *plugin* pour Eclipse, `Eclipse Profiler` [SS04]. Les calculs ont tous été faits trois fois avec un AMD Athlon 64bits à 2GHz et le temps médian a été gardé.

Programmes	Tailles (classes)	Temps de calcul (secondes)
Juzzle v0.5	99	3
JHotDraw v5.1	261	45
QuickUML 2001	373	135

Tab. 5.1 – Temps de calcul pour construire les chaînes de caractères représentant les programmes

La première étape de notre algorithme consiste à construire une représentation du programme sous la forme d'une chaîne de caractères. Le tableau 5.1 représente les temps de calcul pour la construction de la chaîne des trois logiciels. La

construction de la chaîne de caractères ne doit être effectuée qu'une seule fois. Le tableau 5.2 représente les temps d'identification en secondes de la programmation par contraintes avec explications (CP) [GJ01b, Gué05], de la programmation par contraintes avec explications et réduction de l'espace de recherche par les métriques (CP+M) [GSZ04], de l'algorithme de vecteurs de bits (BV) et d'une version optimisée de l'algorithme de vecteurs de bits (BV+O) [KGH06]. La différence entre BV et BV+O consiste à limiter les calculs en les faisant en début d'analyse et en gardant en mémoire le résultat de la conjonction du vecteur caractéristique de chaque entité avec celui de chaque relation présente dans la chaîne du motif. Ceci évite de refaire plusieurs fois certains calculs lorsqu'une relation se retrouve plusieurs fois dans un motif. Cette simple modification améliore grandement les temps d'identification. Toutes les approximations discutées dans la section 4.1 sont prises en compte. Les temps d'identification avec la technique avec vecteurs de bits sont nettement plus bas que ceux de la programmation par contraintes avec ou sans réduction de l'espace de recherche par les métriques.

	CP	CP+M	BV	BV+O
	Abstract Factory			
JUZZLE v0.5	1	2	0.9	0.3
JHOTDRAW v5.1	1202	275	218	27
QUICKUML 2001	785	153	97	29
	Composite			
JUZZLE v0.5	45	3	0.5	0.3
JHOTDRAW v5.1	$+\infty$	17362	129	25
QUICKUML 2001	$+\infty$	26514	185	27

TAB. 5.2 – Temps d'identification (en secondes) des motifs de conception

Malgré un nombre important de travaux sur l'identification de patrons de conception, aucun consensus n'existe pour la définition de ce qu'est une occurrence. Par exemple, le programme de la figure 4.3 peut contenir une ou deux occurrences

	Exactes			Approchées		
	CP	CP+M	BV	CP	CP+M	BV
	Abstract Factory					
JUZZLE v0.5	19/0	0/0	19/0	179/9	6/0	53/13
JHOTDRAW v5.1	216/221[3]	104/69	216/221	5245/2994	1444/849	408/194
QUICKUML 2001	164/57	46/23	164/57	2002/593	356/124	273/118
	Composite					
JUZZLE v0.5	0/0	0/0	0/0	1726/20	0/0	72/0
JHOTDRAW v5.1	N/A	0/0	0/0	N/A	31709/16983	1083/609
QUICKUML 2001	N/A	0/0	0/0	N/A	14920/4743	5536/513

TAB. 5.3 – Nombre d'occurrences identifiées des deux motifs de conception avec et sans entités fantômes

du motif de conception Composite, $\{Component = \texttt{B}, Composite = \texttt{F}, Leaf = \{\texttt{D}, \texttt{E}\}\}$ ou $\{Component = \texttt{B}, Composite = \texttt{F}, Leaf = \texttt{D}\}$ et $\{Component = \texttt{B}, Composite = \texttt{F}, Leaf = \texttt{E}\}$. Afin de comparer nos résultats avec ceux de PTIDEJ, nous considérons qu'une occurrence d'un motif de conception consiste en une micro-architecture où chaque rôle du motif est joué par une seule entité du programme. Une micro-architecture semblable au motif de conception Composite avec trois *Leaf* représente donc trois occurrences. Le tableau 5.3 représente le nombre d'occurrences des motifs Abstract Factory et Composite trouvées par les trois approches dans les trois logiciels. Le tableau contient le nombre d'occurrences avec et sans entités *fantômes* (*ghost entities*), qui sont des entités connues seulement par référence.

