Extraction d’une architecture logicielle à base de composants depuis un système orienté objet. Une aproche par exploration

Sylvain Chardigny

▶ **To cite this version:**

Sylvain Chardigny. Extraction d’une architecture logicielle à base de composants depuis un système orienté objet. Une aproche par exploration. Génie logiciel [cs.SE]. Université de Nantes, 2009. Français. <tel-00456367>

HAL Id: tel-00456367

https://tel.archives-ouvertes.fr/tel-00456367

Submitted on 14 Feb 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Nantes **École des Mines de Douai**

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2010

N° attribué par la bibliothèque

Extraction d'une architecture logicielle à base de composants depuis un système orienté objet

Une approche par exploration

THÈSE DE DOCTORAT

Discipline : INFORMATIQUE

Spécialité : INFORMATIQUE

Présentée
et soutenue publiquement par

Sylvain CHARDIGNY

Le 23 octobre 2009 à l'UFR Sciences & Techniques, Université de Nantes,
devant le jury ci-dessous

Président	:	Pr. Jean BEZIVIN	Université de Nantes
Rapporteurs	:	Pr. Franck BARBIER, Professeur	Université de Pau
		Pr. Houari SAHRAOUI, Professeur	Université de Montréal
Examinateurs	:	Pr. Marianne HUCHARD, Professeur	Université de Montpellier II
		Pr. Alain APRIL, Professeur	Université du Québec
		Dr. Abdelhak-Djamel SERIAI, Maître de conférence	Université de Montpellier II
		Dr. Dalila TAMZALIT, Maître de conférence	Université de Nantes
		Pr. Mourad OUSSALAH, Professeur	Université de Nantes

Directeur de thèse : Pr. Mourad OUSSALAH

Encadrant de thèse : Dr. Abdelhak-Djamel SERIAI et Dr. Dalila TAMZALIT

Laboratoire: LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

N° ED 503-064

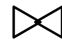
Extraction d'une architecture logicielle à base de composants depuis un système orienté objet

Une approche par exploration

Component-based software architecture recovery from an object oriented system

A search-based approach

Sylvain Chardigny



favet neptunus eunti

Université de Nantes

Sylvain CHARDIGNY

Extraction d'une architecture logicielle à base de composants depuis un système orienté objet

Une approche par exploration

IV+XIV+236 p.

Ce document a été préparé avec LATEX2e et la classe these-LINA version v. 2.7 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe these-LINA est disponible à l'adresse : http://login.irin.sciences.univ-nantes.fr/.

Cette classe est conforme aux recommandations du ministère de l'éducation nationale, de l'enseignement supérieur et de la recherche (circulaire n° 05-094 du 29 mars 2005), de l'Université de Nantes, de l'école doctorale « Sciences et Technologies de l'Information et des Matériaux » (ED-STIM), et respecte les normes de l'association française de normalisation (AFNOR) suivantes :

– AFNOR NF Z41-006 (octobre 1983)
  Présentation des thèses et documents assimilés ;
– AFNOR NF Z44-005 (décembre 1987)
  Documentation – Références bibliographiques – Contenu, forme et structure ;
– AFNOR NF Z44-005-2/ISO NF 690-2 (février 1998)
  Information et documentation – Références bibliographiques – Partie 2 : documents électroniques, documents complets ou parties de documents.

Impression : these.tex – 08/01/2010 – 16:06.

Révision pour la classe : these-LINA.cls,v 2.7 2006/09/12 17:18:53 mancheron Exp

A mon père,
pour m'avoir toujours envoyé chercher
ce qu'il aurait pu se contenter de me donner.

Résumé

La modélisation et la représentation des architectures logicielles sont devenues une des phases principales du processus de développement des systèmes complexes. En effet, la représentation de l'architecture fournit de nombreux avantages pendant tout le cycle de vie du logiciel. Cependant, pour beaucoup de systèmes existants, aucune représentation fiable de leurs architectures n'est disponible. Afin de pallier cette absence, source de nombreuses difficultés principalement lors des phases de maintenance et d'évolution, nous proposons dans cette thèse une approche, appelée ROMANTIC, visant à extraire une architecture à base de composants à partir d'un système orienté objet existant. L'idée première de cette approche est de proposer un processus quasi-automatique d'identification d'architectures en formulant le problème comme un problème d'optimisation et en le résolvant au moyen de méta-heuristiques. Ces dernières explorent l'espace composé des architectures pouvant être abstraites du système en utilisant la sémantique et la qualité architecturale pour sélectionner les meilleures solutions. Le processus s'appuie également sur l'architecture intentionnelle du système, à travers l'utilisation de la documentation et des recommandations de l'architecte.

Mots-clés : architecture logicielle, composant logiciel, connecteur, extraction, sémantique architecturale, qualité, architecture intentionnelle, méta-heuristiques.

Abstract

Software architecture modeling and representation are a main phase of the development process of complex systems. In fact, software architecture representation provides many advantages during all phases of software life cycle. Nevertheless, for many systems, like legacy or eroded ones, there is no available representation of their architectures. In order to benefit from this representation, we propose, in this thesis, an approach called ROMANTIC which focuses on recovering a component-based architecture from an existing object-oriented system. This recover is a balancing problem of competing constraints which aims at obtaining the best architecture that can be abstracted from a system. Consequently, the main idea of this approach is to propose a quasi-automatic process of architecture identification by formulating it as a search-based problem. The latter acts on the space composed of all possible architectures abstracting the object-oriented system and use the architectural semantic and quality to choose the best solution. The process uses the intentional system architecture by means of the documentation and the architect's recommendations.

Keywords: software architecture, software component, connector, recovery, architectural semantic, quality, intentional architecture, search-based software engineering.

Remerciements

Mes remerciements s'adressent d'abord à l'ensemble des membres de mon jury. Je suis très honoré que M. Houari Sahraoui, professeur à l'université de Montréal et M. Franck Barbier, professeur à l'université de Pau, aient accepté d'être rapporteurs de ma thèse. Je remercie également Mme Marianne Huchard, professeur à l'université de Montpellier, M. Alain April, professeur à l'université du Québec et M. Jean Bezivin, professeur à l'université de Nantes Atlantique d'avoir bien voulu participer à ce jury.

Je remercie mon directeur de thèse, Mourad Oussalah et mes encadrants Abdelhak-Djamel Seriai et Dalila Tamzalit pour leur confiance et les précieux conseils qu'ils m'ont dispensés tout au long de cette thèse. Les discussions que nous avons eu m'ont permis d'avancer dans cette thèse et dans ma perception de notre travail.

Merci en particulier à Djamel, pour ces heures passées à débattre, du fond comme de la forme. J'espére qu'il verra avec ce manuscrit que j'ai retenu certaines choses. J'espére aussi qu'il ne marquera pas la fin de ces débats parfois passionnés.

Je tiens à remercier Phillippe Hasbroucq pour m'avoir accueilli dans le département Informatique et Automatique de l'école des mines de Douai. Je remercie également très chaleureusement Noury Bouraquadi et Stéphane Lecœuche pour leurs conseils et leurs soutiens durant ces trois années, ainsi que Muriel et christine pour leur patience et leur aide dans les méandres de l'administration. Je pense aussi à tout le personnel que j'ai côtoyé pendant ma thèse (enseignants-chercheurs, techniciens, ...).

Je n'oublierai pas les gens que j'ai rencontré dans « le grand nord » et qui m'ont accompagné tout au long de cette thèse : Tudor, Tuan, Misagh, Mickaël, Peter, Vincent, Fangfe (*sic*) et les autres. Merci donc pour ces soirées « studieuses » faites de joie et d'amitié.

Je dois un merci particulier à mes amis Guillaume et Gautier. Merci pour m'avoir accueilli, conseillé, accompagné, réprimandé *etc.*. Ce manuscrit ne serait pas le même sans vos précieuses relectures. Vos encouragements et votre soutien ont été ma lumière d'Eärendil pendant toute la rédaction.

Je pense également aux enseignants que j'ai eu depuis la primaire jusqu'au DEA. J'ai rencontré parmi eux beaucoup de gens passionnés et passionnant qui m'ont encouragé. En particulier, je remercie M. De La Gontrie et M. Brunet. Ce qu'ils m'ont appris m'a donné l'envie de savoir, de comprendre et de chercher. Ils m'ont donné l'envie de faire ce que je fais aujourd'hui.

Enfin, je remercie ma famille. Merci à mes parents et à ma soeur pour leur aide morale et financière ; leurs encouragements, leurs présences m'ont conduit ici.

Et surtout merci à toi ma Claire, pour m'avoir accompagné le long de ce chemin. Sans ta présence et ton caractére, jamais je n'aurai pu mener cette tâche à bien. Ca y est, j'y suis, à mon tour.

Sommaire

Introduction xi

— Corps du document —

I Contexte et travaux connexes

1 Architectures logicielles 3
2 Extraction d'architectures logicielles 31

II La démarche ROMANTIC pour l'extraction d'architectures

3 Problématique et approche de ROMANTIC 53
4 Fondement du processus d'extraction 81
5 Guider le processus d'extraction 117

III Mise en pratique et validation de ROMANTIC

6 Algorithmes pour l'exploration 143
7 Validation et évaluation de l'approche ROMANTIC 179

IV Conclusion et perspectives

Conclusion 201
Perspectives 205

— Pages annexées —

Bibliographie 209
Liste des tableaux 223
Liste des figures 225
Liste des exemples 229
Table des matières 231

Introduction

Problématique et objectifs de la thèse

Face à la croissance fulgurante de la taille et de la complexité des systèmes informatiques, les architectures logicielles s'imposent comme un allié précieux aussi bien pour la conception que la maintenance de ces systèmes. Cette vue abstraite des systèmes est ainsi devenue, pendant la dernière décennie, un sous-domaine central du génie logiciel [48].

Les recherches ont permis de préciser la définition des architectures et des éléments architecturaux : les composants, les connecteurs et la configuration. Elles ont également démontré les avantages et l'étendue des utilisations possibles de ces architectures logicielles [114]. Ces avantages ont été décrits, aussi bien dans le cadre de la conception [47], que dans celui de la maintenance [74]. Ainsi, l'architecture permet, par exemple, de faciliter la compréhension du système ou de l'analyser.

L' architecture logicielle d'un système est ainsi devenue un élément essentiel de chaque phase du cycle de vie du logiciel [16]. Elle est utilisée pour concevoir le système, évaluer la qualité d'un système, le documenter ou encore pour orienter les phases de maintenance.

Or, beaucoup de systèmes existants ne disposent pas d'une représentation fidèle de leur architecture. En effet, le système peut avoir été conçu sans représentation de son architecture, comme dans le cas de certains systèmes patrimoniaux. Pour d'autres systèmes, la représentation disponible est décalée par rapport à l'implémentation du système à cause des écarts entre l'architecture prévue et implémentée puis du manque de synchronisation entre la documentation et l'implémentation.

Dans tous les cas, ce manque de représentation peut annuler tous les avantages de l'architecture pour la maintenance et dégrader encore plus les chances de succès des opérations de maintenance. En effet, si l'architecture utilisée ne correspond pas à celle du système, le recours à des techniques de maintenance l'utilisant peut conduire à une mauvaise réalisation ou à des effets de bord inattendus.

Pour pallier ce problème, de nombreuses approches visent à extraire l'architecture logicielle d'un système existant. Ces approches étudient le code source de l'application pour identifier les éléments constituant une entité architecturale.

Cependant, l'émergence rapide des architectures logicielles a fait apparaitre certaines difficultés qui limitent la portée des travaux d'extraction. En effet, les architectures logicielles sont un sujet de recherche qui est apparu parallèlement dans différents domaines. Les différentes définitions sont donc fortement influencées par le domaine de leurs auteurs et convergent lentement. Ce flou concernant les définitions des architectures logicielles et des éléments architecturaux a un fort impact sur les travaux d'extraction. La plupart de ces approches utilisent une définition spécifique de l'architecture et il convient alors d'utiliser une approche d'extraction compatible avec cette définition de l'architecture logicielle.

Les travaux d'extraction d'architectures se heurtent aussi au problème de l'architecture intentionnelle. Cette architecture est celle du système tel qu'il est imaginé par les architectes et les concepteurs. En effet, tout système comporte une part d'informations qui ne sont pas documentées. Ces informations concernent les intentions et les réflexions des concepteurs du système ou des architectes chargés de sa maintenance et illustrent un système légèrement différent de l'implémentation existante. De par sa

nature, cette architecture intentionnelle est très précieuse pour réaliser l'extraction de l'architecture logicielle d'un système mais elle est complexe à obtenir. Par conséquent, les travaux d'extraction existant ne permettent pas de concilier l'utilisation de l'architecture intentionnelle avec un bon niveau d'automatisation. Ceux qui choisissent l'architecture intentionnelle bénéficient donc de précieuses informations mais nécessitent des experts pour réaliser l'extraction [72]. Au contraire, ceux qui choisissent l'automatisation ne nécessitent pas d'experts mais ne permettent pas de les utiliser s'ils sont disponibles [5].

En se basant sur ces considérations, notre objectif, dans cette thèse, est d'étudier la problématique de l'extraction d'architectures logicielles. Nous proposons d'étudier cette problématique pour obtenir une approche basée sur une définition précise de l'architecture et utilisant de manière semi-automatique l'ensemble des informations disponibles sur le système cible.

Plan du mémoire

Ce manuscrit contient trois parties. La première présente le contexte de notre travail et une synthèse des travaux connexes. La deuxième partie décrit notre proposition. Enfin la troisième partie propose une validation de notre approche à travers une mise en pratique.

Partie 1 : contexte et travaux connexes. La première partie de ce manuscrit expose la terminologie utilisée dans ce travail. Elle présente également le contexte de l'extraction d'architectures en proposant une comparaison des approches existantes. Cette partie est organisée en deux chapitres :

- **chapitre 1 : architectures logicielles.**
  Ce chapitre permet de poser la terminologie associée aux architectures logicielles et de décrire les travaux existants sur ce thème. Il est organisé en trois sections. L'objectif de la première section est de présenter la terminologie commune aux domaines utilisant l'architecture logicielle. Pour cela, nous présentons les définitions couramment admises de l'architecture logicielle et des éléments architecturaux : composants, connecteurs et configuration. Nous abordons également la terminologie des styles architecturaux et revenons sur les avantages d'utiliser une telle terminologie. Dans la deuxième section, nous abordons les différentes terminologies de l'architecture logicielle en fonction de la communauté et de l'époque. Enfin, dans la troisième section, nous proposons une synthèse des travaux utilisant l'architecture logicielle pour mesurer la qualité d'un système logiciel. En effet, cette utilisation de l'architecture logicielle est l'une des plus répandues et surtout la plus formalisée alors que les autres utilisations reposent sur des méthodes *ad-hoc* ;
- **chapitre 2 : extraction d'architectures logicielles.**
  Le deuxième chapitre introduit la problématique de l'extraction d'architectures et présente les travaux existants sur ce thème. Il est organisé en trois sections. L'objectif de la première section est de montrer l'importance de l'architecture durant la maintenance. Pour cela, nous présentons une terminologie du domaine de la maintenance et les apports de l'architecture logicielle à ce domaine. Ensuite, nous étudions les causes du manque de représentation correcte de l'architecture et nous présentons les conséquences que ces manques peuvent avoir sur la localisation des défauts ou des cibles de la maintenance. La deuxième section propose une présentation des principes de l'extraction d'architectures logicielles. Pour cela, nous détaillons les modèles existants des processus d'extraction puis nous proposons un modèle plus générique pour pouvoir prendre en compte l'ensemble des approches d'extraction. Enfin, la dernière section présente notre cadre de comparaison des approches d'extraction d'architectures logicielles. Ce cadre de comparaison repose sur le mo-

dèle de l'extraction présenté dans la section précédente et utilise en plus d'autres paramètres tel que les techniques et les informations utilisées.

Partie 2 : la démarche ROMANTIC pour l'extraction. La deuxième partie expose notre approche d'extraction d'architectures logicielles à partir d'un système orienté objet. Cette partie est organisée en trois chapitres :

- **chapitre 3 : problématique et approche dans ROMANTIC.**
  Ce chapitre présente les motivations et les principes de notre approche d'extraction. Il est organisé en quatre sections. L'objectif de la première section est de poser le problème de l'extraction en s'appuyant sur l'état de l'art présenté dans le chapitre 2 pour motiver notre approche. Dans la deuxième section, nous présentons les principes de notre approche d'extraction, c'est-à-dire une approche méta-heuristique d'exploration de l'espace des architectures possibles reposant sur une évaluation de leurs qualités. La troisième section contient la modélisation du problème d'extraction en terme de problème d'exploration. Pour cela, nous présentons notre modèle du code source objet ainsi que celui de correspondance entre le paradigme objet et les concepts architecturaux. Nous montrons également les principes fondamentaux de l'exploration que sont l'espace des solutions et de recherche. La dernière section présente les guides de l'exploration. Nous détaillons les différents éléments qui nous permettent de diriger l'exploration vers l'architecture la plus pertinente suivant la sémantique des architectures logicielles, la qualité logicielle ou l'appréciation des architectes ;
- **chapitre 4 : fondement du processus d'extraction.**
  Nous étudions dans ce chapitre, les guides de l'extraction qui nous permettent de définir ce qu'est une bonne solution. Pour cela, nous proposons dans ce chapitre une fonction d'évaluation de la qualité d'un élément de l'espace des solutions. Cette évaluation repose sur deux axes qui sont la sémantique architecturale et la qualité architecturale. Le chapitre est organisé en quatre sections. La première décrit le méta-modèle de mesure qui nous permet de définir notre fonction objectif. Ce méta-modèle repose sur la norme ISO 9126 [44], et décompose les éléments mesurés en un ensemble de caractéristiques, sous-caractéristiques et propriétés mesurables. Dans la deuxième section, nous exposons le modèle de mesure de la validité sémantique de l'architecture. Ce modèle permet d'évaluer le respect par une solution de la sémantique associée aux notions d'architecture logicielle, de composant et de connecteur. La troisième section présente le modèle de mesure de la qualité architecturale. Ce modèle évalue les caractéristiques de maintenabilité et de fiabilité d'une architecture et permet de sélectionner la meilleure solution du point de vue de ces deux caractéristiques. Enfin, la quatrième section expose la fonction objectif de notre processus d'exploration. Cette fonction repose sur les deux modèles de mesures précédents et permet ainsi de sélectionner la meilleure architecture possible selon la sémantique et la qualité architecturales ;
- **chapitre 5 : guides du processus d'extraction.**
  Ce chapitre détaille les guides, identifiés dans le chapitre 3, qui ne sont pas utilisés pour définir la fonction objectif de l'exploration. Ce chapitre est organisé en trois sections. La première section décrit les guides disponibles et l'influence qu'ils ont sur le déroulement de l'exploration. Nous détaillons comment la documentation, les recommandations de l'architecte et le contexte de déploiement permettent d'obtenir une vue de l'architecture intentionnelle. Cette vue permet d'éliminer ou d'éviter les solutions qui ne sont pas pertinentes du point de vue de la documentation, l'architecte ou l'architecture matérielle. Dans la deuxième section, nous expliquons comment est extrait l'architecture intentionnelle à partir des différents types de documents et à partir des recommandations de l'architecte. La troisième section détaille la création d'un réseau de contraintes

hiérarchiques à partir des informations intentionnelles et explique l'impact de ce réseau sur le processus d'exploration.

Partie 3 : mise en pratique et validation de ROMANTIC. La troisième partie de ce manuscrit présente les algorithmes mettant en pratique ROMANIC ainsi qu'une validation de cette approche à travers un cas d'étude détaillé. Cette partie est organisée en deux chapitres :

- **chapitre 6 : algorithmes pour l'exploration.**
  Ce chapitre propose une mise en pratique de l'approche ROMANTIC que nous avons décrite dans la partie 2. Pour cela, il présente différents algorithmes qui instancient notre approche et permettent ainsi de réaliser l'extraction de l'architecture logicielle d'un système orienté objet. Ce chapitre est organisé en trois sections, chacune présentant un algorithme particulier. La première section décrit un processus basé sur un algorithme de regroupement hiérarchique [68]. Ce n'est pas une méta-heuristique, mais plutôt un algorithme test. En effet, cet algorithme, historiquement le premier que nous avons développé, nous a permis de tester les différents paramètres et étapes de notre approche sur un processus non-stochastique dont les résultats peuvent être facilement reproduits à l'identique. La deuxième section présente un processus d'extraction basé sur un algorithme de recuit simulé [78]. Le processus décrit alors est une méta-heuristique qui explore l'espace de recherche en utilisant les guides et la fonction objectif définie dans la partie 2. Cet algorithme consiste essentiellement en une série de tentatives de modifications sur les solutions et sur la conservation des changements qui améliorent le résultat. La particularité du recuit est d'accepter certains changements qui provoquent une réduction de la qualité, afin d'éviter à l'exploration d'être bloquée dans un maximum local. Nous détaillons, dans la troisième section, un autre processus d'extraction basé sur les algorithmes génétiques. Ces algorithmes, qui visent à simuler l'évolution d'une population soumise à une sélection, sont des méta-heuristiques efficaces et, comme le recuit utilisent tous les aspects de notre approche décrits dans la partie 2 ;
- **chapitre 7 : cas d'étude et comparaison des algorithmes.**
  Le chapitre 7 propose une validation de notre approche à travers un cas d'étude basé sur les algorithmes détaillés dans le chapitre précédent. Il décrit également une comparaison des algorithmes afin de déterminer leur efficacité relative. Le chapitre est organisé en trois sections. La première section expose le contexte du cas d'étude. Nous donnons une description de deux logiciels utilisés pour valider notre approche : Jigsaw et ArgoUML, ainsi que les paramètres utilisés pour chacun de nos algorithmes. Les deux autres sections décrivent les deux étapes de notre cas d'étude. Chacune décrit la méthode et la réalisation de cette étape sur dans le contexte décrit dans la première section. Ainsi, la seconde section vise à valider les fondements théoriques de notre approche, alors que la troisième établit une comparaison entre les différents algorithmes que nous avons proposés.

Enfin, nous concluons ce manuscrit par un bilan de notre approche et une présentation des perspectives d'ouverture et d'extension de notre approche.

Partie I

Contexte et travaux connexes

CHAPITRE 1
Architectures logicielles

Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled
— Eoin WOODS.

Architecture logicielle — Composant — Connecteur — Configuration — Avantages de l'architecture — Mesure de la qualité logicielle

Les systèmes informatiques connaissent une croissance exponentielle aussi bien en terme de taille que de complexité. Pour concevoir ces systèmes de plus en plus complexes, les langages de programmation ont gagné en pouvoir d'expression et en abstraction. Des approches de maintenance ont également été proposées pour maintenir ces systèmes.

Les architectures logicielles constituent une autre solution, aussi bien pour la conception que la maintenance de ces systèmes informatiques complexes. En effet, en offrant une vue abstraite d'un système, cette représentation permet de gérer cette croissance fulgurante. A ce titre, les architectures ont connu un intérêt croissant pendant la dernière décennie, et sont ainsi devenues un sous-domaine central du génie logiciel [48]. Ces recherches ont permis de préciser la définition des architectures et des éléments architecturaux. Elles ont également démontré les avantages et l'étendue des utilisations possibles de ces architectures.

Malgré ce développement, l'architecture reste une notion émergente. En effet, le concept d'architecture est apparu de manière indépendante dans différents domaines du génie logiciel. Par la suite, les différentes notions, issues de chacun des domaines, se sont progressivement rapprochées. Ces rapprochements sont le fruit d'une collaboration entre différentes communautés du génie logiciel telle que la communauté objet ou réingénierie. Ils ont permis d'obtenir une définition relativement homogène de la notion d'architecture.

Cependant, ces rapprochements ne sont pas encore achevés. Ainsi, il existe encore une forte disparité dans la définition des éléments architecturaux qui composent l'architecture. Selon les communautés, les éléments architecturaux ne sont pas tous considérés au même niveau et surtout ne font pas référence exactement aux mêmes concepts.

Dans la suite de ce chapitre, nous présentons la terminologie et les concepts associés à la notion d'architecture. Pour cela, nous décrivons les différents éléments architecturaux et nous présentons les utilisations de l'architecture logicielle. Nous étudions ensuite les divergences qui existent sur cette notion selon le domaine et la période d'utilisation. Enfin, nous terminons en présentant les apports de l'architecture logicielle dans l'évaluation de la qualité logicielle. Pour cela, nous proposons une classification des travaux utilisant les architectures logicielles pour évaluer la qualité d'un système logiciel.

1.1 Concepts et terminologie des architectures logicielles

Une définition couramment admise de l'architecture logicielle est celle de BASS, CLEMENTS et KAZMAN [11] (*cf.* Définition 1.1). Cette définition décrit l'architecture comme une abstraction constituée d'éléments logiciels et leurs relations. Cependant, elle ne décrit pas la nature des entités logicielles représentées dans l'architecture. Elle permet ainsi à chacun de choisir les entités et peut donc être utilisée dans toutes les communautés utilisant le concept d'architecture.

Définition 1.1. L'architecture logicielle d'un programme ou d'un système est la ou les structures du système, c'est-à-dire en particulier les éléments logiciels, les propriétés visibles extérieurement de ces éléments et leurs relations.

Une définition plus ancienne, proposée par PERRY et WOLF [102], décrit plus clairement les entités qui composent l'architecture (*cf.* Définition 1.2). Cette définition utilise pour les décrire le terme d'éléments architecturaux, que nous utiliserons par la suite.

Définition 1.2. ... l'architecture logicielle est un ensemble d'éléments architecturaux (ou, si vous préférez, de conception) qui ont une forme particulière.

Nous distinguons trois classes différentes d'éléments architecturaux :

- les éléments de calcul ;
- les éléments de données ; et
- les éléments de connexion.

Par la suite, la définition des éléments architecturaux a progressé et un consensus s'est formé sur leurs noms et leurs rôles dans l'architecture. On peut ainsi définir l'architecture selon la définition 1.3. Chaque élément architectural est ainsi nommé et leur rôle dans l'architecture est clairement défini. Il reste que la définition est encore suffisamment souple pour offrir une grande marge de manœuvre dans la description de éléments architecturaux.

Définition 1.3. L'architecture est une vue abstraite d'un système en terme d'éléments architecturaux. Ces éléments sont :

- les composants qui décrivent les fonctionnalités métier de l'application ;
- les connecteurs qui décrivent les communications et connexions entre les composants ;
- la configuration qui décrit la topologie des connexions entre composants et connecteurs.

Dans cette section, nous présentons les différents éléments architecturaux. Pour chacun, nous proposons une synthèse des points communs entre les définitions couramment utilisées. Ensuite, nous présentons la notion de style architectural qui constitue un autre élément essentiel d'une architecture. En effet, il définit un ensemble de contraintes ainsi qu'une sémantique et une terminologie sur les éléments architecturaux.

1.1.1 Les composants

Les composants sont les éléments architecturaux qui encapsulent la partie métier de l'architecture. Les définitions de cet élément sont nombreuses *. Néanmoins celle de SZYPERSKI [118] est la plus répandue dans la littérature (*cf.* Définition 1.4). Cette définition souligne les propriétés fondamentales

*Nous présentons une étude détaillée des définitions des composants logiciels dans le chapitre 4 lors de l'étude de la sémantique du concept d'architecture.

du composant. Elle présente également une partie des éléments externes représentant un composant : les interfaces fournies et requises.

Définition 1.4. Un composant logiciel est une unité de composition possédant des interfaces spécifiées par contrat et des dépendances contextuelles explicites. Un composant logiciel peut être déployé indépendamment et est sujet à la composition par un tiers.

Le consensus relatif autour de cette définition provient sans doute du fait qu'elle englobe les définitions des composants issues de différents domaines. Ainsi, elle permet de décrire aussi bien un composant abstrait tel que ceux présents dans les premières phases de conception qu'un composant exécutable selon un modèle tel que CCM (Component Corba Model) [55] ou COM (Component Object Model) [22], par exemple. Cette généricité est dûe à la fois au manque de précision sur les structures externes du composant et sur l'absence de description des structures internes du composant.

Au delà de cette définition, les composants ont bénéficié des rapprochements entre les différentes communautés utilisant l'architecture logicielle. La définition a, ainsi, progressivement évolué pour préciser les éléments communs aux différentes communautés. Grâce à cette évolution, nous pouvons compléter la définition des composants en décrivant plus précisément leurs structures externes et internes ainsi que leurs relations de composition.

1.1.1.1 Structure externe d'un composant

La structure externe d'un composant est généralement caractérisée par deux types d'éléments [100]. Les premiers sont les interfaces du composant. Elles sont la spécification des services fournis et requis par le composant. Les seconds sont les propriétés du composant. Elles servent à documenter l'architecture en décrivant les aspects relevant de la conception ou de l'analyse du composant.

Interfaces. Les interfaces sont la partie visible d'un composant (*cf.* Figure 1.1). Elles servent à déclarer les services fournis et requis par le composant et constituent, d'après la définition 1.5 proposée par CHEFROUR [27], la caractéristique majeure de celui-ci.

Définition 1.5. Un composant est une entité logicielle qui fournit un service particulier via une interface séparée de l'implantation mettant en œuvre ce service.

Une interface est composée de deux dimensions :

- **une dimension sémantique** : les interfaces décrivent la sémantique des fonctionnalités fournies et requises proposées par le composant. Ces fonctionnalités, appelées aussi services, déterminent le type de l'interface. Ainsi, si le service associé à une interface décrit le comportement fonctionnel du composant. c'est une interface fournie. Au contraire, si le service décrit les fonctionnalités dont le composant a besoin pour fonctionner, l'interface est une interface requise ;
- **une dimension structurelle** : en plus de décrire les fonctionnalités du composant, l'interface est le point d'interaction entre le composant et son environnement. Cet aspect structurel est parfois décrit comme une entité séparée, appelée port. C'est cette dimension qui est représentée lorsque l'on schématise un composant (*cf.* Figure 1.1).

Propriétés. Les propriétés des composants sont de trois types :

- **propriétés non fonctionnelles** : ces propriétés peuvent concerner la structure, le comportement ou les fonctionnalités du composant. Elles sont, par exemple, liées à la sécurité, la performance ou la portabilité du composant. Elles peuvent être configurables en fonction du contexte d'exécution

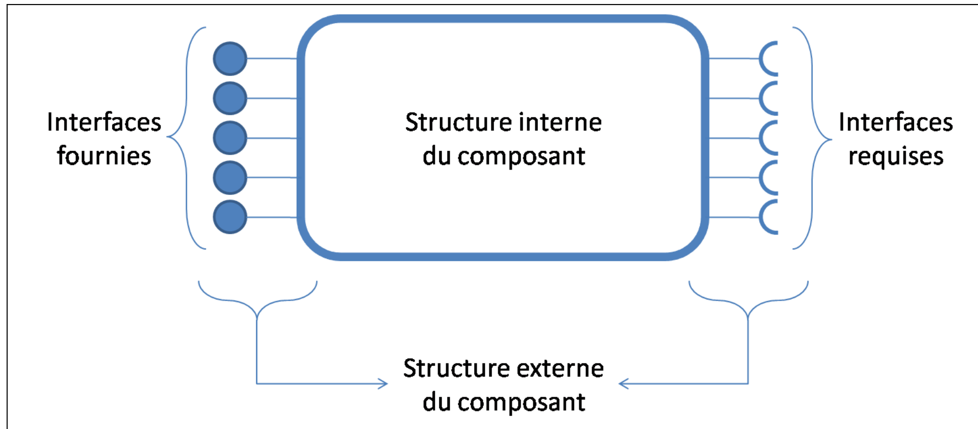


Figure 1.1 – Représentation d'un composant

particulier. Elles peuvent, entre autre, permettre de simuler le comportement d'un composant avant même son implémentation ;

- **contraintes** : ces propriétés définissent des contraintes sur les aspects fonctionnels ou non du composant. Ce sont des propriétés qui doivent être vérifiées pour que le système soit considéré comme cohérent ;
- **contrats** : ces propriétés portent sur les interfaces du composant. Les contrats portant sur les interfaces fournies définissent des contraintes et garantissent que les services de l'interface respectent ces contraintes. Les contrats portant sur les interfaces requises imposent aux composants fournisseurs un ensemble de contraintes qui doivent être vérifiées par les services fournis au travers de leurs contrats.

1.1.1.2 Structure interne d'un composant

Il existe deux types de structures internes des composants. D'abord, elle peut être constituée par une description ou une implémentation, dans un langage de programmation, des fonctionnalités du composant. Les composants possédant une telle structure interne sont des composants atomiques. Ils sont les blocs de base de l'architecture.

Le second type de structures internes est composé d'autres composants. Ces composants sont des composites constitués de composants internes. Ces composants internes peuvent eux aussi être atomiques ou composites. Comme le composant atomique, le composant composite dispose de ses propres interfaces. Les fonctionnalités proposées ou utilisées par ces interfaces peuvent alors être déléguées aux composants internes ou directement pris en charge par l'implémentation du composant composite. Pour réaliser cette délégation et pour interagir avec ses composants internes, le composant composite possède également des interfaces internes qui comme pour les externes peuvent être fournies ou requises.

Néanmoins, si la structure interne de tous les composants composites est constituée de composants, les composants composites peuvent être classés selon deux axes :

- **les interactions entre les composants internes** : soit les composants internes peuvent interagir directement entre eux, soit les composants internes n'interagissent qu'avec le composant composite. Dans ce dernier cas, le composite peut avoir à jouer le rôle de médiateur entre les différents composants internes ;
- **la délégation des interfaces** : soit le composant composite peut prendre en charge les interfaces,

soit il délègue ces interfaces aux composants internes à travers des ports spécifiques. Dans ce dernier cas, le composant composite constitue une enveloppe pour les services fournis par ses composants internes. Au contraire, dans le premier cas le composite propose des services supplémentaires par rapport à ses composants internes.

La figure 1.2 illustre les quatre cas possibles en fonction de ces deux axes.

1.1.1.3 Relations de compositions des composants

Les relations de compositions entre composants sont de deux types. Il existe d'abord des relations de compositions horizontales. Ceux sont les relations déterminées par les interfaces requises et les connecteurs. Ces relations permettent d'assembler les composants et les connecteurs. Elles n'imposent pas de contrainte particulière sur les éléments impliqués. A ce titre, ces relations sont les plus simples à utiliser pour assembler des composants provenant de sources diverses.

Les autres relations sont dites verticales. Ces relations représentent les relations entre un composant composite et ses sous-composants. Elles possèdent une sémantique différente de celle de la composition horizontale et constituent la différence majeure entre les composants composites et les composants atomiques. En effet, contrairement aux relations de compositions horizontales, la composition verticale impose des contraintes strictes sur les différentes parties impliquées. Ces contraintes varient en fonction du type de la relation qui peut être plus ou moins forte [14]. Par exemple, une relation de composition verticale forte implique la simultanéité dans la création et la destruction du composites et des sous-composants. Ces contraintes sont associées à la sémantique de la relation de composition verticale, mais elles ne sont pas représentées directement dans l'architecture. Ainsi, ces relations apparaissent, comme les relations horizontales, à travers des interfaces requises ou fournies qui relient le composite et les sous-composants.

1.1.2 Les connecteurs

Les connecteurs constituent un élément architectural au même titre que les composants. Ainsi, ce sont des entités de premier plan dans une architecture. Cette considération identique pour les composants et les connecteurs est l'atout principal des architectures. En effet, la distinction entre les aspects métiers et ceux de communication se fait plus facilement en considérant de la même manière les deux entités.

A la différence des composants, les définitions des connecteurs sont peu nombreuses et relativement convergentes[†]. Par exemple, SHAW et GARLAN [114] présentent les connecteurs selon la définition 1.6.

Définition 1.6. Les connecteurs gèrent les interactions entre les composants ; c'est-à-dire, ils établissent les règles qui gouvernent les interactions entre composants et spécifient tous les mécanismes auxiliaires nécessaires.

D'après cette définition, les connecteurs décrivent les communications et les connexions entre les composants en modélisant de manière explicite les interactions entre les composants. Un connecteur peut décrire des interactions simples comme un appel de méthode ou des interactions plus élaborées telles que des accès à des bases de données.

Cette définition propose une description des rôles des connecteurs. De la même manière, dans la plupart des travaux, l'aspect structurel est laissé de coté au profit de l'aspect fonctionnel. Ceci permet de conserver suffisamment de généricité et de pouvoir utiliser toutes entités remplissant le rôle adéquat.

[†]Comme pour les composants, nous étudions plus en détails ces définitions dans le chapitre 4 lorsque de l'étude de la sémantique du concept d'architecture logicielle.

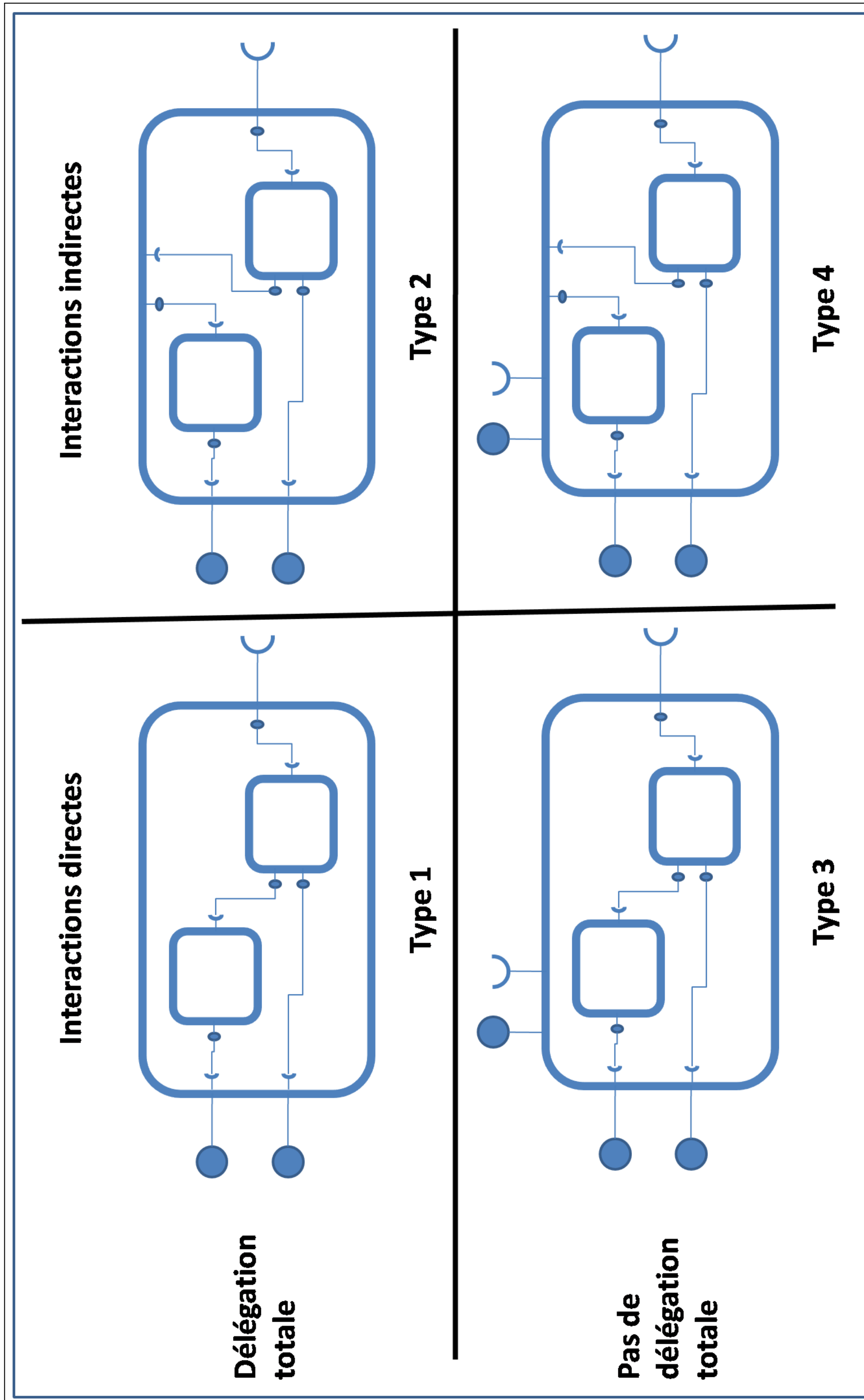


Figure 1.2 – Exemple de structure interne pour les composants composites

Cependant, l'augmentation de l'intérêt pour cet élément architectural à conduit à une formalisation de sa structure. Nous pouvons ainsi décrire plus précisément les structures internes et externes des connecteurs.

1.1.2.1 Aspect Fonctionnel du connecteur

L'aspect fonctionnel des connecteurs est décrit en détail dans la taxonomie de MEDVIDOVIC [90]. Elle classe les connecteurs selon deux niveaux décrivant les services et les techniques utilisés par les connecteurs.

Le premier niveau de la taxonomie de Medvidovic contient quatre types de connecteurs qui se distinguent suivant les services qu'ils proposent :

- **la communication** : les connecteurs de communication transfèrent les données entre les composants. Ce service est l'élément de base de l'interaction entre composants. Les connecteurs de communication permettent aux composants de transmettre les messages, d'échanger les données ou de communiquer les résultats des calculs ;
- **la coordination** : les connecteurs de coordination gèrent le transfert de contrôle entre les composants, permettant aux composants d'interagir par transfert du flot d'exécution. Les appels de fonctions ou les invocations de méthodes sont des exemples de connecteurs de coordination simple ;
- **la conversion** : les connecteurs de conversion servent d'interprètes entre composants hétérogènes, c'est-à-dire provenant de sources différentes et surtout non prévus pour interagir à l'origine. Les connecteurs convertissent le service fourni par un composant pour permettre à un autre de recevoir son service requis sous le bon format. Les conversions peuvent porter sur le nombre ou le type des données échangées, sur la fréquence ou l'ordre des interactions ou encore sur le nom des services ;
- **la facilitation** : même lorsque les composants sont prévus pour être assemblés entre eux, il peut être nécessaire d'introduire des mécanismes pour faciliter et optimiser leurs interactions. Les connecteurs de facilitation offrent donc des services tels que l'équilibrage des charges, la planification ou encore le contrôle de la concurrence.

Les services permettent de faire un premier classement des connecteurs mais ils laissent de coté de nombreux détails essentiels sur leur mise en pratique par les connecteurs. Pour répondre à cela, le second niveau classe les composants en fonction de la manière dont ils réalisent le service. Les huit types proposés peuvent être utilisés pour proposer plusieurs services. Leurs mises en pratique dépendent de plusieurs paramètres qui sont représentés, dans la taxonomie, par leurs dimensions et dont les valeurs possibles sont représentées par les valeurs de la taxonomie. Les huit types sont :

- **les appels de procédures** : ils permettent aux connecteurs de proposer les services de coordination et de communication. Dans le premier cas, le flot de contrôle est modélisé par différentes techniques d'invocation. Dans le deuxième cas, le passage des paramètres permet le transfert de données d'un composant à un autre ;
- **les évènements** : un évènement est un effet instantané de la fin (normale ou anormale) de l'invocation d'une opération sur un objet. Elle se produit dans le contenant de l'objet [109]. Ces évènements permettent une modélisation du flot de contrôle similaire aux appels de procédures. Ce type de connecteurs propose donc des services de coordination. De plus, les messages associés à chaque évènement peuvent être adaptés pour contenir plus d'informations et ainsi permettre de fournir un service de communication ;
- **les accès aux données** : les connecteurs d'accès aux données permettent aux composants d'interagir avec des composants de stockage de données. Ils offrent ainsi un service de communication. Cependant, la communication avec ce type de composant peut nécessiter des post ou prétraitements tel qu'une conversion de format des données ou encore un nettoyage après la suppres-

sion de données. Le connecteur propose alors un service de conversion ;

- **les liens** : les connecteurs de type « liens » permettent la mise en place de canaux de communication entre les composants. Ces canaux sont ensuite utilisés par les autres connecteurs pour mettre en place leur services. Les connecteurs liens proposent donc un service de facilitation, en particulier de la phase de mise en place du système. Ils peuvent d'ailleurs disparaitre après cette phase ;
- **les flots** : les flots permettent le transfert de grande quantité de données. Ils sont donc utilisés par les connecteurs pour proposer un service de communication. Ils sont particulièrement mis à contribution lorsque la communication requiert un protocole particulier et complexe ;
- **les arbitres** : les connecteurs arbitres organisent le fonctionnement du système. Ainsi, ils peuvent remplir deux services. Le premier est un service de facilitation. En effet, les connecteurs arbitres fournissent les outils pour négocier le niveau des services ou encore les interactions nécessitant un certain niveau d'isolation ou de fiabilité. L'autre service rendu est la coordination. En effet, les connecteurs arbitres peuvent rediriger le flot d'exécution entre les composants. Cette redirection peut dépendre, par exemple, d'une planification ou d'un objectif d'équilibrage des charges entre les composants ;
- **les adaptateurs** : les connecteurs adaptateurs proposent un service de conversion. L'adaptateur peut ainsi permettre l'interaction entre composants qui ne sont pas prévus pour communiquer. Pour cela, il doit adapter la politique de communication et le protocole d'interaction des composants. Par exemple, les adaptateurs permettent de résoudre les problèmes de polymorphisme en faisant le lien entre des interfaces fournies et requises qui se complètent mais n'ont pas le même nom ;
- **les distributeurs** : les connecteurs distributeurs proposent un service de facilitation. Ils sont chargés d'identifier les chemins d'interaction et de fournir des informations de routage pour la communication ou la coordination entre les composants. Ces connecteurs ne sont jamais utilisés directement, mais fournissent plutôt une assistance aux autres connecteurs pour remplir leurs fonctions.

1.1.2.2 Structure externe du connecteur

A l'image des composants, un connecteur est caractérisé par deux éléments [89] : ses interfaces qui spécifient les types et le rôle des composants communicant à travers le connecteur ; et ses propriétés qui décrivent les aspects relevant de la conception ou de l'analyse des connecteurs et documentent l'architecture d'une manière similaire aux propriétés des composants.

Interfaces. Les interfaces, appelées rôles dans certains langages de description d'architecture tel que Wright [2], sont la partie visible du connecteur (*cf.* Figure 1.3). Elles servent à déclarer les participants à l'interaction décrite par le connecteur. Comme celles des composants, elles constituent les points de connexion entre les connecteurs et les composants. Néanmoins, à la différence des composants, les interfaces ne décrivent pas de services fonctionnels mais des mécanismes de connexion. Elles décrivent également le rôle de chacun des composants impliqués.

Contrairement aux composants, on ne distingue généralement pas de directions pour les interfaces de connecteurs. Cependant, dans le reste de ce manuscrit, nous utilisons, comme pour les composants, les appellations d'interfaces entrantes et sortantes. Les premières relient le connecteur à des interfaces fournies de composants alors que les secondes relient le connecteur à des interfaces requises de composants.

Propriétés. Les propriétés des connecteurs sont de deux types :

- **propriétés non fonctionnelles** : elles spécifient les besoins du connecteur pour une implémentation correcte. Par exemple, elles peuvent concerner la performance ou la sécurité. Comme pour

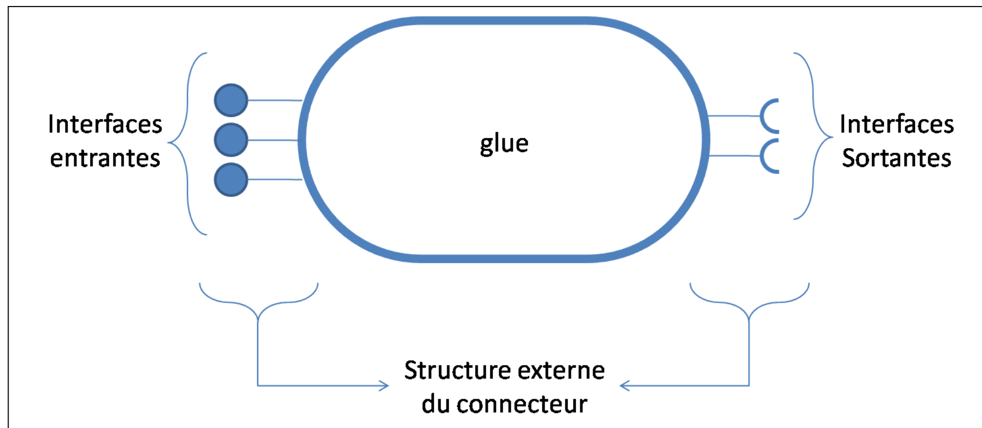


Figure 1.3 – Représentation d'un connecteur

les composants, ces propriétés marquent une séparation claire entre les aspects fonctionnels et non fonctionnels. Elles permettent ainsi de simuler le comportement des connecteurs à des fins d'analyse, de définition des contraintes ou encore de sélection des connecteurs ;

- **contraintes** : ces propriétés définissent les conditions d'utilisation du connecteur. Comme pour les contraintes portant sur les composants, ces contraintes doivent être vérifiées pour que le système soit considéré comme cohérent.

1.1.2.3 Structure interne du connecteur

La structure interne des connecteurs est une autre de leurs caractéristiques. De la même manière que pour les composants, on distingue deux types de structures internes pour les connecteurs : structure atomique ou composite.

Connecteur atomique. La structure interne des connecteurs atomiques est appelée glu (*cf.* Figure 1.3). Elle forme la passerelle entre les interfaces du connecteur. Pour cela, elle décrit le protocole de communication entre les interfaces, points d'accès des composants vers le connecteur.

Ce type de connecteur est le plus couramment utilisé. En effet, même si les connecteurs sont, en théorie, des entités du même niveau que les composants, dans la pratique, les outils utilisant ou décrivant les architectures considèrent les composants comme les éléments prépondérant de l'architecture. Ainsi, alors que les composants composites sont utilisés et répandus, les connecteurs sont souvent considérés comme des interactions simples, et sont donc décrit par des connecteurs atomiques. Par exemple, dans les langages de description d'architecture, les connecteurs sont proposés de trois façons : il existe un seul type de connecteur simple, parfois même sans représentation explicite (Rapide [3]) ; les connecteurs doivent être choisis dans un ensemble prédéfini (UNICON [136]) ; enfin, il est parfois possible de définir un connecteur atomique (Wright [2], Aesop [46]).

Connecteur composite. Les connecteurs composites ont une structure interne plus complexe que celle des connecteurs atomiques. A l'image des composants composites, les connecteurs composites possèdent une structure interne composée de composants, de connecteurs et d'une configuration. Ainsi la structure interne d'un connecteur composite est une architecture interne à ce connecteur (*cf.* Figure 1.4). Ce type de connecteur permet, par exemple, de décrire des protocoles complexes de communication qui demandent un pré et un post-traitement des données.

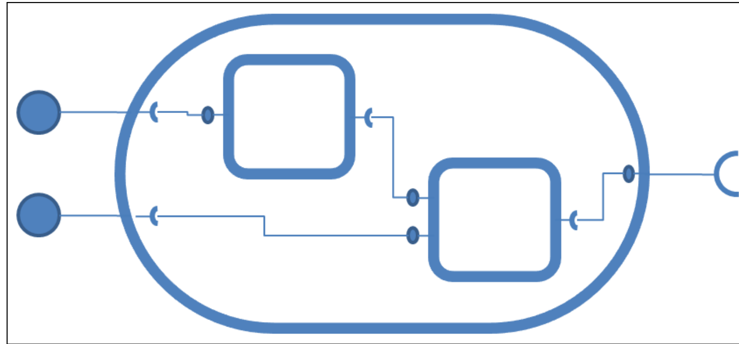


Figure 1.4 – Exemple de connecteur composite

1.1.3 La configuration

La configuration définit la structure de l'architecture. Pour cela, la configuration d'une architecture présente la topologie des connexions entre les composants et les connecteurs ainsi que les propriétés de cette topologie. Par conséquent, la configuration est caractérisée par deux éléments : la structure de la configuration, représentant la topologie des connexions entre composants et connecteurs, et les propriétés de la configuration.

1.1.3.1 Structure de la configuration

La structure de la configuration représente la topologie des connexions entre les composants et les connecteurs. Elle vérifie la correspondance entre les interfaces des composants et des connecteurs. Pour représenter cette topologie, la structure de la configuration peut prendre deux formes :

- **multigraphe** : cette représentation de la structure de la configuration utilise un multigraphe, c'est-à-dire un graphe dont les arêtes peuvent connecter plus de deux sommets. Les sommets du multigraphe représentent les composants de l'architecture, alors que les arêtes représentent les connecteurs. Le défaut principal de cette représentation est qu'elle ne place pas les composants et les connecteurs au même niveau ;
- **graphe biparti** : un graphe biparti contient deux types de sommets. L'origine et l'extrémité de chaque arête du graphe appartiennent obligatoirement à un type de sommets différent. Ce graphe biparti permet de représenter les deux éléments architecturaux, *i.e.* composants et connecteurs, par des sommets. Cette représentation permet ainsi de pallier les défauts du multigraphe en considérant les composants et les connecteurs comme des entités du même niveau. Pour représenter une structure de configuration, le graphe biparti doit également vérifier que chaque sommet de type connecteur est de degré minimum deux. Ceci permet de vérifier que chaque connecteur relie au moins deux composants.

Par la suite, nous considérons la structure de la configuration selon un graphe biparti.

1.1.3.2 Les propriétés de la configuration

Les propriétés de la configuration sont similaires à celles des composants et des connecteurs. Comme pour les autres éléments architecturaux, ces propriétés sont de deux types :

- **propriétés non fonctionnelles** : certaines propriétés non fonctionnelles ne peuvent pas être exprimées au niveau des composants ou des connecteurs. Il faut donc exprimer ces propriétés au niveau

de la configuration. Ces propriétés concernent par exemple l'environnement de déploiement de l'architecture ;

- **contraintes** : les contraintes portant sur la configuration s'ajoutent à celles portant sur les autres éléments architecturaux. Elles permettent d'exprimer des contraintes portant sur plusieurs éléments architecturaux. Par exemple, une contrainte peut exprimer une relation entre deux composants. Les contraintes de la configuration peuvent également être globales et porter sur l'ensemble des éléments architecturaux.

1.1.4 Le style architectural

Le style d'une architecture est un patron de conception. Il définit l'architecture à deux niveaux : la réalisation, qui impose des contraintes et des types pour les éléments architecturaux, et l'interprétation, qui impose une sémantique et un vocabulaire pour décrire les éléments architecturaux [49].

1.1.4.1 Style et interprétation de l'architecture

Le style précise une sémantique pour l'architecture et ses éléments. Il décrit le vocabulaire utilisé pour désigner les éléments architecturaux. Ainsi, dans le style *pipe and filter*, l'architecture représente un processus de traitement par flots. Les informations entrent par un point dans l'application et sortent par un autre. Dans l'application, les informations sont traitées en passant alternativement à travers des filtres et des tuyaux. Les filtres peuvent modifier le contenu du flot d'information mais traitent ce flot de manière séquentielle. Les tuyaux, eux, ne font qu'acheminer le flux d'un filtre à l'autre. Ainsi, une architecture basée sur ce style désigne les composants par le terme « filtres » et les connecteurs par le terme « tuyaux ».

Le style définit également une certaine représentation de l'architecture. Par exemple, une architecture *pipe and filter* représente les composants par des rectangles et les connecteurs par des flèches entre les composants (*cf.* Figure 1.5).

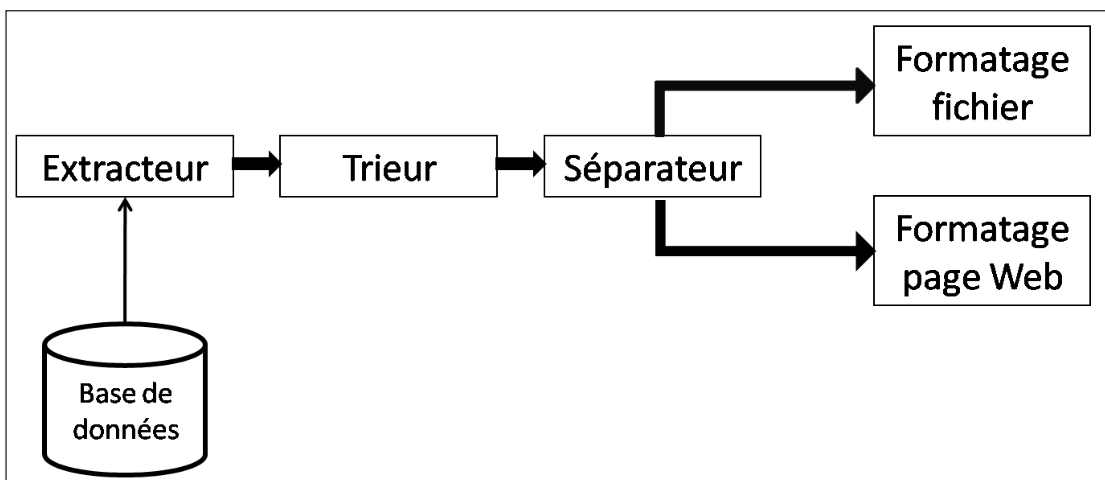


Figure 1.5 – Architecture selon le style *pipe and filter*

1.1.4.2 Style et réalisation de l'architecture

Le style d'une architecture impose des limitations et des contraintes, plus ou moins formelles, sur le type et sur les différents éléments architecturaux [1] :

- **les composants** : le style limite le type des composants. Ces limites sur le type portent sur le nombre d'interfaces requises, celui des interfaces fournies ou encore sur les services fournis par le composant. Ainsi le style *pipe and filter* impose que les composants aient exactement une interface requise et une interface fournie, sauf pour un certain type de composants qui permettent de réunir ou de séparer le flot de donnée (*cf.* Figure 1.5) ;
- **les connecteurs** : le style impose aussi des limites sur le type des connecteurs. Ces limites concernent le nombre d'interfaces entrantes ou sortantes. Par exemple, le style *pipe and filter* requiert des connecteurs ayant exactement deux interfaces, une entrante et une sortante (*cf.* Figure 1.5) ;
- **la configuration** : le style impose également des contraintes sur la configuration. Ces contraintes portent sur le nombre de composants, de connecteurs, ou d'autre contraintes ne pouvant pas être exprimées au niveau du composant ou du connecteur. Ainsi, le style *pipe and filter* impose que exactement un composant ait une interface requise libre, c'est l'entrée de l'application, et un composant ayant une interface fournie, c'est la sortie de l'application (*cf.* Figure 1.5).

1.1.5 Avantages de l'architecture

L'architecture est un point clé affectant la plupart des attributs d'un système [114]. Ces impacts ont été décrits par GARLAN et PERRY [47], du point de vue de la conception. En se plaçant du point de vue de la maintenance, nous pouvons énumérer les impacts de l'architecture selon cinq angles [74] :

- **compréhension du système** : l'architecture fournie une représentation d'un système à un haut niveau d'abstraction. Cette vue synthétique du système met en valeur la plupart des décisions de conception ainsi que les conséquences de ces mauvaises décisions. En effet, l'architecture met en valeur les contraintes du système qui justement intéressent les personnes réalisant la maintenance. Elle permet ainsi à ces personnes de concentrer les efforts d'exploration du système sur les informations architecturales et ainsi de comprendre plus rapidement la conception et le fonctionnement du système ;
- **réutilisation** : l'architecture permet facilement d'identifier les composants réutilisables d'un système. Elle permet également, à travers les connecteurs, d'identifier les dépendances existantes entre ces parties réutilisables d'un système. Au final, ces identifications rendent la réutilisation des fonctionnalités du système plus facile mais aussi plus sûre ;
- **évolution** : l'architecture fournie un squelette du système. Ce squelette permet d'identifier les parties fortement utilisées ainsi que les parties potentiellement fragiles. L'architecture permet ainsi de mettre en valeur les parties nécessitant une attention particulière lors de l'évolution du système. Mais l'architecture permet également de révéler une image précise des dépendances entre les composants. Cette image est nécessaire pour connaitre les impacts de la modification d'un composant sur les autres composants du système et donc les conséquences des différentes évolutions. Elle permet aussi de modifier ces dépendances pour améliorer certains attributs du système tels que la performance ou l'interopérabilité. Enfin, l'architecture permet de corriger les erreurs à la source plutôt que là où elles apparaissent. Ceci peut être réalisé en localisant le composant fautif ou encore certaines dépendances ou contraintes non documentées ;
- **analyse** : la vue abstraite fournie par l'architecture permet de mesurer différents attributs tels que la consistance du système, son respect du style architectural ou encore d'autres attributs de qualité.

Elle permet également de vérifier que les changements prévus dans le système sont conformes au style et aux objectifs de qualité fixés à la conception ;

- **gestion de projet** : la gestion des projets de maintenance du système peuvent reposer sur les composants du système. De plus, l'architecture permet une gestion plus précise des coûts et des risques de modifications, en particulier en soulignant les dépendances entre les composants. Elle permet également une évaluation des qualités du système dans son ensemble, mais aussi des qualités de chaque composant. Ceci permet d'identifier les parties les plus faibles du système et ainsi d'examiner et de cibler précisément leurs faiblesses. Cette identification des composants les plus faibles permet de mettre en valeur les composants les plus problématiques et de décider de leur réingénierie ou de leur redéveloppement. Enfin, la connaissance de la valeur de chaque composant et de leurs dépendances permet de planifier la réingénierie d'un système complexe en ordonnant les modifications selon leurs impacts sur la qualité du logiciel et le risque qu'elles soulèvent.

1.2 Les différentes facettes des architectures logicielles

La terminologie que nous avons présentée est le résultat d'une évolution progressive, portant principalement sur la réification des composants, c'est-à-dire l'entité logicielle correspondant au composant architectural dans le code de l'application, et sur l'intérêt porté aux connecteurs.

La raison principale de cette évolution est que cette terminologie est le résultat du rapprochement des travaux portant sur l'architecture logicielle dans différentes communautés scientifiques. Chacune de ces communautés s'est donc forgée une vue différente de l'architecture en fonction de ces préoccupations et de son historique. La terminologie commune a ensuite émergé progressivement au gré des rapprochements entre les communautés.

De plus, l'architecture étant intimement liée au système et à son paradigme de programmation, l'évolution de la terminologie a aussi été marquée par les fluctuations des paradigmes utilisés par l'ingénierie pour organiser les grands systèmes.

1.2.1 Fluctuation des définitions de l'architecture

La définition de l'architecture est soumise à des variations en fonction du contexte des travaux qui l'utilisent ou la produisent. A ce titre, les paradigmes de programmation jouent un rôle prépondérant dans les variations de la définition d'architecture. En effet, les entités appartenant à chaque paradigme sont identifiées aux éléments architecturaux. Cette identification impose une sémantique et une représentation à l'élément architectural qui découle plus du paradigme que de la définition de l'architecture.

Or, le paradigme idéal à utiliser pour organiser un système a beaucoup fluctué au cours de l'histoire de l'informatique. Selon le modèle de RACOON [104] (*cf.* Figure 1.6), différents paradigmes ont successivement été jugés comme le meilleur.

Par conséquent, ces fluctuations des paradigmes de l'ingénierie ont entrainé des variations de la notion d'architecture. Ainsi les paradigmes procédural, objet et composants ont entrainé l'apparition de cinq variations dans la définition de l'architecture et, en particulier, de celle des composants architecturaux [42].

Architectures et modules. La notion de module a beaucoup évolué, en même temps que les technologies. Ainsi YOURDON et CONSTANTIN [135] définissent un module comme une séquence continue d'instructions, bornée par des éléments frontières et possédant un identifiant. Cette définition est cependant plus proche de l'image que nous avons d'une fonction. Dans les langages plus récents, un module

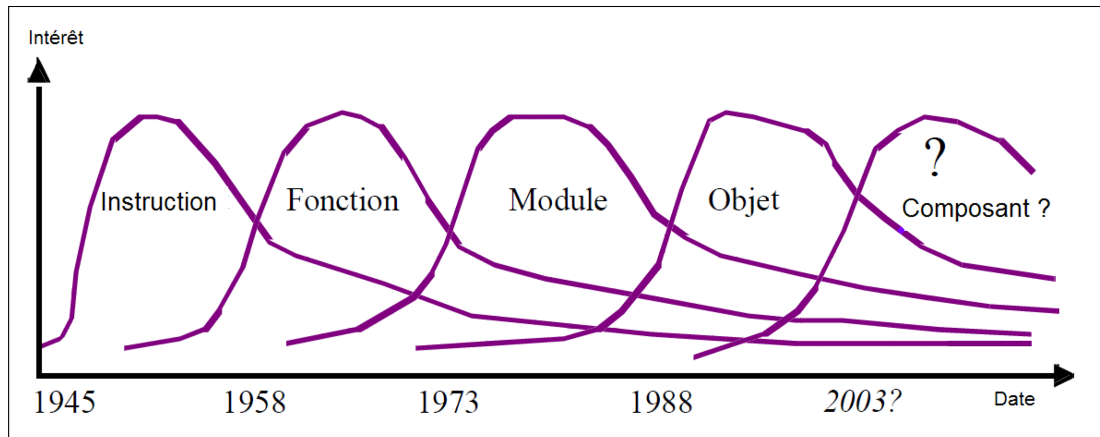


Figure 1.6 – Fluctuation de l'intérêt pour les paradigmes d'ingénierie

est une unité syntaxique contenant des fonctions, des sous-programmes et des déclarations de types et supportant l'encapsulation. Les entités du module sont accessibles aux autres modules à travers son interface, alors que l'implémentation est cachée.

Cette définition des modules fait apparaitre de nombreux parallèles avec celle des composants : encapsulation et interface principalement. Ainsi, par identification, l'implémentation d'une architecture consiste à créer un module pour chaque composant architectural. A l'opposé, l'identification de l'architecture d'un système est réalisée en identifiant les modules du système [4, 123].

Architectures et sous-systèmes. L'identification entre les composants de l'architecture et les sous-systèmes d'un logiciel, résulte de l'introduction de la notion de hiérarchie au sein des composants architecturaux. Ainsi, un sous-système est un regroupement de modules ou de sous-systèmes de granularité plus faible. La différence fondamentale avec la vue modulaire est la granularité. Malgré cette différence de granularité et cette hiérarchie, l'objectif des travaux d'extraction et de maintenance identifiant sous-systèmes et composants [74], reste d'obtenir des composants entretenant un faible couplage et présentant une forte cohésion.

Architectures et objets. L'émergence du paradigme objet a conduit au développement de la notion d'architecture objet. Dans cette architecture, les composants représentent les classes du système. L'architecture permet de représenter un système procédural selon les concepts objets. Chaque composant est un regroupement de fonctions, sous-programmes, variables et de déclarations de types qui forment un objet [110]. La différence majeure avec les définitions précédentes, en plus de l'introduction de la notion d'objet, est encore une fois la granularité des composants. Les classes peuvent être regroupées dans un module.

Patrons de conception. Comme pour les objets, l'émergence de la notion de patron de conception a provoqué l'émergence d'une nouvelle définition de l'architecture. Les patrons de conception représentent les bonnes pratiques de conception et de maintenance dans le paradigme objet. Une architecture basée sur ces éléments est utilisée pour représenter des systèmes orientés objet en termes de patrons de conception et d'interactions entre ces patrons. Les architectures à base de patrons permettent de vérifier facilement la qualité de conception d'un système mais aussi d'identifier rapidement les problèmes majeurs de l'application et enfin de proposer des solutions à ces problèmes [57].

Architectures et composants logiciels. De la même manière que pour l'architecture objet ou patron, le développement de la technologie des composants logiciels a eu lieu en parallèle avec le développement des architectures à base de composants logiciels. Les composants et connecteurs architecturaux représentent alors respectivement les composants logiciels et les connecteurs. Les architectures à base de composants sont couramment utilisées pour la maintenance et l'évolution des systèmes orientés objet.

Comparaison des vues architecturales Au delà de la différence dans la définition associée aux composants, ces définitions d'architectures ont une granularité différente. Ainsi, nous pouvons établir un ordre partiel entre les composants de ces architectures suivant leurs granularités (*cf.* Figure 1.7). D'abord, la plus fine granularité concerne les objets. Ensuite, les granularités des approches composants, patrons de conception et modules sont incomparables. Enfin, la granularité des modules est plus fine que celle des sous-systèmes.

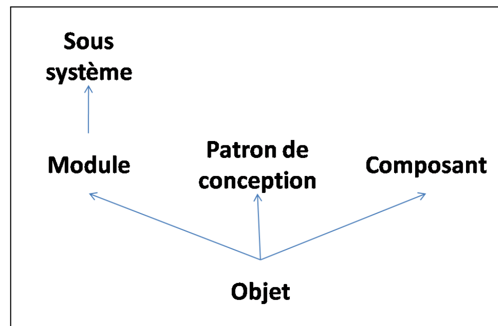


Figure 1.7 – Ordre partiel des types d'architecture selon leur granularité

Un autre point de comparaison entre ces définitions est le paradigme sur lequel la définition a été appliquée (*cf.* Figure1.8). En effet, les définitions à base de modules et de sous-systèmes ont été utilisées principalement dans le cadre du paradigme procédural. Au contraire, les définitions à base de composants et de patrons de conception proviennent principalement d'approches reposant sur le paradigme objet. Enfin, la définition reposant sur les objets est utilisée pour ces deux paradigmes. Cette définition a donc un statut particulier puisqu'elle établit un lien entre les deux paradigmes et, de ce fait, entre les différentes définitions de l'architecture.

1.2.2 Définitions communautaires de l'architecture

Comme nous l'avons vu, la définition de l'architecture varie en parallèle avec les paradigmes de programmation. Cependant, ce facteur n'est pas la seule source des variations dans la définition de la notion d'architecture. Une autre cause majeure de cette hétérogénéité est l'utilisation de l'architecture dans différentes communautés scientifiques puisque leurs définitions de l'architecture sont directement issues de leurs préoccupations et de leurs historiques [42].

Au final, les communautés possèdent chacune une définition propre de l'architecture. Les intérêts portés aux éléments architecturaux varient également d'une communauté à l'autre, de la même manière que l'utilisation qui est faite de l'architecture.

Les principales communauté utilisant l'architecture sont : la communauté « objet », la communauté « Langage de Description d'Architecture » (*Architecture Description Langage*, ADL), la communauté « réingénierie » et enfin la communauté « d'ingénierie logicielle à base de composants » (*Component-based Software Engineering*, CBSE).

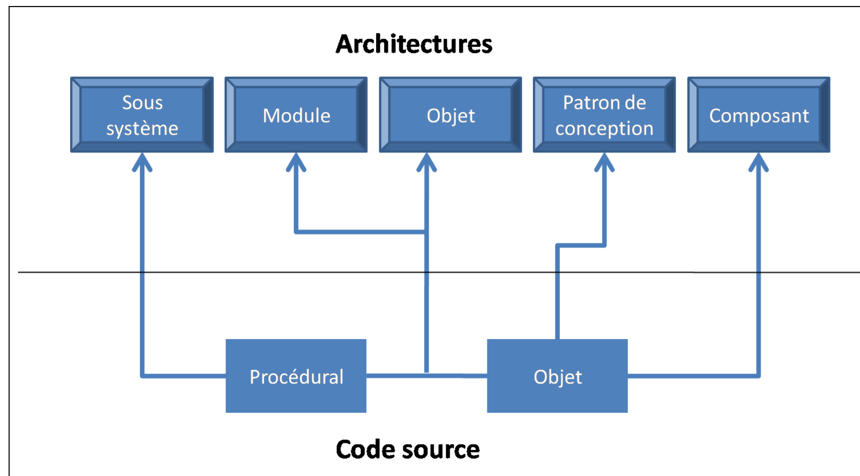


Figure 1.8 – Type de code source et représentations architecturales

Architectures et communauté ADL. Cette communauté propose un ensemble de langages plus ou moins formels pour décrire une architecture logicielle. A ce titre, la définition et l'étude de la notion d'architecture sont le sujet principal des travaux. L'objectif de ces langages de description est de décrire directement les entités architecturales sans utiliser de notions issues d'un autre paradigme. La notion de connecteur bénéficie d'une attention particulière dans cette communauté par rapport aux autres. En effet, c'est dans les travaux de cette communauté que la parité entre les composants et les connecteurs est la plus établie.

Cependant, le traitement accordé aux connecteurs dans cette communauté reste très hétérogène. Ainsi, certains langages de description proposent un seul type de connecteurs simples [3] alors que d'autres permettent de définir des connecteurs atomiques [2, 46].

Architectures et communauté CBSE. La communauté d'ingénierie logicielle à base composants vise à assembler des applications en utilisant des composants logiciels. Ces composants sont une notion très proche des composants architecturaux. Ainsi, les architectures sont ici exprimées en termes de composants logiciels. Par contre, les connecteurs sont encore souvent considérés comme des entités de second plan, se résumant alors à un simple appel de méthode. Cette situation tend, cependant, à s'atténuer suite aux rapprochements avec la communauté ADL.

Architectures et communauté objet. La structuration de grands systèmes orientés objet est un problème ouvert. La communauté est parvenue à la conclusion que les notions de classes et d'objets sont insuffisantes pour résoudre ce problème. Par conséquence, l'architecture d'un système est souvent, dans ce domaine, exprimée en termes de patrons de conception. Comme pour la communauté CBSE, les connecteurs constituent des éléments secondaires dans l'architecture. Ils représentent des appels de méthodes ou des relations de dépendances entre les patrons de conception.

Architectures et communauté réingénierie. Cette communauté a également exprimé de plusieurs façons la notion d'architecture d'un système. Pendant la période d'intérêt pour les modules, les travaux de cette communauté portaient principalement sur l'expression de l'architecture à travers la notion de module. Par la suite, l'apparition et le développement des objets ont conduit les travaux de cette communauté vers des architectures exprimées en objets. Enfin, l'émergence du développement à base de

composants à mis en lumière l'utilisation des composants logiciels pour structurer l'architecture.

1.3 Architectures logicielles et mesure de la qualité logicielle

Les avantages de l'architecture font qu'elle est souvent utilisée de manière *ad-hoc*. Ainsi, l'architecture est couramment utilisée manuellement pour bénéficier des avantages que nous avons évoqués dans la section 1.1.

D'autres travaux proposent des approches spécifiques permettant d'utiliser l'architecture. Parmi ces travaux, une large part est consacrée à l'analyse de la qualité des systèmes. En effet, la vue abstraite fournie par l'architecture permet la mesure de différentes attributs qui reflètent la qualité d'un système.

Nous étudions, dans cette section, ces approches de mesure de la qualité logicielle. Pour cela, nous présentons, dans un premier temps, notre classification de ces travaux. Ensuite, nous décrivons chacun des axes de notre classification.

1.3.1 Classification des approches de mesure de la qualité logicielle

La vue abstraite fournie par l'architecture permet la mesure de différents attributs. Ces attributs reflètent la qualité du système représenté par l'architecture. Tirant partie de cet avantage, de nombreux travaux visent à utiliser l'architecture pour mesurer la qualité des systèmes. Ces travaux de mesure de la qualité logicielle (*Software Quality Measurement*, SQM) peuvent être classés selon cinq axes (*cf.* Figure 1.9) :

- **les objectifs de l'approche** : la mesure de la qualité d'une architecture peut avoir plusieurs objectifs selon que cette évaluation est réalisée de manière préventive (prédiction), ou corrective/validante (mesure). L'évaluation peut également viser différentes caractéristiques de qualité ;
- **le modèle de qualité** : il détermine la façon dont chaque caractéristique est mesurée. Certaines approches suivent des modèles de mesure établis et clairement définis. D'autres, mesurant une seule caractéristique de qualité, n'adhèrent à aucun modèle ;
- **les entrées de l'approche** : le processus d'évaluation peut prendre différents artefacts en entrée. Les approches utilisent toute l'architecture, mais certaines utilisent également la modélisation UML ou le code pour représenter le système. Elles utilisent également un ensemble de valeurs obtenues par test du logiciel, lorsque celui existe déjà, ou par approximation ;
- **la technique de l'approche** : le processus d'évaluation peut être reproductible ou non. Les processus reproductibles utilisent des métriques ou une modélisation du système pour évaluer la qualité d'un logiciel. Chaque exécution du processus sur un système particulier produit un résultat similaire. Les processus non reproductibles sont en grande partie manuels. Ils utilisent des groupes d'experts pour créer des scénarios et évaluer le système en fonction de ces scénarios ;
- **les sorties de l'approche** : toutes les approches visent à évaluer la qualité de l'architecture ou du système étudié. Cependant, certaines fournissent une valeur pour cette caractéristique alors que d'autres déterminent la meilleure solution selon un aspect précis. Enfin, certaines utilisent cette évaluation pour orienter un processus de maintenance.

Dans la suite de cette section, nous étudions successivement chaque axe de cette classification.

1.3.2 Les objectifs de l'approche

Les objectifs des approches SQM peuvent être classés selon deux critères. Le premier se rapporte à la phase du cycle de vie durant laquelle l'approche évalue la qualité du système. Le second décrit le point

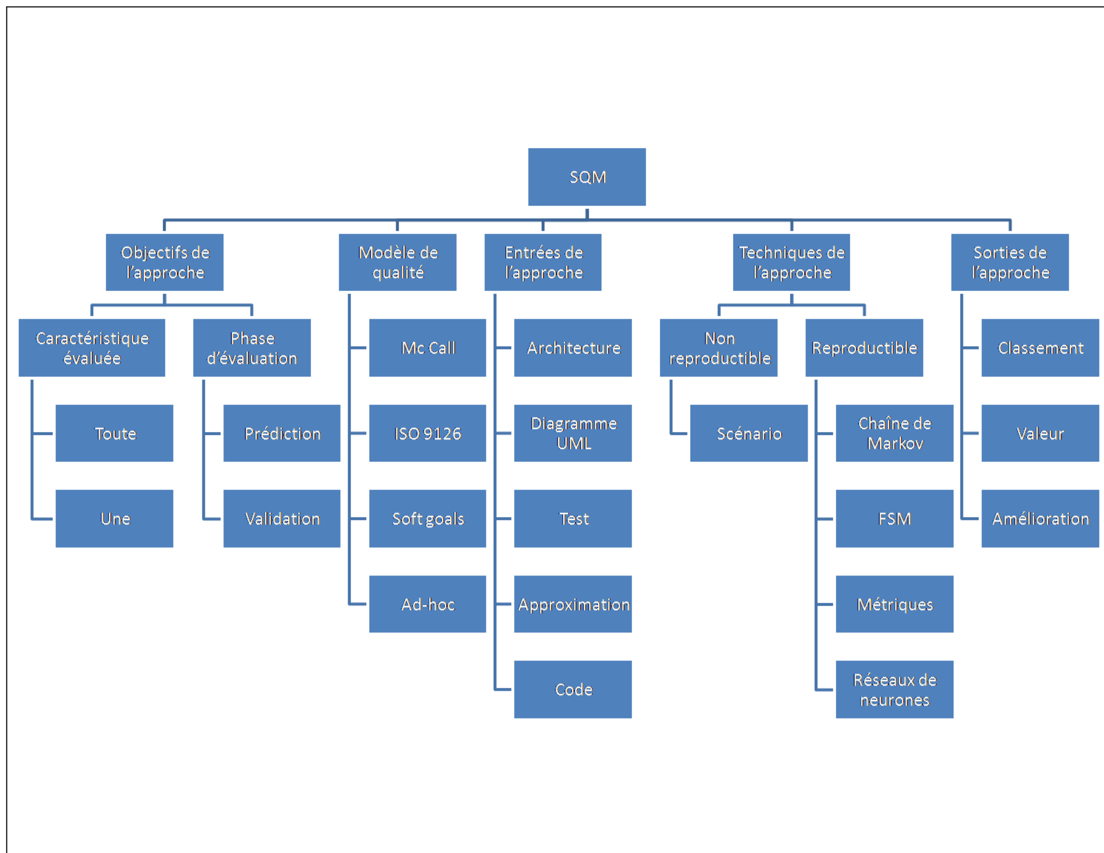


Figure 1.9 – Modèle de classification des travaux d'évaluation d'architectures

de vue des approches sur la qualité logicielle. Il détermine les caractéristiques qui constituent la qualité d'un système et sont évaluées par l'approche.

Phases d'évaluation. La mesure de la qualité d'un logiciel est une activité importante à tous les niveaux du cycle de vie du logiciel. Si les premiers travaux en la matière se concentraient sur la mesure de la qualité d'un système existant, les travaux actuels visent à fournir une évaluation de la qualité d'un logiciel dès la conception de son architecture. On distingue ainsi les travaux qui permettent de prédire la qualité d'un logiciel pendant sa conception et ceux permettant de mesurer cette qualité après conception.

Tirant partie de l'architecture, la plupart des travaux visent à prédire la qualité d'un système au cours de la phase de conception [15, 21, 31, 84, 117]. Ces travaux permettent une évaluation précoce de l'architecture du système et évitent au projet de partir dans une mauvaise direction dès l'origine. En effet, les évaluations de qualité d'un logiciel mettent souvent en lumière des lacunes dans sa conception qui sont couteuses à combler en fin de cycle mais peuvent facilement être évitées en début de conception. Ainsi, SAAM [71] utilise un groupe d'experts pour élaborer un ensemble de scénarios pour le système et ensuite exécuter ces scénarios sur l'architecture que l'on doit tester. Cette ensemble d'experts permet une évaluation précoce de la qualité du système.

Les autres travaux utilisent l'architecture pour évaluer la qualité d'un système existant [56, 105, 120, 133]. En effet, ces travaux sont plus automatisés et requièrent donc plus d'informations sur le système et son comportement. Ces informations nécessitent donc que le développement du système soit suffisamment avancé pour permettre de réaliser certaines mesures ou tests. Ainsi HMSRM [81] ou l'approche de REUSSNER [106] nécessitent une évaluation du temps passé dans chaque partie du système et de la probabilité de passage d'une partie à l'autre.

Cependant, la frontière entre ces deux moments d'évaluation est fragile. En effet, de nombreuses approches proposant de mesurer la qualité d'un système existant peuvent être utilisées pour prédire la qualité d'un système [67, 84, 128]. Il suffit, dans ce cas, d'utiliser un groupe d'experts pour évaluer et prédire, en fonction de l'architecture, les différentes informations sur le comportement du système nécessaires à l'évaluation de la qualité.

Caractéristiques évaluées. Les travaux peuvent aussi être classés selon les caractéristiques de qualité qu'ils mesurent. En effet, la qualité d'un logiciel est une notion abstraite qui a été raffinée en de nombreuses caractéristiques plus ou moins distinctes [44]. Ainsi, les approches de mesure de la qualité évaluent chacune un ensemble particulier de caractéristiques qui recouvre, du point de vue de l'utilisateur, la qualité d'un système.

Certains travaux permettent d'évaluer les caractéristiques importantes pour l'utilisateur [15, 21, 31, 84, 71, 117]. Ces approches utilisent un groupe d'experts pour définir l'ensemble des caractéristiques pertinentes pour le système et les objectifs qu'il doit atteindre.

D'autres approches proposent un ensemble de caractéristiques liées à un type particulier d'applications. Par exemple, GRUNSKE propose de mesurer la fiabilité, la performance, la sécurité et la robustesse d'applications embarquées [56].

Enfin, certains travaux se concentrent sur l'évaluation d'un unique critère ou d'un ensemble de caractéristiques proches [84]. Parmi ces travaux, les plus nombreux portent sur la fiabilité [81, 106, 128] et sur la maintenabilité [67, 101, 120].

Le tableau 1.1 résume, pour cet axe, les différentes caractéristiques des approches abordées. Les deux premières colonnes indiquent la phases d'évaluation : prédiction (Pre), mesure (Mes). La dernière colonne indique les caractéristiques de qualité mesurées : définies par des experts (Exp), fiabilité (Fia),

(REUSSNER et al. 2003)	Pre	Mes	Fia
HMSRM (LI et al. 2007)	Pre	Mes	Fia
(WANG et al. 2006)	Pre	Mes	Fia
(SHARMA et al. 2007)	Pre		Fia
SREPT (RAMANI et al. 2000)		Mes	Fia, Perf
MALRRA (YACOUB et al. 2001)	Pre	Mes	Fia
AEM (LOSAVIO et al. 2004)	Pre	Mes	Fia, Perf, Fonc, Uti, Maint, Port
(GRUNSKE 2007)		Mes	Fiab, Perf, sec, Rob, Comp
(PEDRYCZ et al. 2001)	Pre	Mes	Fia, Maint, Rob, Comp
(JENKINS et al. 2007)	Pre	Mes	Maint
(TAHVILDARI et al. 2002)		Mes	Perf, Maint
ALMA (BENGTSSON et al. 2004)	Pre		Exp
SAAM (KAZMAN et al. 1996)	Pre		Exp
ATAM (CLEMENTS et al. 2002)	Pre		Exp
(SVAHNBERG 2004)	Pre		Exp
DUSA (BOSCH 2000)	Pre		Exp

Table 1.1 – Caractéristiques de l'axe objectif des principales approches SQM

performance (Perf), sécurité (Sec), fonctionnalité (Fonc), utilisabilité (Uti), maintenabilité (Maint), portabilité (Port), robustesse (Rob) et facilité de compréhension (Comp).

1.3.3 Le modèle de qualité

Les travaux sur l'évaluation de la qualité utilisent différents modèles de qualité pour diriger leur processus ou orienter les choix des architectes. Ces modèles proposent une réification de la notion de qualité d'une architecture ou d'un logiciel. Le tableau 1.2 résume, pour cet axe, les différentes caractéristiques des approches abordées.

Le modèle de McCall Un des premiers modèles proposés est celui de MCCALL [26] (*cf.* Figure 1.10). Dans ce modèle, la qualité d'un logiciel est mesurée suivant trois niveaux :

- **le premier niveau contient les facteurs de qualité**. Ceux sont les attributs de haut niveau du logiciel et ils constituent les caractéristiques influant sur la qualité d'un système. Ces facteurs peuvent être la fiabilité ou la maintenabilité par exemple ;
- **le second niveau décrit les critères de qualité**. Ce niveau permet de détailler les facteurs de qualité et de faciliter l'identification des métriques permettant de les mesurer. Ces critères permettent de décomposer les facteurs de qualité du premier niveau et d'obtenir des attributs internes de plus bas niveau. De cette façon, il suffit de définir des métriques pour mesurer ces critères puis de recomposer les métriques pour obtenir une évaluation des facteurs de qualité. A titre d'exemple, la fiabilité est décomposée en trois critères : la consistance, la précision et la tolérance aux erreurs ;
- **le troisième niveau décrit les métriques de qualité**. Ces métriques constituent le dernier niveau de décomposition et évaluent les critères de qualité du second niveau. Au contraire des deux niveaux précédents, les métriques ne sont pas définies dans le modèle. Il convient à chaque utilisateur du modèle de définir l'ensemble de métriques le plus adapté à son domaine, son système et ses objectifs.

