UNIVERSITE ABDELMALEK ESSAADI
FACULTE DES SCIENCES et TECHNIQUES
TANGER

THESE

Présentée à la Faculté des Sciences et Techniques de Tanger
Pour obtenir le titre de

DOCTEUR EN SCIENCES ET TECHNIQUES DE L'INGENIEUR

Discipline : Informatique

Présentée et soutenue publiquement par

Mounir ZEKKAOUI

Le 12 Février 2015

Une Approche de Construction et de Gestion de l'Incohérence des Artefacts Logiciels Hétérogènes pour le Contrôle de l'Evolution Logiciel

Devant le jury

Président	Pr. Benaissa AMAMI	Faculté des Sciences et Technique (Tanger)
Rapporteurs	Pr. El Mokhtar EN-NAIMI	Faculté des Sciences et Technique (Tanger)
	Pr. Abdelhak EZZINE	Ecole Nationale des Sciences Appliquées (Tanger)
	Pr. Noureddine CHENFOUR	Faculté des Sciences Dhar Mahraz (Fès)
Examinateur	Pr. Mohammed Bouhorma	Faculté des Sciences et Technique (Tanger)
Directeur de thèse	Pr. Abdelhadi FENNAN	Faculté des Sciences et Technique (Tanger)

Centre des Etudes Doctorales : « Sciences et Techniques de l'Ingénieur »
Laboratoire d'Informatique et Systèmes de Télécommunication de la FST de Tanger

« Le savoir scientifique n'est pas absolu, mais socialement, culturellement, technologiquement et historiquement marqué, donc provisoire. » ***Steven Rose***

Remerciement

Les travaux présentés dans ce manuscrit ont été effectués au sein du Laboratoire d'Informatique et Systèmes de Télécommunication (LIST) de la Faculté des Sciences et Techniques (FST) de Tanger de l'Université Abdelmalek Essaadi. Je souhaite adresser mes remerciements à toutes les personnes qui ont contribué de près ou de loin à l'élaboration de ces travaux.

En tout premier lieu, je remercie le bon Dieu, tout puissant, de m'avoir donné la force et l'audace pour dépasser toutes les difficultés.

Ma plus grande gratitude va à mon directeur de cette thèse, Pr. Abdelhadi FENNAN, pour sa grande disponibilité, ses grandes qualités scientifiques et la confiance qu'il m'a accordée. J'aimerais aussi le remercier pour l'autonomie qu'elle m'a accordée, et ses précieux conseils qui m'ont permis de bien mener ce travail.

Au terme de ce travail, je remercie les membres du jury de ma thèse pour l'intérêt qu'ils ont porté à mon travail et pour leur disponibilité : Monsieur Benaissa AMAMI d'avoir accepté de présider mon jury. Monsieur El Mokhtar EN-NAIMI, Monsieur Abdelhak EZZINE et Monsieur Noureddine CHENFOUR, d'avoir accepté d'être les rapporteurs de cette thèse. Je leur suis redevable pour le temps accordé à la lecture de ce manuscrit et pour toutes leurs remarques et conseils, qui étaient tous très constructifs, et qui m'ont beaucoup aidé à améliorer ce travail. Je remercie monsieur Mohammed Bouhorma d'avoir accepté d'examiner mon travail et de me faire ses précieuses observations.

Enfin, je remercie mes collègues de l'équipe (LIST), et tout particulièrement Jaber EL BOUHDIDI et Mohamed EL GHAILANI, qui, en plus d'être toujours disponibles, m'ont constamment prodigué d'excellents conseils.

Bien sûr, je ne peux terminer sans remercier mes proches de tout mon cœur et notamment mes parents, ma femme, ma petite fille Lina, ainsi que mes beaux-parents qui, au cours de ces années de thèse, m'ont toujours soutenu et encouragé, comme d'habitude.

Afin de n'oublier personne, mes vifs remerciements s'adressent à tous ceux qui m'ont aidé à la réalisation de ce modeste mémoire.

Résumé

Le contrôle de l'évolution des systèmes logiciels exige une description détaillée des différents changements apportés sur les artefacts logiciels hétérogènes. Une compréhension des connaissances descriptives des artefacts est une condition indispensable pour la réussite du processus d'évolution.

Un artefact est un terme général désignant toute sorte d'information créée, produite, modifiée ou utilisée par les développeurs dans la mise en place des systèmes logiciels. Le développement de systèmes logiciels complexes implique l'intervention de plusieurs développeurs qui collaborent à l'édition de nombreux artefacts logiciels hétérogènes, tel que le code source, les modèles d'analyse et de conception, tests unitaires, les descripteurs de déploiement XML, les guides utilisateur, etc. La nature même de ces applications fait en sorte que ces artefacts sont répartis sur plusieurs sites de conception et donc stockés dans plusieurs gestionnaires de version. Dans ce contexte, il devient de plus en plus difficile d'assurer la cohérence entre tous ces artefacts, ainsi gérer l'impact de leur évolution tout au long du processus de développement.

La mise en œuvre d'un processus d'évolution implique des changements importants de nombreux artefacts logiciels hétérogènes, ces changements peuvent dégrader la qualité et le fonctionnement normal du logiciel modifié. D'où la nécessité d'une approche unifiée de représentation des artefacts ainsi qu'un formalisme uniforme pour la gestion de la cohérence des artefacts logiciels hétérogènes.

Notre première contribution consiste, en premier lieu, de mettre en place une approche unifiée d'extraction et de représentation des différents artefacts hétérogènes dans le but d'assurer une description unifiée et détaillée des artefacts logiciels hétérogènes, exploitable par plusieurs outils informatiques et permettant aux chargés de l'évolution de mener à bien le raisonnement adopté au changement concerné.

Une deuxième contribution consiste à proposer un formalisme uniforme de spécification des règles de cohérence méthodologiques basé sur les traces de construction (résultats de la première contribution) et nous avons validé notre approche en construisant un système (moteur de vérification) de suivi d'impact de leur évolution (détecte les violations de contraintes méthodologiques).

L'ensemble des contributions est accompagné par la réalisation d'un prototype pour la validation des travaux réalisés, permettant, entre autres, d'assister les développeurs et les chargés de

l'évolution des logiciels de mener à bien le contrôle de leur évolution.

***Mots clés* :** Evolution du Logiciel, Artefacts logiciels hétérogènes, Contrôle de l'évolution du Logiciel, approche unifié, Formalisme uniforme, méta-modèle, cohérence des artefacts.

Abstract

The evolution control of software systems requires a detailed description of the various changes on heterogeneous software artifacts. An understanding of descriptive knowledge artifacts is a prerequisite for the success of the evolutionary process.

An artifact is a general term for any kind of information created, produced, modified or used by developers in the development of software systems. The development of complex software systems requires the intervention of several developers who collaborate in publishing numerous heterogeneous software artifacts such as source code, analysis and design models, unit testing, XML deployment descriptors, user guides. The nature of these applications ensures that these artifacts are spread over several design sites and therefore stored in multiple version management. In this context, it is becoming increasingly difficult to ensure consistency between all these artifacts and manage the impact of their evolution throughout the development process.

The implementation of an evolutionary process returns to make significant changes many heterogeneous software artifacts, such changes may affect the quality and the normal functioning of the modified software. Hence the need for a unified approach to the representation of artifacts and a uniform formalism for the consistency management of heterogeneous software artifacts.

Our first contribution consists, first, to establish a unified approach to extraction and representation of different heterogeneous artifacts in order to ensure a unified and detailed description of heterogeneous software artifacts, exploitable by many IT tools and enabling responsible for the evolution of carrying out the reasoning of the change in question.

A second contribution is to provide a uniform formalism specification methodological consistency rules based on traces of construction (results of the first contribution) and we validated our approach by building a system (check engine) of impact monitoring evolution (detects methodological constraints violations).

All contributions is accompanied by the production of a prototype for the validation of the work, allowing, among others, to assist developers and responsible for the development of software to complete control of their evolution.

Keywords: Software evolution, heterogeneous software artifacts, Software evolution control, unified approach, Uniform formalism, meta-model, artefacts consistency.

Table des matières

Remerciement 4
Résumé 5
Abstract 7
Table des matières 8
Liste des tableaux 11
Liste des figures 12
Liste des acronymes 13
Publications et communications 14
CHAPITRE 1 : INTRODUCTION GENERALE 15
1.1 Contexte : Evolution et maintenance des logiciels 17
1.2 Problématique : L'incohérence des artefacts logiciels hétérogènes 18
1.3 Objectifs et contributions 21
1.4 Organisation du rapport 23
CHAPITRE 2 : ETAT DE L'ART ET TRAVAUX ANTERIEURS 25
2.1 Introduction 27
2.2 Evolution logicielle 28
2.2.1 Le processus de développement logiciel 28
2.2.1.1 Les activités du processus de développement 29
2.2.1.2 Modèles et méthodes de cycle de développement 34
2.2.2 La problématique de l'évolution logicielle 35
2.2.2.1 Le cycle de vie d'un logiciel 36
2.2.2.2 Les lois de l'évolution logicielle 37
2.2.2.3 L'érosion du design 38
2.2.2.4 La maintenance et l'évolution logicielle 39
2.2.3 Travaux antérieurs 40
2.2.4 Relation avec la thèse 42
2.3 La gestion de l'incohérence 42
2.3.1 Définitions de base et processus 43
2.3.1.1 Principales définitions 43
2.3.1.2 Les processus de gestion de l'incohérence 45
2.3.2 Détection des chevauchements 48
2.3.2.1 Conventions de représentation 49

2.3.2.2 Ontologies partagées 50
2.3.2.3 Inspection humain 51
2.3.2.4 L'analyse de similarité 53
2.3.2.5 Résumé 54
2.3.3 Détection de l'incohérence 55
2.3.3.1 La détection basée sur la logique 55
2.3.3.2 La détection basée sur le contrôle de modèle 56
2.3.3.3 La détection des incohérences basées sur des formes spécialisées de l'analyse automatisée 56
2.3.3.4 La détection des incohérences basée sur l'exploration collaborative centrée sur l'humain... 57
2.3.3.5 Résumé 58
2.3.4 Diagnostic d'incohérences 59
2.3.5 Traitement des incohérences 61
2.3.5.1 Actions de changement 62
2.3.5.2 Actions de non-changement 62
2.3.6 Localisation 63
2.3.7 Spécification et l'application des politiques de gestion d'incohérence 64
2.4 Conclusion 67
CHAPITRE 3 : APPROCHE DE CONSTRUCTION ET DE GESTION DE COHERENCE 69
3.1 Introduction 71
3.2 Travaux connexes 74
3.3 Vue d'ensemble de l'approche 76
3.4 La construction des artefacts logiciels hétérogènes 78
3.4.1 Les artefacts logiciels 78
2.3.1.1 La définition des artefacts logiciels 78
4.3.1.1 La classification des artefacts logiciels 80
3.4.2 La surveillance des changements 82
3.4.3 Le Méta-model de représentation unifié des artefacts 82
3.4.4 La gestion des taches d'extraction 84
3.5 La gestion de la cohérence des artefacts logiciels 89
3.5.1 La cohérence des artefacts 90
2.5.1.1 L'incohérence : définition de base 90
2.5.1.2 Les causes de l'incohérence 91
2.5.1.3 La détection et l'identification de l'incohérence 93
2.5.1.4 La manipulation de l'incohérence 94

3.5.2 Gestionnaire des règles de cohérence ... 94

3.5.2.1 La classification de la règle de cohérence ... 95

3.5.2.2 La définition des règles de cohérence ... 96

3.5.3 Moteur de validation ... 101

3.6 Conclusion ... 101

CHAPITRE 4 : PROTOTYPE DE VALIDATION ... 103

4.1 Introduction ... 105

4.2 Architecture globale du prototype de validation ... 106

4.3 Le module « artifactsBuilder » ... 109

4.3.1 Liste des taches d'extraction ... 110

4.3.2 Nouvelle tâche d'extraction ... 111

4.4 Le module « activityMonitor » ... 112

4.5 Le module « dataRepopsitoryManager » ... 113

4.6 Le module « consistencyRulesBuilder » ... 114

4.6.1 Liste des règles de cohérence ... 114

4.6.2 Nouvelle règle de cohérence ... 115

4.7 Le module « checkEngine » ... 117

4.8 Conclusion ... 118

CHAPITRE 5 : CONCLUSION GENERALE ET PERSPECTIVES ... 119

5.1 Bilan des contributions ... 121

5.1.1 Contribution à la construction des artefacts logiciels ... 122

5.1.2 Contribution à la gestion de la cohérence des artefacts logiciels ... 122

5.2 Perspectives des travaux ... 123

Bibliographie ... 124

Liste des tableaux

Tableau 1 : Classification des activités du génie logiciel [46] 30
Tableau 2 : Résultats de la résolution du projet de recherche CHAOS - 2004-2012 [80]..................... 35
Tableau 3 : Les lois de l'évolution logicielle 37
Tableau 4 : Exemples informelles des règles de cohérence 44
Tableau 5 : Hypothèses, avantages et limitations des approches de la détection des chevauchements 54
Tableau 6 : Hypothèses, avantages et limitations des approches de la détection des incohérences...... 58
Tableau 7 : Hypothèses, avantages et limitations des techniques de diagnostic des incohérences 61
Tableau 8 : Hypothèses, avantages et limitations des techniques de manipulation des incohérences .. 63
Tableau 9 : Hypothèses, avantages et limitations des techniques de localisation 64
Tableau 10 : Hypothèses, avantages et limitations des techniques du processus de spécification et d'application 66
Tableau 11 : Artefacts logiciels communs 79
Tableau 12 : Les niveaux abstro-granulaire de l'exemple 89
Tableau 13 : Les artefacts extraits de l'exemple.. 89