	Occurrences existantes	
	Abstract Factory	Composite
JUZZLE v0.5	0	0
JHOTDRAW v5.1	0	70
QUICKUML 2001	13	22

TAB. 5.4 – Nombre d'occurrences existantes des deux motifs de conception

[3]Nombre d'occurrences avec/sans entités fantômes

Les micro-architectures similaires aux motifs de conception ont été identifiées manuellement dans les trois programmes pour valider les résultats (voir le tableau 5.4). Nous ne prétendons pas avoir identifié manuellement toutes les occurrences (exactes et approchées) mais celles identifiées ont été trouvées par les trois approches, ce qui montre leur fiabilité. Ces résultats confirment l'hypothèse formulée et démontrée dans [GSZ04] que les occurrences éliminées par l'étude empirique des métriques sont des faux-positifs. Aucune occurrence exacte du motif de conception Composite n'a été identifié dans les trois programmes car la version actuelle de PADL ne reconnaît pas les relations de type *composition*. Elles sont reconnues en tant qu'*agrégation*.

Le nombre d'occurrences approchées est très élevé mais elles peuvent être triées selon leur distance par rapport à la solution exacte. L'approximation de type 1 augmente considérablement le nombre d'occurrences identifiées puisque les relations d'agrégation et de composition du motif peuvent être remplacées par des relations d'association ou d'utilisation, qui sont beaucoup plus fréquentes dans les programmes. De plus, seule la structure des motifs est identifiée. Dans un programme, plusieurs microarchitectures peuvent avoir la même structure qu'un motif de conception mais avec une utilité et un comportement différents. Le nombre d'occurrences obtenues par la programmation par contraintes est plus élevé puisque davantage d'approximations sont effectuées. Les contraintes sont remplacées ou retirées jusqu'à ce qu'il ne reste plus de contraintes dans le système. La majorité de ces occurrences est toutefois très éloignée du motif de conception. La définition d'occurrence utilisée contribue également au nombre élevé d'occurrences. Par exemple, les 22 occurrences existantes du motif de conception Composite dans le programme QuickUML correspondent à seulement 2 micro-architectures avec plusieurs *Leaf*.

L'analyse de programmes de plus grosse taille n'est pas vraiment possible avec la programmation par contraintes. L'approche avec vecteurs de bits le permet et la combinaison avec la réduction de l'espace de recherche par l'étude empirique des métriques permet l'analyse de très grands programmes. La figure 5.6 représente les temps d'identification de l'approche avec vecteurs de bits avec (courbe continue) et sans réduction de l'espace de recherche (courbe pointillée) sur plusieurs programmes de différentes tailles allant de 116 à 910 classes avec le motif Composite. L'approximation de type 1 seulement est prise en compte. Les calculs ont été effectués sur un AMD Athlon 64 bits à 3Ghz. Le tableau 5.5 donne la liste des programmes utilisés avec une courte description. L'analyse de Xerces v2.7.0, programme de 910 classes, nécessite moins de 10 minutes avec l'approche avec vecteurs de bits et l'étude des métriques.