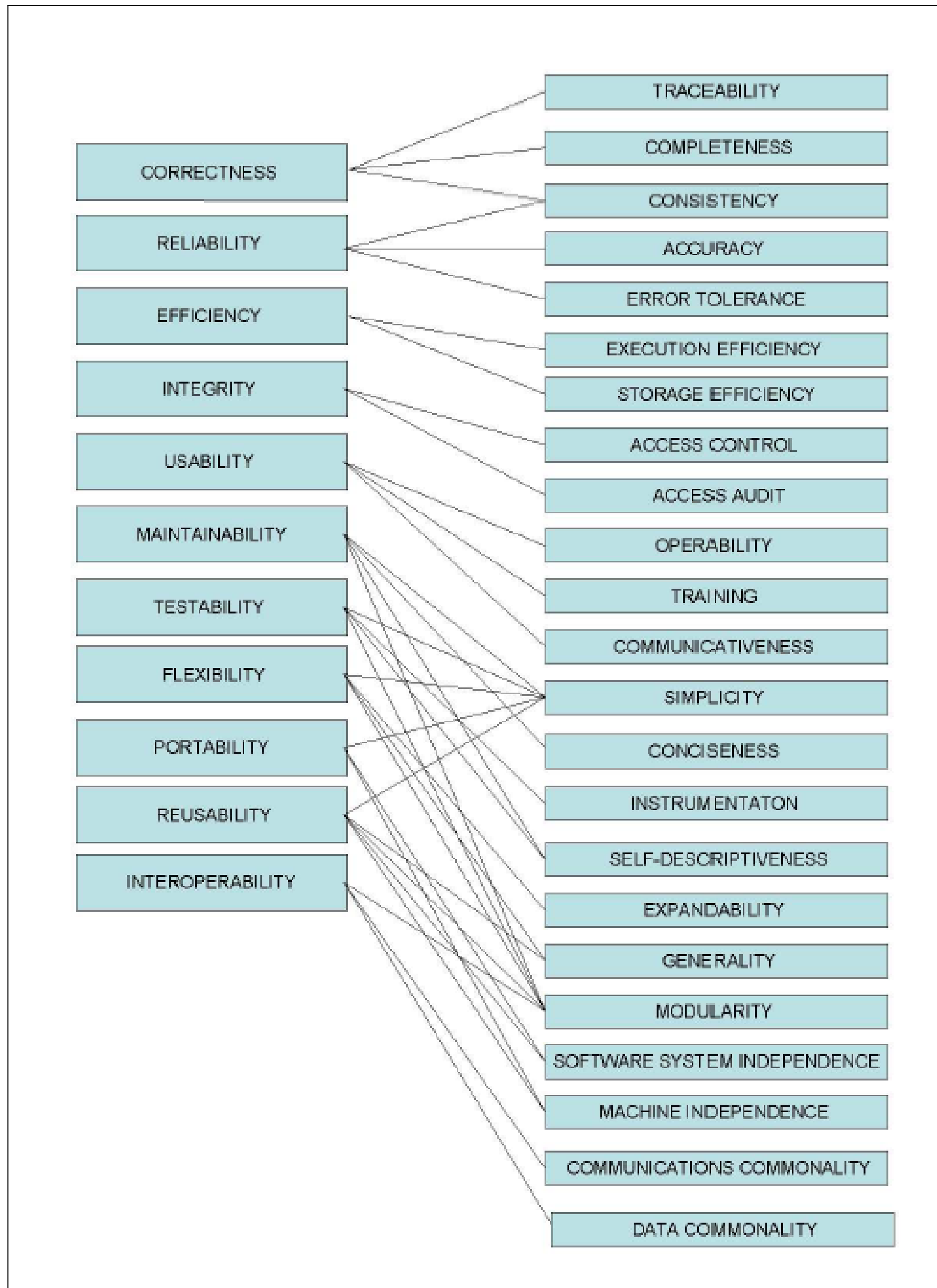


Figure 1.10 – Modèle de qualité de MCCALL

Le modèle de McCall a ouvert la voie des modèles de qualité et a permis de développer certaines approches de SQM essentielles [31, 71, 101]. Cependant, ce modèle particulier est moins utilisé dans les approches récentes.

La norme ISO-9126. A partir du modèle de McCall mais aussi de BOEHM [20] ou encore le modèle FURPS [51], l'organisme ISO a défini une norme sur la qualité logicielle : la norme ISO-9126 [44]. Cette norme définit un méta-modèle qui impose au modèle de mesure quatre niveaux et précise les relations entre ces niveaux (*cf.* Figure 1.11). Les correspondances avec les éléments proposés par MCCALL sont évidentes :

- **le premier niveau contient les caractéristiques de qualité**. Ces caractéristiques correspondent aux éléments du premier niveau de McCall. Comme les facteurs de qualité, elles décrivent les éléments qui influencent la qualité d'un système ou d'une architecture. Comme dans le modèle de McCall, on retrouve les caractéristiques de fiabilité, utilisabilité, efficacité, maintenabilité et portabilité (*cf.* Figure 1.12). Par contre, la caractéristique de fonctionnalité est présente uniquement dans la norme alors que d'autres facteurs de qualité du modèle de McCall, telle que l'intégrité, ne sont pas présents dans la norme ;
- **le second niveau contient les sous-caractéristiques**. Elles raffinent les caractéristiques et précisent le contenu et le sens des caractéristiques. Elles facilitent ainsi la définition des éléments constituant le niveau suivant. Dans le principe, elles correspondent aux critères de qualité de McCall. Cependant, certaines sous-caractéristiques, telle que l'intéropérabilité, correspondent à des facteurs de qualité, alors que d'autres, telle que la précision, correspondent à des critères de qualités ;
- **le troisième niveau contient les propriétés mesurables**. Elles font le lien entre les sous-caractéristiques et les métriques. Elles n'ont pas de correspondance directe dans le modèle de McCall. La norme ne décrit pas totalement ce niveau mais donne une méthodologie pour l'instancier en fonction des objectifs ;
- **le dernier niveau de la norme est le même que celui du modèle de McCall**. Il décrit les métriques qui mesurent les propriétés. Comme dans le modèle McCall, les métriques ne sont pas précisées dans le modèle. La norme laisse l'utilisateur établir les liens entre les propriétés et les métriques.

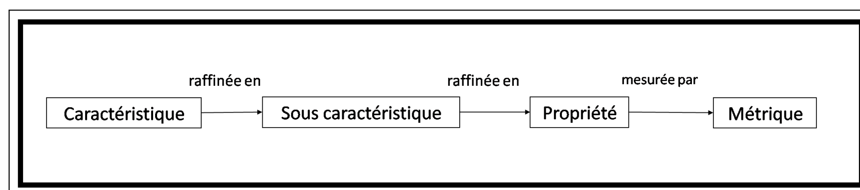


Figure 1.11 – Méta-modèle de mesure des caractéristiques d'un logiciel dans la norme ISO-9126

Les travaux utilisant ce modèle proposent un ensemble de propriétés et de métriques ainsi que les liens entre ces éléments et les sous-caractéristiques de la norme. Par exemple, AEM [84] propose un ensemble de liens établis entre les sous-caractéristiques et un ensemble de métriques. Ces liens sont utilisés pour mesurer la qualité d'un système pendant l'exécution d'un ensemble de scénarios. Le même principe est utilisé par ALMA [15] ou encore par l'approche de SVAHNBERG [117].

Le modèle des *soft-goals*. Le modèle des *soft-goals* [119] est une approche différente des précédentes. Ce modèle ne propose pas un décomposition de la notion de qualité mais une représentation et

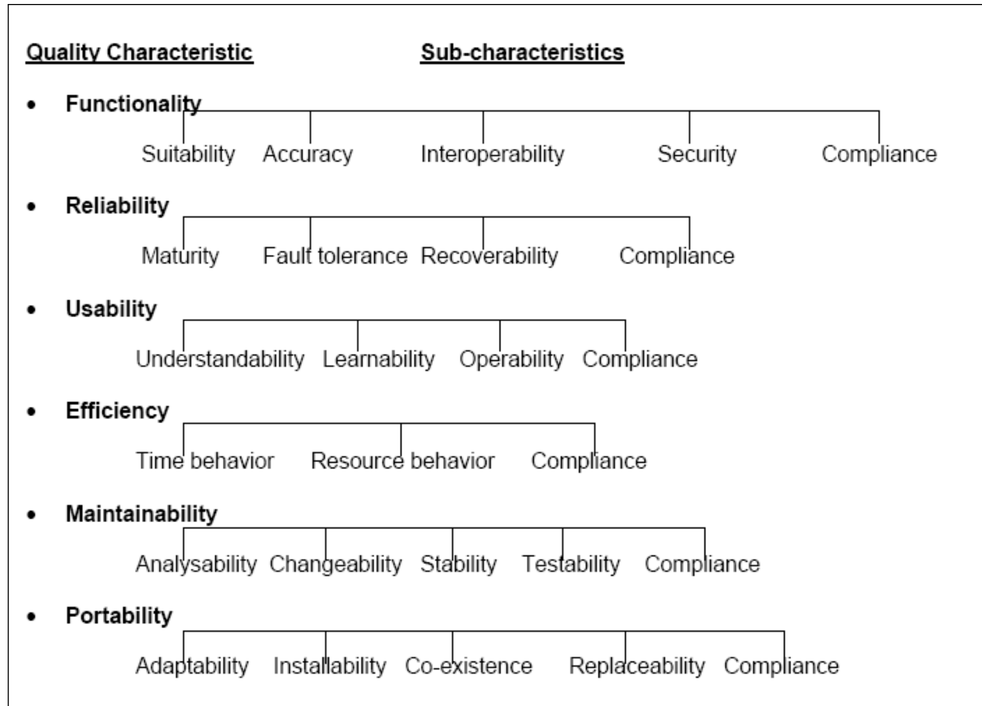


Figure 1.12 – Modèle des caractéristiques d'un logiciel dans la norme ISO-9126

une méthodologie pour réaliser cette décomposition en fonction du contexte (*cf.* Figure 1.13).

Chaque utilisateur définit ces caractéristiques de qualité. Il associe ensuite à chacune un graphe de décomposition. Ce graphe est un arbre dont la racine est la caractéristique mesurée et les feuilles, les métriques. Les liens entre les sommets du graphe peuvent être de différents types selon que tous les fils d'un sommet doivent être testés pour l'évaluer ou qu'un seul fils suffit pour réaliser cette évaluation.

La création des graphes de *soft-goals* est réalisée à partir d'une étude bibliographique de la caractéristique étudiée. Ainsi, l'étude des travaux portant sur une caractéristique permet de la raffiner en différents fils d'une manière similaire aux sous-caractéristiques dans le modèle ISO. Il faut ensuite reprendre l'étude de manière récursive sur les nouveaux fils. Le graphe est complet lorsque toutes les feuilles correspondent à des métriques ou des tests réalisables sur les données disponibles.

Selon ce modèle, TAHVILDARI propose une décomposition pour les caractéristiques de performance et de maintenabilité [120]. Il utilise ensuite une approche manuelle pour évaluer les feuilles du graphe et, en remontant l'arbre, obtenir une évaluation de la caractéristique.

Les modèles *ad-hoc*. Certains travaux ne se conforment à aucun modèle. Ces derniers se concentrent sur la mesure d'une seule caractéristique et n'ont pas besoin d'un modèle général pour obtenir une mesure de celle-ci. La caractéristique est alors immédiatement reliée à un ensemble de métriques.

1.3.4 Les entrées de l'approche

Si la grande majorité des approches SQM nécessitent une représentation de l'architecture en entrée, il reste que de nombreuse approches utilisent également d'autres éléments pour évaluer la qualité d'un système. Ces éléments sont de quatre types :

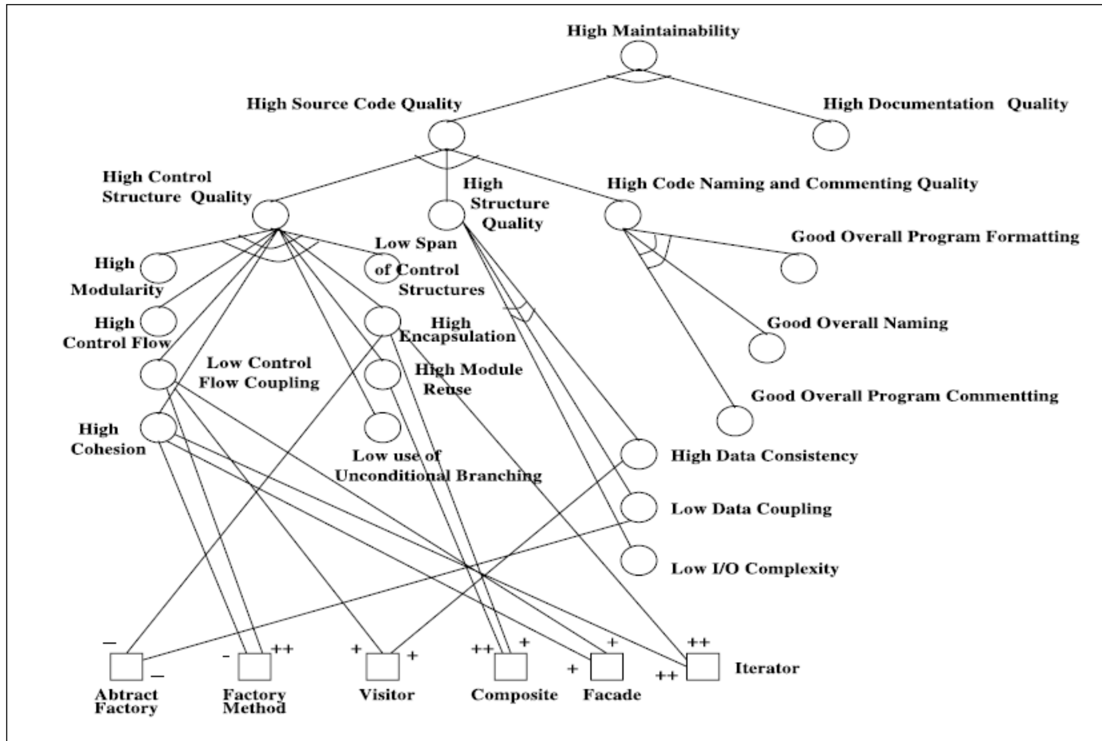


Figure 1.13 – Décomposition de la caractéristique de maintenabilité

(REUSSNER et al. 2003)	*ad-hoc*
HMSRM (LI et al. 2007)	*ad-hoc*
(WANG et al. 2006)	*ad-hoc*
(SHARMA et al. 2007)	*ad-hoc*
SREPT (RAMANI et al. 2000)	*ad-hoc*
MALRRA (YACOUB et al. 2001)	*ad-hoc*
AEM (LOSAVIO et al. 2004)	ISO
(GRUNSKE 2007)	*ad-hoc*
(PEDRYCZ et al. 2001)	McCall
(JENKINS et al. 2007)	*ad-hoc*
(TAHVILDARI et al. 2002)	Soft goals
ALMA (BENGTSSON et al. 2004)	ISO
SAAM (KAZMAN et al. 1996)	McCall
ATAM (CLEMENTS et al. 2002)	McCall
(SVAHNBERG 2004)	ISO
DUSA (BOSCH 2000)	*ad-hoc*

Table 1.2 – Caractéristiques de l'axe modèle des principales approches SQM

- **le code source** : il est utilisé pour affiner la représentation fournie par l'architecture et permettre une modélisation du système [81, 105, 113] ou une simulation de son comportement [31, 71, 117] ;
- **les diagrammes UML** : ils sont utilisés en complément de l'architecture et du code pour obtenir une modélisation [133] ou une simulation plus fidèle aux intentions des architectes [56]. Ceci permet d'améliorer la fiabilité du résultat de l'évaluation ;
- **les tests** : les approches simulant le système nécessitent un ensemble de mesures qui doivent être obtenues à travers des tests [101, 113, 128]. Ces données sont, par exemple, le temps de présence dans chaque composant. Elles permettent, comme les autres entrées, d'affiner la simulation et d'obtenir un résultat le plus fidèle possible à la réalité du système ;
- **les approximations** : les tests peuvent, dans certaines approches, être remplacés par des approximations de leurs résultats [56, 120]. Il est ainsi possible de donner une approximation de la probabilité de passage d'un composant à un autre plutôt que de tester le temps de présence dans chaque composant. Ces approximations permettent d'utiliser l'approche de manière plus précoce.

Le tableau 1.3 résume les différentes entrées utilisées par les approches abordées.

(REUSSNER et al. 2003)	Archi	Test	Approx		
HMSRM (LI et al. 2007)	Archi	Test	Approx		
(WANG et al. 2006)	Archi	Test	Approx		
(SHARMA et al. 2007)	Archi	Test	Approx		
SREPT (RAMANI et al. 2000)	Archi	Test	Approx		
MALRRA (YACOUB et al. 2001)	Archi			UML	Code
AEM (LOSAVIO et al. 2004)	Archi				
(GRUNSKE 2007)	Archi	Test	Approx	UML	Code
(PEDRYCZ et al. 2001)	Archi	Test			
(JENKINS et al. 2007)	Archi				
(TAHVILDARI et al. 2002)	Archi	Test	Approx		Code
ALMA (BENGTSSON et al. 2004)	Archi				Code
SAAM (KAZMAN et al. 1996)	Archi				Code
ATAM (CLEMENTS et al. 2002)	Archi				Code
(SVAHNBERG 2004)	Archi				Code
DUSA (BOSCH 2000)	Archi				Code

Table 1.3 – Caractéristiques de l'axe entrée des principales approches SQM

1.3.5 La technique de l'approche

Les approches de mesure de la qualité logicielle peuvent être classées selon la nature reproductible ou non de leurs résultats. Les premières sont plus automatisées et fournissent les mêmes résultats pour les mêmes entrées alors que les résultats des secondes dépendent des experts qui appliquent ces approches.

Approches non reproductibles. Les approches non reproductibles sont basées sur l'utilisation de scénarios. Dans ces travaux, l'architecture du système et ses objectifs sont utilisés par un groupe d'experts pour réaliser un processus d'évaluation en deux étapes. Dans une première étape, les scénarios sont définis par collaboration entre les architectes et les clients. Chaque scénario est conçu pour évaluer certains critères de qualité constituant les objectifs du logiciel. Les clients et les architectes s'accordent également pour définir les conditions de réussite de chaque scénario ainsi que leurs importances relatives.

La deuxième étape commence une fois ces scénarios définis. Les architectes utilisent, alors, l'architecture et leurs expertises pour simuler l'exécution de ce scénario par l'architecture. La réussite ou l'échec d'un scénario valide les objectifs fixés par les clients.

L'avantage principal de ces approches est leur utilisation possible dans les premières phases du développement d'un système. En effet, le manque d'information peut alors être comblé par les connaissances des experts. Dans les phases les plus avancées du cycle de vie du logiciel, cette expertise s'appuie sur le code disponible.

Les principales différences entre ces approches concernent le processus de conception des scénarios et la constitution du groupe d'experts. GRIMAN propose un cadre de comparaison des approches à base de scénarios qui repose sur ces deux caractéristiques [53]. Ce cadre utilise également le type de résultats fournis par l'évaluation, c'est-à-dire si elle fournit une mesure chiffrée de la qualité ou si elle permet simplement de classer un ensemble d'architectures selon leurs qualités.

Approches reproductibles. Les approches reproductibles peuvent être classées en deux catégories, selon leurs méthodologies :

- **approches par simulation** : dans ces travaux, les informations disponibles sur le système sont utilisées pour réaliser une modélisation du système. Cette modélisation peut prendre la forme d'une chaine de Markov [50, 106], d'un automate d'état fini (*Finite State Machine*, FSM)[56, 105] ou encore d'un réseau de neurones [101] ;
- **approches par métriques** : les approches par métriques utilisent un ensemble de métriques pour mesurer une ou plusieurs caractéristiques de qualité. Dans MALRRA [133], par exmple, les auteurs utilisent des métriques dynamiques. Ces métriques utilisent les diagrammes UML pour fournir une mesure de la complexité du système. Cette complexité est, en effet, reliée à la fiabilité que souhaitent mesurer les auteurs. JENKINS [67] utilise des mesures sur les degrés du graphe constitué par les classes et leurs relations, pour obtenir une mesure de la maintenabilité d'un système. TAHVILDARI [119] utilise les *soft goals* pour relier un ensemble de métriques à une caractéristique de qualité.

Le tableau 1.4 résume les différentes techniques utilisées par les approches abordées.

1.3.6 Les sorties de l'approche

Toutes les approches SQM visent à évaluer la qualité d'un ou plusieurs systèmes. Cependant, la forme des résultats n'est pas toujours la même. En effet, les résultats de l'évaluation peuvent être de trois types :

- **les valeurs** : la plupart des approches de SQM donnent une mesure quantitative de la qualité d'un système [105, 120]. Le résultat de cette mesure est donc une valeur qui peut être comparée. Elle permet ainsi de comparer des évaluations réalisées de manière séparée mais avec la même approche ;
- **le classement** : le résultat de certaines approches est qualitatif [31, 71]. Ce n'est pas une valeur représentant la qualité du système, mais plutôt un classement entre plusieurs architectures. Ce classement est alors utilisé pour déterminer, dans les premières phases du développement, la meilleure architecture pour un système. Dans ces approches, il est difficile de comparer les architectures qui ont été évaluées séparément ;
- **les cibles d'amélioration** : certains approches proposent, en plus d'une évaluation quantitative de la qualité, un guide pour l'amélioration du logiciel évalué [15, 21, 117]. Ainsi, ces approches proposent, grâce à un choix judicieux des scénarios et à une analyse précise des résultats, un ensemble d'améliorations pour la solution architecturale étudiée.

(REUSSNER et al. 2003)	Markov				
HMSRM (LI et al. 2007)	Markov				
(WANG et al. 2006)	Markov				
(SHARMA et al. 2007)	Markov				
SREPT (RAMANI et al. 2000)		FSM			
MALRRA (YACOUB et al. 2001)			Métriques		
AEM (LOSAVIO et al. 2004)				Scénario	
(GRUNSKE 2007)	Markov	FSM	Métriques	Scénario	
(PEDRYCZ et al. 2001)					Neurones
(JENKINS et al. 2007)			Métriques		
(TAHVILDARI et al. 2002)			Métriques		
ALMA (BENGTSSON et al. 2004)				Scénario	
SAAM (KAZMAN et al. 1996)				Scénario	
ATAM (CLEMENTS et al. 2002)				Scénario	
(SVAHNBERG 2004)				Scénario	
DUSA (BOSCH 2000)				Scénario	

Table 1.4 – Caractéristiques de l'axe technique des principales approches SQM

Le tableau 1.5 résume les différentes sorties utilisées par les approches abordées.

1.4 Conclusion

Nous avons introduit, dans ce chapitre, la terminologie liée aux architectures logicielles et les avantages obtenus grâce à l'utilisation de cette notion dans la maintenance et la conception de systèmes informatiques. L'étude de cette terminologie a permis de mettre en lumière l'évolution de la notion d'architecture à travers les différentes communautés scientifiques qui l'utilisent, mais aussi à travers les différentes évolutions de paradigmes. L'étude de cette terminologie nous a aussi permis de montrer que les définitions de l'architecture et des différents éléments architecturaux sont suffisamment précises et admises pour pouvoir en tirer un consensus.

Nous avons également étudié le domaine de l'évaluation de la qualité logicielle. Dans ce domaine, la définition de la qualité d'une architecture logicielle permet d'améliorer l'efficacité et la précocité de l'évaluation de la qualité d'un système informatique.

(REUSSNER et al. 2003)	Valeur		
HMSRM (LI et al. 2007)	Valeur		
(WANG et al. 2006)	Valeur		
(SHARMA et al. 2007)	Valeur		
SREPT (RAMANI et al. 2000)	Valeur		
MALRRA (YACOUB et al. 2001)	Valeur		
AEM (LOSAVIO et al. 2004)	Valeur	Classement	
(GRUNSKE 2007)	Valeur		
(PEDRYCZ et al. 2001)	Valeur		
(JENKINS et al. 2007)	Valeur		
(TAHVILDARI et al. 2002)	Valeur		
ALMA (BENGTSSON et al. 2004)	Valeur		Amélioration
SAAM (KAZMAN et al. 1996)		Classement	
ATAM (CLEMENTS et al. 2002)		Classement	
(SVAHNBERG 2004)	Valeur		Amélioration
DUSA (BOSCH 2000)	Valeur		Amélioration

Table 1.5 – Caractéristiques de l'axe sortie des principales approches SQM

CHAPITRE 2
Extraction d'architectures logicielles

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race

— A.N. WHITEHEAD.

Maintenance — Extraction d'architectures — Modèle conceptuel — Classification des travaux d'extraction

Comme nous l'avons vu dans le chapitre précédent, l'architecture fournit des avantages importants tout au long du cycle de vie d'un logiciel. Parmi les phases bénéficiant de la présence d'une architecture, la maintenance est particulière. En effet, cette phase se déroule sur un système existant qui possède une architecture. Cette architecture peut être connue avec des degrés de précision divers. Certains systèmes existants ne possèdent pas de représentation alors que d'autres ont une vue incomplète de leur architecture.

Cependant, l'absence cruciale de représentation complète de l'architecture dans beaucoup de systèmes, a conduit à rechercher des méthodes pour extraire efficacement l'architecture d'un système existant. A l'image de la diversité des définitions de l'architecture, les travaux d'extraction sont tous différents : ils extraient des expressions différentes de l'architecture depuis des systèmes reposant sur des paradigmes différents. Cependant, tous ces travaux dessinent une approche générale commune.

Après avoir décrit la phase de maintenance et ses principales problématiques, nous expliquons les causes majeures de l'absence de représentation complète de l'architecture ainsi que les conséquences de ces absences. Nous présentons ensuite l'approche générale de l'extraction d'architecture. Enfin, nous exposons notre modèle de comparaison des travaux d'extraction d'architectures et la projection des principaux travaux sur ce modèle.

2.1 Une architecture cruciale mais pas toujours disponible

Si l'architecture occupe, aujourd'hui, une place de choix dans toutes phases de maintenance, son utilisation implique une confiance importante dans la validité de sa représentation. Cette confiance est

nécessaire dans la mesure où l'architecture est utilisée pour diriger les opérations de maintenance sur le code source. Ainsi, un décalage entre l'architecture et la réalité du système peut entrainer la détection de fausses erreurs, la non détection d'erreurs ou encore empêcher de détecter tous les impacts d'une évolution.

Dans cette section, nous revenons sur le lien entre la maintenance et les architectures. Nous étudions ensuite les raisons qui limitent souvent l'utilisation de l'architecture durant la maintenance. Enfin, nous précisons les conséquences qui peuvent découler de l'utilisation d'une représentation incorrecte de l'architecture.

2.1.1 Maintenance et architecture

La maintenance des logiciels est définie, dans la norme IEEE 1219 [97], comme la modification d'un logiciel après la livraison, pour corriger les fautes, améliorer la performance ou les autres attributs, ou encore adapter le produit à un environnement modifié. La phase de maintenance est donc une étape du cycle de vie du logiciel. Cette étape a pour objectif de maintenir le logiciel conforme aux besoins des utilisateurs. Cette phase est la plus longue du cycle de vie puisqu'elle débute après le développement et ne finit qu'avec la fin du logiciel.

La phase de maintenance se heurte à cinq problèmes majeurs :

- **la compréhension** : il est nécessaire pour réaliser une opération de maintenance de bien comprendre le système afin de ne pas commettre d'erreur ;
- **les tests** : ils permettent de valider le résultat des opérations de maintenance ;
- **l'analyse d'impact** : elle permet de mesurer de manière préventive les impacts d'une opération sur l'ensemble du système, et donc de mesurer le coût de cette opération ;
- **la maintenabilité** : elle mesure l'aisance avec laquelle un logiciel peut être maintenu, amélioré, adapté ou corrigé pour satisfaire aux exigences spécifiées [97]. Elle permet de mesurer le coût des opérations de maintenance ;
- **la dynamicité des modifications** : certains systèmes imposent que les opérations de maintenance soient réalisées sans arrêt des services, de manière dynamique.

Pour répondre à ces problématiques, de nombreux domaines sont apparus. Chacun est plus spécifique que la maintenance et se consacre à une partie seulement des problématiques précédentes. Chacun inclut également plusieurs sous-domaines qui se différencient par le type de maintenance ou la problématique abordée :

- **ré-ingénierie** : ces techniques cherchent à reconstruire le système sous une forme nouvelle. C'est une technique utilisée pour des changements importants dans le système, en particulier pour remplacer des systèmes vieillissant ;
- **rétro-ingénierie** : ce processus vise à identifier les structures du système et à fournir une représentation du logiciel à des niveaux d'abstraction plus élevés que celui de l'implémentation. Ces techniques sont passives et permettent uniquement une meilleure visualisation. Ce domaine inclut l'extraction d'architectures, la re-documentation et l'exploration du code ;
- **restructuration** : c'est la transformation d'un système d'une représentation à une autre en conservant, entre les deux, le même comportement et un même niveau d'abstraction. Ce domaine inclut les techniques de migration et de « *refactoring* » ;
- **techniques de tests** : les tests constituent un domaine particulier de la maintenance en raison de leur importance et de leur difficulté. Ils permettent de vérifier que les opérations de maintenance ont atteint leurs objectifs et sans provoquer de modifications inattendues.

L'architecture est un appui crucial dans les différents sous-domaines de la maintenance. Les avantages, que nous avons détaillés dans le chapitre précédent, facilitent les différentes approches en proposant une vue simplifiée du système et en facilitant sa compréhension.

2.1.2 Le manque de représentations architecturales fiables

Malheureusement, il existe peu de systèmes pour lesquels l'architecture est disponible avec le niveau de précision et de validité exigé par les opérations de maintenance. Ainsi, l'architecture peut être totalement indisponible, c'est-à-dire que l'on ne possède aucune représentation architecturale du système. Elle peut également être plus ou moins gravement décalée par rapport au système.

2.1.2.1 Représentation architecturale indisponible

La première raison du manque de précision de certaines architectures, est l'indisponibilité totale d'une vue architecturale d'un système. Ainsi, certains systèmes sont conçus de manière *ad-hoc* sans considération pour les abstractions en général et pour l'architecture en particulier. Par conséquence, à la fin du développement de tels systèmes, aucune représentation de l'architecture n'est disponible, en dehors peut être de l'esprit des concepteurs et développeurs [72].

Le problème supplémentaire de ce type de systèmes vient des défauts de conception qui peuvent apparaître, lors d'un développement réalisé sans les apports d'une vue abstraite. Ce genre de systèmes, qui possèdent donc le moins d'informations architecturales, sont ainsi les systèmes nécessitant le plus de phases d'évolution et de maintenance critiques pour palier les défauts introduits pendant le développement. Il est donc d'autant plus crucial d'avoir une représentation de l'architecture de ces systèmes.

2.1.2.2 Représentation architecturale décalée

La deuxième raison du manque de précision de certaines architectures, concernent les systèmes conçus en s'appuyant sur une ou plusieurs vues abstraites ou architecturales. En effet, même les systèmes les mieux conçus subissent un décalage progressif entre la représentation de l'architecture du système et son code. Ce phénomène, appelé érosion, apparait dès les premières phases du développement et s'amplifie ensuite pendant les phases de maintenance, augmentant progressivement les risques encourus pendant l'évolution [108].

Dès le début du développement, il existe le risque que les développeurs ne respectent pas parfaitement l'architecture qui leur est soumise. Ce non-respect est aussi bien dû à des erreurs de programmation ou de compréhension de l'architecture qu'à des adaptations ou optimisations dont les modifications dans l'architecture ne sont pas reportées dans la représentation architecturale.

Par la suite, durant la phase de maintenance, l'érosion risque de s'amplifier à chaque modification du code. En effet, chaque fois qu'une modification du code n'est pas reportée dans la vue architectural du système, l'écart s'accroît entre l'architecture documentée et l'architecture réelle du système. La maintenance suivante risque ensuite d'accroître l'écart puisqu'elle repose déjà sur une vue faussée de l'architecture. Le report de la modification du code sur cette architecture peut alors être incorrect et ne pas refléter parfaitement la modification faite dans le code.

2.1.3 Conséquence de l'absence de représentation architecturale

Les avantages de l'architecture pendant les phases de maintenance, ont rendu cette abstraction cruciale pour diriger les opérations correctives ou d'amélioration des systèmes. De nombreuses approches de maintenance reposent en grande partie sur la représentation architecturale pour visualiser le système. Ainsi, une architecture, décalée par rapport à la réalité du système, fournit une image fausse de ce système. Cette erreur, introduite des le début du processus de maintenance, peut conduire à deux types de conséquences.

Les premières conséquences sont des erreurs de non-détection. Si on tente de détecter des erreurs dans un système, la non-représentation de certaines relations dans l'architecture, risque de cacher certaines erreurs ou mauvaises utilisations d'un patron de conception. Dans le cas d'une phase de maintenance visant à introduire une modification dans le système, le masquage de certaines dépendances peut entrainer des effets de bords inattendus ou encore un échec de la modification du système.

Le deuxième type d'erreurs concerne les faux positifs. Dans les phases de maintenance corrective, les dépendances ajoutées dans l'architecture peuvent entrainer la détection d'erreurs qui n'existent pas dans le système. Dans ce cas, la correction de ces erreurs risquent de dégrader le système. Dans le cas d'une phase de maintenance visant à introduire une modification dans le système, les dépendances supplémentaires peuvent provoquer des modifications dans une partie du système indépendante et entrainer des dégradations dans cette partie.

Une incohérence entre l'architecture et le système représenté (masquage ou ajout de relations), peut, à tout moment, provoquer des conséquences imprévues durant la maintenance. Ces conséquences peuvent aller de l'échec de la maintenance à la dégradation du système.

Au final, il est pratiquement impossible de réaliser des évolutions sur un système en s'appuyant sur une architecture qui ne correspond pas précisément à l'architecture réelle du système. Or, il est impossible d'avoir la garantie que le processus de développement, puis de maintenance, ont conservé une synchronisation parfaite entre ces deux architectures. Pour résoudre ce manque et pouvoir bénéficier de toutes les approches d'évolution disponibles ainsi que de tous les avantages de l'architecture, de nombreuses approches proposent d'extraire une représentation de l'architecture réelle d'un système.

2.2 Principes de l'extraction d'architectures

Le domaine de l'extraction d'architectures (*Software Architecture Recovery*, SAR) vise à apporter une réponse aux problèmes identifiés dans la section précédente. Nous décrivons dans cette section les principes des approches d'extraction. Pour cela, nous proposons d'abord une définition du domaine de l'extraction d'architectures et de ces objectifs. Ensuite, nous présentons les modèles conceptuels de l'activité d'extraction qui ont déjà été proposés et qui établissent une terminologie pour l'extraction d'architectures. Enfin, nous décrivons notre modèle conceptuel et ses avantages par rapport aux modèles existants.

2.2.1 Définition de l'extraction d'architectures

RIVA définit l'extraction d'architectures comme **une activité archéologique où les analystes doivent révéler toutes les decisions de conception passées en explorant l'implémentation et la documentation existante du système** [108]. KRIKHAAR précise la forme sous laquelle sont révélées les décisions et positionne le domaine de l'extraction d'architectures : l'extraction d'architectures est **la partie de la ré-ingénierie qui concerne toutes les activités visant à rendre explicite l'existence des architectures**

logicielles [75]. JAZAYERI étend les sources d'informations utilisables pendant l'extraction en la décrivant comme **les techniques et les processus utilisés pour révéler l'architecture d'un système à partir de l'ensemble des informations disponibles** [66].

D'après ces définitions, l'objectif de l'extraction d'architectures est de révéler l'architecture d'un système. L'utilisation qui peut être faite ensuite de cette architecture ne fait pas partie du domaine. Cependant, certaines approches d'extraction sont développées avec un objectif particulier au délà de l'identification de l'architecture. Cet objectif primaire n'a pas d'impact majeur sur le processus d'extraction. Il concerne plus la forme du résultat que la façon dont celui-ci est obtenu. Ces objectifs primaires de l'extraction peuvent ainsi être de six types [103] qui sont à rapprocher des avantages de l'architecture décrit dans le chapitre précédent :

- **re-documentation et compréhension** : l'architecture extraite d'un système permet de documenter le système et aide les ingénieurs à le comprendre ;
- **maintenance** : l'architecture extraite permet aux ingénieurs de bénéficier de tous les avantages offerts par l'architecture dans le cadre de la maintenance ;
- **réutilisation** : l'architecture facilite l'identification de composants réutilisables dans un système ;
- **analyse** : l'extraction de l'architecture d'un système fournit un point d'entrée pour réaliser l'analyse du système. Cette analyse consiste alors à une évaluation de la qualité du système au moyen de travaux tels que ceux présentés dans le chapitre précedent ;
- **co-évolution** : l'architecture extraite permet de lutter contre l'érosion. En effet, il suffit de réaliser régulièrement l'extraction pour re-synchroniser l'architecture du système et la documentation qui la représente. L'extraction permet aussi de vérifier que les procédures mises en place pour lutter contre l'érosion sont efficaces ou respectées ;
- **conformité** : l'architecture extraite permet de vérifier la conformité de l'architecture réelle du système avec l'architecture conceptuelle, c'est-à-dire celle que les architectes imaginent que le système respecte. Elle permet également de vérifier la conformité du système par rapport à certaines règles ou styles imposés.

2.2.2 Modèles conceptuels existants de l'extraction d'architectures

KAZMAN et *al.* [70] puis KOSCHKE [74] ont défini deux modèles conceptuels de l'extraction. Ces deux modèles décomposent l'extraction en quatre niveaux et permettent aux auteurs de décrire le processus d'extraction selon une chronologie déterminée par le passage d'un niveau à l'autre.

2.2.2.1 Modèle en fer à cheval de l'extraction d'architectures

KAZMAN [70] propose un modèle pour analyser les processus d'extraction d'architectures. Ce modèle, constitué de quatre niveaux, permet de décrire l'extraction d'architectures comme un ensemble de transitions entre ces niveaux.

Les quatre niveaux du modèle en fer à cheval de Kazman sont :

- **le niveau du code source (*source level*)** : ce niveau est celui des fichiers du code source. C'est une représentation textuelle du code source ;
- **le niveau de la structure du code (*code structure level*)** : ce niveau présente le code source dans une forme plus formelle que le texte du niveau précédent. Ce niveau permet une analyse syntaxique du code source ;
- **le niveau fonctionnel (*function level*)** : ce niveau représente les relations parmi les fonctions, données et modules. Il fournit une vue globale et résumée du système ;

- **le niveau architectural (*architecture level*)** : ce niveau est constitué des éléments architecturaux, i.e. connecteurs et composants.

D'après KAZMAN, la plupart des approches d'extraction débutent au niveau du code source. Ce code source est analysé syntaxiquement et sémantiquement. Cette analyse résulte dans des représentations du code source telles que des arbres syntaxiques qui constituent le niveau de la structure du code. Les représentations du niveau 2 sont ensuite soumises à une analyse plus technique portant, par exemple, sur les flots de données ou de contrôle. Le résultat de cette analyse exprime le niveau fonctionnel. Il représente donc les fonctions et autres modules et leurs relations. Cependant cette vue n'est pas encore suffisamment abstraite pour être considérée comme architecturale. La dernière étape consiste alors à abstraire les regroupements du niveau fonctionnel en entités architecturales (*cf.* Figure 2.1).

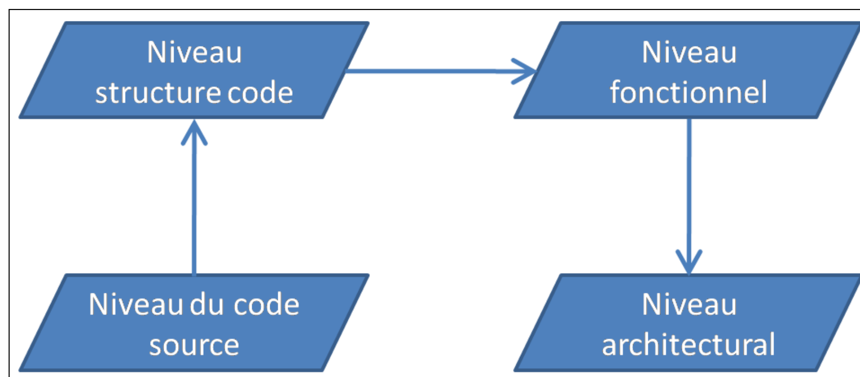


Figure 2.1 – Etapes de l'extraction d'architectures selon le modèle de KAZMAN

2.2.2.2 Modèle conceptuel de l'extraction d'architectures

KOSCHKE [74] propose une amélioration du fer à cheval de Kazman. Il considère en effet que les deux premiers niveaux, que sont le code et la structure du code source, reflètent en réalité le code source du système. Il démontre également que le niveau fonctionnel établit le lien entre le niveau architectural et celui du code source. A ce titre, KOSCHKE précise le contenu de ce niveau en imposant qu'il exprime les regroupements d'entités extraites du code source ainsi que les relations entre ces regroupements et les éléments architecturaux.

Suivant cette critique du fer à cheval de Kazman, KOSCHKE définit les quatre niveaux suivants :

- **le niveau inférieur du code (*lower code level*)** : il contient les expressions et instructions contenues dans les fonctions ;
- **le niveau globale du code (*global code level*)** : ce niveau contient les entités que KOSCHKE appelle les quarks architecturaux. Ces quarks sont les éléments élémentaires de l'extraction d'architecture. Ceux sont ces entités qui doivent être regroupées pour former les éléments architecturaux. Ces entités sont les constantes et variables globales, les fonctions, les définitions de type et les relations entre ces éléments ;
- **le niveau architectural inférieur (*lower architectural level*)** : ce niveau contient les regroupements de quarks. C'est le premier niveau du modèle qui comporte des informations supplémentaires par rapport au code source. Il constitue le ciment entre le code source et la représentation architecturale ;
- **le niveau architectural supérieur (*higher architectural level*)** : ce niveau représente l'architecture du système en termes de sous-systèmes et de connecteurs.

Les niveaux de ce modèle mettent clairement en évidence la séparation entre la partie code source et la partie architecturale :

- **partie code source** : les deux premiers niveaux contiennent les éléments présents dans le code source, ils ne contiennent pas d'information supplémentaire et ne requièrent pas de traitement du code source. Le niveau inférieur du code contient les éléments ignorés dans le processus d'extraction à cause de leur granularité trop faible. Le deuxième niveau contient, lui, les éléments sur lesquels agit le processus d'extraction. Ceux sont les éléments de ce niveau, appelés quarks architecturaux, qui doivent être regroupés pour former le premier niveau de la partie architecturale ;
- **partie architecturale** : le niveau architectural inférieur présente les regroupements de quarks et leurs relations. Ce niveau ne contient pas d'éléments architecturaux puisque le niveau d'abstraction n'a pas changé par rapport au code source. Cependant les regroupements, calculés par le processus d'extraction, représentent une information supplémentaire par rapport au code source. Cette information est également essentielle pour pouvoir utiliser l'architecture dans la maintenance. En effet, c'est ce niveau, qui en établissant les correspondances entre les éléments architecturaux et les entités du code source, permet lors de la maintenance de répercuter les analyses ou les modifications faites sur l'architecture dans le code. Le niveau architectural supérieur contient les éléments architecturaux abstraits depuis les regroupements du niveau architectural inférieur. Le passage entre ces deux niveaux nécessite donc de sélectionner parmi les regroupements du niveau inférieur ceux qui constituent la base des éléments architecturaux (*cf.* Figure 2.2).

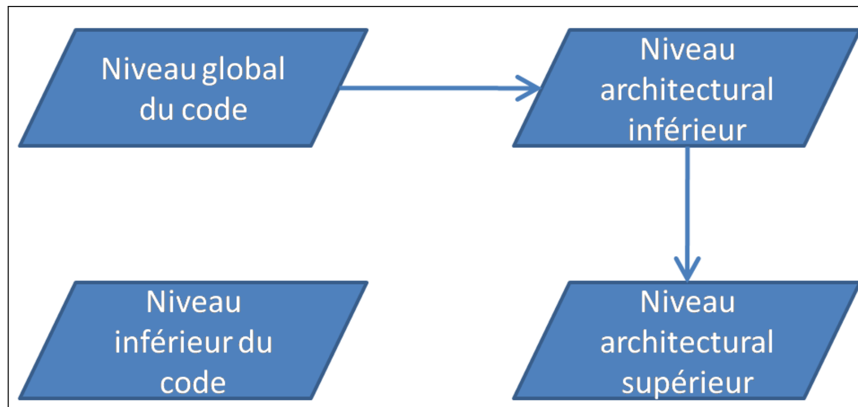


Figure 2.2 – Etapes de l'extraction d'architectures selon le modèle de KOSCHKE

2.2.2.3 Limites des modèles existants

Le problème majeur des modèles de KOSCHKE et de KAZMAN est la description des approches qui reposent sur une définition différente de la notion d'architecture. En effet, bien qu'il existe différents moyens de représenter l'architecture (*e.g.* module, sous-système ou composant), les définitions des niveaux des modèles existant se concentrent uniquement sur les types d'architectures étudiés par leurs auteurs, c'est-à-dire module et sous-système.

Les modèles existants posent un autre problème. Leurs descriptions des phases de l'extraction s'appuient sur les niveaux du modèle, mais elles se révèlent trop linéaires pour décrire certaines approches plus récentes. En effet, elles décrivent uniquement des approches partant du niveau le plus bas vers le niveau le plus haut, alors qu'il existe des approches, plus récentes, qui utilisent une approche hybride, partant à la fois du niveau haut et bas.

2.2.3 Notre modèle conceptuel de l'extraction d'architectures

Face au manque de généricité des modèles de KAZMAN et KOSCHKE, nous proposons un modèle plus général pour la description des approches d'extraction des architectures logicielles. Ce modèle repose sur quatre niveaux conceptuels, ainsi qu'une description des objectifs et méthodologies des travaux d'extraction.

Les quatre niveaux conceptuels. Notre présentation de l'extraction repose sur un modèle composé de quatre niveaux :

- **le niveau du code source** : ce niveau est celui du système. Il contient le code source dans sa représentation textuelle. Il correspond au premier niveau proposé par KAZMAN et à l'union des deux niveaux du code source de KOSCHKE ;
- **le niveau modèle du code source** : ce niveau contient un modèle du code source. Les entités de ce modèle sont des éléments du code source et dépendent de la granularité de l'approche. Ces entités correspondent aux éléments que KOSCHKE appelle les quarks architecturaux. Ce modèle doit permettre une analyse syntaxique et sémantique du code source. Ce niveau correspond à peu prêt au niveau de la structure de code et au niveau global du code ;
- **le niveau de mise en correspondance** : ce niveau correspond au niveau architectural inférieur de KOSCHKE. Ce niveau contient des regroupements des entités du niveau modèle du code source. Ce niveau établit le lien entre les éléments architecturaux et les entités du code source ;
- **le niveau architectural** : ce niveau est celui de l'architecture du système. Il contient les éléments architecturaux, composants et connecteurs, qui composent le système représenté par le niveau du code source. Il correspond aux derniers niveaux des deux modèles précédents. La différence majeure est que notre niveau ne préjuge pas du type des composants utilisés pour décrire le système.

Objectifs de l'extraction. L'objectif d'une extraction est d'obtenir le niveau architectural à partir du niveau du code source, c'est-à-dire le système, . Cependant, la construction du niveau de mise en correspondance est également un objectif essentiel de nombreuses approches. En effet, si le niveau architectural permet de détecter, comme dit précédemment, les erreurs de conception ou les cibles de l'évolution, l'évolution ou la correction associée doit se faire également dans le code source. Cette correction dans le code source nécessite de connaître les correspondances entre les éléments architecturaux et les entités du code source.

Méthodologie de l'extraction. Le niveau modèle du code source est l'élément caractéristique d'une approche d'extraction. Ce niveau détermine la granularité de l'architecture à extraire. Ainsi, les éléments du modèle peuvent être, comme pour KOSCHKE, les constantes globales ou, pour une granularité plus forte, les classes. Le passage du niveau code source au niveau modèle consiste essentiellement en une sélection des entités d'intérêts. Cependant, ce modèle peut également contenir des informations qui ne sont pas directement contenues dans le code source. Dans certains modèles, les relations entre entités sont de deux types : certaines, syntaxiques, qui sont présentes explicitement dans le code et d'autres, sémantiques, qui ne sont pas présentes dans le code ou alors de manière implicite.

Le passage du niveau modèle vers le niveau de correspondance est le point crucial des approches d'extraction. Ce passage repose sur une analyse du modèle et parfois sur l'utilisation d'informations extérieures à ce modèle, telle que l'expertise des utilisateurs. Ce passage s'effectue par regroupement au sein des entités du modèle. Le passage entre le niveau de correspondance et celui de l'architecture requiert une sélection des regroupements qui peuvent être abstraits en un élément architectural. Ce choix

repose en général sur les mêmes critères qui ont permis les regroupements lors du passage du niveau du modèle à celui de correspondance (*cf.* Figure 2.3).

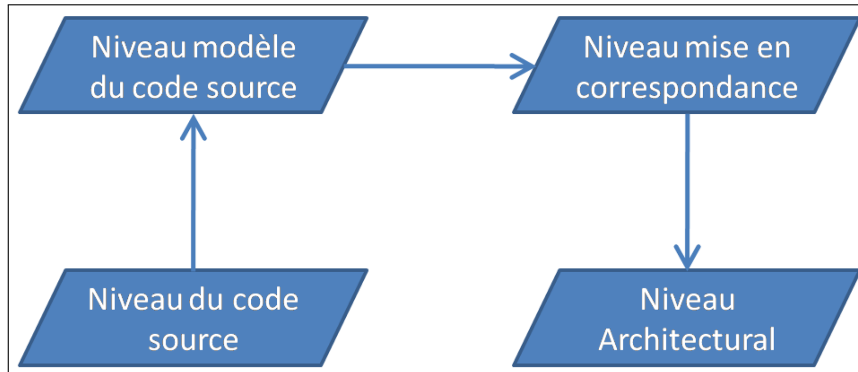


Figure 2.3 – Etapes possibles de l'extraction d'architectures selon notre modèle

La plupart des approches d'extraction suivent la succession des trois derniers niveaux présentée précédemment. Cependant, certaines approches récentes suivent un enchainement différent. Pour ces approches, le dernier niveau est celui de correspondance. Ces travaux génèrent d'abord le niveau du modèle et celui de l'architecture puis celui de correspondance. Ainsi, la technique centrale de ces approches n'est plus le regroupement, comme les autres travaux d'extraction, mais la mise en correspondance. Cette mise en correspondance est réalisée entre les entités du modèle et les éléments architecturaux. La génération du niveau de l'architecture, dans ces approches, n'est pas faite depuis le niveau de correspondance. A la place, elles utilisent l'expertise des utilisateurs, la documentation et parfois le style architectural du système (*cf.* Figure 2.4).

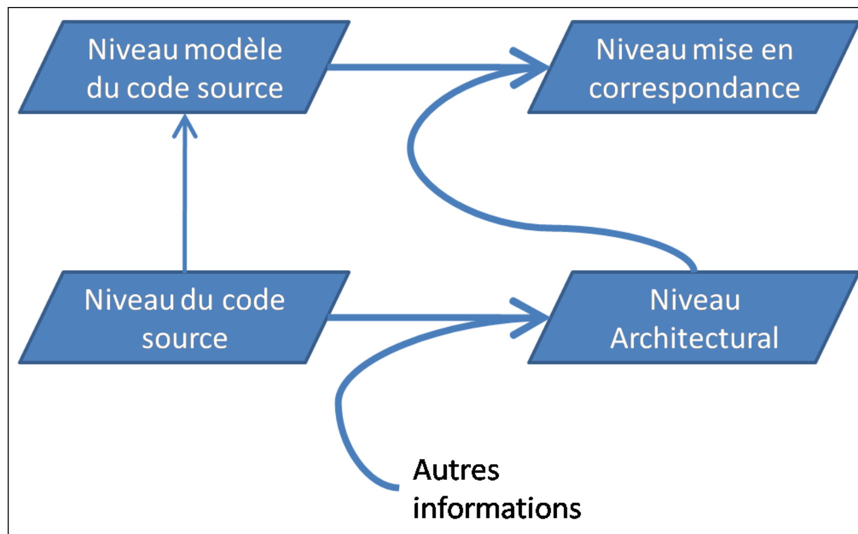


Figure 2.4 – Autres étapes possibles de l'extraction d'architectures selon notre modèle

Notre modèle d'analyse et notre description de l'extraction d'architectures donne une vue générique du problème posé et des approches existantes. Mais, cette description met surtout en valeur la diversité

des voies permettant d'aborder le problème d'extraction ainsi que celle des approches existantes. Face à cette diversité, le modèle et la description proposés sont insuffisants pour appréhender et comparer l'ensemble des approches existantes. Pour permettre cette comparaison, nous proposons dans la suite un cadre de comparaison des approches d'extraction d'architecture.

2.3 Comparaison des approches d'extraction d'architectures

Si les approches d'extraction ont toutes le même objectif, les différences entre elles sont nombreuses. Cependant, ces différences sont difficiles à mettre en évidence sans un cadre de comparaison adapté. Ce cadre doit définir les points de comparaison pertinents entre les travaux ainsi que les différentes alternatives. Même si les modèles conceptuels, présentés précédemment, sont cruciaux pour appréhender les problèmes posés par l'extraction d'architecture, ils sont insuffisants pour répondre à ce besoin de comparaison et permettre une classification efficace et pertinente des travaux existants.

Pour permettre ces comparaisons, nous proposons d'abord un cadre, basé sur notre modèle conceptuel de l'extraction. Nous décrivons ensuite chacun des axes de notre classification.

2.3.1 Cadre de comparaison des approches d'extraction d'architectures

Les critères de notre cadre de comparaison peuvent être classés selon trois axes :

- **axe des propriétés conceptuelles** : les critères de cet axe reposent sur les propriétés des différents niveaux de notre modèle conceptuel. Ils précisent le type de code source (niveau du code source), les éléments du modèle du code source (niveau du modèle du code source), les propriétés des regroupements (niveau de mise en correspondance), les propriétés de l'architecture (niveau de l'architecture) et enfin l'enchaînement des niveaux ;
- **axe des informations utilisées** : cet axe concerne les informations, autres que le code source, qui sont utilisées dans le processus. Les critères précisent les sources et les types d'informations, le moment et le mode d'intervention de l'architecte ;
- **axe des propriétés techniques** : cet axe repose sur les propriétés techniques de l'approche telle que l'algorithme, ses paramètres et son degré d'interactivité. Les critères précisent le critère de regroupement, l'algorithme, le degré d'automatisation et le mode d'utilisation de l'approche.

2.3.2 Axe des propriétés conceptuelles

Le modèle conceptuel de l'extraction, que nous avons présenté, permet de décrire le cœur d'une approche d'extraction. Les quatre niveaux de ce modèle permettent de caractériser une approche et de la distinguer de plusieurs autres. Mais, comme nous l'avons vu, ses niveaux ne permettent pas de distinguer et de comparer tous les travaux. Cependant, même si ce modèle est insuffisant pour constituer à lui seul un cadre de comparaison, ses niveaux constituent les éléments fondamentaux d'un cadre pertinent. Par conséquent, nous avons étudié les propriétés de ces niveaux et nous proposons un ensemble de critères pour chacun d'eux afin de comparer les différents processus d'extraction (*cf.* Figure 2.5). Par la suite, nous présentons, en fonction du niveau conceptuel, l'ensemble de ces critères de comparaison et les différentes alternatives pour chacun.

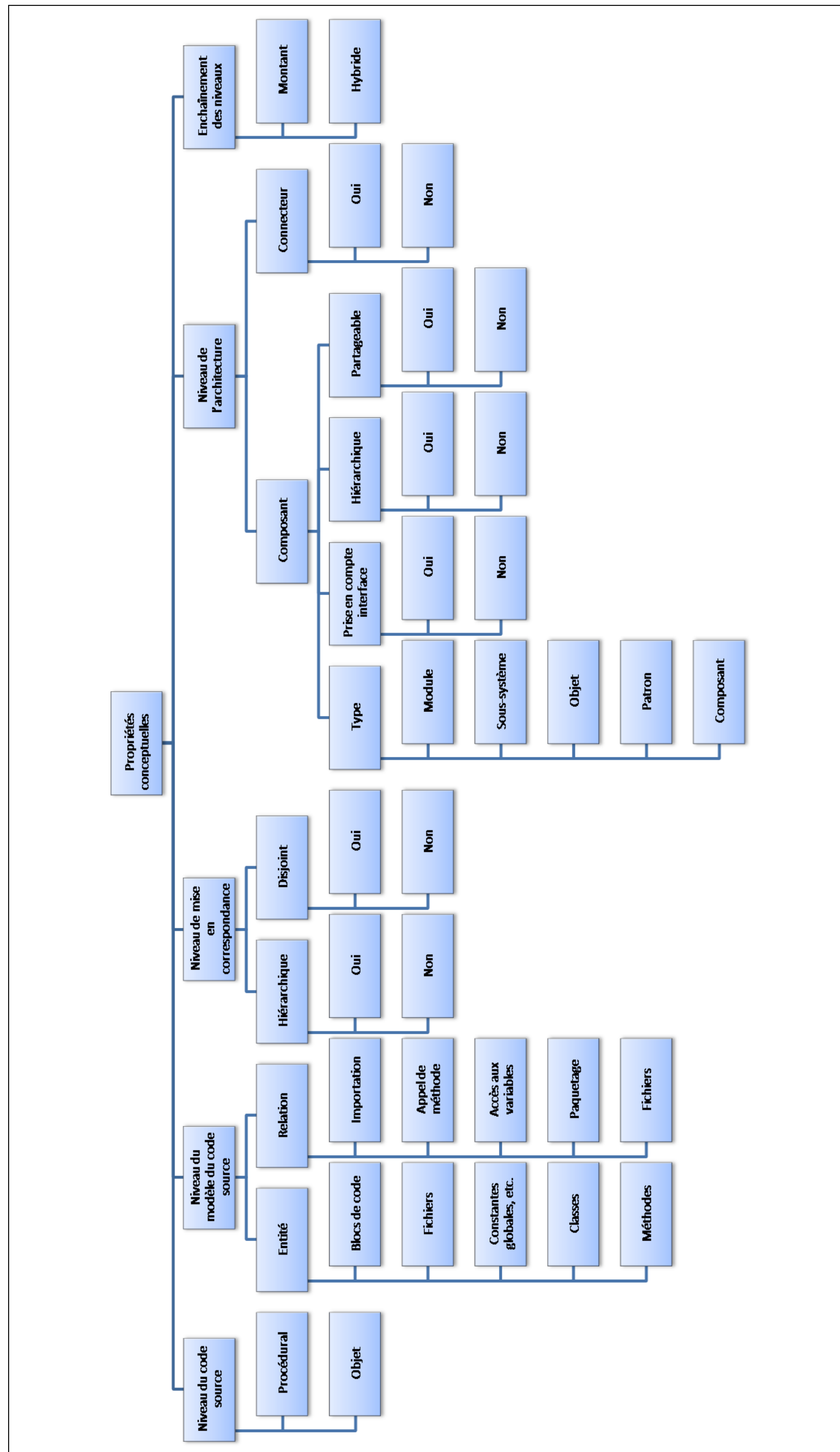


Figure 2.5 – Critères de comparaison de l'axe propriétés conceptuelles

2.3.2.1 Niveau du code source

Comme nous l'avons vu, le niveau du code source est le reflet du système cible du processus d'extraction. Ce système constitue la principale source d'information du processus. A ce titre, le paradigme utilisé est un facteur déterminant de la plupart des caractéristiques de l'extraction et constitue le premier de nos critères.

Il existe aujourd'hui principalement trois paradigmes : procédural, objet et composant. Le paradigme composant est le plus récent. A ce titre, les programmes suivant ce paradigme ne représentent pas une fraction importante des programmes existants. De plus, ces programmes récents n'ont pas encore subi l'érosion. Enfin, le paradigme qu'ils suivent imposent à ces programmes, une conception utilisant l'architecture et permet une synchronisation aisée entre le code et l'architecture. En conséquence, ces systèmes à base de composants ne sont pas des cibles pour les approches d'extraction actuelles.

Les approches existantes se concentrent sur les paradigmes procédural [19, 25, 39, 61, 43, 69, 74, 77, 95, 110, 123] et objet [4, 5, 35, 76, 93, 88, 111, 126]. Ces deux paradigmes représentent une grande part des systèmes patrimoniaux. A ce titre, ils sont à la base des systèmes qui nécessitent le plus une extraction de l'architecture. De plus, pour chacun de ces paradigmes, l'extraction d'architectures constitue le point de départ d'un processus de migration vers un paradigme plus récent : objets pour les systèmes procéduraux et composants pour les systèmes à base d'objets.

2.3.2.2 Niveau du modèle du code source

Les propriétés principales de ce niveau sont les entités et les relations qui constituent le modèle. Elles déterminent directement les éléments mis en correspondance dans le niveau suivant.

Les entités du modèle du code source. Les entités de ce modèle du code source sont les éléments sur lesquels se concentrent le processus d'extraction. En effet, les approches d'extraction établissent des liens entre ces entités et les éléments architecturaux. Certains travaux [74], regroupent les constantes, les variables globales et les définitions de types. Ces regroupements sont alors identifiés à des composants architecturaux. Les entités de niveau du modèle du code source sont par exemple : des blocs de code, des fichiers [4], des constantes et variables globales[110], des définitions de types [74], des méthodes ou encore des classes[88, 111].

Ces entités de base déterminent les liens entre le paradigme caractérisant le niveau du code source et les éléments du niveau architectural. Ils influencent également le niveau de granularité de l'architecture résultante de l'extraction. En effet, une approche basée sur les méthodes du paradigme objet fournit des composants architecturaux à partir de regroupement de méthodes. Ces composants seront en général plus petits que des composants extraits par regroupement de classes.

Les relations du modèle du code source. Les relations présentes dans le modèle du code source sont utilisées pour diriger les regroupements et les associations effectuées entre les entités du modèle. Les relations présentes dans le modèle ont donc une influence directe sur le processus d'extraction puisqu'elles permettent de diriger la construction des entités qui seront mises en correspondance avec les éléments architecturaux dans le niveau de mise en correspondance. Ces relations peuvent être, par exemple, des appels de méthodes, de fonctions, l'appartenance à un même paquetage, l'importation d'un paquetage ou encore le partage d'une variable.

2.3.2.3 Niveau de mise en correspondance

Le critère de comparaison principal, pour ce niveau, est la méthode de regroupement. Cependant, ce critère est considéré comme une propriété technique et sera abordé dans la suite. Deux autres propriétés intéressantes sont liées aux propriétés de l'ensemble des regroupements :

- **regroupements hiérarchiques** : la première propriété concerne le type des entités constituant un groupe. Elle détermine si les regroupements effectués par le processus d'extraction sont hiérarchiques ou non, c'est-à-dire si un groupe contient uniquement des éléments du modèle du code source ou bien si il peut contenir d'autres groupes. L'approche hiérarchique est utilisée uniquement par les approches traitant du paradigme procédural. Par exemple, les regroupements de variables et autres constantes globales peuvent être hiérarchisés [74, 69] ;
- **regroupements disjoints** : le second critère concerne la répartition des entités dans les regroupements. En effet, les entités peuvent appartenir à plusieurs regroupements. Selon que l'approche autorise ce partage ou non, l'ensemble des regroupements forme une partition des entités du modèle (entités non partagées) ou uniquement une couverture (entités partagées). Beaucoup des approches existantes ne précisent pas si les regroupements doivent être disjoints mais dans les faits ils sont effectivement disjoints. Les autres approches imposent que les entités ne soit pas dupliquées et que les regroupements soient disjoints.

2.3.2.4 Niveau de l'architecture

Le niveau de l'architecture est caractérisé par les différents éléments architecturaux. Les critères de ce niveau portent sur les types de composants et de connecteurs qui constituent ce niveau de l'architecture. Ces critères précisent l'architecture extraite et par conséquent les utilisations possibles de ce résultat.

Type de composants. Comme nous l'avons vu dans le chapitre 1, les définitions de l'architecture varient légèrement en fonction du paradigme et de la communauté. Ces variations induisent des types d'architectures qui varient suivant la signification associée aux éléments architecturaux. Les approches d'extraction sont soumises aux mêmes variations et peuvent donc être classées suivant le type d'architectures qu'elles extraient.

Les travaux visent à extraire des composants représentés par diverses entités des paradigmes de programmation. Ces entités peuvent être des modules [4, 5, 86, 61, 123] ou des sous-systèmes [74, 19, 25]. Elles peuvent aussi être des classes[110], des patrons de conception ou encore des composants logiciels [88].

Interfaces des composants. Un autre critère important pour la description des composants est la place accordée aux interfaces des composants. En effet, ces interfaces sont des éléments essentiels des composants et leur prise en compte est cruciale pour bénéficier de tous les avantages de l'architecture. Cependant, les interfaces des composants sont rarement prises en compte. En fait, la majorité des approches d'extraction n'extraient pas les interfaces et fournissent uniquement un regroupement d'entités identifiées à un composant.

Autres propriétés des composants. Ces critères doit être rapprochés de ceux concernant le niveau de de mise en correspondance. En effet, comme pour les regroupements, les composants extraits peuvent être hiérarchiques ou non et ils peuvent être partageables ou non. Ces critères sont, dans toutes les approches, directement liés à ceux concernant le niveau de mise en correspondance.

Prise en compte des connecteurs. Concernant les connecteurs, la quasi-totalité des approches existantes les considèrent comme des liens simples entre les composants extraits. Les approches réalisent l'extraction des composants et les connecteurs sont une conséquence de cette extraction qui n'est pas directement prise en compte.

2.3.2.5 Enchainements des niveaux

Le dernier critère du niveau conceptuel concerne l'enchainement des différents niveaux du modèle conceptuel. Il existe deux types approches possibles. La première est une approche montante. Celle-ci débute au niveau du code source et remonte la hiérarchie des niveaux en effectuant des regroupements dans les entités du modèle du code source puis en identifiant à partir de ces regroupements les éléments architecturaux. C'est l'approche la plus utilisée [35, 76, 93, 111, 126].

Le deuxième type est une approche hybride. Dans ces approches [61, 43, 74, 88, 123] le point de départ est à la fois le niveau de l'architecture et celui du modèle du code source. L'architecture est celle prévue pour le système. Le processus effectue des regroupements au sein des entités du modèle du code source, puis il tente de mettre en relation les éléments architecturaux et les regroupements. Cette étape de mise en relation réclame, alors, des modifications dans les regroupements ou les éléments architecturaux. L'ampleur de ces modifications est en lien direct avec le niveau de décalage de l'architecture prévue utilisée par l'approche et l'architecture réelle du système.

2.3.3 Axe des informations utilisées

Les critères portant sur le modèle conceptuel ne permettent pas de décrire tous les aspects des approches d'extraction. En particulier, ce modèle ne décrit qu'un seul élément en entrée de processus : le code source. Cependant, l'extraction s'appuie également sur les interventions de l'utilisateur et sur la documentation existante du système. A partir de l'étude des facteurs influençant l'extraction, nous avons déterminé un ensemble de critères décrivant l'intervention de l'utilisateur et la documentation utilisée (*cf.* Figure 2.6).

2.3.3.1 Intervention de l'utilisateur

Le code source constitue la source commune d'information pour l'extraction. Cependant, les personnes réalisant l'extraction constituent une autre source d'information pertinente. En effet, ces utilisateurs possèdent, dans la majorité des cas, une certaine expertise sur le système étudié. La façon dont ces connaissances sont prises en compte durant l'extraction permet d'établir plusieurs critères de comparaison entre les approches d'extraction.

Moment de l'intervention. Les informations disponibles auprès des utilisateurs sont utilisables à tout moment dans le processus d'extraction. On distingue trois moments possibles pour utiliser ces informations. La première façon implique l'utilisateur en amont de l'extraction [43, 69, 95]. L'utilisateur spécifie toutes les informations avant l'extraction en utilisant différents paradigmes plus ou moins formels. Ces informations permettent de réduire l'espace des solutions possibles de l'extraction.

Le second moment de l'implication de l'utilisateur est en aval de l'extraction [39, 86, 111]. A ce moment, l'utilisateur peut, à partir des premiers résultats de l'extraction, spécifier les parties qui lui semblent correctes et celles qui semblent incorrectes.

La dernière possibilité est l'intervention de l'utilisateur tout au long du processus d'extraction [61, 72, 88]. Dans ce cas, l'utilisateur peut fournir des informations tout au long de l'extraction, en rejetant

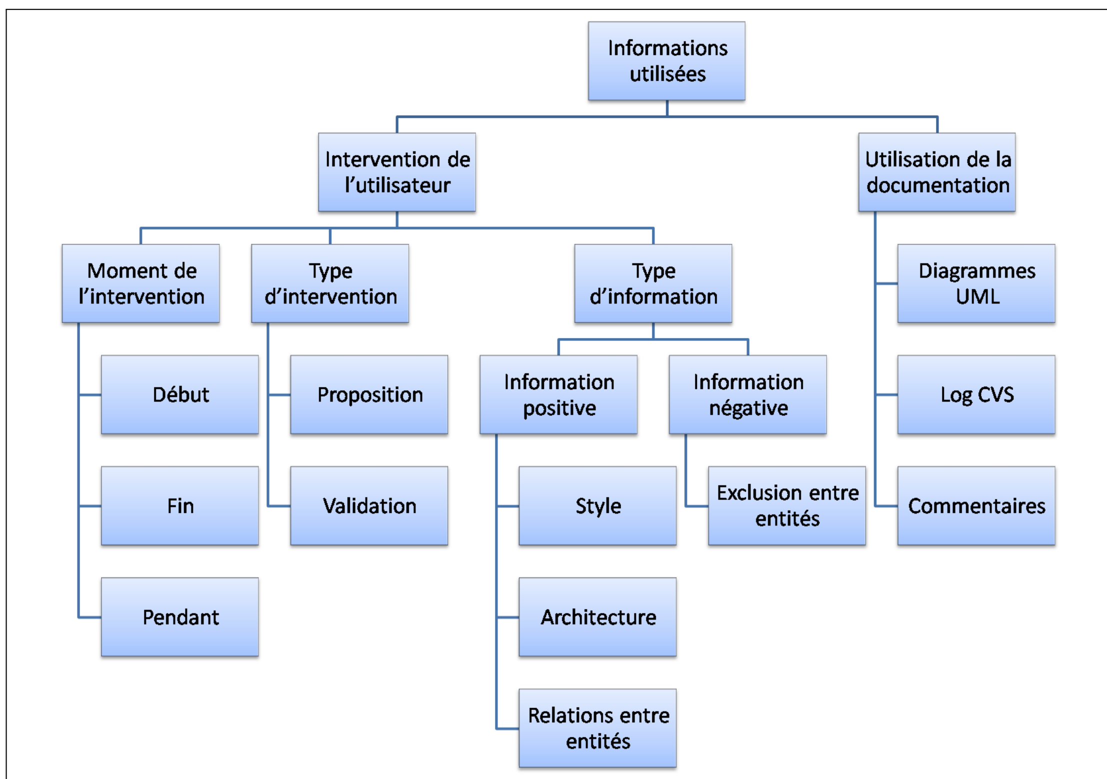


Figure 2.6 – Critères de comparaison de l'axe des informations utilisées

certaines solutions et en favorisant certaines. Suivant ce mode d'intervention, l'utilisateur a un impact plus grand sur le processus. Cet impact est évidemment dû à la multiplication des interventions, mais il est surtout le résultat de la plus grande diversité d'informations qui peuvent être fournies pendant, avant ou après le processus.

Type d'intervention. Le moment de l'intervention de l'utilisateur est lié au type de ces interventions. Ces interventions sont de deux types. L'utilisateur peut proposer des informations permettant de diriger le processus et de réduire l'espace des solutions [61, 43, 72, 69, 88, 95]. Ces propositions concernent les interventions réalisées avant ou pendant l'extraction. Par exemple, les utilisateurs peuvent spécifier les classes ou les définitions de type qui constituent un composant.

La validation est une seconde voie d'intervention pour l'utilisateur. L'utilisateur doit alors donner son avis sur des propositions du processus. Les informations de validation ou de rejet sont utilisées pour confirmer ou infirmer les voies explorées par le processus. Ce type d'intervention est utilisé pendant ou après l'extraction.

Type d'information. Le type d'information constitue un troisième aspect des interventions de l'utilisateur. En effet ces interventions varient suivant leur moment et leur type, mais elles varient aussi en fonction du type des informations : positives ou négatives.

Les informations positives précisent ce que doit être l'architecture. Ce type d'informations est utilisé dans chaque type d'interventions [61, 43, 74, 88, 123]. Dans les interventions informatives, ces informations expriment ce que l'utilisateur souhaite voir apparaître dans la solution. Ces informations peuvent être, par exemple, le style, une architecture probable ou encore une relation entre entités. Par contre, dans les interventions de validation, ces informations expriment les éléments de la solution qui conviennent à l'utilisateur. Dans ce cas, ces informations peuvent indiquer qu'un groupe d'entités est un bon composant ou forme un connecteur.

Les informations négatives indiquent ce qui ne doit pas être dans l'architecture. Ce type d'informations est aussi utilisable dans chaque type d'intervention. Quel que soit le type ou le moment de l'intervention, ces informations expriment des éléments qui ne doivent pas exister dans l'architecture. Ces éléments peuvent concerner des relations entre entités ou l'appartenance de deux entités au sein d'un même groupement. Malgré les avantages qu'il fournit, ce type d'information est peu utilisé par les travaux existants.

2.3.3.2 Utilisation de la documentation

Les informations obtenues à travers les utilisateurs sont très variables. Elles sont soumises à l'expertise des personnes réalisant l'extraction. Les éléments de documentation constituent, quant à eux, une source plus régulière d'informations. En effet, ces documents, créés pendant le développement et la maintenance des systèmes, contiennent de nombreuses informations accumulées par des personnes expertes du système. Ces documents sont donc régulièrement utilisés durant l'extraction. A ce titre, Les différents documents utilisés sont un critère de notre cadre de comparaison des approches d'extraction.

Les différents diagrammes UML sont les documents les plus couramment utilisés pour décrire un système. Ces diagrammes contiennent une image de l'architecture du système tel qu'il a été conçu. Ils peuvent donc être utilisés pour identifier des relations entre entités du code source qui n'apparaissent pas dans le code mais qui existent dans l'esprit des créateurs du système. Cependant, ces informations s'appuient sur une version idéalisée du système qui ne correspond pas forcément au système réel. Ainsi, ces diagrammes ne constituent pas une source parfaitement sûre. Malgré cela, les approches d'extraction

utilisant ces documents, ne fournissent pas de méthodes pour prendre en compte cette insuffisance [88]. En effet, dans tous les cas, les diagrammes sont utilisés à travers l’expertise de l’utilisateur qui doit vérifier la validité des informations qu’il fournit au processus d’extraction.

Le code source, qui constitue la base de toutes les approches d’extraction, contient également des informations sur l’architecture prévue du système [4, 76, 126]. Ces informations sont représentées par les commentaires contenus dans le code et par les différents noms des entités du code source. En effet, ces éléments décrivent souvent une intention des programmeurs. Comme les diagrammes UML, la validité de ces documents est variable, mais au vu de leur proximité avec le code, ces informations semblent plus fiables.

Il existe d’autres documents qui accompagnent un système pendant son cycle de vie et de maintenance. Pourtant, les approches d’extraction se concentrent surtout sur les deux types de documents précédents.

2.3.4 Axe des propriétés techniques

Les propriétés techniques de chaque approche constituent un point de comparaison aussi pertinent que les informations utilisées. Les critères de cet axe fournissent une comparaison basée sur des aspects techniques du processus d’extraction qui n’apparaissent pas dans le modèle conceptuel. Ces aspects, tel que l’algorithme, constituent un point central dans la description d’une approche d’extraction. Ces propriétés techniques sont donc des éléments essentiels à prendre en compte pour réaliser une comparaison correcte des processus d’extraction (*cf.* Figure 2.7).

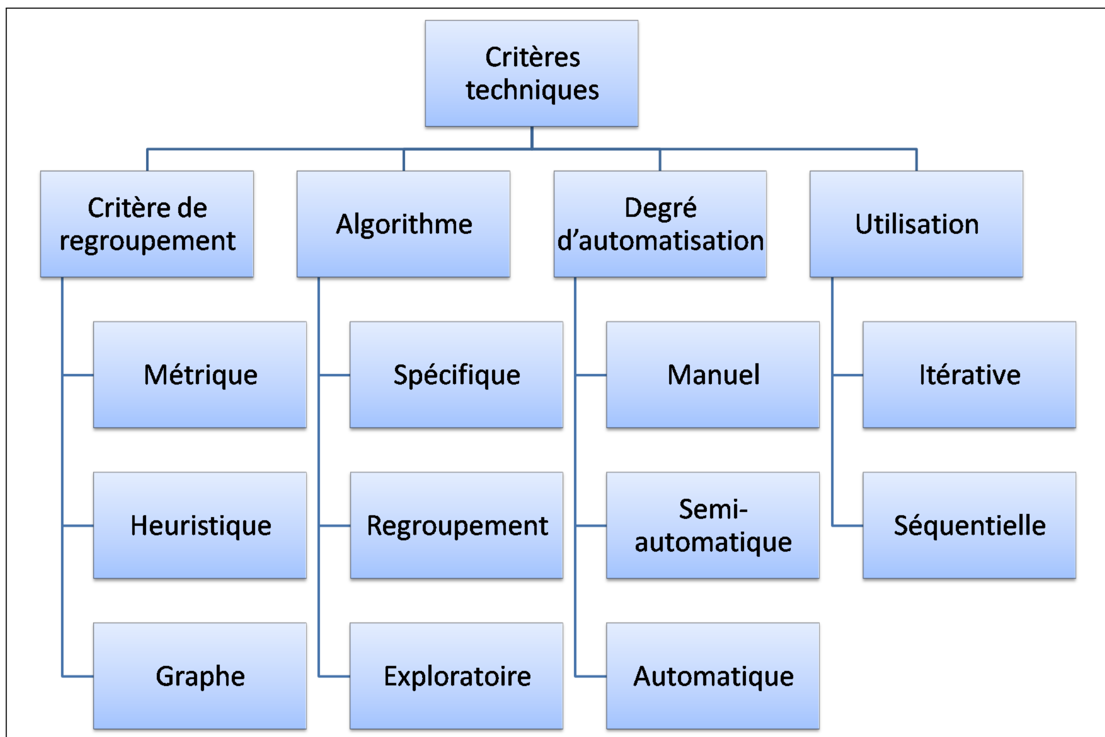


Figure 2.7 – Critères de comparaison de l’axe des propriétés techniques utilisées pour l’extraction

2.3.4.1 Critères de regroupement

Le premier critère technique de comparaison concerne le critère de regroupement des entités du code source. Ces critères définissent la méthode utilisée pour déterminer quelles entités du code source constitue un groupe pouvant être identifié à un élément architectural. Ces critères sont principalement de trois types : les métriques, les graphes et les heuristiques.

Les métriques. Les métriques constituent un des critères principaux de regroupement des entités du code source. Les approches utilisant ce critère définissent un ensemble de propriétés que doivent vérifier les entités regroupées [5, 39, 86, 93, 87, 111]. Les métriques permettent de mesurer ces propriétés dans le code source et donc de déterminer les regroupements d'entités. Ainsi, elles sont utilisées pour mesurer la force des relations entre les entités, la qualité des entités ou des regroupements d'entités. Ce critère permet d'automatiser une grande partie du processus en choisissant les regroupements en fonction des métriques calculées automatiquement sur le code source.

Les graphes. D'autres approches définissent des propriétés sur des graphes [91]. Dans ces approches, le modèle du code source est représenté par un graphe. Le processus utilise alors un ensemble de propriétés de ce graphe pour diriger les regroupements d'entités. Ces propriétés peuvent être la connexité [33] ou le degré des sommets du graphe. Ce critère permet souvent d'automatiser les regroupements. Cependant, certaines propriétés doivent être soumises à des seuils. Ces seuils impliquent une intervention de l'utilisateur pour les fixer et limitent donc l'automatisation.

Les heuristiques. Beaucoup d'approches proposent une heuristique pour déterminer les regroupements possibles entre les entités. Ces approches proposent un ensemble de règles qui définissent les regroupements à effectuer [43, 72, 69, 88]. Ces règles sont plus ou moins automatisables mais dans la majorité des cas, les approches utilisant des heuristiques sont fortement manuelles. Ces règles peuvent, par exemple, définir des regroupements à faire parmi les entités du code source, l'utilisateur devant en fonction de son expertise décider de l'ordre d'application de ces règles [88].

2.3.4.2 Algorithmes

L'algorithme utilisé par chaque approche d'extraction est un élément crucial pour déterminer le comportement de ces approches en termes de performance, de précision et d'automatisation. Nous distinguons trois types d'algorithmes : les algorithmes de regroupement, les algorithmes d'exploration et enfin les algorithmes spécifiques.

Algorithmes de regroupement. Les algorithmes de regroupement ou « *clustering algorithms* » visent à regrouper un ensemble d'entités en différents groupes homogènes, en ce sens que les éléments de chaque groupe partagent des caractéristiques communes, qui correspondent le plus souvent à des critères de proximité que l'on définit en introduisant des mesures de similarité. Ces algorithmes sont utilisés pour extraire l'architecture d'un système en regroupant les entités du code source.

Pour guider ces regroupements, les approches définissent des mesures de similarité différentes. Ces mesures dépendent des critères de regroupement de l'approche. Comme ces algorithmes sont automatiques, le critère de regroupement associé peut être des métriques [5, 87] ou des propriétés de graphes [91].

Algorithmes d'exploration. Les algorithmes d'exploration reposent sur un principe opposé à ceux des algorithmes de regroupement. Dans ces algorithmes, l'objectif n'est pas de créer des groupes d'entités mais de sélectionner la meilleur répartition des entités en groupes. Ainsi, dans un processus par exploration une solution est disponible dés la première itération et l'algorithme vise à augmenter la qualité de la solution trouvée en explorant l'espace des solutions.

Les approches d'extraction utilisent ces algorithmes en définissant différentes mesures de qualité et différents espaces de solution. Les mesures sont définies en fonction du critère de regroupement et, comme pour les algorithmes de regroupement, elles nécessitent des critères automatisables et excluent donc l'utilisation d'heuristiques. L'espace des solutions est lui défini en fonction des niveaux du modèle du code source et de mise en correspondance.

Les algorithmes de recuit simulé et d'exploration tabou sont des exemples typiques d'algorithmes d'exploration utilisés par les approches d'extraction [93]. Les algorithmes génétiques constituent un autre type d'algorithmes qui commence à être utilisé pour l'extraction de l'architecture [39, 111].

Algorithmes spécifiques. Les premières approches d'extraction d'architectures ont proposé différents algorithmes ou guides pour obtenir l'architecture. Ces travaux proposent un critère de regroupement sous la forme d'un ensemble de règles basées sur une heuristique. L'algorithme proposé consiste alors en un guide d'utilisation et d'application des règles [19, 43, 72, 69, 95, 88].

2.3.4.3 Degré d'automatisation

Le degré d'automatisation du processus d'extraction est avant tout lié à l'algorithme sur lequel il repose. Cependant, les approches proposant des algorithmes spécifiques sont plus diverses dans leurs degrés d'automatisation. Les niveaux d'automatisation vont d'entièrement manuels à entièrement automatiques.

Processus manuel. Les approches utilisant un algorithme spécifique et un critère de regroupement basé sur une heuristique sont majoritairement manuelles [19, 43, 72, 88]. Elles proposent des règles et des guides pour appliquer ces règles et réaliser l'extraction. Ces approches manuelles requièrent une certaine expertise de la part de l'utilisateur dans le système étudié et dans le domaine de ce système.

Processus semi-automatique Les approches semi-automatiques sont de deux types. Ces approches, semblables aux approches automatiques, proposent en faite des mécanismes pour automatiser certains aspects de l'extraction. Ces mécanismes permettent ainsi de réaliser de manière automatique certains passages répétitifs du processus d'extraction. Néanmoins, cette automatisation ne portant que sur des aspects simples du processus, elle ne permet pas de réduire le besoin en expertise. Ces approches nécessitent donc le même niveau d'expertise de la part de l'utilisateur que les approches manuelles. Le premier type représente des approches reposant sur une heuristique et un algorithme spécifique [69, 95].

Le second type d'approches semi-automatiques est constitué d'approches qui sont en grande partie automatisées, mais qui permettent à l'utilisateur d'intervenir pour initier ou rediriger le processus. Ces approches sont très différentes du premier type. Elles sont en effet beaucoup moins dépendantes de l'expertise de l'utilisateur puisqu'elles automatisent toutes les phases de l'extraction et permettent à l'utilisateur de donner des indications sur la marche à suivre. Elles reposent sur des algorithmes de regroupement et d'exploration associés à un critère de regroupement à base de métriques ou de graphes [93].

Processus automatique Les approches automatiques reposent sur des algorithmes de regroupement ou d'exploration et sur des critères de regroupement à base de métriques ou de graphes. Ces approches automatisent toutes les phases de l'extraction [5, 39, 111]. Elles ne nécessitent pas d'intervention de l'utilisateur. Cela permet de réduire à zéro le besoin d'expertise et permet d'obtenir une architecture sans effort. Cependant, cet automatisation interdit les interventions de l'utilisateur pendant le processus. Dans ces approches, la seule façon pour l'utilisateur d'influencer le processus est donc de faire varier les différents paramètres modifiables du processus. Ces paramètres peuvent être des poids dans les métriques constituant les critères de regroupement, par exemple.

2.3.4.4 Les modes d'utilisation

Un autre critère de comparaison entre les approches est le mode d'utilisation du processus d'extraction. En effet, selon les approches, le processus peut être itératif ou séquentiel.

Processus séquentiel. Les approches séquentielles sont les plus courantes. Dans ces approches, le processus d'extraction est exécuté une seul fois et il ne prend en compte aucune exécution précédente. L'affinage des résultats nécessite alors d'exécuter une nouvelle fois le processus en modifiant les paramètres et les indications fournies par l'utilisateur.

Processus itératif. Une autre approche possible est d'utiliser le processus d'extraction de manière itérative en permettant au processus de réutiliser les résultats des exécutions précédentes. Cette approche permet d'affiner progressivement les résultats de manière automatique. Elle permet donc de réduire le niveau d'expertise nécessaire. Cette approche est cependant peu utilisée dans les travaux existants

2.4 Conclusion

Nous avons souligné, dans ce chapitre, le lien fort qui existe entre l'architecture logicielle et la maintenance des systèmes informatiques. Cependant, nous avons également montré que beaucoup de systèmes existants ne disposent pas d'une représentation fiable de leurs architectures. Cette indisponibilité peut provenir d'une absence totale de représentation architecturale ou d'un décalage progressif entre la représentation et la réalité du système au cours du développement et de l'évolution du système. Dans ce sens, nous avons détaillé les problèmes que provoque l'utilisation d'une représentation fausse d'un système.

A partir de ces considérations, nous avons étudié les travaux portant sur l'extraction d'architectures logicielles. Pour cela, nous avons d'abord présenté les principes de ces travaux. La description de ces principes s'appuie sur un modèle conceptuel que nous avons proposé pour pallier les limites des modèles existants.

Nous avons proposé un cadre de comparaison des approches d'extraction reposant sur trois axes. Le premier correspond au modèle conceptuel et les deux suivants aux informations et aux techniques utilisées. Nous avons présenté une classification des principaux travaux d'extraction d'architectures en fonction de ce cadre de comparaison. Nous avons ainsi pu observer que la majorité des approches apportent des réponses similaires et que certains choix de critères ne sont jamais présents dans les approches comparées.

PARTIE II

La démarche ROMANTIC pour l’extraction d’architectures

CHAPITRE 3

Problématique et approche de ROMANTIC

It is better to have an approximate answer to the right question than an exact answer to the wrong one.

— J. TUKEY.