Liste des figures

Figure 1 : Schéma global de contribution 22
Figure 2 : The versioned staged model [44] 29
Figure 3 : Forces de changement et l'évolution des logiciels 38
Figure 4 : Le processus de maintenance 39
Figure 5 : Architecture fonctionnelle de l'approche 76
Figure 6 : Relation entre les artefacts logiciels 79
Figure 7 : Classification des Artefacts logiciels par type de fichier 81
Figure 8 : U2MHA : Méta-model unifié des artefacts hétérogènes 83
Figure 9 : Processus de création des taches d'extraction 85
Figure 10 : Exemple du code java d'une nouvelle classe d'extraction 86
Figure 11 : EJ2M : Méta-model des taches d'extraction 87
Figure 12 : Exemple du fichier java "Calculator.java" 88
Figure 13 : Exemple u fichier XML "app-context.xml" 88
Figure 14 : Exemple du fichier properties "build.properties" 88
Figure 15 : U2MACR - Méta-Modèle des artefacts et des règles de cohérences 98
Figure 16 : Architecture modulaire globale de CMAC 108
Figure 17 : Vue Eclipse de tous les modules de CMAC 109
Figure 18 : Ecran de gestion des tâches d'extraction 111
Figure 19 : Ecran de création de la tâche d'extraction 112
Figure 20 : Ecran de gestion des règles decohérence 115
Figure 21 : Ecran du premier scénario de création de la règle de cohérence 116
Figure 22 : Ecran du deuxième scénario de création de la règle de cohérence 117

Liste des acronymes

UML	Unified Modeling Language
XML	Extensible Markup Language
HTML	Hypertext Markup Language
ETGM	Error Tolerant Graph Matching
IEEE	Institute of Electrical and Electronics Engineers
SLC	Software Life Cycle
DFD	Data Flow Diagram
SRS	Systems Requirement Specification
PRB	Project Review Board
QARCC	Quality Attribute Risk and Conflict Consultant
CMAC	Consistency Management based on Artefact Construction
SDD	Software Design Description
IDE	Integrated Development Environment
JPA	Java Persistence API

Publications et communications

1. M. ZEKKAOUI et A. FENNAN, *"UNIFIED APPROACH FOR BUILDING HETEROGENEOUS ARTIFACTS AND CONSISTENCY RULES", In the International Conference on Intelligent Information and Network Technology, 2013, Settat-Morocco*.

2. M. ZEKKAOUI et A. FENNAN, *«HETEROGENEOUS ARTIFACTS CONSTRUCTION FOR SOFTWARE EVOLUTION CONTROL", In the ICSCE 2014: XII International Conference on Software and Computer Engineering, waset, 2014, Paris-France*.

3. M. ZEKKAOUI et A. FENNAN, *" UNIFIED APPROACH FOR BUILDING HETEROGENEOUS ARTIFACTS AND CONSISTENCY RULES ", In the Journal of Emerging Technologies in Web Intelligence, Vol 6, No 1, 26-31, 2014*.

4. M. ZEKKAOUI et A. FENNAN, *"SEMI-AUTOMATED CONSTRUCTION MECHANISM OF HETEROGENEOUS ARTIFACTS", In the International Journal of Computer Technology and Applications, Vol 5, No 5, 1696-1702, 2014*.

CHAPITRE 1

Introduction générale

Objectif

Nous décrivons brièvement, dans ce chapitre, le contexte et les problèmes qui ont motivé cette recherche, les objectifs de notre recherche, ainsi que les solutions que nous avons proposées. Nous terminons en donnant un aperçu des différents chapitres du rapport.

Sommaire

1.1 Contexte : Evolution et maintenance des logiciels 17
1.2 Problématique : L'incohérence des artefacts logiciels hétérogènes 18
1.3 Objectifs et contributions 21
1.4 Organisation du rapport 23

1.1 Contexte : Evolution et maintenance des logiciels

Dans toutes les applications et les systèmes informatisés associés à tous les domaines d'activités, l'évolution des logiciels est le processus qui assure l'innovation et l'amélioration continue des techniques, des méthodes et des outils de traitement de l'information. Ceci explique que le coût consacré à l'ensemble des activités informatiques est le plus important. **L'objectif principal du génie logiciel est d'optimiser l'effort et le coût élevé du développement des logiciels**.

L'optimisation et la réussite du processus de l'évolution nécessitent une compréhension descriptive et une représentation unifiée de tous les artefacts logiciels développés, permettant de mieux contrôler toutes les activités de l'évolution logicielle. Cette connaissance inclut en premier lieu la description détaillée de l'artefact ainsi que sa structure de dépendance avec le reste du programme. L'architecture, la structure descriptive des artefacts logiciels et les graphes construits des différentes relations (l'appel, l'importation, l'héritage, l'implémentation, la communication, etc.) entre artefacts, font partie de l'ensemble des informations à rendre accessibles aux développeurs et aux différents outils informatique pour mener à bien le contrôle de l'évolution des logiciels tout au long du processus de développement.

Les efforts de la recherche consacrés aux problèmes de l'évolution du logiciel montre l'importance de ce domaine. Différentes approches de valeur ont été proposées pour le but d'analyser les aspects de l'évolution du logiciel, comme l'identification de l'impact de changement de l'évolution du logiciel [1], [2], [3], comme la prédiction [4], [5], ou comme l'ingénierie inverse [6], [7], [8], [9], [10], [11]. Cependant, si l'on veut comprendre l'évolution du logiciel dans son ensemble, nous avons besoin des moyens de combiner et comparer les résultats des différentes analyses.

Les systèmes industriels actuels s'intéressent de plus en plus au développement des applications et systèmes complexes, et exigent en même temps la qualité et la performance afin d'assurer la maintenabilité et la réutilisabilité. Dans les systèmes industriels, le coût de la maintenance est estimé entre 50% et 75% du coût total [12]. Corbie estime que la compréhension d'un logiciel prends plus de la moitié de sa maintenance [13]. **La maitrise de la compréhension du logiciel et l'acquisition des connaissances sur le système s'avèrent donc indispensable**.

Un logiciel doit constamment évoluer dans le temps pour garder l'utilité et la continuité d'exploitation. Ceci est principalement dû aux changements continus des besoins utilisateurs et l'accroissement de leurs exigences. Faire évoluer un logiciel reste toujours un réel défi. L'expérience pratique des grands experts au monde en matière de maintenance de logiciels a confirmé que les développeurs font toujours face à des problèmes et difficultés dans la phase de maintenance [14]. D'autre part,

depuis l'arrivée des logiciels à code source libre (open-source), le développement peut être réalisé par l'intervention des milliers de développeurs répartis dans le monde entier, ce qui accroit les difficultés et les problèmes de la maintenance. Du coup, la communauté du génie logiciel face à des complexités et à des grands défis :

- Saisir la complexité d'un logiciel
- Repérer toutes les composantes du logiciel concernées par un tel changement
- Effectuer les changements sans avoir de bogues ou impacter les choix conceptuels
- Prévoir de nouvelles méthodes et outils informatiques pour aider à l'évaluation et la compréhension des activités de maintenance.

1.2 Problématique : L'incohérence des artefacts logiciels hétérogènes

1. L'évolution du logiciel est un processus de changement de logiciels pour faire face à l'évolution des systèmes ou aux besoins des utilisateurs. Le problème majeur dans le développement de logiciels d'aujourd'hui se produit lorsque les différents artefacts d'un système logiciel évoluent à des rythmes différents. Le code source sera mis à jour afin d'inclure tous les changements nécessaires, mais les spécifications et les documents de conception ne sont souvent pas modifiés pour refléter ces changements. Les cas de test peuvent être approfondies pour le système initial, mais, en l'absence d'une méthodologie de développement appropriée, tend à se négliger avec l'ajout de nouvelles fonctionnalités. Tout utilisateur et développeur est familiarisé avec la manière dont la documentation devient dépassée, et comment les implémentations des changements prennent beaucoup de temps à percoler à la documentation. Le résultat est que les développeurs apprennent à ne pas faire confiance et donc de ne pas utiliser autre chose que le code source tout, ce qui rend le logiciel moins fiable et beaucoup plus difficile à comprendre et à évoluer.

2. Le développement des systèmes logiciels complexes implique la création de nombreux artefacts

hétérogènes, tels que le code source, les modèles de conception UML[1], tests unitaires, les descripteurs de déploiement XML[2] ou les documents de spécification fonctionnelles et métiers, parmi beaucoup d'autres. Étant donné que ces objets sont créés et modifiés par les grandes équipes, ils sont répartis sur différents sites physiques. Ces changements peuvent dégrader la qualité, le comportement et le fonctionnement normal du logiciel modifié, dans ce contexte, il devient de plus en plus difficile d'assurer la cohérence des artefacts logiciels et gérer l'impact de leur modification tout au long du processus de développement.

3. Il est important de souligner le fait que les approches traditionnelles actuelles ne permettent en général que de travailler sur des artefacts homogènes (documents UML, ou bien le code source par exemple) [15], [16], [17], ou bien ils masquent l'hétérogénéité en utilisant des formats pivots (comme XML) [18], [19]. En outre, ils ne peuvent généralement pas faire face à l'évolution des différents artefacts logiciels hétérogènes de l'application développée.

4. Dans la compréhension de l'évolution du logiciel, les travaux de recherches actuels incluent l'utilisation de l'historique des systèmes de logiciels pour analyser leur situation actuelle et prévoir les changements futurs [20] complétée par des approches de refactoring, ou reverse engineering existantes [21], [22], [23], [24] , [25]. Ces approches extraient les données soit en parcourant la totalité des artefacts avec la mise en place des listners en écoute sur les opérations de construction des différents développeurs dans le but de mettre à jour le référentiel des artefacts [26], ou bien à travers la conversion de tous les artefacts en XML avant d'appliquer les règles de cohérences [18]. Ceci nécessite la manipulation et le stockage d'un grand ensemble de données de description de logiciel. Les approches traditionnellement adoptées pour ces études empiriques nécessitent la collecte de données coûteuses (souvent manuel) et l'analyse statistique complexe.

5. Dans le but d'étudier l'évolution des logiciels, la plupart des outils informatiques développés dans le marché sont basés sur des comparateurs comme diff [27], qui permet de comparer et montrer la différence entre le contenu de deux fichiers en termes de lignes modifiées, ajoutées ou supprimées. Bien que fort utiles dans certains cas, l'utilisation de ces outils n'est pas adaptée

1 http://www.uml.org
2 http://www.w3.org/XML

aux programmes de grandes tailles pour les raisons suivantes :

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

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

6. De nombreux travaux de recherches actuels [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38] s'intéressent à l'extraction de motifs récurrents depuis l'historique d'un système logiciel pour offrir aux chargés de l'évolution une vision détaillée et simplifiée du logiciel dans le temps. Ces motifs ont pour but de faciliter la compréhension et de diminuer les bugs lors de l'évolution des logiciels. Par exemple, dans les travaux de recherches existantes, Zimmermann [39] a défini le patron de changement par l'ensemble d'éléments (classes et/ou méthodes) d'un logiciel qui ont changé ensemble au cours de leur évolution. Ce type de patron permet aux chargés de l'évolution de repérer l'incohérence des changements appliqués à une version. Dans un autre travail, Yann-Gaël et al. [40] ont étendu le travail de Zimmermann en introduisant le concept de patrons de changements :

   - Un ensemble de modifications du programme qui occurrent régulièrement pendant l'évolution du programme
   - En utilisant la technique de Dynamic TimeWarping [41] pour améliorer grandement la précision et le rappel de la détection des motifs d'évolution.

Dernièrement, Kopjedo et al. [32] ont proposé d'identifier l'ensemble des classes qui ne changent pas dans le temps d'un logiciel développé, en utilisant un algorithme d'appariement de graphes approchés, basé sur des méta-heuristiques (ETGM).

Tandis que, en application sur de grands systèmes logiciels toutes les approches actuelles, sont limitées au niveau précision de leurs résultats et performances en mémoire et en temps. Ainsi, les deux principales limites sont :

- Le manque de patrons (d'évolution et de changements) connus décrivant des situations récurrentes de changement dans l'évolution d'un programme, et facilitant la compréhension de cette évolution ;
- Pour comprendre l'évolution de programmes de grande taille.

1.3 Objectifs et contributions

Au fil des rencontres avec les experts en évolution et maintenance des logiciels, un certain nombre d'objectifs a été fixé afin de répondre aux différents problèmes exposés auparavant. L'approche que nous présentons tente de satisfaire ces objectifs.

Cette recherche vise l'évolution des artefacts logiciels hétérogènes et répartis dans le cadre des systèmes logiciels complexes. Il est devenu de plus en plus difficile d'assurer la cohérence entre tous les artefacts dans les applications logiciels complexes, ainsi gérer l'impact de leur évolution tout au long du processus de développement. L'assistance informatique dans la détection et la résolution des problèmes d'incohérence peut aider à améliorer la qualité de conceptions et de développement logicielles sophistiqués.