Programmes	Tailles	Descriptions
HolubSQL v1.0	116	Interpréteur SQL
JavaMonopoly v1.4	148	Jeu de Monopoly
Eteria IRC Client 20010201	166	Client IRC
JUnit v3.7	209	*Framework* de tests unitaires
JHotDraw v5.1	261	Programme de dessins vectoriels
Trove v1.1b5	296	Bibliothèque de classes pour Tea
Gantt Project v1.10.2	489	Application de planification de projets
PCGen v5.0	645	Générateur de personnages de jeux de rôles
JRefactory v2.6.24	742	Outil de restructurations
Paquetage *com* de Azureus 2.3.0.6	743	Client BitTorrent
Xerces v2.7.0	910	Parseur XML

Tab. 5.5 – Description et taille des programmes analysés

Le tableau 5.6 montre le nombre d'occurrences obtenues lors de l'analyse de ces programmes avec l'approche avec vecteurs de bits. Les nouveaux programmes n'ont pas été analysés manuellement. Donc, la précision et le rappel des résultats ne peuvent pas être analysés mais, en considérant que les métriques n'éliminent pas

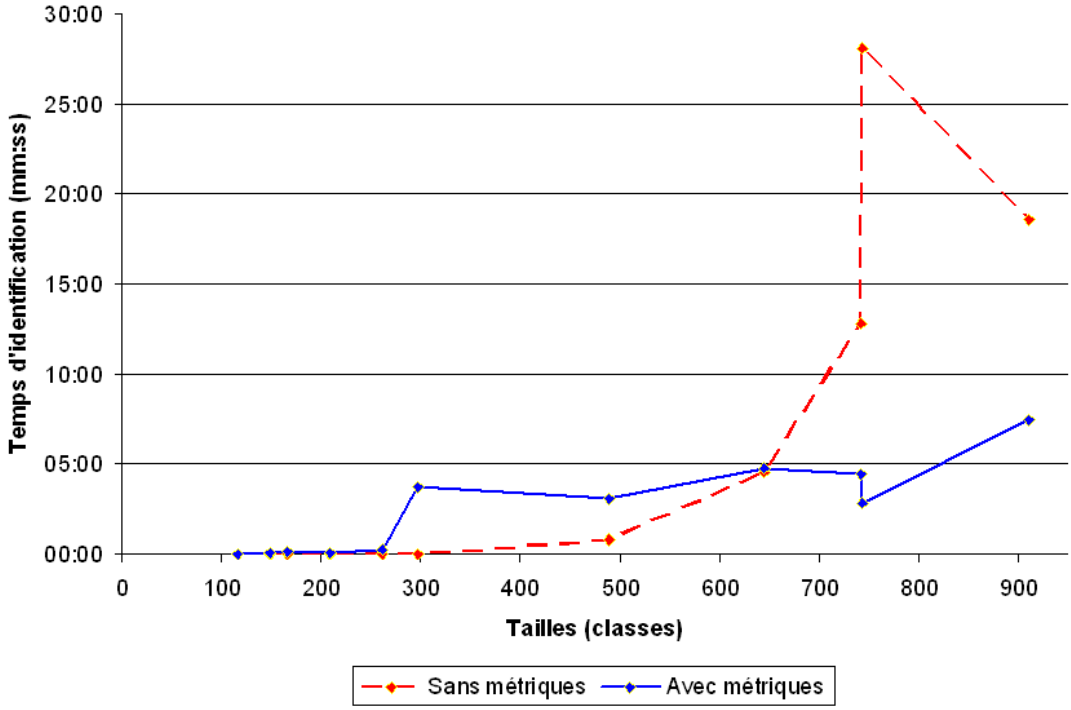


FIG. 5.6 – Temps d'identification avec et sans métriques selon la taille des programmes

Programmes	Sans métriques	Avec métriques
HOLUBSQL V1.0	443/8	0/0
JAVAMONOPOLY V1.4	363/179	0/0
ETERIA IRC CLIENT 20010201	896/24	0/0
JUNIT V3.7	359/628	0/518
JHOTDRAW V5.1	768/600	0/13
TROVE V1.1B5	0/0	0/0
GANTT PROJECT V1.10.2	7441/354	0/1
PCGEN V5.0	15253/3107	0/420
JREFACTORY V2.6.24	29849/8680	0/119
PAQUETAGE *com* DE AZUREUS 2.3.0.6	61167/130	0/6
XERCES V2.7.0	41715/4926	0/4012

TAB. 5.6 – Nombre d'occurrences du motif Composite identifiées avec et sans les métriques

de bonnes occurrences, la précision des résultats est grandement améliorée.