Motivation — Vue générale de ROMANTIC — Modèle du code source — Espace de solutions — Guides de l'extraction

Nous avons présenté, dans la partie I, les concepts clés de l'architecture logicielle ainsi que son impact sur le cycle de vie du logiciel. Nous avons également montré la nécessité d'extraire l'architecture de certains systèmes. Notre étude des travaux d'extraction a ensuite mis en lumière certains manques dans les approches d'extraction existantes.

Ces manques motivent nos travaux et permettent d'expliquer et de fixer le cadre et les hypothèses de notre approche. Nous avons ainsi développé une approche d'extraction selon deux particularités majeures. La première vise à réduire le besoin en expertise des phases d'extraction afin de limiter le coût de cette opération que ce soit en temps ou en argent. La deuxième consiste à extraire une architecture à base de composants explicite, c'est-à-dire en identifiant chaque élément architectural et tous les éléments qui le composent.

Pour décrire cette approche, nous en présentons d'abord une vue générale. Cette vue est basée sur les différents manques identifiés dans les travaux existants. Elle constitue le cadre de notre approche et en fixe les aspects majeurs : une architecture logicielle explicite à base de composants, permettant les interventions de l'utilisateur et basée sur des méta-heuristiques d'exploration.

Ensuite, nous proposons une vue plus détaillée basée sur notre modèle conceptuel des processus d'extraction (*cf.* Section 2.2.3). Nous décrivons ainsi les niveaux du code source, du modèle du code source, de la mise en correspondance et de l'architecture. Nous précisons également les liens et le parcours du processus à travers ces niveaux.

Enfin, nous proposons un ensemble de guides utilisés pour diriger notre processus. Nous présentons donc les différents éléments qui constituent ces guides et la manière dont ils peuvent être utilisés pour diriger et améliorer l'extraction.

3.1 Motivation et problème

La modélisation et la représentation de l'architecture logicielle des systèmes complexes sont devenues une phase importante dans leur processus de développement [16]. L'architecture d'un système décrit sa structure à un haut niveau d'abstraction et offre de ce fait de nombreux avantages tout au long du cycle de vie du logiciel [114]. Nous avons présenté et expliqué ces avantages dans la partie I. Nous avons, en particulier, présenté les avantages de l'architecture pendant la phase de maintenance.

Cependant, nous avons également montré que beaucoup de systèmes existants ne disposent pas d'une représentation fidèle de leur architecture. En effet, le système peut avoir été conçu sans représentation de son architecture, comme dans le cas de certains systèmes patrimoniaux. Pour d'autres systèmes, la représentation disponible est décalée par rapport à l'implémentation du système, à cause des écarts entre l'architecture prévue et implémentée puis du manque de synchronisation entre la documentation et l'implémentation.

Ce manque de représentation peut annuler tous les avantages de l'architecture durant la maintenance et dégrader encore plus les chances de succès des opérations de maintenance. Pour pallier ce problème, de nombreuses approches, que nous avons présentées dans la partie I, visent à extraire l'architecture logicielle d'un système existant. Cependant, ces approches présentent des lacunes qui motivent notre travail et définissent un cadre pour notre approche.

Ainsi, nous présentons donc les principaux manques parmi les travaux existants puis nous motivons notre choix d'une approche par exploration en présentant les avantages de l'ingénierie logicielle basée sur l'exploration (*Search-based software engineering*, SBSE).

3.1.1 Extraction d'architectures

Face à la nécessité d'une vue architecturale pertinente pour faciliter la maintenance d'un système, les approches d'extraction existantes proposent de multiples solutions. Cependant, ces approches ne répondent pas parfaitement à ce problème. Ces défauts peuvent être classés selon le cadre de comparaison défini dans le chapitre 2. Nous présentons donc les limites et défauts des approches existantes selon les trois axes de notre cadre : les propriétés conceptuelles, les informations utilisées et les propriétés techniques.

3.1.1.1 Propriétés conceptuelles

Les approches existantes présentent plusieurs limitations en termes de propriétés conceptuelles et plus particulièrement sur la définition de l'architecture extraite. La première de ces limites est le manque de précision dans la définition de l'architecture extraite. En effet, peu d'approches posent une définition claire de l'architecture qu'elles visent à extraire. Ainsi, dans la plupart des approches, les éléments architecturaux considérés ne sont pas précisés de manière complète et la définition de ces éléments reste floue.

Ces imprécisions dans les définitions des éléments architecturaux mettent également en évidence les manques dans la définition de l'architecture utilisée. En effet, aucune approche ne prend en considération tous les éléments architecturaux. Dans tous les cas, les connecteurs sont traités comme des entités de second plan. Les connecteurs sont, au mieux, considérés comme une représentation des appels de méthodes ou autres dépendances entre les composants et au pire ils sont ignorés et considérés comme implicites.

La définition des composants extraits représente également une des limites des approches existantes. En effet, les processus d'extraction n'identifient pas, dans la plupart des cas, clairement les interfaces des

composants et elles demeurent donc implicites. Une autre limitation porte sur l'aspect hiérarchique des composants. En effet, beaucoup d'approches ne permettent pas d'identifier des relations de compositions hiérarchiques entre les composants. Cet aspect n'est pas pris en compte dans la majorité des approches et seules quelques unes considèrent ces relations de compositions et permettent d'extraire des composants hiérarchiques [74]. De la même façon, la plupart des approches, et en particulier les approches manuelles, ne traitent pas le caractère partageable des composants. Ce caractère partageable est laissé à l'appréciation des utilisateurs ce qui rend ces approches floues sur cet aspect.

Enfin, la majorité des approches d'extraction d'architectures portent sur les paradigmes de programmation les plus anciens. Ainsi, si les architectures basées sur les modules ou les objets sont couramment traitées [5, 69, 111], celles basées sur les composants logiciels sont plus rares [88].

3.1.1.2 Informations utilisées

L'extraction vise à extraire l'architecture réelle du système en se basant sur l'architecture fournie par le code source, mais également en prenant en compte l'architecture intentionnelle du système, c'est-à-dire l'architecture imaginée par les concepteurs. Cette architecture intentionnelle n'est pas contenue dans le code source mais elle est répartie dans l'ensemble de la documentation et, bien sûr, dans les connaissances de l'architecte.

Les approches d'extraction utilisent, pour la plupart, l'expertise des utilisateurs pour accéder à cette architecture intentionnelle, la rendant absolument nécessaire [72]. Les autres n'utilisent absolument pas les informations disponibles à travers les experts [4]. Malheureusement, ces deux cas présentent des défauts. Le premier oblige à utiliser un expert du domaine et du système pour réaliser l'extraction. Cette expertise est souvent coûteuse, en argent et en temps. Le deuxième ne permet pas à l'utilisateur d'incorporer des informations intentionnelles dans le processus d'extraction. Le résultat de l'extraction se limite alors à l'architecture du système représenté par le code source au lieu de l'architecture du système dans son ensemble.

Parmi les travaux utilisant les recommandations des experts, la principale limite est le manque d'expressivité de ces recommandations. L'expert est cantonné soit à proposer des informations, soit à valider les informations extraites. De la même façon, l'expert ne peut, dans la majorité des approches, fournir uniquement des informations positives, telle que le fait que deux entités du code source appartiennent à un même composant. Ces approches se privent des informations négatives que peut fournir l'expert, telle que le fait que deux entités du code source n'appartiennent pas à un même composant.

L'expertise utilisée pour l'extraction n'est pas la seule source d'information pour obtenir l'architecture intentionnelle du système. La documentation contient également une grande part des informations sur cette vue architecturale. Malgré cela, aucune approche n'utilise directement cette documentation. Dans le meilleur cas, la documentation est utilisée à travers l'utilisateur ainsi que sa compréhension et/ou interprétation des documents.

3.1.1.3 Propriétés techniques

Les approches d'extraction existantes sont, en majorité, manuelles ou quasi manuelles. A ce titre, elles nécessitent l'intervention d'experts pour réaliser l'extraction. En effet, ces experts doivent avoir une grande connaissance du système étudié afin de pouvoir faire les choix sur lesquels reposent les approches manuelles. Mais, ces experts doivent également posséder une grande maitrise de l'outil d'extraction afin de comprendre les conséquences de leurs choix et de pouvoir décider de la façon d'appliquer la méthodologie proposée pour réaliser l'extraction. Cette expertise est coûteuse, en temps et en moyen

financier, et soumet le processus à d'importants risques d'erreurs humaines.

D'autres approches sont totalement automatiques. Elles ne permettent pas à l'utilisateur d'influer sur le processus en lui fournissant une quelconque information. Par contre, elles peuvent fournir certains paramètres permettant d'adapter légèrement le processus. Ces approches fournissent donc des processus d'extraction sûrs et rapides mais qui ne tirent pas partie de toutes les informations disponibles et en particulier de l'architecture intentionnelle.

Enfin, les approches existantes ne proposent pas de mécanismes d'itération. Elles procèdent toutes de manière séquentielle. Si le résultat n'est pas satisfaisant, le processus complet doit être réinitialisé avec des paramètres modifiés. Un processus itératif, permettant de prendre en entrée le résultat d'une précédente exécution, permettrait d'améliorer les résultats en procédant par affinage au lieu de devoir recommencer le processus à chaque fois.

3.1.2 Approches par exploration

Notre choix d'une approche méta-heuristique est motivé par les travaux récents dans le domaine du *Search-based software engineering* (SBSE) qui ont montré l'efficacité de ces techniques pour résoudre des problèmes de ce type [58]. Ainsi, l'un des avantages majeurs de ce type d'approche est son degré élevé d'automatisation qui diminue le besoin en expertise humaine mais permet une forte interaction avec l'utilisateur.

Le SBSE peut être défini comme l'application de méta-heuristiques pour résoudre les problèmes d'optimisation rencontrés dans l'ingénierie logiciel [58]. Ces techniques ont été appliquées à diverses activités de l'ingénierie logicielle, tout au long du cycle de vie, depuis la spécification des besoins [8], jusqu'à la maintenance [98].

En particulier, les travaux de MANCORIDIS [86] sont représentatifs du SBSE. Ils modélisent le problème de modularisation du logiciel comme un problème de regroupement où les méta-heuristiques peuvent être appliquées. Leur outil Bunch propose ensuite plusieurs méta-heuristiques et retourne un graphe de dépendances entre les modules. Ce résultat peut alors vue être comme une architecture où les composants sont identifiés aux modules.

3.2 Présentation de notre solution : ROMANTIC

Partant des constats faits sur les limites des approches existantes d'extraction d'architectures, nous proposons une approche, appelée ROMANTIC*, visant à extraire une architecture logicielle à base de composants depuis un système orienté objet existant. Notre processus permet de sélectionner parmi l'ensemble des architectures pouvant être abstraites du système, la meilleure possible par rapport à un ensemble de propriétés de qualité et en respectant un ensemble de guides tel que la documentation du système. Ainsi, nous proposons de formuler ces propriétés comme des contraintes mesurables et nous utilisons une approche d'optimisation méta-heuristique afin d'extraire l'architecture qui satisfait le maximum de ces contraintes.

L'objectif de notre approche est de pallier les limites identifiées dans la section précédente. Pour cela, nous proposons une approche d'extraction possédant les propriétés suivantes :

- **interactive** : c'est-à-dire permettant à l'utilisateur d'intervenir tout au long du processus en fournissant toutes les informations qu'il possède. Les interventions ne sont cependant pas nécessaires

*ROMANTIC : Re-engineering of Object-oriented sytesMs by Architecture extractioN and migraTIon to Component based ones.

au bon fonctionnement du processus ;
- **itérative** : c'est-à-dire que notre processus peut être utilisé de manière itérative en utilisant les sorties d'une exécution comme entrée d'une nouvelle ;
- **semi-automatique** : c'est-à-dire utilisant des méta-heuristiques pour explorer l'espace des solutions du problème d'extraction.

Dans la suite, nous décrivons les propriétés de notre approche en fonction du cadre de comparaison que nous avons utilisé pour comparer les approches d'extraction existantes.

3.2.1 Propriétés conceptuelles dans ROMANTIC

ROMANTIC vise à extraire une architecture à base de composants à partir d'un système orienté objet. Cette approche peut être décrite selon notre modèle conceptuel de l'extraction d'architectures (*cf.* Figure 3.1).

3.2.1.1 Niveau du code source et du modèle du code source

Depuis son émergence, le paradigme objet s'est répandu dans les entreprises. Au cours de son utilisation, on a vu apparaitre un grand nombre de systèmes objets nécessitant des phases de maintenance complexes. De plus, face à la diversité des langages à base d'objets, il est nécessaire de proposer une approche suffisamment générique pour être appliquée à tous les systèmes objets.

Face à ce problème, ROMANTIC propose d'extraire l'architecture logicielle d'un système orienté objet. Nous avons choisi un modèle du code source générique contenant les principales entités du monde objet. De plus, ce modèle est suffisamment générique pour pouvoir être instancié à travers une analyse statique ou dynamique du code source. Les entités du modèle du code source sont donc principalement les classes, les méthodes, ainsi que les relations entre les classes ou entre les méthodes telles que la relation de composition entre classes ou l'appel de méthode.

3.2.1.2 Niveau de l'architecture

ROMANTIC propose d'extraire une architecture logicielle à base de composants conforme aux définitions. Cette architecture doit donc représenter de manière explicite les trois éléments architecturaux : les composants, les connecteurs et la configuration. De même, ces éléments doivent expliciter chaque élément de leurs définitions, tels que les interfaces pour les composants.

Concernant les éléments architecturaux, nous avons fait plusieurs choix quant aux propriétés que nous considérons. Ainsi, nous avons choisi de considérer les composants composites, mais, dans un souci de simplification, nous n'autorisons pas le partage de composants. Au lieu d'avoir un sous-composant partagé entre deux composites, le résultat de notre extraction sera deux composites reliés à un même composant représentant le sous-composant. Concernant les connecteurs, nous avons choisi de ne pas considérer les connecteurs composites.

Pour obtenir cette architecture complète, notre approche décompose l'extraction d'architectures en un ensemble de processus étroitement liés (*cf.* Figure 3.2). L'extraction d'une architecture complète requiert l'identification de chaque élément architectural, c'est-à-dire les composants, les connecteurs et la configuration. Enfin pour que l'architecture soit complète, il faut encore ajouter plusieurs sous-processus à l'identification des composants. Le premier est l'identification des interfaces des composants, c'est-à-dire établir formellement les points de contact entre le composant et le reste du système. Le second sous-processus vise à identifier les relations de compositions hiérarchiques entre les composants et à extraire les composants hiérarchiques.

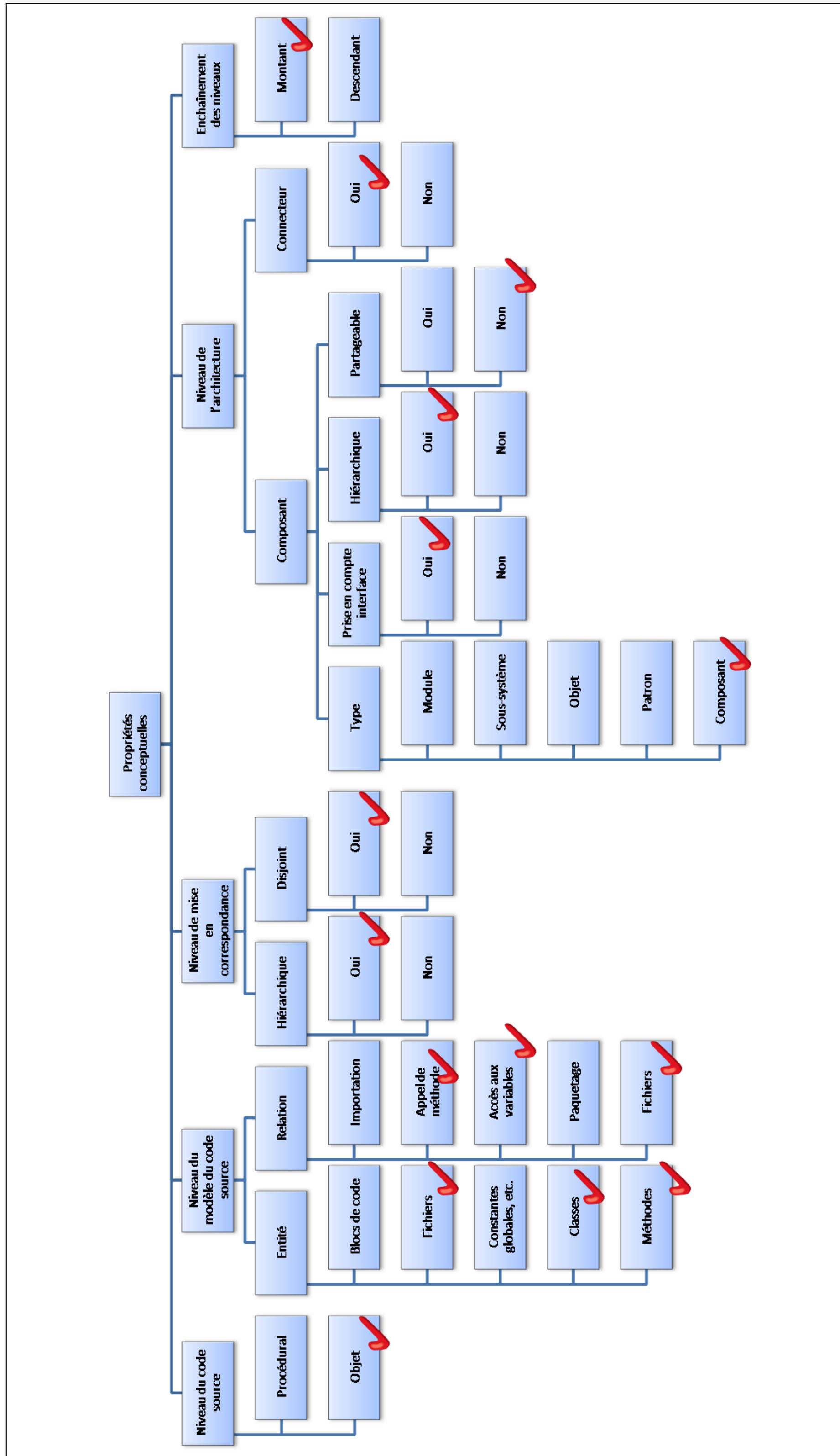


Figure 3.1 – Caractéristiques de ROMANTIC suivant l'axe des propriétés conceptuelles

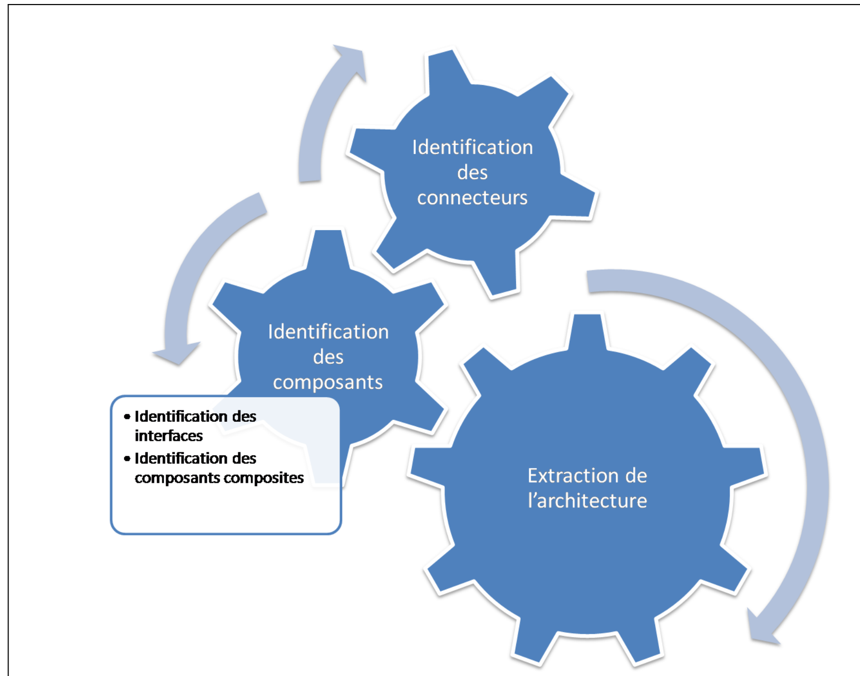


Figure 3.2 – Décomposition du processus d'extraction d'architectures

3.2.1.3 Niveau de correspondance

L'extraction d'architectures vise à fournir une vue architecturale d'un système. Cependant, comme nous l'avons vu dans le chapitre 2, cette vue architecturale n'est pas le seul objectif. Pour que cette vue soit utilisable, le processus d'extraction doit également fournir les correspondances entre les éléments architecturaux et ceux du modèle du code source. Dans notre approche, ces correspondances sont matérialisées par les entités du modèle de correspondance.

Chaque entité du modèle de correspondance associe un type d'élément architectural et un regroupement d'entité du modèle du code source. Par exemple, le composant est associé à l'entité « contour de composant » qui est constituée d'un ensemble de classe (*cf.* Figure 3.3), et le connecteur est associé à l'entité « contour de connecteur » qui est constituée d'un ensemble de méthodes et d'attributs.

Chaque instance de ce modèle de correspondance permet d'obtenir une architecture possible du système en associant à chaque entité du modèle de correspondance, l'élément architectural correspondant. L'espace constitué de l'ensemble de ces instances constitue l'espace des solutions de notre processus, c'est-à-dire l'ensemble des éléments que nous devons explorer pour résoudre le problème d'extraction.

3.2.2 Informations utilisées dans ROMANTIC

L'objectif de ROMANTIC est d'extraire l'architecture réelle d'un système. Pour cela, nous nous appuyons sur le code source mais aussi sur l'architecture intentionnelle du système (*cf.* Figure 3.4). Cette architecture est celle souhaitée par les architectes ayant conçu et maintenu le système. Elle peut donc fournir de précieuses informations sur l'architecture réelle de ce dernier.

Pour utiliser l'architecture intentionnelle durant l'extraction de l'architecture réelle d'un système, ROMANTIC utilise les recommandations des architectes. Pour cela, le processus d'extraction est interactif et permet donc à l'utilisateur de fournir des informations à différents moments. Pour permettre une

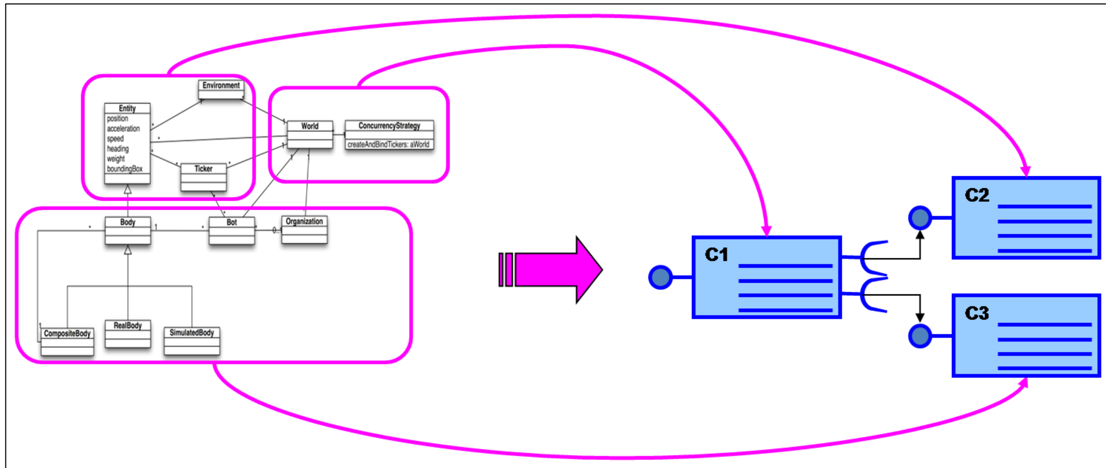


Figure 3.3 – Correspondance entre les entités du code source et architecturales

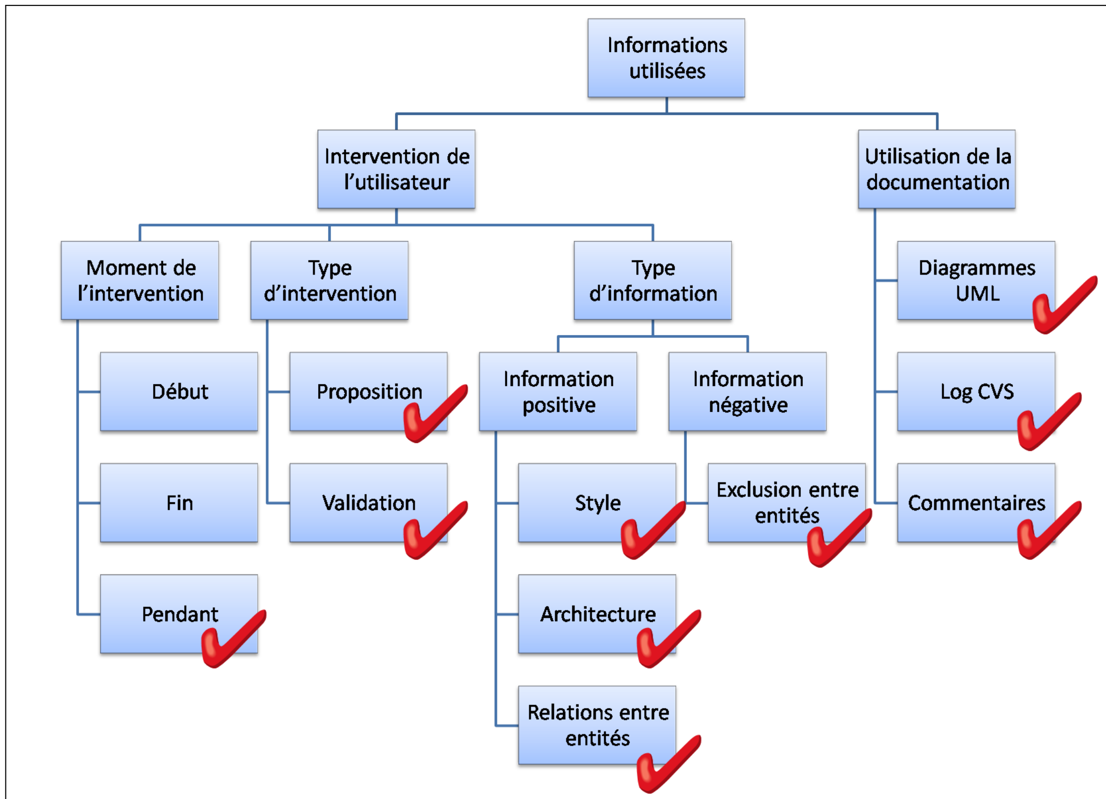


Figure 3.4 – Caractéristiques de ROMANTIC suivant l'axe des informations utilisées

expressivité maximale à l'utilisateur, notre processus accepte des interactions sous forme de validation et de proposition. Il accepte également à la fois des informations positives et négatives.

Nous utilisons également l'ensemble des documents concernant le système. Pour cela, nous proposons un modèle des informations contenues dans les documents. Les instances de ce modèle sont obtenues à partir de différents types de documents et sont utilisées pour diriger l'extraction.

En plus de l'architecture intentionnelle, nous utilisons l'architecture matérielle sur laquelle doit être déployé le logiciel. Pour cela, le processus d'extraction que nous proposons, utilise les informations contextuelles pour déterminer l'architecture la plus adéquate dans ce contexte.

3.2.3 Propriétés techniques de ROMANTIC

Pour extraire l'architecture, nous proposons un processus semi-automatique d'exploration (*cf.* Figure 3.5). Notre processus doit sélectionner parmi l'ensemble des architectures pouvant être abstraites du système, la plus adéquate en utilisant les informations telle que la documentation et les recommandations des utilisateurs. ROMANTIC explore donc l'espace des architectures possibles et sélectionne la meilleure solution. Pour réaliser cette sélection, notre processus s'appuie sur une fonction objectif mesurant la qualité de chaque architecture. Cette qualité repose, à la fois, sur des critères de qualité fonctionnelle et sur l'adéquation entre la solution et la définition du concept d'architecture.

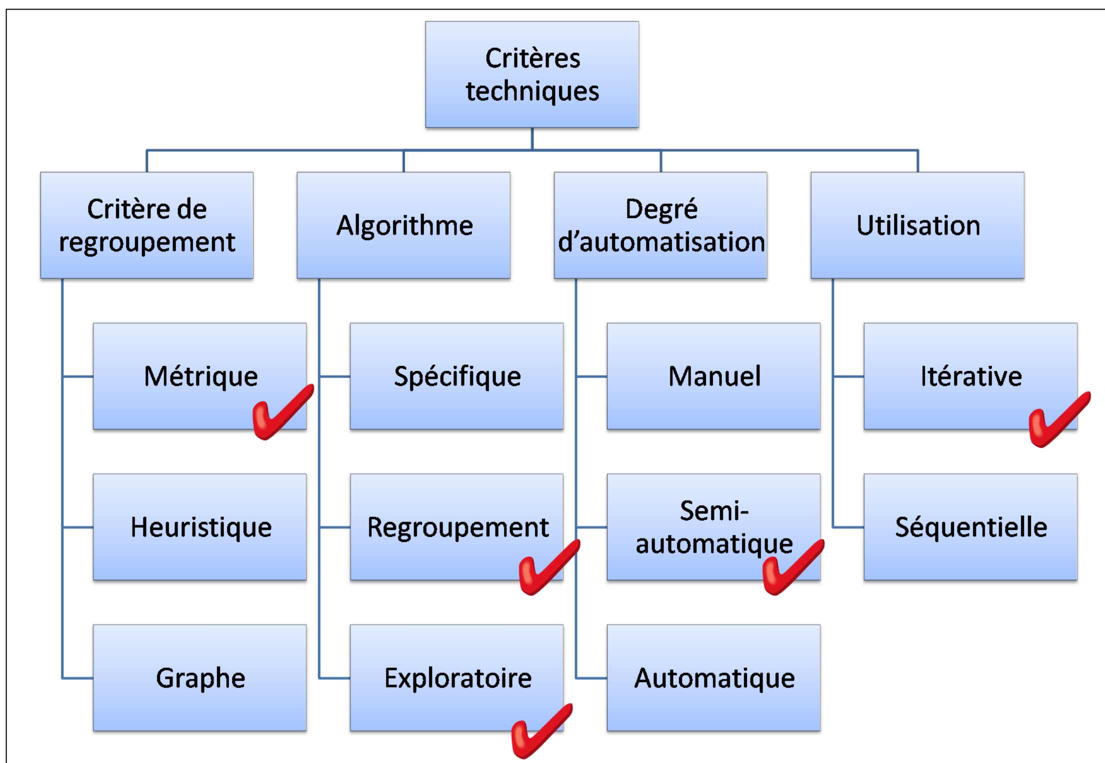


Figure 3.5 – Caractéristiques de ROMANTIC suivant l'axe des propriétés techniques

ROMANTIC est également un processus itératif utilisant les sorties d'une exécution comme entrée d'une nouvelle exécution. L'objectif de ces itérations est de permettre un affinage progressif des résultats et de compenser, si besoin, le manque d'expertise des utilisateurs.

3.3 Modélisation du problème

Afin de modéliser le problème d'extraction comme un problème d'exploration, nous définissons d'abord plusieurs modèles. Le premier des modèles à définir est celui du code source. Comme nous l'avons vu, ce modèle décrit les entités du code source que nous utilisons pendant l'extraction, ainsi que les relations entre ces entités. Dans le cadre de notre approche visant à extraire l'architecture d'un système orienté objet, le modèle que nous proposons repose sur des modèles objets que nous présentons également. Le deuxième modèle, essentiel à une approche d'extraction, est le modèle de correspondance entre les entités du modèle du code source et les éléments architecturaux.

A partir de ces deux modèles nous proposons un modèle de notre approche d'extraction en exposant les trois entités constituant une approche par exploration. Ces trois entités sont l'espace des solutions, qui dérive directement du modèle de correspondance, l'espace de recherche, qui est un sous-espace du précédent, décrivant les solutions pouvant être explorées et enfin la méthode d'exploration qui décrit comment se déroule l'exploration de l'espace de recherche.

3.3.1 Notre modèle du code source

Notre processus d'extraction de l'architecture d'un système repose sur un modèle qui permet de représenter le code source tout en faisant abstraction de certaines entités que nous ne souhaitons pas prendre en compte. En effet, ce modèle du code source contient uniquement les entités et les relations du code source qui ont une influence au niveau architectural.

Le choix de ce modèle impose une certaine granularité au résultat de notre processus. L'objectif principal, lors de la définition de ce modèle, est donc de choisir des entités objets d'un niveau suffisant pour obtenir une granularité fine de l'architecture mais en gardant le niveau d'abstraction attendu pour une vue architecturale.

Une autre contrainte importante, sur ce modèle du code source, est le besoin de généricité. Cette contrainte de généricité porte surtout sur les deux aspects que sont l'indépendance par rapport à un langage et l'indépendance par rapport à la méthode d'instanciation. En effet, notre approche d'extraction vise surtout à extraire l'architecture logicielle de systèmes patrimoniaux orientés objet. Or, les langages utilisés pour implémenter ces systèmes sont très divers. La proposition d'un modèle indépendant du langage nous permet d'appliquer facilement notre approche à ces langages. Il suffit de proposer un outil permettant d'instancier notre modèle du code source à partir de chaque langage (*cf.* Figure 3.6).

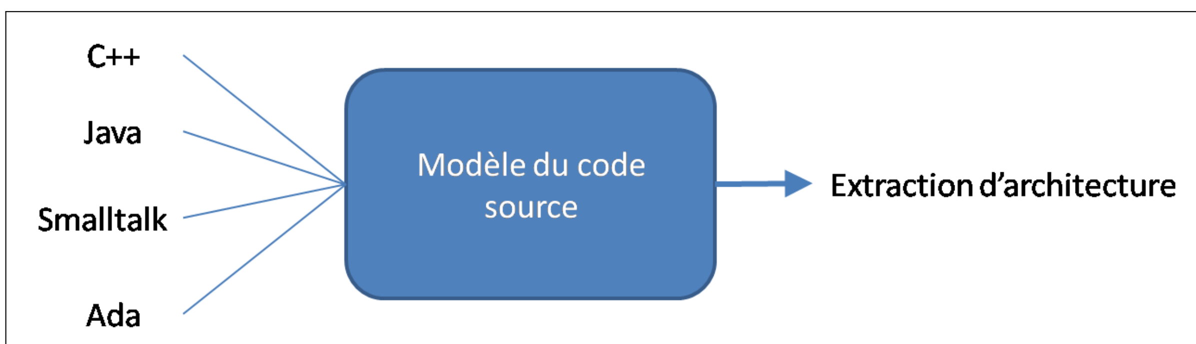


Figure 3.6 – Un modèle du code source indépendant du langage

Le modèle doit également être indépendant de la méthode d'instanciation. En effet, l'instanciation du modèle à partir du code source peut se faire de manière statique ou dynamique. *Dans l'approche*

statique, les entités et les relations entre elles sont identifiées en parcourant les fichiers du code source. Les fichiers utilisés peuvent être des fichiers compilés ou non et ne contiennent pas tous les mêmes informations. Ainsi, un fichier résultant d'une compilation peut fournir des informations sur les adresses en mémoire des différents paramètres par exemple. *L'approche dynamique* repose sur l'étude du comportement du système à l'exécution. Cette approche permet de détecter les entités effectivement utilisées à l'exécution mais elle repose entièrement sur les scénarios utilisés pour tester le système. Chacune de ces approches présentant des avantages et des inconvénients, nous souhaitons définir un modèle permettant à l'utilisateur de choisir la méthode d'instanciation du modèle.

Pour développer notre modèle du code source en respectant ces contraintes, nous proposons de spécialiser le modèle objet FAMIX [38] issu du projet FAMOOS [36]. Nous présentons donc FAMIX puis notre spécialisation de ce modèle.

3.3.1.1 Le modèle FAMIX

FAMOOS [36] est l'acronyme de « *Framework-based Approach for Mastering Object-Oriented Software Evolution* ». L'objectif de ce projet est de fournir un support pour l'évolution des logiciels orientés objet de première génération. Le projet a fourni différents outils et méthodes permettant d'analyser et de détecter les problèmes de conception et de transformer ces systèmes pour utiliser des architectures plus flexibles. Pour permettre à ces outils de communiquer et de coopérer, FAMOOS a conduit au développement d'un modèle fournissant une représentation du code source objet indépendante du langage de programmation. Ce modèle, appelé FAMIX [38], constitue donc la base des échanges d'informations sur les systèmes orientés objet entre les outils proposés dans le cadre du projet FAMOOS.

Les avantages du modèle FAMIX sont nombreux et permettent de compenser certains défauts de UML [38, 37] par exemple. Nous citons entre autres :

- **extensible** : FAMIX est facilement extensible pour incorporer les entités et propriétés spécifiques à chaque langage. Ainsi, plusieurs extensions ont été proposées pour Java [122], Ada [96] ou C++ [24] ;
- **support complet pour les métriques, heuristiques et autres méthodes de réingénierie** : FAMIX contient tout les éléments nécessaires pour l'ensemble des mesures et méthodes envisagées dans le cadre de FAMOOS ;
- **facilité d'instanciation** : FAMIX peut être instancié à partir d'une analyse lexicale simple du code source. Cette analyse ne nécessite pas d'interprétation. Ainsi, la détection des relations de composition ou d'agrégation n'est pas nécessaire pour obtenir une instance de FAMIX ;
- **combinaison possible de plusieurs sources** : FAMIX repose essentiellement sur les informations extraites depuis le code source des systèmes. Cependant, il permet également de prendre en compte d'autres sources d'informations tels que les diagrammes de conception. Pour cela, il offre des facilités pour regrouper des modèles issus de différentes sources ;
- **instanciation dynamique ou statique** : FAMIX offre la possibilité d'être instancié à partir d'une analyse statique ou dynamique d'un système. Cet avantage n'est pas un des objectifs fixés par le projet, mais il découle de la simplicité d'instanciation et de la généricité du modèle FAMIX.

Le modèle complet de FAMIX, dont la hiérarchie est représentée sur la figure 3.7, comporte les principales entités du paradigme objet : les paquetages, les classes, les méthodes et les attributs. Il comporte également les principales relations disponibles dans le monde des objets : invocations, accès et héritages. Les autres éléments objets sont également présents tels que les fonctions, les variables locales, les paramètres ou les arguments.

Cependant, la généricité de FAMIX est adaptée au projet FAMOOS qui nécessite un modèle pour

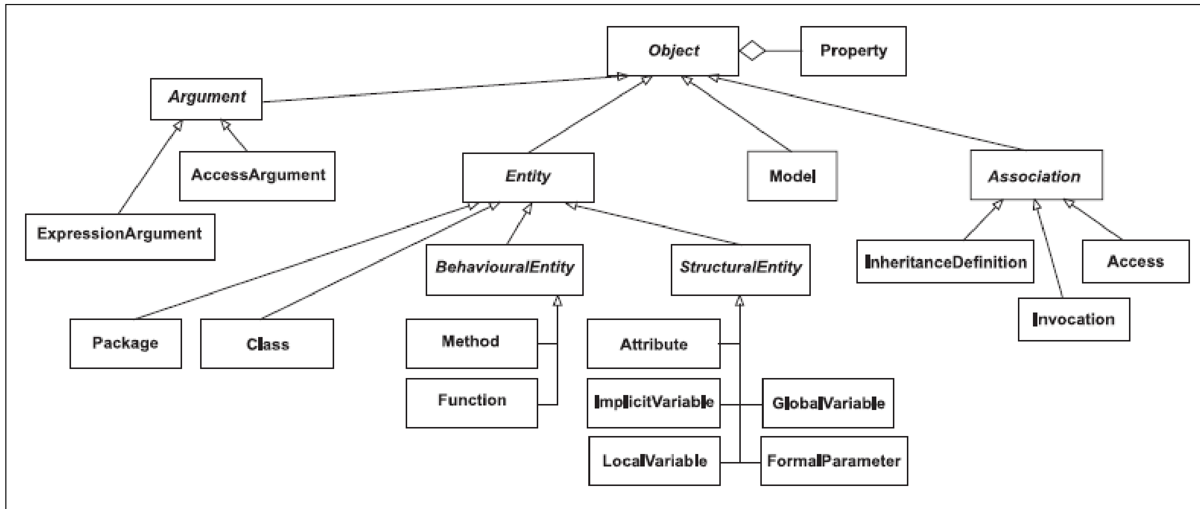


Figure 3.7 – Le modèle FAMIX 2.0

échanger des informations entre des outils issus de diverses équipes et conçus avec un minimum de concertation. Dans notre cas, cette généricité est trop forte et certaines entités du modèle ne sont pas nécessaires à notre utilisation. Ainsi, les entités décrivant les variables implicites ou globales ne sont pas nécessaires dans le cadre de notre approche.

La simplicité de FAMIX pose également le problème inverse. En effet, certaines relations qui ne peuvent pas être détectées par une analyse lexicale simple ne sont pas incluses dans FAMIX, mais sont cruciales pour notre approche. Les relations de composition ou d'agrégation ne sont pas détectables directement dans le code source. Pourtant, elles constituent une dépendance forte entre classes et peuvent ainsi fortement influencer le choix de considérer deux classes comme appartenant à un même composant.

3.3.1.2 Spécialisation de FAMIX pour l'extraction d'architectures

Pour résoudre les problèmes de FAMIX dans le cadre de notre approche, nous proposons un nouveau modèle du code source, basé sur FAMIX. Dans ce modèle (*cf.* Figure 3.8), nous supprimons les entités inutilisées dans notre approche, telles que les variables globales. Nous avons également ajouté les relations de composition et d'agrégation entre classes qui révèlent des dépendances fortes et sont donc très utiles dans le cadre de l'extraction d'architectures et, en particulier, des composants composites.

Cette extension nous permet d'avoir un modèle du code source adapté à notre problème d'extraction et bénéficiant des avantages de FAMIX. En particulier, nous nous intéressons aux avantages, évoqués précédemment, qui sont l'indépendance par rapport aux langages, l'indépendance par rapport à l'aspect dynamique ou statique de l'analyse et surtout la facilité à combiner plusieurs sources d'information. Ce dernier aspect est crucial pour tirer tout les avantages des guides que nous avons pu identifier pour orienter le processus d'extraction.

Les éléments abstraits. Les éléments *objet, entité, et relation* permettent au modèle d'être extensible. Ils sont directement issus de FAMIX sauf *relation* qui est un simple renommage de l'entité *association* de FAMIX. *Objet* possède deux attributs qui stockent respectivement la source à partir de laquelle l'objet est extrait et la liste de commentaires associés à cet objet dans la source. Les *entités* possèdent deux attributs qui sont un nom et un identifiant unique. Le premier fournit une version lisible de l'identifiant

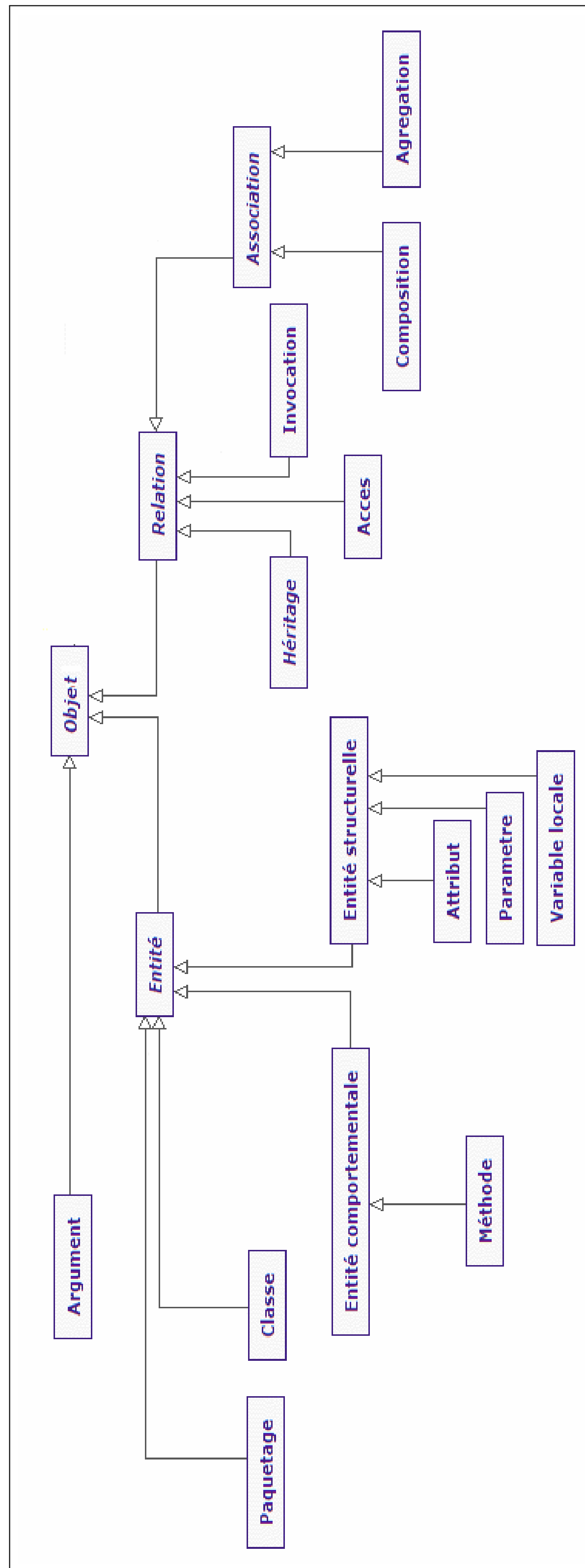


Figure 3.8 – Notre modèle du code source

de l'entité qui sert de référence dans le modèle.

Les arguments. Un *argument* représente le passage d'un argument lors de l'invocation d'une méthode. Les deux attributs d'un *argument* déterminent la position de l'argument dans la liste des arguments et la qualité de receveur ou non de l'argument, c'est-à-dire si l'argument désigne l'entité qui reçoit le message.

Les entités. Une *entité* peut être de quatre types. Elle peut d'abord être un *paquetage*. Elle représente alors une sous-unité nommée du code source telle qu'un paquetage en Java ou un espace de nom en C++.

Une *entité* peut également être une *classe*, c'est-à-dire qu'elle représente une classe du code source. Cette *classe* possède deux attributs qui déterminent si la classe est abstraite et à quel paquetage elle appartient.

L'*entité* peut également être une *méthode* qui représente la définition dans le code source d'un aspect du comportement de la classe à travers une méthode. Cette entité hérite des attributs des *entités comportementales* qui décrivent la signature, la visibilité et le type de retour de l'entité du code source représentée. Elles possèdent également des attributs décrivant les différentes propriétés des méthodes, c'est-à-dire la classe à laquelle elle appartient, si elle est abstraite, si c'est un constructeur et si c'est une méthode de classe ou d'instance.

Enfin, l'*entité* peut être une *entité structurelle*. Elle représente alors un aspect de l'état du système. Elle possède un attribut décrivant son type et sa classe. Une *entité structurelle* est un type abstrait qui possède trois sous-types. Le premier est l'*attribut* qui représente un attribut du code source. Les attributs de cette entité décrivent la portée, la visibilité et la classe à laquelle appartient l'attribut représenté par l'entité. Les autres sous-types sont la *variable locale* et le *paramètre* qui représentent respectivement les variables définies dans une méthode du code source et les paramètres d'une méthode du code source. Dans les deux cas, un attribut précise à quelle méthode cette entité appartient. Le *paramètre* décrit en plus sa position dans la liste des paramètres de la méthode.

Les relations. Une *relation* représente une relation entre éléments du code source. Cette entité correspond à l'*association* dans FAMIX et a été renommée pour éviter les ambiguités. Une relation peut être de quatre types. D'abord, l'*héritage* représente les relations d'héritages entre classes dans le code source. Cette relation est décrite en précisant la super-classe, la sous-classe, la visibilité de l'une par rapport à l'autre et l'index de la relation qui est utilisé dans les langages où l'héritage multiple est autorisé.

L'*invocation* représente l'invocation dans le code source d'une méthode par une autre. Les attributs de cette entité décrivent la méthode qui invoque, ainsi que différentes informations permettant de résoudre, en partie, le polymorphisme qui peut empêcher de déterminer quelle méthode est invoquée. L'attribut le plus utile est donc la liste des méthodes qui correspondent à la signature de la méthode invoquée. Cet attribut, couplé avec les arguments utilisés, permet de résoudre le polymorphisme.

L'*accès* représente l'accès dans le code source d'une méthode à une entité structurelle. Les attributs de cette relation précisent la méthode qui accède, la variable et si l'accès vise à modifier ou à lire la variable.

L'*association* ne correspond pas à l'entité du même nom dans FAMIX. Cet ajout par rapport à FAMIX représente des relations d'association d'un niveau sémantique supérieur aux précédentes. Ces relations ne sont pas prises en compte dans FAMIX car leur détection nécessite une analyse sémantique plutôt que syntaxique. Ces associations peuvent être des *compositions* ou des *agrégations* et représentent des relations de composition et d'agrégation entre classes du code source. Les attributs de ces relations décrivent les deux entités impliquées dans la relation.

L'ensemble des entités du modèle à l'exception des relations d'association sont décrites dans le modèle FAMIX. Une description précise de ces éléments se trouve dans [38].

3.3.2 Notre modèle de mise en correspondance

Comme nous l'avons vu précédemment, l'extraction de l'architecture d'un système orienté objet repose d'abord sur une mise en correspondance des entités objets avec les éléments architecturaux : les composants qui décrivent la partie métier, les connecteurs qui décrivent les interactions et la configuration qui représente la topologie des connexions entre les composants. Pour cela, nous proposons un modèle de correspondance (modèle COA, *cf.* Figure 3.9) dont les entités permettent d'établir les liens entre les entités du modèle du code source présenté précédemment et celles du modèle de l'architecture que nous avons présenté dans le chapitre 1 : les composants, les connecteurs et la configuration.

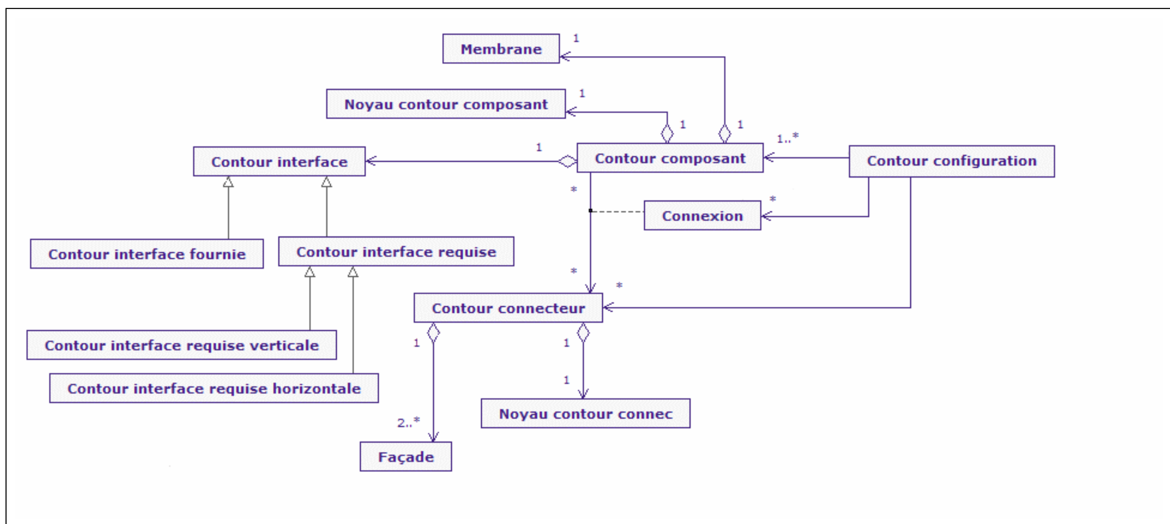


Figure 3.9 – Notre modèle de correspondance objet-composant (COA)

Il existe une entité, dans le modèle de correspondance, pour chaque élément architectural. Ces entités, appelées « contours », représentent les composants, les connecteurs ou la configuration (*cf.* Figure 3.10). Les contours sont également liés aux éléments du modèle du code source (*cf.* Figure 3.11).

Contours des composants. Les composants logiciels encapsulent la partie métier de l'architecture. Ils proposent un ensemble de fonctionnalités à travers un ensemble d'interfaces. A ce titre, ils sont proches de la notion de classes qui proposent un ensemble de fonctionnalités à travers un ensemble de méthodes. Par conséquent, le contour d'un composant est un ensemble de classes pouvant appartenir à différents paquetages. Chaque contour est constitué de plusieurs éléments (*cf.* Figure 3.12).

Le premier élément est « la membrane du contour » qui contient l'ensemble des classes qui ont un lien avec des classes situées à l'extérieur du contour, par exemple un appel de méthode vers l'extérieur. Cette membrane représente l'ensemble des points de contact possibles entre le composant et l'extérieur. Le deuxième élément est le « noyau » qui contient le reste des classes du contour, c'est-à-dire celles qui n'ont pas liens de dépendance avec une classe à l'extérieur du contour. Par définition, les classes sont incluses dans le noyau ou dans la membrane.

Le dernier élément d'un contour est un ensemble de contours d'interfaces qui représentent les interfaces fournies et requises du composant. Chacun de ces contours est un ensemble de noms de méthodes.

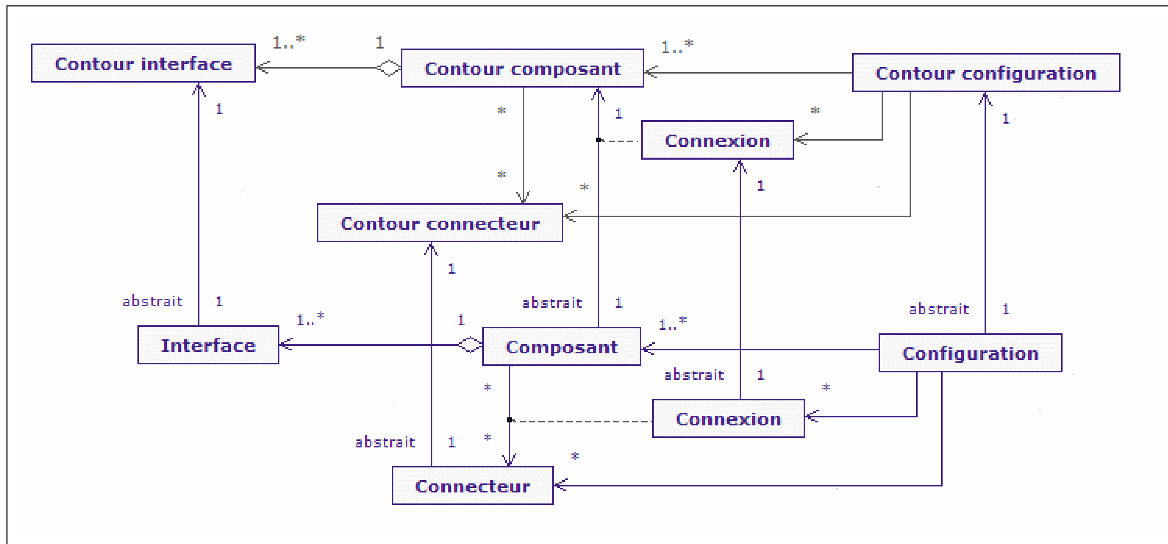


Figure 3.10 – Relations entre les entités COA et architecturales

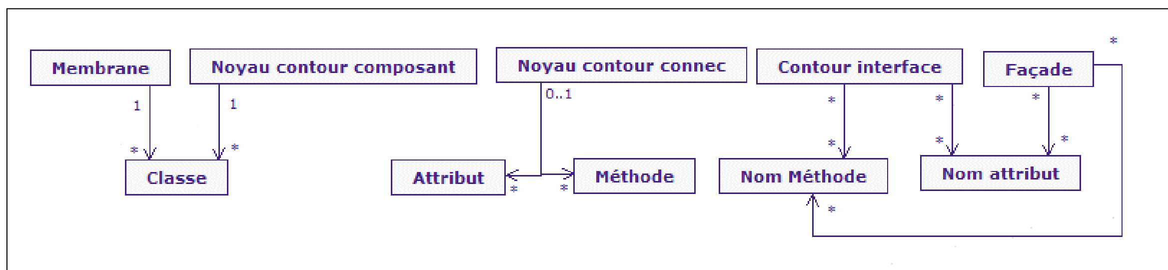


Figure 3.11 – Relations entre les entités COA et du modèle du code source

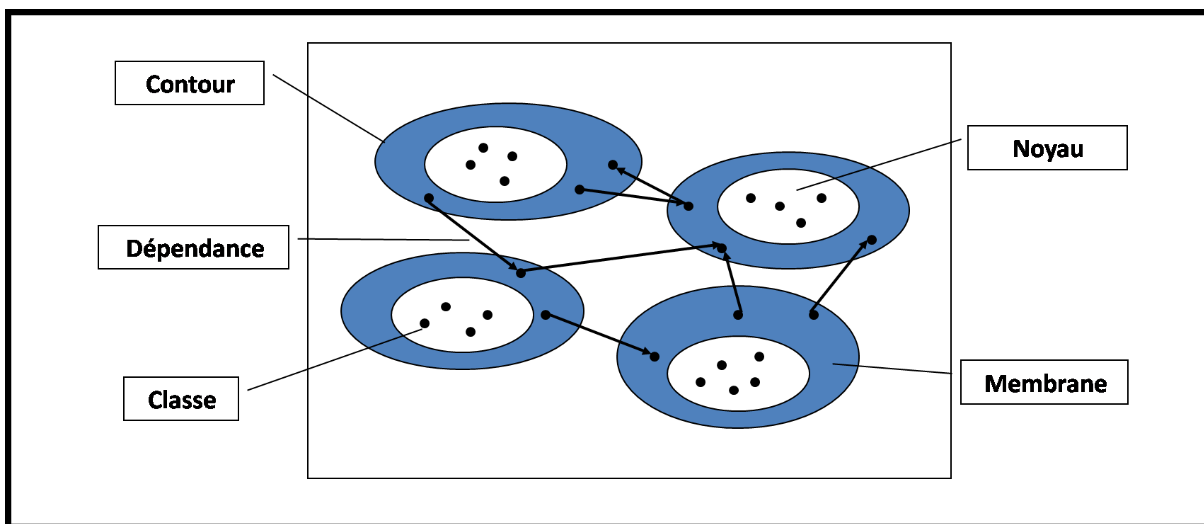


Figure 3.12 – Éléments des contours des composants

Contours des interfaces. Le contour d'une interface représente l'interface d'un composant. On distingue les contours des interfaces requises et des interfaces fournies. Cependant, dans les deux cas, le contour est un ensemble de noms de méthodes issues de différentes classes.

Interface fournie. L'interface fournie d'un composant est un point d'accès vers une fonctionnalité d'un composant. Elle se décompose en un ensemble de services qui décrivent un point d'entrée dans le composant. Chacun de ces services peut être décrit par une méthode incluse dans le contour du composant incluant le contour de l'interface. Ainsi, le contour d'une interface fournie correspond à un ensemble de noms de méthodes contenues dans le contour de composant associé à ce contour d'interface.

Interface requise. L'interface requise d'un composant est un point de contact entre ce composant et un connecteur. Comme une interface fournie, elle se décompose en services que doit fournir le connecteur à travers le rôle. Notre modèle associe donc à chaque service requis une méthode. Le contour d'une interface requise d'un contour de composant A correspond donc à un ensemble de noms de méthodes qui n'appartiennent pas aux classes de A. L'appartenance des méthodes à une entité différente du composant concerné marque la différence entre les interfaces requises et fournies.

De plus, pour nous permettre de distinguer les relations de compositions verticales et horizontale entre les composants, nous distinguons les contours d'interfaces requises verticales et horizontales.

Contours des connecteurs. Les connecteurs décrivent les communications entre les composants. Comme les fonctionnalités du système, cet aspect de communication est, dans le paradigme objet, inclu dans les classes. Cependant, au contraire des fonctionnalités de métiers, les aspects de communication sont souvent répartis entre les classes et ont donc une granularité plus faible. Nous proposons de définir les contours des connecteurs comme un ensemble de méthodes et d'attributs.

Ce contour se décompose en plusieurs éléments (*cf.* Figure 3.13). Le premier élément est le noyau. Il représente la glu du connecteur. C'est un ensemble de méthodes et d'attributs pouvant appartenir à différentes classes. Par contre, ils appartiennent aux classes contenues dans les contours de composants interconnectés par le connecteur représenté par le contour. Ainsi, le contour d'un connecteur rassemble les différentes parties des classes qui gèrent la communication entre les classes de différents contours de composants.

Les autres éléments du contour du connecteur sont les façades qui représentent les différents rôles du connecteur. Pour chaque contour de composant en contact avec lui, le contour de connecteur contient une façade. Cette façade, à l'image des interfaces est un ensemble de noms de méthodes. Ces méthodes peuvent appartenir au noyau du contour. Elles représentent, dans ce cas, un ensemble d'interfaces fournies proposé à travers ce rôle par le connecteur. Mais ces méthodes peuvent également ne pas appartenir à ce noyau. Elles représentent, alors, un ensemble d'interfaces requises qui doivent être proposées par le composant qui se connecte au connecteur à travers ce rôle.

Les contours des connecteurs ont une granularité plus fine que ceux des composants. Ils peuvent ainsi décrire des aspects répartis dans différentes classes. Les contours d'interfaces ont une granularité plus petite ceux des connecteurs qui sont aussi des ensembles de méthodes. En effet, les interfaces décrivent des points de contact entre entités alors que les connecteurs décrivent une entité logicielle.

Contour de configuration. La configuration décrit l'ensemble des composants, des connecteurs et la topologie de leurs connexions. Le contour de la configuration est donc un triplet $< M, N, E >$:

- M est un ensemble de contours de composants. Cet ensemble représente l'ensemble des composants présents dans l'architecture. Nous imposons que le regroupement des classes selon ces

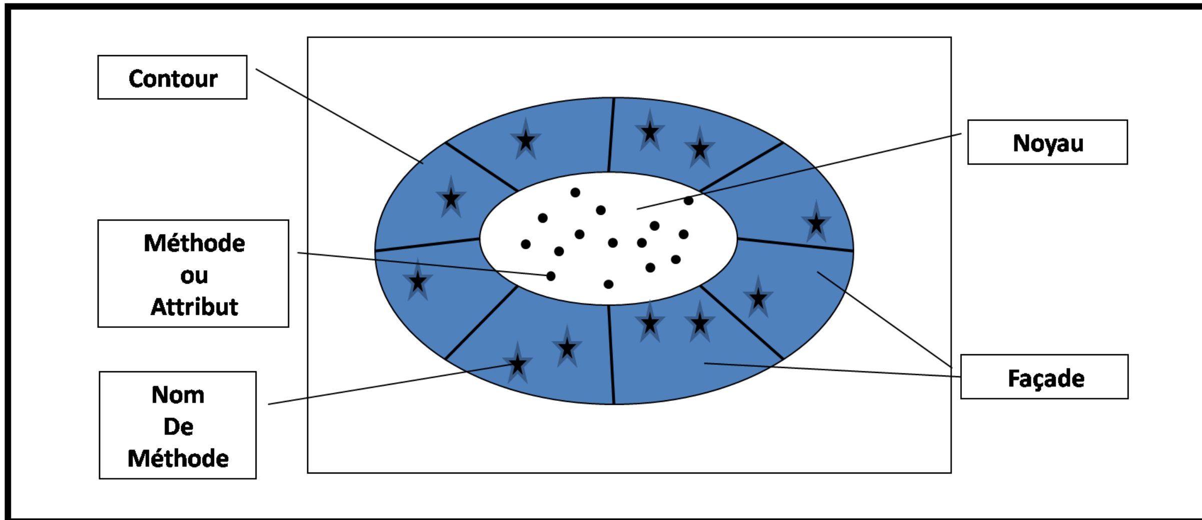


Figure 3.13 – Éléments du contour de connecteur

contours forme une partition des classes du système. Ainsi, chaque classe doit appartenir à un et un seul contour. Cette hypothèse permet d'éviter le problème de la duplication de classes dans plusieurs composants. Mais, elle nous garantit également d'obtenir une architecture dont les composants ne sont pas partagés ;

- N est un ensemble de contours de connecteurs. Cet ensemble représente les connecteurs utilisés par l'architecture pour relier les différents composants. Par construction, les ensembles de méthodes et d'attributs présents dans les contours doivent être disjoints. Cette contrainte s'ajoute à la contrainte portant sur les contours de composants imposant une partition des classes du système, pour renforcer son impact sur la duplication de code. En effet, la contrainte sur les contours de composants n'est pas suffisante pour garantir la non-duplication du code et en particulier des attributs. Ainsi, comme chaque classe appartient à un unique composant, les contours de connecteurs reliés à différents contours de composants ne peuvent contenir un même attribut ou méthode. Cependant, la contrainte sur les contours de composant n'empêche pas que des contours de connecteurs reliés à un même contour de composant puissent contenir une même méthode ou attribut ;
- E est un ensemble de *connexions*. Cet élément du modèle de correspondance est une classe d'association qui représente les connexions entre les contours de composants et de connecteurs. A ce titre, les contraintes portant sur E ont pour objectif de vérifier la validité de l'assemblage représenté par ce contour de configuration en vérifiant la compatibilité des interfaces des composants et des rôles des connecteurs.
  La première contrainte impose que les classes dont certaines méthodes ou attributs sont contenus dans le noyau du contour d'un connecteur, appartiennent à des contours de composants qui ont une relation de connexion avec le contour de connecteur.
  La deuxième contrainte porte plus précisément sur les interfaces et rôles. Elle impose que chaque façade d'un contour de connecteur soit un sous-ensemble de l'union des contours d'interfaces d'un des contours de composants qui possède une relation de connexion avec ce contour de connecteur.

Exemple 3.1 (Correspondance entre un ensemble de classes et les éléments architecturaux).
La figure 3.14 représente un diagramme de classes simple, les contours de composants, de connecteurs et d'interfaces identifiables parmi les entités objets présentes et enfin les composants et connecteurs as-

sociés à cette instance de COA. Le diagramme définit sept classes permettant la gestion d'un panier dans une application de commerce en ligne. Les classes sont représentées avec leurs méthodes et attributs. Les classes du diagramme peuvent être réparties en trois groupes : le panier, le contrôle et le dialogue. Ces trois groupes permettent de répartir les classes en deux contours de composants : le Panier et l'Affichage. L'identification des interfaces des composants correspondants nécessite l'identification du connecteur entre ces deux composants.
Pour définir le contour de ce connecteur, nous utilisons les méthodes et attributs définis dans les classes GestionPanier et ControlPanier. Ces deux classes constituent en effet le point de communication entre les deux contours de composants. Ainsi, les méthodes de GestionPanier forment le noyau du contour de connecteur.
Les deux façades du contour de connecteur sont définies à partir des méthodes du noyau. Ainsi, les méthodes du noyau qui appartiennent à GestionPanier sont utilisées par cette classe et le connecteur doit donc fournir ces méthodes dans un rôle. Elles forment donc une des façades du contour du connecteur. De plus ces méthodes utilisent les méthodes de ControlPanier qui doivent donc être fournies au connecteur par un composant. Les méthodes de ControlPanier constituent donc une autre façade du contour du connecteur. Au final, le contour du connecteur possède deux façades représentant chacune un rôle constitué d'une interface unique, fournie ou requise.
Les interfaces des composants sont directement issues des rôles définis par le connecteur. Ainsi, les contours des interfaces des composants Affichage et Panier sont les noms de méthodes qui forment les façades du contour du connecteur. Le contour de l'interface du composant Affichage représente une interface requise et contient les noms des méthodes de la classe GestionPanier. Au contraire, le contour de l'interface du composant Panier représente une interface fournie et contient les noms des méthodes de la classe ControlPanier.

Validité du modèle de correspondance. Les éléments contenus dans le contour de configuration permettent de déterminer précisément l'ensemble des éléments architecturaux. En effet, le contour de configuration définit un ensemble de composants, de connecteurs ainsi que la topologie des liens entre eux. De plus, l'ensemble des composants est accessible à travers cette configuration, de même que l'ensemble des connecteurs. Notre modèle de correspondance permet donc d'établir les liens entre les différents éléments architecturaux et les entités objets.

3.3.3 Notre approche par exploration

Notre approche par exploration repose sur trois aspects principaux qui définissent totalement le processus d'extraction d'architectures. Le premier aspect est l'espace des solutions du problème. Le deuxième aspect est un sous-espace de celui des solutions qui contient les solutions qui peuvent être effectivement explorées. Le dernier aspect concerne la méthode d'exploration en elle-même.

3.3.3.1 L'espace des solutions

Un processus d'exploration consiste à explorer l'ensemble des solutions possibles à un problème pour déterminer la meilleure. La première étape pour définir notre processus d'exploration, est donc la définition de l'espace des solutions, c'est-à-dire l'espace contenant toutes les architectures qui peuvent être extraites du système étudié.

La définition de cet espace des solutions est basée sur notre modèle COA. Il établit les liens entre les éléments architecturaux et les éléments objets. Ainsi, chaque instance de ce méta-modèle nous fournit un

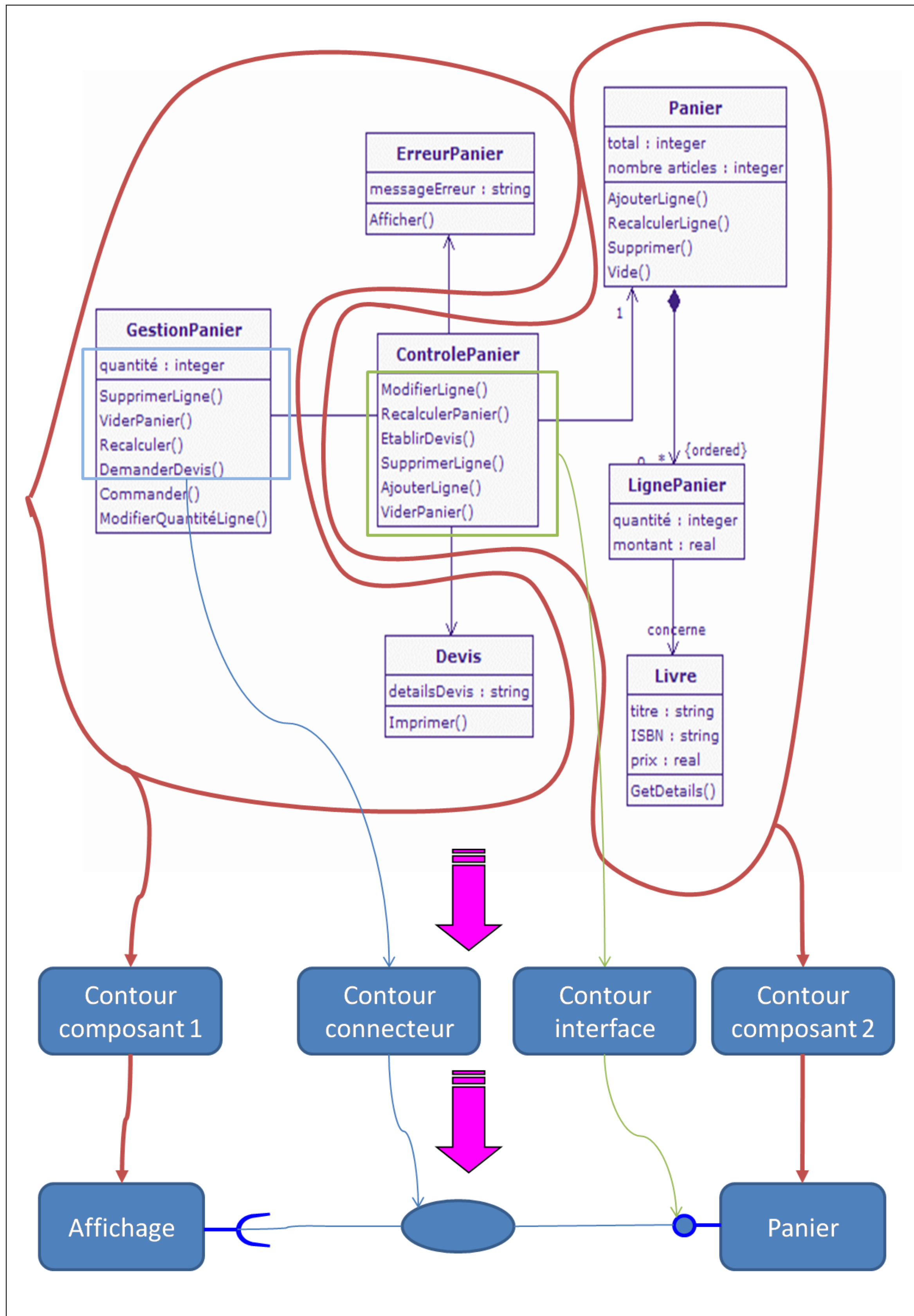


Figure 3.14 – Correspondance entre un ensemble de classes et les éléments architecturaux

ensemble de composants, de connecteurs et une configuration ainsi que les liens entre ces entités et les entités du système. L'instance obtenue est donc une architecture pouvant être extraite du système étudié. L'espace des solutions est l'ensemble des instances de notre modèle COA (*cf.* Figure 3.15).

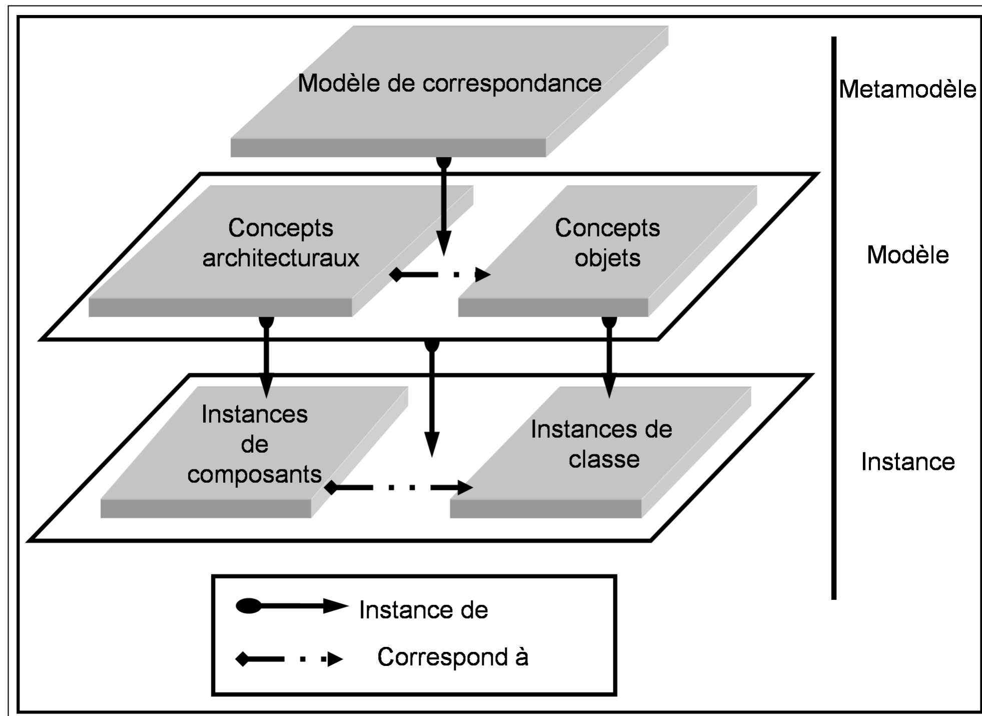


Figure 3.15 – Modèle de mise en correspondance (COA) et architecture à base de composants

D'après cette définition, l'espace des solutions de notre problème d'exploration inclut toutes les partitions des classes du système. Cela signifie, que pour un système contenant n classes, l'espace des solutions contient $O(n!)$ architectures possibles.

3.3.3.2 L'espace de recherche

Les approches par exploration sont, la plupart du temps, utilisées pour résoudre des problèmes ayant un espace de solutions très grand. Ces approches méta-heuristiques permettent en effet d'explorer un tel espace de manière plus rapide qu'une approche algorithmique spécifique, en obtenant des résultats de niveau comparable ou meilleur. Cependant, comme toute approche, un espace de solutions plus réduit augmente l'efficacité en terme de qualité du résultat et de temps d'exécution. Aussi, les processus d'exploration limitent leur parcours à un sous-espace de l'espace des solutions appelé l'espace de recherche.

Cet espace de recherche contient les solutions qui peuvent effectivement être explorées durant le processus. Il peut être égal à l'espace des solutions. Dans ce cas, le processus peut potentiellement avoir à explorer l'ensemble des solutions possibles du système pour trouver la meilleure.

Nous proposons de réduire cette espace de recherche en définissant un ensemble de guides pour notre processus d'extraction. Ces guides doivent permettre d'éliminer de l'espace explorable les solutions qui ne peuvent pas convenir. Cette élimination repose sur une évaluation a priori de la qualité d'une solution. Par exemple, l'architecte réalisant l'extraction peut décider que la solution qu'il recherche doit remplir un certain nombre de critères. Dans ce cas, toutes les solutions qui ne respectent pas ces critères peuvent

être supprimées de l'espace de recherche sans préjudice sur la qualité du résultat, mais en améliorant le temps d'exécution.

3.3.3.3 L'exploration

Le dernier aspect de notre approche par exploration est la méthode d'exploration utilisée pour parcourir l'espace de recherche. En effet, la méthode utilisée a des conséquences sur la performance temporelle, mais aussi en terme de qualité, de notre processus. L'approche par exploration systématique étant impossible pour des raisons évidentes de complexité, nous avons orienté notre choix vers une approche à base de méta-heuristiques. Ces méta-heuristiques d'exploration sont nombreuses mais nous pouvons distinguer deux classes principales : les algorithmes séquentiels et les algorithmes génétiques.

Les algorithmes séquentiels. Ces algorithmes explorent l'espace de recherche à partir d'une solution initiale en utilisant une fonction objectif pour déterminer la meilleure solution. Ils procèdent par modification successive de la solution courante. La méta-heuristique décrit comment se fait le passage d'une solution à une autre et dans quelles conditions. Par exemple, l'algorithme *Hill-Climbing* [107] sélectionne comme prochaine solution courante, la première solution, atteinte à partir de la solution actuelle, qui améliore le résultat de la fonction objectif. Plus complexe, l'algorithme *Tabu-search* [107] impose en plus que la solution suivante n'ai pas déjà été explorée.

Ces algorithmes sont stochastiques. Chaque exécution conduit à une solution potentiellement différente. La qualité du résultat dépend donc de la fonction objectif et de la qualité de l'espace de recherche mais surtout du temps accordé à l'exécution.

Les algorithmes génétiques. Ces algorithmes simulent les processus biologiques [65]. A partir d'une population de solutions, les algorithmes génétiques génèrent une nouvelle population en générant de nouvelles solutions par reproduction au sein de la population actuelle et en sélectionnant les individus en fonction d'une fonction objectif. Dans ces algorithmes, une heuristique détermine la façon dont les nouvelles solutions sont générées et dont les individus sont sélectionnés.

Comme les algorithmes séquentiels, ces algorithmes sont stochastiques. La qualité de la solution dépend du temps accordé à l'exécution mais surtout des opérateurs de génération et de sélection, des conditions de leurs déclenchements et de la population initiale.

Toutes ces approches présentent, dans le cas général, des avantages et des défauts. Il nous faut comparer les résultats de ces approches dans le cas particulier du problème d'extraction et de notre espace de solutions. Pour cela, nous proposons une approche d'exploration générique en définissant l'ensemble des éléments nécessaires pour tous ces algorithmes. A partir de ce modèle d'approche d'exploration, nous proposons une instanciation par classe d'algorithmes afin de pouvoir comparer les avantages et inconvénients de chacune dans le cadre de notre modélisation du problème d'extraction.

3.4 Guides du processus d'extraction d'architectures

Pour explorer l'espace des solutions, notre processus nécessite des éléments permettant de le guider parmi l'ensemble des architectures possibles. Nous avons identifié plusieurs guides (*cf.* Figure 3.16) permettant de diriger l'exploration des solutions et d'orienter notre processus vers les architectures les plus pertinentes, que se soit en terme de qualité ou vis-à-vis des attentes des utilisateurs.

Ces guides influencent le processus d'exploration de différentes manières. Certains permettent d'identifier l'objectif de l'exploration en déterminant la qualité des solutions. Ils définissent la fonction objectif de notre processus d'exploration. Les autres guides permettent de faire une première sélection des solutions, en rejetant celles qui sont clairement mauvaises et en orientant l'exploration vers celles qui sont probablement bonnes. Ils définissent l'espace de recherche à partir de l'espace des solutions. Nous présentons donc les guides de notre processus selon deux types : les guides pour l'identification de l'objectif et les guides pour la réduction de l'espace de recherche.

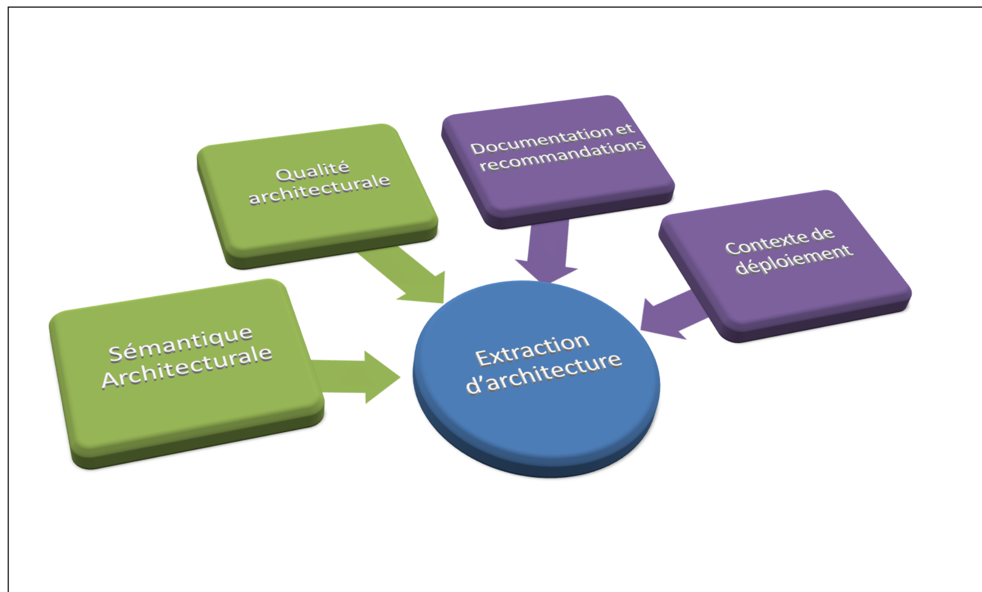


Figure 3.16 – Guides de l'extraction d'architectures

3.4.1 Guides pour l'identification de l'objectif

Déterminer la qualité d'une solution constitue la principale activité d'un processus d'exploration. En effet, celui-ci parcourt l'espace des solutions et doit déterminer à chaque étape la qualité de la solution explorée et ainsi décider s'il s'agit de la meilleure solution rencontrée. La définition d'une mesure de la qualité des solutions est donc requise pour utiliser une approche par exploration. Pour définir cette mesure, appelée fonction objectif, nous utilisons deux guides qui sont la sémantique et la qualité architecturale.

3.4.1.1 Sémantique architecturale

La qualité d'une solution du problème d'extraction tient d'abord au respect de la définition du concept d'architecture. En effet, une partition des classes en contours constitue une bonne architecture si elle correspond à la définition couramment admise du concept architecture, c'est-à-dire si elle correspond à une architecture sémantiquement valide. Nous considérons qu'une architecture est sémantiquement valide si ses éléments (composants, connecteurs et configuration) sont sémantiquement valides.

Nous proposons donc de réifier les propriétés sémantiques des éléments architecturaux pour définir notre fonction objectif. Pour cela, nous avons étudié les définitions des concepts composant et connecteur. A partir de leurs définitions les plus répandues et utilisées, nous avons proposé une définition précise et

correspondant à l'idée couramment admise de ces concepts. Ces définitions nous permettent ensuite de proposer un ensemble de propriétés sémantiques pour ces éléments architecturaux. Ces propriétés doivent être vérifiées par les composants ou les connecteurs pour qu'ils soient considérés comme sémantiquement valides.

Nous utilisons ces propriétés pour définir une mesure de la validité sémantique pour chacun des éléments architecturaux. Ces mesures s'appliquent sur les entités de notre modèle de correspondance et permettent de mesurer la validité sémantique des éléments architecturaux qui correspondent à ces entités. Finalement ces mesures de la validité sémantique sont combinées en une fonction de mesure de la validité sémantique d'une architecture qui s'applique aux éléments de notre espace de solution.

Nous utilisons cette fonction pour mesurer la qualité des solutions rencontrées durant l'exploration. Cette mesure est la fonction objectif de base de notre processus. Elle permet une recherche de solutions correspondant à une architecture.

3.4.1.2 Qualité architecturale

La qualité d'une solution dépend également de la qualité de l'architecture qu'elle représente. En effet, la validité sémantique n'est pas la seule qualité d'une architecture et de nombreux travaux [83] ont établi des listes de qualités pour les architectures telles que la maintenabilité ou la fiabilité.

Baser notre fonction objectif sur les qualités de l'architecture, en plus de sa validité sémantique, permet à notre processus d'extraire une meilleure architecture par rapport à certains objectifs de qualité. Pour cela, nous avons étudié la notion de qualité architecturale et ses caractéristiques. Par cette étude, nous avons choisi plusieurs caractéristiques essentielles pour les architectures. Nous avons également procédé à une analyse similaire pour les différents éléments architecturaux et nous avons obtenu d'autres caractéristiques de qualité portant, cette fois, sur ces éléments.

A partir de ces caractéristiques de qualité, nous proposons un ensemble de mesures pour évaluer les qualités des éléments architecturaux et de l'architecture. Ces mesures permettent de définir une fonction d'évaluation de la qualité architecturale. Finalement, cette fonction peut être combinée avec la fonction de mesure de la validité sémantique pour définir une nouvelle fonction objectif pour notre processus permettant de rechercher une solution correspondant à une architecture de qualité.

3.4.2 Guides pour la réduction de l'espace de recherche

La taille de l'espace des solutions rend nécessaire sa réduction à un espace de recherche plus petit. Pour obtenir cette réduction, nous devons fournir un moyen au processus de se diriger préférentiellement vers les solutions potentiellement bonnes en évitant au maximum l'exploration des solutions qui sont, de manière «évidente», mauvaises.

Nous proposons deux guides permettant de réduire l'espace de recherche. Ils fournissent une description des solutions potentiellement bonnes ainsi que des mauvaises solutions. Ils permettent aussi d'identifier des zones de l'espace des solutions qui contiennent chaque type de solutions (bonnes ou mauvaises).

3.4.2.1 Documentation et recommandations

La documentation d'un logiciel contient de nombreuses informations architecturales. Ces informations décrivent plus particulièrement l'architecture intentionnelle du système. Elles permettent donc d'obtenir une vue architecturale de ce que les concepteurs du système ont souhaité obtenir. Nous souhaitons utiliser cette vue pour réduire l'espace de recherche de notre processus. Cette réduction intervient

en permettant au processus d'ignorer les solutions qui sont impossibles du point de vue de la documentation ou rejetées par l'utilisateur. Cette réduction consiste également à permettre au processus de faire une sélection parmi les solutions contenues dans l'espace de recherche, en fonction de leur adéquation avec l'architecture intentionnelle extraite de la documentation.