Nous avons donc analysé les systèmes existants qui permettent de définir et de vérifier des règles sur des artefacts logiciels. Il est important de souligner le fait que ces approches ne permettent en général que de travailler sur des artefacts homogènes (sur le code source par exemple ou bien sur les documents de modélisation UML). Et pour faire face à ce problème d'hétérogénéité, nous avons proposé une approche unifiée d'extraction et de stockage de tous les artefacts logiciels hétérogènes dans le but d'assurer une description unifié et détaillées de ces artefacts, exploitable en premier par notre mécanisme de gestion de cohérence (exposé auparavant) et par plusieurs outils informatiques

dans le cadre de contrôler l'évolution des logiciels afin de réduire les coûts de développement et de maintenance [42], [43].

Dans le cas de notre approche, et pour des raisons de performance, nous permettons aux utilisateurs (qui sont souvent les chargés de l'évolution) de spécifier les niveaux d'abstraction des artefacts à extraire (exemple : Classes, Méthodes, attributs, beans du descripteur de déploiement) au lieu de parcourir la totalité des artefacts logiciels et impacter la performance de l'environnement de développement [42], [43].

Dans le cadre du contrôle de l'évolution des logiciels, nous avons proposé un formalisme uniforme permettant aux développeurs (qui sont souvent les chargés de l'évolution) d'ajouter, d'exprimer et de personnaliser les règles de cohérence sur l'ensemble des artefacts hétérogènes de l'application développée, ainsi qu'une approche efficace d'évaluation et de validation de ces règles [44], [45].

L'objectif principal est de permettre aux développeurs d'assurer la cohérence des artefacts logiciels hétérogènes tout au long du processus de développement, et la figure 1 représente un schéma global de notre contribution. Il est demandé aussi d'intégrer la problématique de version et de l'historique d'évaluation dans notre plateforme. C'est pour cette raison qu'un focus particulier est mis sur la validation de l'approche proposée, notamment en l'intégrant dans plusieurs environnements de développement (Eclipse[3], Jenkins[4], application web[5]) et en analysant son utilisation dans des projets réels. Il sera aussi primordial d'assurer le passage à l'échelle sur des applications très volumineuses.

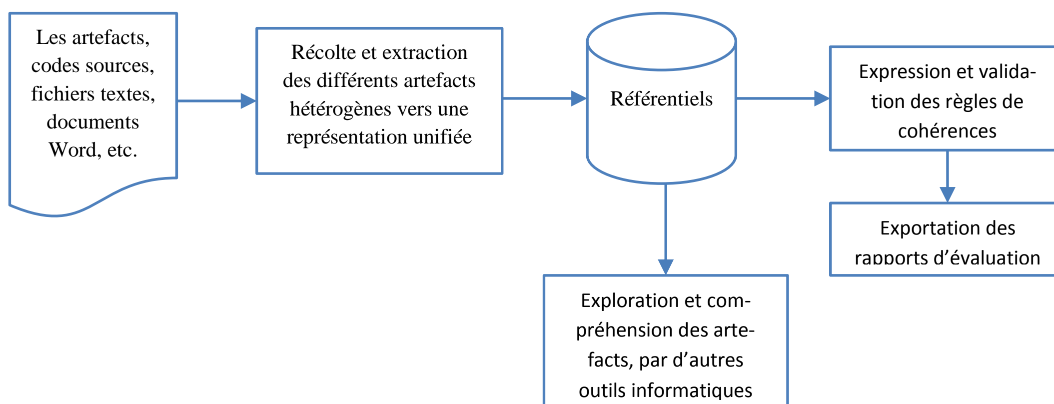

Figure 1 : Schéma global de contribution

3 https://www.eclipse.org
4 http://jenkins-ci.org
5 http://fr.wikipedia.org/wiki/Application_web

L'ensemble des contributions est accompagné par la réalisation d'un prototype pour la validation des travaux réalisés.

1.4 Organisation du rapport

L'objectif de cette manuscrite est de présenter les travaux de recherches effectués durant les trois ans de recherches. Le mémoire est structuré en cinq chapitres :

Chapitre 1 : Dans ce chapitre, nous décrivons brièvement le contexte et les problèmes qui ont motivé cette recherche, les objectifs de notre recherche, ainsi que les solutions que nous avons proposées pour faire face aux différentes questions exposées.

Chapitre 2 : Nous nous intéressons, dans ce chapitre aux différents travaux préalables dans les domaines reliés à notre recherche. Ce chapitre est articulé en deux parties. La première partie est dédiée à l'évolution logicielle, tandis que la seconde est dédiée aux techniques et méthodes proposées dans la gestion de l'incohérence.

Chapitre 3 : Nous présentons, dans ce chapitre, notre approche que nous avons proposée pour faire face aux différentes problématiques exposées dans cette recherche, et nous décrivons en détail, comment nous avons attaqué les problèmes, quelles méthodes et instruments nous avons employé et comment nous avons procédé aux résultats obtenus.

Chapitre 4 : Dans ce chapitre, nous présentons le détail de développement de notre plateforme appelée CMAC « Consistency Management based on Artefacts Construction » en détaillant les différents modules qui constituent son architecture.

Chapitre 5 : La conclusion générale de nos contributions et les perspectives des travaux de recherche qui ouvrent d'autres voies de recherche comme extension de la solution que nous proposons.

CHAPITRE 2

Etat de l’art et travaux antérieurs

Objectif

Nous nous intéressons dans ce chapitre aux différents travaux préalables dans les domaines reliés à notre recherche. Ce chapitre est articulé en deux parties. La première partie est dédiée à l'évolution logicielle, tandis que la seconde est dédiée aux techniques et méthodes proposées dans la gestion de l’incohérence.

Sommaire

2.1 Introduction 27

2.2 Evolution logicielle 28

2.2.1 Le processus de développement logiciel 28

2.2.2 La problématique de l’évolution logicielle 35

2.2.3 Travaux antérieurs 40

2.2.4 Relation avec la thèse 42

2.3 La gestion de l’incohérence 42

2.3.1 Définitions de base et processus 43

2.3.2 Détection des chevauchements 48

2.3.3 Détection de l’incohérence 55

2.3.4 Diagnostic d'incohérences 59

2.3.5 Traitement des incohérences 61

2.3.6 Localisation 63

2.3.7 Spécification et l'application des politiques de gestion d’incohérence 64

2.4 Conclusion 67

2.1 Introduction

Pour les logiciels industriels de longue durée, la plus grande partie des coûts de cycle de vie est préoccupée par l'évolution du logiciel afin de permettre aux entreprises de répondre aux exigences du marché et aux inflexions des besoins de leurs différents interlocuteurs [46]. Il est nécessaire de modifier les logiciels sur une base constante avec les grandes améliorations dans un délai très court afin de faire face à de nouvelles opportunités commerciales. Ceci impose des exigences critiques sur la capacité de modification rapide et d’amélioration des systèmes logiciels pour obtenir une évolution rentable du système logiciel.

Lehman [47] met en évidence 2 lois de l’évolution des logiciels qui permettent d’expliquer pourquoi la phase de maintenance ou d’évolution logiciels peut être considérée comme la plus longue phase du cycle de vie du logiciel. Sa première loi est qualifiée de « *Law of Continuing Change* », c’est-à-dire que le système doit nécessairement changer afin d’être toujours utile pour ses utilisateurs. La seconde loi « *Law of Increasing Complexity* » signifie que la structure d’un programme se détériore à mesure que celui-ci évolue. Au fil du temps, la structure du code d’un programme se dégrade jusqu’à ce qu’il devienne plus rentable de le réécrire.

Le maintien de la cohérence des spécifications, de la documentation technique, des fichiers de configuration et du code source est une tâche qui prend du temps et qui peut impliquer plusieurs personnes de différents niveaux de l'organisation. Une telle tâche s'avère coûteuse et par conséquent facilement abandonnée de la portée d'un projet. Les incohérences peuvent être introduites au cours de la phase de développement et restent cachées jusqu'à une phase tardive de développement ou même jusqu'à ce que le produit soit libéré et la maintenance est nécessaire. Dans ce dernier cas, l'impact et le coût des incohérences peuvent affecter le projet. La phase d'entretien est également une source commune de nouvelles incohérences si les changements requis par la tâche de maintenance ne sont pas documentés et tous les documents relatifs au changement ne sont pas mis à jour en conséquence. Un mécanisme pour relier deux ou plusieurs artefacts différents afin que les changements dans l'un d'eux se retrouvent dans les autres, va aider à maintenir la cohérence de ces artefacts.

L’objectif de ce chapitre est de proposer une synthèse de l’évolution logicielle et de montrer l’importance du mécanisme de gestion de la cohérence pour faire face aux challenges imposés par les forces du changement.

Le chapitre comporte deux grandes sections :

- La première expose la problématique de l’évolution logicielle ainsi que le positionne-

ment de la cohérence par rapport à cette activité majeure de l’ingénierie logicielle.

- La deuxième section donne la définition de la cohérence et résume les nombreuses techniques et méthodes proposées par la communauté du génie logiciel pour soutenir la gestion de la cohérence des différents artefacts logiciels.

2.2 Evolution logicielle

2.2.1 Le processus de développement logiciel

Le développement de logiciel consiste à concevoir, étudier, construire, vérifier, valider, maintenir et améliorer des logiciels.

Habituellement, le logiciel subit plusieurs étapes distinctes durant son cycle de vie [48] (voir la figure 2) :

1. **Le développement initial** : le développement de la première version de fonctionnement du système.
2. **Evolution** : les ingénieurs étendent les capacités et les fonctionnalités du système pour répondre aux besoins de ses utilisateurs.
3. **Entretien** : le logiciel est soumis à la réparation de défauts mineurs et des changements très simples dans la fonction.
4. **Élimination progressive** : pas plus d'entretien, les propriétaires cherchent à générer des revenus à partir de l'utilisation aussi longtemps que possible.
5. **Fermeture** : le logiciel est retiré du marché, et tous les utilisateurs se dirigent vers un système de remplacement s’il existe.

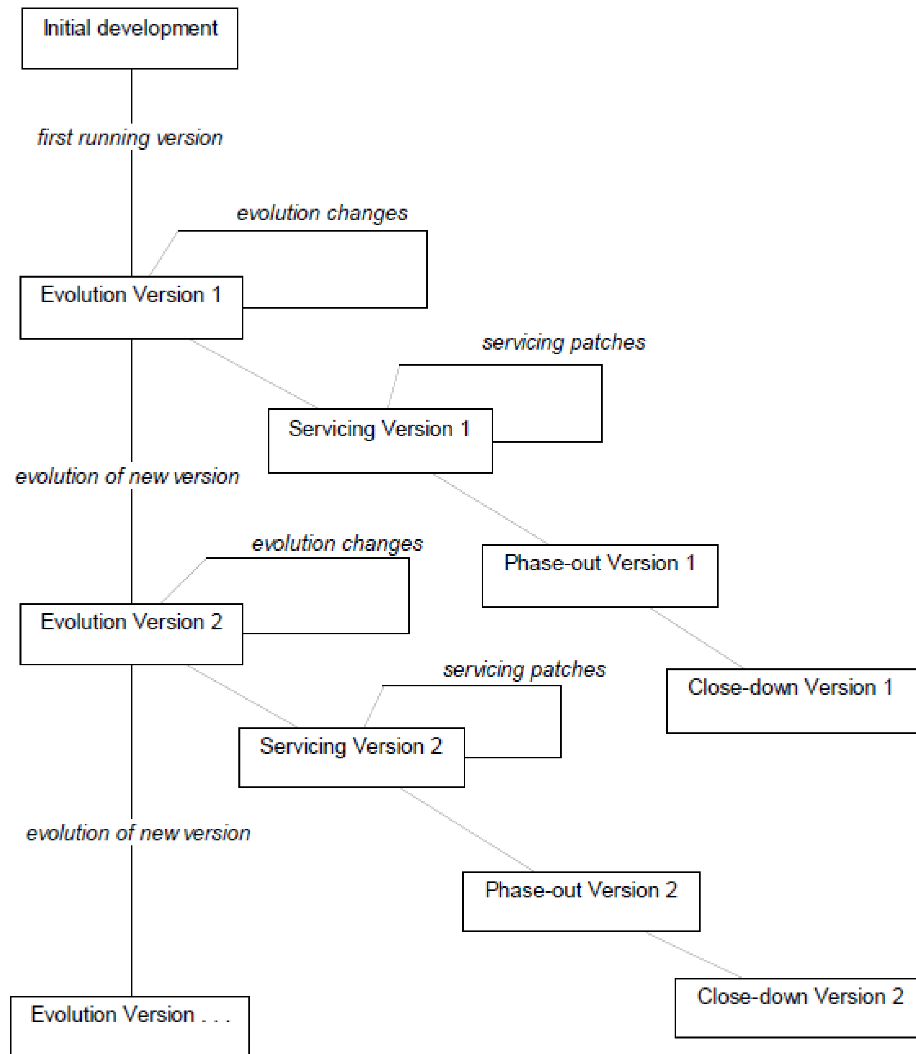

Figure 2 : The versioned staged model [48]

2.2.1.1 Les activités du processus de développement

« *The software engineering process is the total set of software engineering activities needed to transform a user's requirements into software.* »[6] [49]

Le processus de développement logiciel est un processus complexe, qui est réalisé à travers de

6 « Le processus de génie logiciel est l'ensemble des activités de génie logiciel nécessaire pour transformer les besoins d'un utilisateur en un logiciel. »

nombreuses activités. Notre objet n'est pas de recenser de manière exhaustive ces activités, ce qui semble impossible. En effet, les processus de développement se diffèrent d'un projet à l'autre et d'une organisation à l'autre, les activités qui y sont menées sont également différentes. Il s'agit plutôt ici de dresser un panorama représentatif des différents types d'activités qui sont susceptibles d'intervenir au cours du processus de développement logiciel. Pour y arriver, nous nous référons à la taxonomie des normes du génie logiciel mise en place par l'association professionnelle IEEE [50]. Cette taxonomie décrit le contenu de l'ingénierie logiciel en comptant les activités distinctes qui en font partie. D'après cette classification, qui figure dans le tableau 1, ces activités peuvent être classées en trois groupes : les activités relatives à l'ingénierie du produit, les activités de vérification et de validation, ainsi que les activités ayant trait à la gestion technique. Ces trois groupes contiennent des activités majeures pouvant être réalisées en parallèle, visant à produire, vérifier et piloter, sans être figées dans une étape particulière du processus de développement.