L'étude de cas montre l'efficacité de l'approche avec vecteurs de bits par rapport aux deux autres approches. Plusieurs ordres de magnitude séparent la performance de notre algorithme avec celui de la programmation par contraintes. L'étude montre également que l'analyse de grands programmes est possible relativement rapidement.

5.3 Discussions

L'étude de cas nous a permis de constater l'efficacité de notre algorithme mais aussi de mieux le comprendre et de connaître ses limites.

Corrélation entre les temps d'identification et la taille du programme analysé. Il est intéressant de constater dans la figure 5.6 que les temps d'identification ne dépendent pas uniquement de la taille du programme analysé. Par exemple, l'analyse de Trove v1.1b5 est plus rapide que celle de JUnit v3.7 qui est pourtant de plus petite taille. Le temps d'identification possède un coefficient de corrélation plus grand avec le nombre de relations dans le programme du type le plus rare présent dans le motif, par exemple, le nombre de relation d'*inheritance* ou d'*association+agrégation+composition* lors de la recherche du motif Composite avec l'approximation de type 1. Le type de relation que l'on retrouve le moins dans le programme est le premier à être lu dans la chaîne du motif. Plus le nombre d'occurrences potentielles est élevé en début d'analyse, plus le temps d'identification est élevé. Comme le nombre de relations de type *association* est généralement très grand dans un programme de grande taille, l'approximation de type 1 ralentit l'analyse. L'analyse du paquetage *com* d'Azureus 2.3.0.6, avec l'approximation

de type 1 permettant à la relation de *composition* d'être remplacée seulement par une relation d'*agrégation* et non d'*association*, diminue le temps d'identification d'environ 28 minutes à environ 45 secondes. Cette corrélation explique les temps d'identification de TROVE v1.1b5 puisqu'il s'agit d'une bibliothèque de classes contenant très peu de relations de type *association*.

Graphes de programmes non connexes. Lors de l'étude de cas, les chaînes de caractères représentant les programmes DRJAVA v20060127-2145 et GANTT PROJECT v1.10.2 n'ont pu être générées. Le problème vient du fait que les graphes de ces programmes ne sont pas connexes, une des conditions pour trouver un cycle eulérien. Afin de résoudre ce problème, toutes les classes ne possédant aucune relation avec d'autres classes sont retirées du graphe. Cette simple transformation du graphe permet l'analyse de GANTT PROJECT v1.10.2. Par contre, elle ne permet pas de rendre le graphe de DRJAVA v20060127-2145 connexe. Une solution serait de lancer l'identification sur chacun des sous-graphes connexes du programme.

Temps de construction des chaînes de caractères. La construction de chaînes représentant de petits programmes se fait très rapidement mais l'algorithme résolvant le problème du postier chinois peut être très long pour les très gros graphes. La construction des chaînes de programmes de moins de 500 classes se fait généralement en moins de 5 minutes mais peut prendre jusqu'à 1 heure pour des programmes d'environ 1 000 classes. Comme l'étape de génération de la chaîne ne doit être faite qu'une seule fois, ceci ne représente pas un problème. Il est toutefois possible de réduire ce temps de calcul en modifiant l'algorithme résolvant le problème du postier chinois pour permettre la présence de cycles. La super-chaîne obtenue avec cette

modification n'est plus la plus petite possible et les temps d'identification en sont affectés. Par exemple, le temps requis pour construire la chaîne du paquetage *com* d'AZUREUS 2.3.0.6 passe de 65 minutes à 7 minutes pour des temps d'identification avec métriques du motif Composite de 2 minutes et 50 secondes contre 4 minutes et 10 secondes.