Parmi les nombreuses documentations disponibles tout au long du cycle de vie du logiciel, nous avons sélectionné quatre types qui semblent les plus pertinents. Le guide documentation peut ainsi être éclaté en un ensemble de quatre guides (*cf.* Figure 3.17) : les recommandations de l'architecte, les diagrammes UML, la documentation incluse dans le code source et les outils de versionnement. Nous proposons aussi un modèle de documentation qui permet de facilement prendre en compte un nouveau type de documentation.

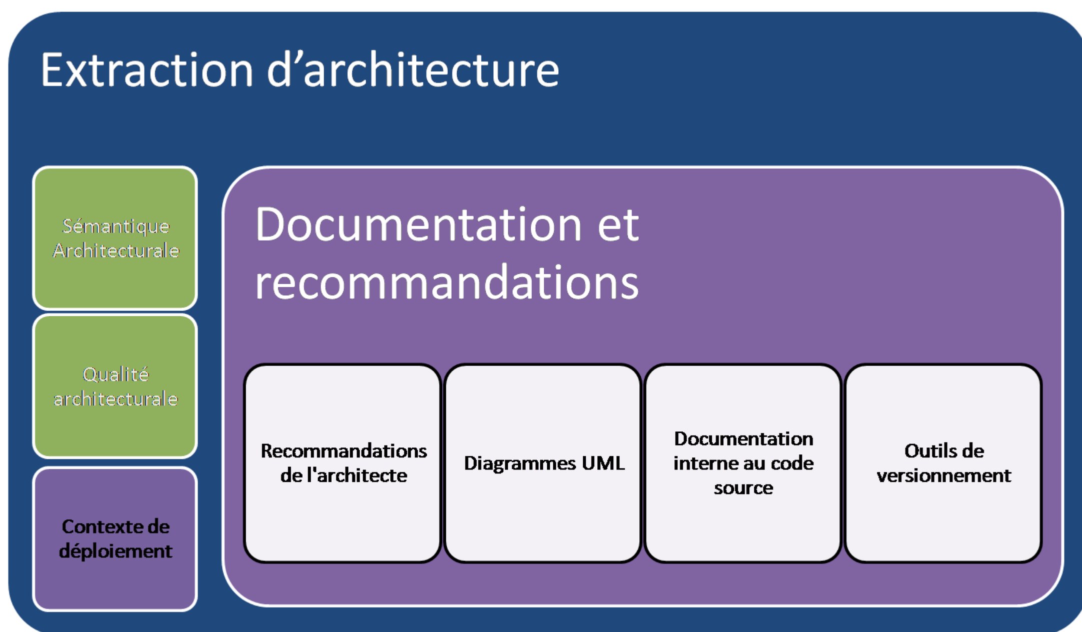


Figure 3.17 – Les documents et les recommandations disponibles

Recommandations de l'architecte. L'architecte possède des informations sur le système qui peuvent être plus ou moins précises en fonction de son expertise. Ces recommandations peuvent être de différents types.

D'abord, elles peuvent prendre la forme d'une proposition ou d'une validation. Notre processus permet ainsi à l'architecte de fournir, à différents moments, des propositions d'informations telle qu'une partition des classes en composants. Il interroge également régulièrement l'architecte pour obtenir une validation des résultats. L'architecte peut alors accepter le résultat, le rejeter ou ignorer la demande.

Les recommandations fournies par l'architecte peuvent également prendre la forme d'une information positive ou négative. Dans le premier cas, l'information met en lumière une relation, alors que le second cas précise l'absence de relation entre entités.

Diagrammes UML. Les diagrammes UML sont utilisés dès les premières étapes du cycle de vie pour modéliser les relations entre les classes et les fonctionnalités qui leurs sont associées. Ces relations entre entités objets et entre fonctionnalités sont utilisées pour obtenir une vue de l'architecture intentionnelle

et donc une vue partielle de l'architecture du système. Nous utilisons cette vue partielle pour identifier les éléments probablement bons de l'espace des solutions ainsi que pour éviter les éléments qui sont en totale contradiction avec la vue architecturale fournie par les diagrammes UML.

Documentation interne au code source. Le code source est un autre type de documents écrits. En effet, il contient des commentaires et des noms d'entités qui sont choisis pour donner une indication sur la sémantique des opérations. Ces informations nous permettent de regrouper les classes en fonction de la partie métier qu'elles réalisent, en utilisant par exemple un algorithme telle que LSA «Latent Semantic Analysis» [79]. Nous utilisons ensuite ces regroupements pour orienter notre processus vers les solutions présentant des regroupements similaires et qui correspondent donc à l'architecture intentionnelle reflétée par les éléments de documentation présents dans le code source.

Outils de versionnement. Les phases de maintenance font souvent l'objet d'un suivi précis. En particulier, les outils de versionnement fournissent des informations sur les modifications apportées aux fichiers et sur la simultanéité de ces modifications. Ces informations établissent des liens entre les entités du code source. Ainsi, deux méthodes modifiées simultanément sont probablement liées par une relation dans le code source ou intentionnelle. Nous proposons d'utiliser les relations identifiées en utilisant les données des outils de versionnement pour calculer une nouvelle vue de l'architecture intentionnelle. Cette vue architecturale permet, comme pour les diagrammes UML, d'identifier les éléments probablement bons de l'espace de solutions et d'éviter les éléments qui sont en totale contradiction avec elle.

3.4.2.2 Contexte de déploiement

L'architecture matérielle du système sur lequel le logiciel est déployé a une forte influence sur son architecture. Ainsi, un environnement constitué de nombreuse machines distribuées et disposant de ressources limitées n'impose pas la même architecture qu'un environnement constitué d'un unique serveur de calcul. Ces contraintes sur le contexte de déploiement peuvent avoir subies des modifications pendant l'évolution du système logiciel. Il est donc essentiel de prendre en compte le contexte de déploiement lorsqu'on réalise l'extraction de l'architecture logicielle.

Par conséquent, nous utilisons les propriétés du contexte pour guider notre processus d'extraction afin d'obtenir une meilleure adéquation entre les architectures matérielles et logicielles. Ainsi, nous utilisons le contexte pour éliminer de l'espace de recherche les solutions qui ne peuvent pas être déployées sur cette architecture matérielle. Par exemple, l'adaptation d'une architecture extraite à une utilisation embarquée impose des limitations sur la taille des composants ou le nombre de connecteurs.

3.5 Conclusion

Nous avons présenté, dans ce chapitre, les motivations et les principes de notre approche. Nous sommes revenus sur les travaux présentés dans le chapitre 2. Nous avons souligné les limites auxquelles ils sont confrontés aussi bien au niveau conceptuel que technique ou des informations utilisées.

Pour pallier ces limitations, nous proposons une approche d'extraction d'architectures logicielles itérative, interactive et semi-automatique. Notre approche utilise des méta-heuristiques pour explorer l'espace des solutions défini par notre modèle COA. Ce modèle établit les correspondances entre les entités objet et les éléments architecturaux. Pour cela, il définit un ensemble d'éléments appellés contours qui sont constitués d'entités objet et associés à un élément architectural.

L'exploration de l'espace des solutions est dirigée par une fonction objectif basée sur la qualité architecturale de la solution et son adéquation avec les définitions du concept d'architecture. Elle s'appuie également sur l'architecture intentionnelle et matérielle pour affiner l'exploration en ciblant les solutions compatibles avec ces architectures. Pour cela, nous avons présenté les principes d'une extraction de l'architecture intentionnelle à partir des recommandations de l'architecte et de la documentation, en particulier les diagrammes UML, les informations contenues dans le code source et les archives des outils de versionnement.

CHAPITRE 4

Fondement du processus d'extraction : l'identification des objectifs

Management by objective works - if you know the objectives. Ninety percent of the time you don't.

— Peter F. DRUCKER.

Fonction objectif — Sémantique architecturale — Qualité architecturale — Définition des composants — Définition des connecteurs — Norme ISO-9126

Dans le chapitre précédent, nous avons défini l'espace des solutions de notre problème d'extraction d'architectures. Cependant, un processus d'exploration nécessite plus que la simple définition de cette espace. En effet, notre processus d'exploration a besoin d'un moyen de se diriger dans l'espace et de choisir la solution du problème. Ces éléments constituent le fondement de notre processus puisqu'ils déterminent directement la qualité de la solution et le parcours pour la trouver dans l'espace des solutions, c'est-à-dire le temps de recherche.

Pour définir ces fondements, nous devons d'abord définir l'objectif de notre processus. La définition de l'espace des solutions répond partiellement à cette question en définissant la solution comme étant une partition des éléments du code objet. Cependant, l'architecture à extraire n'est pas pour autant un élément quelconque de l'espace des solutions. En effet, si chacun des éléments de l'espace peut, d'après notre modèle, représenter une architecture, l'architecture attendue par l'architecte doit respecter des contraintes supplémentaires. Ainsi, nous devons étudier les caractéristiques essentielles que doit avoir un élément de l'espace des solutions pour être considéré comme une architecture acceptable par l'architecte.

Le processus nécessite, aussi, la définition d'une fonction permettant de mesurer la qualité d'une solution par rapport aux caractéristiques objectifs. Cette fonction permet de sélectionner la meilleure solution, mais surtout elle rend possible l'exploration de l'espace des solutions en permettant une comparaison entre les entités de cet espace.

Pour établir ces deux points, nous utilisons deux guides qui établissent les caractéristiques que l'architecte attend de l'architecture extraite :

- **le premier guide** utilise la sémantique associée au concept d'architecture pour définir la validité sémantique d'une solution. Cette propriété d'une solution désigne l'adéquation entre l'entité solution et la définition associée au concept d'architecture. Elle est la première attente de l'architecte puisqu'il désire utiliser la solution pour représenter l'architecture logicielle du système ;
- **le deuxième guide** définit les caractéristiques de qualité d'une architecture. En effet l'architecte n'a pas pour seul objectif d'extraire une architecture quelconque, mais il souhaite plutôt extraire une bonne architecture. Nous devons donc définir les caractéristiques d'une bonne architecture afin de mesurer la qualité des solutions de ce point de vue.

La définition d'une fonction objectif à partir de ces guides nécessite trois étapes :

1. **définition du méta-modèle de mesure** : nous proposons un méta-modèle de mesure dérivé de celui proposé par la norme ISO-9126 [44] ;
2. **définition du modèle de mesure** : nous utilisons ce méta-modèle pour instancier un modèle de mesure de la qualité d'une solution en étudiant les caractéristiques de validité sémantique et de qualité architecturale, issues de chacun des guides ;
3. **Définition des fonctions d'évaluation** : nous utilisons le modèle de mesure pour définir deux fonctions d'évaluation. La première mesure la validité sémantique d'une solution alors que la deuxième mesure la qualité de la solution en terme de qualité architecturale.

La fonction objectif de notre processus est alors définie en combinant ces fonctions d'évaluation. Elle permet ainsi à notre processus de rechercher l'architecture qui correspond le mieux au concept et qui démontre les meilleures qualités.

La suite du chapitre est organisée de la façon suivante. Nous présentons d'abord le méta-modèle de mesure que nous souhaitons instancier. Ensuite, nous étudions la sémantique architecturale et nous proposons un modèle de mesure de la caractéristique de validité sémantique. De la même façon, nous étudions la qualité architecturale et nous présentons un modèle de mesure similaire pour cette caractéristique. Enfin, nous concluons ce chapitre en présentant les fonctions d'évaluations obtenues à partir de chacun de nos modèles de mesure, et en définissant la fonction objectif de notre processus d'exploration.

4.1 Méta-modèle de mesure

Pour définir la fonction objectif de notre processus, nous avons besoin de définir un modèle de mesure de la qualité des solutions basé sur les caractéristiques définies à travers les guides que nous présentons dans ce chapitre. Pour cela, nous présentons d'abord un méta-modèle de mesure que nous utilisons par la suite pour diriger l'étude des guides.

La norme ISO-9126 [44] propose un modèle de mesure de la qualité du logiciel qui repose sur un méta-modèle de mesure générique. Cependant, ce méta-modèle générique doit être adapté pour tenir compte des spécificités de notre approche. Nous présentons donc le méta-modèle de la norme ISO-9126 puis notre méta-modèle.

4.1.1 Méta-modèle de mesure de la norme ISO-9126

La norme ISO-9126 [44] propose une définition de la qualité des systèmes informatiques. Dans ce cadre, elle propose un modèle de mesure de la qualité des systèmes. Ce modèle repose sur un méta-

modèle inspiré des travaux existants sur la mesure de la qualité et en particulier de ceux de Mc-Call [26].

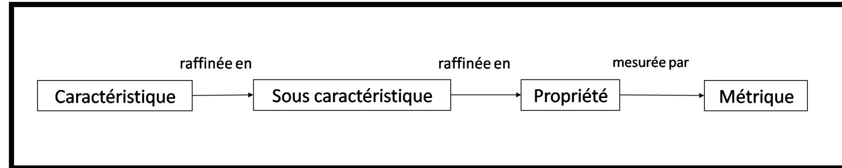


Figure 4.1 – Méta-modèle de mesure des caractéristiques d'un produit logiciel dans la norme ISO-9126

Le méta-modèle de mesure de la norme ISO-9126 (*cf.* Figure 4.1) repose sur quatre entités qui constituent les quatre niveaux du modèle de mesure. La *caractéristique* est l'entité qui représente le premier niveau du modèle. Elle représente les différentes caractéristiques de l'attribut qui doivent être mesurées. Ainsi, le modèle de mesure de la qualité du logiciel proposé par la norme, décompose la qualité en six caractéristiques telles que la fonctionnalité, la fiabilité, la maintenabilité ou encore la portabilité. Chacune de ces caractéristiques décrit un aspect de la qualité logiciel plus ou moins indépendant des autres caractéristiques.

L'élément *sous-caractéristique* décrit le second niveau du modèle de mesure. Cette entité représente les sous-caractéristiques du modèle. Elles constituent un niveau de raffinement qui permet d'affiner le niveau des caractéristiques en précisant ce qu'on attend de chaque caractéristique. En effet, chacune des sous-caractéristiques dépend d'une caractéristique et en précise le sens. Par exemple, dans la norme ISO, la caractéristique *efficacité* est raffinée en trois sous-caractéristiques : la consommation en temps, la consommation en ressource et la conformité.

L'entité *propriété* décrit le troisième niveau du modèle de mesure. Cet élément représente les propriétés mesurables de l'entité sur laquelle porte l'étude. Elles font le lien entre les caractéristiques étudiées et l'objet de l'étude en définissant les éléments mesurables sur l'objet. Chaque propriété raffine un ensemble de sous-caractéristiques et permet ainsi d'associer chaque caractéristique abstraite à un élément mesurable de l'objet. Dans la norme ISO, l'instanciation de ce niveau est laissée à la charge de l'utilisateur. En effet, les propriétés qui sont effectivement mesurables sur un système varient en fonction de facteurs divers tels que la technologie utilisée, l'accès au code source ou encore les outils disponibles. La norme permet ainsi à chaque utilisateur de définir son propre jeu de propriétés.

La *métrique* est la dernière entité du méta-modèle. Elle définit le dernier niveau du modèle de mesure et représente les métriques qui permettent de mesurer les propriétés mesurables du niveau précédent. Comme pour le niveau des propriétés mesurables, ce niveau n'est pas instancié dans la norme ISO-9126.

Ce méta-modèle est générique et peut être utilisé pour des mesures portant sur une grande variété d'entités. Cependant, il reste difficile à appliquer directement dans notre cas. En effet, dans le cadre de l'extraction d'architectures, les caractéristiques décrivent une architecture alors que les métriques portent sur des entités du modèle COA (modèle de correspondance objet/architecture).

L'utilisation de ce méta-modèle de mesure pour l'extraction impose donc un changement dans les entités de référence. Ce changement peut se faire entre les sous-caractéristiques et les propriétés ou entre ces dernières et les métriques. Ce deuxième cas rend particulièrement complexe une étape qui est quasi-transparente en temps normal. Ce choix ne semble donc pas approprié.

Le changement d'entités lors du passage des sous-caractéristiques aux propriétés est également problématique. En effet, cette étape cruciale est déjà, dans l'utilisation courante, difficile. L'ajout de ce

changement la rend encore plus délicate et donc potentiellement source d'erreurs.

Au final, il est difficile d'utiliser l'une ou l'autre de ces approches. Il est donc nécessaire de proposer un méta-modèle plus adapté à notre problème.

4.1.2 Méta-modèle de mesure dans ROMANTIC

Nous avons vu les avantages et les limites du méta-modèle de la norme ISO-9126. Nous proposons donc une extension de ce méta-modèle qui permet de prendre en compte facilement la particularité de notre approche, c'est-à-dire la présence de deux types d'entités différentes : l'architecture et les entités COA.

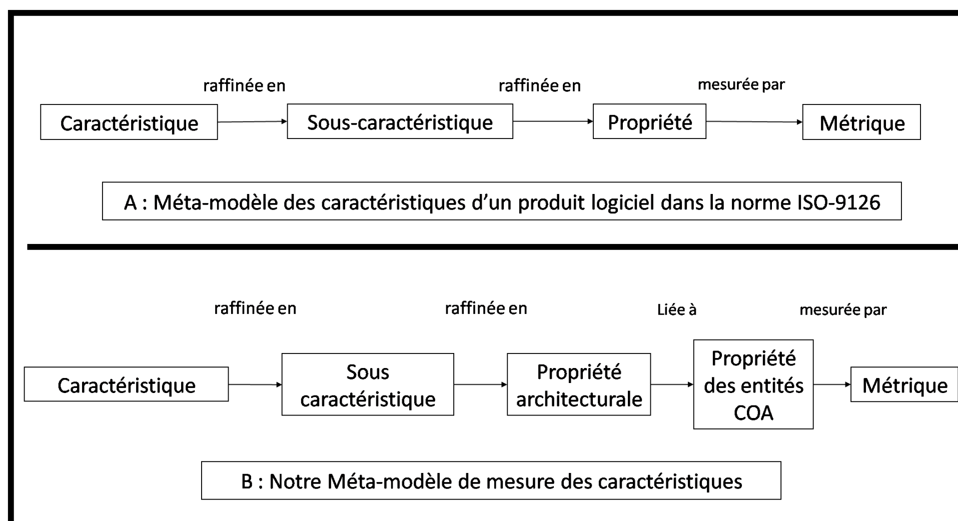


Figure 4.2 – Méta-modèle de mesure de caractéristique dans notre approche

Afin de gérer la prise en compte de deux entités différentes dans le modèle, nous devons intégrer, dans le méta-modèle, une étape de transition. Cette transition peut être intégrée entre les sous-caractéristiques et les propriétés ou entre les propriétés et les métriques. Dans les deux cas, le résultat est identique : nous obtenons un méta-modèle de cinq éléments (*cf.* Figure 4.2) dont les deux premiers, *caractéristique* et *sous-caractéristique*, sont identiques à ceux du méta-modèle ISO.

L'élément *propriété architecturale* correspond à l'entité *propriété* du méta-modèle ISO. Cet élément représente les propriétés architecturales raffinant les sous-caractéristiques. Il décrit le niveau 3 de notre modèle de mesure qui est le dernier niveau se rapportant à l'architecture.

L'élément *propriété des entités COA* est un ajout par rapport au méta-modèle ISO. Il est l'équivalent de l'élément précédent mais pour les entités du modèle COA au lieu de l'architecture. Il représente donc les propriétés des entités COA. Il décrit ainsi le niveau 4 de notre méta-modèle et le premier niveau se rapportant au modèle COA. Le passage du niveau 3 à ce niveau marque le passage de l'espace architectural à l'espace COA. Il est constitué par des correspondances entre les entités des deux niveaux, c'est-à-dire entre les propriétés des différentes entités.

L'élément *métrique* est relativement proche de l'entité du même nom, dans le méta-modèle ISO. Elle représente les métriques qui permettent de mesurer les propriétés des entités COA. La seule différence avec l'entité du méta-modèle ISO est donc le support qui sert au calcul de ces métriques : les entités COA ici ; l'architecture dans le méta-modèle ISO.

Grâce à l'ajout du niveau *propriété des entités COA*, notre méta-modèle facilite la prise en compte de deux entités différentes dans le modèle de mesure. Il est ainsi plus facile de définir les relations entre les caractéristiques architecturales et les métriques portant sur les entités COA.

Nous avons présenté ce méta-modèle en le projetant directement sur les entités que nous utilisons dans notre approche. Cependant, il peut être adapté à toutes entités, ainsi qu'à n'importe quel nombre d'espaces différents. De plus, le méta-modèle ISO est un cas particulier de cette généralisation, où il n'existe qu'une seule entité et qu'un seul espace.

Par la suite, nous étudions les deux guides qui nous permettent de définir les objectifs de notre processus. Cette étude utilise le chemin proposé par ce méta-modèle de mesure. Nous obtenons ainsi un modèle de mesure qui nous permet de définir la fonction objectif nécessaire à l'exploration.

4.2 Étude de la sémantique architecturale

La validité sémantique d'une architecture désigne l'adéquation entre l'entité que l'on désigne et la définition associée au concept d'architecture. Une architecture est donc sémantiquement valide si elle correspond à la définition du concept d'architecture. Le terme sémantique fait ici référence au concept et non à l'entité. Il ne désigne donc pas les fonctionnalités d'une architecture. Nous utiliserons, par la suite, le terme sémantique pour désigner la sémantique du concept et non de l'entité.

Cette caractéristique de validité sémantique est très intéressante dans le cadre d'une approche par exploration de l'extraction d'architectures logicielles. En effet, l'espace des solutions du problème d'extraction ne contient pas uniquement des solutions sémantiquement correctes. Il est donc essentiel de fournir à notre processus une méthode pour écarter ces solutions qui ne peuvent pas donner un résultat correct. Pour cela, nous étudions la sémantique de l'architecture afin de définir une fonction d'évaluation de la validité sémantique. L'utilisation de cette fonction, durant l'exploration, pour la sélection des solutions garantit que le résultat de l'extraction sera une architecture sémantiquement valide.

4.2.1 Caractéristiques sémantiques de l'architecture

La sémantique de l'architecture repose, avant tout, sur la sémantique des différents éléments architecturaux. Ainsi, une architecture est sémantiquement valide si ses éléments (composants, connecteurs et configuration) sont sémantiquement valides. Nous étudions, par la suite, la sémantique des composants et des connecteurs. Nous utilisons cette sémantique pour définir un modèle de mesure de la validité sémantique de chacun de ces éléments architecturaux et donc de l'architecture.

4.2.1.1 Caractéristiques sémantiques des composants

Afin de mesurer la validité sémantique des composants, nous étudions leurs caractéristiques sémantiques et nous proposons un modèle de mesure de cette validité. Cette étude se base sur les définitions les plus couramment admises des composants logiciels et architecturaux. En effet, la majorité des définitions, comme nous l'avons vu dans le chapitre 1, ne distingue pas ces deux notions autrement que par le caractère exécutable des composants logiciels au contraire des composants architecturaux. Nous pouvons ainsi utiliser la plupart des caractéristiques sémantiques contenues dans les définitions pour décrire les deux notions.

Il existe de nombreuses définitions qui caractérisent un composant. Ces définitions abordent différents aspects des composants logiciels ou architecturaux qui dépendent du domaine de l'auteur de la définition.

Néanmoins, il existe d'importantes similitudes parmi les définitions les plus courantes. Ainsi, l'étude des définitions les plus couramment admises et utilisées doit nous permettre de proposer une définition du composant architectural exprimant la sémantique de ce concept.

La définition proposée par SZYPERSKI [118] est probablement la plus utilisée. Il définit un composant comme « **une unité de composition possédant des interfaces spécifiées par contrats et des dépendances explicites avec le contexte. Il peut être déployé indépendamment et peut être composé par un tiers** ». La partie concernant le déploiement cible plus particulièrement des composants instanciés pouvant être exécutés, c'est-à-dire les composants logiciels. Cependant, cette définition s'adresse à la fois aux composants architecturaux et logiciels. Ainsi, la propriété de composabilité est essentielle, selon cette définition, pour considérer une entité comme étant un composant architectural. La définition souligne aussi le besoin d'autonomie de l'entité puisqu'elle doit être déployée indépendamment.

HEINEMANN et COUNCILL proposent une autre définition très utilisée [62]. Ils définissent un composant comme « **un élément logiciel qui est conforme à un modèle de composant et peut être déployé indépendamment et composé sans modification selon un standard de composition** ». Cette définition est très similaire à celle de Szyperski. Comme cette dernière, elle concerne les composants logiciels et architecturaux. Elle aborde la propriété de déploiement et met encore une fois l'accent sur les propriétés de composition et d'autonomie des composants.

D'autres proposent clairement deux définitions différentes pour les composants logiciels et architecturaux. Ainsi, LUER distingue les composants architecturaux et logiciels en proposant une définition pour les composants et une pour les composants déployables [85]. Il définit un composant comme « **un élément logiciel qui encapsule une implémentation réutilisable d'une fonctionnalité, peut être composé sans modification et adhère à un modèle de composant** ». Un composant déployable est lui défini comme « **un composant pré-paqueté, distribué indépendamment, facile à installer et désinstaller et auto-descriptif** ». Ces deux définitions soulignent clairement la propriété de déploiement comme la difference majeure entre le composant architectural et logiciel. La définition du composant met en valeur, comme les définitions précédentes, les propriétés de composabilité et d'autonomie des composants. Cependant, elle dévoile également la nécessité pour un composant d'encapsuler une fonctionnalité. La définition 4.1 précise le modèle de composant au sens de LUER.

Définition 4.1. Un modèle de composant est une combinaison de (1) un standard de composants qui gouverne la construction de chaque composant et (2) un standard de composition qui régit comment organiser un ensemble de composants en une application et comment ces composants communiquent et interagissent entre eux.

En combinant et en raffinant les éléments communs de ces définitions, nous proposons la définition suivante du composant :

Définition 4.2. Un composant est un élément logiciel qui (a) est composable sans modification, (b) peut être distribué de manière autonome, (c) encapsule une fonctionnalité, et (d) qui adhère à un modèle de composant.

On retrouve, dans cette définition du composant, les propriétés que nous avons mises en valeur dans les définitions couramment utilisées. Ainsi, l'encapsulation renvoie à l'implémentation d'une fonctionnalité de LUER. De même, la distribution autonome et la composition sans modification renvoient, respectivement, aux propriétés d'autonomie et de composabilité imposées par toutes les définitions précédentes.

De plus, le modèle d'un composant décrit dans notre définition, correspond à la définition 4.1. Cette définition du modèle de composant est suffisamment générique pour permettre de tenir compte, à la fois,

des modèles non-exécutables, qui décrivent les composants architecturaux, et des modèles exécutables, qui décrivent les composants logiciels. Notre définition reste valable pour les deux types de composants et nous déléguons au modèle de composants la distinction entre les composants logiciels et architecturaux.

En conclusion, d'après notre définition d'un composant architectural, nous avons identifié trois caractéristiques sémantiques des composants :

- **la composabilité** : cette caractéristique permet de vérifier que l'entité logiciel est composable sans modification. Elle décrit la capacité du composant architectural à être assemblé avec d'autres entités. Cette composition passe d'abord par la présence d'interfaces dans l'entité. Cependant, ces interfaces sont imposées à l'entité logicielle à travers l'adhésion au modèle de composant. La composabilité est donc surtout liée au contenu et au comportement des interfaces ;
- **l'autonomie** : cette caractéristique décrit la capacité de l'entité à être distribuée de manière autonome. Elle impose des limites sur les éléments requis par l'entité pour pouvoir être utilisée. Ces éléments sont, bien sûr, des interfaces requises. Cependant, un composant doit limiter le nombre d'interfaces requises et donc ces dépendances avec des éléments extérieurs ;
- **la spécificité** : cette caractéristique permet de vérifier que l'entité encapsule une fonctionnalité. Elle repose donc sur l'évaluation du nombre de fonctionnalités fournies par l'entité. Ces fonctionnalités peuvent être internes ou proposées à travers des interfaces. Dans tous les cas, la spécificité impose qu'un composant ait un nombre limité de fonctionnalités.

4.2.1.2 Caractéristiques sémantiques des connecteurs

Les connecteurs sont, au même titre que le composants, des entités de premier plan des architectures logicielles. Cependant, dans la pratique, la considération et l'utilisation qui est faite de chacune de ces entités sont totalement différentes. Si les composants sont reconnus et utilisés comme des entités réutilisables, les connecteurs sont souvent considérés comme des entités que l'on doit créer pour réaliser l'assemblage voulu à partir d'un stock de composants. De cette façon, chaque utilisateur crée ses connecteurs avec pour seul objectif qu'ils puissent gérer les interactions entre les composants. Cette considération pour les connecteurs fait que l'on s'intéresse plus à leurs rôles qu'à ce qu'ils sont.

Les définitions. Cette différence dans l'utilisation et la considération est particulièrement visible dans les définitions des connecteurs. En effet, ces définitions sont peu nombreuses et sont toutes convergentes pour définir le rôle des connecteurs. La définition la plus couramment utilisée [114] se concentre uniquement sur le rôle des connecteurs : « **les connecteurs gèrent les interactions entre les composants ; c'est-à-dire qu'ils définissent les règles qui gouvernent les interactions entre les composants et spécifient les mécanismes auxiliaires nécessaires** ». Elle ne décrit pas ce qu'est un connecteur mais ce que fait un connecteur. A ce titre, cette définition est totalement différente de celles proposées pour les composants qui décrivent principalement le composant. Par contre, on peut la rapprocher de la description des composants comme une encapsulation d'une fonctionnalité.

La taxonomie. La différence de considération apparait aussi dans la littérature traitant des connecteurs. L'exemple le plus flagrant est la taxonomie proposée par Medvidovic [90]. Cet article est l'un des plus couramment cité concernant les connecteurs. Pourtant, il ne contient qu'une taxonomie, c'est-à-dire une méthode de classement, qui repose sur le service proposé par le connecteur et les techniques utilisées pour fournir ce service (*cf.* Section 1.1.2.1). Elle ne traite pas de la structure des connecteurs, et fournit uniquement une description fonctionnelle.

Malgré sa focalisation sur le rôle du connecteur, l'étude de cette taxonomie, et en particulier des différents services et types de connecteurs, met en évidence un certain nombre de caractéristiques que les architectes associent, de manière implicite, à la notion de connecteur au delà justement du simple rôle qu'il doit remplir. Ce fait se retrouve en filigrane dans toute la littérature consacrée aux connecteurs.

Cependant, ces caractéristiques restent floues et sont difficiles à définir formellement. Cependant, elles se rapprochent de celles énoncées clairement pour les composants. Ainsi, on retrouve, pour les connecteurs, des caractéristiques de généricité qui rappellent celles de composabilité et de spécificité des composants.

Pour pouvoir définir formellement les caractéristiques sémantiques des connecteurs, nous proposons d'étudier les caractéristiques sémantiques des éléments architecturaux. Nous déduisons ensuite les caractéristiques sémantiques des connecteurs en sous-typant ces caractéristiques en utilisant les informations disponibles dans la littérature sur les connecteurs.

Les caractéristiques sémantiques des éléments architecturaux. L'utilisation de l'architecture vise avant tout trois objectifs : l'identification des fonctionnalités, la séparation des aspects de communication et la réutilisation. Ces trois objectifs imposent aux éléments architecturaux certaines caractéristiques pour permettre à l'architecture de les atteindre. Ainsi, la réutilisation impose aux éléments architecturaux d'être génériques et autonomes. Les éléments sont génériques s'ils peuvent être utilisés dans des architectures et environnements différents. L'autonomie des éléments permet de les réutiliser individuellement ou au moins en réutilisant seulement un sous-groupe des éléments de l'architecture. L'identification des fonctionnalités et la séparation des aspects de communication imposent que les éléments architecturaux soient spécifiques c'est-à-dire qu'ils encapsulent une seule fonctionnalité.

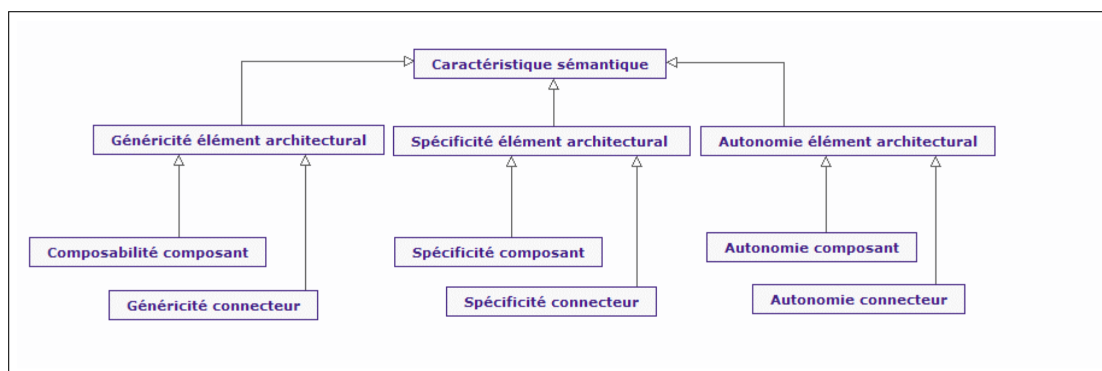


Figure 4.3 – Les caractéristiques sémantiques des éléments architecturaux

Le sous-typage des ces trois caractéristiques pour les composants est évident (*cf.* Figure 4.3). L'autonomie et la spécificité sont les sous-types respectifs de l'autonomie et la spécificité des éléments architecturaux. La composabilité est le sous-type de la généricité. Ces deux dernières permettent, en effet, de mesurer la capacité de réutilisation des éléments architecturaux.

Les caractéristiques sémantiques des connecteurs. A partir des caractéristiques sémantiques des éléments architecturaux et des travaux tels que la taxonomie des connecteurs, nous proposons un ensemble de caractéristiques sémantiques des connecteurs. Ces caractéristiques sont, comme pour les composants, des sous-types des caractéristiques des éléments architecturaux. Nous retrouvons donc trois caractéristiques sémantiques pour les connecteurs simples :

- **la spécificité** : la taxonomie des connecteurs [90] montre que les connecteurs peuvent remplir quatre services différentes. Elle montre également que les services peuvent être combinés au sein d'un connecteur en utilisant certains types spécifiques. Cependant, dans le cas des connecteurs simples, les combinaisons possibles sont très limitées et n'impliquent jamais plus de deux services. Par conséquent, un connecteur doit être spécifique, c'est-à-dire ne pas proposer plusieurs services mais concentrer ses capacités sur la réalisation d'un ou deux services qui reposent sur un même type ;
- **la généricité** : en tant qu'élément architectural, le connecteur doit être réutilisable. Pour cela, le connecteur doit être le plus générique possible, c'est-à-dire qu'il doit être facilement composable avec différents composants ;
- **l'autonomie** : comme tout élément architectural, le connecteur doit, pour être réutilisable, être autonome. Cependant, cette autonomie est différente de celle des composants puisque, de manière évidente, un connecteur sans rôle, c'est-à-dire sans lien avec l'extérieur, n'a pas de sens. L'autonomie d'un connecteur est en faite relative aux liens entre ses rôles. Ainsi, les rôles du connecteur doivent tous être impliqués dans l'interaction régie par le connecteur. La présence d'un rôle, et donc d'un composant, qui ne serait pas directement impliqué dans l'interaction des autres rôles rend le connecteur non-autonome.

4.2.2 Des caractéristiques sémantiques aux propriétés architecturales

Suivant notre méta-modèle de mesure, les caractéristiques sémantiques constituent les sous-caractéristiques de la caractéristique validité sémantique. Ces sous-caractéristiques doivent maintenant être raffinées en un ensemble de propriétés mesurables sur l'architecture et ses éléments architecturaux.

Nous présentons d'abord les propriétés des éléments architecturaux que nous prenons en considération. Ensuite, nous présentons les liens que nous avons identifiés entre les caractéristiques sémantiques et ces propriétés mesurables.

4.2.2.1 Les propriétés des composants

Les propriétés mesurables du composant peuvent être réparties en deux groupes : le premier comporte les propriétés portant sur des éléments internes du composant ; le deuxième porte sur les éléments de son enveloppe.

Les éléments internes des composants. La première propriété est la cohésion interne du composant, c'est-à-dire la cohésion du code du composant. Cette cohésion reflète le degré de collaboration qu'entretiennent les éléments qui constituent le code du composant. La deuxième propriété est le couplage interne du composant. Ce couplage représente le degré de dépendance des éléments internes du code entre eux.

Ces deux propriétés montrent le niveau de proximité entre les éléments internes du composant. Ainsi, elles permettent de déterminer si les éléments internes du composant collaborent pour accomplir une fonctionnalité ou s'il existe des éléments indépendants fournissant différents services.

Les éléments de l'enveloppe des composants. Les éléments de l'enveloppe du composant sont les interfaces, fournies et requises, et les services qui composent ces interfaces.

Interfaces fournies. La propriété des interfaces fournies la plus évidente est leur nombre. Elle reflète le nombre de fonctionnalités proposées par le composant.

Cependant, la cohésion des interfaces fournies permet une mesure plus précise du nombre de fonctionnalités proposées par le composant. En effet, si les interfaces sont fortement cohésives, elles collaborent et donc elles proposent sans doute des fonctionnalités complémentaires ou différents éléments d'une même fonctionnalité. Cette affirmation s'applique de même à l'intérieur des interfaces, sur les services. La cohésion des services d'une interface représentent ainsi le niveau de collaboration des services proposés par l'interface.

Interfaces requises. Comme nous l'avons discuté lors de la présentation du modèle COA, on distingue, parmi les interfaces requises, les interfaces qui représentent une relation de composition verticale, appelée interfaces requises verticales, et celles qui représentent une relation de composition horizontale, appelée interfaces requises horizontales.

Ces deux types de relations permettent de définir différentes propriétés des composants basées sur la mesure de la force de ces relations. Pour cela, nous proposons d'utiliser le nombre d'interfaces requises impliquées dans chaque type de relations comme deux propriétés mesurables de notre modèle. Ces deux nombres permettent respectivement de mesurer le couplage du composant avec les autres composants au sein de l'architecture, et le couplage du composant avec le composant composite qui l'englobe.

4.2.2.2 Les liens entre les caractéristiques et les propriétés des composants

Les propriétés mesurables des composants que nous avons présentées, permettent de raffiner les caractéristiques sémantiques des composants. Dans la suite, nous présentons les liens entre chaque caractéristique et les propriétés mesurables.

L'autonomie. L'autonomie du composant s'évalue différemment selon que le composant est un sous-composant ou non. Ainsi, un composant simple ou composite est autonome s'il ne possède pas d'interface requise. Son autonomie décroit lorsque le nombre d'interfaces requises augmente. Par conséquent, la propriété **nombre d'interfaces requises** raffine la sous-caractéristique autonomie.

Par contre, un sous-composant, c'est-à-dire un composant qui possède des relations de composition verticale, doit tenir compte de ces relations et des contraintes supplémentaires qui sont imposées par ces relations. Ainsi une interface requise qui est satisfaite par un composant externe au travers d'une relation de composition horizontale entraine un couplage plus faible que la même interface satisfaite par un composite à travers une relation verticale [14]. Par conséquent, les interfaces relevant d'une relation verticale ont un impact plus nocif sur l'autonomie que celle relevant d'une relation horizontale. En effet, un sous-composant est difficilement séparable du composant composite auquel il appartient. Par conséquent, l'autonomie du sous-composant est également raffinée par la **nombre d'interfaces requises verticales**.

La composabilité. Un composant est composé à travers ses interfaces fournies et requises. Cependant, un composant sera plus facile à utiliser avec d'autres si les services, dans chaque interface, sont cohésifs. Ainsi la propriété **moyenne de la cohésion des services par interface** permet donc de raffiner la sous-caractéristique composabilité.

Le cas des relations de composition verticale est, la encore, particulier. En effet, ces relations nuisent à la composition puisqu'elles imposent un composant composite pour contenir le sous-composant. Un composant sera d'autant plus composable que le nombre de relations de compositions verticales sera

faible. La propriété **nombre d'interfaces requises verticales** raffine donc également la sous-caractéristique composabilité.

La spécificité. L'évaluation du nombre de fonctionnalités est basée sur les affirmations suivantes. Premièrement, un composant qui propose plusieurs interfaces, doit fournir plusieurs fonctionnalités. Ainsi plus grand est le nombre d'interfaces fournies, plus grand est le nombre de fonctionnalités.

Deuxièmement, si les interfaces fournies (*resp.* les services dans chaque interface fournie) sont cohésives (*i.e.* partagent des ressources), elles offrent probablement des fonctionnalités proches.

Troisièmement, si le code du composant est très couplé (*resp.* cohésif), les différentes parties du code du composant s'entraident (*resp.* utilisent des ressources communes). Par conséquent, elles travaillent probablement ensemble pour fournir un petit nombre de fonctionnalités.

De ces affirmations nous raffinons la sous-caractéristique spécificité par les propriétés suivantes : **nombre d'interfaces fournies**, **moyenne de la cohésion des services par interface fournie**, **cohésion des interfaces fournies** et **cohésion et couplage internes du composant**.

	Autonomie	Composabilité	Spécificité
Nombre d'interfaces requises	$\uparrow \searrow \downarrow$	$\emptyset$	$\emptyset$
Nombre d'interfaces requises verticales	$\uparrow \searrow \downarrow$	$\uparrow \searrow \downarrow$	$\emptyset$
Cohésion des services par interface fournie	$\emptyset$	$\uparrow \searrow \uparrow$	$\uparrow \searrow \uparrow$
Nombre d'interfaces fournies	$\emptyset$	$\emptyset$	$\uparrow \searrow \downarrow$
Cohésion des interfaces fournies	$\emptyset$	$\emptyset$	$\uparrow \searrow \uparrow$
Cohésion interne du composant	$\emptyset$	$\emptyset$	$\uparrow \searrow \uparrow$
Couplage interne du composant	$\emptyset$	$\emptyset$	$\uparrow \searrow \uparrow$

Table 4.1 – Relation entre les sous-caractéristiques sémantiques et les propriétés des composants

Les liens que nous avons présentés entre les sous-caractéristiques et les propriétés des composants sont résumés dans le tableau 4.1. Ce tableau précise les sens de variations relatifs des propriétés et des sous-caractéristiques.

4.2.2.3 Les propriétés des connecteurs

Comme pour les composants, nous distinguons les propriétés des connecteurs en fonction de la position de l'élément considéré : interne ou enveloppe.

Les éléments internes des connecteurs. Les propriétés des éléments internes des connecteurs sont les mêmes que pour les composants. Nous utilisons donc la cohésion et le couplage des éléments du code du connecteur. Ces propriétés nous permettent d'évaluer les liens entre les éléments du code et ainsi, de la même façon que pour les composants, d'évaluer le nombre de services proposés par le connecteur.

Ces propriétés, similaires à celles des composants, ne sont pas les seules propriétés internes du connecteur. En effet, la connexité de la glue est une autre propriété essentielle du connecteur. Elle permet de détecter les liens entre les différents rôles et de vérifier que chaque rôle a un impact sur l'interaction gérée par le connecteur.

Les éléments de l'enveloppe des connecteurs. Les éléments de l'enveloppe du connecteur sont les rôles et les interfaces. La cohésion et le couplage entre les rôles ou entre les interfaces dans chaque rôle

donne une indication sur la force des liens qui unissent les composants reliés par le connecteur. L'autre propriété est le nombre de rôles du connecteur qui précise le nombre de composants qu'il interconnecte.

4.2.2.4 Les liens entre les caractéristiques et les propriétés des connecteurs

Les propriétés mesurables des connecteurs que nous avons présentées, permettent de raffiner les caractéristiques sémantiques des connecteurs. Dans la suite, nous présentons les liens entre chaque caractéristique et les propriétés mesurables.

La spécificité. La spécificité d'un connecteur se mesure en fonction du nombre de services proposés et du nombre de types utilisés. La mesure de ces deux paramètres peut être rapprochée de celle du nombre de fonctionnalités dans les composants. Cependant, alors que dans le composant les interfaces et leurs cohésions donnent des indications sur le nombre de fonctionnalités, les rôles du connecteur ne permettent pas de juger du nombre de services proposés mais plutôt du type de service.

Par contre, les propriétés de cohésion et de couplage interne, qui permettent de mesurer la spécificité d'un composant, se transposent facilement au cas de la spécificité des connecteurs. Ainsi, l'évaluation du nombre de services et du nombre de types utilisés dans un connecteur repose sur les **propriétés internes de cohésion et de couplage** du connecteur. En effet, la cohésion et le couplage mettent en évidence la collaboration et les dépendances entre les éléments internes du connecteur. Par conséquent, si les éléments du code du connecteur sont fortement couplés et cohésifs, ils collaborent et sont fortement dépendants. Ils doivent donc participer à un seul service ou reposer sur un seul type.

La généricité. Un connecteur est générique s'il peut être facilement réutilisé. Cette facilité de réutilisation repose, bien sûr sur de nombreux facteurs qui sont, pour certains, difficiles à mesurer. Cependant, certains de ces facteurs nous permettent de raffiner la sous-caractéristique généricité. Le premier facteur est lié au fait que, de manière évidente, un connecteur qui relie de nombreux composants sera plus difficilement réutilisable qu'un autre qui ne relie que deux composants. En effet, il est plus probable de retrouver une autre architecture qui possède deux composants similaires que quatre. Par conséquent, la propriété **nombre de rôles** raffine la sous-caractéristique généricité.

Un autre facteur essentiel est le lien entre les composants reliés par le connecteur. Ces composants peuvent être plus ou moins étroitement dépendants et en collaboration entre-eux. Or, si ces composants sont étroitement liés, il est plus probable qu'ils soient réutilisés ensemble dans une autre application et donc que l'on ait à nouveau besoin du connecteur. Ainsi, un connecteur a d'autant plus de chance d'être réutilisé que les composants qu'il connecte sont couplés et cohésifs. Afin de mesurer cette cohésion et ce couplage des composants à partir des propriétés du connecteur, nous proposons d'utiliser les propriétés de **cohésion et couplage entre les rôles** pour raffiner la sous-caractéristique généricité.

Le dernier facteur que nous prenons en compte, est la qualité de la définition du rôle du connecteur. Ce rôle, qui est défini par le connecteur, doit être remplie par un composant de l'architecture pour que le connecteur soit utilisable. L'interface des composants pouvant remplir le rôle doit donc être valide sémantiquement, c'est-à-dire qu'elle doit représenter un ensemble de services fortement cohésifs et couplés. Par conséquent, chaque rôle du connecteur doit être constitué d'un ensemble d'interfaces qui soit fortement cohésif et couplé. Ainsi, les propriétés de **couplage et de cohésion du rôle** raffinent également la sous-caractéristique généricité.

L'autonomie. La sous-caractéristique autonomie est particulière. Contrairement aux autres sous-caractéristiques, elle ne peut prendre que deux valeurs : le connecteur est autonome ou il ne l'est pas. En

effet, s'il possède un rôle qui n'est pas impliqué dans l'interaction qu'il encapsule, le connecteur n'est pas autonome, sinon il l'est.

Le raffinement de cette sous-caractéristique repose donc sur la mesure de l'implication de chaque rôle dans l'interaction. Cette implication peut être établie en déterminant si les rôles forment un ensemble connexe. Ceci peut être déterminé en évaluant la connexité de la glu qui est la partie du connecteur qui relie les rôles. Par conséquent, nous proposons de raffiner la sous-caractéristique par la propriété **connexité de la glu**. Cette propriété peut prendre deux valeurs qui sont 1 ou 0 selon que la glu forme un ensemble connexe ou non.

	Spécificité	Généricité	Autonomie
Cohésion interne du connecteur	$\uparrow\searrow\uparrow$	$\emptyset$	$\emptyset$
Couplage interne du connecteur	$\uparrow\searrow\uparrow$	$\emptyset$	$\emptyset$
Nombre de rôles	$\emptyset$	$\uparrow\searrow\downarrow$	$\emptyset$
Cohésion entre les rôles	$\emptyset$	$\uparrow\searrow\uparrow$	$\emptyset$
Couplage entre les rôles	$\emptyset$	$\uparrow\searrow\uparrow$	$\emptyset$
Couplage du rôle	$\emptyset$	$\uparrow\searrow\uparrow$	$\emptyset$
Cohésion du rôle	$\emptyset$	$\uparrow\searrow\uparrow$	$\emptyset$
Connexité de la glu	$\emptyset$	$\emptyset$	$\equiv$

Table 4.2 – Relation entre les sous-caractéristiques sémantiques et les propriétés des connecteurs

Les liens que nous avons présentés entre les sous-caractéristiques et les propriétés des connecteurs sont résumés dans le tableau 4.2. Ce tableau précise aussi les sens de variations relatifs des propriétés et des sous-caractéristiques.

4.2.3 Des propriétés architecturales aux propriétés des entités COA

Suivant notre méta-modèle de mesure (*cf.* Figure 4.2), les propriétés mesurables sur les éléments architecturaux doivent être reliées aux propriétés mesurables des entités du modèle COA. Nous étudions donc successivement les liens des propriétés internes et de l'enveloppe avec les propriétés des entités COA.

4.2.3.1 Liens entre les propriétés internes des éléments architecturaux et des entités COA

Nous présentons d'abord les liens entre les propriétés internes des composants et les propriétés des entités COA. Nous étudions ensuite les liens entre les propriétés internes des connecteurs et les propriétés des entités COA.

Les propriétés internes des composants. D'après le modèle COA (*cf.* Figure 3.9), le composant est relié aux entités objets à travers l'entité COA contour du composant. Les propriétés internes des composants sont donc liées aux propriétés des éléments objets qui constituent le contour du composant, c'est-à-dire les classes objets contenues dans la membrane et le noyau du contour du composant.

La cohésion et le couplage interne du composant peuvent être respectivement mesurés par les propriétés des classes du contour du composant : **cohésion des classes du contour** et **couplage des classes du contour**.

Les propriétés internes des connecteurs. D'après le modèle COA, le connecteur est relié aux entités objets à travers l'entité COA contour du connecteur. Les propriétés internes des connecteurs sont donc liées aux propriétés des éléments objets qui constituent la partie interne du contour du connecteur, c'est-à-dire les méthodes et les attributs de classes objets contenues dans le noyau du contour du connecteur.

La cohésion et le couplage interne du connecteur peuvent être respectivement mesurés par les propriétés des méthodes du contour du connecteur **cohésion des méthodes du contour du connecteur** et **couplage des méthodes du contour du connecteur**. De façon similaire, la connexité de la glue est mesurée à travers la propriété **connexité des méthodes du contour du connecteur**

4.2.3.2 Liens entre les propriétés de l'enveloppe des éléments architecturaux et des entités COA

Nous présentons d'abord les liens entre les propriétés de l'enveloppe des composants et les propriétés des entités COA. Nous étudions ensuite les liens entre les propriétés de l'enveloppe des connecteurs et les propriétés des entités COA.

Les propriétés de l'enveloppe des composants. L'enveloppe des composants est constituée des interfaces fournies et requises. D'après le modèle COA, ces éléments sont reliés aux entités objets à travers les entités COA contours des interfaces fournies et des interfaces requises. Les propriétés des interfaces des composants sont donc liées aux propriétés des éléments objets qui constituent les contours des interfaces. Ces éléments objets sont des noms de méthodes des classes objets. Les propriétés sont donc mesurées sur les méthodes correspondantes à ces noms qui peuvent être contenues dans différents contours de composant.

Les interfaces fournies. Le nombre d'interfaces fournies d'un composant se mesure sans ambiguïté dans le modèle COA en utilisant les correspondances composant/contour de composant et interface fournie/ contour d'interface fournie. Ainsi, la propriété du nombre d'interfaces fournies d'un composant est, de manière évidente, liée à la propriété **nombre de contours d'interfaces fournies associés au contour du composant**.

La cohésion des interfaces fournies d'un composant se mesure facilement en utilisant la même correspondance que pour le nombre d'interfaces. On mesure la cohésion des interfaces fournies d'un composant A en mesurant la cohésion des méthodes associées aux noms contenus dans l'ensemble des contours d'interfaces fournies associés au contour du composant A. Ainsi, la propriété cohésion des interfaces fournies est liée à la propriété **cohésion des contours d'interfaces fournies**.

La moyenne de la cohésion des services par interface fournie ne peut se mesurer directement à partir des correspondance proposées dans le modèle COA. Nous devons poser l'hypothèse que chaque service d'une interface fournie correspond à la méthode représentée par le nom contenu dans le contour de l'interface fournie. Cette hypothèse est fidèle à la définition des services des interfaces. La cohésion des services par interface fournie se mesure alors en utilisant la cohésion entre les méthodes associées au contour de l'interface. Ainsi, la propriété moyenne de la cohésion des services par interface fournie est liée à la propriété **moyenne de la cohésion de chaque contour d'interfaces fournies**.

Les interfaces requises. Les interfaces requises peuvent, comme nous l'avons constaté, être associées à deux types différents de relation de composition : verticale, c'est-à-dire composite/sous-composant ; horizontale, c'est-à-dire composant/composant. Cependant quelque soit le type de l'interface requise la propriété mesurée est la même : le nombre d'interfaces requises de chaque type.

Le nombre d'interfaces requises, horizontales ou verticales, d'un composant se mesure facilement dans le modèle COA en utilisant les correspondances composant/ contour de composant et interface

requise/ contour d'interface requise. Ainsi les propriétés du nombre d'interfaces requises verticales et horizontales d'un composant sont, de manière évidente, liée aux propriétés **nombre de contours d'interfaces requises verticales et horizontales associés au contour du composant**.

Les propriétés de l'enveloppe des connecteurs. L'enveloppe des connecteurs est constituée des rôles. D'après le modèle COA, ces éléments sont reliés aux entités objets à travers les entités COA façades. Les propriétés des rôles des connecteurs sont donc liées aux propriétés des éléments objets qui constituent les façades des contours des connecteurs. Ces éléments objets sont des noms de méthodes des classes objets. Les propriétés sont donc mesurées sur les méthodes correspondantes à ces noms.

Le nombre de rôles se mesure facilement dans le modèle COA en utilisant les correspondances connecteur/contour de connecteur et rôle/façade. Ainsi, la propriété du nombre de rôle d'un connecteur est, de manière évidente, liée à la propriété **nombre de façades associées au contour de connecteur**.

La cohésion et le couplage entre les rôles se mesurent facilement en utilisant la même correspondance que pour le nombre de rôles. On mesure la cohésion et le couplage des rôles d'un connecteur A en mesurant la cohésion et le couplage des méthodes associées aux noms contenus dans l'ensemble des façades associées au contour du connecteur A. Ainsi les propriétés cohésion et couplage des rôles sont liées aux propriétés **cohésion et couplage des façades du contour de connecteur**.

La cohésion et le couplage du rôle se mesure de manière similaire à la cohésion et le couplage des rôles. La différence porte sur les méthodes prisent en compte. En effet, dans le cas de la cohésion et le couplage du rôle, la mesure n'est faite que sur les méthodes associées aux noms contenus dans la façade associée au rôle. Ainsi, les propriétés cohésion et couplage du rôle sont liées aux propriétés **cohésion et couplage de la façade du contour de connecteur**.

4.2.4 Des propriétés du modèle COA au modèle objet

Nous avons établi dans les sections précédentes, les premiers niveaux de notre modèle de mesure de la validité sémantique de l'architecture (*cf.* Figures 4.4 et 4.5). Nous définissons maintenant le dernier niveau de ce modèle, c'est-à-dire les liens entre les propriétés mesurables des entités COA et le modèle objet et ses métriques.

Nous établissons d'abord la correspondance entre la propriété de composition des contours interfaces requises et les propriétés des relations entre les éléments objets. Cette étude nous permet de lier ces deux ensembles de relations et donc de déterminer une propriété essentielle de notre modèle : le type de chaque contour d'interface requise.

Ensuite, nous étudions les métriques objets nécessaires à l'instanciation de notre modèle de mesure. Ces métriques portent sur le couplage et la cohésion qui sont deux propriétés qui ont déjà été étudiées et pour lesquelles il existe un grand éventail de métriques proposées. Nous commençons donc, pour le couplage puis la cohésion, par définir la propriété que l'on veut mesurer et les contraintes qui s'appliquent sur les métriques dans notre approche. Nous présentons ensuite les mesures que nous avons choisies pour compléter notre modèle de mesure.

4.2.4.1 Liens entre les interfaces verticales et les entités COA

Nous étudions les relations de composition des composants et les relations entre entités objets afin de déterminer le type, horizontale ou verticale, des contours d'interfaces requises pour pouvoir utiliser les autres liens identifiés entre les propriétés des éléments architecturaux et COA. Pour cela, nous étudions d'abord les propriétés des relations de compositions entre composants. Ensuite, nous montrons les liens entre ces relations et les relations entre classes.

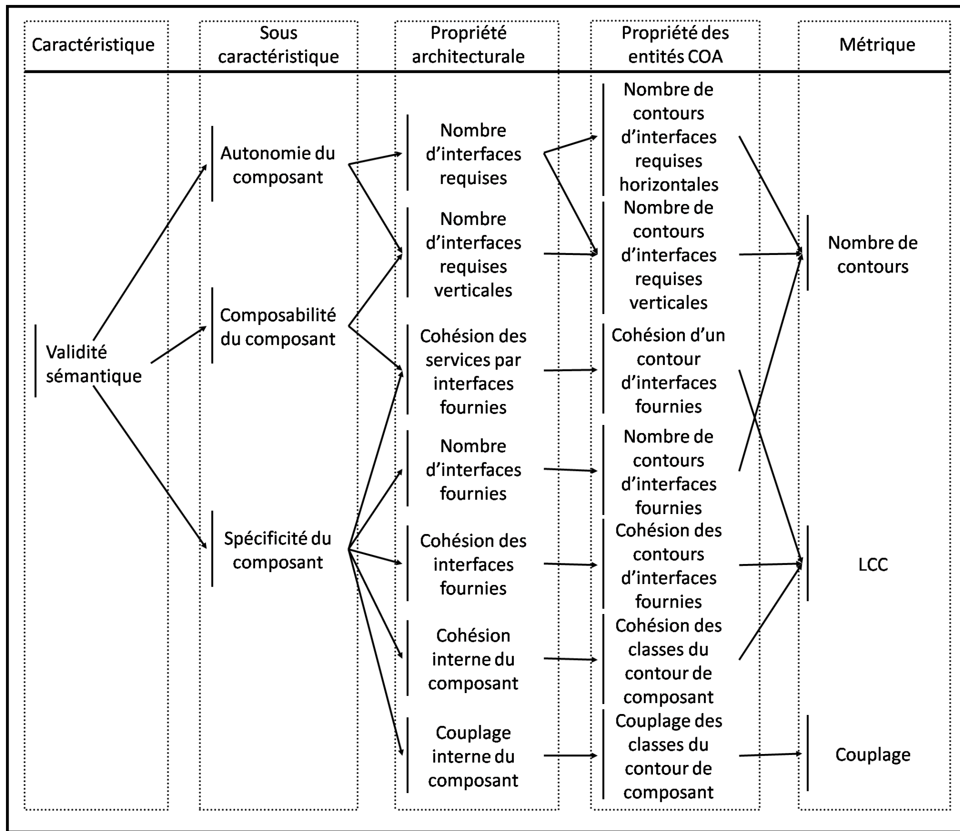


Figure 4.4 – Notre modèle de mesure de la validité sémantique de l'architecture : les sous-caractéristiques des composants

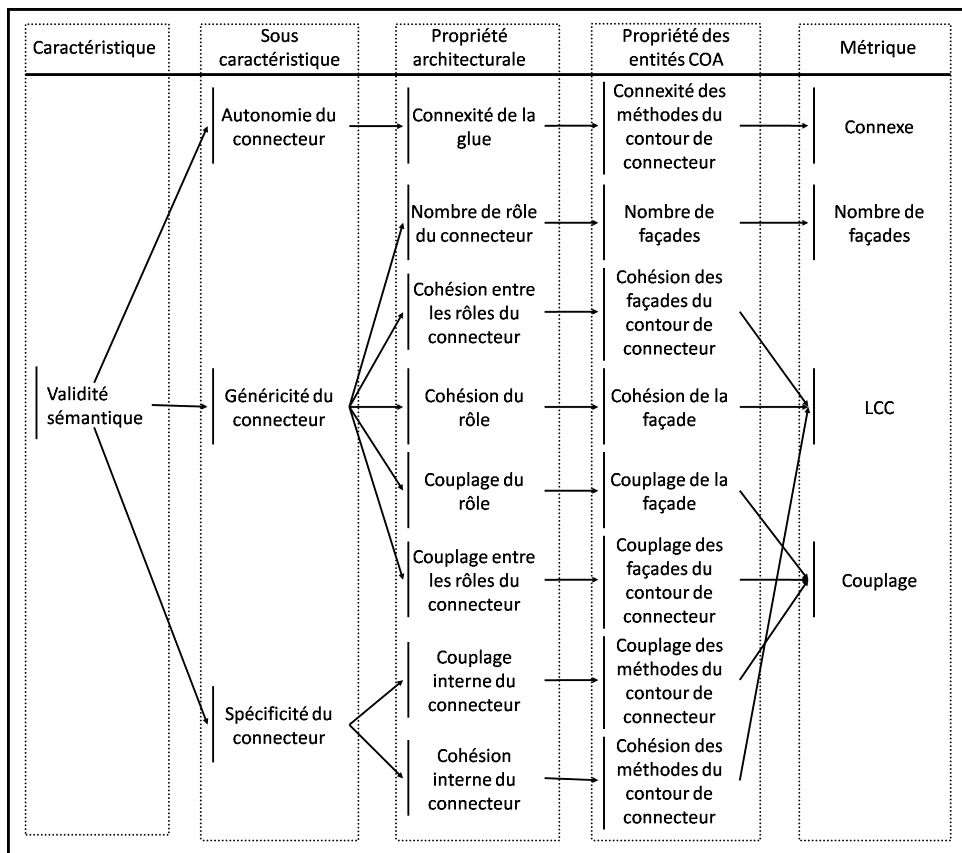


Figure 4.5 – Notre modèle de mesure de la validité sémantique de l'architecture : les sous-caractéristiques des connecteurs

Propriétés des relations de composition. La dernière version de UML permet de modéliser les composants [99]. Pour cela, le méta-modèle UML comporte différentes entités représentant les composants et leurs relations. Cependant, les relations de composition sont peu traitées dans UML. En effet, le méta-modèle ne stipule pas d'entités particulières pour représenter les relations entre composants. Les seules relations disponibles sont donc la composition et l'agrégation objet qui sont insuffisantes pour représenter toutes les facettes de la composition entre composants [14]. Le modèle UML est donc, a priori, insuffisant pour nous permettre de caractériser les relations de composition entre composants et en particulier les relations de compositions verticales.

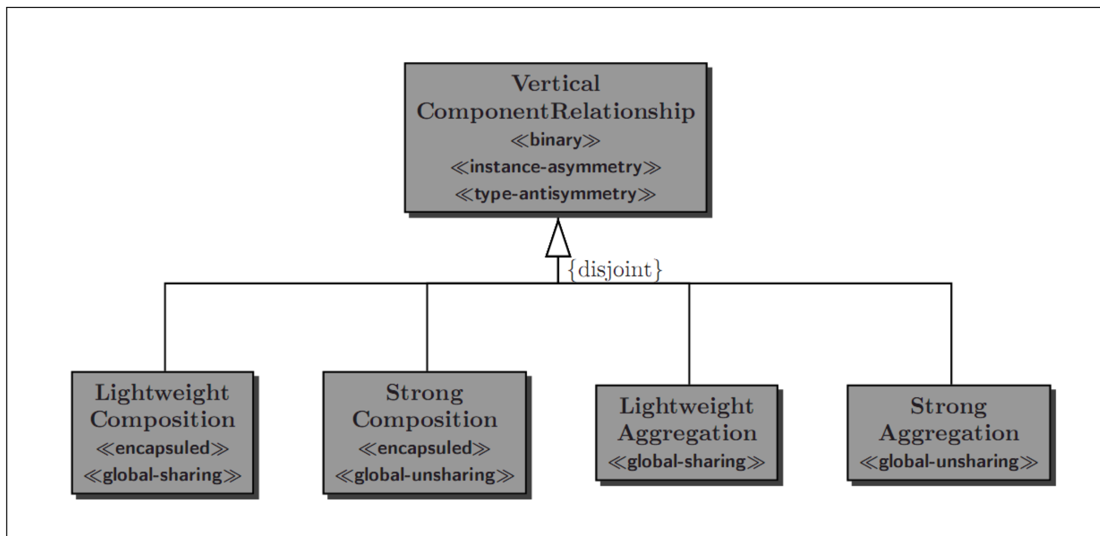


Figure 4.6 – Les spécialisations de la méta-classe relation de composition verticale

Dans sa thèse [14], BELLOIR propose une caractérisation des relations de compositions verticales. Cette caractérisation repose sur les relations tout/partie et consiste en une extension du méta-modèle UML. BELLOIR ajoute au méta-modèle UML la relation de composant (*ComponentRelationShip*) ainsi que la relation de composition verticale (*VerticalComponentRelationShip*) et horizontale (*Horizontal-ComponentRelationShip*). Il propose ensuite quatre spécialisations de la relation de composition verticale (*cf.* Figure 4.6).

Les deux premières spécialisations sont des compositions. Elles imposent l'encapsulation des sous-composants dans le composite. La composition forte correspond à la composition dans UML alors que la composition légère impose, en prime, la partageabilité globale des sous-composants, c'est-à-dire avec des composants externes au composite. Les deux autres spécialisations sont des agrégations. Elles imposent la non-encapsulation. L'agrégation légère correspond à l'agrégation définie dans UML alors que l'agrégation forte impose en plus la non-partageabilité.

Chacune de ces relations à un certain impact sur l'aspect dynamique du système. En effet, les différentes relations ne correspondent pas à tous les scénarios de cycle de vie des sous-composants et du composite (*cf.* Figure 4.7). Les relations de composition forte et légère sont compatibles uniquement avec les scénarios (1), (2), (3) et (4), puisque elles imposent que le cycle de vie du sous-composant soit inclu dans celui du composite. Au contraire, les relations d'agrégation sont compatibles avec tous les cas de figure.

Les propriétés des relations proposées par BELLOIR sont résumées dans le tableau 4.3. Parmi ces propriétés, la relation de composition verticale que nous avons définie impose l'encapsulation et une

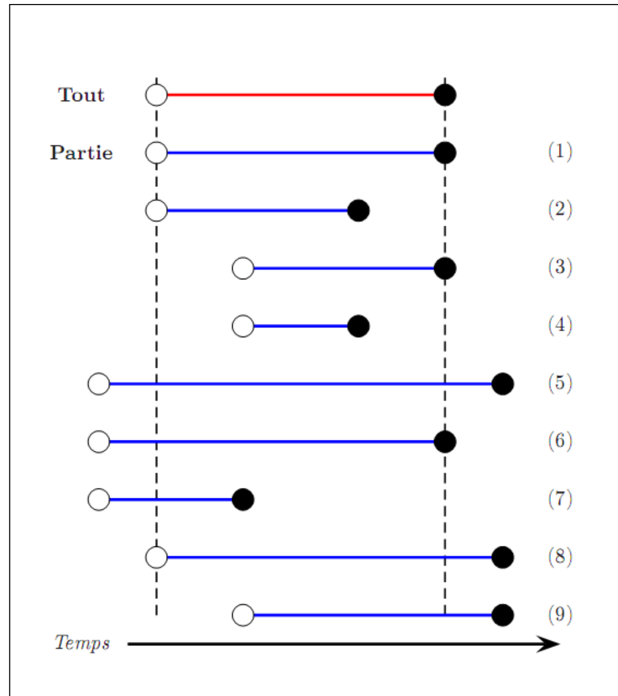


Figure 4.7 – Les neufs cas de cycles de vie

parteageabilité locale, c'est-à-dire que les sous-composants peuvent interagir directement entre eux mais pas avec des composants externes au composite. Au final, selon ces propriétés, la relation de composition forte décrit le mieux notre relation de composition verticale. Or, cette relation correspond à la définition de la composition utilisée dans UML. Par conséquent, nous cherchons à identifier les liens entre la relation de composition UML entre les composants et les relations entre les classes objets.

	Composition forte	Composition légère	Agrégation forte	Agrégation légère
Encapsulation	OUI	OUI	NON	NON
Partageabilité	NON	OUI	NON	OUI
Cycle de vie	4 cas	4 cas	9 cas	9 cas

Table 4.3 – Les différents types de relations de composition verticale

Liens avec les relations entre classes. Afin d'identifier les liens unissant la relation de composition UML entre les composants et les relations entre classes, nous avons étudié les caractéristiques de la relation de composition entre les classes en UML. Cette relation a, bien sûr, la même définition que celle pour les composants cependant plusieurs travaux, dont ceux de BARBIER [9], permettent de préciser la sémantique associée à cette relation et donc de la comparer avec la relation de composition verticale entre composants.

En UML [99], la composition est définie comme une forme d'agrégation qui impose que l'instance partie soit incluse dans au plus un composite à la fois, et que l'objet composite soit responsable de la création et de la destruction des parties. L'agrégation est définie comme une association particulière qui spécifie une relation tout-partie entre un agrégat et une partie. Cette définition de la composition met

clairement en lumière des contraintes sur les cycles de vie du composite et des parties : le cycle de vie des parties est inclu dans celui du composite. Ce cas de figure correspond aux propriétés de la composition forte selon BELLOIR que l'on cherche à relier.

BARBIER [9] précise cette définition en ajoutant les propriétés d'encapsulation forte ou faible. L'encapsulation forte impose que la partie n'est aucun autre lien en dehors de celui qu'elle entretient avec le composite. Par contre, l'encapsulation faible autorise les parties d'un composite à collaborer. On retrouve, ainsi, dans la composition UML les mêmes propriétés que nous avions entre les composants dans la relation de composition forte.

Identification des types de contours d'interfaces requises Nous avons montré à travers l'étude des relations de composition entre les composants et les classes, les liens forts qui existent entre une relation de composition verticale entre composants et une relation de composition entre classes. Il apparait que si une classe composite se trouve dans un composant, ses classes parties doivent appartenir au même composant et, par conséquent, le composant qui les englobe aussi. La relation de composition entre deux classes englobe ainsi les autres relations qui peuvent exister entre deux autres classes de ces composants et impose l'encapsulation et donc la composition verticale des composants.

Nous utilisons ces liens pour identifier, parmi les contours d'interfaces requises ceux qui définissent des relations composant/composite dans le composant correspondant. Ainsi, un contour d'interface requise est un contour d'interface requise verticale si et seulement si il existe dans le contour une méthode qui relie deux classes possédant une relation de composition.

4.2.4.2 Mesure de couplage

Le couplage est la mesure de la dépendance d'un élément. Cette notion est très utilisée dans le génie logiciel et, en particulier, dans le paradigme objet. En effet, ce paradigme vise à découpler les entités du système en utilisant le concept de classes pour encapsuler les fonctionnalités et réduire les dépendances. Ainsi, cette mesure est essentielle pour visualiser la dépendance entre les classes d'un système logiciel et faciliter sa maintenance en localisant, par exemple, les classes qui sont trop dépendantes. Cet intérêt a donné lieu à de nombreux travaux sur des métriques permettant de mesurer le couplage des classes objets [23, 29, 30, 64, 80, 82].

Cependant, les métriques proposées mesurent le couplage d'une classe avec l'extérieur. Dans ces métriques, une classe est fortement couplée si elle est couplée avec de nombreuses classes, peu importe la force de la relation avec chacune de ces classes. Par exemple :

- $NIH - ICP$ [80] mesure le nombre d'appels de méthodes de classes avec lesquelles une classe n'a pas de relation d'héritage pondéré par le nombre de paramètres.
- DAC [82] mesure le nombre d'attributs de la classe qui ont pour type une autre classe.
- $OCMIC$ [23] mesure le nombre de paramètres qui ne sont pas de type a ou primitifs dans les méthodes de la classe a.

Ces métriques permettent de détecter les classes qui sont trop couplées. Mais elles ne permettent pas de déterminer si la classe est dépendante de trop de classes extérieures ou si elle dépend trop fortement d'une seule classe. Or, ce type d'information est nécessaire pour compléter notre modèle de mesure.

Devant les problèmes posés par les métriques existantes dans la littérature, nous proposons une nouvelle métrique pour le couplage. Cette métrique vise à satisfaire les conditions particulières de notre approche. D'abord, notre modèle de mesure nécessite une mesure du couplage qui tienne compte de la

force des relations entre chaque classe. Ainsi, au lieu de simplement détecter une classe qui pose problème, cette métrique doit nous permettre d'orienter le processus à partir de cette classe vers celle qui est la source principale du couplage.

Notre approche doit prendre en compte l'ensemble des relations qui existent entre deux classes, c'est-à-dire les appels de méthodes, les attributs et les paramètres. Sachant que les attributs sont déjà pris en compte dans l'identification des contours d'interfaces requises verticales à travers la relation de composition, la métrique doit prendre en compte les appels de méthodes et les paramètres. Les liens d'héritage, par contre, ne doivent pas influencer la possibilité que deux classes appartiennent au même contour. Pour cela, la mesure ne doit pas tenir compte des relations d'héritage, et en particulier, des appels de méthodes vers la classe mère dans le même objet. On n'utilisera pas les métriques telle que IH-ICP [80].

Ensuite, notre approche repose sur un modèle du système qui est indépendant du langage. Les métriques de notre modèle de mesure doivent donc être calculables sur notre modèle du système et en particulier elles doivent être indépendantes du langage utilisé. Ainsi, nous ne pouvons utiliser des métriques telle que IFCAIC [23] qui utilise la notion d'amitié, spécifique au langage C++.

Enfin, notre modèle de mesure ne nécessite pas uniquement de mesurer le couplage entre deux classes. Au contraire, nous devons mesurer le couplage d'un ensemble de classes, d'un ensemble de méthodes et d'attributs ainsi que celui de deux ensembles de méthodes et d'attributs. La mesure du couplage d'un ensemble de classes peut être facilement assimilée à la mesure du couplage des méthodes et attributs contenus dans l'ensemble de classes. Nous obtenons deux cas de figure selon que l'on désire mesurer le couplage des éléments d'un ensemble ou le couplage de deux ensembles d'éléments.

La mesure du couplage des éléments d'un ensemble peut être obtenue en additionnant le couplage de chaque paire d'éléments. Pour le couplage de deux ensembles d'éléments, nous pouvons agir de la même manière en considérant la somme du couplage de chaque couple d'éléments appartenant chacun à un ensemble différent. Par conséquent, nous avons uniquement besoin de définir une métrique mesurant le couplage entre deux méthodes ou attributs.

Pour répondre au besoin de notre modèle de mesure, nous proposons donc une métrique, appelée $couplage\,(x, y)$, qui mesure le couplage entre deux entités x et y qui peuvent être des méthodes ou des attributs. Elle est définie selon la formule suivante, où M désigne l'ensemble des méthodes et A désigne l'ensemble des attributs :

Définition 4.3. $couplage\,(x, y) =$

- nombre d'appels entre x et y, si $x \in M \wedge y \in M$
- 0, si $x \in A \wedge y \in A$
- nombre d'accès à x dans y, si $x \in A \wedge y \in M$

Cette fonction est une adaptation de $NIH - ICP$ [80] qui répond à l'ensemble des contraintes que l'on a fixé. Elle nous permet de définir le couplage d'un ensemble X ou de deux ensembles X et Y selon les formules :

$$couplage\,(X) = \sum_{(x,y) \in X^2} couplage\,(x, y) \tag{4.1}$$

$$couplage\,(X, Y) = \sum_{(x,y) \in X \times Y} couplage\,(x, y) \qquad \text{Si } X \neq Y \qquad = 0 \qquad \text{Sinon} \tag{4.2}$$

4.2.4.3 Mesure de cohésion

La cohésion mesure la force de la collaboration au sein d'un ensemble d'éléments. Comme le couplage c'est une notion très utilisée dans le génie logiciel pour permettre de détecter les défauts dans les systèmes, en particulier objets. Par conséquent, le nombre de métriques proposées pour l'évaluation de cette notion est lui aussi important [18, 29, 64, 80]

Notre modèle de mesure (*cf.* Figures 4.4 et 4.5) utilise, en complément du couplage, une mesure de cette notion de cohésion. Elle doit mesurer la cohésion d'un ensemble de classes ainsi que la cohésion d'une classe. Dans les deux cas, la mesure peut être obtenue en calculant la cohésion de l'ensemble des méthodes et attributs respectivement de l'ensemble des classes ou de la classe.

La métrique « *Loose Class Cohesion* » (LCC), proposée par [18], mesure le pourcentage de paires de méthodes qui sont directement ou indirectement connectées. Deux méthodes sont connectées si elles utilisent directement ou indirectement un attribut commun. Deux méthodes sont indirectement connectées s'il existe une chaine de méthodes connectées qui les relie.

LCC est une mesure couramment utilisée. En effet, elle présente l'avantage d'être facilement calculable et de permettre d'obtenir une mesure de la cohésion sous la forme d'un pourcentage. Ceci permet de simplifier les formules de nos fonctions de mesures. Par conséquent, nous utilisons cette métrique pour calculer la cohésion dans notre modèle de mesure.

4.3 Etude de la qualité architecturale

La qualité d'un système logiciel est l'ensemble des caractéristiques permettant à ce logiciel de remplir les objectifs fixés par les spécifications [44]. Ces caractéristiques ont été beaucoup étudiées et plusieurs auteurs proposent une liste de caractéristiques [20, 26, 44, 51]. La qualité d'une architecture est définie de manière similaire en adaptant la liste des caractéristiques par ajout ou modification de la définition [83].

Ces caractéristiques de qualité sont essentielles dans le cadre d'une approche par exploration de l'extraction d'architectures logicielles. En effet, même si la validité sémantique permet de déterminer quelles solutions ne correspondent pas au concept d'architecture, l'espace de recherche comporte encore un grand nombre de solutions qui n'ont pas le même niveau de qualité. L'évaluation de ces caractéristiques permet au processus d'extraction de sélectionner une solution de qualité et donc de fournir un résultat représentant une bonne architecture par rapport aux caractéristiques choisies.

Dans cette section, nous présentons notre modèle de mesure de la caractéristique « qualité architecturale ». Ce modèle, comme celui de la caractéristique « validité sémantique », repose sur notre méta-modèle de mesure, présenté dans la section 4.1. La section se décompose, comme le méta-modèle, en quatre parties. Nous étudions, d'abord, les caractéristiques de qualité des systèmes logiciels pour déterminer celles qui doivent dirigées notre processus et donc être les sous-caractéristiques de notre modèle. Ensuite, nous exposons successivement les liens entre ces sous-caractéristiques et les propriétés des éléments architecturaux, entre les propriétés des éléments architecturaux et celles des entités COA et enfin entre les propriétés des entités COA et les métriques objets.

4.3.1 Caractéristiques de qualité

Comme nous l'avons vu sur la figure 4.1, la norme ISO-9126 définit un ensemble de caractéristiques de qualité qui sont ensuite raffinées en un ensemble de sous-caractéristiques. La norme propose ensuite à

chaque utilisateur de raffiner ces sous-caractéristiques en un ensemble de propriétés mesurables puis de définir des métriques pour mesurer ces propriétés.

L'étude de ces caractéristiques et sous-caractéristiques, nous permet ensuite de présenter les sous-caractéristiques de notre modèle de mesure de la caractéristique qualité architecturale. Pour cela, nous montrons l'intérêt particulier de certaines des caractéristiques de la norme ISO-9126 ainsi que les difficultés posées par d'autres pour être utilisées dans notre approche.

4.3.1.1 Les caractéristiques de qualité de la norme ISO-9126

La norme ISO-9126 identifie six caractéristiques et vingt-six sous-caractéristiques de qualité d'un logiciel (*cf.* Figure 4.8). Parmi les sous-caractéristiques, la conformité (*compliance*) possède un statut particulier puisqu'elle participe au raffinement de chaque caractéristique. Ceci décrit la nécessité pour chacune des caractéristiques de qualité de respecter un modèle pour pouvoir être comparée, comprise, réutilisée et évaluée. L'architecture est donc conforme pour une caractéristique si un modèle existe pour elle et si l'architecture est conforme à ce modèle.

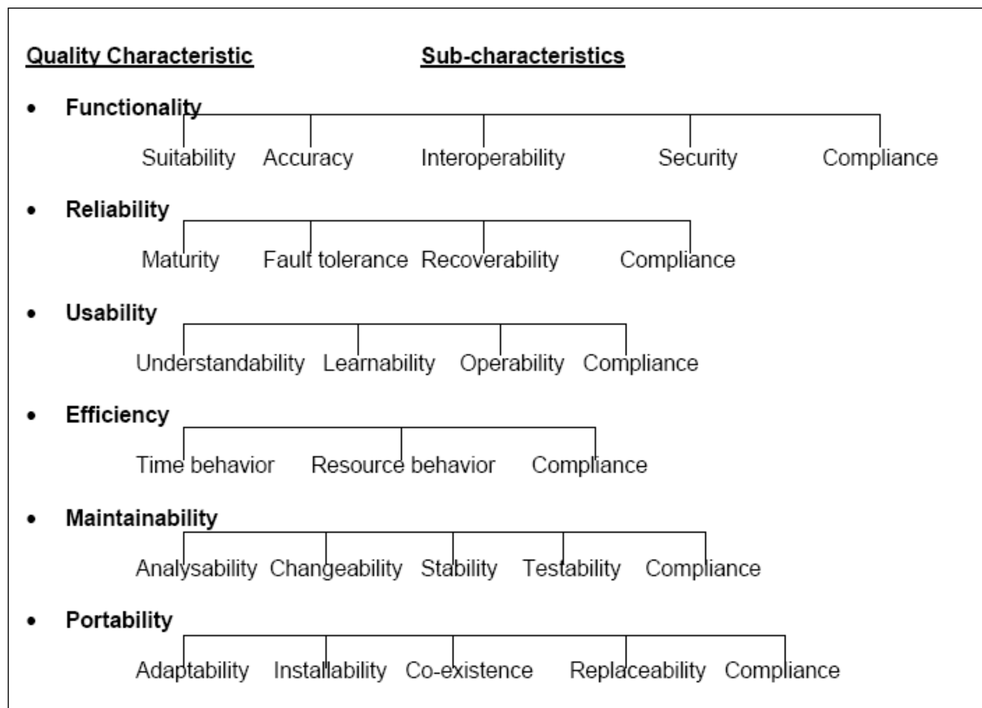


Figure 4.8 – Modèle des caractéristiques d'un produit logiciel dans la norme ISO-9126

Fonctionnalité. Cette caractéristique décrit la capacité d'un produit logiciel à fournir les fonctions respectant les besoins, explicites ou implicites, lorsque le logiciel est utilisé sous certaines conditions. Cette caractéristique est raffinée par quatre sous-caractéristiques :

- **la pertinence** représente le fait d'avoir les fonctions adéquates pour les tâches requises ;
- **la précision** représente le fait de fournir des résultats ou effets correctes avec le niveau de précision nécessaire ;
- **l'interopérabilité** représente la capacité du système à interagir avec un ou plusieurs systèmes spécifiés ;

- **la sécurité** représente la capacité à prévenir les accès non-autorisés aux programmes ou aux données.

Fiabilité. Cette caractéristique décrit la capacité d'un produit logiciel à maintenir son niveau de performances sous les conditions prévues et pour une durée prévue. Cette caractéristique est raffinée par trois sous-caractéristiques :

- **la maturité** représente la capacité d'un logiciel à éviter les échecs résultants de fautes dans le système ;
- **la tolérance aux pannes** représente la capacité d'un système à maintenir un certain niveau de performance en cas de fautes logicielles ;
- **la recouvrabilité** représente à la fois, la capacité d'un système à rétablir un niveau de performance, la capacité à récupérer les données, ainsi que le temps et les efforts nécessaires pour cela.

Facilité d'utilisation. Cette caractéristique décrit la capacité d'un produit logiciel à être compris, maitrisé et attractif pour l'utilisateur lorsqu'il est utilisé dans les conditions prévues. Elle décrit donc l'effort nécessaire pour utiliser le système. Cette caractéristique est raffinée par trois sous-caractéristiques :

- **l'intelligibilité** représente la capacité du système à permettre à l'utilisateur de comprendre son utilité et comment il peut être utilisé pour des taches et des conditions d'utilisation particulières ;
- **la facilité d'apprentissage** représente la capacité du système à permettre à l'utilisateur d'apprendre comment utiliser le logiciel ;
- **l'opérabilité** représente la capacité d'un système à permettre à l'utilisateur de l'utiliser et le contrôler.

Efficacité. Cette caractéristique décrit la capacité d'un produit logiciel à fournir les performances appropriées, en fonction de la quantité de ressources utilisées sous les conditions prévues. Cette caractéristique est raffinée par deux sous-caractéristiques :

- **la consommation en temps** représente la capacité d'un système à respecter le temps de réponse et d'exécution lors du calcul d'une fonction sous les conditions prévues ;
- **la consommation en espace** représente la quantité et le type de ressources utilisées ainsi que la durée de cette utilisation lors du calcul d'une fonction.

Maintenabilité. Cette caractéristique décrit la capacité d'un produit logiciel à être modifié. Ces modifications peuvent être des corrections, des améliorations ou des adaptations du système aux changements dans l'environnement et dans les spécifications fonctionnelles ou non. Elle décrit donc l'effort nécessaire à la modification. Cette caractéristique est raffinée par quatre sous-caractéristiques :

- **l'analysabilité** représente la capacité d'un système à subir un diagnostique concernant les faiblesses ou les causes d'échecs ou les parties nécessitant une modification ou une identification ;
- **la variabilité** représente la capacité d'un système à permettre à des modifications prévues d'être implémentées ;
- **la stabilité** représente la capacité d'un système à éviter le risque d'apparition d'effets inattendues dues à des modifications du logiciel ;
- **la testabilité** représente la capacité d'un système à être validé.

Portabilité. Cette caractéristique décrit la capacité d'un produit logiciel à être transféré d'un environnement à un autre. L'environnement inclut l'organisation, le matériel et le logiciel. Cette caractéristique est raffinée par quatre sous-caractéristiques :

- **l'adaptabilité** représente la capacité d'un système à être adapté à différents environnements spécifiques en utilisant uniquement ces propres fonctionnalités ;
- **la facilité d'installation** représente la capacité d'un système à être installé dans un environnement spécifié ;
- **la co-existence** représente la capacité d'un système à co-exister avec un autre système indépendant dans un environnement commun et à partager des ressources communes ;
- **la facilité de remplacement** représente la capacité d'un système à être utilisé à la place d'un autre logiciel spécifique pour le même but et dans le même environnement.

4.3.1.2 Les caractéristiques de qualité de notre modèle de mesure

Losavio [83] a proposé une projection de ces caractéristiques de qualité des produits logiciels sur la qualité architecturale. Cette étude repose principalement sur une redéfinition de certaine sous-caractéristiques et surtout sur la proposition d'un ensemble de propriétés mesurables et de leur lien avec les caractéristiques de qualité architecturale.

L'étude de ces caractéristiques de qualité révèle que certaines nécessitent des informations qui ne sont pas liées à l'architecture résultante. Ainsi, la caractéristique de fonctionnalité requiert des informations sur les fonctionnalités souhaitées du système afin de mesurer, par exemple, la précision du résultat ou la pertinence du logiciel. La facilité d'utilisation et la portabilité nécessitent une expertise humaine pour tester, entre autres, la facilité de compréhension ou la capacité d'adaptation de l'architecture. Enfin, l'efficacité requiert une simulation ou un test du système pour mesurer sa consommation en temps et en ressources.