2.2.1.1.1 Ingénierie du produit

Les activités d'ingénierie du produit contribuent à la constitution du produit final, en passant par les différentes phases de la définition, la réalisation et le support du logiciel.

INGÉNIERIE DU PRODUIT (product engineering)
Analyse des besoins (requirements analysis)
Conception (design)
Codage (coding)
Intégration (integration)
Conversion (conversion)
Débogage (debugging)
Support du produit (product support)
Maintenance (maintenance)
VÉRIFICATION ET VALIDATION (verification and validation)
Revues et audits (reviews and audits)
Analyse du produit (product analysis)
Test (testing)
GESTION TECHNIQUE (technical management)
Gestion du processus (process management)
Gestion du produit (product management)
Gestion des ressources (resource management)

Tableau 1 : Classification des activités du génie logiciel [50]

L'activité d'analyse des besoins : Analyser les besoins des utilisateurs puis définir et spécifier les fonctionnalités du produit [51], [52]. Le résultat de cette activité consiste à préparer un document appelé spécification du produit ou cahier des charges du produit. Cette activité peut être réalisée en passant par plusieurs étapes :

1. Recueillir les besoins auprès des parties prenantes (réunions, communications écrites ou orales, etc.),

2. Interprétation et analyse des données recueillies,

3. Modélisation et validation par des experts en vue de transformer ces besoins en spécifications logicielles.

B. Nuseibeh et S. Easterbrook ont présenté les différentes activités inclues dans l'analyse et la spécification des besoins, ainsi que des différentes manières d'y arriver [52]. Les créateurs annoncent le fait que ces activités nécessitent la coopération et l'intervention de plusieurs acteurs différents dans le but de parvenir à une spécification cohérente du logiciel à partir des besoins exprimés.

L'activité de conception : cette activité peut être résumée dans la préparation de deux documents, le document de la conception générale ou architecturale, et le document de la conception détaillée.

Le document de la conception générale permet d'étudier et de concevoir plusieurs solutions en sélectionnant celle qui répond plus aux exigences du client. Ce choix est le résultat d'un processus coopératif impliquant plusieurs acteurs de l'équipe de développement. La solution choisie sera ensuite détaillée par l'implication de plusieurs experts de l'équipe de développement. En particulier, Il s'agit de mettre en place l'architecture détaillée de la solution, en précisant son organisation structurée en entités, les interfaces de ces entités ainsi que leurs interactions.

La phase de conception détaillée affine la conception générale. Après la validation de la conception générale par le client, les experts de développement procèdent à la conception détaillée en décomposant successivement les éléments définis par la conception générale en entités plus élémentaires, jusqu'au niveau où ces entités peuvent être implémentées, c'est-à-dire correspondent à des composants logiciels élémentaires. Cette activité dépend fortement du langage utilisé pour l'implémentation. Chacun de ces composants est également décrit en détail (son interface, les algorithmes qu'il implémente, le traitement des erreurs, ses performances, etc.). Ces activités de conception sont également l'occasion de préparer les tests qui seront appliqués sur les composants développés. En effet, la conception est rarement définitive au sein d'un projet : elle est souvent corrigée, modifiée, adaptée pour les nouveaux besoins ou en fonction de nouvelles exigences [53]; dans certains pro-

jets, elle a même lieu de façon informelle tout au long du processus de développement [54].

L’activité de codage : c’est la phase d’implémentation du logiciel et ces composants en traduisant la conception dans un langage de programmation. Cette activité peut être détaillée en sous-activités, citons par exemple l’implémentation du code source, la liaison avec des librairies externes, etc. Dans le cas des applications complexes, le travail d’implémentation peut être confié et réparti sur plusieurs équipes de développeurs.

L’activité d’intégration : Durant cette activité, les développeurs chargés de l’intégration prennent la responsabilité d’assembler les différents code sources et composants issus des activités de codage dans le but de construire un logiciel complet, en respectant rigoureusement les spécifications des tests d’intégration. L’activité d’intégration tend aussi à favoriser la réutilisation de composants logiciels existants, ce qui implique d’autres problématiques, notamment le problème de standardisation des interfaces, le fait de garantir la cohérence de l’environnement alors que les composants logiciels tiers sont susceptibles d’évoluer indépendamment, et d’une manière générale les problèmes d’interopérabilité [55], [56]. Par ailleurs, la globalisation du processus de développement rend cette activité d’autant plus critique et nécessite des mécanismes de gestion de l’incohérence logicielle tout au long de ce processus de développement.

L’activité de conversion : Consiste à modifier ou adapter le logiciel de telle sorte qu’il soit en mesure d’être opérationnel avec les mêmes fonctionnalités dans un environnement (langage, système d’exploitation, etc.) différent [57]. Cette activité fait partie d’un domaine plus large de l’évolution du logiciel que nous évoquons plus loin.

L’activité de débogage : Détecter, analyser et corriger des erreurs où de bogues. Ils y a plusieurs outils de tests unitaires et fonctionnels permettant d’assister la détection de certains bogues. Pour améliorer l’efficacité de l’activité de débogage, une étroite collaboration entre les développeurs est nécessaire [58]. Dans le domaine de l’Open Source, les utilisateurs sont également mis à contribution pour rapporter à l’équipe de développement les bogues qu’ils observent lors de l’utilisation du logiciel [59]. Ce processus collaboratif nécessite donc également un bon support.

L’activité de support : cette activité consiste à donner une formation pour installer le logiciel, assister et fournir des informations aux utilisateurs finaux, dans le dut de rendre le logiciel accessible et opérationnel aux utilisateurs dans l’environnement cible. Ces activités sous-entendent également une collaboration étroite entre les différents acteurs du développement et les utilisateurs. Généralement, une fois le produit installé et livré, ce support se poursuit, souvent à distance, et il est nécessaire de fournir les moyens qui faciliteront le transfert de connaissance et d’expérience des experts vers les utilisateurs [60], [61], [62].

L'activité de maintenance : consiste à modifier le logiciel après sa mise en service, afin de corriger les fautes, d'améliorer les performances ou d'autres critères, ou pour adapter le produit à un environnement modifié [63], [64]. L'objectif est de modifier le logiciel ou un de ses composants, tout en préservant son intégrité. La complexité dans les systèmes croissants c'est que les technologies évoluent rapidement et la maintenance est devenue un processus long et coûteux, et qui implique de nombreux acteurs. Aujourd'hui, on préfère parler d'évolution du logiciel, en référence au caractère complexe de ce processus, et qui est devenu un sujet brûlant dans le domaine du génie logiciel [65], [64], [66], [67].

2.2.1.1.2 Vérification et validation

Les activités de vérification et de validation sont les activités techniques mises en œuvre pour maîtriser la qualité du logiciel tout au long du processus de développement [68]. **La validation** permet de s'assurer que l'on a construit le bon produit qui correspond bien au cahier des charges et qui donne satisfaction aux clients. **La vérification** permet de s'assurer que le produit satisfait tous les exigences de qualité et qu'il a été bien construit.

L'activité d'analyse du produit : Consiste à déterminer si le logiciel produit possède certaines caractéristiques, en passant par une évaluation manuelle ou automatique. On peut différencier l'analyse statique (qui n'impacte pas l'exécution du logiciel) de l'analyse dynamique (qui correspond aux tests) [69], [70]. Les activités de revue et de test font partie de l'analyse du produit.

L'activité de revue : Cette phase consiste à observer en détail le travail des développeurs par les experts de l'évolution, ce travail peut être une spécification, une conception ou un code source, et l'objectif de cette phase consiste à repérer les défauts, les violations de normes de développement ou d'autres problèmes [71]. Cette opération peut être effectuée par un organisme externe, et dans ce cas on parle d'audit. Un audit a pour but de déterminer que les procédures, les instructions, les règles et les normes de développement, ainsi que d'autres exigences contractuelles, sont adéquates et respectées, et que le produit final a effectivement été réalisé [72].

L'activité de test : Egalement appelée l'analyse dynamique, il consiste à exécuter un programme ou un de ses composants pour s'assurer que les résultats effectifs correspondent à ceux attendus, ou bien qu'il répond à d'autres exigences (comme des critères de performance, par exemple). On distingue plusieurs sortes de tests comme par exemple les tests unitaires, qui permettent l'évaluation individuelle de composants logiciels, les tests d'intégration, qui vérifient le bon assemblage des composants, et d'autres encore [73].

2.2.1.1.3 Gestion technique

Les activités de gestion technique servent à structurer, planifier et piloter les activités de génie logiciel. La plupart de ces activités (la gestion de projet, la gestion de la configuration, la gestion de la documentation, etc.) couvrent presque la totalité du processus de développement logiciel.

La gestion du processus : Cette activité consiste à conduire, suivre et coordonner les travaux entamés pour développer un produit ou fournir un service. L'assurance de qualité [74], représente un exemple de gestion du processus.

La gestion du produit : Cette activité consiste à définir, coordonner et suivre des caractéristiques d'un produit durant son développement. La gestion de la configuration [75], représente un exemple de gestion de produit et qui a pour objectif de garantir la cohérence des artefacts logiciels tout au long de leur évolution.

La gestion des ressources : Cette activité consiste à identifier, estimer, allouer et surveiller les ressources utilisées pour le développement et l'évolution d'un produit, ou bien la fourniture d'un service.

2.2.1.2 Modèles et méthodes de cycle de développement

Ces différentes activités couvrent tout le cycle de vie de logiciel. Elles sont organisées et structurées pour soutenir la production d'un logiciel de qualité [76], [77]. Ce concept décrit la « vie » standard d'un produit logiciel de manière chronologique. Les cycles de vie de développement sont décomposés en une séquence de phases, chacune de ces phases est composée en un ensemble d'activités. Plusieurs modèles de cycle de vie de développement ont été présentés, qui normalisent la décomposition du processus de développement en phases et en activités ainsi que les artefacts ou documents qui transiteront entre les phases. On parle aussi de méthodes de développement. Pour un aperçu historique des différents modèles et méthodes, le lecteur pourra se référer à [78].

Les premiers modèles de cycle de vie du logiciel s'inspirent des modèles de production de l'industrie traditionnelle. Par exemple, dans le modèle en cascade [76] le développement logiciel est considéré comme un processus linéaire, toutes les phases du cycle de développement se poursuivent successivement. Généralement, les perspectives issues du domaine industriel ne sont pas adaptées au

domaine du développement logiciel [79], [80], [81]. Les autres modèles qui s'en inspirent, comme le modèle du cycle en V [82], [83], essaient de simplifier cet enchaînement pour tenir compte des « anomalies » pouvant intervenir dans le processus de développement. Néanmoins, ces approches séquentielles restent critiquées notamment en raison de leur rigidité [81], mais aussi en raison de leur caractère exclusivement temporel [77].

2.2.2 La problématique de l'évolution logicielle

Lors d'une conférence organisée par l'OTAN en 1968, ils ont étudié le phénomène de la crise logicielle qui concerne la difficulté des projets informatiques pour le respect des délais, les coûts et les besoins des utilisateurs.

Face à cette crise, des langages plus avancés ont été proposés pour améliorer la productivité, les méthodes de conception permettant de lier les fonctionnalités du logiciel avec les besoins des utilisateurs et les méthodes de gestion de projets plus adaptées, etc.

Les résultats de CHAOS en 2012 montrent une autre augmentation du taux de réussite des projets [84] (voir le tableau 2), 39% de tous les projets réussis (livrée à temps, budget respecté, avec les caractéristiques et les fonctions requises); 43% ont été contestées (de la fin, sur le budget, et / ou avec moins de caractéristiques et de fonctions requises); et 18% ont été échoué (annulé avant la fin ou livrés et jamais utilisé). Ces chiffres représentent une légère hausse dans les taux de réussite de l'étude précédente, ainsi que d'une diminution du nombre de défaillances. Le point bas au cours des cinq dernières périodes de l'étude était de 2004, dans lequel seulement 29% des projets ont réussis. Les résultats de cette année représentent la meilleure augmentation pour les taux de réussite dans l'histoire de la recherche CHAOS.

	2004	2006	2008	2010	2012
Successful	29%	35%	32%	37%	39%
Failed	18%	19%	24%	21%	18%
Challenged	53%	46%	44%	42%	43%

Tableau 2 : Résultats de la résolution du projet de recherche CHAOS - 2004-2012 [84]

Cette étude ne correspond qu'à la première version d'un logiciel. Or celui-ci doit forcément

évoluer pour s’adapter aux différentes exigences et besoins de ses utilisateurs.

De notre point de vue, la réussite d’un projet ne se limite pas à pouvoir livrer une première version opérationnelle, mais plutôt de développer un logiciel assez performant pour faire face aux différents changements.