CHAPITRE 6

CONCLUSION

Le travail de recherche présenté dans ce mémoire apporte une solution au problème de performance des outils de détection de motifs de conception dans des systèmes orientés objets en utilisant des adaptations d'algorithmes de comparaisons et d'alignements de chaînes de caractères de bio-informatique. Les algorithmes développés peuvent analyser des programmes de grandes tailles et l'outil (EPI) peut être utilisé par les mainteneurs dans leurs tâches quotidiennes et ainsi améliorer leur compréhension d'un programme puisque les analyses sont rapides.

Nous avons étudié les principales techniques de détection de motifs de conception utilisées jusqu'à maintenant et constaté leur limite en terme de performance. Toutefois, l'étude empirique des métriques pour réduire l'espace de recherche est une approche très intéressante pour réduire les temps d'identification mais aussi pour améliorer la précision. Nous avons donc combiné nos algorithmes avec l'étude des métriques afin d'obtenir un outil efficace et précis. Étant donné les similitudes entre le problème de détection de motifs de conception dans un programme et ceux de recherche exacte et approchée en bio-informatique, plusieurs des algorithmes résolvant ces problèmes ont été introduits. Nous avons également présenté une méthode permettant de construire des représentations sous la forme de chaînes de caractères des programmes et des motifs de conception. Cette méthode est inspirée d'algorithmes d'assemblages utilisés pour le séquençage en bio-informatique. Les différentes approximations que doit prendre en compte un outil de détection ont ensuite été discutées ainsi que trois approches pour l'identification de micro-

architectures similaires aux motifs de conception, la programmation dynamique, la simulation d'automates et les vecteurs de bits. Leurs avantages et leurs limites ont été discutées et deux solutions ont été proposées pour améliorer la précision des résultats, l'étude empirique des métriques et l'ajout de relations de type *ignorance* dans les motifs de conception. Finalement, l'implémentation de EPI a été présentée avec une étude de cas. Nous avons comparé dans cette étude de cas notre approche vectorielle à deux approches utilisant la programmation par contraintes avec explications, l'une avec réduction de l'espace de recherche par l'étude des métriques et l'autre sans cette réduction. Nous avons démontré par cette comparaison, la bonne performance de nos algorithmes mais aussi une qualité d'identification au moins aussi bonne.

Perspectives

La précision des résultats est grandement améliorée avec l'étude empirique des métriques et par l'ajout de relation d'*ignorance*. Cependant, elle pourrait l'être davantage en ajoutant de l'information dynamique dans les chaînes de caractères. L'analyse plus détaillée du comportement d'un programme permettrait la détection avec plus de précision de micro-architectures similaires aux motifs des patrons de conception comportementaux et créationnels.

Le problème de connectivité des graphes de programmes doit aussi être réglé, soit en identifiant leurs sous-graphes connexes et en lançant plusieurs analyses ou en ajoutant des relations *dummyRelationship* afin de les rendre connexes. Il serait aussi intéressant de travailler sur le métamodèle PADL pour permettre la différenciation entre les relations de type *composition* et de type *agrégation*.

Finalement, les algorithmes de bio-informatique présentés pourraient également servir à résoudre d'autres problèmes en génie logiciel, leur efficacité étant leur plus grand avantage. Par exemple, l'étude de l'évolution de programmes pourrait être faite en essayant d'aligner les différentes chaînes de caractères représentant les différentes versions du programme. Les ajouts et les suppressions d'entités et relations seraient alors rapidement repérables.

Création de la thèse

Afin de réaliser cette thèse, plusieurs outils ont été utilisés. Premièrement, l'écriture a été faite en LaTeX avec l'éditeur WinEdt[1]. Les diagrammes de classes ont été conçus avec MagicDraw UML[2] et les automates avec Gastex[3]. Microsoft Visio et Excel ont également été utilisés pour la création de graphes et diagrammes.

[1]http ://www.winedt.com/
[2]http ://www.magicdraw.com/
[3]http ://www.lsv.ens-cachan.fr/ gastin/gastex/gastex.html

BIBLIOGRAPHIE

[AFC98] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. Design pattern recovery in object-oriented software. In Scott Tilley and Giuseppe Visaggio, editors, *proceedings of the* 6^{th} *International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*, volume 2 of *Center for Environmental Structure Series*. Oxford University Press, New York, NY, 1977.