L'étude de ces caractéristiques de qualité révèle qu'elles ne dépendent pas toutes du résultat de l'extraction, c'est-à-dire de l'architecture. En effet, l'extraction de l'architecture ne modifie ni le code ni les dépendances entre les éléments du système ; elle ne fait que les mettre en valeur et permettre leur visualisation. Ainsi, quelque soit l'architecture représentée par l'extraction les fonctionnalités souhaitées pour le système sont les mêmes, tout comme les fonctionnalités offertes. De même, l'interface utilisateur ou les performances ne sont pas modifiées lors de l'extraction de l'architecture. Les caractéristiques de fonctionnalité, facilité d'utilisation, portabilité ou efficacité ne dépendent donc pas du résultat de l'extraction. Par conséquent, elles ne permettent pas de distinguer deux éléments dans l'espace de recherche et ne peuvent donc pas diriger notre processus d'exploration.

Le résultat de notre étude des caractéristiques de qualité architecturale est donc que seules les caractéristiques de maintenabilité et de fiabilité dépendent de l'architecture extraite et peuvent être mesurées uniquement à partir des informations liées à cette architecture. De plus, ces caractéristiques sont particulièrement utiles dans le cadre de notre approche.

Comme on l'a vu plus haut, la maintenabilité d'une architecture mesure les efforts nécessaires à la maintenance de cette architecture. Une bonne maintenabilité constitue un élément essentiel dans les phases de maintenance qui peuvent suivre l'extraction. De plus, le niveau de maintenabilité peut permettre de déterminer si le système nécessite une phase d'amélioration pour augmenter la maintenabilité de l'architecture ou s'il peut être utilisé tel quel dans les phases suivantes de son cycle de vie.

La fiabilité mesure la capacité du système à maintenir son niveau de performance. L'architecture extraite n'est pas essentielle pour cette caractéristique. Cependant, comme pour la maintenabilité, son niveau permet de déterminer si le système nécessite une phase de maintenance corrective, pour corriger les problèmes de fiabilité ou s'il peut être utilisé tel quel dans les phases suivantes de son cycle de vie.

Par conséquent, nous conservons seulement ces deux caractéristiques. Elles constituent les deux sous-caractéristiques de la caractéristique qualité architecturale de notre modèle de mesure. Nous présentons dans la suite notre modèle de mesure de cette caractéristique en présentant les raffinements successifs des sous-caractéristiques maintenabilité et fiabilité.

4.3.2 Des caractéristiques de qualité aux propriétés architecturales

Les caractéristiques de qualité architecturale que nous avons selectionnée constituent, dans notre modèle de mesure, les sous-caractéristiques de la caractéristique qualité architecturale. Ces sous-caractéristiques doivent maintenant être raffinées en un ensemble de propriétés mesurables sur l'architecture et ses éléments architecturaux.

Nous présentons d'abord les liens que nous avons identifiés entre la maintenabilité et les propriétés mesurables des éléments architecturaux. Nous examinons ensuite les liens entre la fiabilité et les propriétés mesurables des éléments architecturaux.

4.3.2.1 Maintenabilité de l'architecture

La maintenabilité mesure l'effort requis pour maintenir l'architecture. Cette caractéristique est dépendante, par exemple, de la stabilité et de la capacité de changement de l'architecture. La maintenabilité d'une architecture dépend de la maintenabilité de chacun des éléments architecturaux qui la composent : configuration, connecteurs et composants. En effet, une architecture, dont la configuration est facilement maintenable, peut être difficile à maintenir si ses composants sont difficiles à maintenir.

Par conséquent, nous raffinons la maintenabilité par les propriétés des éléments architecturaux **maintenabilité de la configuration**, **maintenabilité des composants** et **maintenabilité des connecteurs**.

4.3.2.2 Fiabilité de l'architecture

La fiabilité est la capacité d'un système à exécuter la fonction demandée au moment demandé. Plusieurs approches ont été proposées pour mesurer cette caractéristique. Dans la plupart de ces approches, la mesure repose sur une évaluation dynamique [130] ou une approximation reposant sur des experts du système [54]. Ces deux méthodes utilisent des informations qui ne peuvent être obtenues qu'à travers l'expertise humaine. Elles sont donc à l'opposé de notre approche qui vise à réduire ce besoin en expertise humaine.

Ces travaux nous permettent cependant de raffiner la fiabilité. En effet, la mesure de la fiabilité de l'architecture dépend, dans ces travaux, de celle des composants et de leurs liens. Dans [54], la fiabilité de l'architecture est la moyenne de la fiabilité des composants, pondérée par le nombre de degré de chaque composant dans l'architecture. En effet, plus le composant est lié dans l'architecture plus sa fiabilité aura un impact sur celle de l'architecture. Ainsi, la fiabilité de l'architecture se mesure à travers **la fiabilité des composants** et **le nombre de connecteurs reliés au composant**.

Cependant, le degré des composants utilisé dans ces travaux repose sur des connecteurs représentés par des liens simples. Or, nous considérons des connecteurs simples mais non-vides et qui par conséquent peuvent avoir une certaine mesure de fiabilité. Nous prenons en compte cette fiabilité de la même façon que celle des composants. En effet, plus le connecteur est lié dans l'architecture plus sa fiabilité aura un impact sur celle de l'architecture. Ainsi, la fiabilité de l'architecture se mesure également à travers **la fiabilité des connecteurs** et **le nombre de composants reliés au connecteur**.

4.3.3 Des propriétés architecturales aux propriétés des entités COA

Suivant notre méta-modèle de mesure (*cf.* Figure 4.2), les propriétés mesurables sur les éléments architecturaux doivent être reliées aux propriétés mesurables des entités du modèle COA. Nous étudions donc successivement les liens entre les propriétés concernant la maintenabilité et la fiabilité avec les propriétés des entités COA.

4.3.3.1 Propriétés de la maintenabilité

La maintenabilité est une propriété qui a été beaucoup étudiée dans le cadre du paradigme objet [67]. Par conséquent, nous considérons que la maintenabilité des différentes entités COA constitue une propriété mesurable. Ainsi, d'aprés le modèle COA (*cf.* Figure 3.9), nous obtenons le raffinement suivant :

- **la propriété maintenabilité des composants** peut être mesurée par la propriété des entités COA **maintenabilité des contours de composants** ;
- **la propriété maintenabilité des connecteurs** peut être mesurée par la propriété des entités COA **maintenabilité des contours de connecteurs** ;
- **la propriété maintenabilité de la configuration** peut être mesurée par la propriété des entités COA **maintenabilité du contour de configuration**.

4.3.3.2 Propriétés de la fiabilité

Les travaux sur les métriques objets ont établi des liens entre la fiabilité et la complexité du logiciel [63]. Ces liens reposent sur le constat suivant : plus un système est complexe et plus le risque qu'il contienne une erreur grave augmente. Ce risque d'erreur induit le lien entre la complexité et la fiabilité d'un système. Par conséquent, les propriétés de fiabilité des composants et des connecteurs peuvent être mesurées par les propriétés des entités COA **complexité des contours de composants** et **complexité des contours de connecteurs**.

Le reste des propriétés relative à la fiabilité, c'est-à-dire le nombre de composants reliés à un connecteur et le nombre de connecteurs reliés à un composant, sont, de manière évidente, mesurable par le nombre des entités COA correspondantes, c'est-à-dire le **nombre de contours de composant par contour de connecteur et inversement**.

4.3.4 Des propriétés COA aux métriques objets

Nous avons établi dans les sections précédentes, les premiers niveaux de notre modèle de mesure de la qualité architecturale (*cf.* Figure 4.9). Nous définissons maintenant le dernier niveau de ce modèle, c'est-à-dire les métriques permettant de mesurer les propriétés mesurables des entités COA.

Ces métriques sont basées sur celles mesurant la maintenabilité et la complexité des modules objets qui sont deux propriétés qui ont déjà été étudiées. Nous commençons donc, pour la maintenabilité puis la complexité, par définir la propriété que l'on veut mesurer et les contraintes qui s'appliquent sur les métriques dans notre approche. Nous présentons ensuite les mesures que nous avons choisies pour compléter notre modèle de mesure.

4.3.4.1 Maintenabilité des modules objets

Selon [67], la maintenabilité d'un module décroit avec le nombre de dépendances entre les classes du module. Par comparaison avec les systèmes physiques, les classes peuvent être réparties en deux phases :

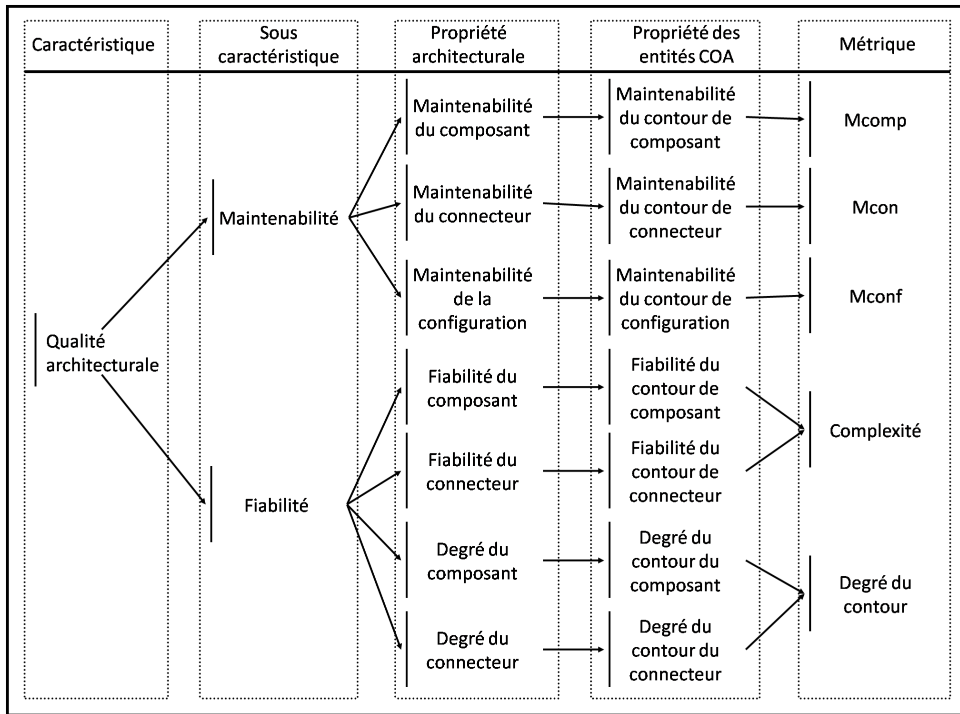


Figure 4.9 – Notre modèle de mesure de la qualité architecturale

une stable et une instable. La mesure de la proportion des classes appartenant à la première catégorie donne une mesure de la maintenabilité du module.

Jenkins définit un graphe pour chaque module dont les sommets sont ses classes. Il existe un arc entre deux sommets a et b si la classe a utilise la classe b, *i.e.* utilise un attribut, un paramètre ou une variable de retour dont le type est b, crée ou utilise un objet dont le type peut être b. Par analogie avec les graphes de changement de phases dans les systèmes physiques (*cf.* Figure 4.10), ce graphe peut être partitionné en deux phases en fonction du degré des sommets. Par référence aux changements de phases, l'une de ces phases est dites stable et l'autre instable.

Pour décrire les deux phases, Jenkins définit k_t comme la solution de l'équation 4.3, où $n(k)$ est le nombre de sommets ayant un degré k et k_{max} est le degré maximum du graphe. Un sommet est donc dans la première phase, la phase stable, si son degré est inférieur à k_t et dans la seconde sinon, la phase instable.

$$\sum_{k'=0}^{k_t} k' \cdot n(k') = \sum_{k'=k_t}^{k_{max}} k' \cdot n(k') \tag{4.3}$$

La mesure de la maintenabilité du module est alors la mesure du pourcentage de classes appartenant à la phase stable.

Nous utilisons la mesure de la maintenabilité proposée par Jenkins pour mesurer la maintenabilité des différentes entités COA. Pour cela, nous assimilons ces différentes entités ainsi que les entités objets qui les composent à celles utilisées par Jenkins

Le calcul de la maintenabilité du contour d'un composant est réalisé sur les classes contenues dans chaque contour de composant. **Nous assimilons un contour de composant à un module et une classe**

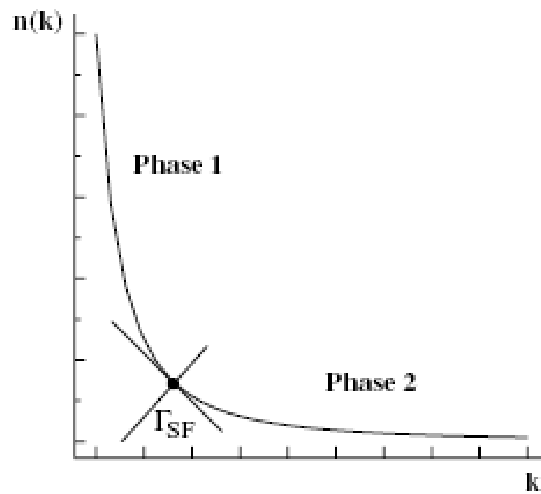


Figure 4.10 – Graphe de changement de phases dans les systèmes physiques

du contour à une classe. Ainsi, les sommets du graphe sont les classes du contour de composant et il existe une arête entre deux sommets A et B si la classe A est connecté à la classe B selon les règles établies par Jenkins. La mesure du pourcentage de sommets appartenant à la phase stable mesure la maintenabilité du contour du composant et définit notre fonction $MComp(c)$ où c est le contour du composant.

Le calcul de la maintenabilité du contour d'un connecteur est réalisé sur les méthodes et les attributs contenus dans chaque contour de connecteur. **Nous assimilons un contour de connecteur à un module et chaque méthode ou attribut à une classe**. Ainsi, les sommets du graphe sont les méthodes et attributs du contour de connecteur et il existe une arête entre deux sommets A et B si la méthode A appelle la méthode B ou accède à l'attribut B. La mesure du pourcentage de sommets appartenant à la phase stable mesure la maintenabilité du contour du connecteur et définit notre fonction $MCon(c)$ où c est le contour du connecteur.

Le calcul de la maintenabilité du contour de la configuration est réalisé sur la partition des classes constituée par les contours de composant. **Nous assimilons une configuration à un module et un composant à une classe**. Ainsi, les sommets du graphe sont les composants et il existe une arête entre deux sommets A et B si le composant A est connecté au composant B. La mesure du pourcentage de sommets appartenant à la phase stable mesure la maintenabilité de la configuration et définit notre fonction $MConf(c)$ où c est le contour de la configuration.

4.3.4.2 Complexité des modules objets

Selon [63], nous pouvons mesurer la complexité d'un module en utilisant l'équation (4.4), où $Inter$ est la complexité inter-module reflétant le couplage des modules et $Intra$ est la complexité intra-module reflétant la cohésion du module.

$$Compl = Intra \cdot (Inter)^2 \tag{4.4}$$

Nous utilisons cette métrique en assimilant successivement nos entités COA, contour de composant et contour de connecteur, à un module. Ainsi l'équation (4.4) peut être utilisée avec les métriques de cohésion et de couplage que nous utilisons (*cf.* Section 4.2.4) pour mesurer la complexité inter et intra

contour. Pour uniformiser les valeurs avec les autres métriques, nous définissons notre métrique pour la complexité selon l'équation suivante, ou c désigne un contour de composants ou de connecteurs :

$$Complexite\,(c) = Compl/10^4 \tag{4.5}$$

4.4 Définition de la fonction objectif

Notre processus d'exploration nécessite la définition d'une fonction permettant de mesurer la qualité d'une solution. Cette fonction permet de sélectionner la meilleure solution mais surtout elle permet l'exploration de l'espace des solutions en offrant un moyen de comparaison entre les entités de cet espace. Elle repose sur la qualité mesurée par rapport aux guides que nous venons d'étudier et qui définissent les objectifs de notre processus.

La définition de cette fonction repose sur le modèle de mesure de la qualité de la solution que nous avons présenté dans les sections précédentes. Nos deux guides, nous ont permis de définir deux notions décrivant la qualité d'une solution ainsi que tout un modèle de mesure pour chacune d'entre elles.

Dans la suite, nous présentons les deux fonctions permettant de mesurer la validité sémantique et la qualité de l'architecture. Nous utilisons, ensuite, ces deux fonctions pour définir la fonction objectif de notre processus.

4.4.1 Mesure de la validité sémantique de l'architecture

Nous utilisons le modèle de mesure de la validité sémantique de l'architecture afin de définir une fonction d'évaluation utilisant la sémantique associée à l'architecture et aux différents éléments architecturaux.

Ce modèle décrit la caractéristique « validité sémantique de l'architecture » à travers deux ensembles de sous-caractéristiques : les sous-caractéristiques se rapportant à la validité sémantique des composants et celle décrivant la validité sémantique des connecteurs. Par conséquent, nous définissons deux fonctions basées sur le modèle de mesure et utilisant nos métriques pour évaluer chacun de ces ensembles.

A partir de ces deux fonctions d'évaluation, nous définissons une fonction d'évaluation de la validité sémantique d'une architecture. Cette fonction peut être utilisée comme fonction objectif dans notre processus d'extraction. Elle oriente alors l'extraction vers des architectures sémantiquement valides.

4.4.1.1 Evaluation de la validité sémantique des composants

En exploitant le modèle de mesure de la validité sémantique de l'architecture (*cf.* Figure 4.4), nous définissons une fonction d'évaluation pour chacune des sous-caractéristiques de la validité sémantique des composants : l'autonomie, la spécificité et la composabilité.

Pour cela, il faut considérer le graphe formé par le modèle de mesure comme une forêt. Chaque caractéristique constitue une racine et les métriques sont les feuilles. Il suffit alors d'assembler les métriques racines du sous-arbre formé à partir de la sous-caractéristique. Il reste enfin à établir les pondérations permettant d'équilibrer les differentes métriques. Nous procédons alors par expérimentations et affinages successifs.

Nous utilisons ensuite ces fonctions pour définir la fonction d'évaluation de la validité sémantique des composants.

Fonction d'évaluation de la spécificité. Nous avons montré que la spécificité dépend de la cohésion moyenne des services par interfaces, du nombre d'interfaces fournies, de la cohésion entre les interfaces fournies, de la cohésion interne et du couplage interne du composant. D'après notre modèle de mesure, la spécificité d'un contour de composant c peut donc être mesuré par la fonction $Spe_{comp}(c)$, où IF_c est l'ensemble des contours d'interfaces fournies associées au contour c :

$$Spe_{comp}(c) = \frac{1}{4} \cdot \left(\frac{1}{|IF_c|} \cdot \sum_{i \in IF_c} LCC(i) + LCC(IF) + LCC(c) + Couplage(c) - 10 \cdot |IF_c| \right)$$

Dans cette formule $LCC(i)$ mesure la cohésion entre les services d'un contour d'interface i, $LCC(IF)$ mesure la cohésion entre les contours d'interfaces fournies et $LCC(c)$ et $Couplage(c)$ mesurent respectivement la cohésion et le couplage entre les classes du contour de composant c.

Fonction d'évaluation de l'autonomie. Nous avons montré que l'autonomie dépend du nombre d'interfaces requises et du nombre d'interfaces requises verticales. D'après notre modèle de mesure, l'autonomie d'un contour de composant c est donc mesurée à travers la fonction $Auto_{comp}(c)$, où $IR(c)$ est l'ensemble des contours d'interfaces requises associées au contour c et $IRv(c)$ est l'ensemble des contours d'interfaces requises verticales associées au contour c :

$$Auto_{comp}(c) = -5 \cdot [|IR| + 2 \cdot |IRv|]$$

Fonction d'évaluation de la composabilité. Nous avons montré que la composabilité dépend du nombre d'interfaces requises verticales et de la moyenne de la cohésion entre les services d'une interface fournie. D'après notre modèle de mesure, la composabilité d'un contour de composant c est donc mesurée par la fonction $Comp_{comp}(c)$, où IRv_c est l'ensemble des contours d'interfaces requises verticales associées au contour c et IF_c est l'ensemble des contours d'interfaces fournies associées au contour c :

$$Comp_{comp}(c) = \frac{1}{|IF_c|} \cdot \sum_{i \in IF} LCC(i) - 10 \cdot |IRv_c|$$

Fonction d'évaluation de la sémantique des composants. L'évaluation de la caractéristique « validité sémantique des composants » est basée sur l'évaluation de chaque sous-caractéristique. Nous définissons cette fonction comme une combinaison linéaire des fonctions d'évaluation (Spe_{comp}, $Auto_{comp}$, et $Comp_{comp}$) :

$$Sem_{comp}(c) = \frac{1}{\sum_i \lambda_i} [\lambda_1 \cdot Comp_{comp}(c) + \lambda_2 \cdot Auto_{comp}(c) + \lambda_3 \cdot Spe_{comp}(c)] \quad (4.6)$$

Cette forme linéaire nous permet de considérer uniformément les sous-caractéristiques. Le poids associé à chaque fonction permet à l'architecte de modifier, au besoin, l'importance relative des sous-caractéristiques.

4.4.1.2 Evaluation de la validité sémantique des connecteurs

En exploitant le modèle de mesure de la validité sémantique de l'architecture (*cf.* Figure 4.5), nous définissons une fonction d'évaluation pour chacune des sous-caractéristiques de la validité sémantique

des connecteurs : l'autonomie, la spécificité et la généricité. Pour cela, nous utilisons une approche identique à celle de la section précédente.

Nous utilisons ensuite ces fonctions pour définir la fonction d'évaluation de la validité sémantique des connecteurs.

Fonction d'évaluation de la spécificité. Nous avons montré que la spécificité dépend de la cohésion et du couplage interne du connecteur. D'après notre modèle de mesure, la spécificité d'un contour de connecteur c est mesurée à travers la fonction $Spe_{con}\left(c\right)$, où $LCC\left(c\right)$ et $Couplage\left(c\right)$ mesurent respectivement la cohésion et le couplage entre les méthodes et attributs du contour de connecteur c :

$$Spe_{con}(c) = \frac{1}{2} \cdot \left(LCC\left(c\right) + Couplage\left(c\right)\right)$$

Fonction d'évaluation de l'autonomie. Nous avons montré que l'autonomie dépend de la connexité de la glu du connecteur. D'après notre modèle de mesure, l'autonomie d'un contour de connecteur c est donc une valeur binaire qui indique si les méthodes et attributs du contour forment un ensemble connexe.

Fonction d'évaluation de la généricité. Nous avons montré que la généricité dépend du nombre de rôles, de la cohésion entre les rôles, du couplage entre les rôles, de la moyenne des couplages de chaque rôle et de la moyenne des cohésions de chaque rôle. D'après notre modèle de mesure, la généricité d'un contour de connecteur c est mesurable par la fonction $Gen_{con}\left(c\right)$, où R_c est l'ensemble des rôles associés au contour c :

$$Gen_{con}\left(c\right) = \frac{1}{4}\left[LCC\left(R_c\right) + Couplage\left(R_c\right)\right] - 10 \cdot |R_c| + \frac{1}{8 \cdot |R_c|} \cdot \sum_{r \in R_c} \left(LCC\left(r\right) + Couplage\left(r\right)\right)$$

Fonction d'évaluation de la sémantique des connecteurs. L'évaluation de la caractéristique « validité sémantique des connecteurs » est basée sur l'évaluation de chaque sous-caractéristique. Nous définissons cette fonction comme une combinaison linéaire des fonctions d'évaluation (Spe_{con} et Gen_{con}, et $Auto_{con}$) :

$$Sem_{con}\left(c\right) = \frac{Auto_{con}}{\sum_i \lambda_i}\left[\lambda_4 \cdot Gen_{con}\left(E\right) + \lambda_5 \cdot Spe_{con}\left(E\right)\right] \tag{4.7}$$

Cette forme linéaire nous permet de considérer uniformément les sous-caractéristiques. Le poids associé à chaque fonction permet à l'architecte de modifier, au besoin, l'importance relative des sous-caractéristiques.

4.4.1.3 Evaluation de la validité sémantique de l'architecture

L'évaluation de la caractéristique de validité sémantique de l'architecture repose sur les deux fonctions d'évaluations de la sémantique des éléments architecturaux : Sem_{con} et Sem_{comp}. Nous proposons de définir cette mesure comme une moyenne pondérée de la validité sémantique de chaque élément de l'architecture extraite. Ainsi, nous définissons la fonction d'évaluation d'une solution s selon l'équation suivante, où $E_{comp}\left(s\right)$ et $E_{con}\left(s\right)$ désignent respectivement l'ensemble des contours de composants et de connecteurs de s :

$$Sem(s) = (\lambda_6 + \lambda_7 + |E_{comp}(s)| + |E_{con}(s)|)^{-1} \cdot \left(\lambda_6 \cdot \sum_{c \in E_{comp}(s)} Sem_{comp}(c) + \lambda_7 \cdot \sum_{c \in E_{con}(s)} Sem_{con}(c)\right) \quad (4.8)$$

Les pondérations sont fixées par l'architecte et permettent de définir l'importance relative accordée aux composants et aux connecteurs.

4.4.2 Mesure de la qualité architecturale

La caractéristique « qualité architecturale » est basée sur le guide qui utilise les propriétés de qualité des architectures et des différents éléments architecturaux. Pour utiliser cette notion durant l'exploration, nous utilisons le modèle de mesure que nous avons proposé afin de définir une fonction d'évaluation de la qualité architecturale.

Pour définir cette fonction, nous proposons pour chaque sous-caractéristique, maintenabilité et fiabilité, une fonction d'évaluation basée sur le modèle de mesure et utilisant nos métriques pour la mesure de la maintenabilité, la complexité et le couplage.

A partir de ces deux fonctions d'évaluation, nous définissons une fonction d'évaluation de la qualité architecturale d'une solution. Cette fonction peut être utilisée pour compléter la fonction traitant de la validité sémantique afin de définir la fonction objectif dans notre processus d'extraction.

4.4.2.1 Mesure de la maintenabilité de l'architecture

Comme nous l'avons vu, la maintenabilité d'une architecture dépend de la maintenabilité de chacun des éléments architecturaux qui la compose : configuration, connecteurs et composants. Nous avons étudié, à travers le modèle de mesure comment mesurer cette propriété sur chacun des éléments architecturaux et nous présentons ici la fonction utilisant ces mesures pour évaluer la maintenabilité d'une architecture.

La maintenabilité de l'architecture est définie comme la moyenne de la maintenabilité des composants, de celle des connecteurs et de la maintenabilité de la configuration. Elle est mesurée pour une solution s par la fonction $Maint(s)$, où $E_{comp}(s)$ et $E_{con}(s)$ désignent respectivement l'ensemble des contours de composants et de connecteurs de s, et $c(s)$ désigne la configuration associée à s :

$$Maint(s) = \frac{1}{3 \cdot |E_{comp}(s)|} \sum_{c \in E_{comp}(s)} MComp(c) + \frac{1}{3 \cdot |E_{con}(s)|} \sum_{c \in E_{con}(s)} MCon(c) + \frac{1}{3} \cdot MConf(c(s))$$

4.4.2.2 Mesure de la fiabilité de l'architecture

La mesure de la fiabilité de l'architecture dépend des composants et de leurs liens. Selon notre modèle de mesure, nous proposons de mesurer cette fiabilité par une moyenne pondérée de la complexité des contours de composants et de celle des connecteurs. Le poids associé à chaque composant est le nombre

de connecteurs qui sont reliés au composant, alors que le poids associé à chaque connecteur est le nombre de composants reliés à travers le connecteur.

Nous proposons la fonction $Fiab\left(s\right)$ pour mesurer la fiabilité d'une solution s, où $E_{comp}\left(s\right)$ et $E_{con}\left(s\right)$ désignent respectivement l'ensemble des contours de composants et de connecteurs de s, $Comp\left(c\right)$ et $Con\left(c\right)$ désignent respectivement l'ensemble des contours de composants reliés au connecteur c et l'ensemble des contours de connecteurs reliés au composant c :

$$Fiab\left(s\right) = \left[\sum_{c \in E_{comp}(s)} \left|Comp\left(c\right)\right| + \sum_{c \in E_{con}(s)} \left|Con\left(c\right)\right|\right]^{-1} \cdot \left[\sum_{c \in E_{comp}(s)} Complexite\left(c\right) \cdot \left|Con\left(c\right)\right| + \sum_{c \in E_{con}(s)} Complexite\left(c\right) \cdot \left|Comp\left(c\right)\right|\right]$$

4.4.2.3 Mesure de la qualité architecturale

L'évaluation de la caractéristique « qualité architecturale » repose sur les deux fonctions d'évaluation des sous-caractéristiques : $Maint\left(s\right)$ et $Fiab\left(s\right)$. Nous proposons de définir cette mesure comme la moyenne pondérée de ces fonctions. Ainsi, nous définissons la fonction d'évaluation d'une solution s selon l'équation suivante, où $E_{comp}\left(s\right)$ et $E_{con}\left(s\right)$ désignent respectivement l'ensemble des contours de composants et de connecteurs de s :

$$Qual\left(s\right) = \frac{1}{\sum_i \lambda_i} \left(\lambda_8 \cdot Maint\left(s\right) + \lambda_9 \cdot Fiab\left(s\right)\right) \tag{4.9}$$

Comme pour l'équation 4.8, celle-ci est une combinaison linéaire afin de considérer uniformément chaque partie. Le poids associé à chaque fonction autorise l'architecte à modifier l'importance de chaque caractéristique de qualité.

4.4.3 Fonction objectif du processus d'exploration

La fonction objectif de notre processus, nous permet de mesurer la qualité d'une solution et de comparer les solutions entre elles tout au long du processus d'extraction. Elle repose sur les guides que nous avons présentés dans ce chapitre, c'est-à-dire la sémantique de la notion d'architecture et la qualité architecturale.

Pour définir cette fonction, nous avons présenté un modèle de mesure de la qualité d'une solution. Ce modèle repose sur les propriétés, extraites des deux guides, que nous avons présentées. Nous avons également présenté une fonction d'évaluation pour chaque caractéristique, $Qual\left(s\right)$ et $Sem\left(s\right)$. Au final, la définition d'une fonction objectif basée sur ces deux fonctions possède les propriétés requises pour être utilisée dans notre processus : elle permet d'évaluer la qualité d'une solution en mesurant la validité sémantique et la qualité architecturale de l'architecture correspondante à la solution.

$$Goal\left(s\right) = \frac{1}{\sum_i \lambda_i} \left(\lambda_{10} \cdot Sem\left(s\right) + \lambda_{11} \cdot Qual\left(s\right)\right) \tag{4.10}$$

Nous définissons la fonction objectif $Goal\left(s\right)$ comme la moyenne pondérée des fonctions définies précédemment. Les pondérations sont fixées par l'architecte et permettent de sélectionner l'importance relative accordée aux deux guides.

4.5 Conclusion

Nous avons présenté dans ce chapitre les objectifs de notre approche d'extraction. Ces objectifs définissent la qualité des solutions à travers la qualité et la sémantique architecturale. Ils permettent ainsi d'obtenir une solution qui répond aux exigences de qualité de l'architecte et qui respecte la définition qu'il a des architectures logicielles et des éléments architecturaux.

Pour utiliser ces objectifs dans une approche par exploration, nous avons proposé une fonction objectif qui évalue la qualité des solutions en fonction de ces deux critères. Pour cela, nous avons réalisé une fonction d'évaluation de la qualité architecturale et une autre évaluant l'adéquation entre la solution et le concept d'architecture. La première repose sur un modèle de mesure utilisant les caractéristiques de fiabilité et de maintenabilité pour décrire la qualité architecturale. La seconde fonction repose sur une réification des définitions couramment admises des différents éléments architecturaux en un ensemble de caractéristiques sémantiques des composants et des connecteurs. Nous avons ensuite défini un modèle de mesure de la validité sémantique basé sur ces caractéristiques.

La fonction objectif ainsi définie, nous permet donc d'identifier les bonnes solutions du point de vue de la qualité de l'architecture extraite et de sa validité par rapport aux définitions. Cependant, elle ne repose que sur les informations contenues dans le code source alors qu'il existe d'autres informations telle que la documentation qui permettent d'identifier une certaine architecture du système.

CHAPITRE 5

Guider le processus d'extraction : architecture intentionnelle et contraintes architecturales

Know where to find the information and how to use it - That's the secret of success

— Albert EINSTEIN.

Guidage de l'exploration — Réduction de l'espace de recherche — Documentation — Recommandations de l'architecte — Contexte de déploiement

Parmi les guides que nous avons identifiés pour le processus d'extraction, nous avons utilisé les guides structurels pour définir la fonction objectif de notre processus d'exploration. Cette fonction permet d'extraire une architecture qui répond aux exigences de qualité des architectes et qui correspond à leur définition de l'architecture logicielle et des différents éléments architecturaux. Ainsi, l'exploration de l'espace des solutions en utilisant cette fonction objectif repose uniquement sur les informations structurelles contenues dans le code source du système.

L'architecture extraite représente dans ce cas uniquement le système tel qu'il est implémenté. Pourtant, il est évident que l'implémentation n'est pas une représentation exacte de l'architecture d'un système. En effet, l'implémentation oblige à faire des choix et à adapter l'architecture proposée par les architectes pour tenir compte du langage utilisé, des compétences disponibles ou encore de l'architecture matérielle cible. Ainsi, faire reposer notre processus uniquement sur notre fonction objectif conduit à l'extraction d'une architecture qui, si elle représente correctement le système tel qu'il est dans le code, peine à représenter le système tel qu'il a été conçu ou tel que les architectes le visualisent.

Pour permettre à notre approche d'extraire l'architecture souhaitée par l'architecte, nous proposons d'utiliser les deux guides auxiliaires que nous avons identifiés : la documentation et les recommandations de l'architecte et le contexte de déploiement. En effet, ces guides nous permettent d'obtenir deux types d'informations essentielles pour obtenir une architecture représentant le système dans son ensemble :

- **les informations intentionnelles** : elles décrivent la volonté des créateurs du système, c'est-à-dire les dépendances ou l'architecture qu'ils ont souhaitées mettre en place, ou encore les fonctionnalités qu'ils ont imaginées pour chaque classe du système. Ces informations sont connues des architectes et des développeurs du système et sont surtout *transmises oralement*. Heureusement, elles sont souvent *archivées dans les documents* de conception du système. Nous pouvons donc accéder à ces informations intentionnelles à travers la documentation et les recommandations de l'architecte. Elles peuvent nous permettre, par exemple, d'identifier certains contours de composants ou certaines dépendances entre deux classes, imposant qu'elles appartiennent au même contour ;
- **les informations contextuelles** : elles décrivent le contexte dans lequel le système doit être déployé, c'est-à-dire les limitations en termes de mémoire ou de communication par exemple. Ces informations nous permettent par exemple de définir certaines contraintes sur les contours, telles que des limitations de taille pour respecter les limitations en mémoire ou des limitations de diamètre pour les contours de connecteurs pour répondre aux limitations sur les temps de communication.

Ces deux types d'informations mettent en évidence des propriétés que l'utilisateur souhaite retrouver dans la solution finale proposée. Elles nous permettent donc d'éliminer certaines solutions qui ne correspondent pas aux attentes de l'architecte et de cibler celles qui s'en rapprochent le plus.

La suite du chapitre est organisée de la façon suivante. Nous présentons d'abord les guides de l'extraction en détaillant les informations qu'ils contiennent ainsi que la façon dont elles peuvent être utilisées durant l'exploration. Ensuite, nous étudions le processus d'extraction de l'architecture intentionnelle à partir de la documentation. Pour cela, nous présentons d'abord les aspects généraux de l'extraction et les apports de l'architecte, puis nous détaillons les aspects particuliers à chaque type de documents. Enfin, nous présentons les étapes de l'extraction d'un réseau de contraintes architecturales à partir des recommandations de l'architecte, du contexte de déploiement et des architectures intentionnelles extraites de la documentation.

5.1 Les guides de l'extraction

Nous étudions, dans cette section, les informations intentionnelles et contextuelles ainsi que leurs liens avec les guides auxiliaires. Ensuite, nous proposons un modèle des informations permettant de les utiliser lors de l'exploration. Enfin, nous détaillons les méthodes utilisant ces informations pour guider le processus en réduisant l'espace de recherche.

5.1.1 Les informations intentionnelles

Les informations intentionnelles reflètent l'architecture que les concepteurs ont imaginée pour le système. Elles donnent ainsi une image de l'architecture du système avant qu'elle ne soit soumise au bruit dû à l'utilisation d'un langage de programmation et au passage de plusieurs phases de maintenance. Elles décrivent donc l'architecture intentionnelle d'un système, c'est-à-dire l'architecture que les concepteurs ont souhaitée mettre en place. Elles permettent donc d'identifier des dépendances entre classes ou des fonctionnalités imaginées lors de la conception.

L'utilisation de l'information intentionnelle est un challenge important pour les travaux d'extraction d'architectures. Comme nous l'avons vu dans le chapitre 2, plusieurs travaux [69, 88, 123] utilisent des experts pour retranscrire ces informations et les utiliser dans le processus pour obtenir une vue de l'architecture ou affiner la vue obtenue.

De façon similaire, nous utilisons les recommandations de l'architecte pour fournir des informations intentionnelles. Elles définissent un ensemble de propriétés que doit posséder une architecture pour correspondre à la vision de son architecte. Ces recommandations portent sur tous les éléments architecturaux et peuvent être positives ou négatives. L'architecte peut, par exemple, imposer que deux classes appartiennent au même contour de composant ou à l'inverse que ces deux classes appartiennent à deux contours différents. Cependant, comme pour les autres travaux, cela a pour conséquence directe, une augmentation de la dépendance du processus par rapport aux experts.

Afin de limiter le recours à des experts, nous proposons d'utiliser la documentation pour pouvoir prendre en compte, dans un processus semi-automatique, l'architecture intentionnelle d'un système. En effet, le cycle de vie d'un logiciel voit la création de nombreux documents visant à faciliter la compréhension du système en conservant et en explicitant les informations intentionnelles.

Les informations intentionnelles contenues dans la documentation sont différentes de celles fournies par l'architecte. D'abord, elles définissent des propriétés sur les composants qui sont positives. Elles peuvent, par exemple, montrer que des classes collaborent étroitement et appartiennent donc à un même contour. De plus, les informations contenues dans la documentation concernent principalement les composants et la configuration mais peu les connecteurs. En effet, la documentation s'attache surtout à décrire les fonctionnalités du système et comment elles sont mises en œuvre. Les informations concernant les modes de communication entre les modules, par exemple, sont souvent considérées comme des « détails » d'implémentation.

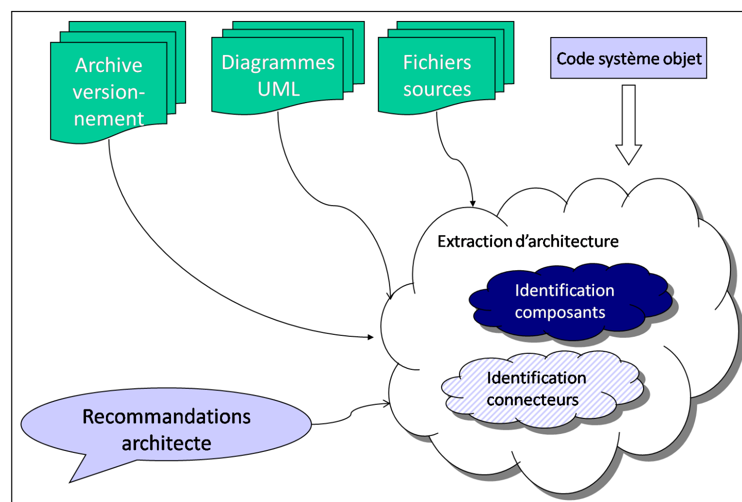


Figure 5.1 – Les documents et les recommandations disponibles

En plus des recommandations de l'architecte, nous avons identifié trois sources d'informations, parmi l'ensemble des documents disponibles, qui nous offrent quantité d'informations sur les relations entre les classes et nous donnent une vue au moins partielle de l'architecture du système (*cf.* Figure 5.1) : les diagrammes UML, les fichiers du code source et les archives des outils de versionnement.

Diagrammes UML. Nous utilisons les diagrammes UML pour diriger notre processus vers l'architecture intentionnelle du système. En effet, la modélisation UML d'un système est l'un des types de documents le plus couramment utilisé aujourd'hui. Les diagrammes de ce modèle accompagnent un logiciel durant tout son cycle de vie.

Ils sont d'abord utilisés dans la phase de conception, pour modéliser les fonctionnalités du système, les classes associées à ses fonctionnalités et les relations entre ces classes. Par la suite, ils sont affinés pendant la phase de production et se rapprochent alors de la structure réelle du système. Enfin, pendant les phases de maintenance, s'ils sont maintenus à jour, ils permettent de suivre l'évolution du système.

Ces diagrammes constituent une source importante d'informations intentionnelles. Ils modélisent les liens entre les classes, tels que les concepteurs les prévoient puis les supposent. A ce titre, ils contiennent suffisamment d'informations pour permettre d'obtenir une vue de l'architecture intentionnelle d'un système.

Exemple 5.1 (Informations intentionnelles disponibles dans un diagramme UML).
La figure 5.2 montre un diagramme de classes UML. Le diagramme montre que les classes ***A*** *et* ***B*** *sont liées par une relation de composition, alors que la classe* ***C*** *est disjointe et n'a pas de relation avec* ***A*** *ou* ***B****. Nous pouvons donc supposer que les classes* ***A*** *et* ***B*** *doivent appartenir à un même composant, et que la classe* ***C*** *fait partie d'un autre composant.*

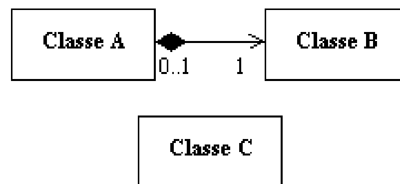


Figure 5.2 – Informations intentionnelles disponibles dans un diagramme UML

Fichiers sources. Nous utilisons le code source pour diriger notre processus vers l'architecture intentionnelle du système. En effet, les fichiers du code source constituent un autre type de documents contenant des informations utiles pour l'extraction d'une architecture intentionnelle. Ils contiennent, outre le code source de l'application, des informations sur les intentions des programmeurs.

Ces informations sont stockées à la fois, dans les commentaires qui sont associés au code et dans le code lui-même, à travers le choix des noms pour les entités du programme. Elles établissent un lien entre les classes et les fonctionnalités que les programmeurs leur associent. Ces liens permettent donc d'établir une vue de l'architecture intentionnelle du système.

Exemple 5.2 (Informations intentionnelles disponibles dans un fichier source).
La figure 5.3 montre deux exemples de fichiers de classe Java. Le premier fichier contient une classe qui est décrite par son nom et ses commentaires associés comme réalisant le tri d'une liste donnée en paramètre selon un algorithme de tri par fusion. Le second fichier contient une classe décrite, par les commentaires, comme une classe de tri selon l'algorithme de tri par insertion. Ainsi, les commentaires mettent en évidence la proximité sémantique des deux classes et constituent un indicateur précieux pour les regrouper sur la base de la proximité de leurs parties métier.

```java
/**
 *  Classe de tri
 *  algorithme de tri fusion
 */

public class Tri_fusion {

    /**
     * @param list la liste à trier
     * méthode de tri fusion
     * copie la liste en parametre
     * retourne une nouvelle instance de List
     */

    public List tri_Fusion_list(List liste){
        List liste_triee= new List();
        //   ...
        return liste_triee;
    }

}
```

```java
/**
 *  Classe de tri
 *  algorithme de tri par insertion
 */
public class ClassB {

    /**
     * @param l liste à trier
     * méthode de tri par insertion
     * modifie la liste en parametre
     */
    public void tri_liste(List l) {
        // ...
    }

}
```

Figure 5.3 – Informations intentionnelles disponibles dans un fichier source

Rapports des outils de versionnement. Nous utilisons les rapports des outils de versionnement pour obtenir une architecture intentionnelle du système qui nous permet d'influer sur notre processus d'extraction. En effet, durant les phases de maintenance, la structure de l'application est modifiée et ces modifications sont, la plupart du temps, archivées d'une manière plus ou moins précise à travers les différents outils de versionnement.

Les informations stockées peuvent être utilisées pour détecter les liens, supposés par les programmeurs, entre les classes. En effet, des classes qui semblent liées pour un programmeur, seront probablement modifiées en cascade ou en simultané. Ces liens entre les classes permettent d'établir une vue de l'architecture intentionnelle du système.

Exemple 5.3 (Informations intentionnelles disponibles à travers un outil de versionnement).

Le tableau 5.1 représente les informations archivées par un outil de versionnement lors de la maintenance d'un système constitué de quatre classes ***A****,* ***B****,* ***C*** *et* ***D****. La première colonne donne le nom de la classe, la deuxième précise l'heure puis la date de la dernière modification et enfin la dernière colonne contient le nom de l'auteur de la modification. Les classes* ***A*** *et* ***B*** *ont été modifiées de manière rapprochée par Mr X, il semble donc que la modification de l'une ait un impact sur l'autre. Ces deux classes ont donc une forte chance d'appartenir au même composant. La classe* ***C*** *a été modifiée pratiquement à la même période mais par une personne différente. Elle n'est donc pas forcément liée aux deux classes précédentes. La classe* ***D****, enfin, a été modifiée à une date différente et seule. Elle semble donc isolée des autres et a donc une forte chance d'appartenir à un autre composant.*

Classe	Modification	Auteur
a	14 :50 :23 :09 :08	X
b	14 :55 :23 :09 :08	X
c	14 :52 :23 :09 :08	Y
d	17 :06 :20 :09 :08	Y

Table 5.1 – Informations intentionnelles disponibles à travers un outil de versionnement

5.1.2 Les informations contextuelles

Les informations contextuelles décrivent le contexte dans lequel le système doit être déployé, c'est-à-dire les limitations en termes de mémoire ou de communication, par exemple. Elles reflètent l'architecture matérielle à laquelle le système doit être adapté. A ce titre, elles donnent une image de l'architecture que doit respecter le système.

L'utilisation des informations contextuelles est peu répandue dans les travaux d'extraction d'architectures. En effet, l'architecture matérielle peut varier au cours de l'utilisation d'un système et, dans le cas général, elle impose peu de contraintes sur l'architecture logicielle. Cependant, dans le cadre d'une préparation à une phase d'évolution vers une architecture matérielle distribuée ou embarquée, les nouvelles contraintes contextuelles peuvent être fortes. Dans ce cas, leur prise en compte dès l'extraction de l'architecture permet de faciliter l'évolution future en proposant une représentation de l'architecture logicielle qui soit la plus adaptée à la future architecture matérielle.

Nous proposons d'utiliser les informations contextuelles pour rejeter les solutions qui sont incompatibles avec l'architecture matérielle prévue. Pour cela, nous devons convertir les propriétés du contexte en contraintes sur les éléments de l'architecture et donc sur les entités COA des solutions. Les informations contextuelles sont donc surtout négatives et portent sur l'ensemble des éléments architecturaux.

Par contre, à la différence des informations intentionnelles, les contraintes extraites des informations contextuelles portent uniquement sur les entités COA. Par exemple, les limitations en mémoire peuvent résulter en une contrainte sur la taille des contours mais elles n'ont pas d'impact sur les classes objets.

Ceci montre les limites de l'utilisation du contexte. En effet, comme l'extraction ne modifie pas le code source, l'adaptation à l'architecture matérielle est limitée par les propriétés des entités objets et ne permet pas d'obtenir une architecture trop éloignée de celle prévue à l'origine. Par exemple, l'occupation mémoire d'un contour de composant dépend surtout de la taille des classes qui le composent. Il est donc impossible de respecter des clauses strictes. L'adaptation à l'architecture matérielle est ainsi uniquement une première étape vers l'adaptation d'un système à une nouvelle architecture matérielle.

5.1.3 Influence des guides sur l'exploration

Le passage d'un espace de solutions à un espace de recherche peut être transparent. L'espace de recherche est alors celui des solutions. Cependant, face à un espace de solutions immense, il est possible de définir un espace de recherche qui soit strictement inclus dans l'espace des solutions. Nous utilisons donc les informations intentionnelles et contextuelles pour réduire l'espace de recherche selon deux méthodes : la suppression et le ciblage.

La suppression. Nous utilisons les informations disponibles pour éliminer les mauvaises solutions. Pour cela, nous devons éliminer de l'espace de recherche les solutions qui, en fonction des informations disponibles, ne peuvent pas être des solutions satisfaisantes au problème.

En effet, les éléments de l'espace des solutions n'ont pas tous la même qualité vis-à-vis des objectifs de l'architecte. Pour certains de ces éléments, il n'est pas utile de mesurer une quelconque fonction objectif pour voir que cette solution ne conviendra pas. Par exemple, l'élément qui associe chaque classe d'un système à un contour de composants et qui ne contient que des contours de connecteurs vides ne peut pas satisfaire l'architecte, quelque soit le résultat de la fonction objectif.

Nous proposons donc de définir un ensemble de contraintes que doivent vérifier les bonnes solutions. De plus, nous utilisons des contraintes hiérarchiques pour représenter la diversité des sources d'informations et leur pertinence tout aussi diverse.

Le ciblage. Les voisins d'un élément de l'espace des solutions sont déterminés par les opérations utilisées par le processus pour passer d'une solution à une autre. Des opérations simples définissent un voisinage réduit alors que des opérations complexes définissent un voisinage étendu.

Dans le cas d'un voisinage réduit, l'espace des solutions n'est pas uniforme. Il existe dans cet espace des zones qui sont plus ou moins pauvres en solutions correctes. En effet, si la notion de voisinage n'est pas très étendue, il est probable que le passage d'une solution à une autre ne modifie pas beaucoup la qualité. On peut donc supposer que, dans ce cas, le voisinage d'une bonne solution doit contenir plus de solutions intéressantes qu'une autre région de l'espace des solutions prise au hasard.

La méthode de réduction par ciblage utilise cette assertion. Elle vise à cibler une région plus intéressante de l'espace des solutions. Pour cela, l'exploration débute à partir d'une solution correcte au problème et, dans l'hypothèse d'un voisinage réduit, demeure dans le voisinage de cette bonne solution.

Nous proposons donc de définir différentes solutions correspondant à une ou plusieurs sources d'informations et représentant une solution satisfaisante.

5.1.4 Modélisations des informations intentionnelles et contextuelles

Les informations, quelles soient intentionnelles ou contextuelles, ne sont pas immédiatement disponibles pour guider notre processus d'extraction d'architectures. Pour résoudre cela, nous proposons deux modèles d'informations qui peuvent être instanciés à partir de la documentation, les recommandations et le contexte de déploiement.

Dans la suite, nous présentons chacun de ces modèles en fonction de la méthode de réduction qu'il permet. Nous montrons également comment il permet de réaliser la réduction. Par contre, nous présentons les transformations d'un modèle à l'autre dans la section 5.3.

5.1.4.1 Modèle pour le ciblage

Le ciblage d'un espace de recherche nécessite la définition d'une solution, c'est-à-dire une instance de notre modèle COA (*cf.* Figure 5.4). Cette solution marque le centre d'un sous-espace de l'espace des solutions qui possède un intérêt particulier du point de vue des informations intentionnelles.

Cependant, les informations disponibles ne sont pas suffisantes pour instancier facilement chaque élément du modèle COA. En effet, nous avons vu que les informations intentionnelles portent principalement sur les contours de composants et de configuration. Elles concernent rarement les contours de connecteurs. De plus, les informations contextuelles ne donnent pas d'informations supplémentaires sur les entités du système mais plutôt des contraintes sur les futures entités de l'architecture logicielle. Il est ainsi impossible d'utiliser ces informations pour instancier un quelconque élément du modèle COA. Par conséquent, le contexte ne permet pas de compenser les manques concernant les contours de connecteurs et d'interfaces.

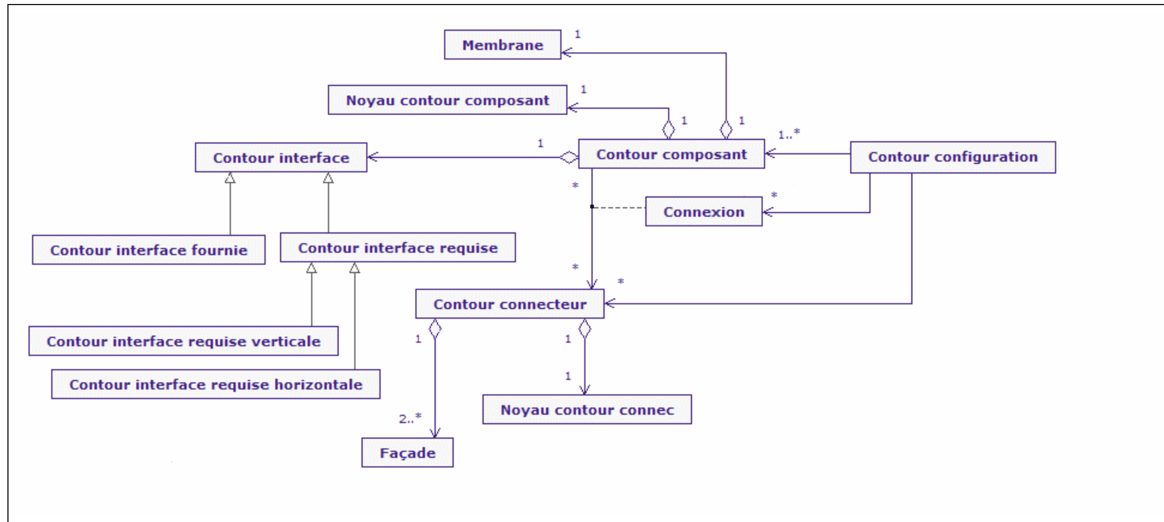


Figure 5.4 – Le modèle de correspondance Objet/Architecture

Ainsi, notre modèle d'informations pour le ciblage est un **sous-ensemble du modèle COA**. Ce sous-ensemble ne contient pas les entités du modèle pour lesquelles il est difficile d'obtenir des informations de types intentionnelles ou contextuelles : les contours d'interfaces, de connecteurs et de configuration. Notre modèle contient donc uniquement les contours de composants. Nous appelons architecture intentionnelle une instance de ce modèle.

Pour cibler une solution à partir d'une instance de ce modèle d'informations, nous devons proposer un ensemble de règles pour transformer cette instance en une instance du modèle COA. Ces règles de transformations sont simples. La première consiste à ajouter des contours de connecteurs vides entre chaque contour de composant qui communique à travers un appel de méthodes. La deuxième règle consiste à ajouter à chaque contour de composant, un contour d'interface par contour de connecteur avec, dans l'interface, le nom de l'appel de méthode. Enfin, la dernière transformation crée un contour de configuration correspondant à l'ensemble des contours de composants et de connecteurs ainsi que leurs connexions.

5.1.4.2 Modèle pour la suppression

La suppression d'éléments dans l'espace de recherche nécessite la définition d'un ensemble de critères déterminant le choix de l'élément à supprimer ou à conserver. Le respect de ces critères marque la validité d'une solution et entraine sa conservation dans l'espace de recherche. Par contre, le non-respect de ces critères provoque la suppression de l'élément concerné de l'espace de recherche.

Ces critères concernent des entités du modèle COA, objets ou les deux à la fois. En effet, les critères qui sont proposés par l'architecte peuvent concerner des classes (*e.g.* appartenance au même contour), des contours de composants (*e.g.* nombre de composants) ou encore les deux à la fois (*e.g.* taille des contours de connecteurs). Par contre, les critères issus du contexte concernent uniquement les relations entre entités COA et objets (*e.g.* taille des contours de composants).

Nous modélisons ces critères par un **réseau de contraintes**. Ainsi, notre modèle d'informations pour la suppression comporte une entité contrainte et trois sous-types :

- **les contraintes COA** décrivent les critères portant uniquement sur les entités COA ;

- **les contraintes objets** décrivent les critères portant uniquement sur les entités objets ;
- **les contraintes mixtes** décrivent les critères portant à la fois sur les entités COA et objets.

Pour tenir compte des diverses sources d'information, nous proposons de ne pas considérer toutes les contraintes au même niveau. La notion de niveau représente aussi le **niveau de confiance ou de pertinence** de la source de cette information. Ainsi, le réseau formé par les contraintes de notre modèle est, en fait, un **réseau hiérarchique**. Chaque contrainte possède donc un attribut reflétant son niveau dans la hiérarchie des contraintes. La résolution de ce réseau implique que les contraintes du niveau maximal soient toutes respectées et que le maximum des contraintes soient respectées dans les niveaux inférieurs.

5.2 Extraction d'architectures intentionnelles depuis la documentation

Comme nous l'avons vu, les informations contenues dans la documentation ne sont pas immédiatement disponibles pour un traitement automatique. Nous proposons donc un processus permettant d'extraire une vue de l'architecture intentionnelle à partir des différents documents que l'on prend en compte dans notre approche. Nous utilisons ensuite cette architecture intentionnelle pour réduire l'espace de recherche de notre processus d'extraction.

Cependant, nous avons montré, à travers les exemples de la section précédente, que l'extraction de l'architecture intentionnelle est difficile manuellement. Elle nécessite une profonde étude du système et des documents et il semble donc difficile de proposer un processus automatique et complet d'extraction des informations contenues dans les documents. En conséquence, nous présentons, dans la suite, un processus heuristique et semi-automatique basé sur les validations, vérifications et conseils de l'architecte en plus de la documentation.

Dans la suite, nous exposons d'abord les aspects de notre processus qui s'appliquent à tous les types de documents. Nous abordons ensuite, les aspects de notre processus spécifiques à chaque type de documents.

5.2.1 Processus de partitionnement

L'objectif de l'extraction de l'architecture intentionnelle peut être ramené à l'extraction d'une partition des classes du système à partir des documents. Cette partition peut ensuite être projetée sur le modèle d'informations de ciblage en associant à chaque partie un contour de composant.

Ce calcul de partition repose sur les mêmes étapes quelque soit le type de document concerné. Les aspects communs à ce calcul de partition sont donc les deux étapes de notre processus de partitionnement : une étape de regroupement puis une étape de partition proprement dite.

5.2.1.1 Étape de regroupement

L'étape de regroupement consiste en un regroupement hiérarchique des classes du système. Le principe de l'algorithme de regroupement hiérarchique (défini par S.C.Johnson [68]) est donné dans la figure 5.5 A. Au démarrage, l'algorithme de regroupement hiérarchique considère que chaque élément initial constitue un groupe (ligne 4). Ensuite, à chaque étape, les deux groupes les plus similaires (fonction $clusterProche$) sont regroupés dans un nouveau groupe (lignes 5 à 10). Le processus se termine lorsqu'il ne reste plus qu'un seul groupe. A partir de ce dernier groupe, on obtient une représentation en forme de dendrogramme de la hiérarchie des groupes (ligne 11). La figure 5.5 B présente la forme du résultat de cet algorithme. Dans cet exemple, les étapes du regroupement sont (a,c), (a,c,e), puis (b,d) et enfin (a,b,c,d).

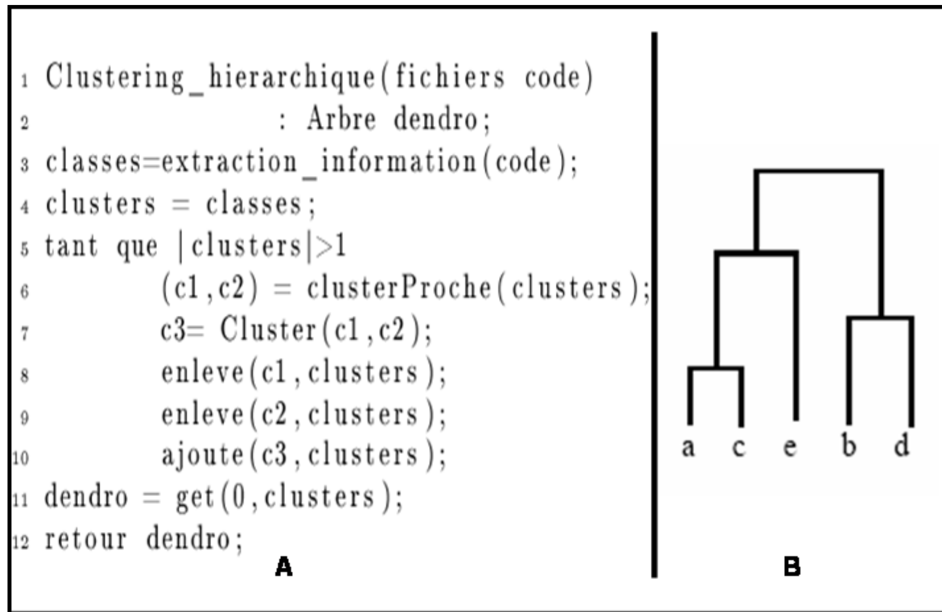


Figure 5.5 – Algorithme de regroupement hiérarchique

Nous exploitons cet algorithme pour regrouper les classes objets du système en utilisant une fonction de similarité spécifique à chaque type de documents. Cette fonction permet de mesurer le niveau de dépendances ou de collaboration entre deux classes. La valeur de cette fonction est donc évaluée sur chaque paire de classes à l’initialisation du processus. Par la suite, la similarité entre deux groupes est la moyenne de leurs similarités.

5.2.1.2 Étape de partitionnement

Pour obtenir une partition des classes du système, on doit sélectionner des nœuds dans la hiérarchie résultant de l’étape précédente. Cette sélection est réalisée par l’algorithme détaillé dans la figure 5.6 qui sélectionne les nœuds présentant la plus forte similarité.

L’algorithme de sélection des nœuds consiste en un parcours en profondeur du dendrogramme (paramètre $dendro$). A chaque nœud, on compare la valeur de similarité S des éléments qui constituent ce groupe avec celle de ses deux fils (ligne 9). Si la valeur de similarité de ce nœud est inférieure à la moyenne de ses deux fils, alors l’algorithme traite le premier fils (lignes 12 et 13), sinon le nœud courant est identifié comme un contour et ajouté à la partition (variable R, ligne 10) et l’algorithme traite ensuite le nœud suivant.

5.2.1.3 Processus de validation des informations

L’architecte peut intervenir à différents moments durant le calcul de la partition des classes : avant la première étape, entre les deux étapes et après la deuxième étape. Chacune de ces interventions a une influence directe sur la confiance accordée au résultat de l’extraction (*cf.* Figure 5.7) :

- il peut **rejeter les informations** en considérant qu’elles ne sont pas suffisamment fiables pour être utilisées ;

```
1  Selection_clusters(arbre dendro)
2                   : Partition R;
3  Pile parcoursClusters;
4  empile(racine(dendro),parcoursClusters);
5  tant que !vide(parcoursClusters)
6           Cluster pere=depile(parcoursClusters);
7           Cluster f1=fils1(pere,dendro);
8           Cluster f2=fils2(pere,dendro);
9           si S(pere)>moyenne(S(f1,f2))
10                  ajouter(pere,R);
11          sinon
12                  empile(f1,parcoursClusters);
13                  empile(f2,parcoursClusters);
14 retour R;
```

Figure 5.6 – Algorithme de partitionnement du dendrogramme

- il peut **ignorer les informations**. C'est le choix par défaut si aucun architecte n'est disponible. Dans ce cas, il ne souhaite pas s'exprimer à ce niveau et ne prend aucune décision ;
- Il peut également **influencer directement le processus**. Avant l'étape de regroupement, il peut modifier le calcul de similarités entre les classes. Par exemple, il peut évaluer que la similarité entre deux classes est plus forte que celle mesurée ou inversement. Avant l'étape de partitionnement, il peut également modifier le dendrogramme résultat pour l'ajuster à ces attentes. Il peut ainsi modifier les relations hiérarchiques entre les groupes et donc les liens entre les classes. Enfin, après le partitionnement, il peut modifier la partition résultat pour l'ajuster à ses attentes. Il peut ainsi modifier les contours de composants ou déplacer des méthodes vers les contours de connecteurs pour signaler les connecteurs qu'il suppose ;
- il peut simplement **valider les résultats** de l'étape et accorder une confiance maximale au mode de calcul automatique.

Cette confiance comporte trois niveaux. Le premier correspond à une absence de confiance, il est atteint si l'architecte rejette les informations. Le second niveau est le niveau par défaut. Il correspond aux informations ignorées par l'architecte. Enfin, le dernier niveau correspond à une confiance totale. Elle est atteinte lorsque l'architecte valide ou modifie les résultats du processus d'extraction.

5.2.2 Mesure de similarité dans la documentation

Si l'algorithme d'extraction de l'architecture intentionnelle est le même pour tous les types de documents, la différence provient de la métrique utilisée dans le déroulement du processus. Selon les types de documents, la proximité entre les classes se mesure de manière différente et donne lieu à des mesures de similarités très diverses.

Nous proposons, ici, les mesures de similarité concernant les trois types de documents que nous avons sélectionnés : les diagrammes UML, le code source et les outils de versionnement.

Il est clair à travers notre démarche que l'on peut aisément prendre en compte un nouveau type de documents en proposant simplement une mesure de similarité spécifique. Ceci rend notre approche

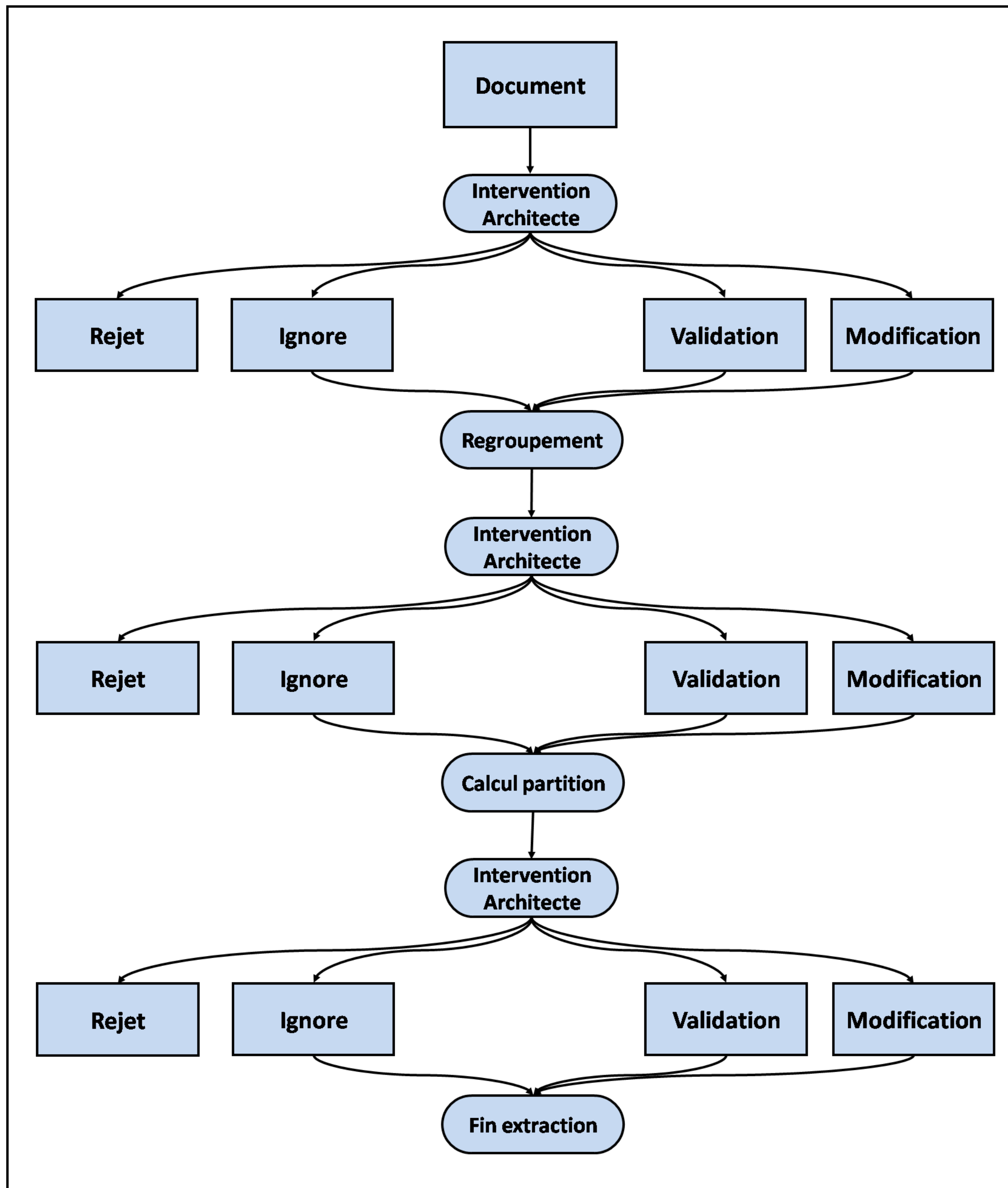


Figure 5.7 – Les étapes de la validation de l'extraction d'information par l'architecte

facilement extensible.

5.2.2.1 La similarité dans les diagrammes UML

Face à la richesse de ce type de document, nous concentrons nos efforts sur l'extraction des informations concernant les classes et leur relations. Pour cela, nous sommes naturellement amenés à prendre en considération les objets, qui sont directement liés aux classes, et les cas d'utilisation qui sont liés aux classes ou aux objets à travers les fonctionnalités représentées par les différents diagrammes.

Nous utilisons donc les diagrammes qui comportent les entités qui nous intéressent pour établir un graphe de dépendances entre ces entités. Nous créons ensuite un graphe de dépendances entre les classes. Enfin, nous utilisons ce graphe pour mesurer la similarité entre les classes.

Les diagrammes UML. Nous concentrons nos efforts sur une partie seulement des diagrammes UML. En effet, les diagrammes UML sont variés et traitent chacun un aspect de la modélisation du logiciel. Ainsi, les entités présentes dans les diagrammes sont diverses et ils ne contiennent pas tous l'ensemble des entités qui nous intéressent. Par conséquent, nous utilisons les six types de diagrammes suivants :

- **les diagrammes de classes et objets** : ces diagrammes contiennent respectivement des informations sur les classes et les objets. En effet, un diagramme de classes (ou objets) est une représentation graphique de la vue statique qui montre une collection d'éléments statiques (dynamiques) du modèle du système, comme des classes, des types et leurs contenus et relations (des objets et leurs relations). Ces diagrammes contiennent également certains éléments comportementaux réifiés, comme des opérations ;
- **les diagrammes de cas d'utilisation** : ces diagrammes contiennent des informations sur les cas d'utilisation. En effet, ils modélisent la fonctionnalité d'un système telle qu'elle est perçue par un ou plusieurs intervenants extérieurs, appelées acteurs. Le diagramme liste les acteurs et les cas d'utilisation et montre les acteurs participant à chaque cas d'utilisation. Il montre également les relations entre les cas d'utilisation, en particulier les relations d'inclusion ;
- **les diagrammes de séquences** : ces diagrammes contiennent des informations sur les objets et les cas d'utilisation. Ils montrent un ensemble de messages organisés en séquences temporelles. Chaque rôle est présenté par une ligne de vie qui représente le rôle dans le temps à travers l'interaction globale. Les messages apparaissent sous forme de flèches entre les lignes de vie. Ce diagramme illustre un scénario correspondant à un cas d'utilisation ;
- **les diagrammes de communication** : comme les diagrammes de séquences, ces diagrammes contiennent donc des informations sur les objets et les cas d'utilisation. Ils illustrent les mêmes interactions que les diagrammes de séquence. Cependant, ils s'attachent à montrer un aspect différent de l'aspect temporelle proposé par les diagrammes de séquence. Ainsi ils s'attachent plus particulièrement aux relation entre les rôles. Ceci sont donc représentés par des rectangles alors que les messages sont associés à des arcs entre les rectangles. L'ordre des messages est alors représenté par une numérotation des arcs ;
- **les diagrammes de collaborations** : il contient des informations sur les classes, les objets et les cas d'utilisation. Il affiche la définition d'une collaboration. Il décrit les entités impliquées dans la réalisation d'une fonctionnalité associée à un cas d'utilisation.

La mesure de similarité. La mesure de la similarité dans les diagrammes UML repose sur le graphe de dépendances entre les classes. Ce graphe est calculé à partir du graphe de dépendances entre entités objets. La similarité entre deux classes a et b est égale à la pondération de l'arête $\{a, b\}$ reliant ces deux classes dans le diagramme de dépendances entre les classes.

Le graphe des dépendances entre entités objets. Le graphe des dépendances est un graphe non-orienté et pondéré $G_a\,(V, E, w)$, où l'ensemble des sommets V contient toutes les entités objets qui nous intéressent, *i.e.* classe, objet et cas d'utilisation, et l'ensemble des arêtes E contient une arête $\{a, a'\}$ si les entités a et a' sont liées dans un diagramme.

La fonction w assigne un poids à chaque arête. Ce poids est le nombre de diagrammes où les entités, représentés par les sommets des arêtes, sont liées. L'architecte peut influer sur cette pondération en attribuant un poids aux différents diagrammes. Il peut ainsi modifier l'importance relative des diagrammes.

Deux entités sont liées dans un diagramme s'il existe dans le diagramme une arête entre les représentations des deux entités. Il existe également un lien entre deux entités si la représentation de l'une est incluse dans la représentation de l'autre dans un diagramme. Enfin, il existe un lien, représenté par une arête de poids nul, entre un objet et la classe dont elle est instance.

Le graphe de dépendance entre classes. Le graphe des dépendances des classes est un graphe non-orienté et pondéré $G_c\,(V, E, w)$, où l'ensemble des sommets V contient toutes les classes, et l'ensemble des arêtes E contient une arête $\{a, a'\}$ s'il existe un chemin entre les sommets a et a' dans G_a. La fonction w assigne un poids à chaque arête. Ce poids est la somme de tous les chemins reliant les deux sommets. Cette valeur représente la force des dépendances entre les deux classes dans l'ensemble des diagrammes UML pris en compte. De plus, les choix de l'architecte sur le poids à affecter à chaque diagramme a une influence directe sur la similarité des classes.

5.2.2.2 La similarité dans les fichiers sources

Les commentaires et les noms des entités qui constituent le code source sont choisis avec attention par les programmeurs. Les commentaires illustrent les fonctionnalités associées à chaque entité alors que les noms des entités reflètent leur utilisation. Ces informations nous informent sur la sémantique associée aux classes et nous permettent de les regrouper en fonction de la similarité de la sémantique associée à chacune.

La mesure de similarité dans le code source est définie selon une analyse de sémantique latente (*Latent Semantic Analysis*, LSA) [79]. C'est une technique mathématique et statistique, totalement automatique, pour l'extraction et l'inférence de relations d'usage conceptuel de mots dans un ensemble de textes. Elle a déjà été utilisée avec de bons résultats dans plusieurs approches qui s'appuient uniquement sur la sémantique embarquée dans le code source pour documenter [35] ou proposer une vue architecturale d'un système [76, 126].

Les principes de LSA. La première étape consiste à modéliser le corpus de textes par une matrice des occurrences dans laquelle chaque ligne représente un terme unique et chaque colonne représente un des textes du corpus. Chaque cellule contient la fréquence d'apparition du mot de la ligne dans le texte de la colonne. Cette fréquence peut être pondérée par une fonction exprimant l'importance d'un terme dans un passage ou un domaine particulier.

Dans la deuxième étape, LSA décompose la matrice selon un processus de décomposition en valeurs singulières (*singular value decomposition*, SVD). Selon cette décomposition, une matrice rectangulaire X est décomposée en un produit de trois autres matrices (*cf.* Equation 5.1). Les matrice W et P décrivent respectivement les entités des lignes et des colonnes de la matrice initiale comme des vecteurs orthogonaux : ceux sont les matrices des vecteurs propres « termes » et « documents ». La troisième matrice S est une matrice diagonale contenant les valeurs singulières de M.

$$\{X\} = \{W\}\{S\}\{P\}' \tag{5.1}$$

La multiplication des trois matrices issues de la décomposition permet de reconstruire la matrice M originale. Mais, lorsqu'on sélectionne les k plus grandes valeurs singulières, ainsi que les vecteurs singuliers correspondants dans W et P, on obtient une approximation de rang k de la matrice des occurrences. En faisant cette approximation, les vecteurs « termes » et « documents » sont traduits dans l'espace des « concepts ». Le nombre k des coefficients conservés peut être choisi par l'utilisateur ou calculé selon les dimensions de la matrice des occurrences.

Du code source à la matrice des occurrences. Afin d'utiliser l'algorithme LSA, nous devons d'abord générer la matrice des occurrences à partir du code source. Chaque fichier représente un document et chaque mot du fichier représente un terme de la matrice des occurrences. Cependant, cette correspondance entre les mots du fichier et les termes de la matrice des occurrences n'est pas directe. En fait, nous procédons à deux étapes pour augmenter la pertinence des termes extraits :

- **le filtrage** : pour réduire le bruit parmi les termes extraits, nous procédons à un filtrage des mots du fichier. Tous les fichiers d'un système utilisant le même langage, nous filtrons les mots réservés du langage. Le bruit est également introduit par les noms de variables trop communs. Pour limiter cet effet, nous filtrons les mots contenus dans tous les fichiers. Nous filtrons également les mots contenant seulement des caractères de ponctuation ou ceux contenant seulement un caractère ;
- **la césure** : les noms de variables sont, la plupart du temps, constitués de plusieurs mots possédant chacun une part d'information. L'utilisation directe des noms de variables nous empêche donc d'identifier les variables qui sont similaires. Par exemple, *contains_key* et *key_contains* sont similaires et contiennent les mêmes mots cependant les noms de ces deux variables sont différents. Pour éviter ceci, les noms des variables sont coupés en leurs différents constituants en utilisant les caractères tels que les soulignés, les tirets, les capitales (comme dans «replaceNamespaceSep») ou encore les chiffres.

Après ces deux étapes, la matrice des occurrences peut être construite avec les termes extraits et les fichiers de classes. La fréquence de chaque cellule est calculée selon l'équation suivante, où t est un terme, d un document, tf_t est la fréquence du terme t dans le document d, $|D|$ est le nombre total de documents et $|\{t \in d\}|$ est le nombre de documents contenant le terme t :

$$w_{t,d} = tf_t \cdot log \frac{|D|}{|\{t \in d\}|} \tag{5.2}$$

De la matrice des occurrences à la mesure de similarité Nous appliquons LSA à la matrice des occurrences que nous venons de présenter. Nous avons choisi de conserver k coefficients, où k est calculé selon les dimensions de la matrice $(n * m)$ comme : $k = (n * m)^{0.2}$.

Le résultat de LSA nous permet de calculer une mesure de similarité entre deux classes j et q. La similarité est basée sur le cosinus de l'angle θ formé par le vecteur correspondant à chaque document représentant j et q dans la matrice, $\vec{d_j}$ et $\vec{d_q}$. Ce cosinus est calculé suivant la formule suivante :

$$cos\, \theta = \frac{\vec{d_j} \cdot \vec{d_q}}{\left\|\vec{d_j}\right\| \cdot \left\|\vec{d_q}\right\|} \tag{5.3}$$

Exemple 5.4 (Calcul de la similarité entre méthodes java).

```java
final String getSignature(final Method method, final String methodName) {
    if (signatureCache.containsKey(method)) {
        return (String) signatureCache.get(method);
    } else {
        String signature = methodName + "(";
        final Type[] params = method.getArgumentTypes();
        for (int p = 0; p < params.length; p++) {
            if (p > 0) signature += ", ";
            signature += replaceNamespaceSep(sig2cn(params[p].getSignature()));
        }
        signature += ")";
        signatureCache.put(method, signature);
        return signature;
    }
}
```

```java
public static String getAsString(final Invocation invocation, final int id)
    return new StringBuffer("(Invocation FM").
            append(id).
            append("\")\n)\n\n").toString();
}
```

```java
protected static String replaceNamespaceSep(final String name) {
    if (rns.containsKey(name)) {
        return (String) rns.get(name);
    } else {
        String n = name.replaceAll("\\.", "::");
        rns.put(name, n);
        return n;
    }
}
```

Figure 5.8 – Exemples de méthodes Java

La figure 5.8 présente trois méthodes extraites de classes Java. En considérant que chaque méthode représente une classe et un fichier différent, nous souhaitons appliquer LSA pour calculer la similarité entre ces méthodes. Pour cela, nous appliquons d'abord les deux étapes qui nous permettent de générer la matrice des occurrences. Ainsi, après le filtrage des mots clefs de Java et la césure des termes présents dans les documents, nous obtenons une matrice des occurrences de 26 termes et 3 documents (cf. Tableau 5.2).

L'application de LSA sur cette matrice des occurrences a pour résultat une matrice reconstruite, représentée par le tableau 5.3. Cette matrice fait apparaitre des relations entre les termes qui ne sont pas présentes dans la première matrice. LSA semble donc avoir fait apparaitre des relations sémantiques entre les termes qui n'étaient pas visibles de prime abord.

Enfin, nous utilisons la matrice reconstruite pour calculer les mesures de similarité entre les trois méthodes. Pour cela, nous calculons le cosinus de l'angle formé par les vecteurs représentant chaque document. La mesure de la similarité est alors résumée dans le tableau 5.4

terme	Docu 1	Docu 2	Docu 3
replace	0.17	0	0.35
namespace	0.17	0	0.17
sep	0.17	0	0.17
get	0.70	0.17	0
method	3.82	0	0
name	0.35	0	$O.85$
signature	5.25	0	0
cache	1.43	0	0
contains	0.17	0	0.17
key	0.17	0	0.17
type	0.48	0	0
params	1.43	0	0
argument	0.48	0	0
types	0.48	0	0
length	0.48	0	0
sig2cn	0.48	0	0
put	0.17	0	0.17
as	0	0.48	0
invocation	0	0.96	0
id	0	0.96	0
buffer	0	0.48	0
fm	0	0.48	0
append	0	1.92	0
to	0	0.48	0
rns	0	0	1.44
all	0	0	0.48

Table 5.2 – La matrice des occurrences

terme	Docu 1	Docu 2	Docu 3
replace	0.17	0.05	0.49
namespace	0	0	0.02
sep	0.35	0	0
get	0.17	0.09	0.97
method	0	0	0.05
name	0.17	0	O
signature	0.17	0.09	0.95
cache	0.01	0.01	0.14
contains	0.17	0	0
key	0.7	0.04	0.48
type	0.17	0	0.05
params	0	0	0
argument	3.82	0.01	0.48
types	0	0	0.05
length	0	0	0
sig2cn	0.35	0.19	1.90
put	0	0	0.05
as	0.85	−0.01	0
invocation	5.25	0	0.48
id	0	0	0.05
buffer	0	0	0
fm	1.43	−0.01	0
append	0	0.14	1.44
to	0	0	0
rns	0.17	0	0.02
all	0	0.05	0.48

Table 5.3 – La matrice reconstruite

	Méthode 1	Méthode 2	Méthode 3
Méthode 1	1	0.31	0.44
Méthode 2	0.31	1	0.71
Méthode 3	0.44	0.71	1

Table 5.4 – Mesure de similarité entre les méthodes exemples

La mesure de similarité est maximale entre les méthodes 2 et 3. L'étude détaillée des trois méthodes permet de comprendre ces résultats. Les trois méthodes réalisent des traitements sur des chaines de caractères. Cependant, les deux dernières méthodes travaillent sur des chaines de caractères quelconques alors que la première travaille sur des entités plus spécifiques : des chaines de caractères représentant la définition d'une méthode.
Ainsi, la première méthode utilise une approche légèrement différente des deux dernières. Cette différence n'est pas évidente à première vue. Mais l'application de LSA nous a permis de mettre en évidence cette plus grande similarité entre les deux dernières méthodes qu'entre ces méthodes et la première.

5.2.2.3 La similarité dans les archives des outils de versionnement

Les outils de versionnement archivent de nombreuses informations sur le système dont ils encadrent le développement ou la maintenance. Au delà des différentes versions du système, ces outils archivent aussi les dates, les auteurs et les cibles des modifications. Ces informations permettent de suivre les modifications apportées à un moment donné à un ensemble de classes par une personne.

Or, lorsqu'un programmeur doit ajouter une fonctionnalité ou apporter une modification, il modifie de manière rapprochée l'ensemble des classes concernées par cette modification. Ainsi, les informations archivées par les outils de versionnement nous permettent de mesurer la similarité des classes en fonction des dates et des auteurs des modifications qu'elles ont subies.

Beyer présente un outil de visualisation reposant sur une mesure de similarité utilisant les informations de dates et d'auteurs stockés par les outils de versionnement [17]. Cette mesure s'appuie avant tout sur un modèle des changements communs des artefacts logiciels appelé graphe condensé de co-changement (*condensed co-change graph*) qui est extrait des outils de versionnement du système.

Le graphe condensé de co-changement d'une version d'un répertoire, est un graphe non-dirigé et pondéré (V, E, w), où l'ensemble des sommets V contient tous les artefacts logiciels du répertoire, et l'ensemble des arêtes E contient une arête $\{a, a'\}$ si les artefacts a et a' sont modifiés en même temps au cours d'une transaction de changement. La fonction w assigne un poids à chaque arête. Ce poids est le nombre de fois où les artefacts, représentés par les sommets des arêtes, sont modifiés de manière simultanée.

Nous utilisons les travaux de Beyer pour mesurer la similarité des classes dans les outils de versionnement. Pour cela, nous utilisons le graphe condensé de co-changement des fichiers des classes d'un système.

La mesure de similarité entre deux classes se calcule à partir des pondérations du graphe de co-changement. Ainsi, la similarité de deux classes a et a' est le poids associé à l'arête $\{a, a'\}$, c'est-à-dire $w\,(a, a')$.

5.3 Création d'un réseau de contrainte hiérarchique

Nous avons montré comment utiliser la documentation et les recommandations de l'architecte pour obtenir une vue de l'architecture intentionnelle. Cette architecture intentionnelle permet de réduire l'espace de recherche en ciblant un sous-ensemble de l'espace des solutions.

Ce ciblage n'est pas, comme on l'a vu, le seul moyen de réduire l'espace de recherche de notre processus. Afin de pouvoir supprimer des éléments de l'espace de recherche, nous devons proposer un processus de génération d'un réseau de contraintes hiérarchiques conformément à notre modèle d'informations pour la suppression. Pour cela, il nous faut déterminer les sources d'informations qui nous permettent de définir les contraintes, mais aussi celles qui nous permettent d'établir le niveau de la contrainte dans la hiérarchie du réseau.

Nous présentons, dans la suite, comment nous utilisons les recommandations de l'architecte pour définir une partie des contraintes de notre réseau. Ensuite, nous étudions les mécanismes qui nous permettent de définir des contraintes à partir des architectures intentionnelles extraites de la documentation. Enfin, nous exposons le processus de génération du réseau de contraintes proprement dit à partir des contraintes établies.

5.3.1 Les recommandations de l'architecte

La première source d'information lors de l'extraction est l'architecte. Il connait l'environnement de déploiement du système, les raisons de l'extraction d'architectures ainsi que les objectifs des prochaines phases de maintenance. De plus, selon son expertise, il possède des connaissances plus ou moins profondes sur le système qui peuvent lui permettre d'avoir une vue partielle de l'architecture.

Nous souhaitons utiliser ces informations pour définir un ensemble de contraintes permettant de réduire l'espace de recherche en supprimant les solutions qui ne les respectent pas. L'objectif est d'utiliser les connaissances de l'architecte pour éliminer les solutions qui sont, de manière évidente pour lui, incorrectes et en favorisant les solutions qui lui semblent correctes.