2.2.2.1 Le cycle de vie d’un logiciel

Un cycle de vie du logiciel (SLC : Software Life Cycle) est essentiellement une série d'étapes, ou phases, qui fournissent un modèle pour le développement et la gestion du cycle de vie d'un logiciel. La méthodologie dans le processus SLC peut varier entre les industries et les organisations, mais des normes telles que ISO / IEC 12207 représentent des processus qui établissent un cycle de vie du logiciel, et de fournir un mode pour le développement, l'acquisition et la configuration de systèmes logiciels.

Le but d'un processus SLC et d’aider à produire un produit qui est rentable, efficace et de haute qualité. La méthodologie SLC connaît cinq grandes phases :

- **Naissance :** La création et le développement des logiciels est basé sur l’expression et l’analyse des contraintes et des besoins utilisateurs.
- **Croissance :** A chaque nouvelle version, des corrections implémentées et des nouvelles fonctionnalités ajoutées.
- **Maturité :** Le besoin en fonctionnalités est réduit. Les nouvelles versions contiennent que des adaptations et des corrections.
- **Déclin :** L’évolution du logiciel est contestable, le coût de la maintenance est inadmissible. La refonte ou bien le changement du logiciel est prévu.
- **Mort :** La décision de la refonte est prise. 0n participe à une migration de compétences et de données entre l’ancien et le nouveau système. Cette migration peut représenter une problématique si la connaissance de l’ancien système n’a pas été bien documentée.

Un logiciel est généralement lié à d’autres logiciels (systèmes d’exploitation, SGBD, librairies de composants, etc.). Alors, forcément le cycle de vie de ce logiciel est influencé par le cycle de vie des autres logiciels.

Il est devenu de plus en plus difficile de gérer le cycle de vie des logiciels suite à la complexité due à la dépendance aux autres logiciels. Cette problématique doit être prise en compte pour assurer la pérennité des investissements.

2.2.2.2 Les lois de l'évolution logicielle

S'appuyant sur des études empiriques sur l'évolution de IBM OS360 et d'autres systèmes industriels à grande échelle [85], [86]. Meir Lehman a identifié dans le cadre de son étude de l'évolution des systèmes logiciels un certain nombre d'observations sur la façon dont le logiciel évolue avec le temps. Initialement, trois lois ont été postulées, mais cinq autres ont été ajoutés par la suite [87], [88]. Les huit lois et leurs années de formulation sont résumées dans le tableau 3.

N/S	Nom de la loi	Description	Année
1	Changement continuel	Un système deviendra de moins en moins satisfaisant pour ses utilisateurs au fil du temps, sauf si elle est adaptée en permanence aux nouveaux besoins.	1974
2	Complexité croissante	Un système deviendra de plus en plus complexe, à moins que le travail soit fait pour réduire explicitement la complexité.	1974
3	Autorégulation	Le processus d'évolution du logiciel est auto régulateur par rapport à la distribution des produits et les artefacts de processus qui sont produites.	1974
4	Conservation de stabilité organisationnelle	Le taux d'activité moyen effectif global sur un système en évolution ne change pas au fil du temps; disant que la quantité moyenne de travail qui va dans chaque version est sensiblement le même.	1978
5	Conservation de la connaissance	La quantité de nouveau contenu dans chaque version successive d'un système tend à rester constante ou diminuer au fil du temps.	1978
6	Poursuite de la croissance	La quantité des fonctionnalités d'un système augmentera au fil du temps, pour faire plaisir à ses utilisateurs.	1991
7	Qualité déclinante	Un système sera perçu comme perte de qualité au fil du temps, à moins que son design est soigneusement entretenu et adapté aux nouvelles contraintes opérationnelles.	1996
8	Système de réaction	Réussir l'évolution d'un système logiciel exige la reconnaissance que le processus de développement est une multi-boucle, multi-agents, système de rétroaction multi-niveaux; ainsi, par exemple, comme un système de logiciel âge, il a tendance à devenir de plus en plus difficile de changer en raison de la complexité à la fois des objets ainsi que des processus impliqués dans le changement.	1996

Tableau 3 : Les lois de l'évolution logicielle

Les lois ci-dessus s'appliquent principalement aux logiciels propriétaires. En outre, les lois sont applicables sur les systèmes complexes développés par les grandes organisations. En fait, les lois représentent une théorie émergente du processus logiciel et l'évolution des logiciels basés sur de nom-

breuses entrées qui comprennent le développement de logiciels [89].

Parmi les conclusions extraites de ces lois, c’est qu’un système logiciel dépend fortement de son environnement extérieur. Les forces du changement appliquées sur un logiciel dépendent de celles qui s’appliquent à une entreprise. Ces changements poussent les logiciels à évoluer, la figure 3 présente quelques exemples.

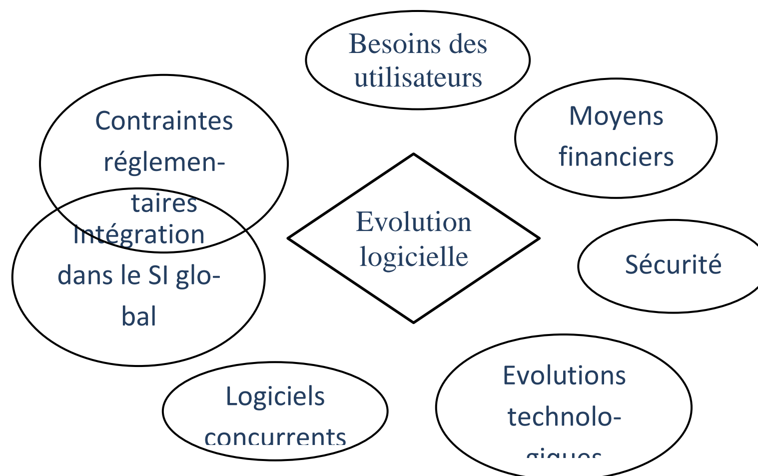

Figure 3 : Forces de changement et l'évolution des logiciels

2.2.2.3 L’érosion du design

L'érosion de la conception est un problème qui affecte les grands systèmes logiciels. Ce phénomène est aussi connu sous le nom de l'architecture dérivé [90], le logiciel de vieillissement [91] ou de l'architecture érosion [92]. Essentiellement, le problème est que le logiciel évolue, et progressivement change pour répondre aux nouvelles exigences, corriger les défauts ou optimiser les attributs de qualité (adaptation, maintenance corrective et perfectif [93]). Toutefois, ces exigences peuvent entrer en conflit avec les exigences des itérations antérieures ou peuvent modifier les hypothèses dans lesquelles les décisions de conception dans les itérations précédentes ont été faites. Il devient alors préférable de réécrire le logiciel plutôt de le faire évoluer.

2.2.2.4 La maintenance et l'évolution logicielle

La maintenance est une activité primordiale dans le domaine du génie logiciel. Elle est considérée parmi les problématiques majeures exprimées par les lois de l'évolution logicielle, permettant au logiciel de respecter les exigences des utilisateurs et de rentabiliser les investissements consacrés au logiciel.

Généralement, la maintenance est un processus de modification pour une version majeure d'un logiciel. Ce processus produit de petites évolutions ou des correctifs destinés à la production ou bien regroupés dans des versions mineures.

La figure 4 montre les étapes du processus de maintenance pour traitement des évolutions et des anomalies.

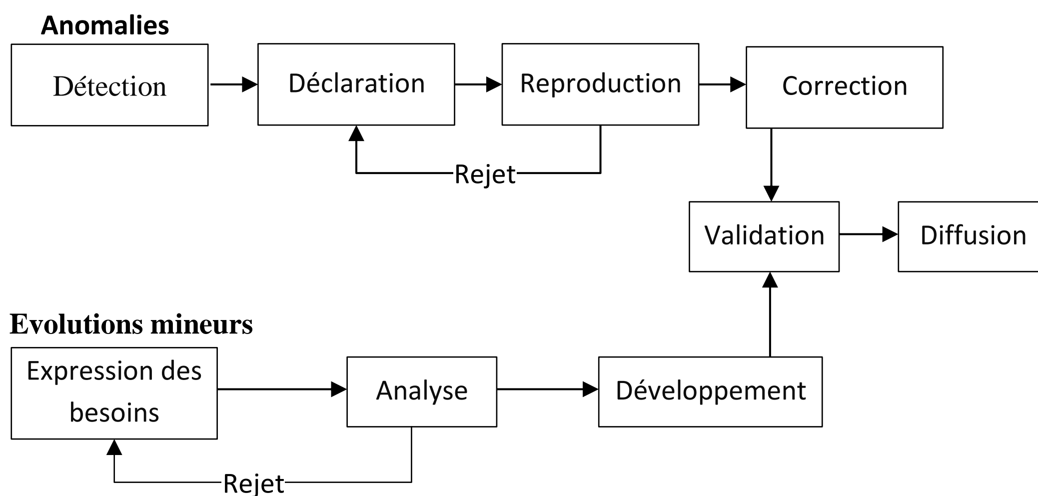

Figure 4 : Le processus de maintenance

Pour la gestion des anomalies (**maintenance corrective**), le processus de maintenance ressemble à celui de la recette, sauf que le test et la déclaration des anomalies sont réalisés par les clients finaux au lieu des testeurs. Dans ce cas, la correction devient plus complexe vu que la description et les scénarios de reproduction des anomalies sont généralement moins précis, À cela s'ajoutent les exigences de production en termes de délais de réaction, de contraintes de diffusion, etc.

Pour la gestion des évolutions mineures (**maintenance évolutive**), le processus de maintenance se rejoint à la gestion d'un petit projet de développement. Seulement les évolutions mineures qui peuvent être acceptées. Une évolution majeure peut être rejetée, comme elle peut être prise en compte dans la prochaine version du logiciel.

Selon Swanson [93], ci-dessous les deux autres catégories de maintenance :

- **Maintenance préventive :** il s’agit d’effectuer des développements pour prévenir des problèmes futurs.
- **Maintenance adaptative :** il s’agit d’ajuster le logiciel aux changements apparaissant dans l’environnement d’exécution.

En conclusion, la spécificité de la phase de maintenance par rapport à un processus de développement est la durée. Durant le cycle de vie d’un logiciel, la durée de la maintenance est généralement supérieure à celle du développement.

2.2.3 Travaux antérieurs

Les lois de l'évolution du logiciel sont formulées dans [47] et [88], sur la base des observations du fonctionnement du système IBM OS / 360 et le projet FEAST. Le terme évolution du logiciel est délibérément utilisé dans les travaux de Lehman pour aborder la différence avec l'activité de post-déploiement de la maintenance du logiciel. Il utilise le terme logiciel de type E pour désigner les programmes qui doivent être évolués parce qu'ils opèrent ou adressent le problème ou l'activité du monde réel. Par conséquent, des changements dans le monde réel vont affecter le logiciel et nécessitera des adaptations ultérieures.

L'architecture logicielle est forcément concernée par l’évolution en raison des phénomènes mentionnés dans le tableau 3 de l'évolution logicielle, prenant par exemple le changement continuel, la complexité croissante, la croissance continuelle et la baisse de qualité.

En outre, les propriétés des grands systèmes logiciels sont indiquées dans [94] :

- **La complexité** est une propriété essentielle dans les grands systèmes logiciels, conduisant à des problèmes suivants :
    - <u>Difficulté de communication</u> entre les membres de l'équipe de développement, ce qui conduit à des défauts dans les produits, des dépassements de coûts et des retards dans le calendrier ;
    - <u>Difficulté de compréhension</u> de tous les états possibles du programme ;

- Difficulté de l'extension des programmes avec des nouvelles fonctions sans créer des effets secondaires ;
- Difficulté d'obtenir une vue d'ensemble du système, empêchant ainsi l'intégrité conceptuelle.

- **La conformité** de nombreux systèmes logiciels sont limités par la nécessité de se conformer à des institutions humaines et des systèmes.
- Changeable : L'entité logicielle est constamment soumise à des pressions pour le changement.
- **Invisibilité :** Le logiciel est invisible et un-visualisable. Il n'y a aucune représentation géométrique. Au lieu de cela, il y a plusieurs graphiques distincts mais interdépendants des liens qui représentent différents aspects du système.

Les propriétés des grands systèmes logiciels notées dans [94], par exemple, la complexité des logiciels, des changements inévitables des systèmes logiciels et l'invisibilité en termes de représentation de la structure du logiciel, confirment davantage les phénomènes d'évolution des logiciels et présentent la nécessité intensifiée d'avoir des systèmes logiciels évolutifs qui s'adaptent aux changements d'une manière rentable tout en préservant l'intégrité architecturale. Sans contre-mesures actives, la qualité d'un système de logiciel va progressivement se dégrader pendant que le système évolue.

En outre, le vieillissement du logiciel est inévitable. Parnas utilise la métaphore de la décadence pour décrire comment et pourquoi le logiciel devient de plus en plus fragile au fil du temps [95]. Il existe deux types de vieillissement de logiciel qui peuvent conduire à un déclin rapide de la valeur d'un produit logiciel. La première est causée par l'échec des propriétaires du produit : modifier le produit pour répondre aux besoins des changeants ; le second est le résultat des modifications apportées. Les deux types de logiciels de vieillissement conduisent à l'évolutivité insuffisante. Les problèmes suivants sont associés au vieillissement des logiciels [95]:

- L’incapacité de suivre le marché en raison de la taille croissante et la complexité ;
- La réduction de la performance en raison de la détérioration progressive de la structure ;
- Diminution de la fiabilité en raison d'erreurs introduites lors des différents changements.