[Bac98] Bouazza Bachar. Détection des patrons de conception dans des systèmes orientés objets. Master's thesis, Université de Montréal, août 1998.

[BM77] Robert Stephen Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10) :762–772, Oct 1977.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England, 1996.

[Bri00] Johan Brichau. Declarative meta programming for a language extensibility mechanism. Technical Report Vub-Prog-TR-00-09, Programming Technology Lab, Vrije Universiteit Brussel, March 2000.

[BYG92] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10) :74–82, Oct 1992.

[DLBC89] Radoje Drmanac, Ivan Labat, Ivan Brukner, and Radomir Crkvenjakov. Sequencing of megabase plus dna by hybridization theory of the

method. *Genomics*, 4 :114–128, 1989.

[Epp95] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In Kenneth Clarkson, editor, *proceedings of the* 6^{th} *annual Symposium On Discrete Algorithms*, pages 632–640. ACM Press, January 1995.

[GAA04] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships : Putting icing on the UML cake. In Doug C. Schmidt, editor, *proceedings of the* 19^{th} *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman And Company, New York, 1979.

[GJ01a] Yann-Gaël Guéhéneuc and Narendra Jussien. Quelques explications pour les patrons – Une application de la PPC avec explications pour l'identification de patrons de conception. In Bertrand Neveu, editor, *actes des* 7^{e} *Journées Nationales sur la résolution de Problèmes NP-Complets*, pages 111–122. ONERA, juin 2001.

[GJ01b] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In Christian Bessière, editor, *proceedings of the* 1^{st} *IJCAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.

[Glu61] Victor M. Gluskov Gluskov. The abstract theory of automata. *Russian Mathematical Surveys*, 16 :1–53, 1961.

[GSZ04] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In Eleni Stroulia and Andrea de Lucia, editors, *proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press, November 2004.

[Gué05] Yann-Gaël Guéhéneuc. Ptidej : Promoting patterns with patterns. In *proceedings of the* 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK, 1997.

[HHL02] Dirk Heuzeroth, Thomas Holl, and Welf Löwe. Combining static and dynamic analyses to detect interaction patterns. In Hartmut Ehrig, Bernd J. Krämer, and Atila Ertas, editors, *proceedings the* 6^{th} *world conference on Integrated Design and Process Technology*. Society for Design and Process Science, June 2002.

[Hol97] Jan Holub. Simulation of nfa in approximate string and sequence matching. In *Proceedings of the Prague Stringology Club Workshop '97*, pages 39–46, 1997.

[HZCD05] Heyuan Huang, Shen-Sheng Zhang, Jian Cao, and Yonghong Duan. A practical pattern recovery approach based on both structural and behavioral analysis. *Journal of Systems and Software*, 75(1-2) :69–87, 2005.

[JSZ97] Jens H. Jahnke, Wilhelm Schäfer, and Albert Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In Mehdi Jazayeri, editor, *proceedings of the* 6^{th} *European Software Engineering Conference*, pages 193–210. ACM Press, September 1997.

[Jus03] Narendra Jussien. Programmation par contraintes pour les technologies logicielles. In Gilles Muller, editor, *actes du colloque GEMSTIC*. Groupe des Ecoles des Mines, avril 2003.

[JZ97] Jens H. Jahnke and Albert Zündorf. Rewriting poor design patterns by good design patterns. In Serge Demeyer and Harald C. Gall, editors, *proceedings the* 1^{st} *ESEC/FSE workshop on Object-Oriented Reengineering*. Distributed Systems Group, Technical University of Vienna, September 1997. TUV-1841-97-10.

[KGH06] Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Efficient identification of design patterns with bit-vector algorithm. In Giuseppe Antonio di Lucca and Nicolas Gold, editors, *proceedings of the* 10^{th} *Conference on Software Maintenance and Reengineering*, pages 173–182. IEEE Computer Society Press, March 2006.