Pour cela, nous étudions les informations qu'il peut nous offrir du point de vue des intentions des concepteurs et du contexte de déploiement. Nous présentons ensuite la méthode de collecte que nous utilisons pour récupérer les recommandations de l'architecte.

5.3.1.1 Informations contextuelles

Nous avons vu les atouts que représentent les informations contextuelles dans le cadre de l'extraction d'architectures. Cependant, l'obtention de ces informations n'est pas facile. Il existe des outils qui permettent d'obtenir les éléments du contexte sous différentes formes manipulables par ordinateur, mais ces informations ne peuvent pas être utilisées dans notre approche. En effet, comme l'extraction porte sur un système existant, nous ne pouvons pas le modifier totalement pour obtenir une architecture respectant des limitations liées au contexte trop précises. Ainsi, nous ne pouvons pas garantir la taille mémoire représentée par un composant extrait puisque cette taille dépend directement de la taille des classes du contour de composants.

Afin d'utiliser les informations contextuelles, nous utilisons l'architecte pour fournir des informations portant sur le contexte et utilisables dans notre approche. Pour cela, nous demandons à l'architecte de spécifier un ensemble d'informations portant sur les entités COA et objets et représentant les limites liées au contexte auxquelles le système sera confronté. Il doit donc traduire les limites du contexte en limites sur les entités COA et objets. Par exemple, les limitations de mémoire seront traduites en limitation

sur le nombre de classes ou méthodes par contour. Les limitations en termes de communications peuvent être traduites en un nombre maximum de contours de connecteurs ou en une contrainte sur le diamètre des appels de méthodes dans les contours de connecteurs.

5.3.1.2 Informations intentionnelles

Pour permettre à l'architecte de fournir toutes les informations dont il dispose, notre approche doit prendre en compte toutes informations partielles qu'il possède. Pour cela, notre processus doit accepter les propriétés positives et négatives, c'est-à-dire les propriétés qui décrivent respectivement une relation entre deux entités ou l'absence d'une telle relation. Par exemple, l'architecte peut nous indiquer que deux classes doivent appartenir au même contour de composant (information positive) ou qu'elles ne doivent pas appartenir au même contour (information négative).

Cependant, en fonction de l'expertise de l'architecte, nous pouvons aussi obtenir des informations plus complètes. Ainsi, l'architecte peut connaitre les classes du système qui constituent un composant ou les méthodes qui constituent un connecteur. Il peut même indiquer une architecture du système qui lui semble correcte. Dans ce cas, l'objectif de notre processus est d'affiner sa proposition.

5.3.1.3 Collecte des recommandations

Les recommandations de l'architecte peuvent concerner différents types d'informations. De la même manière, elles peuvent prendre plusieurs formes selon le type d'informations qu'elles portent : informations contextuelles, informations intentionnelles partielles, informations intentionnelles complètes. Cependant, toutes ces formes sont des contraintes selon notre modèle d'information pour la suppression. De plus, le niveau de confiance de chacune de ces contraintes est par défaut de 2, c'est-à-dire le niveau maximum de confiance, puisque l'architecte les impose. Cependant, nous permettons à l'architecte de modifier le niveau de confiance pour souligner une hésitation.

Informations contextuelles. Les limites que l'architecte définit sur les entités COA et objets permettent de détecter les solutions qui ne sont pas correctes du point de vue du contexte. Pour permettre cette détection et faciliter la collecte de ces informations, nous proposons de définir ces limites à travers des contraintes portant sur les entités COA et objets.

Les limites de chaque entité COA s'expriment, en effet, facilement en termes de contraintes portant à la fois sur les entités COA et les éléments objets. Par exemple, les limites en nombres de classes par contour de composants sont collectés sous la forme d'une contrainte impliquant les contours de composants et les classes objets et limitant le nombre de classes par contour de composants.

Les informations contextuelles sont fournies par l'architecte en définissant des contraintes mixtes de notre modèle d'informations pour la suppression.

Informations intentionnelles complètes. Nous utilisons d'abord les informations complètes proposées par l'architecte pour orienter notre processus vers une solution correspondant à ces attentes et correcte selon nos autres guides. Si ces informations sont suffisamment conséquentes (proposition d'architecture), nous utilisons notre processus uniquement pour améliorer la solution proposée.

Ces informations intentionnelles complètes définissent des entités COA que l'architecte attend dans l'architecture résultant de l'extraction. Notre processus d'exploration doit donc veiller au respect des recommandations de l'architecte en favorisant les solutions comportant ses entités. Pour cela, nous proposons de définir ces informations à travers des contraintes portant sur les entités objets. En effet, chaque entité COA peut être définie comme une contrainte sur les entités objets. Ainsi, on crée une contrainte

imposant aux éléments objets de l'entité COA d'appartenir à la même entité COA. Par exemple, si l'architecte souhaite un contour de composants, qui contient les classes a et b, on crée une contrainte sur a et b qui impose aux deux classes d'appartenir au même contour de composants.

Les informations intentionnelles complètes sont fournies par l'architecte en définissant des contraintes objets de notre modèle d'informations pour la suppression.

Informations intentionnelles partielles. Nous utilisons les informations partielles que l'architecte fournit pour détecter les solutions qui ne sont pas correctes de son point de vue. Pour permettre cette détection et faciliter la collecte de ces informations, nous proposons de définir ces informations au travers de contraintes portant sur les entités COA ou sur les entités objets.

L'architecte peut en effet proposer des informations nécessitant la définition de contraintes portant sur les entités objets ou les entités COA. Par exemple, si deux méthodes doivent appartenir à un connecteur, nous définissons une contrainte portant sur les deux méthodes et imposant qu'elles appartiennent à un seul contour de connecteur. Par contre, si l'architecte veut signaler un nombre maximum de composants dans l'architecture, il doit définir une contrainte portant sur le nombre de contours de composants dans la solution.

De plus, les propriétés affirmatives ou négatives s'expriment facilement en termes de contraintes portant sur les entités objets ou COA. Par exemple, si deux classes a et b doivent appartenir au même contour de composant, l'architecte soumet une contrainte sur l'ensemble de classes $\{a, b\}$ imposant l'égalité entre les contours de chaque élément de l'ensemble. Inversement, si deux classes a et b doivent appartenir à des contours différents, l'architecte soumet une contrainte sur l'ensemble de classes $\{a, b\}$ imposant une inégalité entre les contours de chaque élément de l'ensemble.

5.3.2 Les contraintes issues des architectures intentionnelles

Les informations issues de la documentation donnent une vue partielle de l'architecture du système. A ce titre, nous avons montré comment elles permettent directement de définir des solutions au problème d'extraction et de les utiliser comme point de départ pour l'exploration. Cependant, les informations issues de la documentation ne sont pas des contraintes.

Pour pouvoir utiliser l'architecture intentionnelle extraite de la documentation, nous devons transformer les entités du modèle d'information pour le ciblage en entités du modèle de transformation pour la suppression.

L'architecture intentionnelle est un ensemble de contours de composants. C'est donc avant tout une partition des classes objets. De manière similaire à la méthode utilisée pour les informations intentionnelles complètes de l'architecte, nous transformons d'abord chaque contour de composant en une contrainte. Cette contrainte porte sur l'ensemble des classes du contour et impose que les classes appartiennent au même contour de composant.

Cette contrainte est cependant insuffisante pour représenter parfaitement la partition. En effet, une architecture intentionnelle de deux contours de deux classes chacun donne deux contraintes de ce type. Mais, la reconstruction de l'architecture intentionnelle, à partir des contraintes, n'est pas unique puisque ces deux contraintes sont vérifiées à la fois par l'architecture intentionnelles initiale et par une architecture composée d'un seul contour de composant réunissant toutes les classes.

Pour éviter cela, nous définissons une autre contrainte à partir de l'ensemble des contours de composants. Nous devons ajouter une contrainte qui interdit à deux classes issues de contours différents d'appartenir au même contour. Pour cela, nous créons une contrainte de ce type entre chaque paire de classes appartenant à des contours différents.

Pour compléter la transformation entre les deux modèles nous devons définir le niveau de confiance de chaque contrainte. Pour cela, nous utilisons les interventions de l'architecte pour juger de la confiance qu'il porte à l'architecture intentionnelle (*cf.* Figure 5.9).

Si l'architecte rejette les informations à n'importe quelle étape le document n'est pas utilisé et possède un niveau de confiance nul. Les autres interventions permettent d'obtenir une architecture intentionnelle et donc un ensemble de contraintes. La première validation donne le niveau de confiance initiale pour le document et donc l'architecture. Si l'architecte ignore les informations, le niveau de confiance est nul, c'est-à-dire non vérifié. Par contre s'il valide les informations, le niveau de confiance est de 1, c'est-à-dire que les informations sont validées par l'architecte. Enfin, s'il modifie les informations, le niveau de confiance est de 2, c'est-à-dire que les informations sont proposées par l'architecte et que le niveau de confiance est maximum.

Les étapes suivantes de validation modifie le niveau de confiance initiale. Si l'architecte ignore une étape de validation le niveau de confiance diminue de 1. S'il valide une étape de validation, le niveau de confiance reste identique. Enfin, si l'architecte modifie les informations le niveau de confiance augmente au niveau 2.

5.3.3 Combinaison des contraintes

Les contraintes que nous avons extraites des recommandations de l'architecte, du contexte et de la documentation doivent être assemblées pour former le réseau de contraintes hiérarchiques, instance du modèle d'information de suppression (*cf.* Figure 5.10).

L'assemblage de l'ensemble des contraintes nous donne un réseau hiérarchique, où la hiérarchie est déterminée par le niveau de confiance de chaque contrainte dans le réseau. Cependant, étant donné la diversité des sources des contraintes il est possible qu'il existe des conflits entre les contraintes rendant le réseau inconsistant. Il nous faut donc assurer la consistance de ce réseau.

Pour assurer cette consistance, nous procédons par niveau. Ainsi, nous vérifions d'abord la consistance de chaque niveau. Si deux contraintes sont en conflit, l'architecte peut, s'il est disponible, choisir celle à conserver. Si l'architecte n'est pas disponible, nous supprimons l'une des deux au hasard.

Nous vérifions ensuite la consistance entre les niveaux. Si deux contraintes, de niveaux différents, sont en conflits, nous supprimons la contrainte du plus bas niveau puisque c'est la contrainte qui a le moins de confiance.

5.4 Conclusion

Dans ce chapitre, nous avons étudié les guides auxiliaires de l'extraction. Ces guides permettent d'étendre les sources d'informations utilisées par le processus d'extraction au-delà du code source. L'architecture reflète ainsi le système dans son ensemble et non plus simplement son implémentation.

Nous avons d'abord proposé **deux modèles d'informations** permettant d'utiliser les informations intentionnelles et contextuelles pour guider l'exploration. Ainsi, les instances de ces modèles permettent de diriger l'extraction en réduisant l'espace de recherche. Cette réduction repose à la fois sur une suppression des solutions évidement incorrectes et sur le ciblage des solutions probablement bonnes.

Nous avons ensuite proposé un processus d'**extraction des informations intentionnelles et contextuelles**. Cette extraction est faite depuis trois types de documents que sont les diagrammes UML, les fichiers sources et les archives des outils de versionnement.

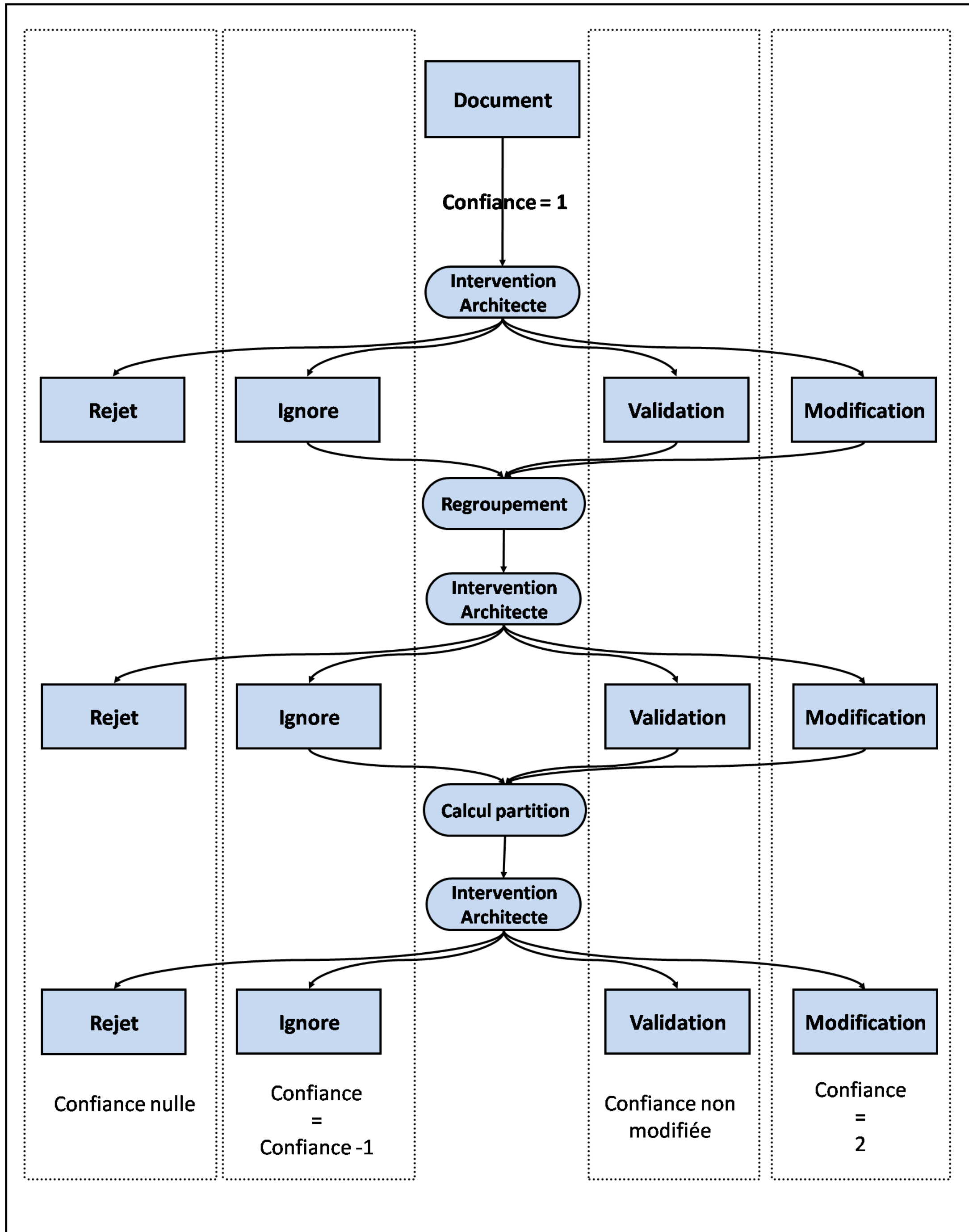


Figure 5.9 – Le niveau de confiance selon les interventions de l'architecte

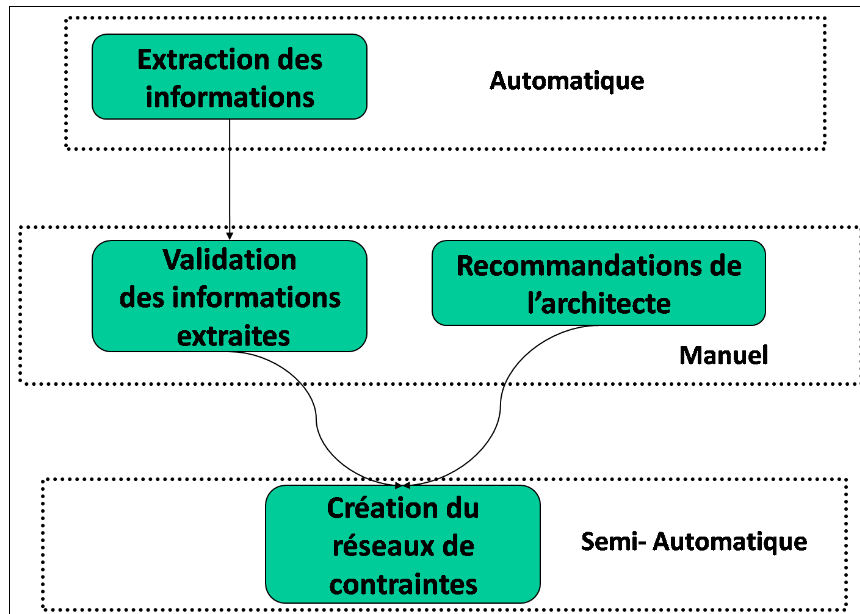


Figure 5.10 – Utilisation de la documentation et des recommandations

L'ensemble du processus d'extraction de l'architecture intentionnelle et des contraintes contextuelles est conçu pour être « à la disposition » des experts. En effet, si un architecte est disponible, il peut grandement influencer les éléments extraits de la documentation et donc agir sur le processus d'extraction d'architectures utilisant ces guides. Dans le cas ou l'architecte est peu ou pas disponible, il peut se reposer sur des mécanismes automatiques qui, s'ils sont moins efficaces, permettent d'utiliser les informations intentionnelles malgré le manque de disponibilité des experts.

Partie III

Mise en pratique et validation de ROMANTIC

CHAPITRE 6
Algorithmes pour l'exploration

An algorithm must be seen to be believed.

— Donald KNUTH.

Algorithmes d'exploration — Heuristique — Méta-heuristique — Algorithmes de regroupement — Recuit simulé — Algorithmes génétiques

ROMANTIC est une approche d'extraction d'architectures par exploration. Cette exploration se fait dans l'espace des solutions. Elle utilise pour se diriger une fonction objectif basée sur la sémantique et la qualité architecturale. ROMANTIC utilise aussi la documentation, les recommandations de l'architecte et le contexte pour réduire l'espace de recherche.

Tous ces éléments, qui constituent le cœur de ROMANTIC, ont été présentés dans la partie II. Ils ont permis de comprendre le fonctionnement de l'approche en détaillant les objectifs et les guides de l'exploration. Cependant, nous n'avons pas donné d'indication sur la façon dont se déroule l'exploration.

Ce découpage de notre approche, en deux parties, nous permet de découpler l'aspect théorique et l'aspect algorithmique de notre approche. Nous pouvons ainsi modifier pratiquement indépendamment chacun de ces aspects. En particulier, ceci nous permet de comparer les performances des différents algorithmes d'exploration en décrivant une instance de notre approche pour chacun d'entre eux. En effet, cette comparaison est nécessaire face au grand nombre de méta-heuristiques et à la diversité de leurs propriétés.

La suite du chapitre est organisée de la façon suivante. Nous présentons d'abord les algorithmes d'exploration et notre choix d'algorithmes pour réaliser notre approche. Nous détaillons ensuite une instance de ROMANTIC reposant sur un algorithme de regroupement hiérarchique. Puis, nous décrivons les instances utilisant des algorithmes méta-heuristiques : une instance basée sur l'algorithme du recuit simulé puis une autre basée sur les algorithmes génétiques.

6.1 Algorithmes d'exploration pour ROMANTIC

Les algorithmes d'exploration peuvent être classés en plusieurs catégories selon leurs propriétés principales [40]. Chacune de ces catégories représente des algorithmes qui ont un mode d'exploration similaire et surtout qui présentent des avantages et inconvénients très proches. Les critères de comparaison

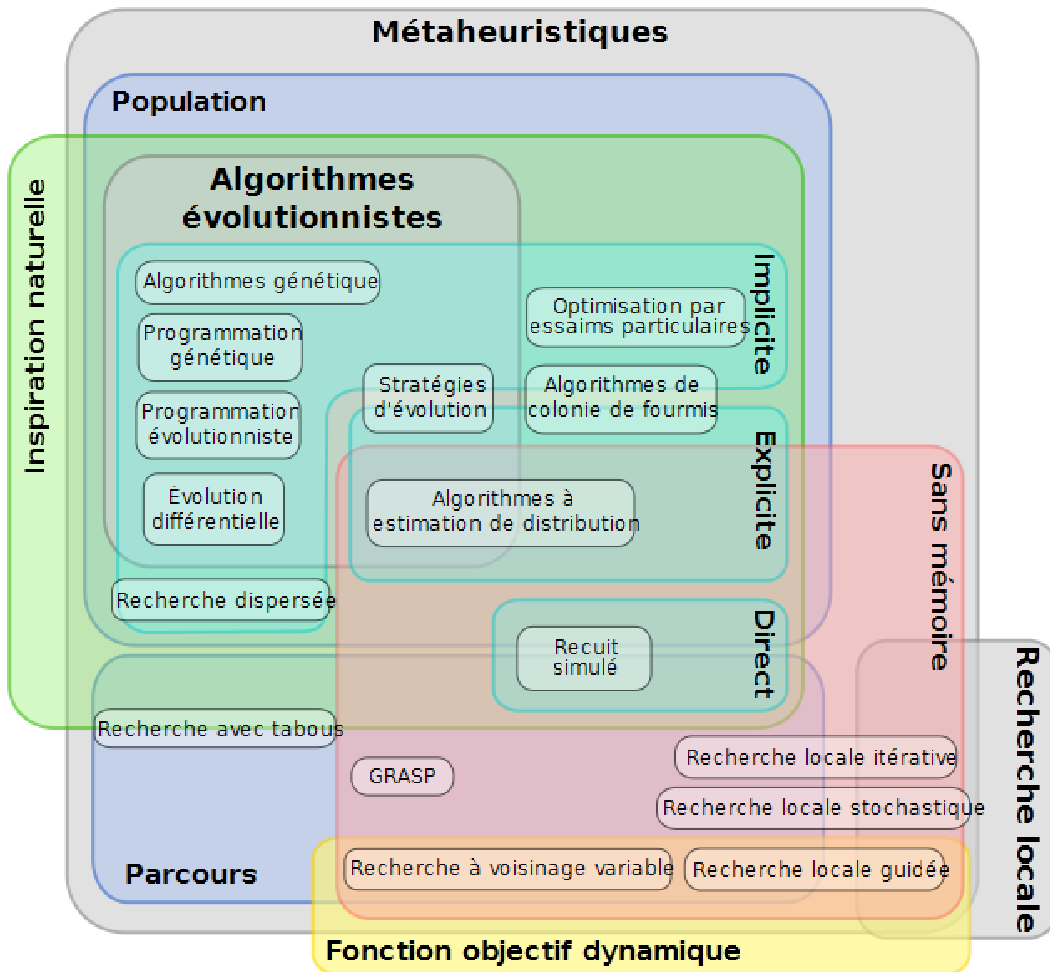


Figure 6.1 – Classification des algorithmes méta-heuristiques

sont le nombre de solutions parcourues en même temps, l'utilisation de la mémoire durant l'algorithme, l'aspect statique ou dynamique de la fonction objectif, le mode d'échantillonnage de cette fonction et enfin l'aspect évolutionnaire ou non de l'algorithme (*cf.* Figure 6.1).

Malgré le grand nombre d'algorithmes, il est impossible de déterminer un meilleur algorithme absolu. En effet, le théorème « *no free lunch* » [132] établit qu'une méta-heuristique ne peut prétendre être plus efficace sur tous les problèmes, bien que certaines instances (c'est-à-dire l'algorithme lui-même, mais aussi un choix de paramètres et une implémentation donnée) puissent être plus adaptées que d'autres sur certaines classes de problèmes.

La réduction à la classe des problèmes d'ingénierie logicielle n'est pas suffisante pour faire émerger un algorithme. En effet, il n'existe pas de consensus réel sur le meilleur algorithme d'exploration au sein de la communauté SBSE (*Search-Based Software Engineering*). Certains utilisent les algorithmes génétiques [39, 111], d'autres le recuit simulé [98], l'exploration avec tabou (*tabu search*) [93] ou encore les algorithmes de regroupement (*clustering*) [4, 87]. D'autres utilisent justement différents algorithmes pour comparer ou pour pouvoir bénéficier des avantages de chacun en fonction de l'utilisation [6, 86].

Nous proposons de développer différents algorithmes d'exploration pour notre approche. La définition de ces algorithmes nous permet de comparer leurs avantages et inconvénients dans le cadre précis de notre problème d'extraction d'architectures. En effet, l'espace de recherche, la fonction objectif ou encore les guides permettant de réduire l'espace de recherche sont autant d'influences sur la qualité d'un algorithme dans un contexte donné.

Parmi les algorithmes méta-heuristiques, nous avons sélectionné les algorithmes de recuit simulé et génétique. Ces deux algorithmes permettent de couvrir quasiment toutes les familles de méta-heuristiques, à l'exception de la recherche locale (*cf.* Figure 6.1). A ce titre, leur utilisation permet de mesurer les avantages de chaque famille. De plus, leur proximité avec l'ensemble des autres méta-heuristiques permet de facilement adapter ces dernières à notre problème si l'on souhaite les tester spécifiquement.

Cependant, avant d'utiliser les méta-heuristiques, nous avons implémenté un algorithme reposant sur une étape de regroupement pour pouvoir tester notre fonction objectif et les premiers éléments de notre approche. Ces algorithmes de regroupement visent à répartir un ensemble d'éléments dans des groupes de telle sorte que les éléments d'un groupe sont plus similaires entre eux qu'avec les éléments des autres groupes. Ils sont utilisables uniquement dans le cas où le problème peut être ramené à un problème de partitionnement. Ils sont alors particulièrement efficaces puisqu'ils ne sont pas stochastiques et ont une complexité linéaire ou presque. Ils s'exécutent donc en un temps quasi-constant et raisonnable pour un jeu de données.

6.2 Instance de ROMANTIC à base de regroupement hiérarchique

Avant de proposer des algorithmes d'exploration pour instancier notre approche, nous proposons une instance de ROMANTIC basée sur un algorithme de regroupement. Cet algorithme est le premier que nous avons développé. Son aspect non-stochastique nous a permis de tester les fonctions objectifs et de comparer les apports des différents guides.

La première version de cet algorithme visait uniquement à identifier une architecture simplifiée, c'est-à-dire dont seuls les composants nous intéressent. Nous avons ensuite étendu cet algorithme pour compléter l'architecture extraite en identifiant les connecteurs et les autres éléments de l'architecture.

La version étendue utilise également en partie les guides pour la réduction de l'espace de recherche. Mais, étant donné que notre algorithme n'explore pas l'espace des solutions, il ne tire pas totalement partie des méthodes de réduction de l'espace de recherche que nous avons présentées dans le chapitre 5.

Par la suite, nous présentons les deux versions de notre algorithme. La première repose sur un algorithme de regroupement hiérarchique et utilise notre fonction objectif pour identifier une architecture simplifiée. Elle vise en fait à calculer une partition des classes du système, telle que chaque partie puisse être identifiée à un composant.

La deuxième version étend la première. Elle ajoute à celle-ci différentes étapes permettant d'identifier les autres éléments architecturaux. Elle propose également de modifier l'algorithme de regroupement pour prendre en compte les méthodes de réduction de l'espace de recherche.

Enfin, nous analysons cette instance de ROMANTIC en étudiant sa complexité, ses avantages et ses inconvénients.

6.2.1 Extraction d'architectures simplifiées

Nous avons d'abord utilisé un algorithme non-stochastique d'extraction d'architectures simplifiées. Cet algorithme nous a permis de tester les différents aspects théoriques de notre approche. Ces tests sont possibles surtout grâce à la simplicité de l'implémentation qui nous permet une adaptation à volonté mais aussi grâce à son statut non-stochastique qui nous permet de reproduire et de comparer facilement les différentes exécutions en fonction du paramètrage de la fonction objectif par exemple.

Cet algorithme est conçu pour l'extraction d'une vue architecturale simplifiée. L'objectif est uniquement d'identifier les composants de l'architecture selon notre fonction objectif. Pour cela, nous procédons à la partition des classes du système et nous identifions chaque partie à un contour de composant. Cette identification est faite en deux étapes : le regroupement hiérarchique des classes puis l'identification des contours de composants.

Mais, le calcul de notre fonction objectif nécessite différents éléments qui ne sont pas mesurables sur la base unique des contours de composant. Il nous faut donc proposer des heuristiques pour identifier les éléments architecturaux manquants à partir des contours de composants.

Dans la suite, nous présentons d'abord les heuristiques permettant le calcul de notre fonction objectif. Nous présentons ensuite les deux étapes de l'extraction : le regroupement hiérarchique des classes du système puis l'identification des composants.

6.2.1.1 Heuristiques pour le calcul de la fonction objectif

Cet algorithme a été conçu à l'origine pour un modèle de correspondance et une fonction objectif qui ne prend en compte que le contour des composants. L'algorithme vise à extraire une partition des classes du système puis à identifier ces classes à des contours de composants puis à des composants. Il ne calcule pas de contours autres que celui des composants et ne permet donc pas de mesurer des propriétés sur les contours d'interfaces et de connecteurs.

Pour pouvoir l'utiliser dans le cadre de notre modèle actuel, nous proposons des heuristiques permettant d'obtenir les détails des éléments manquants. Ces heuristiques visent à pouvoir calculer la fonction objectif uniquement sur les contours de composants et sans avoir besoin d'identifier les contours d'interfaces, de connecteurs ou de configuration :

- **heuristiques pour les contours d'interfaces** : pour chaque voisin d'un contour de composant c, nous identifions les contours d'interfaces fournies et requises en fonction de l'ensemble des appels de méthodes entre les deux contours de composant. Ainsi, dans chaque contour de composants c, chaque méthode appellée est identifiée à un contour d'interface fournie et chaque appel de méthode externe à c identifie la méthode externe à un contour d'interface requise ;

- **heuristiques pour les contours de connecteurs** : nous proposons de considérer les connecteurs de l'architecture comme des connecteurs simples, c'est-à-dire de simples appels de méthodes entre deux composants. Ainsi, entre deux contours de composants, les contours d'interfaces correspondantes sont connectées à travers un contour de connecteur vide.

6.2.1.2 Algorithme de regroupement hiérarchique

Afin d'identifier les contours de composants, nous commençons par regrouper les classes du système en fonction de la qualité du groupe formé du point de vue de notre fonction objectif. Pour cela, nous utilisons un algorithme de regroupement. En effet, ces algorithmes visent à répartir un ensemble d'éléments dans des groupes de telle sorte que les éléments d'un groupe sont plus similaires entre eux qu'avec les éléments des autres groupes. Nous pouvons donc utiliser ces algorithmes en utilisant notre fonction objectif comme mesure de la similarité entre les éléments.

Parmi les algorithmes de regroupement existants, nous avons choisi un algorithme hiérarchique [68]. En effet, les autres algorithmes de regroupement, tel que celui des K-voisins [116], nécessitent une notion de voisinage qui est différente de celle de similarité et qui ne permet pas d'utiliser directement notre fonction objectif.

Le résultat de l'algorithme de regroupement hiérarchique est un arbre binaire où chaque nœud représente un groupe, les feuilles les éléments initiaux et la racine le groupe final. Cette arbre binaire est aussi appelé dendrogramme.

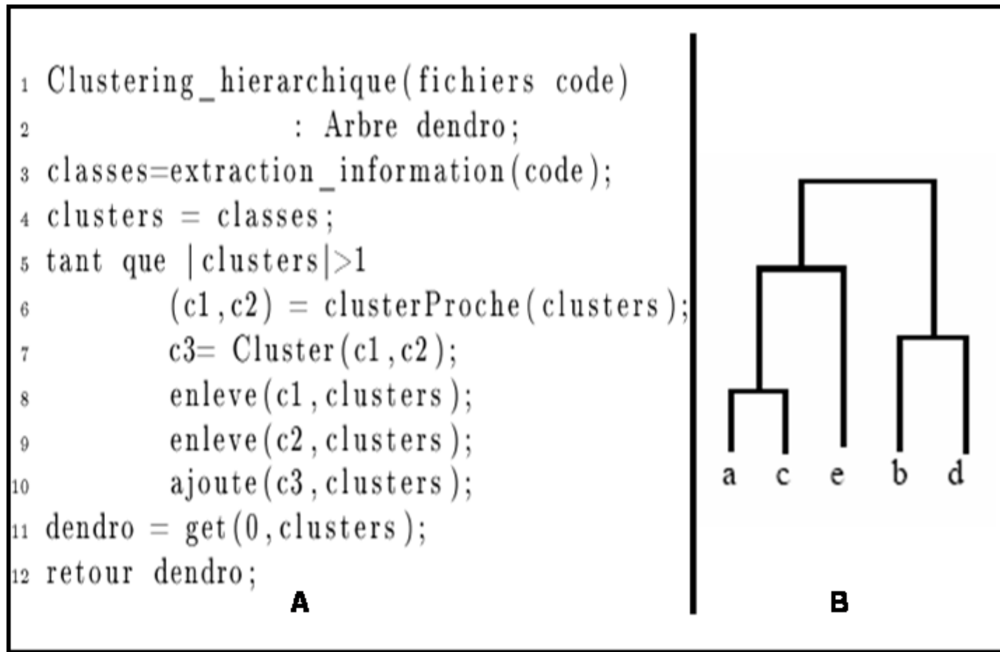
Le principe de l'algorithme de regroupement hiérarchique est donné dans la figure 6.2.A. Au démarrage, l'algorithme de regroupement hiérarchique considère que chaque élément initial constitue un groupe (ligne 4). Ensuite, à chaque étape, les deux groupes les plus similaires (fonction $clusterProche$) sont fusionnés (lignes 5 à 10). Le processus se termine lorsqu'il ne reste plus qu'un seul groupe. A partir de ce dernier, on obtient une représentation en forme de dendrogramme de la hiérarchie des groupes (ligne 11). La figure 6.2.B présente la forme du résultat de cet algorithme. Dans cet exemple, les étapes du regroupement sont (a,c), (a,c,e), puis (b,d) et enfin (a,b,c,d).

Nous exploitons cet algorithme pour regrouper les classes objets du système en utilisant comme fonction de similarité notre fonction objectif. Ainsi, l'évaluation de la similarité de deux groupes consiste à appliquer la fonction objectif sur le contour de composant formé par les classes de l'union des deux groupes.

6.2.1.3 Algorithme d'identification des composants à partir du dendrogramme

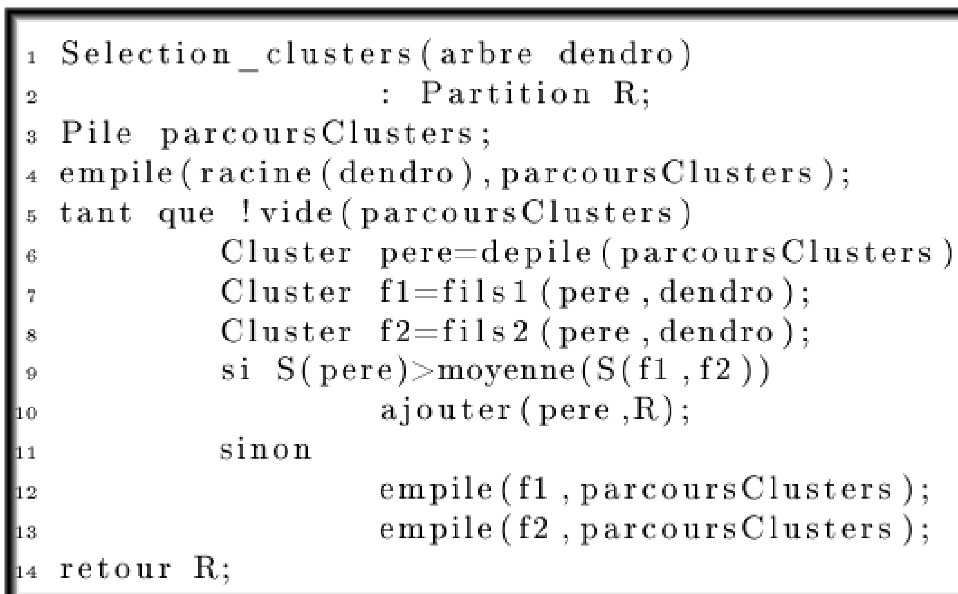
Pour obtenir une partition des classes du système, on doit sélectionner des nœuds dans la hiérarchie résultant de l'étape précédente. Cette sélection est réalisée par l'algorithme présenté dans la figure 6.3 qui sélectionne les nœuds qui représentent les meilleurs contours de composants du point de vue de notre fonction objectif.

L'algorithme de sélection des nœuds consiste en un parcours en profondeur du dendrogramme (paramètre $dendro$). A chaque nœud, on compare le résultat de la fonction d'évaluation S de ce nœud avec celle de ses deux fils (ligne 9). Si le résultat de la fonction d'évaluation de ce nœud est inférieur à la moyenne de ses deux fils, alors l'algorithme traite le premier fils (lignes 12 et 13), sinon le nœud en question est identifié comme un contour et ajouté à la partition (variable R, ligne 10) et l'algorithme traite ensuite le nœud suivant.



```
1  Clustering_hierarchique(fichiers code)
2                  : Arbre dendro;
3  classes=extraction_information(code);
4  clusters = classes;
5  tant que |clusters|>1
6          (c1,c2) = clusterProche(clusters);
7          c3= Cluster(c1,c2);
8          enleve(c1,clusters);
9          enleve(c2,clusters);
10         ajoute(c3,clusters);
11 dendro = get(0,clusters);
12 retour dendro;
```


Figure 6.2 – Algorithme de regroupement hiérarchique



```
1  Selection_clusters(arbre dendro)
2                      : Partition R;
3  Pile parcoursClusters;
4  empile(racine(dendro),parcoursClusters);
5  tant que !vide(parcoursClusters)
6          Cluster pere=depile(parcoursClusters);
7          Cluster f1=fils1(pere,dendro);
8          Cluster f2=fils2(pere,dendro);
9          si S(pere)>moyenne(S(f1,f2))
10                 ajouter(pere,R);
11         sinon
12                 empile(f1,parcoursClusters);
13                 empile(f2,parcoursClusters);
14 retour R;
```

Figure 6.3 – Algorithme d'identification des composants à partir du dendrogramme

6.2.2 Extraction d'architectures logicielles

Le résultat de l'extraction d'architectures simplifiées est un ensemble de composants. Nous avons calculé des contours d'interfaces et de connecteurs, mais cette identification est faite sur la base d'heuristiques qui considèrent les connecteurs comme des connecteurs simples, c'est-à-dire ne contenant qu'un appel de méthode.

Notre approche vise cependant à fournir une vue architecturale avec des composants et des connecteurs du même niveau. Nous devons donc procéder à l'identification de connecteurs non triviaux qui représentent plus qu'un simple appel de méthode. Nous devons également identifier les autres éléments de l'architecture que cette vue simplifiée a ignorés : les interfaces et les composites. Nous proposons donc un ensemble d'heuristiques pour identifier les contours de chacun de ces éléments ainsi que les composites.

Enfin, l'algorithme que nous avons proposé ne prend pas en compte les guides que nous avons définis pour réduire l'espace de recherche. Il nous faut donc étudier l'impact de ces guides sur notre algorithme.

Dans la suite, nous présentons les heuristiques que nous avons développées pour identifier les contours d'interfaces, de connecteurs, de configuration et les composants composites. Ces heuristiques visent à calculer les meilleurs contours en fonction des contours de composants que nous avons calculés dans le premier algorithme. Ils constituent donc la seconde phase de l'algorithme précédent et complètent notre algorithme d'extraction d'architectures à base de regroupement hiérarchique.

Nous proposons également des modifications dans le déroulement de la première phase de notre algorithme. Ces modifications nous permettent de prendre en compte les guides pour la réduction de l'espace de recherche, malgré le fait que nous n'utilisons pas un algorithme d'exploration.

6.2.2.1 Identification des interfaces

Principes. A partir de l'architecture simplifiée extraite lors de la première phase, nous identifions les interfaces des composants. Pour cela, nous utilisons une heuristique sur les contours de composants afin de déterminer les contours des interfaces associés à chacun.

Notre heuristique vise à déterminer la séparation entre les méthodes qui restent dans le composant, et celles qui appartiennent au contour. Pour cela, elle se base sur l'hypothèse que les méthodes qui sont plus en contact avec l'extérieur d'un contour de composant doivent appartenir à un contour de connecteur plutôt que de composant. En effet, ces méthodes jouent un rôle de communication entre les classes qui les contiennent et les autres classes situées dans un autre contour de composant.

L'algorithme parcourt donc les méthodes des contours de composants et teste leurs liens avec les autres méthodes du contour et celles à l'extérieur. Il décide alors si la méthode appartient à un contour d'interface, de composant ou de connecteur.

Notre algorithme impose une limite sur les interfaces identifiées. En effet, l'identification est faite en attribuant une méthode par contour d'interfaces. Par conséquent, les interfaces identifiées ne proposent ou ne nécessitent qu'un seul service.

Algorithme. Afin d'identifier les méthodes qui constituent un contour d'interface, notre algorithme parcourt l'ensemble des contours de composants (*cf.* Algorithme 1). Pour chaque composant, nous identifions d'abord les contours d'interfaces requises puis ceux d'interfaces fournies. Notre algorithme utilise les méthodes suivantes : $out()$, s'applique à un contour ou à une méthode et désigne l'ensemble des méthodes appelées par les méthodes, c'est-à-dire les successeurs du contour ou de la méthode ; $in()$ désigne l'ensemble des méthodes qui sont appelées depuis l'extérieur d'un contour ou l'ensemble des méthodes

qui appelle une méthode c'est-à-dire les prédécesseurs du contour ou de la méthode ; $pop\,()$ supprime un élément d'un ensemble et retourne cet élément.

Algorithme 1 Algorithme d'identification des interfaces

ENTRÉES: $Archi$ un ensemble de contours de composant
SORTIES: calcul les contours des interfaces associées aux contours de composants de $Archi$

1: **pour tout** $c \in Archi$ **faire**
2: $S \leftarrow c.out\,()$
3: **tantque** $S \neq \emptyset$ **faire**
4: $s \leftarrow S.pop\,()$
5: **si** $s.out\,() \cap c.getMethodes\,() \neq \emptyset$ ou $s.in\,() = \emptyset$ **alors**
6: $I_R\,(c)\,.add\,(s.out\,() - c.getMethodes\,())$
7: **sinon**
8: $S.add\,(s.in\,())$
9: $I_R\,(c)\,.add\,(s)$
10: $I_R\,(c)\,.remove\,(s.out\,())$
11: **finsi**
12: **fin tantque**
13: $E \leftarrow c.in\,()$
14: **tantque** $E \neq \emptyset$ **faire**
15: $e \leftarrow E.pop\,()$
16: **si** $e.in\,() \cap c.getMethodes\,() \neq \emptyset$ ou $e.out\,() = \emptyset$ **alors**
17: $I_F\,(c)\,.add\,(e)$
18: **sinon**
19: $E.add\,(e.out\,())$
20: **finsi**
21: **fin tantque**
22: **fin pour**
23: **return** $Archi$

L'identification des interfaces requises est un parcours des méthodes du contour à partir des méthodes qui font un appel vers l'extérieur du contour ($c.out\,()$) (lignes 2 à 12). Pour chaque méthode, si elle contient des appels vers l'intérieur du contour alors elle reste dans le contour et les méthodes externes au contour qu'elle appelle sont ajoutées à la liste des interfaces requises du contour ($I_R\,(c)$, ligne 6). Sinon la méthode est ajoutée à la liste des interfaces requises (ligne 9), on supprime ses successeurs de la liste des interfaces puisque cette méthode les remplace (ligne 10). Enfin on ajoute les méthodes qui appellent la méthode courante à la liste des méthodes à vérifier (ligne 8).

L'identification des interfaces fournies est un parcours des méthodes du contour à partir des méthodes qui sont appelées depuis l'extérieur du contour ($c.in\,()$) (lignes 13 à 22). Pour chaque méthode, si elle est appelée depuis l'intérieur du contour alors elle est ajoutée à la liste des interfaces fournies du contour ($I_F\,(c)$, ligne 17). Sinon les méthodes qui sont appelées par la méthode courante sont ajoutées à la liste des méthodes à vérifier (ligne 19).

Le résultat de cet algorithme d'identification est un ensemble de contours de composants et d'interfaces ($Archi$, ligne 23).

6.2.2.2 Identification des connecteurs

Principes. A partir de l'ensemble de contours de composants et d'interfaces issus de l'algorithme précédent, nous devons identifier les connecteurs de l'architecture. Pour cela, nous utilisons une heuristique basée sur les contours de composants et d'interfaces afin de déterminer les contours des connecteurs associés à chacun.

Notre heuristique vise à déterminer la séparation entre les méthodes qui sont contenues dans un connecteur et de quelle façon elles sont regroupées. Pour cela, elle se base sur les méthodes contenues dans les contours d'interfaces. En effet, ces méthodes définissent la limite entre les méthodes appartenant au contour de composants et de connecteurs. Elles permettent donc de calculer les méthodes qui appartiennent à un contour de connecteur. Il suffit ensuite de mettre en relation les contours de connecteurs en fonction des methodes qu'ils fournissent ou requièrent pour obtenir la répartition des méthodes dans les contours de connecteurs.

Notre algorithme impose une limite sur le nombre de connecteurs entre deux composants. Ainsi, par construction, il n'existe qu'un seul connecteur entre deux composants. Ce connecteur peut accepter des interfaces fournies et requises de chaque composant impliqué dans la connexion.

Afin d'identifier les contours de connecteurs, notre algorithme parcourt d'abord l'ensemble des contours de composants (*cf.* Algorithme 2). Pour chaque contour de composant, nous identifions les méthodes qui, lors de l'identification des contours d'interfaces, ont été désignées comme appartenant à un contour de connecteurs. Ceci nous donne deux ensembles de contours de connecteurs selon que les méthodes contenues dans le contour sont requises ou pas par le contour de composant dont elles proviennent. Ensuite, notre algorithme parcourt l'ensemble des contours de connecteurs requis et les associent aux contours de connecteurs fournies. Notre algorithme utilise les mêmes méthodes que pour l'identification des interfaces.

L'identification des méthodes qui sont incluses dans un connecteur est un parcours des interfaces du contour de composant (lignes 2 à 13). Pour chaque contour d'interface requise, nous créons un contour de connecteur requis. Ce contour est représenté par un doublet dont le premier élément représente l'ensemble des méthodes internes du contour du connecteur et le second l'ensemble des méthodes requises par le contour de connecteur. Si la méthode de l'interface appartient à une classe du contour (ligne 3), le contour contient la méthode et requiert les méthodes qu'elle appelle (ligne 4). Sinon, il ne contient aucune méthode et requiert la méthode du contour d'interface (ligne 6). Ensuite, pour chaque contour d'interface fournie, si la méthode de l'interface est appelée par des méthodes du contour qui sont elles-mêmes appelées depuis l'extérieur (ligne 10), nous créons un contour de connecteur fourni qui contient les méthodes du contour appelant l'interface fournie et appelées depuis l'extérieur du contour de composant et qui requiert la méthode de l'interface (ligne 11).

L'identification des connecteurs est un parcours des contours de connecteurs requis (lignes 15 à 23). Pour chaque contour, nous cherchons les méthodes requises parmi les contours de connecteurs puis les contours de connecteurs fournis (Con et Con_F, ligne 17). Si un contour de connecteur contient une méthode requise, nous fusionnons les deux contours en unissant les éléments internes et les éléments requis et en supprimant de l'ensemble des éléments requis les nouveaux éléments internes (ligne 19). Nous supprimons ensuite le contour de connecteur correspondant de l'ensemble des contours de connecteurs (ligne 20).

Algorithme 2 Algorithme d'identification des connecteurs

ENTRÉES: $Archi$ un ensemble de contours de composant
SORTIES: Con, l'ensemble des contours de connecteurs

1: **pour tout** $c \in Archi$ **faire**
2:    **pour tout** $i \in I_R(c)$ **faire**
3:       **si** $i \in c.getMethodes()$ **alors**
4:          $Con_R.add((\{i\}, i.out()))$
5:       **sinon**
6:          $Con_R.add((\emptyset, \{i\}))$
7:       **finsi**
8:    **fin pour**
9:    **pour tout** $i \in I_F(c)$ **faire**
10:       **si** $i.in() \cap c.in() \neq \emptyset$ **alors**
11:          $Con_F.add((i.in() \cap c.in(), \{i\}))$
12:       **finsi**
13:    **fin pour**
14: **fin pour**
15: **pour tout** $c_1 \in Con_R$ **faire**
16:    $R \leftarrow c_1.req()$
17:    **pour tout** $c_2 \in Con, , Con_F$ **faire**
18:       **si** $R \cap c_2.int() \neq \emptyset$ **alors**
19:          $Con.add(c_1 + c_2)$
20:          $Con.remove(c_2)$
21:       **finsi**
22:    **fin pour**
23: **fin pour**
24: **return** Con

6.2.2.3 Identification de la configuration

Les algorithmes d'identification des interfaces et des connecteurs ne mettent pas en correspondance les connecteurs et les interfaces. Pour réaliser cette liaison, nous devons identifier la configuration de l'architecture.

Cette identification est faite en **parcourant l'ensemble des contours de connecteurs**. En effet, le connecteur est lié aux composants qui contiennent une interface requise au nom d'une méthode contenue ou requise par le contour de connecteur. Le connecteur est également lié aux composants qui contiennent une interface fournie au nom d'une méthode requise par le contour de connecteur.

6.2.2.4 Identifications des composites

Après l'identification des composants, des interfaces, des connecteurs et de la configuration, il reste à identifier les composants composites. Pour cela nous utilisons les résultats de la première phase de l'extraction : le dendrogramme. En effet, celui-ci comporte des informations d'inclusions sur les groupes que nous n'avons pas utilisées et qui peuvent être assimilées aux relations de compositions verticales : les groupes sont hiérarchisés par ces informations de la même manière que les composants le sont par les relations de compositions verticales.

Ainsi, dans le dendrogramme, **si l'un au moins des fils d'un contour de composant est un groupe, alors le contour de composant est un composite**. Le calcul de l'architecture de ce composite est alors obtenue en appliquant de manière récursive notre approche sur les éléments du contour du composant composite.

6.2.2.5 Utilisation des méthodes de réductions de l'espace de recherche

Cet instance de ROMANTIC repose sur un algorithme de regroupement hiérarchique. C'est une heuristique qui n'explore donc pas un espace de solutions. A ce titre, elle ne peut pas bénéficier pleinement, des guides et des méthodes de réduction de l'espace de recherche. Cependant, les informations extraites de ces guides peuvent quand même influer sur le comportement de cette instance de notre approche.

Nous utilisons le réseau de contraintes, extrait de la documentation et des recommandations de l'architecte (*cf.* Chapitre 5), pour **limiter les choix de réunions des groupes**. En effet, dans l'algorithme de regroupement hiérarchique générique, ce choix est totalement aléatoire, mais nous pouvons modifier cela et rendre le choix dépendant des contraintes.

Si la réunion de deux groupes viole une contrainte requise (niveau 2), cette réunion est interdite même si ces deux groupes sont les plus similaires. Les contraintes plus faibles influencent le choix des groupes à réunir dans le cas où les paires de groupes les plus similaires ne sont pas uniques. Dans ce cas, l'algorithme sélectionne la paire dont le résultat enfreint le moins de contraintes. Il sélectionne d'abord les paires qui enfreignent le moins de contraintes de niveau 2 puis il opère de même pour celles de niveau 1. Enfin, il procède par tirage aléatoire entre les paires restantes.

6.2.3 Analyse du processus par regroupement hiérarchique

Nous avons présenté une instance de notre approche basée sur un algorithme de regroupement. Cette instance n'est pas une méta-heuristique et elle ne réalise pas l'exploration de l'espace des architectures possibles. Cependant, elle présente différents intérêts pour nos travaux.

Nous analysons d'abord la complexité de cette instance puis nous détaillons les avantages et les inconvénients de cette approche heuristique de ROMANTIC.

6.2.3.1 Complexité des algorithmes

Nous étudions la complexité de notre algorithme en fonction des phases. Ainsi, nous analysons d'abord la complexité de l'extraction d'architectures simplifiées puis de l'extraction d'architectures logicielles.

La première phase comporte deux étapes : le regroupement hiérarchique des classes du système, puis le partitionnement des classes. Dans les deux cas, la complexité est linéaire par rapport au nombre de classes du système. En effet, le regroupement hiérarchique débute avec n groupes, où n est le nombre de classes. Il élimine ensuite un groupe par étape jusqu'à ce qu'il ne reste plus qu'un seul groupe. Il y a donc n itérations. De même, le partitionnement réalise un parcours en profondeur du dendrogramme. Il y a donc au plus autant d'itérations que de sommets dans le dendrogramme, c'est-à-dire $2 * n$.

La complexité de la première phase de notre algorithme est donc linéaire par rapport au nombre de classes du système.

La seconde phase est plus complexe. L'identification des interfaces peut nécessiter le parcours de l'ensemble des méthodes du système. Elle est donc linéaire par rapport au nombre m de méthodes dans le système. Ensuite, l'identification des connecteurs nécessite le parcours de tous les contours de connecteurs à partir de chaque contour de connecteurs requis. Cette partie est la plus complexe de cette étape. Dans le pire cas, il y a un contour de connecteur par méthode et la complexité est donc de l'ordre de m^2. Enfin, l'identification de la configuration et des composites, sont de simples parcours de l'ensemble des connecteurs et de l'ensemble des sommets du dendrogramme, leur complexité est donc respectivement de l'ordre de m et de $2 * n$.

La complexité de la deuxième phase de notre algorithme est donc quadratique en le nombre de méthodes du système.

6.2.3.2 Avantages et inconvénients

Le choix d'utiliser une heuristique pour réaliser notre approche a des conséquences profondes sur l'utilisation et le fonctionnement de notre approche. Parmi ces conséquences, la comparaison de l'efficacité nécessite de comparer cette instance avec les autres au sein d'un cas d'études. Cependant, ce choix a d'autres conséquences, bénéfiques ou nocives, qui peuvent être analysées à partir de l'algorithme et de sa conception.

Nous présentons donc les avantages et les inconvénients de cette réalisation heuristique de notre approche.

Avantages. Les principaux avantages de cette heuristique sont son caractère non-stochastique et sa faible complexité algorithmique. Le fait de pouvoir reproduire des résultats et de pouvoir le faire rapidement nous a permis d'affiner les poids des fonctions de mesure. Cela nous a aussi permis de choisir un paramétrage pour la fonction objectif.

En plus de ces avantages, la simplicité d'implémentation de notre algorithme a permis de tester rapidement les métriques et les fonctions de notre approche. Nous avons ainsi débuté l'étude de chaque aspect de notre approche en le projetant sur cet algorithme pour tester de manière informelle la validité et la pertinence de chaque étape de l'approche.

Du point de vue de l'utilisateur, cet algorithme peut être utilisé de façon similaire pour préparer une exécution gourmande en temps de calcul. Ainsi, l'approche par regroupement permet à l'architecte de se familiariser avec le paramétrage de notre fonction objectif avant de réaliser l'extraction proprement dite en utilisant une approche par exploration.

Inconvénients. L'aspect heuristique de cet algorithme a néanmoins des inconvénients majeurs pour notre approche. En effet, les guides et les modèles de notre approche sont conçus pour une approche par exploration , c'est-à-dire réalisée par une méta-heuristique. Cet algorithme, s'il permet de vérifier notre approche, ne peut pas bénéficier totalement de ses apports.

Mais le principal inconvénient provient de la présence de deux phases. En effet, la première phase est la seule qui utilise la fonction objectif. Elle est donc la seule qui réalise une certaine optimisation. La seconde phase est un calcul heuristique dépendant de l'optimisation précédente. En termes architecturaux, cela signifie que les composants sont optimisés mais que les autres éléments architecturaux sont calculés en fonction des composants. Cet algorithme introduit donc une différence fondamentale de traitement entre les éléments architecturaux.

6.3 Instance de ROMANTIC à base de recuit simulé

Le recuit simulé (*«Low-Temperature Simulated Annealing»*) [78] est une méta-heuristique qui a été souvent utilisée dans le domaine du génie logiciel et en particulier dans le SBSE [98, 93, 6]. Cet algorithme simule un processus utilisé en métallurgie. Ce processus alterne des cycles de refroidissement lent et de réchauffage (recuit) qui tendent à minimiser l'énergie du matériau.

Le recuit simulé est une méta-heuristique qui explore l'espace des solutions d'un problème afin de trouver la meilleure solution. Cette exploration est faite en utilisant le principe du parcours, c'est-à-dire qu'il fait évoluer une seule solution sur l'espace de recherche à chaque itération.

Pour réaliser ce parcours, le recuit n'utilise pas de mémoire, c'est-à-dire que les solutions rencontrées n'ont pas d'influence sur les choix futurs. Ces choix reposent uniquement sur une fonction dynamique qui dépend d'une fonction objectif statique et d'un paramètre dynamique qui évolue pendant l'exploration. Ce paramètre représente la température du processus physique et permet au recuit de réorienter une solution d'un maximum local vers une autre partie de l'espace de recherche.

Nous utilisons le recuit simulé pour sélectionner la meilleure architecture au sens de notre fonction objectif (*cf.* Chapitre 4). Cependant, nous devons définir un ensemble de paramètres qui sont spécifiques à chaque problème. Ces paramètres définissent la façon dont le processus parcourt l'espace de recherche. Ils précisent, en fait, le voisinage de chaque solution et la façon dont le recuit passe d'une solution à l'autre.

Nous proposons également une adaptation du recuit aux guides que nous avons identifiés. Pour cela, nous modifions l'algorithme pour prendre en compte les deux méthodes de réduction de l'espace de recherche que nous avons identifiées. Nous adaptons donc le recuit pour prendre en compte le réseau de contraintes et les architectures intentionnelles extraites des guides que sont la documentation et les recommandations de l'architecte.

Dans la suite, nous présentons notre algorithme d'extraction basé sur le recuit simulé. Pour cela, nous présentons d'abords l'algorithme d'exploration. Nous étudions donc l'algorithme de recuit simulé et le paramétrage que nous utilisons. Nous proposons également des modifications du recuit simulé afin de prendre en compte les guides réduisant l'espace de recherche.

Ensuite, nous définissons la notion de voisinage qui permet au recuit simulé d'explorer l'espace de recherche. Pour cela, nous proposons un ensemble d'opérateurs de manipulations qui permettent de modifier les solutions et de passer de l'une à l'autre. Nous exposons également les méthodes de sélection de ces opérateurs et de leur cible.

Après le voisinage, nous définissons le point de départ de l'exploration. Pour cela, nous utilisons les guides, décrits dans le chapitre 5, pour définir différents points de départ possibles. Nous utilisons également une heuristique pour définir un autre point de départ.

Enfin, nous analysons cette instance de ROMANTIC. Pour cela nous étudions sa complexité, ses avantages et ses inconvénients.

6.3.1 Algorithme d'extraction

Nous avons choisi d'utiliser le recuit simulé pour explorer l'espace des solutions de notre problème d'extraction de l'architecture logicielle. Nous présentons ici les principes du recuit simulé et le paramétrage que nous utilisons. Nous étudions également la façon dont nous pouvons réduire l'espace de recherche. Pour cela, nous adaptons le recuit pour utiliser le réseau de contraintes sur les solutions et les architectures intentionnelles.

6.3.1.1 Principes de l'algorithme de recuit simulé

Le recuit simulé simule un processus utilisé en métallurgie pour minimiser l'énergie d'un matériau [127, 73, 78]. L'alternance entre les phases de refroidissement et de réchauffage permettent au recuit d'explorer une plus grande partie de l'espace de recherche. En effet, le réchauffage tend à éviter d'enfermer trop vite un parcours dans la recherche d'un optimum local.

Le recuit s'appuie sur l'algorithme de Metropolis-Hastings, qui permet de décrire l'évolution d'un système thermodynamique [28]. Par analogie avec le processus physique, l'énergie du système est décrite par la fonction objectif du problème d'optimisation et un paramètre T représente la température du système.

L'algorithme du recuit simulé consiste donc en une série de tentatives de modification sur une solution. Les changements qui accroissent la qualité de la solution sont acceptés et la solution modifiée devient le point de départ d'une nouvelle série de tentatives de modification. De plus, certains changements qui réduisent la qualité de la solution sont acceptés afin de permettre à l'exploration de s'échapper des minima locaux. De tels changements sont acceptés avec une probabilité qui décroit progressivement durant le processus d'exploration selon l'*équation* 6.1, où p est la probabilité d'accepter la solution donnée, δq est la variation de qualité par rapport à la solution courante, et T est la valeur de la température. Chaque début de nouvelles séries de tentatives de modification provoque une décroissance de la température. L'algorithme s'arrête lorsque $T = 0$.

$$p = e^{-\frac{\Delta q}{T}} \tag{6.1}$$

Les paramètres du recuit simulé sont nombreux. Cependant, ils peuvent être classés en deux groupes : les paramètres génériques et ceux spécifiques au problème. Les paramètres génériques, regroupés sous le terme de schéma de refroidissement, décrivent la façon dont l'exploration progresse. Par analogie avec le système physique, ils décrivent la façon dont le système refroidit.

Les paramètres spécifiques dépendent du problème. Ils doivent définir la forme des solutions et la façon dont le recuit passe d'une solution à une autre. Le premier de ces paramètres est la fonction objectif qui mesure la qualité de la solution et reflète l'énergie du système. Nous utilisons notre fonction objectif définie dans le chapitre 4.

6.3.1.2 Schéma de refroidissement

Le schéma de refroidissement inclut plusieurs paramètres : T_{start}, la température de départ, M, le nombre de modifications à chaque étape et f la fonction de refroidissement. Ces paramètres définissent la durée de l'exploration ainsi que le nombre de solutions qui peuvent être explorées. Ils décrivent donc totalement le parcours de l'espace de recherche que peut réaliser l'algorithme de recuit.

La fonction de refroidissement f décrit la façon dont la température décroit à chaque itération. Avec la température initiale, elle détermine le nombre d'itérations du processus de recuit. Parmi les fonctions les plus utilisées, nous trouvons la décroissance arithmétique ou géométrique. Dans le premier cas, la fonction diminue la température d'une valeur constante α à chaque étape : $T_{n+1} = T_n - \alpha$. La deuxième fonction diminue la température d'une fraction constante λ (λ est compris entre 0 et 1) à chaque itération : $T_{n+1} = T_n * \lambda$.

La température de départ décrit avec f le nombre d'itérations maximum du processus. Mais elle décrit également un autre aspect important du recuit. En effet, la température a un impact direct sur la probabilité d'acceptation d'une solution, la température initiale fixe donc la probabilité d'acceptation à chaque étape du processus et en particulier au début. Elle constitue donc un paramètre primordial qui doit surtout être choisi en fonction de cet aspect plutôt que pour influer sur la durée de calcul.

Le nombre de modifications M décrit le nombre de solutions qui peuvent être acceptées à chaque itération du recuit. Il définit donc directement le nombre de solutions qui peuvent être explorées durant le processus.

Nous utilisons un schéma de refroidissement géométrique, c'est-à-dire une décroissance géométrique, et nous faisons tendre le paramètre λ vers 1 en fonction du temps disponible.

Concernant la température, nous calculons T_{start} selon la qualité de la solution de départ. L'objectif est d'obtenir une certaine probabilité d'acceptation lors de la première itération. Pour cela, nous calculons T_{start} de façon que la probabilité d'accepter une solution de qualité nulle soit égale à notre probabilité d'acceptation. Nous utilisons la valeur de 0.8 pour cette probabilité. C'est la probabilité couramment admise dans les utilisations de cet algorithme [78].

Enfin, le nombre de modifications est calculé en fonction des opérations de manipulations que nous utilisons. Nous décrivons ce paramètre en définissant le voisinage dans la section 6.3.2.

6.3.1.3 Réduction de l'espace de recherche

Nous avons proposé dans le chapitre 5, deux méthodes pour réduire l'espace de recherche. Le ciblage utilise d'abord les représentations de l'architecture intentionnelle pour centrer la recherche sur une zone de l'espace de recherche. La suppression vise à éliminer les solutions qui ne sont pas en accord avec la documentation, les recommandations de l'architecte ou le contexte de déploiement.

Le ciblage peut être utilisé pour déterminer le point de départ de l'exploration. Il n'est donc pas nécessaire de modifier l'algorithme du recuit pour prendre en compte cet aspect de la réduction de l'espace de recherche.

Par contre, la suppression impose de modifier l'algorithme du recuit pour pouvoir éviter les solutions à éliminer. Nous devons donc modifier le processus du recuit pour qu'il tienne compte du réseau de contraintes hiérarchiques qui définit les solutions acceptables.

L'adaptation de l'algorithme de recuit consiste à modifier la possibilité pour une solution d'être acceptée. En effet, dans le recuit, l'acceptation d'une solution dépend uniquement de la température et de la différence de qualité. De plus si la solution est meilleure du point de vue de la fonction objectif,

elle est automatiquement acceptée. Nous modifions donc ce calcul pour prendre en compte le réseau des contraintes et en particulier le nombre de contraintes violées par niveau.

Nous commençons par définir la différence de nombre de contraintes violées entre deux solutions. Cette différence se calcule par niveau hiérarchique des contraintes. Ainsi, la différence se calcule au niveau le plus élevé où elle est différente de zéro. Par exemple, si deux solutions a et b enfreignent aucune contrainte du deuxième niveau, puis respectivement 1 et 2 contraintes du premier niveau et enfin 2 et 8 contraintes du niveau zéro, alors la différence de contraintes violées, notée $a - b$ est de 1.

Le calcul de la probabilité d'acceptation se fait alors en fonction de cette différence. Si la différence entre la solution courante, s, et la nouvelle solution, s_n, est négative ($s - s_n \leq 0$) alors la probabilité est calculée comme dans le recuit classique (*cf.* Equation 6.1). Par contre si elle est positive, la probabilité est calculée selon la formule 6.2.

$$p = e^{-\frac{\Delta q}{T*(s-s_n)}} \tag{6.2}$$

6.3.2 Définition du voisinage

Parmi les paramètres spécifiques du recuit, la définition du voisinage est le plus essentiel après la fonction objectif. Le voisinage détermine les solutions accessibles à chaque itération en fonction de la solution courante. La conséquence de son impact sur le processus est que le voisinage doit respecter plusieurs conditions pour assurer la convergence du recuit [52].

La fonction de voisinage détermine, avec le point de départ, le sous-espace de l'espace de recherche qui peut être effectivement exploré. Pour assurer au processus l'exploration de l'ensemble de l'espace de recherche, le voisinage doit pouvoir fournir un chemin suffisamment court entre la solution initiale et n'importe quelle solution étant un maximum potentiel.

La progression du recuit se mesure en fonction de l'amélioration de la qualité de la solution courante. Cette progression doit être plus ou moins constante tout au long du processus. Pour cela, GRANVILLE conseille d'utiliser un voisinage homogène en terme de qualité [52]. En effet, si le voisinage contient un écart important en terme de qualité, le processus peut passer à une très bonne ou très mauvaise solution. Or, les mauvaises solutions sont, à priori les plus nombreuses et donc les plus probables. Au contraire, un voisinage homogène réduit le risque de choisir une mauvaise solution.

Pour définir le voisinage, nous utilisons un ensemble d'opérateurs. Ces opérateurs permettent de manipuler les solutions et de générer le voisinage par applications successives. Le nombre de ces applications est déterminé par la valeur de M, le nombre de modifications autorisées par itération.

Nous proposons donc un ensemble d'opérations de manipulation sur les solutions. Nous présentons aussi les règles qui régissent le calcul et la sélection de ces opérateurs.

6.3.2.1 Opérateurs de manipulation

Les opérateurs de manipulation permettent de **générer une solution à partir d'une autre**. Chaque opérateur possède une valeur de modification qui reflète l'importance de la modification subie par la solution. Cette valeur est utilisée par notre processus pour le calcul de M. Ceci permet d'équilibrer l'application des opérateurs modifiant deux contours de composants et ceux modifiant une méthode dans le contour d'une interface.

Les opérateurs sont classés en fonction des entités COA sur lesquelles ils agissent. Ce classement

correspond aussi à un classement en fonction de la valeur de modification des opérateurs* : une valeur de 10 pour les opérateurs sur les contours de composants et une valeur de 1 pour les opérateurs de connecteurs et interfaces.

Opérateurs sur les contours de composants. Il existe trois opérateurs sur les contours de composants : la fusion, la séparation et l'adoption. **La fusion** de deux contours regroupe les classes des deux contours dans un seul. Pour assurer la validité de la solution, les contours de connecteurs reliant uniquement les deux contours de composants impliqués sont supprimés, comme les contours d'interfaces. Les contours de connecteurs reliant les deux contours de composant avec d'autres contours sont plus complexes à traiter. On conserve le contour de connecteur mais on supprime du contour les méthodes qui relient les deux contours de composants. Enfin, les contours de connecteurs reliant chacun des deux contours de composants à un même contour de composant sont fusionnés.

La séparation d'un contour crée un nouveau contour, à partir d'un contour existant, en migrant certaines classes de l'ancien contour vers le nouveau. Cette séparation entraine l'apparition de nouveaux contours d'interfaces et de connecteurs. Nous utilisons, pour créer ces contours, les heuristiques que nous avons utilisés dans l'instance basée sur le regroupement hiérarchique (*cf.* Algorithmes 1 et 2). Ces heuristiques nous permettent de calculer des contours de qualité plutôt que de proposer des contours de connecteurs vides et des contours d'interfaces pour chaque méthode du contour de composant.

L'adoption permet à un contour de composant de fusionner avec un contour qui ne contient qu'une seul classe. En effet, il est démontré dans les approches de re-modularisation [124], que cette règle de l'orphelin est nécessaire, de manière implicite ou explicite, pour la bonne marche du processus. L'application de cette opération est similaire à celle de fusion. La différence tient au mode de sélection des contours impliqués.

Opérateurs sur les contours d'interfaces. Il existe deux opérateurs sur les contours d'interfaces : la fusion et la séparation. **La fusion** de deux contours regroupe les méthodes représentées dans les deux contours au sein d'un même contour. Cela peut avoir pour conséquence la fusion des contours de connecteurs associés.

La séparation d'un contour d'interface crée un nouveau contour à partir d'un contour existant en migrant certaines méthodes de l'ancien contour vers le nouveau. Les contours de connecteurs liés à l'ancien contour deviennent donc connectés au deux nouvelles interfaces.

Opérateurs sur les contours de connecteurs. Il existe quatre opérateurs sur les contours de connecteurs : la fusion, la séparation, l'ajout et la suppression. **La fusion** de deux contours regroupe les méthodes des deux contours dans un seul et relie le nouveau contour à tous les contours d'interfaces reliés à l'ancien.

La séparation d'un contour de connecteur crée un nouveau contour à partir d'un contour existant en migrant certaines méthodes de l'ancien contour vers le nouveau. Il faut ensuite réassembler en conséquence les contours d'interfaces et les deux nouveaux contours.

L'ajout déplace une méthode de l'intérieur d'un contour de composant vers un contour de connecteur. Cette méthode est déplacée à l'intérieur du contour de connecteur et on modifie les contours d'interfaces impliqués pour conserver une solution valide.

La suppression est exactement l'inverse de l'ajout. Une méthode est supprimée du contour de

*Cette valeur de modification permet d'équilibrer la granularité des modifications à chaque itération en autorisant plus de « petites opérations ». La valeur des opérateurs est déterminée empiriquement.

connecteur. Elle retourne donc dans le contour de composant qui contient sa classe. Il faut ensuite modifier les contours d'interfaces du contour concerné.

Ces opérateurs ne sont pas suffisants pour définir complètement la notion de voisinage. En effet, en l'état, les opérateurs s'appliquent de manière aléatoire sur les différents éléments de la solution. Or, ce caractère aléatoire fait que l'on peut atteindre n'importe quelle solution à partir de la solution courante. Le diamètre de l'espace de recherche est donc petit comme nous l'avons imposé, mais il est trop petit pour nous permettre de bénéficier de l'avantage des méta-heuristiques.

Nous définissons donc des règles déterminant les opérateurs applicables et la façon de les appliquer, en précisant par exemple comment choisir le point de séparation d'un contour de composant. Cependant, il est évident que, même avec un choix aléatoire, la deuxième condition, sur l'homogénéité de voisinage, est respectée.

6.3.2.2 Choix des opérations de manipulations

Afin de définir les critères de choix des opérations de manipulations, nous devons définir trois types de graphes. D'une part, le premier est **un graphe de dépendances d'un ensemble de méthodes**, c'est-à-dire un graphe orienté (V, E) dont les sommets V sont les méthodes et l'ensemble des arcs E contient un arc (a, b) si et seulement si la méthode a appelle la méthode b.

D'autre part, le second graphe est **le graphe de dépendances d'un ensemble de classes**, c'est-à-dire un graphe orienté et pondéré (V, E, w) dont les sommets sont les classes et l'ensemble des arcs contient un arc (a, b) si et seulement si la classe a contient une méthode qui appelle une méthode contenue dans b. La fonction de pondération associe un poids à chaque arc. Le poids de l'arc (a, b) est le nombre de méthodes de a qui appellent une méthode de b

Enfin, **le graphe de dépendances d'un ensemble de contours de composants** est un graphe orienté et pondéré (V, E, w) dont les sommets sont les contours de composants et l'ensemble des arcs contient un arc (a, b) si et seulement si le contour de composant contient une méthode qui appelle une méthode contenue dans b. La fonction de pondération est similaire à celle du graphe des dépendances d'un ensemble de classes.

Le choix des opérateurs à appliquer est aléatoire. Cependant, afin de réduire le diamètre du voisinage, le choix des opérations doit être précisé. Pour cela, nous utilisons les graphes de dépendances définis précédemment et nous détaillons les opérations suivant le même classement que les opérateurs.

- **le choix des opérations sur les contours de composants** utilise le graphe de dépendances des contours de composants et celui des classes. Ainsi, l'opérateur *fusion* s'applique entre les deux contours les plus couplés, c'est-à-dire dont le poids de l'arc est le plus important. L'opérateur *séparation* lui s'applique au sein d'un contour de composant. La « séparation » se fait au niveau du point d'association le plus faible dans le graphe de dépendances des classes du contour. On obtient ainsi les deux contours les moins couplés possibles. Enfin, l'opérateur *adoption* fait adopter une classe orpheline par le contour de composant avec lequel son contour est le plus couplé ;
- **le choix des opérations sur les contours d'interfaces** repose sur le graphe de dépendances d'un ensemble de méthodes. Les choix sont similaires à ceux concernant les opérateurs fusion et séparation des contours de composants. La fusion réunit les deux contours d'interfaces les plus couplés, alors que la séparation agit au niveau du point d'association le plus faible ;
- **le choix des opérations sur les contours de connecteurs** repose sur le graphe de dépendances des méthodes. Les opérations sur la fusion et la séparation sont identiques à celle des contours

d'interfaces. Les opérations sur la séparation et l'ajout sont aussi très proches. Dans ce cas, le graphe prend en compte toutes les méthodes du contour du connecteur et des contours de composants impliqués. La méthode concernée est la plus fortement ou faiblement couplée selon que se soit pour l'opérateur d'ajout ou de suppression.

6.3.3 Point de départ de l'exploration

Le dernier paramètre du recuit à choisir est son point de départ, *i.e.* la solution initiale à partir de laquelle le processus explore l'espace de recherche. On peut choisir aléatoirement ce point de départ mais il est également possible de choisir une solution pour cibler la recherche sur une zone particulière de l'espace de recherche.

Nous proposons deux types de points de départ. Le premier utilise les informations extraites des guides de l'extraction. Le deuxième utilise des heuristiques pour déterminer un point de départ correct. Ces points de départ sont utilisés, si les guides ne sont pas utilisables, comme choix par défaut à la place d'une solution initiale aléatoire.

6.3.3.1 Utilisation des guides de l'approche

Nous avons proposé dans le chapitre 5 deux modèles d'informations pour la réduction de l'espace de recherche. Celui pour le ciblage est particulièrement adapté à la définition du point de départ du recuit simulé. Les architectures intentionnelles extraites de la documentation ou des recommandations de l'architecte constituent un bon point de départ pour explorer l'espace de recherche. En effet, grâce au diamètre faible du voisinage, le processus recherche la solution à proximité d'une solution validée par l'architecte et la documentation. Il est donc probable que la solution retenue ressemble à l'architecture intentionnelle et soit donc plus satisfaisante pour l'architecte qu'une solution de la même qualité mais différente de ses attentes.

Nous laissons donc le choix à l'architecte de sélectionner une des architectures intentionnelles pour constituer le point de départ du recuit simulé. Cependant, les architectures intentionnelles extraites de la documentation considèrent les connecteurs comme des liens simples. Nous appliquons donc, avant le début du processus d'extraction, l'algorithme 1 d'identification des interfaces puis l'algorithme 2 d'identification des connecteurs.

Nous avons utilisé le modèle de suppression pour modifier les probabilités d'acceptation des solutions. Cependant, nous pouvons également l'utiliser pour cibler une solution. En effet, le fait de choisir une architecture intentionnelle limite l'utilisation de la documentation. Cependant, l'ensemble des architectures intentionnelles a été utilisé pour créer le réseau de contraintes. Par conséquent, une solution vérifiant le réseau de contraintes contient les informations issues de tous les types de documents utilisés.

Nous proposons donc d'utiliser un algorithme classique de résolution de réseau de contraintes hiérarchiques pour générer la solution respectant le plus de contraintes. Nous pouvons ensuite utiliser cette solution comme point de départ. Cette solution initiale représente alors l'ensemble des documents, les recommandations et le contexte de déploiement.

6.3.3.2 Utilisation d'heuristiques

Le point de départ du recuit simulé peut également s'appuyer sur une heuristique. Cette méthode présente l'avantage de pouvoir être utilisée même si aucune documentation ou recommandation n'est

disponible. La première heuristique que nous pouvons utiliser est celle utilisant le regroupement hiérarchique que nous avons proposée. Elle permet de préparer le paramétrage de la fonction objectif et de bénéficier d'une aide pour définir une solution initiale pour l'exploration.

Une autre heuristique utilise les composantes fortement connexes. Pour cela, nous définissons un graphe représentant le système. Les sommets de ce graphe représentent les classes du système et l'existence d'un arc entre deux sommets a et b signifie que la classe a utilise la classe b, *i.e.* utilise un attribut, un paramètre ou une variable de retour dont le type est b et crée ou utilise un objet dont le type peut être b. Une composante fortement connexe est un sous-ensemble de sommets dans lequel toute paire de sommets est reliée par un chemin. Ces composantes fortement connexes forment une partition des sommets du graphe et par conséquent une partition des classes du système.

Cette partition des classes en composantes fortement connexes a une complexité linéaire dans le nombre d'arêtes du graphe. En effet, l'algorithme, proposé par Tarjan [121], réalise l'identification des composantes de chaque classe en un parcours du graphe. Lors de ce parcours chaque classe est traitée une seule fois et chaque arête est traitée au plus deux fois.

Tous les éléments de cette partition sont un ensemble de classes qui partagent un fort couplage et sont plus couplés entre eux qu'avec les autres classes du système. Ainsi, par comparaison avec les travaux de COULANGE [33] proposant de commencer l'identification des composants réutilisables par la localisation des sous-ensembles connexes parmi les éléments du système, la partition obtenue constitue un ensemble pertinent de contours de composants. Nous pouvons ensuite appliquer nos algorithmes d'identification d'interfaces et de connecteurs pour définir une solution initiale pertinente de manière totalement automatique.

6.3.4 Analyse du processus par recuit simulé

Nous avons présenté une instance de notre approche basée sur l'algorithme de recuit simulé. Cette instance est une méta-heuristique qui explore l'espace des architectures possibles. Cependant, elle ne constitue pas forcément la solution parfaite pour résoudre notre problème d'extraction.

Avantages. Le recuit simulé est une méta-heuristique. A ce titre, il bénéficie de tous les avantages de ces approches en termes d'efficacité et de mise en pratique. Il nous permet également d'utiliser pleinement les méthodes de réduction de l'espace de recherche.

Comparé à d'autres méta-heuristiques, tel que l'exploration tabou, le recuit simulé est plus facile à mettre en place et consomme moins de ressources mémoires puisqu'il ne retient pas les solutions parcourues. L'itération suivante du processus est uniquement liée à la solution courante.

Enfin, le paramétrage de la décroissance de la température et du nombre d'opérations par itération permet de régler le temps d'exécution avec une certaine précision. Cette aspect ne concerne pas les heuristiques dont on évalue précisément le temps d'exécution mais les méta-heuristiques qui sont souvent difficiles à contrôler à ce niveau.

Inconvénients. L'inconvénient majeur du recuit est le temps de calcul nécessaire qui est important. Bien que la complexité du recuit en lui même soit raisonnable pour un problème d'optimisation, la complexité et le nombre des étapes de mesures ainsi que la complexité du choix des opérations font de ce processus d'extraction un algorithme très complexe.

Cependant, le temps disponible pour réaliser l'extraction est la plupart du temps suffisant pour que le temps de calcul ne soit pas considéré comme un facteur extrêmement limitant, surtout associé à la facilité de calibration du temps de calcul.

6.4 Instance de ROMANTIC à base d'algorithmes génétiques

Les algorithmes génétiques (GA) sont des méta-heuristiques relativement anciennes. Ils ont été introduit à la fin des années 60 par John Holland [65]. Eux aussi ont souvent été utilisés dans le domaine du génie logiciel [10, 59, 86, 111]. Ils sont basés sur la théorie de l'évolution de Darwin : les individus s'affrontent pour survivre et les plus adaptés ont les plus grandes chances de survivre et de se reproduire. L'idée de base des GA est de démarrer à partir d'un ensemble de solutions initiales et d'utiliser des mécanismes d'évolution inspirés par la biologie pour créer de nouvelles et potentiellement meilleures solutions.

Les GA sont des méta-heuristiques qui explorent l'espace des solutions d'un problème afin de trouver la meilleure solution. Cette exploration est basée sur la notion de population : à chaque itération, la méta-heuristique manipule un ensemble de solutions en parallèle.

Pour réaliser l'exploration, les GA n'utilisent pas de mémoire, c'est-à-dire que les solutions rencontrées n'ont pas d'influence sur les choix futurs. Ces choix reposent uniquement sur la population courante et une fonction statique, dite fonction objectif qui mesure l'adaptation d'un individu par rapport aux objectifs.

Nous utilisons les GA pour sélectionner la meilleure architecture au sens de notre fonction objectif (*cf.* Chapitre 4). Cependant, nous devons, pour utiliser les GA, préciser un ensemble d'éléments qui permettent de déterminer quel algorithme génétique nous utilisons. Ces éléments définissent la représentation d'une solution du problème ainsi que les mécanismes permettant de simuler l'évolution pendant l'exploration.

Nous prenons également en compte dans notre GA les guides auxiliaires que nous avons identifiés. Pour cela, nous devons intégrer les modèles d'information, extraits de la documentation et des recommandations, dans les éléments qui définissent le GA. Ainsi notre modélisation du problème et nos mécanismes d'évolutions doivent tenir compte du réseau de contraintes et des architectures intentionnelles extraites des guides que sont la documentation et les recommandations de l'architecte.

Dans la suite, nous présentons notre algorithme génétique d'extraction d'architectures logicielles. Pour cela nous présentons d'abord l'algorithme génétique générique. Nous étudions particulièrement les éléments qui définissent l'algorithme et quel impact ils ont sur l'exploration.

Ensuite, nous présentons le codage que nous utilisons pour modéliser notre problème et en particulier une solution dans le cadre des GA.

Nous présentons, ensuite un ensemble d'opérateurs génétiques. Il contient des opérateurs spécifiques à notre problème d'extraction qui utilisent donc des heuristiques pour rendre l'évolution plus efficace.

Après les opérateurs, nous terminons la présentation des éléments de notre algorithme génétique en décrivant la population initiale de notre processus. Nous précisons donc les éléments de la population mais aussi sa taille. Nous décrivons également comment l'architecte influe sur le choix de cette population.

Enfin, nous analysons cette instance de ROMANTIC en étudiant sa complexité, ses avantages et ses inconvénients.

6.4.1 Présentation des algorithmes génétiques

Selon la théorie de l'évolution [34], une population d'individus évolue à travers les générations. Cette évolution est fonction de l'adaptation des individus à leur environnement : les individus les plus adaptés

survivent plus longtemps et peuvent transmettre leurs patrimoines en se reproduisant. Cette transmission permet l'apparition de meilleurs individus à chaque génération.

Les algorithmes génétiques appliquent cette théorie aux problèmes d'optimisation. Ainsi, pour explorer l'espace des solutions, les GA codent une solution du problème sous la forme d'un chromosome. Ce chromosome est constitué d'un ensemble de gènes qui décrivent les différents attributs des solutions.

Ensuite, les GA débutent avec une population initiale de solutions qui évoluent à chaque itération en subissant un ensemble de manipulations de leurs chromosomes. Ces manipulations modélisent, entre autre, la reproduction, la sélection naturelle et la mutation qui existe dans le milieu naturel.

Enfin, la mesure de l'adaptation au milieu est, dans les GA, remplacée par une fonction objectif qui mesure la qualité de chaque solution en fonction des objectifs du problème d'optimisation.

L'ensemble de ces éléments, le codage de la solution, les manipulations et la fonction objectif, définissent un algorithme génétique spécifique. Mais l'algorithme général de l'exploration reste partagé par tous les GA.

L'algorithme générique des GA consiste essentiellement à manipuler une population de solutions, représentées par des chromosomes, au moyen d'un ensemble d'opérateurs dit génétiques (*cf.* Algorithme 3). A chaque génération, l'algorithme sélectionne un ensemble d'individus (Ligne 4). Cette première sélection détermine les individus qui participent au brassage génétique que constitue l'application des opérateurs. Le processus applique ensuite aux chromosomes sélectionnés les opérateurs génétiques selon une probabilité définie en entrée du processus (Ligne 5). Le nouvel ensemble de chromosomes est ensuite soumis à une nouvelle sélection pour déterminer les individus qui sont conservés dans la génération suivante (Ligne 6). Cette sélection peut, selon les cas, augmenter, réduire ou conserver le nombre d'individus dans la population. Enfin, la nouvelle génération est testée et la meilleure solution conservée (Ligne 7 à 9).

Algorithme 3 Algorithme génétique générique

ENTRÉES: P_0 une population initiale de solutions
SORTIES: $bestFitEver$ la meilleure solution rencontrée

1: $bestFit \leftarrow best\,(P_0)$
2: $bestFitEver \leftarrow bestFit$
3: $P \leftarrow P_0$
4: **pour** $t \leftarrow 0$ a T **faire**
5: $\quad Q \leftarrow select_1\,(P)$
6: $\quad Q' \leftarrow operate\,(Q)$
7: $\quad P \leftarrow select_2\,(P \cup Q')$
8: $\quad bestFit \leftarrow best\,(P)$
9: $\quad$ **si** $bestFit > bestFitEver$ **alors**
10: $\quad\quad bestFitEver \leftarrow bestFit$
11: $\quad$ **finsi**
12: **fin pour**
13: **return** $bestFitEver$

Les opérateurs génétiques représentent les évènements qui, lors de la reproduction, entre en jeu au niveau des chromosomes. Ils sont principalement de trois types : les opérateurs de croisement, les opérateurs de mutation et les opérateurs de selection. Chacun modélise un aspect particulier du processus naturel. Ils offrent ainsi chacun un service différent à l'exploration.