2.2.4 Relation avec la thèse

Afin de maintenir l'utilité du système comme il était, nous devons d'une manière continue de l'adapter aux exigences en constante évolution. Ça, présente la nécessité de contrôler l'évolution d'un système logiciel pour éviter les problématiques de l'évolution logicielle précédemment présentées. Par conséquent, le suivi de l'évolution du logiciel motive les raisons de la thèse, à savoir que nous devons rechercher les moyens pour gérer l'incohérence des données dans le but de contrôler l'évolution des systèmes logiciels.

2.3 La gestion de l'incohérence

La construction des systèmes logiciels complexes se caractérise par la répartition des rôles et des responsabilités entre les parties prenantes autonomes ou semi-autonomes (par exemple, les clients, les utilisateurs, les analystes, designers, développeurs, tiers, etc.). La répartition des responsabilités et des rôles se traduit souvent par la construction de nombreux objets partiels du système de développement (appelés «Artefacts logiciels"). Ces artefacts peuvent être un cahier des charges, un document de conception UML, un code source, un fichier de configuration XML, etc.

Les développeurs de logiciels travaillent avec une variété des artefacts logiciels dans différents niveaux d'abstraction, y compris les exigences en matière de logiciels, l'analyse, la conception, la mise en œuvre et la documentation. L'incohérence peut apparaitre entre les différents artefacts hétérogènes pendant ou entre les phases de développement de logiciels. Au fil du temps, ces incohérences doivent être résolues afin de produire un système performant et de qualité. L'utilisation d'outils de développement de logiciels sur un projet par plusieurs développeurs est habituellement essentielle lors du développement des systèmes logiciels complexes, mais l'utilisation de ces outils peut doubler la création des incohérences [96], [97].

La gestion des incohérences au cours d'un projet de développement logiciel peut être améliorée lorsque les outils utilisés sont strictement, ou même vaguement, intégré [97], [98], [99], [100]. Dans de tels systèmes, les développeurs interagissent sur différents artefacts logiciels hétérogènes (les exigences du système, l'analyse, la conception, le développement, la documentation, etc.) et dans différents niveaux d'abstraction [101], [98], [97].

Certaines incohérences peuvent être corrigées automatiquement, par des outils de mise à jour des informations par exemple. Cependant, de nombreuses incohérences ne peuvent pas, ou ne devrait

pas, être corrigé automatiquement. Alors, des mécanismes sont nécessaires dans l’environnement de développement des logiciels, permettant d’évaluer les règles de cohérence et informer les développeurs des différentes violations.

Les incohérences peuvent avoir des effets à la fois positifs et négatifs sur le cycle de vie de développement des logiciels. Sur le plan négatif, ils peuvent retarder et, par conséquent, augmenter le coût du processus de développement du système, mettre en péril les propriétés liées à la qualité du système (par exemple, la fiabilité et la sécurité), rendre la maintenance du système plus difficile. Sur le côté positif, les incohérences mettent en évidence les conflits entre les différents artefacts logiciels hétérogènes tout au long du processus de développement. Ces conflits doivent être traités dans une manière responsable, sinon, ils peuvent mettre en risque la performance, la qualité et la sécurité du système logiciel.

2.3.1 Définitions de base et processus

L'un des facteurs qui font qu'il est difficile de comprendre et contraster les résultats et les contributions des différents axes de recherche et des techniques développées pour résoudre les problèmes des incohérences dans les systèmes logiciels a été l'absence d'une terminologie couramment utilisée par les différents chercheurs dans ce domaine [102]. Dans cette section, nous essayons de :

a) Définir les principaux phénomènes qui se rapportent à des incohérences dans les modèles de logiciels, et

b) Décrire les principales activités du processus de gestion des incohérences.

2.3.1.1 Principales définitions

Nous utilisons le terme incohérence pour désigner toute situation dans laquelle un ensemble de descriptions ne respecte pas une relation qui est prescrit de tenir entre eux [103]. Une condition préalable à l'incohérence est que les descriptions en question ont une zone de chevauchement [104]. Une relation entre descriptions peut être exprimée comme une règle de cohérence, avec laquelle les descriptions peuvent être vérifiées. Dans la pratique actuelle, certaines de ces règles de cohérence sont capturées dans différents documents du projet, d'autres sont intégrés dans les outils, et certains ne sont pas prises n'importe où.

Le tableau 4 donne quelques exemples de règles de cohérence, exprimées en langage naturel. La règle 1 est une règle simple pour deux descriptions écrites de la même notation (dans ce cas DFD). La règle 2 est une règle de la cohérence entre trois documents différents ; la zone de chevauchement est l'utilisation des termes « utilisateur » et « emprunteur ». La règle 3 est un exemple d'une règle concernant le processus de développement. Si le code du programme a été entré avant le signe de la fin, alors le projet est incompatible avec une politique de processus (qui, sans doute, est documentée quelque part). A noter que dans ce dernier exemple, la zone de chevauchement est relative au statut, plutôt que le contenu de la description.

Les règles 2 et 3 reflètent un pattern commun : une relation de cohérence existe entre deux descriptions parce que certaine troisième description dite qu'il le devrait. Des problèmes se produisent si la relation à trois voies n’est pas traçable, ou si des modifications sont apportées à l'une des trois descriptions sans vérifier les autres. Dans certains cas, la troisième description n’est pas correctement documentée. Dans ce cas, la relation pourrait ne pas être remarquée. L'évolution vers une meilleure modélisation des processus logiciels a contribué à ce que plusieurs de ces relations sont documentées, mais il faut noter que ces relations ne sont pas toutes orientées processus (par exemple la règle 2).

N° Règle	Description de la règle
Règle 1	Dans un diagramme de flux de données (DFD), si un processus est décomposé en un diagramme séparé, alors, les flux d’entrée dans le processus parent doit être le même que les flux d'entrée en diagramme enfant de flux de données.
Règle 2	Pour un système de bibliothèque particulier, le concept de document des opérations affirme que «l'utilisateur» et «l’emprunteur» sont synonymes. Par conséquent, la liste des actions de l'utilisateur décrit dans les manuels d'aide doit correspondre à la liste des actions de l'emprunteur dans le cahier des charges.
Règle 3	La phase de codage ne devrait pas commencer avant que le document de spécifications des besoins du système soit signé par le conseil d'examen du projet. Par conséquent, le dépôt du code source du programme doit être vide jusqu'à ce que le SRS ait le statut «approuvé » par le PRB.

Tableau 4 : Exemples informelles des règles de cohérence

Cette définition de l'incohérence est plus large : il englobe de nombreux types d'incohérence qui se produisent dans le développement de logiciels. En particulier, la notion de l’incohérence logique est comprise dans notre définition, où la relation qui devrait tenir est celle qui ne devrait pas être possible de dériver une contradiction depuis un ensemble de propositions. La définition de l’incohérence de cette façon offre une certaine souplesse, car il ne nous lie pas à une notation particulière et nous permet d'envisager de nombreuses formes de l’incohérence tout au long du processus de développement. Ainsi, par exemple, la vérification des descriptions contre des règles de cohérence peuvent révéler des conflits des parties prenantes [105], des objectifs divergents [106], des failles dans un système en fonctionnement [107], et des écarts de processus de développement documentés [108].

2.3.1.2 Les processus de gestion de l'incohérence

La gestion des incohérences a été définie comme le processus par lequel les incohérences entre les modèles de logiciels sont traitées de manière à soutenir les objectifs des acteurs concernés [102]. Dans la littérature, deux Frameworks généraux ont été proposés, décrivant les activités qui constituent ce processus : l'un par [102] et l'autre par [52]. Ces deux Frameworks partagent la prémisse que l'incohérence dans les modèles de logiciels doit être établie par rapport à une règle de cohérence spécifique et que le processus de gestion des incohérences inclus des activités pour détecter, diagnostiquer et traiter les incohérences. Ces activités de base sont complétées par d'autres activités qui ne sont pas partagés par les deux Frameworks tels que la spécification et l'application des politiques de gestion des incohérences dans [102].

Spanoudakis et Zismannous en 2001 [109], ont proposé une série d'activités qui unifie les Frameworks ci-dessus et les modifie d'une manière qui reflète plus fidèlement la mise en œuvre du processus de gestion d'incohérence par les différentes techniques et méthodes qui ont été développés pour le soutenir. Ces activités sont les suivantes :

- **La détection des chevauchements**

Cette activité est réalisée afin d'identifier des chevauchements entre les modèles de logiciel. L'identification des chevauchements est un élément crucial de l'ensemble du processus, car les modèles avec aucun chevauchement ne peuvent pas être incohérents [110], [111], [112], [113], [114], [102], [115], [104]. L'identification des chevauchements est effectuée par l'agent (s) qui est spécifié dans la politique de gestion de l'incohérence (voir ci-dessous).

- **La détection des incohérences**

Cette activité est réalisée pour vérifier les violations des règles de cohérence par les modèles de logiciels. Les règles de cohérence qui doivent être vérifiées sont établies par la politique de gestion de l'incohérence adoptée (voir ci-dessous). Cette politique précise également les circonstances qui déclenchent les contrôles.

- **Diagnostic d'incohérences**

Cette activité porte sur l'identification de la source, la cause et l'impact d'une incohérence. La source d'incohérence est l'ensemble des éléments des modèles de logiciels qui ont été utilisés dans la construction de l'argument qui montre que les modèles violent une règle de cohérence [52]. La cause d'une incohérence est définie comme le conflit dans les perspectives et/ou les objectifs des parties

prenantes qui sont exprimées par les éléments des modèles qui donnent lieu à l'incohérence [109]. L'impact d'une incohérence est défini comme les conséquences que l'incohérence a pour un système.

La source et la cause d'incohérence ont un rôle très important dans le processus de gestion de l'incohérence, car elles peuvent être utilisées pour déterminer les options disponibles pour résoudre ou améliorer une incohérence, le coût et les avantages de l'application de chacune de ces options (voir la manipulation des incohérences ci-dessous). Etablissement de l'impact d'une incohérence en termes qualitatifs ou quantitatifs est également nécessaire pour décider avec quelle priorité l'incohérence doit être traitée et pour évaluer les risques associés aux actions pour traiter l'incohérence (voir ci-dessous la manipulation des incohérences).

- **Traitement des incohérences**

Le traitement des incohérences a été considéré comme une activité centrale dans la gestion de l'incohérence [106], [116]. Cette activité concerne :

a) L'identification des actions possibles pour faire face à une incohérence,

b) L'évaluation du coût et les avantages qui viennent de chacune des actions de l’application,

c) L'évaluation des risques qui viennent si l’incohérence n’a pas été résolue.

d) La sélection de l'une des actions à exécuter.

- **Le Tracking**

Cette activité concerne l'enregistrement de :

a) Le fondement du raisonnement de la détection d'une incohérence

b) La source, la cause et l'impact de l’incohérence

c) Les actions de manipulation qui ont été considérées dans la connexion avec l’incohérence

d) Le fondement des arguments de la décision pour choisir une de ces options et de re-jeter l'autre.

Garder la trace de ce qui est arrivé dans le procédé permet la compréhension des résultats,

des décisions et des mesures prises par ceux qui auraient besoin d'utiliser ou de se référer à des modèles de logiciels dans les étapes ultérieures du cycle de vie de développement du système. Cela est particulièrement vrai pour ceux qui peuvent ne pas avoir été impliqué dans le développement du système et / ou le processus de gestion des incohérences détectées. Toutefois, un suivi détaillé de ce type impose certainement un frais de gestion de l'information pour le processus de gestion de l'incohérence et le processus de développement de logiciels. Cette surcharge doit être soigneusement évaluée par rapport aux bénéfices attendus.

- **Spécification et application d'une politique de gestion de l'incohérence**

Il y a de nombreuses questions qui ont besoin d'une réponse avant et pendant l'application d'un processus de gestion de l'incohérence. Ce sont des questions à propos de l'agent qui doit être utilisé pour identifier les chevauchements entre certains types de modèles de logiciels, des questions sur le moment et la fréquence des incohérences qui doivent être détectés, des questions sur les tests de diagnostic qui devraient être appliquées aux violations des règles de cohérence spécifiques, des questions sur la techniques qui devraient être appliquées pour évaluer les coûts, les avantages et les risques liés aux options de manipulation d’incohérence et des questions sur les intervenants qui entreprendront la responsabilité de traiter les incohérences. Les réponses à ces questions dépendent de la nature des modèles de logiciels que le processus de gestion de l'incohérence aura à traiter, le processus de développement de logiciels généralement préconisée dans un projet spécifique [52] et les normes particulières que ce processus veut observer [117], et les caractéristiques de l'équipe de développement de logiciel (par exemple la disponibilité des parties prenantes au cours de certaines activités de gestion de l'incohérence).

Il est clair que pour fournir des réponses cohérentes et efficaces aux questions ci-dessus, il est nécessaire d'avoir une politique sur la gestion de l'incohérence qui doit être appliquée à un projet particulier [102]. Cette politique doit préciser :

a) L'agent qui doit être utilisé pour identifier des chevauchements entre les modèles partiels

b) Les règles de cohérence qui doivent être vérifiées contre les modèles

c) Les circonstances qui déclenchent la détection des chevauchements et les incohérences

d) Les mécanismes qui devraient être utilisés pour diagnostiquer les incohérences et les circonstances qui devraient déclencher cette activité

e) Les mécanismes qui devraient être utilisés pour évaluer l'impact des incohérences et les circonstances qui devraient déclencher cette activité.

f) Les mécanismes qui devraient être utilisés pour évaluer les coûts, les avantages et les risques associés aux différentes options de traitement de l’incohérence.

g) Les parties prenantes qui devraient avoir la responsabilité de traiter les incohérences

Cette activité établit également les mécanismes pour appliquer une politique de gestion des incohérences dans un projet et le suivi de cette application pour s'assurer que des progrès sont réalisés en ce qui concerne les objectifs généraux que la politique vise à atteindre [102].