[KM95] J. D. Kececioglu and E. W. Myers. Combinatorial algorithms for dna sequence assembly. *Algorithmica*, 13 :7–51, 1995.

[Kos04] Jussi Koskinen. Software maintenance costs, September 2004. www.cs.jyu.fi/ koskinen/smcosts.htm.

[KP96] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In

Linda M. Wills and Ira Baxter, editors, *proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.

[KPS94] S. Rao Kosaraju, James K. Park, and Clifford Stein. Long tours and short superstrings. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science.*, pages 166–177, Santa Fe, NM, 1994. ACM Press.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBMJ*, 31 :249–260, 1987.

[KSRP99] Rudolf Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *ICSE '99 : Proceedings of the 21st international conference on Software engineering*, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[LS81] Bennet P. Lientz and E. Burton Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11) :763–769, November 1981.

[Mel95] Borivoj Melichar. Approximate string matching by finite automata. In V. Hlavac and R. Sara, editors, *Computer Analysis of Images and Patterns*, pages 342–349. Springer, Berlin, 1995.

[MHG06] Naouel Moha, Duc-Loc Huynh, and Yann-Gaël Guéhéneuc. Une taxonomie et un métamodèle pour la détection des défauts de conception. *actes du 12e colloque Langages et Modèles à Objets*, pages 201–216, mars 2006.

[MP70] James H. Morris and Vaughan Ronald Pratt. A Linear Pattern Matching Algorithm. Technical Report 40, Computing Center, University of California, Berkeley, 1970.

[NMW04] Jörg Niere, Matthias Meyer, and Lothar Wendehals. User-driven adaption in rule-based pattern recognition. Technical Report tr-ri-04-249, University of Paderborn, June 2004.

[NSW+02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*, pages 338–348, New York, NY, USA, 2002. ACM Press.

[NWW01] Jörg Niere, Jörg P. Wadsack, and Lothar Wendehals. Design pattern recovery based on source code analysis with fuzzy logic. Technical Report tr-ri-01-222, University of Paderborn, March 2001.

[PC00] Michael P. Plezbert and Ron K. Cytron. Recognition and verification of design patterns. Technical Report wustl-00-01, Washington University in St-Louis, 2000.

[QYW97] Alex Quilici, Quing Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. *journal of Automated Software Engineering*, 5(3) :347–372, July 1997.

[SK00] Mária Smolárová and Michal Kadlec. Dpfinder : A tool for automatic recognition of design patterns. In Tomas Hruška and Masa-Aki Hashimoto, editors, *JCKBSE '00 : Knowledge-Based Software Engineering*, pages 107–114. IOS Press Amsterdam, 2000.

[SS04] Konstantin Scheglov and John-Mason P. Shackelford. Eclipse profiler, September 2004. eclipsecolorer.sourceforge.net/index_profiler.html.

[SvG98] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of Java software. In Bill Scherlis, editor, *proceedings of* 5^{th} *international symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, November 1998.

[Thi03] Harold W. Thimbleby. The directed chinese postman problem. *Journal of Software – Practice and Experience*, 33(11) :1081–1096, September 2003.

[Tho68] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11 :419–422, 1968.

[TL00] Abdelmadjid Boudina Tayeb Lemlouma. Programmation Logique avec Contraintes (clp) : Etude et application au puzzle send plus more equal to money. *Revue d'Informatique Scientifique et Technique (RIST)*, 10(1-2) :63–85, 2000.

[Ukk85] Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6 :132–137, 1985.

[Wen03] Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA)*, pages 29–32, Portland, USA, May 2003.

[WF99] Ian H. Witten and Eibe Frank. *Data Mining : Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1^{st} edition, October 1999.

[WM92a] Sun Wu and Udi Manber. Agrep : A fast approximate pattern-matching tool. In *Proceedings of the Winter 1992 USENIX Conference*, pages 153–162, San Francisco, California, 1992.

[WM92b] Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10) :83–91, Oct 1992.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *proceedings of the* 26^{th} *conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.