Les opérateurs de croisement représentent le brassage des chromosomes entre les deux parents afin de créer les chromosomes des enfants. Ces opérateurs agissent donc sur deux individus afin de créer un ou plusieurs nouveaux individus par recombinaison. Ces opérateurs permettent l'échange de gènes entre les individus et constituent le mécanisme principal de l'exploration. Par conséquent, la probabilité d'application de cette opérateur est très forte et dépend généralement de la qualité de l'individu. On favorise ainsi la conservation des meilleurs gènes.

Cependant, ces opérateurs de croisement ne peuvent manipuler que les gènes existants dans la population. Or, le processus, comme dans la nature, peut avoir besoin de certains gènes absents de la population initiale pour parvenir à la solution optimale. Pour permettre l'apparition de ces nouveaux gènes, le deuxième type d'opérateurs représente des mutations des chromosomes. Ces opérateurs consistent, la plupart du temps en des modifications aléatoires du contenu d'un ou plusieurs gènes dans un chromosome. Ce caractère aléatoire oblige à utiliser ces opérateurs avec plus de parcimonie que ceux de croisement. La probabilité d'appliquer ces opérateurs est donc traditionnellement beaucoup plus faible.

Enfin, les opérateurs de sélection représentent le processus de sélection naturel. Ils ne sont pas appliqués au même moment de l'algorithme et ont une probabilité d'utilisation de 100%. Ils doivent selectionner les individus qui vont subir les opérateurs et ceux qui sont conservés d'une génération à l'autre. Une utilisation courante est de selectionner les n premiers meilleurs individus afin de conserver une population stable et éviter l'explosion de la population du à la génération d'enfants par les opérateurs de croisement.

Afin de définir notre algorithme génétique, nous devons définir les différents éléments de spécification : le codage, les opérateurs, la population initiale et la fonction objectif. Nous utilisons, bien sûr, notre fonction objectif dans ce processus.

Par contre, nous décrivons, dans le reste de cette section, le codage d'une architecture sous la forme d'un chromosome ainsi que les opérateurs génétiques : croisement, mutation et sélection. Nous établissons également comment la population initiale de l'algorithme est générée.

6.4.2 Codage de l'architecture

Nous avons proposé, dans le chapitre 3, un modèle de solutions de l'extraction d'architectures logicielles. Ce modèle COA représente une solution comme un ensemble de contours qui sont des regroupements d'entités objets. Ainsi, l'algorithme de regroupement hiérarchique nous propose une instance de ce modèle. De façon similaire, l'algorithme de recuit simulé nous propose une instance de ce modèle obtenue grâce à une exploration de l'espace des instances possibles.

Malheureusement, les GA imposent un méta-modèle spécifique pour les solutions. En effet, celles-ci doivent être représentées sous la forme de chromosomes. c'est-à-dire un ensemble de gènes qui décrivent chacun une propriété de la solution. Le problème est que notre modèle COA ne peut être décrit directement comme une instance de ce méta-modèle.

En conséquence, nous proposons une représentation de notre modèle COA sous la forme de chromosomes. De plus, pour permettre l'utilisation des autres aspects de notre approche, nous devons également proposer un ensemble de transformations permettant de passer d'une représentation à l'autre du modèle COA.

Une solution du problème d'extraction est une représentation de l'architecture et en particulier des éléments architecturaux. Nous représentons donc dans notre modèle les trois éléments architecturaux que sont les composants, les connecteurs et la configuration.

L'utilisation traditionnelle des GA propose de coder chaque solution sous la forme d'un seul chromosome. Cette approche permet de proposer des opérateurs génétiques qui opèrent sur un seul chromosome et accèdent à l'ensemble de la solution. Cependant, nous devons décrire cette fois trois éléments distincts : la configuration, l'ensemble des composants et l'ensemble des connecteurs. Ces éléments sont très différents et peuvent nécessiter des opérations différentes et surtout disjointes, c'est-à-dire que les opérations sur un type d'élément architectural ne doivent pas être dépendantes de celles portant sur un autre type d'élément.

Nous proposons donc de coder chacun de ces éléments par un chromosome et d'associer des opérateurs génétiques spécifiques à chaque chromosome. Ainsi, un individu de l'espace de recherche est représenté par un ensemble de trois chromosomes : le chromosome des composants, le chromosome des connecteurs et le chromosome de la configuration.

6.4.2.1 Chromosome des composants

Le chromosome des composants représente l'ensemble des composants de l'architecture. Il correspond, dans notre modèle COA à l'ensemble des contours de composants. Il doit donc représenter les mêmes entités objets et respecter les mêmes contraintes que les contours de composants. Ainsi, le chromosome doit être une partition des classes du système. Il doit également décrire les contours d'interfaces associés à chaque contour de composants.

Nous pouvons donc représenter ce chromosome de deux manières : un gène représente une classe et sa valeur représente le contour contenant cette classe ; un gène représente un contour et sa valeur représente les classes contenues dans le contour. La première représentation présente plusieurs inconvénients. D'abord, il est difficile dans cette représentation de vérifier que les contours forment une partition des classes. En effet, la complétude de la partition, c'est-à-dire le fait que chaque classe est présente au moins une fois est facilement vérifiable, mais la validité, c'est-à-dire le fait que chaque classe soit présente une seule fois, est plus complexe. Ceci nécessite un parcours de l'ensemble des classes et donc des gènes. De plus, l'ajout des contours d'interfaces nécessite d'ajouter des gènes pour représenter chaque contour d'interfaces, chaque gène contenant les méthodes de l'interface et le numéro du contour de composant associé.

Au contraire, la seconde représentation permet de vérifier facilement les propriétés de l'ensemble des contours. En effet, il suffit de vérifier que l'union des gènes représente toutes les classes du système et que les intersections des gènes deux à deux sont vides. De plus, nous pouvons sans problème ajouter la liste des contours d'interfaces à la liste des classes dans la représentation de chaque contour, c'est-à-dire dans le gène.

Au final, le chromosome des composants est un ensemble de gènes. Chaque gène représente un contour de composants. Il contient l'ensemble des classes qui constituent le contour et l'ensemble des contours d'interfaces associés à ce contour.

6.4.2.2 Chromosome des connecteurs

Le chromosome des connecteurs représente l'ensemble des connecteurs de l'architecture. Il correspond, dans notre modèle COA à l'ensemble des contours de connecteurs. Il doit donc représenter les mêmes entités objets et respecter les mêmes contraintes que les contours de connecteurs. Cependant, les contours de connecteurs ne sont pas soumis à des contraintes spécifiques. Par contre, ils sont impliqués dans des contraintes qui relèvent plutôt de la configuration puisqu'elles concernent les relations entre les contours de composants et les contours de connecteurs. Ainsi, le chromosome doit être un ensemble de

groupe de méthodes. Ces groupes doivent décrire, pour chaque contour, les méthodes contenues dans le contour et les méthodes requises par le contour.

Nous pouvons donc représenter ce chromosome de deux manières : un gène représente une méthode et sa valeur représente le contour contenant cette méthode ; un gène représente un contour et sa valeur représente deux ensembles de méthodes, les contenues et les requises. Le premier choix présente un inconvénient majeur : il impose une taille constante au chromosome qui est le nombre de méthodes existantes dans le système. Le problème est que, dans la plupart des cas, le nombre de méthodes est beaucoup plus important que le nombre de méthodes impliquées dans les contours de connecteurs. En effet, les méthodes concernées sont celle qui sont impliquées dans les interactions entre contours de composants. Or, notre fonction objectif vise, entre autre, à réduire ces interactions et donc le nombre de méthodes concernées. En conséquence, un tel codage conduit à un nombre important de gènes inutiles.

Au contraire, la seconde représentation est optimale en terme de quantité d'information puisqu'elle représente uniquement les méthodes impliquées dans les connecteurs. Elle permet de représenter l'ensemble des méthodes incluses et celui des méthodes requises par le contour. Ceci facilite la vérification des contraintes relevant du niveau de la configuration et portant à la fois sur les composants et les connecteurs. Elle facilite aussi la mise en relation dans la configuration des connecteurs et des composants.

Au final, le chromosome des connecteurs est un ensemble de gènes. Chaque gène représente un contour de connecteur. Il contient l'ensemble des méthodes qui constituent le contour et l'ensemble des méthodes requises par le contour.

6.4.2.3 Chromosome de la configuration

Le chromosome de la configuration représente la configuration de l'architecture, c'est-à-dire la topologie des connexions entre les composants et les connecteurs de l'architecture. Il correspond, dans notre modèle COA au contour de la configuration. Il doit donc représenter les mêmes entités objets et respecter les mêmes contraintes que ce contour.

Les contraintes sur le contour de configuration portent sur l'adéquation entre les connecteurs et les composants qui sont reliés dans la configuration. Les interfaces fournies des composants doivent être utilisées par les connecteurs et les interfaces requises doivent être proposées par les connecteurs. Le chromosome de configuration doit donc représenter les relations entre connecteurs et composants qui sont conformes aux contraintes d'intégrité de l'architecture.

Nous pouvons donc représenter ce chromosome de deux manières : un gène représente un contour de composant et sa valeur représente les contours de connecteurs qui lui sont reliés ; un gène représente un contour de connecteur et sa valeur représente les contours de composants qu'il relie. Les deux représentations présentent l'inconvénient de devoir dupliquer certains contours. En effet, comme un connecteur peut être relié à plusieurs composants et qu'un composant peut également être relié à plusieurs connecteurs, les deux représentations imposent de mettre le connecteur ou le composant partagé à l'intérieur de différents gènes. Cependant, la deuxième représentation permet de vérifier pour chaque contour de connecteur si l'ensemble des composants reliés sont valides du point de vue de la correspondance des interfaces.

Au final, le chromosome de la configuration est un ensemble de gènes dont le premier représente le nombre de connecteurs utilisés dans l'architecture, c'est-à-dire le nombre de gènes du chromosome en plus du gène courant. Chaque autre gène représente un contour de connecteur utilisé dans l'architecture. Le gène contient l'ensemble des contours de composants reliés à ce contour.

6.4.3 Définitions des opérateurs génétiques

Un algorithme génétique nécessite la définition d'au moins trois opérateurs : un opérateur de sélection, de mutation et de croisement. Ces opérateurs permettent l'exploration de l'espace de recherche en créant de nouvelles solutions par combinaisons ou modifications de solutions sélectionnées.

Dans les premières utilisations des algorithmes génétiques, ces opérateurs étaient indépendants du codage et de la sémantique de la solution [65]. Le codage était alors une suite de 0 et de 1 et les opérateurs de croisement et de mutation consistaient simplement à croiser les séquences ou à changer un 1 en 0. D'autre approches ont ensuite proposé des opérateurs qui s'adaptent au codage et aux contraintes du problème [41]. Enfin, certaines approches utilisent la sémantique associée au codage pour déterminer exactement l'action des opérateurs, en déterminant par exemple quel gène va subir une mutation en fonction de critères sémantiques ou de la fonction objectif [111].

Nous proposons un ensemble d'opérateurs qui respectent le codage que nous avons choisi. Ils respectent donc la partition que doit former les contours de composants et la validité de la configuration. Ils utilisent aussi la fonction objectif et la sémantique du codage pour déterminer précisément leurs actions.

Nous présentons donc successivement les opérateurs de sélection, de croisement et de mutation que nous utilisons dans notre algorithme génétique.

6.4.3.1 Opérateurs de sélection

Les opérateurs de sélection sont utilisés à deux reprises dans l'algorithme génétique. En effet, il faut d'abord sélectionner les chromosomes qui vont être soumis aux opérateurs de croisement et de mutation, puis il faut sélectionner ceux qui seront conservés dans la génération suivante. Ces deux sélections ne partagent donc pas les mêmes objectifs et ne sont pas soumis aux mêmes critères de choix.

La première sélection détermine les paires de chromosomes qui vont être croisées au moyen des opérateurs de croisement. Cette sélection simule la compétition qui existe dans la nature pour accéder à la reproduction. Ainsi, on peut imaginer sélectionner les chromosomes en fonction de leurs fonction objectif. On sélectionne donc les paires de chromosomes dans le classement des chromosomes selon le résultat de la fonction objectif et on regroupe les éléments successifs dans le classement. Les chromosomes issus de la reproduction sont alors uniquement issus des meilleurs chromosomes. Malheureusement cette méthode risque de réduire la diversité génétique en favorisant à chaque itération la même branche d'individus et en favorisant les croisements au sein de la même branche.

Pour éviter la perte de diversité, nécessaire au bon déroulement de l'évolution, cette sélection doit donc favoriser la reproduction des individus les plus adaptés, mais elle doit également permettre aux autres individus de se reproduire. Ainsi, un choix basique pour cet opérateur est de choisir au hasard chaque élément des paires de chromosomes. Mais ce choix provoque la dilution des meilleurs gènes dans l'ensemble de la population. On risque ainsi d'obtenir une population d'individus médiocres mais qui portent chacun un ensemble réduit de gènes de qualité et d'autres de mauvaise qualité.

Par conséquent, nous proposons de sélectionner les chromosomes en favorisant les meilleurs mais en conservant aux autres l'opportunité d'être croisés. Pour cela, nous réalisons la sélection des paires de chromosomes selon **la technique de la roulette** [65]. Cette méthode consiste à placer tous les chromosomes sur une roulette imaginaire. Chaque chromosome bénéficie d'une portion de la roue qui est proportionnelle à son adaptation, c'est-à-dire à son évaluation par notre fonction objectif. Ensuite, on lance une boule dans la roulette et on sélectionne le chromosome où la boule s'est arrêtée.

le premier opérateur de selection utilise donc cette technique pour selectionner 75 % des individus de la population. L'ordre de la selection permet ensuite de déterminer les couples qui vont être soumis à l'opérateur de croissement.

La seconde sélection détermine les chromosomes qui sont conservés dans la génération suivante. Le choix de cet opérateur a une influence directe sur la complexité de l'algorithme. En effet, selon le nombre de chromosomes que l'opérateur sélectionne, la population peut décroitre progressivement, rester constante ou subir une explosion démographique. Ainsi, si l'opérateur sélectionne l'ensemble des chromosomes, les enfants nouvellement créés s'ajoutent aux parents et aux autres chromosomes existants et provoquent une augmentation de la population à chaque itération. Au contraire, si l'opérateur sélectionne moins de chromosomes que dans la population initiale, la population va lentement décroitre. Au finale, l'approche la plus courante est de conserver une population constante en sélectionnant un nombre de chromosomes constant à chaque itération. Ce nombre correspond souvent à la taille de la population initiale.

Cet opérateur simule la survie des individus, les meilleurs ayant le plus de chances de survie. Ainsi, on peut imaginer les mêmes options que pour la première sélection. Cependant, la sélection aléatoire et la sélection des meilleurs présentent, dans ce cas, le même écueil que pour la première sélection. Ces deux sélections favorisent trop ou pas assez les bons chromosomes.

Par conséquent, nous proposons une sélection basée sur deux critères : l'âge des chromosomes et leur qualité. Nous utilisons donc encore la technique de la roulette à la différence que la portion de roue associée à un individu dépend maintenant de sa qualité et de son age. Cette portion est en fait proportionnelle au rapport entre la qualité et l'age des individus.

Le second opérateur de selection utilise donc cette technique pour selectionner une nouvelle population parmi l'ensemble des individus de l'ancienne génération et des individus nouvellement créés. La nouvelle population conserve la même taille que la population intiale.

6.4.3.2 Opérateurs de croisement

Les opérateurs de croisement permettent de combiner les individus pour obtenir de nouvelles combinaisons de gènes avec un meilleur résultat pour notre fonction objectif. Ils explorent ainsi l'espace de recherche accessible à partir de l'ensemble des gènes disponibles.

Ayant codé une architecture sous la forme de trois chromosomes, il semble difficile de proposer un simple opérateur de croisement pour gérer l'ensemble des chromosomes. En conséquence, nous devons décomposer cette opération en trois opérations : le croisement de chromosomes de composants, de connecteurs et de configuration. Cependant, le chromosome de configuration est étroitement lié par sa construction et les contraintes aux deux autres. Chaque opération portant sur lui peut donc être remplacée par deux opérations partant chacune sur un chromosome.

Au final, nous proposons deux opérateurs de croisement : l'un réalise le croisement des chromosomes de composant et l'autre des chromosomes de connecteurs, chacun de ces opérateurs ayant en plus un impact sur le chromosome de configuration.

Ces croisements doivent prendre en compte les propriétés et les contraintes de chacun des chromosomes. Ils ne peuvent donc pas être l'application d'un seul opérateur à deux opérandes. Nous détaillons donc séparément l'opérateur de croisement des chromosomes de composant et celui des chromosomes de connecteurs. Nous précisons à chaque fois les effets de l'opérateur sur le chromosome de configuration.

Opérateur de croisement des chromosomes de connecteur. Nous utilisons un opérateur standard pour croiser deux chromosomes de connecteurs. Cet opérateur coupe chacun des deux parents en deux

sous-ensembles de gènes. Les enfants, c'est-à-dire les deux nouveaux chromosomes, sont ensuite crées en croisant les sous-ensembles comme sur la figure 6.4.

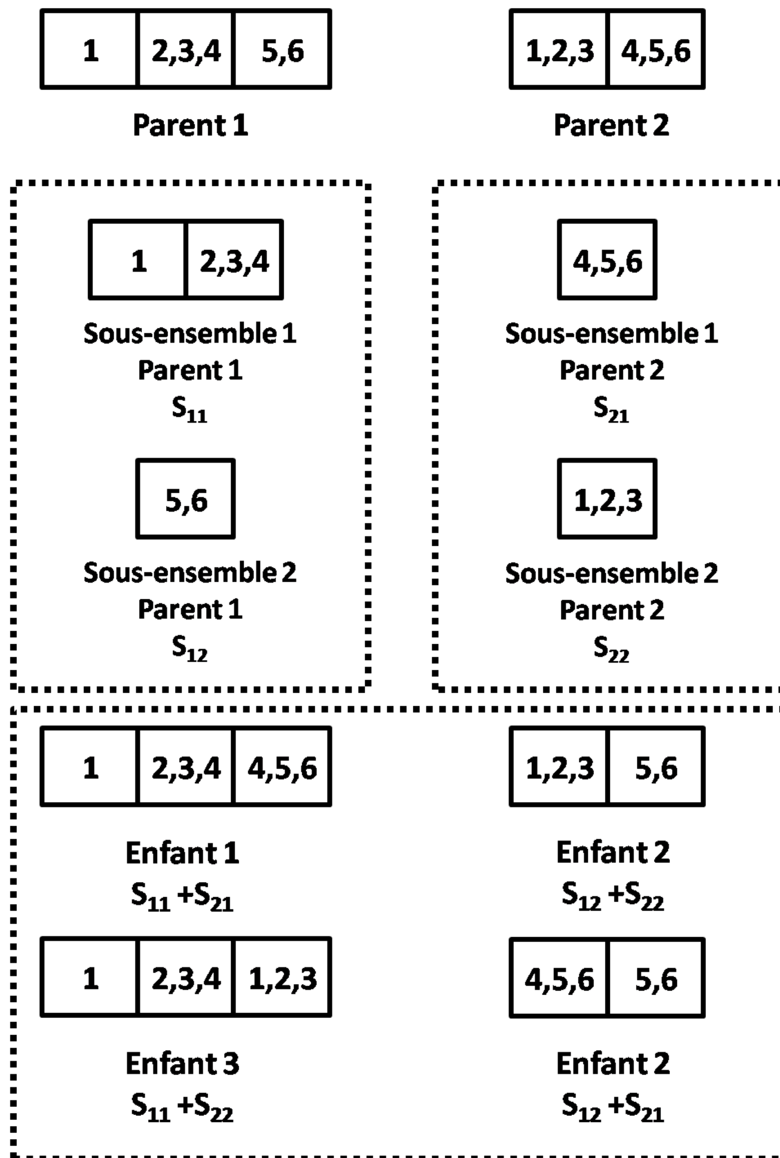


Figure 6.4 – Utilisation du croisement standard

Les connecteurs de l'architecture ayant changés nous recalculons ensuite les correspondances entre les composants et les connecteurs pour mettre à jour la configuration et son chromosome. Cependant, cette mise à jour peut être impossible en l'état. En effet, dans cette opération, les connecteurs sont échangés sans considération pour les composants de l'architecture. Cela engendre plusieurs conséquences sur le chromosome des connecteurs et sur celui de la configuration :

- **manque de connecteurs** : les nouveaux connecteurs disponibles peuvent être insuffisants pour compléter l'architecture. Nous créons donc, pour chaque interface fournie, un connecteur vide

qui fournit cette interface à tous les composants qui requièrent ce service. Nous ajoutons ensuite les gènes correspondant dans le chromosome de configuration. Nous n'ajoutons pas ces gènes dans le chromosome des connecteurs afin de ne pas échanger ces connecteurs lors des prochaines itérations ;

- **excès de connecteurs** : les nouveaux connecteurs peuvent ne pas être utiles pour l'architecture. Dans ce cas, ces connecteurs ne sont pas représentés dans le chromosome de configuration. Comme nous calculons la fonction objectif en fonction de la configuration, ces connecteurs n'ont pas d'impact sur la qualité de l'architecture. Par contre, leurs gènes restent présents dans le chromosome des connecteurs. Ils peuvent ainsi apparaitre à nouveau dans un individu qui nécessite ce connecteur et participer à l'architecture ;
- **duplication de connecteurs** : les nouveaux connecteurs peuvent être dupliqués dans le nouveau chromosome. En effet, il est possible qu'il existe plusieurs contours de connecteurs qui requièrent les mêmes ensembles de méthodes sans tenir compte des interfaces. Dans ce cas et quel que soit le contenu des contours, nous regroupons les connecteurs clones. Nous créons ainsi un contour de connecteur qui contient l'union des contours de connecteurs clones. Les méthodes requises du contour sont celles requises par l'ensemble des contours de connecteurs clones. Ensuite, nous supprimons les gènes correspondant aux connecteurs clones et nous ajoutons le gène correspondant au nouveau contour.

Opérateur de croisement des chromosomes des composants. Le chromosome des composants représente l'ensemble des contours de composants de l'architecture. A ce titre, il doit, d'après notre définition des contours de composants, respecter la contrainte que les gènes forment une partition des classes du système.

L'application de l'opérateur standard sur le chromosome des composants peut résulter en un chromosome qui ne forme plus une partition des classes. En effet, deux problèmes spécifiques peuvent se produire (*cf.* Figure 6.5). Si l'intersection de deux contours n'est pas vide, on dit que le chromosome est inconsistant. Le second problème apparait lorsque le chromosome est incomplet, c'est-à-dire que l'union de l'ensemble des contours ne contient pas l'ensemble des classes du système. Dans les deux cas, l'ensemble des contours de composants ne représente pas une partition.

Pour préserver la consistance et la complétude des enfants, nous proposons un opérateur de croisement basé sur l'opérateur défini pour les problèmes de regroupement [41] (*cf.* Figure 6.6). Pour obtenir un enfant, nous sélectionnons un sous-ensemble de gènes d'un des parents et nous ajoutons ces gènes au chromosome du second parent. Le fait de conserver tous les gènes d'un des parents nous assure la complétude de l'enfant. Afin de garantir la consistance, nous éliminons les classes contenues dans les nouveaux gènes des anciens gènes (les nouveaux gènes « récupèrent » les classes des anciens gènes).

Pour terminer l'opération de croisement, nous procédons à une vérification de la configuration. Comme pour le traitement du chromosome des connecteurs, il est possible que des connecteurs deviennent inutiles ou manquants. Nous procédons alors de la même manière que lors du croisement des connecteurs.

6.4.3.3 Opérateurs de mutations

Les opérateurs de mutations permettent l'apparition de nouveaux gènes. Ils évitent ainsi à l'exploration d'être limitée à une zone réduite de l'espace de recherche en fonction des opérateurs de croisement et surtout de la population initiale. Ces opérateurs, comme la mutation génétique qu'ils modélisent, représentent des modifications aléatoires des chromosomes. Ils s'apparentent ainsi aux opérateurs de

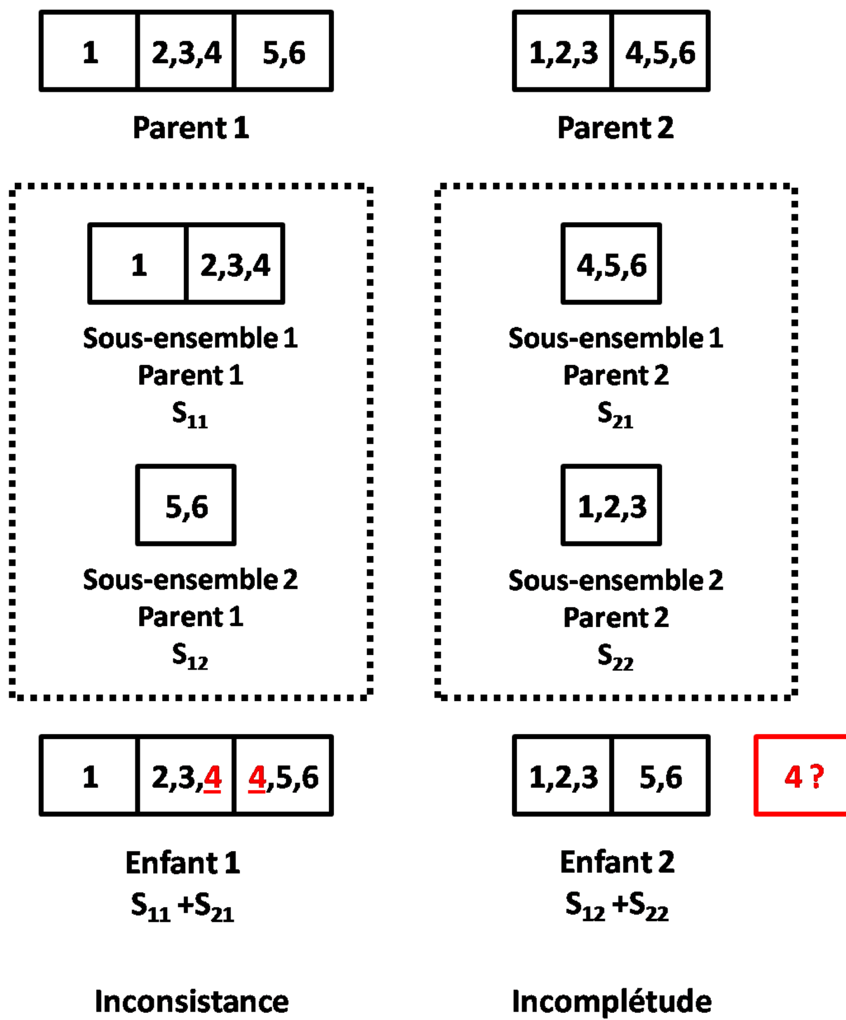


Figure 6.5 – Limites du croisement standard

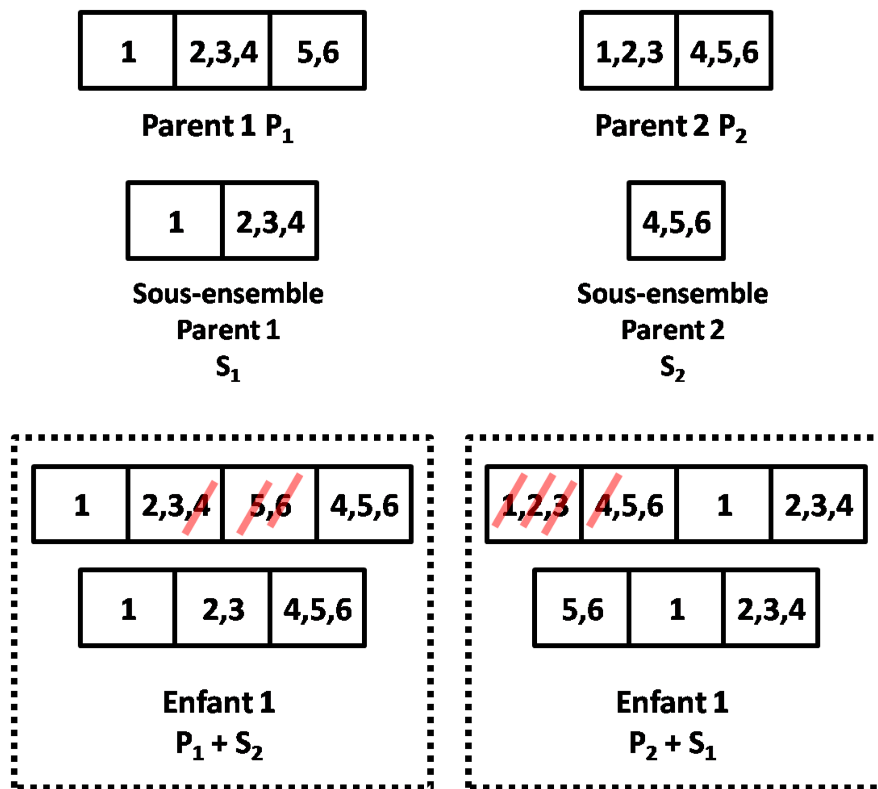


Figure 6.6 – Croisement préservant la complétude et la consistance

manipulations du recuit simulé qui visent à modifier les solutions existantes pour faire apparaitre de nouvelles solutions.

Notre codage de l'architecture utilise trois chromosomes par individu. Nous avons choisi de proposer plusieurs opérateurs de mutations en fonction du chromosome sur lequel ils s'appliquent. Chacun de ces opérateurs s'appuie sur les opérations de manipulations que nous avons décrites dans le cadre du recuit simulé. Par conséquent, nous ne proposons pas de mutation pour le chromosome de configuration.

Concernant la probabilité de l'apparition des mutations, elles doivent se produire avec une probabilité faible afin de ne pas dénaturer le processus et modifier profondément les individus qui composent la population. De plus, les probabilités d'apparition des mutations sur chaque chromosome sont indépendantes. Ainsi, si la probabilité de mutation est de p_m, la probabilité de mutation sur un chromosome d'un individu est de p_m et celle d'un individu de $3 * p_m$.

Mutations du chromosome composant. La mutation du chromosome composant peut avoir deux cibles. Elle peut modifier les contours de composants ou les contours d'interfaces. Nous proposons donc de définir trois opérations de mutations qui peuvent être rapprochées des opérateurs de manipulations que nous avons définis sur les contours de composant et d'interface pour le recuit simulé. La probabilité d'application de chacun des opérateurs est de $\frac{p_m}{3}$.

Nous définissons un premier opérateur de mutation qui opère sur les contours de composants. Il correspond à l'union des opérateurs de manipulation du recuit « séparation » et « fusion » de contours de composants. L'opérateur de mutation applique l'une ou l'autre des opérations de manière aléatoire. L'application suit ensuite le même processus que dans le recuit. La seule exception est que l'on modifie les chromosomes au lieu de modifier l'instance du modèle COA. Cette opération a donc un impact sur le chromosome connecteur dont on modifie les gènes correspondant aux contours de connecteurs associées aux contours de composants.

Le deuxième opérateur de mutation du chromosome composant opère sur les contours d'interfaces associés à chaque gène. Il correspond à l'union des opérateurs de manipulation du recuit « séparation » et « fusion » de contours d'interfaces. Comme pour les contours de composants, l'opérateur de mutation applique l'une ou l'autre des opérations de manière aléatoire et de la même façon que dans le recuit.

Enfin, nous ajoutons un dernier opérateur de manipulation qui reflète l'opérateur adoption du recuit. Il permet d'éviter qu'un contour de composant contenant uniquement une classe pollue les résultats. Il fonctionne de la même manière que l'opérateur de mutation $fusion$ appliquer au contour orphelin et à celui qui est le plus couplé avec cette classe.

Mutations du chromosome connecteur. Les mutations du chromosome connecteur peuvent être divisés en deux groupes. Le premier groupe modifie uniquement les contours de connecteurs, alors que les modifications du deuxième ont aussi un impact sur les contours d'interfaces des contours de composants associés. Nous proposons donc de définir deux opérations de mutation qui peuvent être rapprochées des opérateurs de manipulation que nous avons définis pour les contours de connecteurs dans le recuit simulé. La probabilité d'application de chacun des opérateurs est de $\frac{p_m}{2}$.

Nous définissons un premier opérateur de mutations qui opère uniquement sur les contours de connecteurs. Il correspond à l'union des opérateurs de manipulations du recuit « séparation » et « fusion » de contours de connecteur. L'opérateur de mutations applique l'une ou l'autre des opérations de manière aléatoire. L'application suit ensuite le même processus que dans le recuit mais en s'appliquant sur le chromosome des connecteurs au lieu d'une instance de COA.

Le deuxième opérateur de mutations du chromosome des connecteurs opère également sur le chromosome des composants. Il correspond à l'union des opérateurs de manipulation du recuit « ajout » et

« suppression » de contours de connecteurs. Comme pour les contours de composants, l'opérateur de mutations applique l'une ou l'autre des opérations de manière aléatoire et de la même façon que dans le recuit. L'opération implique une modification des gènes du chromosome des composants qui sont référencés par le chromosome de configuration comme étant reliés au gène muté. Cette modification consiste en une modification des contours d'interfaces du gène suivant le processus décrit dans la section 6.3.

6.4.4 Choix de la population initiale

La population initiale de l'algorithme génétique est une donnée cruciale. Elle fournit les pièces qui doivent permettre d'atteindre la meilleure solution. Il convient donc que cette population fournisse une grande variété de gènes afin d'augmenter la taille de l'espace accessible sans mutation. En conséquence, cette population est souvent générée de manière aléatoire, mais en respectant une distribution précise des solutions pour couvrir une zone suffisamment large de l'espace de recherche.

Cependant, la génération aléatoire ajoute dans les gènes disponibles des gènes de qualité quelconque. La zone accessible de l'espace de recherche est donc étendue dans toutes les directions, y compris vers les mauvaises solutions. Il est donc souhaitable de favoriser la présence de bon gène dans la population initiale, afin d'orienter l'extension de la zone accessible vers les meilleures solutions.

Il est, cependant, essentiel de ne pas fournir uniquement des solutions calibrées et identiques provenant de la même source. Il faut, au contraire, fournir une population initiale diverse, provenant de différentes sources. Cette population doit comporter de bonnes solutions et d'autres mauvaises afin de conserver une diversité génétique suffisante.

Selon ces considérations, nous proposons d'utiliser une population initiale contenant à la fois des solutions générées de manière aléatoire et des solutions calculées à partir de différentes sources.

Les solutions calculées sont issues d'une large variété de sources pour ne pas nuire à la variété génétique. Ainsi, nous utilisons les guides de l'extraction que nous avons identifiés dans le chapitre 5. Nous utilisons également les autres approches d'exploration que nous avons proposées pour générer des solutions pertinentes.

6.4.4.1 Individus basés sur les guides de l'extraction

La documentation et les recommandations de l'architecte sont des guides essentiels pour extraire une architecture pertinente du système. Nous avons utilisé ces éléments pour obtenir deux modèles d'informations qui nous permettent respectivement de supprimer certaines solutions et d'en cibler d'autres.

Nous utilisons donc les modèles de ciblage et de suppression pour générer des solutions. Ces solutions ont un génome de qualité du point de vue la documentation , de l'architecte ou du contexte. Nous devons donc générer des solutions à partir de ces modèles d'information.

Modèle de ciblage. Les architectures intentionnelles extraites de la documentation avec l'aide de l'architecte montrent différentes vues architecturales qui sont toutes en accord avec un certain aspect de la documentation ou de l'architecte. Si ces architectures intentionnelles sont insuffisantes pour constituer la solution de l'extraction d'architectures, elles sont suffisamment pertinentes pour fournir un génome de qualité à notre approche par algorithme génétique.

Cependant, comme dans l'approche par recuit simulé, nous devons compléter ces architectures intentionnelles. En effet, elles considèrent les connecteurs comme des liens simples. Nous appliquons donc, avant le début du processus d'extraction, l'algorithme 1 d'identification des interfaces puis l'algorithme 2 d'identification des connecteurs.

Modèle de suppression. Le modèle de suppression regroupe les informations issues de la documentation, des recommandations de l'architecte et du contexte de déploiement. Cette synergie des sources modifie les informations proposées par le modèle en comparaison avec les architectures intentionnelles basées uniquement sur la documentation. Ce modèle d'informations offre donc une source supplémentaire pour la génération de la population initiale.

Nous proposons donc, comme dans le recuit simulé, d'utiliser un algorithme classique de résolution de réseau de contraintes hiérarchiques pour générer la solution respectant le plus de contraintes. Nous pouvons ensuite utiliser cette solution comme individu au sein de la population initiale de notre algorithme génétique.

6.4.4.2 Individus basés sur les autres approches par exploration

Une autre option pour générer des individus pertinents et diversifiés, consiste à utiliser les heuristiques et les autres approches que nous avons déjà proposées. En effet, nous avons proposé plusieurs techniques d'extraction ou d'identification d'architectures. Chacune présente des performances différentes, mais toutes permettent d'extraire une architecture pertinente et une vue légèrement différente de l'architecture du système étudié.

Ainsi, nous pouvons utiliser les résultats de l'extraction d'architectures réalisée avec l'algorithme de regroupement hiérarchique. En effet, cet algorithme a une complexité faible et des bons résultats, il peut donc être utilisé dans une phase antérieure à l'algorithme génétique pour générer un individu de la population initiale.

Nous pouvons aussi envisager d'utiliser l'algorithme de recuit simulé pour générer un ou plusieurs individus de la population initiale. Cependant, la complexité du recuit étant importante, cette solution n'est pas aussi évidente que celle du regroupement hiérarchique. Il convient de bien évaluer le rapport entre le gain pour l'algorithme génétique et le coût de calcul de l'algorithme de recuit. Ainsi, dans la plupart des cas, l'apport du recuit est insuffisant puisque ces résultats ne sont pas suffisamment différents de ceux du regroupement pour justifier le temps de calcul.

Par contre, nous avons proposé pour le recuit simulé, un point de départ basé sur les composantes fortement connexes des classes du système. Cette solution a une complexité linéaire et fournit une solution suffisamment pertinente pour être le point de départ unique du recuit. Nous utilisons donc également cette solution dans la population initiale de l'algorithme génétique.

6.4.5 Analyse du processus par algorithme génétique

Nous avons présenté une instance de notre approche basée sur un algorithme génétique. Cette instance est une méta-heuristique qui explore l'espace des architectures possibles. Cependant, il ne constitue pas forcément la solution parfaite pour résoudre notre problème d'extraction. En effet, comme le recuit simulé et le regroupement hiérarchique, l'approche par algorithme génétique présente un ensemble de qualités et de défauts.

Les GA, comme le recuit simulé, sont une méta-heuristique. A ce titre, ils bénéficient de tous les avantages de ces approches en termes d'efficacité et de mise en pratique. Ils nous permettent également d'utiliser pleinement les méthodes de réduction de l'espace de recherche.

Cependant, cet avantage est atténué par le méta-modèle spécifique des algorithmes génétiques. En effet, les autres méta-heuristiques autorisent toutes une grande liberté dans le codage des solutions. Au

contraire, le méta-modèle spécifique des algorithmes génétiques impose un codage particulier et constitue donc l'inconvénient majeur de ces algorithmes. Ces contraintes sur le codage de solutions se reportent également sur les opérateurs qui permettent l'exploration.

Par conséquent, l'algorithme génétique permet une utilisation plus importante des guides de notre approche. Mais il nécessite cependant, une adaptation de ces guides pour les rendre compatibles avec la représentation à base de chromosomes.

Comme le recuit simulé, la complexité du processus génétique est importante. La source de la complexité est encore une fois la mesure de la fonction objectif ainsi que l'application des opérateurs. A ceci s'ajoute la taille de la population qui doit être d'une taille suffisamment importante pour avoir une bonne diversité génétique mais qui fait augmenter encore la complexité.

Mais encore une fois, le temps de calcul n'est pas un facteur extrêmement limitant. d'autant plus que les algorithmes génétiques présentent aussi l'avantage de pouvoir être facilement distribués, ce qui permet de réduire le temps de calcul. Enfin, d'après nos tests, l'algorithme génétique offre une vitesse de convergence supérieur à celle du recuit simulé. En effet, une augmentation du temps de calcul résulte en une augmentation de la qualité d'un facteur supérieur à celui du recuit simulé.

6.5 Conclusion

Nous avons présenté dans ce chapitre plusieurs mises en pratique de notre approche ROMANTIC. La première utilise une heuristique de regroupement hiérarchique pour extraire les composants puis l'architecture. Cette instance de ROMANTIC, nous a surtout permis de tester notre approche sur un algorithme non-stochastique et facilement maitrisable. Nous avons également proposé deux instances de ROMANTIC basées sur des méta-heuristiques : le recuit simulé et l'algorithme génétique

Chacune des méta-heuristiques que nous avons décrites utilise pleinement l'ensemble des guides détaillés dans notre approche. Ainsi, la documentation et les recommandations de l'architecte permettent de limiter l'espace de recherche de manière efficace. L'algorithme génétique offre en plus la possibilité d'utiliser les résultats du recuit ou du regroupement hiérarchique pour diversifier sa population initiale et augmenter la qualité des résultats.

Les avantages et inconvénients que nous avons étudiés pour chacune de ces mises en pratique montrent que le choix d'une instance plutôt que l'autre est difficile. En effet, les caractéristiques des algorithmes sont très différentes et rendent la comparaison « sur le papier» difficile. De plus, les mises en pratique que nous avons proposées ne sont pas suffisantes pour démontrer la validité de notre approche.

En conséquence, il nous faut réaliser un cas d'étude pour comparer les différentes instances de ROMANTIC en action et pour pouvoir attester la validité de notre approche.

CHAPITRE 7

Validation et évaluation de l'approche ROMANTIC

No amount of experimentation can ever prove me right ; a single experiment can prove me wrong.

— Albert EINSTEIN .

Cas d'étude — Consistance — Complétude — Comparaison des algorithmes — Mesure de similarité

Face à la complexité de l'approche ROMANTIC dans son ensemble, il est difficile de démontrer formellement la validité des fondements théoriques de notre approche ou les propriétés des algorithmes proposés. L'objectif de ce chapitre est donc d'**étudier les aspects expérimentaux de l'approche** pour, d'une part mettre en évidence sa validité et, d'autre part, évaluer les performances de ses différentes réalisations. Nous présentons donc un cas d'étude en **deux étapes** : **validation des fondements théoriques**, présentés dans la partie II, puis **comparaison des instances de ROMANTIC**, décrites dans le chapitre précédent.

Pour réaliser ce cas d'étude, nous devons présenter trois éléments [134] :

- **un cadre d'étude** : ce cadre précis permet de fixer tous les paramètres de l'étude. Il décrit donc les algorithmes utilisés, leurs paramètres et les systèmes étudiés. Ce cadre doit permettre de reproduire le cas d'étude et de vérifier les résultats ;
- **une méthodologie** : elle pose les objectifs du cas d'étude et explique comment les atteindre et les mesures nécessaires. Dans notre cas, nous décrivons séparément les méthodes utilisées pour chacune des deux étapes ;
- **une réalisation** : c'est l'application de la méthodologie au cadre d'étude ainsi que les conclusions qui peuvent être tirer des résultats de cette application. Comme pour la méthodologie, nous présentons la réalisation en deux parties correspondant aux deux étapes de notre cas d'étude.

La suite du chapitre est organisée de la façon suivante. Nous décrivons d'abord le cadre de notre cas d'étude et en particulier les deux systèmes réels qui servent de sujet à notre étude : **un serveur WEB JAVA nommé Jigsaw et un outil de modélisation UML nommé ArgoUML**. Nous présentons ensuite chacune des deux étapes de notre étude en détaillant pour chacune la méthodologie utilisée puis les résultats obtenus.

7.1 Cadre du cas d'étude

Nous présentons dans cette section le contexte de notre cas d'étude. Nous détaillons donc les instances de ROMANTIC utilisées ainsi que les paramètres de ROMANTIC. Ensuite, nous décrivons les systèmes qui servent de tests à nos algorithmes.

7.1.1 Instances de ROMANTIC

Dans le cadre de ce cas d'étude, nous utilisons les trois instances de ROMANTIC présentées dans le chapitre 6 : le regroupement hiérarchique, le recuit simulé et l'algorithme génétique. Nous décrivons successivement les paramètres utilisés pour ce cas d'étude, en particulier les poids des différentes fonctions utilisées dans notre fonction objectif, puis les guides utilisés et leurs modes d'utilisation.

7.1.1.1 Paramétrage des algorithmes

Les instances de ROMANTIC utilisent la même fonction objectif pour diriger le processus. Afin de ne pas avantager une instance, nous utilisons les mêmes poids dans cette fonction pour l'ensemble des algorithmes. Ainsi, nous affectons un poids unitaire à la fonction d'évaluation de la qualité et un poids de deux à celle de la sémantique (*cf.* Equation 7.1).

$$Goal\left(s\right) = \frac{1}{3}\left(2 \cdot Sem\left(s\right) + 1 \cdot Qual\left(s\right)\right) \tag{7.1}$$

Ce choix permet de concentrer l'étude sur les résultats de notre étude de la sémantique architecturale. En effet, la mesure de la validité sémantique repose sur un modèle de mesure entiérement développé à partir des définitions les plus couramment admises, alors que la mesure de la qualité repose sur des travaux existants dans le domaine de l'évaluation de la qualité logicielle (SQM, *cf.* Chapitre 1). Il convient donc d'apporter une attention particulière à l'évaluation de la sémantique par rapport à celle de la qualité pour appuyer la validation de notre approche.

Les instances de ROMANTIC utilisent également des paramètres spécifiques. Ainsi, le recuit simulé nécessite un coéficient de décroissance géométrique qui influe sur le nombre d'itérations. Ce paramètre influe donc directement sur le temps de calcul. Afin de réaliser un nombre raisonnable de tests dans un temps limité nous avons donc choisi une valeur de 0.99 pour ce paramètre. Nous définissons également le nombre d'opérations par itération à dix. D'après la valeur atribuée aux opérations du recuit, il est donc possible de réaliser à chaque itération, une opération sur les contours de composants et dix sur les contours de connecteurs et d'interfaces.

L'algorithme génétique nécessite la définition du nombre d'itérations et de la taille de la population. Comme pour le recuit, ces paramètres influencent directement le temps de calcul. Nous avons donc choisi de réaliser les tests avec une population de 25 individus sur 1000 itérations. Nous avons également choisi de conserver les trois meilleurs individus de chaque génération dans la génération suivante. Pour cela, le premier opérateur de selection est paramétré pour selectionner 22 individus.

7.1.1.2 Utilisation des guides

Concernant les guides auxiliaires, nous utilisons uniquement les informations intentionnelles contenues dans les fichiers sources. En effet, l'utilisation des informations contenues dans les archives des outils de versionnement est impossible puisque ces données ne sont pas disponibles. Nous n'utilisons

pas non plus les diagrammes UML à cause de la disparité importante entre les documents disponibles pour les deux systèmes et du manque de confiance dans les documents proposés pour ArgoUML.

Nous n'utilisons pas non plus les recommandations de l'architecte. Ceci vise à ne pas influencer l'analyse des résultats puisque sans spécialiste des systèmes, il faudrait que nous définissions personnellement les contraintes de l'architecture.

Cependant, l'utilisation des guides varie en fonction des tests réalisés et nous précisons donc dans la suite lorsque les fichiers sources sont utilisés. En effet, les guides visant à limiter l'espace de recherche et à cibler des solutions, ils tendent à uniformiser les résultats et peuvent fausser certaines des mesures que nous décrivons dans la suite.

7.1.2 Systèmes étudiés

Nous réalisons ce cas d'étude sur deux systèmes objet libres de taille moyenne : Jigsaw et ArgoUML. Chacun de ces systèmes présente des caractéristiques différentes et apporte un éclairage particulier sur notre approche.

7.1.2.1 Jigsaw

Jigsaw est un serveur Web basé sur la plate-forme Java et faisant partie des projet W3C. Avec une architecture modulaire et une conformité complète à la norme HTTP/1.1, le serveur Jigsaw est la première plate-forme expérimentale de W3C*. Le serveur utilise une approche orientée objet pour la sauvegarde des fichiers et le traitement des requêtes, le rendant aussi bien efficace que facile à étendre. Il a été développé par un petit groupe au sein de W3C basé principalement à l'INRIA.

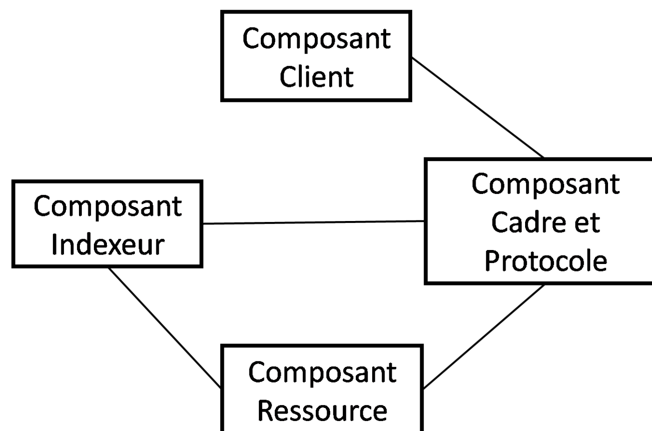


Figure 7.1 – Architecture de Jigsaw selon la documentation

Jigsaw contient environ 300 classes. Il fonctionne en associant chaque identifiant uniforme de ressource (*Uniform Resource Identifier*, URI) a un objet générant le contenu. Cet objet est associé à une

*World Wide Web Consortium, http ://www.w3.org/

ressource qui est encadrée par un cadre de gestion de protocole, créé manuellement en utilisant un client d'administration ou automatiquement à travers un indexeur. Ce fonctionnement est illustré dans la documentation par une architecture orienté objet précise qui est décrite par quatre composants (*cf.* Figure 7.1) :

- **le composant Ressource** : il contient les classes représentant les objets mis à disposition par le serveur. Ces ressources sont des éléments très basiques qui contiennent uniquement les données intrinsèques de la ressource telles que sa taille ou la date de la dernière modification. Ce composant contient également les classes gérant l'ensembles des ressources. Ces classes permettent de mettre l'ensemble des ressources à la disposition de différentes instances du serveur utilisant des protocoles différents ;
- **le composant Cadre et protocoles** : il contient les classes gérant les protocoles. En effet, les ressources ne sont accessibles à travers le serveur qu'une fois associées à un cadre de protocole qui gére les différents accès. Pour cela, le cadre contient toutes les informations nécessaires pour fournir une ressource à travers un protocole spécifique. il contient les classes représentant les filtres du serveur. Ces filtres encadrent l'accès aux ressources disponibles en réclamant par exemple, une authentification. A cause de leurs liens étroits avec le protocole utilisé, ils sont associés, non pas à une ressource particulière, mais au cadre qui la contient. Ils peuvent être appelés avant ou aprés l'accès à la ressource et peuvent modifier la requête reçue ou la réponse du serveur ;
- **le composant Indexeur** : il contient les classes gérant l'indexation automatique des ressources disponibles dans la hiérarchie du serveur. Ces indexeurs ciblent chacun un type particulier de ressources, déterminé par l'extension du fichier, et créent une hiérarchie des ressources de ce type disponibles. Ils s'occupent également d'attacher aux ressources le cadre et les filtres adéquats ;
- **le composant Client** : Il contient un ensemble de classes fournissant une API pour le coté client de Jigsaw. Ces classes incluent un gestionnaire des requêtes Http, des filtres gérant les cookies, l'utilisation de caches et les proxys.

Jigsaw a également été utilisé dans les cas d'étude de Focus [88]. L'architecture extraite dans ce cas d'étude est très proche du découpage de l'application proposée dans la documentation (*cf.* Figure 7.2) :

- **le composant Httpd** : il correspond au composant Client ;
- **le composant Ressource** : il correspand au composant Ressource ;
- **les composants CadreProtocole et Filtres** : ils sont inclus dans le composant Cadre et protocoles ;
- **le composant utilitaire** : il contient une partie du composant Cadre et protocoles ainsi que le composant Indexeur.

7.1.2.2 ArgoUML

ArgoUML est un outil d'aide à la conception orientée objet. Il est codé en Java et contient environ 1500 classes. Il permet de concevoir un système selon la norme UML 1.3 définie par l'OMG.

La documentation de ce logiciel le décrit à travers neufs aspects (*cf.* Figure 7.3) :

- **édition de modèle UML** : ArgoUML utilise l'outil GEF (Graph Editing Framework) de l'Université de Californie Irvine (UCI) pour éditer les diagrammes UML ;
- **gestion des contraintes OCL** : OCL (*Object Constraint Language*) est un langage logique de prédicat. ArgoUML est entièrement conforme à la syntaxe et aux types d'OCL et s'appuie sur le compilateur OCL développé par l'Université Technique de Dresden ;
- **conseils à l'utilisateur** : « *Design Critics* » est un processus de contrôle qui s'exécute en tâche de fond. Il analyse le travail de l'utilisateur, avertit des erreurs de syntaxes et propose des amélio-

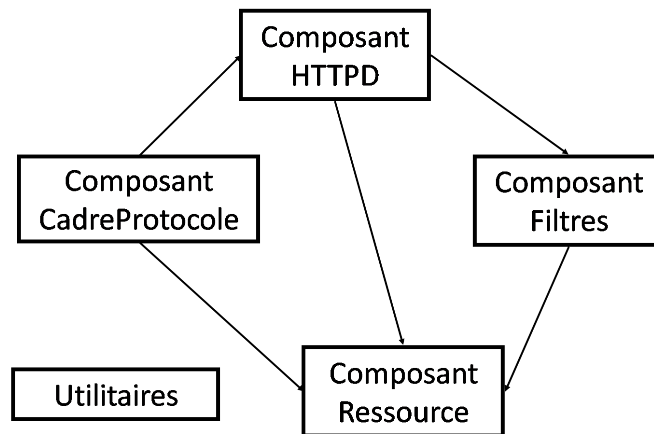


Figure 7.2 – Architecture de Jigsaw selon Focus

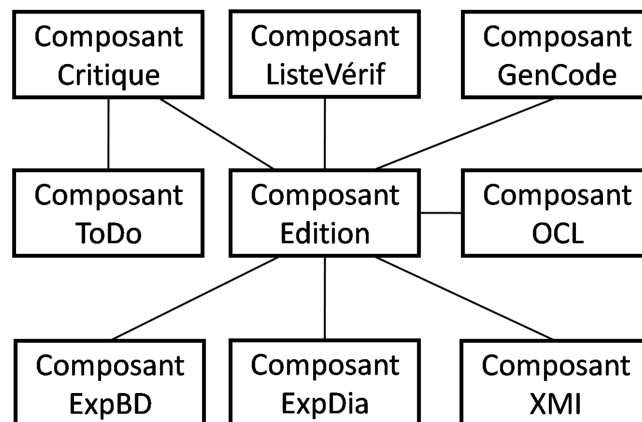


Figure 7.3 – Architecture de ArgoUML selon la documentation

rations. *Design Critics* n'interrompt jamais l'utilisateur, mais envoie toutes ses remarques dans la «To Do» liste ;

- **liste à faire** : grâce à la « *To Do* » liste, ArgoUML facilite le travail des concepteurs en leurs rappelant les tâches élémentaires qui restent à accomplir et les éventuelles erreurs trouvées par le *Design Critics*. Ceci permet alors aux concepteurs de se concentrer d'avantage sur des questions de conceptions et non sur des problèmes de syntaxe. Par ailleurs, le concepteur peut ajouter des notes personnelles dans la « *To Do* » liste pour rappel. Les entrées de la « *To Do* » liste peuvent être ensuite classées par ordre de priorité. Dans certains cas, une assistance est disponible pour la résolution des problèmes ;
- **liste à vérifier** : les « *Checklists* » sont largement répandues lors des réunions d'étude de conception, en partie, parce qu'elles rappellent aux concepteurs, les détails de conception à vérifier et permet d'éviter des erreurs communes de conception. ArgoUML fournit aussi une « *Checklist* » qui répertorie les questions essentielles à se poser. Comme par exemple, le nom de la classe est-il approprié ?
- **génération de code Java** : ArgoUML permet la génération de classes Java à partir des diagrammes de classes ;
- **support XMI** : XMI est un format d'échange XML entre les outils UML. ArgoUML utilise ce standard de sauvegarde pour faciliter l'échange de données avec d'autres applications. Ceci permet de convertir des données Rational Rose vers ArgoUML, par exemple ;
- **exportation vers une base de données** : ArgoUML permet l'exportation des données vers n'importe quelle base de données compatible JDBC ;
- **exportation de diagramme** : les diagrammes peuvent également être exportés vers un format GIF, PostScript, EPS, PGML ou SVG.

Cependant, il est difficile de considérer que ces aspects constituent les composants de l'architecture du système. En effet, l'architecture d'ArgoUML a subi une forte érosion durant la phase de conception puis durant les différentes évolutions. Ainsi, le code du système contient maintenant des relations entre les différents composants non prévues telles qu'entre le composant *ToDo* et *CheckList* ou encore entre chaque composant (à l'exception de *ToDo*) et le composant OCL (*cf.* Figure 7.4).

L'étude d'ArgoUML offre un autre type d'épreuves pour notre approche par rapport à celles posées par Jigsaw. En effet, dans ce cas, il s'agit de considérer un système dont l'architecture réelle diverge fortement de celle documentée. Cette différence qui provient principalement de relations non-documentées et non prévues entre les classes, permet de tester la resistance de notre approche au bruit dans le code.

7.2 Validation des fondements théoriques

La validation des fondements théoriques de ROMANTIC est la première étape de notre cas d'étude. Nous décrivons dans cette section les méthodes et les principes utilisés pour mettre en évidence cette validité. Nous appliquons ensuite ces méthodes à notre cadre d'étude, puis nous concluons sur la validité de ROMANTIC.

7.2.1 Méthode de validation des fondements théoriques

Afin d'attester de la validité des fondements théoriques de notre approche, nous allons montrer deux propriétés essentielles de notre approche. La première est la consistance. Dans la terminologie de la

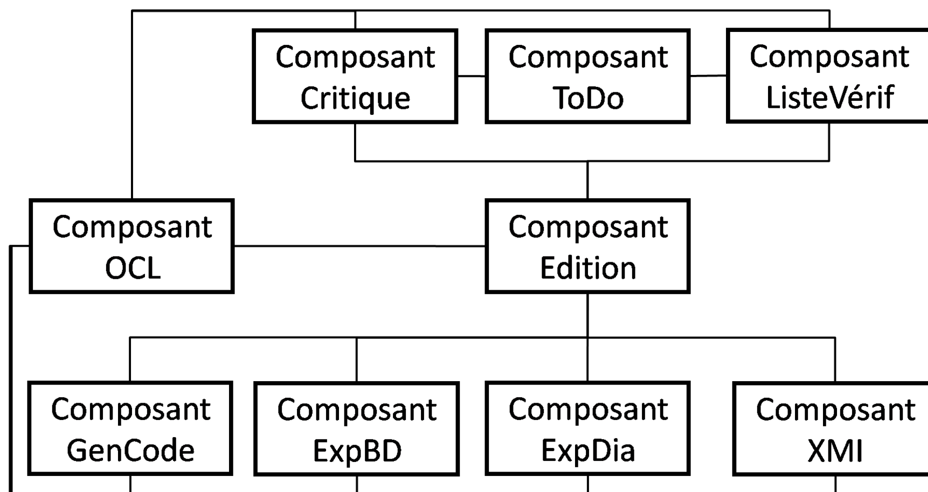


Figure 7.4 – Architecture probable de ArgoUML

logique classique, elle décrit à une logique qui ne contient pas de contradiction, c'est-à-dire à la fois une formule et sa négation.

La seconde propriété est l'adéquation. Dans la terminologie de la logique classique, elle décrit une logique dans laquelle tous les théorèmes, c'est-à-dire les formules générées par la logique, sont des formules valides, c'est-à-dire des formules toujours vrai.

Pour mettre en évidence ces deux propriétés, nous proposons une méthode en deux étapes. Pour cela, nous revenons pour chacune de ces notions sur sa définition dans le cas particulier de notre approche, puis nous décrivons le déroulement de l'étude.

7.2.1.1 Consistance de ROMANTIC

Définition de la consistance. Notre approche doit être consistante : elle ne doit pas contenir de contradiction. Pour cela, **les solutions proposées doivent être similaires**. En effet, elles doivent toutes représenter l'architecture du système. Or, si la représentation de l'architecture d'un système n'est pas unique, l'architecture elle est unique. Elle est concrétisée par le code source et les différentes représentations ne sont que différents points de vues d'un même système et sont donc forcément similaires.

Mise en évidence de la consistance. Pour attester la consistance de notre approche, nous devons donc montrer que toutes les solutions proposées par notre approche, pour un même système, sont similaires. Cette similarité doit porter sur toutes les solutions quelle que soit l'algorithme ou le paramétrage de notre processus.

En conséquence, nous proposons de mesurer la similarité d'un ensemble suffisamment grand d'architectures extraites en utilisant ROMANTIC. Pour cela, nous calculons, pour chacun des algorithmes proposés, un ensemble d'architectures. Cette extraction est faite sans utiliser les guides et nous utilisons un nombre égal de solutions provenant de chaque algorithme. Ces deux choix permettent de limiter l'influence de l'algorithme et de reposer le résultat sur l'impact du fondement théorique de ROMANTIC.

Ensuite, nous mesurons la similarité de l'ensemble des solutions extraites.

Choix de la mesure de similarité. Différentes mesures de similarité ont été proposées pour évaluer les résultats de travaux d'extraction de représentations abstraites existants et en particulier les travaux de modularisation [60, 125, 131, 92, 94]. Toutes ces mesures comparent les partitions résultantes de l'extraction en évaluant le degré de similarité du placement des entités dans les groupes formant la partition. Cependant, elles ne sont pas pour autant équivalentes. Par exemple, MoJo [125, 131] présente une grande déviation lorsque le graphe représentant le système possède de nombreux modules isomorphiques. Celle proposée par HARMAN [60] repose sur une mesure antérieure [32], mais sa complexité impose de limiter les comparaisons à un petit nombre de solutions.

Les solutions résultantes de notre approche sont différentes des approches précédentes d'extraction puisque elles ne sont pas uniquement une partition d'entités. Les contours de composants forment bien une partition des classes mais les contours de connecteurs sont d'une granularité différente puisqu'ils sont composés de méthodes. Nous proposons donc d'utiliser une même mesure de similarité à deux reprises : (i) pour comparer la partition des classes formée par les contours de composants et (ii) pour comparer la partition des méthodes formée par les contours de connecteurs et de composants en considérant qu'une méthode qui se trouve dans un contour de connecteur n'est pas dans le contour de composant de sa classe.

Nous avons choisi d'utiliser la mesure proposée par MITCHELL [94]. En effet, sa complexité est faible et elle permet de varier la finesse de l'évaluation en proposant différents niveaux de similarité. Le principe de cette mesure est de mesurer, pour chaque relation entre entités, le nombre de solutions pour lesquelles la relation est incluse dans un groupement. Cette valeur est ensuite diviseée par le nombre de solutions comparées afin d'obtenir pour chaque relation entre entités le pourcentage d'inclusion dans un groupement. Les différents résultats sont alors agrégés selon les niveaux : $\{[0,0],(0,10],(10,75),[75,100]\}$. Ces niveaux correspondent respectivement à une similarité nulle, faible, moyenne et forte. Enfin, le nombre de relations dans chaque niveau est normalisé en divisant par le nombre de relations dans le système.

Avec cette mesure de similarité, nous pouvons évaluer la similarité d'un ensemble de partitions. Une forte proportion de similarité forte et nulle est une bonne chose, puisque ces valeurs montrent que les relations sont respectivement toujours ou jamais internes à un groupe. La similarité faible représente les relations qui apparaissent moins de 10 % des fois dans le même groupe, ce qui est souhaitable. En effet, cela montre que les solutions proposées séparent systématiquement les entités associées à cette relation. Avoir une forte proportion de similarité moyenne n'est pas souhaitable puisque ce résultat indique que de nombreuses relations apparaissent à la fois entre et dans des groupes dans les différentes solutions. Elle indique donc que les partitions correspondantes sont contradictoires.

Nous utilisons cette mesure avec deux niveaux de granularité. Le premier niveau mesure la similarité des partitions formées par les contours de composants. Les entités prises en compte sont les classes et les relations de dépendances entre ces classes. Ce niveau permet d'évaluer la similarité des composants extraits. Le second niveau mesure la granularité des partitions de méthodes formées par les contours de connecteurs et les contours de composants. Pour cela, nous considérons que les méthodes incluses dans un contour de connecteurs n'appartiennent pas au contour de composants de leur classe. Ce deuxième niveau permet d'évaluer à la fois la similarité entre les composants extraits et les connecteurs extraits. Par comparaison avec le premier niveau, nous obtenons ainsi une évaluation de la similarité des connecteurs extraits.

7.2.1.2 Adéquation de ROMANTIC

Définition de l'adéquation. La consistance des architectures extraites par notre approche n'est pas suffisante. En effet, si les solutions sont globalement consistantes, il reste à montrer que notre approche est adéquate, c'est-à-dire que **l'architecture montrée par cet ensemble de représentations correspond à l'architecture réelle du système**.

Mise en évidence de l'adéquation. Pour montrer l'adéquation de notre approche, nous devons établir que les représentations extraites reflètent l'architecture du système. Nous proposons de soumettre l'une des solutions extraites à une étude qualitative détaillée afin de juger de sa pertinence par rapport à l'architecture du système. Cette solution est obtenue sans utiliser les guides et en selectionnant la meilleure rencontrée.

Pour réaliser cette étude, nous utilisons la documentation et notre compréhension du système pour confronter la solution retenue au système et déterminer la qualité de cette solution du point de vue d'un architecte expert du système.

7.2.2 Analyse des résultats

Afin d'attester de la validité de notre approche, nous étudions la consistance et la complétude de ROMANTIC selon la méthode précédente et dans le cadre défini dans la section précédente.

7.2.2.1 Étude de la consistance de notre approche

Similarité selon la granularité « classe ». Nous avons testé la similarité sur un échantillon de 99 résultats provenant, à par égale, de chaque instance de ROMANTIC. Le tableau 7.1 montre les résultats des mesures de similarité de la partition des classes constituée par les contours de composants.

Application	Degré de similarité			
	Zéro (%) $S = 0\%$	Faible (%) $0\% < S \leq 10\%$	Moyenne (%) $10\% < S < 75\%$	Forte (%) $S \geq 75\%$
Jigsaw	22.9	12.6	6.4	58.1
ArgoUML	49.3	18.4	12.1	20.2

Table 7.1 – Mesure du degré de similarité selon la granularité « classe »

Similarité des solutions pour Jigsaw. 81 % des relations entre les classes se trouvent dans les catégories Zéro ou Forte. Ceci démontre que la grande majorité des classes sont respectivement toujours séparées ou toujours ensemble. Ces valeurs donnent aussi une indication sur le nombre de composants. En effet, dans le cas ou de nombreux composants sont extraits le nombre de relations entre classes qui apparaient à l'extérieur des composants augmente et par conséquent la part de la catégorie Zéro augmente aussi. Ainsi, la valeur importante de la catégorie Forte montre qu'une grande part des classes sont toujours ensembles et que le nombre de composants est plutôt limité.

Seulement 6.4 % des relations sont dans la catégorie Moyenne. Ceci montre que les voisinages des classes sont stables d'une solution à une autre. Les variations portent en générale sur une ou deux classes qui sont « à cheval » sur deux composants. Ce chevauchement est a priori dû au contenu de la classe qui est fortement dépendant de deux composants distincts. Par conséquent, une partie importante de

ces classes constituent les méthodes incluses dans les connecteurs. Nous affinons donc l'étude de ces catégories dans la suite en utilisant la similarité selon la granularité « méthode ».

Similarité des solutions pour ArgoUML. Comme pour Jigsaw, la grande majorité, 69.6%, des relations entre les classes se trouvent dans les catégories Zéro et Forte. Ceci démontre encore la stabilité de nos solutions. Cependant, la différence majeure avec les résultats de Jigsaw est l'inversion des valeurs entre la catégorie Zéro et la catégorie Forte. En effet, avec 49.3% des relations la catégorie Zéro est la plus importante. Ceci montre que les composants sont plus nombreux dans les solutions extraites de ArgoUML que de Jigsaw. Cette granularité plus faible des composants s'explique par le nombre plus important de relations dans ArgoUML qui rend difficile l'optimisation. La priorité est donc donnée à des composants réduits possédant un couplage et une cohésion suffisamment importants pour compenser le couplage avec l'extérieur.

Le pourcentage de relations dans la catégorie Moyenne est deux fois plus important que dans le cas de Jigsaw. En effet, au phénomène observé dans le cas de Jigsaw, s'ajoute le grand nombre de relations entre les classes, c'est-à-dire à la mauvaise conception de ArgoUML. Cependant la proportion de relations dans les « bonnes » catégories reste du même ordre de grandeur que pour Jigsaw.

Similarité selon la granularité « classe ». Le tableau 7.2 montre les résultats des mesures de similarité de la partition des méthodes constituée par les contours de composants modifiés et les contours de connecteurs.

Application	Degré de similarité			
	Zéro (%) $S = 0\%$	Faible (%) $0\% < S \leq 10\%$	Moyenne (%) $10\% < S < 75\%$	Forte (%) $S \geq 75\%$
Jigsaw	24.7	10.6	23.2	41.5
ArgoUML	48.2	11.7	19.2	21.0

Table 7.2 – Mesure du degré de similarité selon la granularité « méthode »

Similarité des solutions pour Jigsaw. Comme dans le cas de la granularité « classe », les catégories Zéro et Forte englobent la majorité des relations (66.2 %). Ces résultats confirment donc la similarité observée au niveau précedent. Par contre, la catégorie Moyenne est beaucoup plus représentée qu'au niveau des classes. Ceci est dû au chevauchement des classes que nous avons évoquée précédemment. En effet, les relations qui appartiennent à la catégorie Moyenne de ce niveau correspondent à celles du niveau des classes. Les classes chevauchant plusieurs composants sont donc celles qui contiennent des méthodes appartenant à un connecteur. Elles appartiennent ainsi, selon les solutions, à un composant ou un autre et une partie plus ou moins importante de leurs méthodes sont contenues dans le connecteur reliant les composants.

La valeur de la catégorie Moyenne montre aussi qu'à travers les solutions, les composants sont plus stables que les connecteurs. Une des raisons que nous avons identifiée pour cette différence est le fait que l'aspect comportemental des connecteurs n'est pas pris en considération lors de leurs identifications. En effet, nous utilions pour les connecteurs et les composants une identification basée sur des critères structurels. Cette méthode fonctionne correctement pour les composants puisque il est possible de détecter structurellement les limites d'un aspect métier. Par contre les connecteurs sont plus difficiles à identifier puisqu'ils sont souvent aussi couplés avec leurs composants qu'avec leurs éléments internes.

Similarité des solutions pour ArgoUML. Les résultats obtenus pour ArgoUML sont très différents de ceux de Jigsaw. En effet, la similarité au niveau des classes est très proche de celle au niveau des méthodes. Nous retrouvons une grande majorité de relations appartenant aux catégories Forte et Zéro (69.2 %) avec un avantage pour la catégorie Zéro. La proportion de la catégorie Moyenne est également du même niveau.

Ceci s'explique encore une fois par la mauvaise conception du système. La petite taille des composants extraits de ArgoUML fait qu'ils ont la même sensibilité que les connecteurs aux changements. Nous avons donc un traitement plus équilibré des connecteurs et des composants dans le cas d'un système tel qu'ArgoUML.

7.2.2.2 Étude de l'adéquation de notre approche

Afin d'attester l'adéquation de notre approche, nous étudions les architectures extraites par ROMANTIC de Jigsaw et de ArgoUML. Comme nous l'avons vu dans la méthodologie, l'objectif est de montrer que les résultats correspondent à ce qu'un architecte, connaissant le système, attend à voir émerger comme architecture.

Architecture extraite de Jigsaw. L'architecture extraite de Jigsaw contient cinq composants (*cf.* Figure 7.5). Le composant Client contient les classes réalisant le coté client du serveur Jigsaw. Il permet de gérer les requêtes Http et les « *cookies* » par exemple. Il est connecté au composant Cadre qui contient les classes chargées de la gestion des ressources. Cependant, d'autres classes de ce type se trouvent dans le composant Filtre. Il s'agit en fait de classes qui gèrent l'accès au ressources en proposant en plus une fonctionnalité de filtrage qui peut être basée sur l'adresse IP ou une horaire spécifique. Ce composant Filtre est connecté au même connecteur que Cadre. Enfin les composants Ressource et Indexeur contiennent respectivement les classes représentant les ressources et celles gérant l'instanciation automatiques des objets Ressources en fonction de leur noms ou de leurs extensions, par exemple.

L'architecture extraite est proche de celle documentée. Nous retrouvons les composants Client, Indexeur et Ressources. Le composant Cadre et Protocoles est divisé en deux : le composant Cadre et le composant Filtre. Cependant, cette proximité n'est pas parfaitement exacte en termes de classes. Ainsi, certaines classes documentées dans le composant Ressources sont, dans notre architecture, dans le composant Indexeurs. Ceux sont ces classes qui constituent les différences entre les solutions proposées par notre approche. Comme nous l'avons montré lors de l'étude de la consistance, ces classes ne sont pas nombreuses et la majorité des classes sont présentes dans le même composant d'une solution à une autre.

Architecture extraite de ArgoUML. L'architecture extraite de ArgoUML contient 17 composants (*cf.* Figure 7.6).

Parmi ces composants, six correspondent en grande partie à une fonctionnalité du système :

- **GenCode** : il correspond à la fonctionnalité de génération de code en java ;
- **ExpBD** : il permet d'exporter le modèle vers une base de donnée ;
- **ExpXMI** : il permet d'exporter le modèle en XMI ;
- **GestionOCL** : il gère les contraintes OCL. Il contient également une partie des classes permettant l'affichage des contraintes dans les diagrammes. Il contient ainsi quelques classes qui sont documentées comme appartenant au composant Edition ;
- **GestionToDo** : il gère la liste des choses à faire. Comme Gestion OCL il contient certaines classes prenant en charge l'affichage de la liste. Mais ces classes appartiennent au même composant dans la documentation ;

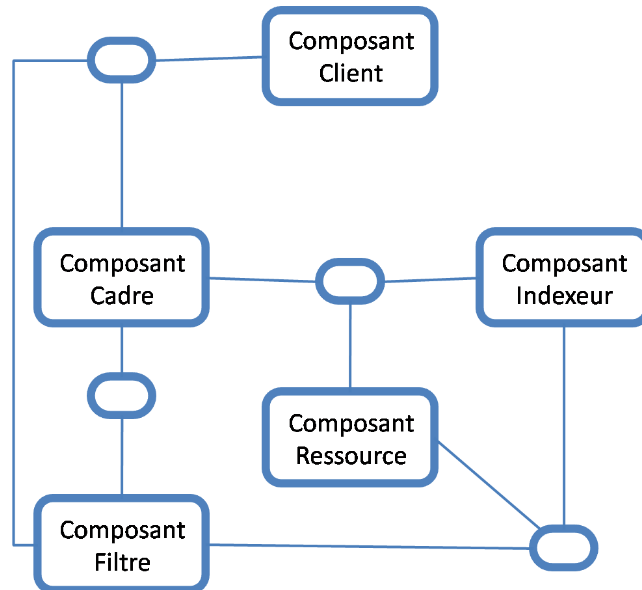


Figure 7.5 – Architecture de Jigsaw selon ROMANTIC

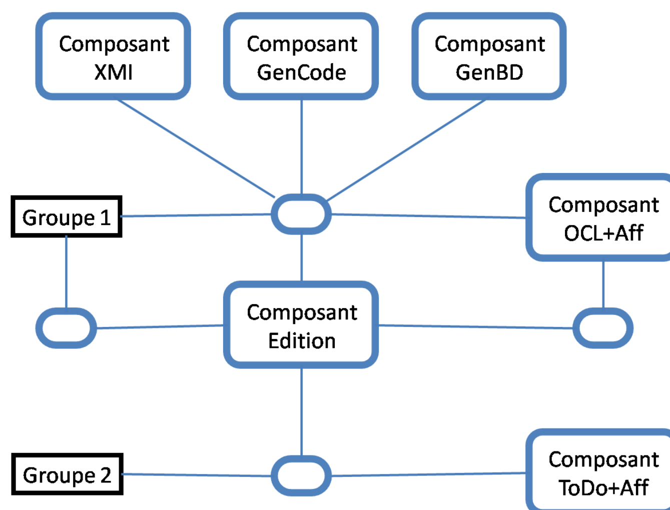


Figure 7.6 – Vue partielle de l'architecture de ArgoUML selon ROMANTIC

- **Edition** : il contient les classes gérant l'édition des diagrammes UML. Comme dans la documentation, c'est le composant central de l'application. Cependant, il contient moins de classes que dans la documentation. En effet, certaines classes fortement liées aux contraintes OCL, à la génération des images ou aux critiques de conception appartiennent en fait aux composants correspondant de l'architecture extraite.

Les autres composants se répartissent en deux groupes représentés sur la figure 7.6 par les groupes 1 et 2. Le premier est formé de trois composants (*cf.* Figure 7.7). Ces composants permettent de générer une image à partir du modèle UML. Ils sont très couplés et reliés par un connecteur en commun. Il n'est pas possible d'identifier chacun à la génération d'un format d'image spécifique. Cependant, la réunion des trois composants définit précisement les contours de la fonctionnalité d'exportation vers une image.

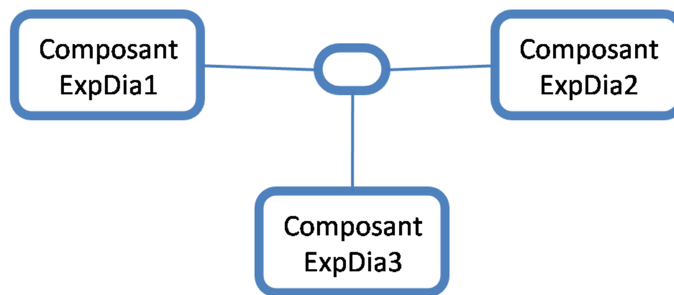


Figure 7.7 – Vue des composants du groupe 1 dans ArgoUML

Le deuxième groupe de composants contient huit composants (*cf.* Figure 7.8). Chacun d'entre eux contient des méthodes participant à la gestion des critiques de modélisation, de la liste des vérifications et de l'édition. Les critiques et les vérifications associées dans un composant semblent avoir des points communs, mais il est difficile de mettre en évidence une fonctionnalité particulière à chacun de ces composants.

7.2.3 Conclusion sur la validité de ROMANTIC

Consistance de ROMANTIC. Les mesures de similarité que nous avons présentées ont mis en évidence une grande similitude entre les solutions extraites par ROMANTIC. Nous avons vu que les connecteurs peuvent présenter une similitude plus faible que celle existant entre les composants, mais cette différence tend à s'aténuer lorsque le système est mal conçu ou érodé, c'est-à-dire lorsqu'il nécessite une extraction d'architecture.

Les différences apparues entre les solutions extraites s'expliquent, en premier lieu, par le caractère méta-heuristique des instances de notre approche. En effet, nos algorithmes sont stochastiques et sont donc intrinsèquement soumis à des variations. La mauvaise conception d'ArgoUML et l'aspect exclusivement structurel de notre identification sont également responsable de certains décalages, mais ils ne nuisent pas fondamentalement à la consistance de notre approche.

Ainsi, les résultats que nous avons présentés attestent de la consistance de notre approche quel que soit l'instance de ROMANTIC utilisée. **Le fondement théorique de notre approche est donc consistant**.

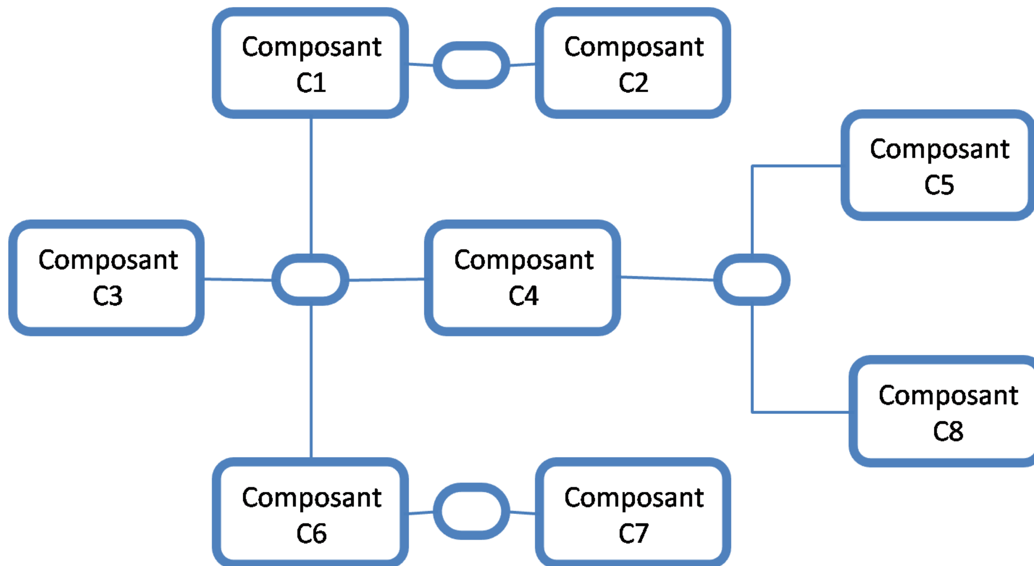


Figure 7.8 – Vue des composants du groupe 2 dans ArgoUML

Adéquation de ROMANTIC. **L'étude des architectures extraites par ROMANTIC à partir des systèmes Jigsaw et ArgoUML a montré l'adéquation de notre approche**. En effet, l'étude qualitative des deux solutions a souligné les liens étroits entre l'architecture extraite et l'architecture du système prévue par un expert.

Dans le cas de Jigsaw, la solution obtenue est très similaire à l'architecture documentée et à celle prévue par d'autres approches. La principale différence est la prise en compte des connecteurs et des interfaces dans notre architecture. Cette prise en compte permet de faciliter l'utilisation du résultat extrait en fournissant une architecture répondant complètement aux définitions actuelles de ce concept.

Dans le cas d'ArgoUML, la solution obtenue présente quelques défauts par rapport à l'architecture attendue. Cependant, ces défauts proviennent de relations présentent dans le code source, mais non-documentées et qui parasitent le processus d'extraction. Au final, l'architecture extraite est plus proche de la réalité du système que celle attendue et permet justement de mettre à jour les connaissances sur le système.

Validité de ROMANTIC. Nous avons établi la consistance et l'adéquation de ROMANTIC quelque soit les algorithmes utilisés. Nous concluons donc que **le fondement théorique de notre approche permet de définir une approche valide**.

7.3 Comparaison des instances de ROMANTIC

Nous avons proposé trois algorithmes qui réalisent ROMANTIC. Le regroupement hiérarchique étant non-stochastique, son comportement se déduit totalement de son algorithme, mais les deux algorithmes méta-heuristiques sont plus difficiles à appréhender. En effet, nous avons vu (*cf.* Chapitre 6), d'un point de vue théorique, que chacun de ces algorithmes présente des avantages et des inconvénients. De plus, le

théorème «no free lunch» [132] a montré qu'aucune méta-heuristique n'est plus performante sur toutes les classes de problèmes. Il est ainsi difficile de déterminer quel algorithme doit être utilisé pour résoudre notre problème de la manière la plus efficace.

Nous proposons donc de comparer les deux instances méta-heuristiques de ROMANTIC en utilisant le regroupement hiérarchique comme référence. Nous décrivons, dans cette section, les méthodes et les principes utilisés pour comparer les différents algorithmes. Nous appliquons ensuite ces méthodes à notre cadre d'étude, puis nous concluons sur les critères de choix de l'instance de notre approche à utiliser.

7.3.1 Méthode de comparaison des algorithmes

Afin de comparer les instances de notre approche, nous étudions l'efficacité des algorithmes. Cette caractéristique permet de déterminer l'instance la plus performante en fonction des paramètres que sont, par exemple, les temps de calcul disponibles et la qualité souhaitée.

Nous détaillons l'efficacité des algorithmes en précisant ces sous-caractéristiques. Nous décrivons ensuite notre méthode de comparaison.

7.3.1.1 Efficacité d'un algorithme

La notion d'efficacité d'une méta-heuristique se rapporte généralement à deux objectifs contradictoires : la vitesse et la précision. Bien souvent, un algorithme méta-heuristique rapide est peu précis, et inversement.

Vitesse des algorithmes. La vitesse d'une méta-heuristique influe directement sur l'efficacité d'un algorithme. Elle représente l'aspect temporel de cette caractéristique. Elle est souvent mesurée en nombre d'évaluations de la fonction objectif. Cette fonction est, en effet, la plupart du temps la partie la plus complexe du processus d'exploration.

Précision des algorithmes. La précision d'une méta-heuristique est l'aspect qualité de l'efficacité. Il décrit la qualité globale des solutions obtenues à travers une méta-heuristique. Pour cela, la précision estime la distance entre l'optimum trouvé par la méta-heuristique et l'optimum réel. Le résultat de la fonction objectif est souvent utilisé pour mesurer la précision. Cependant, l'utilisation de cette mesure impose de comparer des algorithmes utilisant la même fonction afin de ne pas léser une approche dont les objectifs seraient différents.

7.3.1.2 Comparaison des algorithmes

Afin de comparer les deux méta-heuristiques proposés pour réaliser notre approche, nous calculons un ensemble de résultats à partir de chaque algorithme et en utilisant les guides. Afin de réduire le temps de calcul nécessaire, nous appliquons ces tests uniquement sur Jigsaw qui est le système possèdant le moins de classes. Nous comparons ensuite les résultats selon deux axes : la vitesse et le rapport vitesse/précision.

Vitesse de des algorithmes. Nous proposons d'utiliser la mesure du nombre d'évaluations de la fonction objectif pour comparer les vitesses des deux méta-heuristiques que nous avons proposées. Cependant, cette mesure n'a pas le même sens pour notre algorithme basé sur le regroupement hiérarchique. En effet, cet algorithme est non-stochastique et le nombre d'évaluations de la fonction objectif est donc constant. Il dépend de la taille du système. Cette valeur permet malgré tout de définir un point de comparaison pour les valeurs mesurées pour les deux méta-heuristiques ;

Rapport vitesse/précision des algorithmes. La qualité des solutions extraites dépend, pour l'ensemble des méta-heuristiques présentées, du temps de calcul. Ainsi, plus le recuit ou l'algorithme génétique font d'itération, plus la qualité augmente. La précision de nos algorithmes dépend donc plus du temps de calcul que de l'algorithme. Nous proposons d'étudier à la place le rapport entre la précision et la vitesse. Pour cela, nous étudions le nombre la qualité des solutions extraites en fonction du nombre d'évaluations de la fonction objectif.

Nous réalisons ces tests avec différents jeux de paramètres pour chaque algorithme afin de faire varier la vitesse. Cependant, nous n'utilisons pas le regroupement hierarchique puisque, étant non-stochastique, ce rapport est constant.

7.3.2 Comparaison des algorithmes

Conformement à la méthode proposée, nous étudions successivement la vitesse, la précision et le rapport entre ces deux propriétés pour chaque instance de ROMANTIC.

7.3.2.1 Étude de la vitesse des algorithmes

Nous comparons la vitesse des algorithmes en utilisant le nombre d'évaluations de la fonction objectif. Les figures 7.9 et 7.10 présentent les résultats des cinquante tests effectués respectivement sur le recuit simulé et l'algorithme génétique. L'axe horizontal indique le numéro de la solution et l'axe vertical le nombre d'évaluations. Les solutions sont classées par qualité croissante.