Dans les sections suivantes, nous présentons les principaux moyens par lesquels les diverses méthodes et techniques qui ont été développées pour traiter les incohérences dans les modèles de logiciels soutiennent les activités de gestion de l'incohérence ci-dessus. A la fin de chaque section, nous présentons un tableau résumant les techniques existantes.

2.3.2 Détection des chevauchements

Les méthodes et les techniques qui ont été développées pour appuyer la gestion des incohérences dans la détection des chevauchements des modèles logiciels sur la base des conventions de représentation, des ontologies partagées, d’inspection humaine et des formes d'analyse de similarité des modèles concernés.

Les différentes façons dont chacune de ces approches ont été réalisés sont discuté après avoir présenté les propriétés des différents types de relations de chevauchement définies dans [104] ont montré que :

- Chevauchement total est une relation réflexive, symétrique et transitive
- Chevauchement inclusif est une relation non réflexive, antisymétrique et transitive
- Chevauchement partiel est une relation non réflexive et symétrique

Ces propriétés nous donnent une base pour établir le type exact des relations de chevauchement que chacune des approches de la littérature peut identifier.

2.3.2.1 Conventions de représentation

La plus simple et la plus courante convention de représentation est de supposer l'existence d'un chevauchement totale entre les éléments du modèle avec des noms identiques et pas de chevauchement entre toute autre paire d'éléments. Cette convention est largement déployée par la vérification du model [118], [119] et d'autres méthodes et techniques d'analyse de modèle spécialisés [120].

La même convention constitue également la base des algorithmes d'unification classique [121] et des processus d'appariement prédicat qui sont utilisés pour détecter les chevauchements dans toutes les méthodes et techniques basés sur la logique de gestion de l'incohérence [122], [123], [124], [125], [106], [115], [117], [126]. Les algorithmes d'unification classiques trouvent l'unificateur le plus général entre les termes exprimés dans le langage logique du premier ordre. Un terme dans une telle langue est une constante ou un symbole de variable, ou un symbole de fonction suivie par une série d'autres termes séparés par des virgules.

Les algorithmes d'unification effectuent un filtrage syntaxique entre les termes présentés pour unification. Pour les termes qui commencent par un symbole de fonction, la mise en correspondance est réussie que si ces symboles sont les mêmes pour les deux termes, les termes ont le même nombre de sous-termes, et la mise en correspondance de leurs sous-termes est également réussie. Les symboles de variables peuvent être appariés librement avec des constantes, d'autres symboles de variables ou avec des termes qui commencent avec des symboles de fonction, à condition que ceux-ci ne contiennent pas le symbole variable qui est apparié avec l'un de leurs sous-termes (par exemple, la vérification de reproduction [121]). Ce processus d'appariement renvoie une cartographie des variables des termes sur d'autres termes, appelé substitution. L'unificateur le plus général est une substitution σ entre deux termes t_1 et t_2 pour lesquels il y a toujours une autre substitution τ qui peut être appliqué à σ afin de le traduire en toute autre unificateur θ de t_1 et t_2, qui est $\tau(\sigma) = \theta$[7].

Comme un exemple de terme classique d'unification, considérer les attributs « *etudiant* » et « *superviseur_projet* » ci-dessous :

- *etudiant(x)*,
- *etudiant(a)*,
- *etudiant(y)*,

7 $\tau(\sigma)$ désigne l'application de la substitution τ sur le résultat de l'application de la substitution σ.

- *superviseur_projet(x,tuteur_personnel(x))*,
- *superviseur_projet(y,x)*, and
- *superviseur_projet(a,tuteur_personnel(a))*

Où "*etudiant*" et "*superviseur_projet*" sont des symboles de prédicats, "tuteur_personnel" est un symbole de fonction, *x* et *y* sont des symboles de variables et *a* une constante. Dans cet exemple, l'unificateur le plus général de *etudiant(x)* et *etudiant(a)* est {*x* / *a*}, l'unificateur le plus général de *etudiant(x)* et *etudiant(y)* est {*x* / *y*} et l'unificateur le plus général de *superviseur_projet(x,tuteur_personnel(x))* et *superviseur_projet(a,tuteur_personnel(a))* est {*x* / *a*}[8].

2.3.2.2 Ontologies partagées

Une approche alternative pour l'identification des chevauchements est d'utiliser des ontologies partagées. Cette approche exige les auteurs des modèles de marquer les éléments avec les pièces dans une ontologie partagée. L'étiquette d'un élément de modèle est prise pour désigner son interprétation dans le domaine décrit par l'ontologie, et il est donc utilisé pour identifier les chevauchements entre les éléments des différents modèles. Un chevauchement total dans cette approche est présumé lorsque deux éléments du modèle sont "marqués" avec le même point dans l'ontologie [111], [127], [128], [129].

Les ontologies utilisées par [127], et [128] dans leur système Oz sont des modèles de domaine qui prescrivent des hiérarchies détaillées d'objets de domaine, des relations entre eux, des objectifs qui peuvent être détenues par les parties prenantes, et des opérateurs qui réalisent ces objectifs. Les modèles de logiciels qui peuvent être manipulés par leurs techniques sont construits par instanciation des modèles de domaines communs qu'ils ont proposé. Dans la recherche d'incohérences, leurs techniques assument du chevauchement total entre les éléments du modèle qui instancient le même objectif dans le modèle de domaine. L'ontologie utilisée par le système QARCC [129] est une taxonomie de décomposition des attributs de qualité du système logiciel. Cette ontologie concerne également des attributs de qualité avec des architectures logicielles et des procédés de développement qui peuvent être utilisés pour les obtenir ou les inhiber. Par exemple, il existe un attribut de qualité appelé "*portabilité*" dans cette ontologie qui est décomposé de deux attributs de qualité "*évolutivité*" et "*modificabilité*". Portabilité, selon l'ontologie de QARCC peut être atteinte grâce à une architecture de système

8 Le symbole "/" veut dire "est remplacé par".

"*couches*" et un "*prototypage*" approche de développement du système. Les modèles des exigences logicielles que QARCC analyse pour les incohérences sont décrits à travers des conditions qui sont reliées à des attributs de qualité. Un chevauchement entre deux conditions est établi si l'attribut de qualité dont réfèrent ces conditions peut être réalisé ou inhibé par un processus de développement de l'architecture ou un système commun.

Il doit être apprécié que dans le but de faciliter la communication entre les parties prenantes sur un domaine spécifique de discours, sans prendre nécessairement l'existence d'une théorie globale et partagée entre les parties prenantes, les ontologies doivent fournir des définitions des articles commandés. En outre, les parties prenantes ont besoin de "commit" eux-mêmes à l'ontologie. Un engagement dans ce contexte signifie que les actions observables des intervenants sont en accord avec les définitions des éléments de [130], [131]. En conséquence de l'ontologie générale, les modèles de logiciels doivent ajouter beaucoup de détails à une ontologie pour décrire un système complexe avec un degré raisonnable d'exhaustivité. Cela conduit inévitablement à des associations de nombreux éléments du modèle avec le même point dans l'ontologie et la suite que des chevauchements gros grains peuvent être identifiés par l'utilisation de ces articles. En outre, depuis ontologies intègrent des définitions d'éléments, ils sont des modèles eux-mêmes et en tant que tels, ils peuvent être interprétées de différentes manières par différentes parties prenantes ! Ainsi, le même point de l'ontologie peut être utilisé pour attribuer des significations différentes à des éléments de différents modèles. En conséquence, l'existence de chevauchements entre les éléments associés au même article dans une ontologie ne peut présumer de la sécurité à moins de preuves que les parties prenantes ont comprises l'ontologie de la même façon.

2.3.2.3 Inspection humaine

Une troisième approche générale est d'obtenir les parties prenantes pour identifier les relations de chevauchement. De nombreuses méthodes et techniques reposent sur cette approche [110], [112], [113], [132], [114], [133], [115].

Synoptique [110], par exemple, attend les parties prenantes pour identifier les "forts" et les "faibles" correspondances entre les modèles qui correspondent au chevauchement total et partiel. Le soutien apporté par cet outil prend la forme d'une aide visuelle pour la navigation, la sélection graphique et l'enregistrement des chevauchements. Delugach [112], exige des parties prenantes pour définir les relations " homologue" entre les éléments du modèle. Ces relations correspondent au chevauchement total.

Zave et Jackson [113] suggèrent l'utilisation de "description" graphes pour relier les symboles de prédicat dans les «signatures» de différents modèles de logiciels. Une signature de modèle inclut les symboles de prédicats dans le modèle qui ne peuvent pas être définis en termes d'autres prédicats. La décision sur les prédicats exacts qui devraient appartenir à la signature d'un modèle dépend de la langue. Par exemple, dans le cas de représenter un diagramme de transition d'état dans un langage du premier ordre si les prédicats qui représentent les états d'un diagramme deviennent membres de la signature du modèle, puis les prédicats qui représentent les transitions du diagramme ne le seront pas et vice versa [113]. Une relation entre deux prédicats dans la description graphique ne peut être créée que s’il y a un chevauchement entre les deux modèles et inclut des affirmations qui intègrent les prédicats. Les relations spécifiées dans la description graphique ne fait pas de distinction entre les différents types de chevauchements.

Jackson [114], suggère l'identification des chevauchements en vertu de «désignations». Les désignations constituent un moyen d'associer des termes non clos dans les modèles formels avec les termes clos qui sont connus pour avoir des interprétations fiables et précises (appelées «phénomènes»). Une désignation associe un terme non clos avec une règle de reconnaissance qui identifie les phénomènes désignés par elle. La règle est énoncée en langage naturel. Les désignations peuvent certainement aider les parties prenantes à identifier les chevauchements, mais ne doivent pas être utilisés comme indications définitives d'entre eux. La raison est que les règles de reconnaissance des désignations pourraient s’admettre des interprétations différentes.

Boiten et autres [115], attendent les auteurs des schémas Z que leurs techniques sont confrontés à identifier les relations de «correspondance» entre les variables qui apparaissent dans ces schémas (mais pas les prédicats). Ces relations correspondent au chevauchement total et elles sont ensuite utilisées dans la construction de compositions possibles de deux ou plusieurs schémas Z (La méthode de base de l'exploration et la manipulation des incohérences dans leur approche). Une approche similaire est adoptée dans [133] pour le traitement ouvert des modèles répartis exprimés en LOTOS.

Fiadeiro & Maibaum [132], suggèrent que la représentation des relations de chevauchement entre les éléments du modèle en utilisant le framework formel de la théorie des catégories [134]. Ils formalisent des modèles comme des catégories (graphes orientés avec une structure de composition et d’identité) et utilisent des foncteurs entre ces catégories (par exemple les correspondances fonctionnelles entre les nœuds et les bords des catégories) pour les interconnecter. Les foncteurs sont ensuite vérifiées si elles préservent les structures et donc les propriétés de la catégorie des éléments qu'ils relient. Cette vérification constitue la détection des incohérences. L'idée d'utiliser des foncteurs pour représenter les chevauchements a été soutenue par d'autres auteurs [135], cependant, ils ont critiqué

l'attitude qu'il convient de vérifier si les foncteurs préservent les structures des parties des modèles qu'ils relient. Cette critique a été au motif que ce contrôle serait trop strict dans le développement de grands systèmes logiciels.

Spanoudakis et autres [104], ont également reconnu la nécessité de vérifier la cohérence des relations de chevauchement mais ils proposent une vérification moins stricte. Selon eux, un ensemble de relations de chevauchements revendiqués doivent être vérifiées si elles remplissent certaines propriétés qui découlent de la définition formelle de chevauchements des donnés. Par exemple, si il a été affirmé que un élément de modèle *a* se chevauche inclusivement avec un élément de modèle b mais n'a pas de chevauchement avec un troisième élément c alors, il faut vérifier qu'il n'y a pas un chevauchement totale, inclusive ou partielle entre b et c. À leur avis, cette vérification doit être effectuée avant de vérifier la cohérence des modèles concernés. Ce contrôle est particulièrement utile dans les cas de chevauchements affirmés par les humains.

La principale difficulté de l'identification des chevauchements en utilisant des inspections par les humains est que cette identification devient extrêmement longue, même pour les modèles de complexité moyenne.

2.3.2.4 L'analyse de similarité

La quatrième approche générale consiste à identifier les chevauchements par des comparaisons automatisées entre les modèles. Cette approche exploite le fait que les langages de modélisation intègrent les constructions qui impliquent ou suggèrent fortement l'existence de relations de chevauchement. Par exemple, la relation "Is-a" dans divers langages de modélisation orientée objet est une déclaration soit d'un chevauchement inclusif ou un chevauchement total. Ceci est parce que les relations "Is-a" ont normalement une sémantique set-inclusion, qui est un sous-type désigne un sous-ensemble approprié ou non approprié des instances du super-type. De même, l'implication (→) entre deux prédicats constitue une déclaration de recouvrement inclus dans un langage du premier ordre.