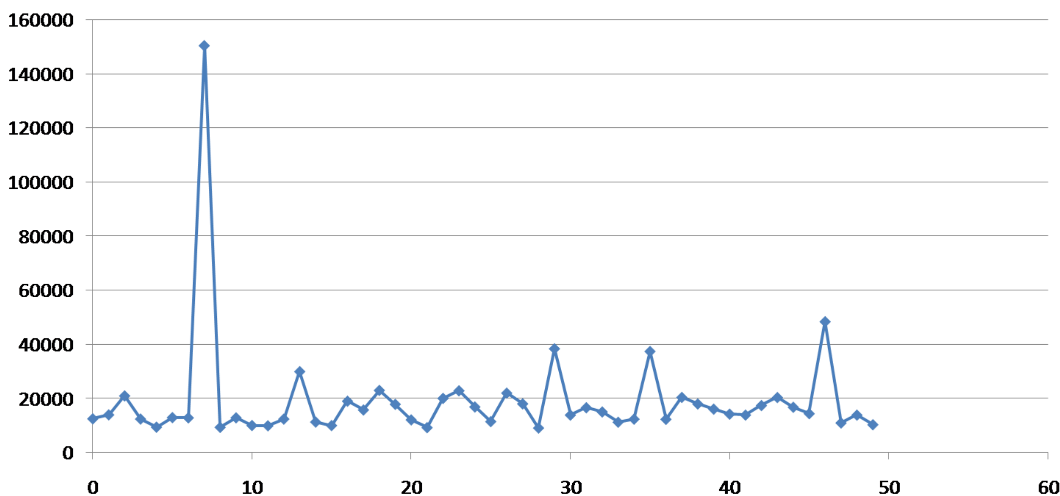


Figure 7.9 – Etude de la vitesse du recuit simulé sur Jigsaw

Vitesse du regroupement hiérarchique. Pour le regroupement hiérarchique, le nombre d'évaluations est constant. En effet, à chaque itération du regroupement, l'algorithme calcule la fonction objectif entre le groupe nouvellement créé et l'ensemble des groupes existants. Les étapes suivantes de l'algorithme utilisent les évaluations déjà réalisées et ne nécessitent donc aucune évaluation supplémentaire. Le nombre

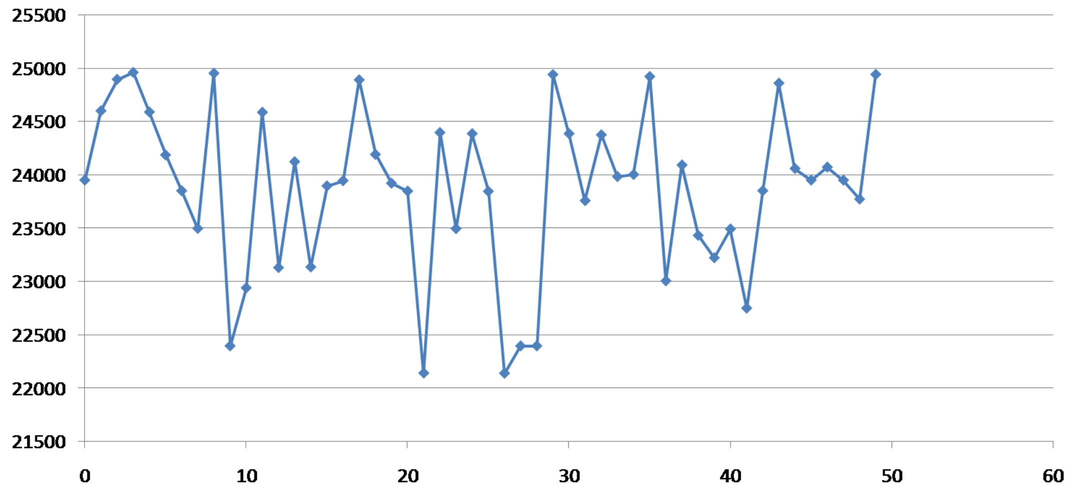


Figure 7.10 – Etude de la vitesse de l'algorithme génétique sur Jigsaw

d'évaluations est donc défini par l'équation suivante, où n est le nombre de classes dans le système :

$$Vitesse\,(n) = 4 + (n-3) \cdot (n+1) \tag{7.2}$$

Jigsaw est constitué de 298 classes et le nombre d'évaluations obtenu est bien conforme à cette formule (88209 évaluations). De la même façon, ArgoUML qui contient 1516 classes nécessite 2295225 évaluations. Ces valeurs donnent une base de comparaison pour les méta-heuristiques.

Vitesse du recuit simulé. La vitesse du recuit est très variable. Elle dépend du nombre d'itérations et du nombre d'opérations autorisées par itération mais surtout du nombre de tentatives faites à chaque itération. Les valeurs sont comprises entre 9062 et 150413 évaluations. Cette grande disparité vient des évaluations de solutions moins bonnes et rejettées par le recuit.

Ces évaluations inutiles peuvent constituer une grande perte de vitesse. En effet, Le paramètre M du recuit vaut dix et le nombre d'itérations, qui dépend de la température initiale et du coefficient de décroissance, est de l'ordre de 3000. Le nombre maximum d'évaluations utiles est donc de l'ordre de 30000. Dans le pire cas, notre algorithme a donc réalisé environ 120000 évaluations inutiles.

Cependant la valeur moyenne ($\approx$ 19000) et la meilleure vitesse ($\approx$ 9000), sont largement plus rapides que le regroupement hiérarchique ($\approx$ 88000).

Vitesse de l'algorithme génétique La vitesse de l'algorithme génétique est nettement plus constante que celle du recuit. En effet, le nombre d'évaluations est borné par le nombre d'itérations et la taille de la population. Dans notre cas, il est donc inférieur ou égal à 25000. De plus, les varitations dépendent du nombre d'individus conservés d'une génération à une autre. Or cette selection favorise largement les nouveaux individus et donc ceux qui nécessitent une évaluation de la fonction objectif. Par conséquent, les individus conservés sont rares (environ deux ou trois par génération dans notre cas) et le nombre d'évaluations reste proche de 25000.

La vitesse de l'algorithme génétique est supérieure à celle du regroupement hiérarchique. Par contre, la comparaison avec le recuit est plus difficile. Dans le pire cas, le recuit est beaucoup plus lent, mais dans

le meilleur cas il est presque trois fois plus rapide. Au finale, la vitesse moyenne du recuit est meilleure que celle de l'agorithme génétique.

7.3.2.2 Étude du rapport entre vitesse et précision des algorithmes

Nous comparons le rapport entre la vitesse et la précision des algorithmes en utilisant les mêmes mesures que précédemment. Les figures 7.11 et 7.12 présentent les résultats des trente tests effectués respectivement sur le recuit simulé et l'algorithme génétique. Cependant, nous utilisons deux configurations de chacune des méta-heuristiques. Ainsi, en plus de la configuration décrite dans la section 7.1, nous utilions une instance du recuit avec un paramètre de décroissance de 0.999, et une instance de l'agorithme génétique avec un nombre de générations de 10000. Ceci nous permet de diversifier la vitesse relevée pour l'algorithme génétique. L'axe horizontale indique le nombre d'évaluations de la fonction objectif et l'axe verticale le résultat de cette fonction.

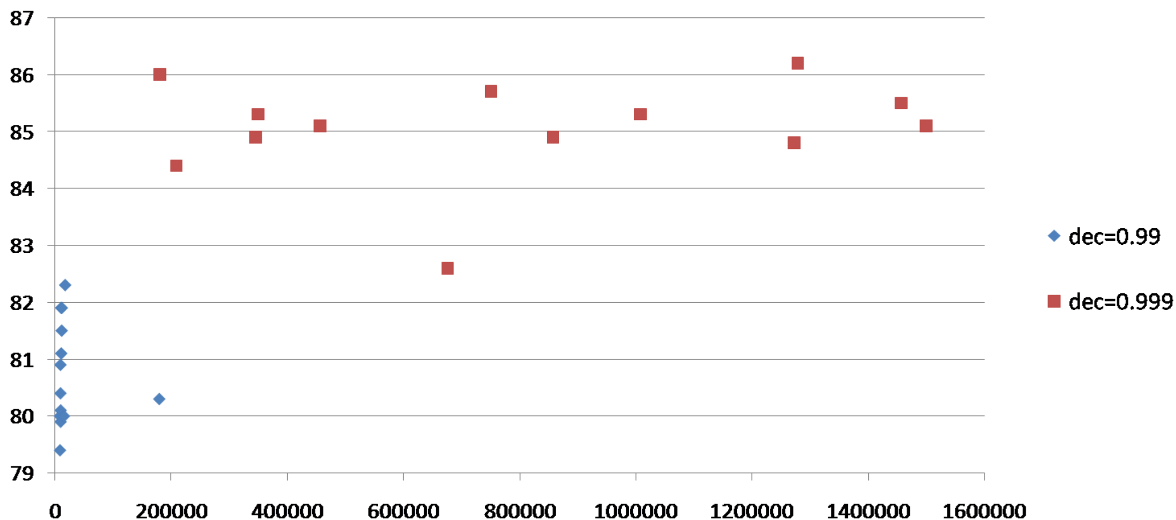


Figure 7.11 – Etude du rapport vitesse/précision du recuit simulé sur Jigsaw

Rapport vitesse/précision dans le recuit simulé. Le diagramme montre que les solutions extraites se situent principalement sur deux niveaux de qualité : entre 79 et 82 puis entre 84 et 86. Chaque groupe correspond à un des deux paramétrages utilisés. Cependant, la différence de qualité entre les deux groupes est faible. Le gain en terme de qualité pour cette augmentation des paramètres est donc faible.

Concernant le nombre d'évaluations, il semble également qu'il y ait deux groupes distincts. Cependant cet aspect est trompeur. Il est du à la plage importante représentée sur l'axe horizontale. La différence entre les valeurs extrèmes de chaque groupe est beaucoup plus important que pour la qualité, environ 170000 évaluations de différence pour le premier et 1300000 pour le second. La différence entre les deux paramètrages est également plus importante puisqu'elle est de l'ordre d'un facteur trente, passant en moyenne d'environ 25000 à 800000.

Ainsi, il apparait que le gain en terme de qualité est largement inférieur à l'augmentation du nombre d'évaluation. La précision du recuit augmente donc beaucoup moins vite que sa vitesse ne diminue. Il est donc risqué de décider d'augmenter le temps de calcul pour améliorer les résultats.

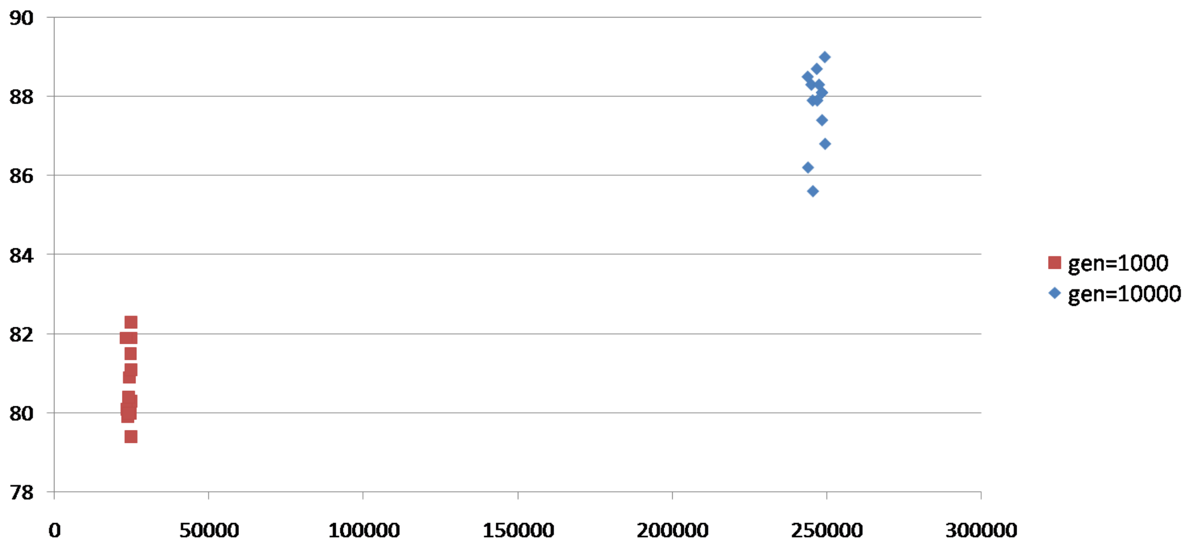


Figure 7.12 – Etude du rapport vitesse/précision de l'algorithme génétique sur Jigsaw

Rapport vitesse/précision dans l'algorithme génétique. Le diagramme de l'algorithme génétique est totalement différent de celui du recuit. Il apparait deux groupes distincts de points ayant un nombre d'itérations et une qualité proche.

L'augmentation de qualité entre les deux groupes est légérement plus importante que pour le recuit. Cependant, l'important est surtout l'augmentation limitée du nombre d'évaluations et la faible dispersion de points. La différence entre les deux groupes de données est de l'ordre d'un facteur dix.

Ainsi le perte en terme de vitesse reste raisonnable en comparaison avec l'augmentation de qualité. Au contraire du recuit, la réduction de la vitesse est contrôlable et peut être utilisée pour améliorer les résultats obtenus en utilisant plus de temps de calcul.

7.3.2.3 Conclusion sur le choix des algorithmes

Choix du regroupement hiérarchique. La comparaison des différentes instances de ROMANTIC a permis de mettre en évidence les différences entre le recuit et les méta-heuristiques. Ainsi, les méta-heuristiques, recuit et génétique, ont montré leurs supériorités en terme de vitesse sur le regroupement hiérarchique. Nous avons également montré les limites du regroupement hiérarchique en terme d'amélioration : il n'est pas possible d'augmenter le temps de calcul pour augmenter la qualité de la solution.

Le seul avantage du regroupement reste de fournir une version non-stochastique de notre approche. **Cette instance permet de tester le comportement de ROMANTIC en vue de maitriser notre approche et d'utiliser ensuite une méta-heuristique plus performante**.

Choix du recuit simulé. Le recuit simulé s'est montré le plus rapide dans le meilleur des cas. Etant donnée l'importante différence de vitesse entre deux executions et la rapidité dans le meilleur cas, il est possible d'utiliser le recuit en limitant les évaluations inutiles et en redémarant le processus en cas de dépassement d'une limite du nombre d'évaluations. Ce principe est utilisé dans le domaine de la programmation par contrainte.

Avec cette proposition, nous pouvons utiliser le recuit pour extraire rapidement une architecture de bonne qualité. Il s'adresse donc particulièrement aux cas où le temps de calcul est plus important que la perfection de l'architecture extraite. Par exemple, il peut être utile pour vérifier une représentation architecturale disponible avant de lancer une phase de maintenance urgente.

Choix de l'algorithme génétique. L'algorithme génétique a montré une grande fiabilité. En effet, la vitesse du processus est fortement corrélée avec le nombre de générations et la taille de la population. Il est ainsi aisé de déterminer le meilleur paramétrage en fonction du temps disponible.

Sa fiabilité apparait aussi dans le rapport entre la vitesse et la précision. En effet, ce processus représente le meilleur investissement en terme de temps de calcul.

D'aprés notre étude, l'algorithme génétique s'adresse particulièrement au cas où le temps de calcul est moins important que la qualité du résultat. Par exemple, il représente le meilleur choix pour un système patrimonial dont aucune représentation architecturale n'est disponible et qui doit subir une opération de maintenance importante et cruciale.

7.4 Conclusion

Dans ce chapitre, nous avons présenté un cas d'étude de notre approche basée sur deux systèmes réels : Jigsaw et ArgoUML. Ce cas a pour objectif d'étudier les propriétés expérimentales de notre approche afin de fournir une validation de notre approche et une comparaison des instances de ROMANTIC que nous avons décrites.

Nous avons d'abord établi la validité des fondements de notre approche en proposant une méthode en deux étapes montrant respectivement la consistance et l'adéquation de ROMANTIC. Nous avons montré la consistance en étudiant la similarité des solutions extraites et en établissant qu'elles représentent la même architecture. Nous avons ensuite démontré l'adéquation en étudiant les architectures extraites de Jigsaw et de ArgoUML. Cette étude qualitative a mis en évidence l'adéquation entre l'architecture extraite et celle prévue manuellement en utilisant nos connaissances des systèmes.

Nous avons ensuite comparé les algorithmes proposés selon deux axes : la vitesse et le rapport vitesse/précision. Pour cela, nous mesurons la vitesse à travers le nombre d'évaluations de la fonction objectif et la précision à travers le résultat de notre fonction objectif. Cette étude a permis de proposer une classification des instances de ROMANTIC en fonction des cas d'utilisation :

- **le regroupement hiérarchique** : il est utile pour découvrir notre approche et apprendre à maitriser les paramètres de notre fonction objectif. Il est moins efficace que les autres instances mais son caractére non-stochastique le rend précieux pour cette utilisation ;
- **le recuit simulé** : les propriétés du recuit simulé le rendent particulièrement utile dans les cas où le temps de calcul est limité et que la qualité du résultat n'est pas cruciale. En effet, en utilisant le recuit avec une limite du nombre d'évaluations de la fonction objectif et en le redémarant en cas de dépassement, il est possible d'obtenir rapidement une bonne représentation de l'architecture d'un système ;
- **l'algorithme génétique** : les propriétés de cet algorithme le rendent particulièrement efficace dans les cas où le temps de calcul est secondaire par rapport à la qualité du résultat. Il permet en effet un réglage fin du temps de calcul et offre la meilleure garantie d'une qualité de résultat en rapport avec le temps machine utilisé.

PARTIE IV

Conclusion et perspectives

Conclusion

Dans le cadre de cette thèse, nous nous sommes intéressés à la problématique de l'extraction d'architectures logicielles à partir d'un système orienté objet. Nous avons étudié différentes facettes de cette problématique en proposant :

- ## un état de l'art sur la mesure de la qualité logicielle

  L'étude de la terminologie des architectures logicielles et des éléments architecturaux, nous a conduit à approfondir une utilisation courante des architectures dans le domaine du génie logiciel : **la mesure de la qualité d'un système logiciel**. Nous avons proposé une classification des travaux existants de mesure de cette qualité suivant différents critères tels que les objectifs de l'approche, le modèle de qualité utilisé ou les informations utilisées.
  Cette étude nous a permis de présenter les avantages des architectures logicielles dans ce domaine, mais elle a surtout mis en évidence les inconvénients et les avantages de l'utilisation des experts pour diriger un tel processus. Nous avons ainsi montré que l'expertise des architectes permet un diagnostic plus précoce de la qualité d'un système. Cependant, cette intervention des experts humains a un impact direct sur le degré d'automatisation du processus et sur son caractère reproductible. Nous avons également comparé les différents modèles de qualité utilisés et montré notamment les avantages en termes de généricité et d'adaptation du modèle proposé par la norme ISO-9126 [44].

- ## un état de l'art sur l'extraction d'architectures

  Afin de motiver notre approche, nous avons étudié la littérature concernant l'extraction d'architectures logicielles. Cette étude nous a permis de positionner l'extraction par rapport au domaine des architectures logicielles et de la maintenance et de présenter les concepts et les principes généraux de l'extraction. Nous avons proposé une classification des travaux d'extraction d'architectures suivant différents critères en relation avec (i) les concepts architecturaux, (ii) les aspects techniques et (iii) les informations utilisées.
  Cette étude a mis en évidence les limites des approches existantes. Nous avons pu constater leurs limites en ce qui concerne la définition de l'architecture qui est le résultat de l'extraction. Les approches fournissent toutes une architecture où les connecteurs sont des éléments architecturaux de seconde catégorie et la plupart utilisent une notion de composants approximative telle que des modules ou des sous-systèmes.
  La prise en compte des informations intentionnelles, telles que l'architecture imaginée par les concepteurs, est une autre limite des approches existantes. En effet, les approches d'extraction se repartissent majoritairement en deux groupes. Les premières sont des approches manuelles. Elles imposent la présence d'un ou plusieurs experts et offrent une bonne prise en compte de ces informations. En contre-partie elles sont couteuses en expertise et plus difficiles à mettre en place. Les secondes sont automatiques. Elles sont donc plus faciles à utiliser, mais prennent peu en compte les informations intentionnelles.

- **une approche générique d'extraction d'architectures par exploration : ROMANTIC**

  A partir de l'analyse des travaux existants, nous avons proposé une approche semi-automatique permettant d'extraire une architecture logicielle à partir d'un système objet : ROMANTIC. L'objectif de notre approche est d'obtenir un ensemble de composants et de connecteurs qui constituent une architecture selon les définitions couramment admises.
  L'extraction d'architectures consiste à **identifier la meilleure architecture** représentant un système donné. La notion de qualité associée reflète, en fait, un ensemble de contraintes d'origines diverses et parfois contradictoires. Pour résoudre ce problème d'**optimisation**, nous avons modélisé ce problème comme un problème d'**exploration d'un espace de solutions**. Nous avons d'abord décrit un modèle de correspondance Objet/Architecture dont les instances définissent l'espace des solutions. Ensuite, nous avons proposé une fonction d'évaluation des solutions permettant de mesurer la qualité de l'architecture correspondante. Enfin, nous avons présenté un processus d'extraction des informations intentionnelles et nous avons montré comment utiliser ces informations pour améliorer l'extraction.

  1. *Un modèle de correspondance Objet/Architecture*

     La modélisation du problème d'extraction consiste essentiellement à définir l'espace des solutions qui doit être exploré. Nous avons donc proposé un modèle de correspondance entre les entités objets et architecturales (COA) dont les instances constituent les éléments de cet espace.

     Ce modèle COA introduit les notions de contours d'éléments architecturaux qui constituent **le pont entre le monde objet et architectural**. Les contours mettent en correspondance les composants avec un ensemble de classes et les connecteurs avec un ensemble de méthodes.

     Le modèle COA est également la base du traitement équivalent apporté aux composants et aux connecteurs. En effet, le modèle considère ces deux éléments architecturaux de manière identique et définit deux entités similaires pour représenter les correspondances de ces éléments avec les éléments objets : les contours de composants et ceux de connecteurs.

  2. *Une fonction d'évaluation de la qualité d'une architecture*

     L'autre élément essentiel à la modélisation d'un problème d'extraction en tant que problème d'optimisation est la définition de la fonction objectif. Pour définir cette fonction, nous avons utilisé deux éléments : **la sémantique des concepts architecturaux** et **la qualité architecturale**. Pour chacun de ces éléments, nous avons proposé un modèle de mesure reposant sur un méta-modèle adapté de la norme ISO-9126.

     Le premier modèle de mesure permet d'**évaluer la validité d'une solution vis-à-vis de la sémantique** couramment associée aux concepts d'architectures, de composants et de connecteurs. Pour cela, nous avons réalisé, dans le chapitre 4, une étude des définitions des éléments architecturaux et nous avons proposé un ensemble de caractéristiques nécessaires pour qu'une entité logicielle soit, du point de vue de ces définitions, un bon composant (spécificité, composabilité et autonomie) ou un bon connecteur (spécificité, généricité et autonomie). Nous avons ensuite détaillé ces caractéristiques en propriétés sur les éléments architecturaux, tels que le couplage des composants ou la cohésion des connecteurs, avant de les lier à des propriétés sur les éléments du modèle COA, tels que le couplage entre les contours ou la cohésion d'un contour.

Le second modèle de mesure permet d'**évaluer la qualité d'une solution du point de vue de la qualité architecturale**. Pour cela, nous avons utilisé notre étude sur la mesure de la qualité architecturale pour proposer deux caractéristiques de qualité essentielles aux architectures et mesurables dans le cadre de notre approche : la fiabilité et la maintenabilité. Comme pour le modèle de validité sémantique, nous avons ensuite détaillé ces caractéristiques en propriétés sur les éléments architecturaux puis sur les éléments COA.

3. *Un processus d'extraction des informations intentionnelles*

   Les informations utilisées pour définir la fonction objectif reposent uniquement sur le code source de l'application. Or, nous avons constaté les limites des approches d'extraction ne reposant que sur ce type d'informations. Par conséquent, nous avons choisi d'utiliser également les **informations intentionnelles** afin d'extraire une architecture qui corresponde à la fois au système réel et à l'**architecture intentionnelle** imaginée par les concepteurs. Nous avons également proposé d'utiliser les **informations contextuelles** pour fournir une architecture conforme à l'architecture matérielle sur laquelle le système doit être déployé.

   Nous avons identifié deux sources d'informations intentionnelles : la documentation et les recommandations de l'architecte. Nous avons proposé un processus automatique d'extraction des informations intentionnelles contenues dans la documentation issue de chaque phase majeure du cycle de vie : la conception avec les diagrammes UML, l'implémentation avec les informations informelles contenues dans le code source et la maintenance avec les informations extraites des outils de versionnement. Ensuite, nous avons décrit un processus pour récupérer les recommandations de l'architecte en termes d'informations intentionnelles et contextuelles.

   Nous avons exposé les méthodes permettant d'utiliser ces informations extraites pour **réduire l'espace des recherches**. Cette réduction se fait de deux manières. La première utilise les informations pour éliminer certains éléments de l'espace de solutions qui sont en contradiction avec l'architecture intentionnelle. La seconde utilise les informations extraites pour cibler certaines zones de l'espace des solutions qui doivent contenir des solutions plus proches de l'architecture intentionnelle.

• une mise en pratique de ROMANTIC

ROMANTIC est une approche d'extraction basée sur l'exploration de l'espace des architectures possibles. Cette approche est volontairement découplée des algorithmes d'exploration utilisés pour la mettre en pratique. Cela lui confère une **généricité** qui permet d'envisager l'ajout facile d'un nouveau guide, d'une nouvelle caractéristique de qualité ou encore la modification de la terminologie associée aux architectures logicielles.
Afin de valider notre approche, nous avons décrit trois instances de ROMANTIC sous la forme de **trois algorithmes réalisant l'exploration**. Chacun utilise au mieux les éléments décrits dans l'approche générique et présente des caractéristiques et des performances différentes.

1. *Un algorithme à base de regroupement hiérarchique*

   Le premier algorithme que nous avons développé est une **heuristique non-stochastique**. Il se déroule en deux étapes. La première extrait une architecture simplifiée dont les connecteurs sont de simples appels de méthodes et dont les composants n'ont pas d'interfaces définies. Cette étape procède par regroupement hiérarchique des classes puis sélection des meilleurs

contours selon notre fonction objectif. La deuxième étape procède à l'identification des interfaces des composants puis des connecteurs en utilisant des heuristiques développées à partir des analyses sémantiques des éléments architecturaux que nous avons réalisées pour définir notre fonction d'évaluation de la validité sémantique.

Le défaut majeur de cet algorithme est qu'il ne tire pas pleinement partie des informations intentionnelles. Elles sont uniquement utilisées à la place d'un tirage aléatoire pour déterminer certains choix de regroupement lorsque plusieurs choix sont possibles. Cependant, il peut être **utiliser pour découvrir notre approche** et apprendre à maitriser les paramétres de notre fonction objectif. Il est moins efficace que les autres instances mais son caractère non-stochastique le rend précieux pour cette utilisation.

2. *Un algorithme à base de recuit simulé*

   Le second algorithme que nous avons proposé repose sur une méta-heuristique couramment utilisée dans le domaine de l'ingénierie logicielle à base de méta-heuristique : **le recuit simulé**. Cet algorithme explore l'espace de recherche décrit par les instances de notre modèle COA et limité par les informations intentionnelles extraites de la documentation ou des recommandations des architectes.

   Le recuit consiste en une série de modifications des solutions pour passer de l'une à l'autre, l'acceptation d'une nouvelle solution dépendant à la fois de notre fonction objectif, du nombre d'itérations et des informations intentionnelles. Nous avons défini les opérations permettant la modification des solutions afin qu'ils agissent à la fois sur les composants et les connecteurs et permettent une exploration efficace du domaine de recherche.

   Les propriétés de vitesse et de précision du recuit simulé le rendent **particulièrement utile dans les cas où le temps de calcul est limité et que la qualité du résultat n'est pas cruciale**. En effet, en utilisant le recuit avec une limite du nombre d'évaluation de la fonction objectif et en le redémarant en cas de dépassement, il est possible d'obtenir rapidement une bonne représentation de l'architecture d'un système. Il peut ainsi être utilisé pour vérifier une représentation architecturale disponible avant de lancer une phase de maintenance urgente.

3. *Un algorithme à base d'algorithmes génétiques*

   Le dernier algorithme que nous avons proposé, est un **algorithme génétique** ; une méta-heuristique qui vise à explorer l'espace des solutions pour identifier la meilleure possible. Cependant, l'algorithme génétique présente l'avantage de parcourir en parallèle plusieurs solutions représentées comme les individus d'une population. L'algorithme consiste essentiellement à faire évoluer cette population en croisant les individus et en les soumettant à une sélection basée sur notre fonction objectif.

   Pour utiliser les algorithmes génétiques, nous avons adapté la représentation de notre modèle COA afin de manipuler les solutions sous la forme de chromosomes. Dans cette adaptation, nous avons conservé l'équivalence de traitement entre les composants et les connecteurs en proposant de représenter un individu par trois chromosomes : un pour chaque élément architectural. Nous avons également proposé des opérations génétiques s'appliquant aux connecteurs comme aux composants.

   Les propriétés de cet algorithme le rendent **particulièrement efficace dans les cas où le temps de calcul est secondaire par rapport à la qualité du résultat**. Il permet en effet un réglage fin du temps de calcul et offre la meilleure garantie d'une qualité de résultat en rapport avec le temps-machine utilisé.

Perspectives

Les perspectives qui se dégagent de ce travail sont classées en deux catégories suivant l'importance du travail exploratoire effectué. Nous distinguons donc les perspectives à court terme et les perspectives plus lointaines.

Perspectives à court terme

L'étude des travaux concernant l'extraction d'architectures et les architectures logicielles a mis en valeur deux applications de notre approche particulièrement intéressantes et qui constituent des perspectives importantes : la migration d'un système orienté objet vers un système à base de composants et l'extraction de composants réutilisables.

Migration d'un système objet vers un système à base de composants

La migration d'un système objet vers un système à base de composants consiste à recréer un système existant en utilisant le paradigme des composants à la place des objets. Cette migration permet d'obtenir un nouveau système dont le comportement est identique à l'ancien mais qui bénéficie de tous les avantages liés à l'ingénierie logicielle à base de composants : réutilisation accrue, facilité de maintenance *etc.*. Cependant, malgré ces avantages, la migration reste une opération coûteuse et difficile à réaliser de manière manuelle.

Différents travaux se sont déjà intéressés à la migration des systèmes procéduraux vers les systèmes objets [25, 45, 110]. Ils ont proposé des approches automatiques ou semi-automatiques d'identification des concepts objet dans les systèmes procéduraux. Par contre, il n'existe pas de travaux proposant de réaliser une **migration automatique d'un système objets vers un système à base de composants**. Par contre, les liens étroits entre les concepts architecturaux et ceux du paradigme composant font que l'architecture détaillée d'un système peut permettre de faciliter une migration en réduisant les risques d'erreurs et en proposant une identification des composants dans le code objet. Cependant, cette migration reste souvent manuelle à cause du manque de détails de la représentation architecturale, en particulier au niveau des connecteurs et des interfaces de composants.

Notre approche, contrairement aux approches existantes d'extraction, propose une identification détaillée des connecteurs et des interfaces. Nous pouvons donc envisager d'utiliser le résultat de notre processus d'extraction pour guider la migration d'un système objet vers un système à base de composants.

Pour réaliser cette migration, nous avons besoin d'un processus permettant de générer le code source du nouveau système à partir du code source existant et de l'architecture extraite par ROMANTIC. Pour cela, nous avons envisagé d'utiliser une approche d'adaptation structurelle des composants nommée SCORPIO [13].

SCORPIO propose de modifier la structure d'un composant tout en préservant ses services et son comportement. Pour cela, BASTIDE propose un processus basé sur l'analyse et l'instrumentation du code ainsi qu'un modèle de composant structurellement adaptable. En fait, le processus agit sur le code

d'un composant pour le fractionner en fonction d'un spécification proposée par l'utilisateur ou générée automatiquement en fonction du contexte [12]. Le résultat est un composant composite dont les sous-composants respectent les spécifications. SCORPIO propose ensuite des mécanismes permettant de distribuer ce composite en répartissant ces sous-composants sur différentes machines [112].

Nous pouvons utiliser SCORPIO en considérant notre application comme un composant que l'on souhaite restructurer. Il faut ensuite considérer l'architecture comme la spécification du processus de fractionnement. Pour réaliser cela, nous devons adapter SCORPIO pour qu'il tienne compte des informations supplémentaires contenues dans l'architecture par rapport à sa spécification habituelle. Nous devons également adapter le modèle de composants adaptables de SCORPIO pour qu'il soit compatible avec notre modèle de composants.

Extraction de composants réutilisables

La réutilisation d'entités logicielles sous la forme de composants permet, dans le cadre du développement à base de composants, de réduire le cout de production des nouveaux systèmes et de leur maintenance. Cependant, les bibliothèques de composants ne sont pas toujours disponibles ou suffisamment fournies pour répondre à toutes les exigences fonctionnelles. Par contre, il existe, à travers les programmes objets existants, une importante source de codes qui peuvent satisfaire, au moins partiellement, ces exigences. L'extraction de ces morceaux de codes sous la forme de composants réutilisables permettrait de **développer de nouveaux logiciels à partir des programmes objets existants et éprouvés**.

Cependant, le réutilisation de parties d'un programme objet est une procédure lente et peu efficace. En effet, la réutilisation d'une fonctionnalité proposée par un ensemble de classes impose un examen minutieux des dépendances entre les classes concernées et le reste du système. Or, un tel examen manuel risque de réduire ou d'éliminer les avantages de la réutilisation.

Pour résoudre cela, des travaux d'extraction de composants réutilisables sont apparus [129, 115]. Ces travaux sont toujours spécifiques à un modèle de composants et visent surtout à embarquer dans un composant tous les éléments nécessaires à une ou plusieurs classes désignées par l'utilisateur.

Nous pouvons utiliser les travaux que nous avons effectué sur l'identification des composants dans une architecture pour développer une approche d'extraction de composants réutilisables. La différence majeure avec notre approche actuelle est le fait que l'on ne souhaite plus décrire l'ensemble du système mais simplement une fonctionnalité. Il est possible d'adapter notre processus pour rechercher un bon composant réutilisable dans le système, en utilisant notre fonction d'évaluation de la validité sémantique des composants ainsi que les fonctions d'évaluation de la qualité des composants.

Nous pouvons également utiliser les informations intentionnelles que nous utilisons comme guide pour faciliter la spécification de notre processus d'extraction. Dans ce cas, l'utilisateur doit fournir, non pas un ensemble de classes qu'il souhaite réutiliser, mais la description d'une fonctionnalité sous la forme d'un cas d'utilisation inclus dans les diagrammes UML ou d'un mot clé présent dans les commentaires du code source.

Perspectives à long terme

Deux autres perspectives importantes se dégagent de nos travaux.

Extension vers la détection fonctionnelle des connecteurs

Nous avons proposé, dans le cadre de notre approche, une identification des connecteurs basée sur leurs structures. En effet, les caractéristiques que nous avons retenues pour décrire la sémantique des connecteurs sont toutes des caractéristiques structurelles : l'autonomie, la généricité et la spécificité.

Cependant, nous avons également vu, notamment à travers la taxonomie proposée par MEDVIDOVIC [90], que les connecteurs sont avant tout définis dans la littérature à travers les services qu'ils proposent aux composants [114].

Ainsi, une extension importante de nos travaux consiste à **inclure des caractéristiques fonctionnelles dans la fonction d'évaluation des connecteurs** afin de prendre en compte cette dimension majeure. Ces caractéristiques doivent être établies en fonction des services que peuvent rendre les connecteurs, mais aussi en fonction des méthodes de détection disponibles.

En effet, certains connecteurs complexes peuvent apparaître comme un composant dont la fonctionnalité consiste à fournir **un service de communication**. Il convient donc avant tout de fournir des méthodes de détection de ces fonctionnalités, par exemple en utilisant des **patrons de conception**.

Extension vers la documentation des composants

L'architecture résultant de notre approche est décrite en détails. Elle contient une configuration, des connecteurs, des composants, ainsi que les interfaces de ces composants. Cependant, la liste des interfaces est la seule description des composants que nous proposons.

Or, la description des composants est un problème essentiel dans le domaine du développement à base de composant. Il est au cœur des travaux de classification [7]. En effet, cette description est essentielle pour faciliter la réutilisation du composant puisqu'elle permet d'identifier les composants utiles sans avoir besoin d'étudier la liste de ces interfaces, pas toujours très parlante, ou même le code source, pas toujours disponible.

Une autre extension importante de nos travaux, consiste à proposer un **processus automatique de documentation des fonctionnalités des composants**. Pour cela, il est possible d'utiliser d'abord les informations intentionnelles pour fournir une première description des fonctionnalités. En effet, il est possible d'utiliser les mots clefs identifiés par notre processus d'extraction des informations intentionnelles contenues dans le code source, ou encore d'utiliser les cas d'utilisation liés aux classes à travers les diagrammes UML. Nous pouvons aussi procéder en utilisant des patrons sur le code ou sur les interfaces pour identifier les éléments décrivant le mieux les fonctionnalités du composant.

Bibliographie

[1] Gregory D. ABOWD, Robert ALLEN et David GARLAN.
Formalizing style to understand descriptions of software architecture.
ACM Trans. Softw. Eng. Methodol., 4(4) :319–364, 1995.

[2] Robert John ALLEN.
A formal approach to software architecture.
Thèse de Doctorat, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.
Chair-David Garlan,

[3] Rapide Design Team Program ANALYSIS et Verification GROUP.
Guide to the rapide 1.0 language reference manuals.
Rapport technique, Computer Systems Lab Stanford University, Stanford, USA, july 1997.

[4] Nicolas ANQUETIL, Cédric FOURRIER et Timothy C. LETHBRIDGE.
Experiments with clustering as a software remodularization method.
In *Proc. of the Sixth WCRE*, page 235. IEEE, 1999.

[5] Nicolas ANQUETIL et Timothy C. LETHBRIDGE.
Recovering software architecture from the names of source files.
Journal of Software Maintenance, 11(3) :201–221, 1999.

[6] Giulio ANTONIOL, Massimiliano Di PENTA et Mark HARMAN.
Search-based techniques applied to optimization of project planning for a massive maintenance project.
In *ICSM*, pages 240–249. IEEE Computer Society, 2005.

[7] Gabriela ARÉVALO, Nicolas DESNOS, Marianne HUCHARD, Christelle URTADO et Sylvain VAUTTIER.
Construction dynamique d'annuaires de composants par classification de services.
In *CAL*, pages 123–138, 2008.

[8] Anthony J. BAGNALL, Victor J. RAYWARD-SMITH et I. M. WHITTLEY.
The next release problem.
Information & Software Technology, 43(14) :883–890, 2001.

[9] Franck BARBIER, Brian HENDERSON-SELLERS, Annig LE PARC-LACAYRELLE et Jean-Michel BRUEL.
Formalization of the whole-part relationship in the unified modeling language.
IEEE Trans. Softw. Eng., 29(5) :459–470, 2003.

[10] André BARESEL, David BINKLEY, Mark HARMAN et Bogdan KOREL.
Evolutionary testing in the presence of loop-assigned flags : a testability transformation approach.
In George S. AVRUNIN et Gregg ROTHERMEL, réds., *ISSTA*, pages 108–118. ACM, 2004.

[11] Len BASS, Paul CLEMENTS et Rick KAZMAN.
Software Architecture in Practice.
Addison-Wesley Professional, 2003.

[12] G. BASTIDE, A. SERIAI et M. OUSSALAH.
Restructuration de composants logiciels : Une approche d'adaptation structurelle de composants logiciels monolithiques basée sur leur refactorisation.

RSTI - L'Objet, 13(1) :81–116, 2007.

[13] Gautier BASTIDE.
Scorpio : une Approche d'Adaptation Structurelle de Composants Logiciels - Application aux Environnements Ubiquitaires.
Thèse de Doctorat, Université de Nantes, 2007.

[14] Nicolas BELLOIR.
Composition conceptuelle basée sur la relation Tout-Partie.
Thèse de Doctorat, Université de Pau et des Pays de l'Adour, 2004.

[15] PerOlof BENGTSSON, Nico H. LASSING, Jan BOSCH et Hans van VLIET.
Architecture-level modifiability analysis (alma).
Journal of Systems and Software, 69(1-2) :129–147, 2004.

[16] Antonia BERTOLINO, Antonio BUCCHIARONE, Stefania GNESI et Henry MUCCINI.
An architecture-centric approach for producing quality systems.
In *QoSA/SOQUA*, pages 21–37, 2005.

[17] Dirk BEYER.
Clustering software artifacts based on frequent common changes.
In *In Proc. IWPC*, pages 259–268. IEEE, 2005.

[18] James M. BIEMAN et Byung-Kyoo KANG.
Cohesion and reuse in an object-oriented system.
In *Proc. of the Symp. on Software reusability,SSR '95*, pages 259–262, USA, 1995. ACM Press.

[19] Ted J. BIGGERSTAFF.
Design recovery for maintenance and reuse.
Computer, 22(7) :36–49, 1989.

[20] B. BOEHM, J.R. BROWN, H. KASPAR, M. LIPOW, G. MCLEOD et M. MERRITT.
Characteristics of Software Quality.
Elsevier, 1978.

[21] J. BOSCH.
Design and Use of Software Architectures : Adopting and Evolving a Product Line Approach.
Pearson Education (Addison-Wesley & ACM Press), 2000.

[22] Don BOX.
Essential COM.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
Foreword By-Booch,, Grady and Foreword By-Kindel,, Charlie,

[23] Lionel BRIAND, Prem DEVANBU et Walcelio MELO.
An investigation into coupling measures for C++.
In *Proc. of the Int. Conf. on Software Engineering*, pages 412–421. ACM, 1997.

[24] Holger BÄR.
Famix c++ language plug-in 1.0.
Rapport technique, University of Berne, September 1999.

[25] G. CANFORA, A. CIMITILE et M. MUNRO.
An improved algorithm for identifying objects in code.
Softw. Pract. Exper., 26(1) :25–48, 1996.

[26] Joseph P. CAVANO et James A. MCCALL.
A framework for the measurement of software quality.

In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pages 133–139, 1978.

[27] Djalel CHEFROUR et Françoise ANDRÉ.
Aceel : modèle de composants auto-adaptatifs.
In *Systèmes à composants adaptables et extensibles, Grenoble, France*, Grenoble, France, 2002.

[28] Siddhartha CHIB et Edward GREENBERG.
Understanding the metropolis-hastings algorithm.
The American Statistician, 49(4) :327–335, 1995.

[29] Shyam CHIDAMBER et Chris KEMERER.
A metrics suite for object-oriented design.
IEEE Trans. on Software Engineering, 20(6) :476–493, June 1994.

[30] Shyam R. CHIDAMBER et Chris F. KEMERER.
Towards a metrics suite for object oriented design.
In *OOPSLA '91 : Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, New York, NY, USA, 1991. ACM Press.

[31] Paul C. CLEMENTS, Rick KAZMAN et Mark KLEIN.
Evaluating software architectures : methods and case studies.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[32] Jacob COHEN.
A coefficient of agreement for nominal scales.
Educational and Psychological Measurement, 20(1) :37–46, April 1960.

[33] Bernard COULANGE.
Software Reuse.
Springer-Verlag, 1998.

[34] C. DARWIN.
On the Origin of Species A facsimile of the first edition.
Harvard University Press, July 1975.

[35] Remco C. de BOER et Hans van VLIET.
Architectural knowledge discovery with latent semantic analysis : Constructing a reading guide for software product audits.
J. Syst. Softw., 81(9) :1456–1469, 2008.

[36] Serge DEMEYER, Stéphane DUCASSE et Oscar Marius NIERSTRASZ.
Object-oriented reengineering patterns.
Morgan Kaufmann, 2003.

[37] Serge DEMEYER, Stéphane DUCASSE et Er TICHELAAR.
Why unified is not universal ? uml shortcomings for coping with round-trip engineering.
In *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, volume 1723, pages 630–645. Springer-Verlag, 1999.

[38] Serge DEMEYER, Sander TICHELAAR et Patrick STEYAERT.
Famix 2.0 - the famoos information exchange model.
Rapport technique, University of Berne, August 1999.

[39] D. DOVAL, S. MANCORIDIS et B. S. MITCHELL.
Automatic clustering of software systems using a genetic algorithm.
In *STEP '99 : Proceedings of the Software Technology and Engineering Practice*, page 73, Washington, DC, USA, 1999. IEEE Computer Society.

[40] J. DRÉO, A. PÉTROWSKI, P. SIARRY et É. D. TAILLARD.
Métaheuristiques pour l'optimisation difficile.
Eyrolles, septembre 2003.

[41] Emanuel FALKENAUER.
Genetic Algorithms and Grouping Problems.
John Wiley & Sons, Inc., New York, NY, USA, 1998.

[42] J.-M. FAVRE, H. CEVANTES, F. DUCLOS, R. SANLAVILLE et J. ESTUBLIER.
Issues in reengineering the architecture of component-based software.
In *In Proceedings of the WCRE 2001 Discussion forum on Software Architecture Recovery and Modeling (SWARM 2001*. CWI, 2001.

[43] P. FINNIGAN, R. HOLT et AL..
The software bookshelf.
IBM systems Journal, 36(4) :564–593, novembre 1997.

[44] International Organization for STANDARDIZATION.
ISO 9126-1 Software Engineering - Product Quality - Part 1 : Quality Model.
International Organization for Standardization, 2001.

[45] H. GALL et R. KLOSCH.
Finding objects in procedural programs : an alternative approach.
In *WCRE '95 : Proceedings of the Second Working Conference on Reverse Engineering*, page 208, Washington, DC, USA, 1995. IEEE Computer Society.

[46] D. GARLAN.
A introduction to the aesop system.
Rapport technique, School of Computer Science Carnegie Mellon Universtity, Pittsburgh, PA, USA, July 1995.

[47] D. GARLAN et D.E. PERRY.
Introduction to the special issue on software architecture.
IEEE Transactions on Software Engineering, 21(4) :269–274, 1995.

[48] David GARLAN.
Software architecture : a roadmap.
In *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, New York, NY, USA, 2000. ACM.

[49] David GARLAN et Mary SHAW.
An introduction to software architecture.
Rapport technique, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[50] Katerina GOSEVA-POPSTOJANOVA, Aditya P. MATHUR et Kishor S. TRIVEDI.
Comparison of architecture-based software reliability models.
In *ISSRE*, pages 22–33. IEEE Computer Society, 2001.

[51] Robert B. GRADY et Deborah L. CASWELL.
Software metrics : establishing a company-wide program.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[52] V. GRANVILLE, M. KRIVÁNEK et J. P. RASSON.
Simulated annealing : A proof of convergence.
IEEE Trans. Pattern Anal. Mach. Intell., 16(6) :652–656, 1994.

[53] Anna GRIMÁN, María A. PÉREZ, Luis Eduardo MENDOZA et Francisca LOSAVIO.

Feature analysis for architectural evaluation methods.
Journal of Systems and Software, 79(6) :871–888, 2006.

[54] Anna GRIMÁN, María A. PÉREZ, Luis Eduardo MENDOZA et Edumilis Maria MÉNDEZ.
A method proposal for architectural reliability evaluation.
In Jorge CARDOSO, José CORDEIRO et Joaquim FILIPE, réds., *ICEIS (1)*, pages 564–568, 2007.

[55] Object Management GROUP.
Corba components specification, version 3.0, omg tc document formal/2002-06-65.
Rapport technique, Object Management GROUP, 2002.
Disponible à l'adresse http ://omg.org/cgi-bin/doc ?formal/2002-06-65.

[56] Lars GRUNSKE.
Early quality prediction of component-based systems - a generic framework.
Journal of Systems and Software, 80(5) :678–686, 2007.

[57] Yann-Gaël GUÉHÉNEUC, Jean-Yves GUYOMARC'H et Houari SAHRAOUI.
Improving design pattern identification : a new approach and an exploratory study.
Software Quality Journal, 2009.

[58] Mark HARMAN.
The current state and future of search based software engineering.
In *Future of Software Engineering*, pages 342–357. IEEE, 2007.

[59] Mark HARMAN, Robert M. HIERONS et Mark PROCTOR.
A new representation and crossover operator for search-based optimization of software modularization.
In *GECCO '02 : Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[60] Mark HARMAN, Stephen SWIFT et Kiarash MAHDAVI.
An empirical study of the robustness of two module clustering fitness functions.
In *GECCO '05 : Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1029–1036, New York, NY, USA, 2005. ACM.

[61] D. R. HARRIS, H. B. REUBENSTEIN et A. S. YEH.
Reverse engineering to the architectural level.
In *Proc. of ICSE*, pages 186–195. ACM, Inc., 1995.

[62] G.T. HEINEMANN et W.T. COUNCILL.
Component-based software engineering.
Addison-Wesley, 2001.

[63] Sallie M. HENRY et Dennis G. KAFURA.
Software structure metrics based on information flow.
IEEE Trans. Software Eng., 7(5) :510–518, 1981.

[64] Martin HITZ et Behzad MONTAZERI.
Measuring coupling and cohesion in object-oriented systems.
In *Proc. Intl. Sym. on Applied Corporate Computing*, Nov 1996.

[65] John H HOLLAND.
Adaptation in Natural and Artificial Systems.
University of Michigan Press, Ann Arbor, 1975.

[66] Mehdi JAZAYERI, Alexander RAN et Frank van der LINDEN.
Software architecture for product families : principles and practice.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[67] S. JENKINS et S. R. KIRK.
Software architecture graphs as complex networks : A novel partitioning scheme to measure stability and evolution.
Inf. Sci., 177(12) :2587–2601, 2007.

[68] S.C. JOHNSON.
Hierarchical clustering schemes.
Psychometrika, 32 :241–245, 1967.

[69] R. KAZMAN, L. O'BRIEN et C. VERHOEF.
Architecture reconstruction guidelines.
Rapport technique, Software engineering institute, Carnegie-Mellon university, 2001.

[70] R. KAZMAN, S. WOODS et S. CARRIÈRE.
Requirements for integrating software architecture and reengineering models : Corum ii.
In *Working Conference on Reverse Engineering*, pages 154–163. IEEE Computer Society Press, 1998.

[71] Rick KAZMAN, Gregory D. ABOWD, Leonard J. BASS et Paul C. CLEMENTS.
Scenario-based analysis of software architecture.
IEEE Software, 13(6) :47–55, 1996.

[72] Rick KAZMAN et S. Jeromy CARRIÈRE.
Playing detective : Reconstructing software architecture from available evidence.
Automated Software Engg., 6(2) :107–138, 1999.

[73] S. KIRKPATRICK, C. D. GELATT et M. P. VECCHI.
Optimization by simulated annealing.
Science, 220 :671–680, 1983.

[74] Rainer KOSCHKE.
Atomic Architectural Component Recovery for Program Understanding and Evolution.
Thèse de Doctorat, University of Stuttgart, 2000.

[75] René L. KRIKHAAR.
Reverse architecting approach for complex systems.
In *ICSM '97 : Proceedings of the International Conference on Software Maintenance*, pages 4–11, Washington, DC, USA, 1997. IEEE Computer Society.

[76] Adrian KUHN, Stéphane DUCASSE et Tudor GÍRBA.
Semantic clustering : Identifying topics in source code.
Inf. Softw. Technol., 49(3) :230–243, 2007.

[77] C. H. KUNG, J. GAO, P. HSIA, J. LIN et Y. TOYOSHIMA.
Design recovery for software testing of object-oriented programs.
In *Proc. of the Working Conf. on Reverse Engineering,WCRE '93*, pages 202–211. IEEE, 1993.

[78] P. J. M. LAARHOVEN et E. H. L. AARTS, réds.
Simulated annealing : theory and applications.
Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[79] Thomas K. LANDAUER, Peter W. FOLTZ et Darrell LAHAM.
An introduction to latent semantic analysis.
Discourse Processes, 25 :259–284, 1998.

[80] Y LEE, B LIANG, S WU et F WANG.
Measuring the coupling and cohesion of an object-oriented program based on information flow.
In *ICSQ'95*, pages 81–90, 1995.

[81] Shaoming LI, Qian YIN, Ping GUO et Michael R. LYU.
A hierarchical mixture model for software reliability prediction.
Applied Mathematics and Computation, 185(2) :1120–1130, 2007.

[82] W. LI et S. HENRY.
Object oriented metrics that predict maintainability.
Journal of Systems and Software, 223 :111–122, 1993.

[83] Francisca LOSAVIO, Ledis CHIRINOS, Nicole LÉVY et Amar RAMDANE-CHERIF.
Quality characteristics for software architecture.
Journal of Object Technology, 2(2) :133–150, 2003.

[84] Francisca LOSAVIO, Ledis CHIRINOS, Alfredo MATTEO, Nicole LÉVY et Amar RAMDANE-CHERIF.
Iso quality standards for measuring architectures.
Journal of Systems and Software, 72(2) :209–223, 2004.

[85] C. LUER et A. van der HOEK.
Composition environments for deployable software components.
Rapport technique, Dept. of Information and Computer Science, University of California, 2002.

[86] Spiros MANCORIDIS, Brian S. MITCHELL, Yih-Farn CHEN et Emden R. GANSNER.
Bunch : A clustering tool for the recovery and maintenance of software system structures.
In *ICSM*, pages 50–, 1999.

[87] Spiros MANCORIDIS, Brian S. MITCHELL, C. RORRES, Yih-Farn CHEN et Emden R. GANSNER.
Using automatic clustering to produce high-level system organizations of source code.
In *IWPC*, pages 45–. IEEE Computer Society, 1998.

[88] Nenad MEDVIDOVIC et Vladimir JAKOBAC.
Using software evolution to focus architectural recovery.
Automated Software Engg., 13(2) :225–256, 2006.

[89] Nenad MEDVIDOVIC et Richard N. TAYLOR.
A classification and comparison framework for software architecture description languages.
IEEE Trans. Softw. Eng., 26(1) :70–93, 2000.

[90] Nikunj R. MEHTA, Nenad MEDVIDOVIC et Sandeep PHADKE.
Towards a taxonomy of software connectors.
In *ICSE '00 : Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM.

[91] Kim MENS, Andy KELLENS, Frédéric PLUQUET et Roel WUYTS.
Co-evolving code and design with intensional views : A case study.
Computer Languages, Systems & Structures, 32(2-3) :140–156, 2006.

[92] Brian S. MITCHELL et Spiros MANCORIDIS.
Craft : A framework for evaluating software clustering results in the absence of benchmark decompositions.
In *WCRE*, pages 93–102, 2001.

[93] Brian S. MITCHELL et Spiros MANCORIDIS.
Using heuristic search techniques to extract design abstractions from source code.
In *GECCO*, pages 1375–1382. Morgan Kaufmann, 2002.

[94] Brian S. MITCHELL et Spiros MANCORIDIS.
On the evaluation of the bunch search-based software modularization algorithm.
Soft Comput., 12(1) :77–93, 2008.

[95] Gail C. MURPHY, David NOTKIN et Kevin SULLIVAN.
Software reflexion models : bridging the gap between source and high-level models.
In *SIGSOFT '95*, pages 18–28. ACM Press, 1995.

[96] Robb NEBBE.
Famix java language plug-in 2.2.
Rapport technique, University of Berne, August 1999.

[97] Software Engineering Standards Committee of the IEEE COMPUTER SOCIETY.
1219-1998, IEEE standard for software maintenance.
the IEEE Computer Society, USA, 1998.

[98] Mark O'KEEFFE et Mel Ó CINNÉIDE.
Search-based software maintenance.
In *Proc. of the CSMR*, pages 249–260. IEEE Computer Society, 2006.

[99] OMG.
UML 2.0 Superstructure Specification.
Object Management Group, 2003.

[100] M. OUSSALAH.
Ingénierie des composants : concepts, techniques et outils.
Editions Vuibert, 2005.

[101] Witold PEDRYCZ, Liting HAN, James F. PETERS, Sheela RAMANNA et R. ZHAI.
Calibration of software quality : Fuzzy neural and rough neural computing approaches.
Neurocomputing, 36(1-4) :149–170, 2001.

[102] Dewayne E. PERRY et Alexander L. WOLF.
Foundations for the study of software architecture.
SIGSOFT Softw. Eng. Notes, 17(4) :40–52, 1992.

[103] Damien POLLET, Stephane DUCASSE, Loic POYET, Ilham ALLOUI, Sorana CIMPAN et Herve VERJUS.
Towards a process-oriented software architecture reconstruction taxonomy.
In *Proc. of the CSMR*, pages 137–148. IEEE, 2007.

[104] L.B.S. RACOON.
Fifty years of progress in software engineering.
Software Engineering Notes, 22(1), 1997.

[105] Srinivasan RAMANI et Kishor S. TRIVEDI.
Srept : Software reliability estimation and prediction tool.
In Boudewijn R. HAVERKORT, Henrik C. BOHNENKAMP et Connie U. SMITH, réds., *Computer Performance Evaluation / TOOLS*, volume 1786 of *Lecture Notes in Computer Science*, pages 358–361. Springer, 2000.

[106] Ralf REUSSNER, Heinz W. SCHMIDT et Iman POERNOMO.
Reliability prediction for component-based software architectures.
Journal of Systems and Software, 66(3) :241–252, 2003.

[107] Elaine RICH et Kevin KNIGHT.
Artificial Intelligence.
McGraw-Hill Higher Education, 1990.

[108] Claudio RIVA.
Reverse architecting : An industrial experience report.

In *WCRE '00 : Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[109] David S. ROSENBLUM et Alexander L. WOLF.
A design framework for internet-scale event observation and notification.
SIGSOFT Softw. Eng. Notes, 22(6) :344–360, 1997.

[110] Houari A. SAHRAOUI, Hakim LOUNIS, Walcélio MELO et Hafedh MILI.
A concept formation based approach to object identification in procedural code.
Automated Software Engg., 6(4) :387–410, 1999.

[111] Olaf SENG, Markus BAUER, Matthias BIEHL et Gert PACHE.
Search-based improvement of subsystem decompositions.
In *GECCO '05 : Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1045–1051, New York, NY, USA, 2005. ACM.

[112] Abdelhak SERIAI, Gautier BASTIDE et Mourad OUSSALAH.
How to generate distributed software components from centralized ones ?
JCP, 1(5) :40–52, 2006.

[113] Vibhu Saujanya SHARMA et Kishor S. TRIVEDI.
Quantifying software performance, reliability and security : An architecture-based approach.
Journal of Systems and Software, 80(4) :493–509, 2007.

[114] Mary SHAW et David GARLAN.
Software architecture : perspectives on an emerging discipline.
Prentice-Hall, USA, 1996.

[115] M.S. SONG, H.T. JUNG et Y.J. YANG.
The analysis technique for extraction of ejb component from legacy system.
In *Proc. 6th IASTED International Conference on Software Engineering and Applications*, pages 241–244, 2002.

[116] H. STEINHAUS.
Sur la division des corps matériels en parties.
Bull. Acad. Polon. Sci, 1 :801–804, 1956.

[117] Mikael SVAHNBERG.
An industrial study on building consensus around software architectures and quality attributes.
Information & Software Technology, 46(12) :805–818, 2004.

[118] Clemens SZYPERSKI.
Component Software.
ISBN : 0-201-17888-5. Addison-Wesley, 1998.

[119] Ladan TAHVILDARI.
Quality-Drive Object-Oriented Re-engineering Framework.
Thèse de Doctorat, University of Waterloo, 2003.

[120] Ladan TAHVILDARI et Kostas KONTOGIANNIS.
A software transformation framework for quality-driven object-oriented re-engineering.
In *ICSM*, pages 596–. IEEE Computer Society, 2002.

[121] Robert TARJAN.
Depth-first search and linear graph algorithms.
SIAM Journal on Computing, 1(2) :146–160, 1972.

[122] Sander TICHELAAR.

Famix java language plug-in 1.0.
Rapport technique, University of Berne, September 1999.

[123] V. TZERPOS et R. HOLT.
A hybrid process for recovering software architecture.
In *Proc. of the conf. of the Centre for Advanced Studies on Collaborative Research,CASCON*, pages 1–6, 1996.

[124] Vassilios TZERPOS.
The orphan adoption problem in architecture maintenance.
In *WCRE '97 : Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 76, Washington, DC, USA, 1997. IEEE Computer Society.

[125] Vassilios TZERPOS et Richard C. HOLT.
Mojo : A distance metric for software clusterings.
In *WCRE*, pages 187–, 1999.

[126] Pieter van der SPEK, Steven KLUSENER et Piërre van de LAAR.
Towards recovering architectural concepts using latent semantic indexing.
In *CSMR*, pages 253–257. IEEE, 2008.

[127] V. ČERNÝ.
A thermodynamical approach to the traveling salesman problem : An efficient simulation algorithm.
Journal of Optimization Theory and Applications, 45(1) :41–51, January 1985.

[128] Wen-Li WANG, Dai PAN et Mei-Hwa CHEN.
Architecture-based software reliability modeling.
Journal of Systems and Software, 79(1) :132–146, 2006.

[129] Hironori WASHIZAKI et Yoshiaki FUKAZAWA.
A technique for automatic component extraction from object-oriented programs by refactoring.
Sci. Comput. Program., 56(1-2) :99–116, 2005.

[130] Hironori WASHIZAKI, Hirokazu YAMAMOTO et Yoshiaki FUKAZAWA.
A metrics suite for measuring reusability of software components.
In *METRICS '03 : Proceedings of the 9th International Symposium on Software Metrics*, page 211, Washington, DC, USA, 2003. IEEE Computer Society.

[131] Zhihua WEN et Vassilios TZERPOS.
Evaluating similarity measures for software decompositions.
In *ICSM*, pages 368–377. IEEE Computer Society, 2004.

[132] David H. WOLPERT et William G. MACREADY.
No free lunch theorems for optimization.
IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, 1(1) :67–82, 1997.

[133] Sherif M. YACOUB et Hany H. AMMAR.
A methodology for architecture-level reliability risk analysis.
IEEE Trans. Softw. Eng., 28(6) :529–547, 2002.

[134] Robert K. YIN.
Case Study Research : Design and Methods (Applied Social Research Methods).
SAGE Publications, 2002.

[135] E. YOURDON et L.L. CONSTANTINE.
Structured Design.
Prentice Hall, Englewood Cliffs, 1979.

[136] G. ZELESNIK.
The unicon language reference manual.
Rapport technique, School of Computer Science Carnegie Mellon Universtity, Pittsburgh, PA, USA, May 1996.
Disponible à l'adresse http ://www.cs.cmu.edu/ UniCon.

Références bibliographiques relatives à ce travail de recherche

Revues

1. S. Chardigny, A. Seriai, M. Oussalah, D. Tamzalit : **Extraction dirigée par les propriétés de qualité d'une architecture à partir d'un système orienté objet** *L'Objet, Vol. 14, pp.113-137, Hermes Editions, 2008.*

Conférences internationales

1. S. Chardigny, A. Seriai, M. Oussalah, D. Tamzalit : **Extraction of Component-Based Architecture From Object Oriented Systems** *WICSA 2008, pp.285-288, 2008.*
2. S. Chardigny, A. Seriai, D. Tamzalit, M. Oussalah : **Quality-Driven Extraction of a Component-based Architecture from an Object-Oriented System** *CSMR 2008, pp.269-273, 2008.*
3. S. Chardigny, A. Seriai, D. Tamzalit, M. Oussalah : **Search-Based Extraction of Component-Based Architecture from Object-Oriented Systems** *ECSA 2008, pp.322-325, 2008.*

Ateliers internationaux

1. S. Chardigny, A. Seriai, M. Oussalah, D. Tamzalit : **Extraction of Component-Based Architecture from Object-Oriented Systems** *the Third International ERCIM Symposium on Software Evolution , Paris , 05 october 2007.*

Conférences francophones

1. S. Chardigny, A. Seriai, M. Oussalah, D. Tamzalit : **Extraction métaheuristique d'une architecture à base de composants à partir d'un système orienté objet** *LMO/CAL 2008, pp.139-154, 2008.*
2. S. Chardigny, A. Seriai, M. Oussalah, D. Tamzalit : **Extraction d'Architecture à Base de Composants d'un Système Orienté Objet** *INFORSID 2007, pp.487-502, 2007.*

Ateliers francophones

1. S. Chardigny, A. Seriai, M. Oussalah, D. Tamzalit : **Guides pour l'extraction d'architecture dans un système orienté objet** *IDM 2007, Atelier Rimel, Toulouse, 19-20 mars 2007.*

Liste des tableaux

— Corps du document —

Partie I — Contexte et travaux connexes

1.1 Caractéristiques de l'axe objectif des principales approches SQM . . . 22
1.2 Caractéristiques de l'axe modèle des principales approches SQM . . . 26
1.3 Caractéristiques de l'axe entrée des principales approches SQM . . . 27
1.4 Caractéristiques de l'axe technique des principales approches SQM . . . 29
1.5 Caractéristiques de l'axe sortie des principales approches SQM . . . 30

Partie II — La démarche ROMANTIC pour l'extraction d'architectures

4.1 Relation entre les sous-caractéristiques sémantiques et les propriétés des composants . . . 91
4.2 Relation entre les sous-caractéristiques sémantiques et les propriétés des connecteurs . . . 93
4.3 Les différents types de relations de composition verticale . . . 99

5.1 Informations intentionnelles disponibles à travers un outil de versionnement . . . 122
5.2 La matrice des occurrences . . . 133
5.3 La matrice reconstruite . . . 133
5.4 Mesure de similarité entre les méthodes exemples . . . 134

Partie III — Mise en pratique et validation de ROMANTIC

7.1 Mesure du degré de similarité selon la granularité « classe » . . . 187
7.2 Mesure du degré de similarité selon la granularité « méthode » . . . 188

Partie IV — Conclusion et perspectives

Liste des figures

— *Corps du document* —

Partie I — Contexte et travaux connexes

1.1 Représentation d'un composant 6
1.2 Exemple de structure interne pour les composants composites 8
1.3 Représentation d'un connecteur 11
1.4 Exemple de connecteur composite 12
1.5 Architecture selon le style *pipe and filter* 13
1.6 Fluctuation de l'intérêt pour les paradigmes d'ingénierie 16
1.7 Ordre partiel des types d'architecture selon leur granularité 17
1.8 Type de code source et représentations architecturales 18
1.9 Modèle de classification des travaux d'évaluation d'architectures 20
1.10 Modèle de qualité de MCCALL 23
1.11 Méta-modèle de mesure des caractéristiques d'un logiciel dans la norme ISO-9126 24
1.12 Modèle des caractéristiques d'un logiciel dans la norme ISO-9126 25
1.13 Décomposition de la caractéristique de maintenabilité 26

2.1 Etapes de l'extraction d'architectures selon le modèle de KAZMAN 36
2.2 Etapes de l'extraction d'architectures selon le modèle de KOSCHKE 37
2.3 Etapes possibles de l'extraction d'architectures selon notre modèle 39
2.4 Autres étapes possibles de l'extraction d'architectures selon notre modèle 39
2.5 Critères de comparaison de l'axe propriétés conceptuelles 41
2.6 Critères de comparaison de l'axe des informations utilisées 45
2.7 Critères de comparaison de l'axe des propriétés techniques utilisées pour l'extraction 47

Partie II — La démarche ROMANTIC pour l'extraction d'architectures

3.1 Caractéristiques de ROMANTIC suivant l'axe des propriétés conceptuelles 58
3.2 Décomposition du processus d'extraction d'architectures 59
3.3 Correspondance entre les entités du code source et architecturales 60
3.4 Caractéristiques de ROMANTIC suivant l'axe des informations utilisées 60
3.5 Caractéristiques de ROMANTIC suivant l'axe des propriétés techniques 61
3.6 Un modèle du code source indépendant du langage 62
3.7 Le modèle FAMIX 2.0 64
3.8 Notre modèle du code source 65
3.9 Notre modèle de correspondance objet-composant (COA) 67
3.10 Relations entre les entités COA et architecturales 68

3.11 Relations entre les entités COA et du modèle du code source 68
3.12 Éléments des contours des composants 68
3.13 Éléments du contour de connecteur 70
3.14 Correspondance entre un ensemble de classes et les éléments architecturaux 72
3.15 Modèle de mise en correspondance (COA) et architecture à base de composants 73
3.16 Guides de l'extraction d'architectures 75
3.17 Les documents et les recommandations disponibles 77

4.1 Méta-modèle de mesure des caractéristiques d'un produit logiciel dans la norme ISO-9126 . 83
4.2 Méta-modèle de mesure de caractéristique dans notre approche 84
4.3 Les caractéristiques sémantiques des éléments architecturaux 88
4.4 Notre modèle de mesure de la validité sémantique de l'architecture : les sous-caractéristiques des composants 96
4.5 Notre modèle de mesure de la validité sémantique de l'architecture : les sous-caractéristiques des connecteurs 97
4.6 Les spécialisations de la méta-classe relation de composition verticale 98
4.7 Les neufs cas de cycles de vie 99
4.8 Modèle des caractéristiques d'un produit logiciel dans la norme ISO-9126 103
4.9 Notre modèle de mesure de la qualité architecturale 108
4.10 Graphe de changement de phases dans les systèmes physiques 109

5.1 Les documents et les recommandations disponibles 119
5.2 Informations intentionnelles disponibles dans un diagramme UML 120
5.3 Informations intentionnelles disponibles dans un fichier source 121
5.4 Le modèle de correspondance Objet/Architecture 124
5.5 Algorithme de regroupement hiérarchique 126
5.6 Algorithme de partitionnement du dendrogramme 127
5.7 Les étapes de la validation de l'extraction d'information par l'architecte 128
5.8 Exemples de méthodes Java 132
5.9 Le niveau de confiance selon les interventions de l'architecte 139
5.10 Utilisation de la documentation et des recommandations 140

Partie III — Mise en pratique et validation de ROMANTIC

6.1 Classification des algorithmes méta-heuristiques 144
6.2 Algorithme de regroupement hiérarchique 148
6.3 Algorithme d'identification des composants à partir du dendrogramme 148
6.4 Utilisation du croisement standard 170
6.5 Limites du croisement standard 172
6.6 Croisement préservant la complétude et la consistance 173

7.1 Architecture de Jigsaw selon la documentation 181
7.2 Architecture de Jigsaw selon Focus 183
7.3 Architecture de ArgoUML selon la documentation 183
7.4 Architecture probable de ArgoUML 185
7.5 Architecture de Jigsaw selon ROMANTIC 190

7.6 Vue partielle de l'architecture de ArgoUML selon ROMANTIC 190
7.7 Vue des composants du groupe 1 dans ArgoUML 191
7.8 Vue des composants du groupe 2 dans ArgoUML 192
7.9 Etude de la vitesse du recuit simulé sur Jigsaw 194
7.10 Etude de la vitesse de l'algorithme génétique sur Jigsaw 195
7.11 Etude du rapport vitesse/précision du recuit simulé sur Jigsaw 196
7.12 Etude du rapport vitesse/précision de l'algorithme génétique sur Jigsaw 197

Partie IV — Conclusion et perspectives

Liste des exemples

— Corps du document —

Partie I — Contexte et travaux connexes

Partie II — La démarche ROMANTIC pour l'extraction d'architectures

3.1 Correspondance entre un ensemble de classes et les éléments architecturaux 70

5.1 Informations intentionnelles disponibles dans un diagramme UML 120
5.2 Informations intentionnelles disponibles dans un fichier source 120
5.3 Informations intentionnelles disponibles à travers un outil de versionnement 121
5.4 Calcul de la similarité entre méthodes java 131

Partie III — Mise en pratique et validation de ROMANTIC

Partie IV — Conclusion et perspectives

Table des matières

Introduction **XI**

— Corps du document —

Partie I — Contexte et travaux connexes

1 Architectures logicielles **3**
1.1 Concepts et terminologie des architectures logicielles 4
1.1.1 Les composants 4
1.1.1.1 Structure externe d'un composant 5
1.1.1.2 Structure interne d'un composant 6
1.1.1.3 Relations de compositions des composants 7
1.1.2 Les connecteurs 7
1.1.2.1 Aspect Fonctionnel du connecteur 9
1.1.2.2 Structure externe du connecteur 10
1.1.2.3 Structure interne du connecteur 11
1.1.3 La configuration 12
1.1.3.1 Structure de la configuration 12
1.1.3.2 Les propriétés de la configuration 12
1.1.4 Le style architectural 13
1.1.4.1 Style et interprétation de l'architecture 13
1.1.4.2 Style et réalisation de l'architecture 14
1.1.5 Avantages de l'architecture 14
1.2 Les différentes facettes des architectures logicielles 15
1.2.1 Fluctuation des définitions de l'architecture 15
1.2.2 Définitions communautaires de l'architecture 17
1.3 Architectures logicielles et mesure de la qualité logicielle 19
1.3.1 Classification des approches de mesure de la qualité logicielle 19
1.3.2 Les objectifs de l'approche 19
1.3.3 Le modèle de qualité 22
1.3.4 Les entrées de l'approche 25
1.3.5 La technique de l'approche 27
1.3.6 Les sorties de l'approche 28
1.4 Conclusion 29

2 Extraction d'architectures logicielles **31**
2.1 Une architecture cruciale mais pas toujours disponible 31
2.1.1 Maintenance et architecture 32

2.1.2 Le manque de représentations architecturales fiables 33
2.1.2.1 Représentation architecturale indisponible 33
2.1.2.2 Représentation architecturale décalée 33
2.1.3 Conséquence de l'absence de représentation architecturale 34
2.2 Principes de l'extraction d'architectures 34
2.2.1 Définition de l'extraction d'architectures 34
2.2.2 Modèles conceptuels existants de l'extraction d'architectures 35
2.2.2.1 Modèle en fer à cheval de l'extraction d'architectures 35
2.2.2.2 Modèle conceptuel de l'extraction d'architectures 36
2.2.2.3 Limites des modèles existants 37
2.2.3 Notre modèle conceptuel de l'extraction d'architectures 38
2.3 Comparaison des approches d'extraction d'architectures 40
2.3.1 Cadre de comparaison des approches d'extraction d'architectures 40
2.3.2 Axe des propriétés conceptuelles 40
2.3.2.1 Niveau du code source 42
2.3.2.2 Niveau du modèle du code source 42
2.3.2.3 Niveau de mise en correspondance 43
2.3.2.4 Niveau de l'architecture 43
2.3.2.5 Enchainements des niveaux 44
2.3.3 Axe des informations utilisées 44
2.3.3.1 Intervention de l'utilisateur 44
2.3.3.2 Utilisation de la documentation 46
2.3.4 Axe des propriétés techniques 47
2.3.4.1 Critères de regroupement 48
2.3.4.2 Algorithmes 48
2.3.4.3 Degré d'automatisation 49
2.3.4.4 Les modes d'utilisation 50
2.4 Conclusion 50

Partie II — La démarche ROMANTIC pour l'extraction d'architectures

3 Problématique et approche de ROMANTIC 53
3.1 Motivation et problème 54
3.1.1 Extraction d'architectures 54
3.1.1.1 Propriétés conceptuelles 54
3.1.1.2 Informations utilisées 55
3.1.1.3 Propriétés techniques 55
3.1.2 Approches par exploration 56
3.2 Présentation de notre solution : ROMANTIC 56
3.2.1 Propriétés conceptuelles dans ROMANTIC 57
3.2.1.1 Niveau du code source et du modèle du code source 57
3.2.1.2 Niveau de l'architecture 57
3.2.1.3 Niveau de correspondance 59
3.2.2 Informations utilisées dans ROMANTIC 59
3.2.3 Propriétés techniques de ROMANTIC 61

3.3 Modélisation du problème 62
3.3.1 Notre modèle du code source 62
3.3.1.1 Le modèle FAMIX 63
3.3.1.2 Spécialisation de FAMIX pour l'extraction d'architectures 64
3.3.2 Notre modèle de mise en correspondance 67
3.3.3 Notre approche par exploration 71
3.3.3.1 L'espace des solutions 71
3.3.3.2 L'espace de recherche 73
3.3.3.3 L'exploration 74
3.4 Guides du processus d'extraction d'architectures 74
3.4.1 Guides pour l'identification de l'objectif 75
3.4.1.1 Sémantique architecturale 75
3.4.1.2 Qualité architecturale 76
3.4.2 Guides pour la réduction de l'espace de recherche 76
3.4.2.1 Documentation et recommandations 76
3.4.2.2 Contexte de déploiement 78
3.5 Conclusion 78

4 Fondement du processus d'extraction 81
4.1 Méta-modèle de mesure 82
4.1.1 Méta-modèle de mesure de la norme ISO-9126 82
4.1.2 Méta-modèle de mesure dans ROMANTIC 84
4.2 Étude de la sémantique architecturale 85
4.2.1 Caractéristiques sémantiques de l'architecture 85
4.2.1.1 Caractéristiques sémantiques des composants 85
4.2.1.2 Caractéristiques sémantiques des connecteurs 87
4.2.2 Des caractéristiques sémantiques aux propriétés architecturales 89
4.2.2.1 Les propriétés des composants 89
4.2.2.2 Les liens entre les caractéristiques et les propriétés des composants 90
4.2.2.3 Les propriétés des connecteurs 91
4.2.2.4 Les liens entre les caractéristiques et les propriétés des connecteurs 92
4.2.3 Des propriétés architecturales aux propriétés des entités COA 93
4.2.3.1 Liens entre les propriétés internes des éléments architecturaux et des entités COA 93
4.2.3.2 Liens entre les propriétés de l'enveloppe des éléments architecturaux et des entités COA 94
4.2.4 Des propriétés du modèle COA au modèle objet 95
4.2.4.1 Liens entre les interfaces verticales et les entités COA 95
4.2.4.2 Mesure de couplage 100
4.2.4.3 Mesure de cohésion 102
4.3 Etude de la qualité architecturale 102
4.3.1 Caractéristiques de qualité 102
4.3.1.1 Les caractéristiques de qualité de la norme ISO-9126 103
4.3.1.2 Les caractéristiques de qualité de notre modèle de mesure 105
4.3.2 Des caractéristiques de qualité aux propriétés architecturales 106
4.3.2.1 Maintenabilité de l'architecture 106
4.3.2.2 Fiabilité de l'architecture 106
4.3.3 Des propriétés architecturales aux propriétés des entités COA 107

4.3.3.1 Propriétés de la maintenabilité . . . 107
4.3.3.2 Propriétés de la fiabilité . . . 107
4.3.4 Des propriétés COA aux métriques objets . . . 107
4.3.4.1 Maintenabilité des modules objets . . . 107
4.3.4.2 Complexité des modules objets . . . 109
4.4 Définition de la fonction objectif . . . 110
4.4.1 Mesure de la validité sémantique de l'architecture . . . 110
4.4.1.1 Evaluation de la validité sémantique des composants . . . 110
4.4.1.2 Evaluation de la validité sémantique des connecteurs . . . 111
4.4.1.3 Evaluation de la validité sémantique de l'architecture . . . 112
4.4.2 Mesure de la qualité architecturale . . . 113
4.4.2.1 Mesure de la maintenabilité de l'architecture . . . 113
4.4.2.2 Mesure de la fiabilité de l'architecture . . . 113
4.4.2.3 Mesure de la qualité architecturale . . . 114
4.4.3 Fonction objectif du processus d'exploration . . . 114
4.5 Conclusion . . . 115

5 Guider le processus d'extraction 117
5.1 Les guides de l'extraction . . . 118
5.1.1 Les informations intentionnelles . . . 118
5.1.2 Les informations contextuelles . . . 122
5.1.3 Influence des guides sur l'exploration . . . 122
5.1.4 Modélisations des informations intentionnelles et contextuelles . . . 123
5.1.4.1 Modèle pour le ciblage . . . 123
5.1.4.2 Modèle pour la suppression . . . 124
5.2 Extraction d'architectures intentionnelles depuis la documentation . . . 125
5.2.1 Processus de partitionnement . . . 125
5.2.1.1 Étape de regroupement . . . 125
5.2.1.2 Étape de partitionnement . . . 126
5.2.1.3 Processus de validation des informations . . . 126
5.2.2 Mesure de similarité dans la documentation . . . 127
5.2.2.1 La similarité dans les diagrammes UML . . . 129
5.2.2.2 La similarité dans les fichiers sources . . . 130
5.2.2.3 La similarité dans les archives des outils de versionnement . . . 134
5.3 Création d'un réseau de contrainte hiérarchique . . . 135
5.3.1 Les recommandations de l'architecte . . . 135
5.3.1.1 Informations contextuelles . . . 135
5.3.1.2 Informations intentionnelles . . . 136
5.3.1.3 Collecte des recommandations . . . 136
5.3.2 Les contraintes issues des architectures intentionnelles . . . 137
5.3.3 Combinaison des contraintes . . . 138
5.4 Conclusion . . . 138

Partie III — Mise en pratique et validation de ROMANTIC

6 Algorithmes pour l'exploration 143
6.1 Algorithmes d'exploration pour ROMANTIC 143
6.2 Instance de ROMANTIC à base de regroupement hiérarchique 145
6.2.1 Extraction d'architectures simplifiées 146
6.2.1.1 Heuristiques pour le calcul de la fonction objectif 146
6.2.1.2 Algorithme de regroupement hiérarchique 147
6.2.1.3 Algorithme d'identification des composants à partir du dendrogramme 147
6.2.2 Extraction d'architectures logicielles 149
6.2.2.1 Identification des interfaces 149
6.2.2.2 Identification des connecteurs 151
6.2.2.3 Identification de la configuration 153
6.2.2.4 Identifications des composites 153
6.2.2.5 Utilisation des méthodes de réductions de l'espace de recherche 153
6.2.3 Analyse du processus par regroupement hiérarchique 153
6.2.3.1 Complexité des algorithmes 154
6.2.3.2 Avantages et inconvénients 154
6.3 Instance de ROMANTIC à base de recuit simulé 155
6.3.1 Algorithme d'extraction 156
6.3.1.1 Principes de l'algorithme de recuit simulé 156
6.3.1.2 Schéma de refroidissement 157
6.3.1.3 Réduction de l'espace de recherche 157
6.3.2 Définition du voisinage 158
6.3.2.1 Opérateurs de manipulation 158
6.3.2.2 Choix des opérations de manipulations 160
6.3.3 Point de départ de l'exploration 161
6.3.3.1 Utilisation des guides de l'approche 161
6.3.3.2 Utilisation d'heuristiques 161
6.3.4 Analyse du processus par recuit simulé 162
6.4 Instance de ROMANTIC à base d'algorithmes génétiques 163
6.4.1 Présentation des algorithmes génétiques 163
6.4.2 Codage de l'architecture 165
6.4.2.1 Chromosome des composants 166
6.4.2.2 Chromosome des connecteurs 166
6.4.2.3 Chromosome de la configuration 167
6.4.3 Définitions des opérateurs génétiques 168
6.4.3.1 Opérateurs de sélection 168
6.4.3.2 Opérateurs de croisement 169
6.4.3.3 Opérateurs de mutations 171
6.4.4 Choix de la population initiale 175
6.4.4.1 Individus basés sur les guides de l'extraction 175
6.4.4.2 Individus basés sur les autres approches par exploration 176
6.4.5 Analyse du processus par algorithme génétique 176
6.5 Conclusion 177

7 Validation et évaluation de l'approche ROMANTIC **179**
7.1 Cadre du cas d'étude . . . 180
7.1.1 Instances de ROMANTIC . . . 180
7.1.1.1 Paramétrage des algorithmes . . . 180
7.1.1.2 Utilisation des guides . . . 180
7.1.2 Systèmes étudiés . . . 181
7.1.2.1 Jigsaw . . . 181
7.1.2.2 ArgoUML . . . 182
7.2 Validation des fondements théoriques . . . 184
7.2.1 Méthode de validation des fondements théoriques . . . 184
7.2.1.1 Consistance de ROMANTIC . . . 185
7.2.1.2 Adéquation de ROMANTIC . . . 187
7.2.2 Analyse des résultats . . . 187
7.2.2.1 Étude de la consistance de notre approche . . . 187
7.2.2.2 Étude de l'adéquation de notre approche . . . 189
7.2.3 Conclusion sur la validité de ROMANTIC . . . 191
7.3 Comparaison des instances de ROMANTIC . . . 192
7.3.1 Méthode de comparaison des algorithmes . . . 193
7.3.1.1 Efficacité d'un algorithme . . . 193
7.3.1.2 Comparaison des algorithmes . . . 193
7.3.2 Comparaison des algorithmes . . . 194
7.3.2.1 Étude de la vitesse des algorithmes . . . 194
7.3.2.2 Étude du rapport entre vitesse et précision des algorithmes . . . 196
7.3.2.3 Conclusion sur le choix des algorithmes . . . 197
7.4 Conclusion . . . 198

Partie IV — Conclusion et perspectives

Conclusion **201**

Perspectives **205**

— Pages annexées —

Bibliographie **209**

Liste des tableaux **223**

Liste des figures **225**

Liste des exemples **229**

Table des matières **231**

Extraction d'une architecture logicielle à base de composants depuis un système orienté objet

Une approche par exploration

Sylvain CHARDIGNY

Résumé

La modélisation et la représentation des architectures logicielles sont devenues une des phases principales du processus de développement des systèmes complexes. En effet, la représentation de l'architecture fournit de nombreux avantages pendant tout le cycle de vie du logiciel. Cependant, pour beaucoup de systèmes existants, aucune représentation fiable de leurs architectures n'est disponible. Afin de pallier cette absence, source de nombreuses difficultés principalement lors des phases de maintenance et d'évolution, nous proposons dans cette thèse une approche, appelée ROMANTIC, visant à extraire une architecture à base de composants à partir d'un système orienté objet existant. L'idée première de cette approche est de proposer un processus quasi-automatique d'identification d'architectures en formulant le problème comme un problème d'optimisation et en le résolvant au moyen de méta-heuristiques. Ces dernières explorent l'espace composé des architectures pouvant être abstraites du système en utilisant la sémantique et la qualité architecturale pour sélectionner les meilleures solutions. Le processus s'appuie également sur l'architecture intentionnelle du système, à travers l'utilisation de la documentation et des recommandations de l'architecte.

Mots-clés : architecture logicielle, composant logiciel, connecteur, extraction, sémantique architecturale, qualité, architecture intentionnelle, méta-heuristiques.

Component-based software architecture recovery from an object oriented system

A search-based approach

Abstract

Software architecture modeling and representation are a main phase of the development process of complex systems. In fact, software architecture representation provides many advantages during all phases of software life cycle. Nevertheless, for many systems, like legacy or eroded ones, there is no available representation of their architectures. In order to benefit from this representation, we propose, in this thesis, an approach called ROMANTIC which focuses on recovering a component-based architecture from an existing object-oriented system. This recover is a balancing problem of competing constraints which aims at obtaining the best architecture that can be abstracted from a system. Consequently, the main idea of this approach is to propose a quasi-automatic process of architecture identification by formulating it as a search-based problem. The latter acts on the space composed of all possible architectures abstracting the object-oriented system and use the architectural semantic and quality to choose the best solution. The process uses the intentional system architecture by means of the documentation and the architect's recommendations.

Keywords: software architecture, software component, connector, recovery, architectural semantic, quality, intentional architecture, search-based software engineering.

Discipline : INFORMATIQUE
Spécialité : INFORMATIQUE

Laboratoire : LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.