Les méthodes de comparaison qui ont été déployées pour réaliser cette recherche de l'approche des similarités structurelles et sémantiques, soit entre les modèles eux-mêmes [136] ou entre chaque modèle et structures abstraites qui constituent les pièces d'ontologies spécifiques de domaine [137]. Dans la méthode de "réconciliation" de [136], la détection des chevauchements est formulée comme une instance du problème d'appariement de graphes bipartites pondéré [138].

Dans l'ensemble, il convient de noter que les techniques d'analyse de similarité ont tendance à être sensibles à l'hétérogénéité étendu dans la représentation du modèle, la granularité et les niveaux d'abstraction.

2.3.2.5 Résumé

Un résumé des hypothèses, et des principaux aspects positifs ainsi que les limites des différentes approches de l'identification des chevauchements est illustré dans le tableau 5.

Approche	Hypothèses	Avantages	Limitations
Conventions de représentation	• Modèles exprimés dans un langage formel	• Moyen relativement peu coûteux de l'identification de chevauchement	• Sensible aux formes simples du modèle hétérogène • Applicable uniquement aux modèles exprimés dans la même langue
Ontologies partagées	• Existence d'ontologies bien définis • Modèles doivent être en rapport avec les ontologies	• Applicable aux modèles exprimés dans différentes langues	• Les parties prenantes peuvent comprendre une ontologie différemment • Identification seulement des chevauchements de coarsegrain
Inspection humain	• Les parties prenantes doivent identifier les relations de chevauchement	• Certitude dans l'identification de chevauchement • Applicable aux modèles exprimés dans différentes langues	• Main-d'œuvre
Analyse de similarité	• Modèles doivent être liés par un méta modèle commun	• Identification automatique des chevauchements • Applicable aux modèles exprimés dans différentes langues	• Sensible à modéliser l'hétérogénéité • Chevauchements résultant ne sont pas toujours exactes

Tableau 5 : Hypothèses, avantages et limitations des approches de la détection des chevauchements

2.3.3 Détection de l'incohérence

Notre enquête a indiqué que, il y a eu quatre grandes approches de la détection des incohérences dans les modèles de logiciels. Ceux-ci sont :

- L'approche basée sur la logique
- L'approche de contrôle de modèle
- L'approche d'analyse spécialisé du modèle, et
- L'approche d'exploration collaborative centrée sur l'humain

Les mécanismes de base utilisés par chacune de ces approches méritent d'être discutés par la suite.

2.3.3.1 La détection basée sur la logique

L'approche basée sur la logique pour la détection des incohérences se caractérise par l'utilisation d'une technique formelle d'inférence pour tirer les incohérences exprimées dans un langage formel (par exemple : logique classique du premier ordre ([123], [122], [103], [139], [104]), la logique temporelle en temps réel ([106], [140]), quasi-classique (QC) logique ([141]), langage de contrainte d'objet ([142]), la langue assertionnel de O-Telos ([125]), Z ([143], [124])).

Une approche similaire, en termes de mécanisme d'inférence utilisé, est également prise par [125], [126], [117], [142].

Nissen et autres dans [125] présentent les modèles de logiciels et les règles de cohérence exprimées dans O-Telos [144]. O-Telos est une variante de Telos [145], une connaissance de la langue de représentation orientée objet qui permet la spécification des contraintes d'intégrité et des règles de déduction en utilisant son propre sous-langage d'assertion. Les contraintes d'intégrité et les règles déductives de O-Telos sont associées à des classes spécifiques (ou méta-classes) pour restreindre les relations de leurs instances avec d'autres objets et d'en déduire des informations sur ces cas, respectivement.

Spanoudakis et Kassis dans [142] adoptent une approche très similaire pour vérifier si les modèles UML ont satisfait certaines règles de cohérence. Dans leur Framework, les règles de cohé-

rence sont spécifiés comme des invariants dans le langage de contrainte objet [146] et ils sont associés à des méta-classes du méta-modèle UML. Ces méta-classes représentent les différents types d'éléments de modèle UML et les relations entre eux. L'invariant est vérifié par rapport à tous les éléments d'un modèle qui sont des instances de la classe méta-UML associées avec lui.

Enfin, il a été proposé deux techniques plus formelles pour la détection des incohérences. Ce sont la dérivation des «conditions aux limites» par la régression de but et la détection des incohérences à l'aide de modèles de divergences. Ces deux techniques ont été utilisées pour détecter les "divergences" dans les modèles de besoins exprimés dans la langue KAOS [106], [140]. Ce langage préconise une approche axée sur les objectifs de la spécification des exigences. La modélisation d'un système commence par préciser les objectifs que le système doit répondre. KAOS intègre en temps réel un langage logique temporel pour spécifier les objectifs officiellement.

2.3.3.2 La détection basée sur le contrôle de modèle

Les méthodes et les techniques de vérification de modèle déploient, comme leur nom l'indique, les algorithmes spécialisés de vérification de modèle, par exemple SMV [147] et Spin [148]. Les algorithmes de contrôle du modèle ont été initialement développés et révélés d’être un moyen efficace pour détecter les erreurs dans la conception de matériel. Leur utilisation pour vérifier la conformité des modèles de logiciels avec des règles de cohérence a commencé au milieu des années 90.

Le principal problème avec la vérification du modèle se pose avec les systèmes qui ont un nombre non fini d'états. La vérification du modèle est également inefficace en raison de l'explosion des séquences des transitions d'état qui doivent être générés lors de la vérification des règles de cohérence contre les systèmes spécifiés par l'État complexe de diagrammes de transition [149]. Des recherches récentes tentent de résoudre ce problème en utilisant des techniques d'abstraction comme le variable de restriction et le variable d’élimination [150].

2.3.3.3 La détection des incohérences basées sur des formes spécialisées de l'analyse automatisée

De nombreuses méthodes et techniques de gestion de l'incohérence utilisent des moyens spécialisés de vérification de la cohérence des modèles de logiciels [112], [128], [151], [136], [152],

[120], [153]. La plupart de ces techniques vérifient la satisfiabilité de règles de cohérence spécifiques.

La méthode de "réconciliation" développée par Spanoudakis et Finkelstein [136], [152] vérifie la cohérence des modèles UML contre les règles qui exigent le chevauchent total des éléments dans deux modèles UML [146] pour avoir des descriptions identiques. Cette méthode utilise des fonctions de distance pour comparer les éléments du modèle, identifier et quantifier les écarts dans leurs descriptions.

Glintz [151] a développé une technique qui vérifie le comportement des modèles de logiciels exprimés comme des diagrammes d'états pour les blocages, l'accessibilité et l'exclusivité mutuelle des Etats.

Ellmer et autres (1999) ont mis au point une technique de détection des incohérences dans les documents distribués avec le chevauchement du contenu. Les documents représentent des modèles soit de logiciels générés au cours du cycle de vie du développement des systèmes logiciels, ou des documents commerciaux généraux (par exemple les documents financiers). La technique est basée sur le langage de balisage extensible et des technologies connexes et utilise des règles de cohérence pour décrire les relations qui sont nécessaires pour maintenir les documents. Zisman et autres présentent un langage basé sur XML et les technologies connexes pour exprimer ces règles de cohérence. Un générateur de lien de cohérence a été mis au point pour vérifier les différentes règles de cohérence à travers les documents et les éléments participant assimilés.

2.3.3.4 La détection des incohérences basée sur l'exploration collaborative centrée sur l'humain

Un grand nombre des techniques et des méthodes a été développé pour appuyer la gestion des incohérences entre les modèles de logiciel suppose des modèles ou des parties de modèles exprimés dans des langages de modélisation informelles (soutien principalement les formes de texte structuré). Ces techniques comprennent synoptique [110], QARCC [129], DealScribe [126]. Dans ces techniques de détection des incohérences, il est supposé être le résultat d'une collaboration d'inspection des modèles par les parties prenantes. Une approche similaire est également utilisé comme une option dans la technique pour la gestion de divergence et d'obstacle développé par [106] et [140].

La détection des incohérences peut également être laissée aux parties prenantes dont les techniques développées par van Lamsweerde et ses collègues [106], [140]. Notez bien que ces techniques ont été développées pour détecter les incohérences entre les modèles formels, leurs dévelop-

peurs se rendent compte que dans les grands modèles de la détection des incohérences à l'aide de leur méthode de but de régression et leur divergence ou obstacles modèles peut se révéler inefficace. Pour résoudre ce problème, van Lamsweerde et ses collègues ont mis au point un certain nombre d'heuristiques qui pourraient être utilisés par les intervenants pour explorer la possibilité de divergences et les obstacles dans le cadre de certains types d'objectifs. Ainsi, par exemple, dans le cas d'un objectif en ce qui concerne la confidentialité d'un morceau de l'information et un deuxième but en ce qui concerne la fourniture de certaines informations, les intervenants sont invités à vérifier si les éléments particuliers d'informations qui sont référencés par les deux objectifs sont les mêmes. Si elles le sont, les objectifs divergent.

2.3.3.5 Résumé

Un résumé des hypothèses, des principaux aspects positifs et des limites des différentes approches de la détection des incohérences est donné dans le tableau 6.

Approche	Hypothèses	Avantages	Limitations
Basé sur la logique	• Modèles exprimés dans un langage formel	• Procédures bien définies de détection d'incohérence avec la sémantique sonores • Applicable aux règles de cohérence arbitraires	• Logique du premier ordre est semi-décidable • La démonstration des théorèmes est mathématiquement inefficace
Contrôle de modèle	• Il doit être possible d'exprimer ou de traduire les modèles dans un langage particulier orientée État, utilisé par le vérificateur de modèle	• Procédures bien définies de détection d'incohérence avec la sémantique sonores	• Pas efficace en raison de l'explosion des Etats • Seuls certains types de règles de cohérence (par exemple, l'accessibilité des Etats) peuvent être vérifiés
Les formes particulières d'analyse	• Les modèles doivent être exprimées dans un langage commun spécifique (par exemple, les graphes conceptuels, UML, réseaux de Petri, XML) ou être traduits en interne	• Procédures bien définies de détection d'incohérence	• Seuls certains types de règles de cohérence peuvent être vérifiés
Exploration collaborative basée sur l'humain	• Modèles (ou des parties de modèles) ont exprimé dans les langages de modélisation informelles	• Seule méthode pour les modèles informels	• Main-d'œuvre et difficile à utiliser avec les grands modèles

Tableau 6 : Hypothèses, avantages et limitations des approches de la détection des incohérences

2.3.4 Diagnostic d'incohérences

Le diagnostic des incohérences est une activité dont l'objectif est d'établir la source, la cause et l'impact d'une incompatibilité. La plupart des techniques et méthodes de gestion de l'incohérence offrent peu ou pas de soutien pour cette activité. Les exceptions notables à cette règle sont le travail de Hunter et Nuseibeh [141] sur QC-logique et le Framework de diagnostic de l'importance de Spanoudakis et Kassis [142]. La première fournit un mécanisme pour identifier la source d'incohérences détectées dans les modèles de logiciels formelles et celui-ci fournit un Framework configurable pour évaluer l'impact des incohérences décelées dans les modèles UML. Le système de DealScribe développé par Robinson et Pawlowski [126] et la méthode de Vord [154] fournissent également des programmes pour procéder à une évaluation quantitative de l'impact des incohérences décelées dans les modèles des exigences orientées vers un but.

Hunter et Nuseibeh [141] suggèrent que certaines commandes de formules dans les modèles originaux (l'ensemble D) pourraient être utilisées pour commander les différentes sources possibles et donc d'identifier la ressource la plus probable d'une incohérence. L'ordre des formules dans D peut être celle qui reflète la conviction des parties prenantes dans la validité des formules dans cette série. Ils suggèrent également que les étiquettes peuvent être utilisées afin d'identifier les intervenants qui ont fourni les formules. Si tel est le cas alors l'identification de la source possible pourrait également identifier les acteurs impliqués dans l'incohérence. Ces intervenants peuvent ensuite être consultés pour vérifier s'il y a un conflit plus profond qui sous-tend l'incohérence manifeste dans les modèles.

Le Framework de Spanoudakis et Kassis [142] définit un ensemble de «caractéristiques» qui présentent la signification des principaux types d'éléments dans les modèles UML (des classes, des attributs, des opérations, des associations et des messages) et intègrent des fonctions de croyance qui mesurent le degré auquel on peut croire à partir du modèle que l'élément présente une caractéristique. Des exemples de caractéristiques utilisées dans ce Framework sont la «capacité de coordination» d'une classe dans un modèle (dire la capacité d'une classe dans la coordination d'autres classes dans les interactions spécifiques au sein d'un système) et la «dominance fonctionnelle» d'un message (à savoir la capacité d'un message pour déclencher une interaction au sein d'un ensemble de système). Le Framework fournit un langage formel (basé sur OCL) qui peut être utilisé pour spécifier des critères d'importance et les associer à des règles de cohérence spécifiques. Ces critères d'importance sont définies comme des combinaisons logiques des caractéristiques du Framework en utilisant un langage formel appelé S-expressions. Considérons, par exemple, une règle de la cohérence exige que pour chaque message qui apparaît dans une interaction (séquence) schéma d'un modèle UML, doit être une association ou un attribut défini à partir de la classe de l'objet qui envoie le message à la classe des objets qui

reçoit le message (cette condition garantit que le message peut être envoyé). Un critère qui pourrait être définie et associée à cette règle est que le message doit avoir la domination fonctionnelle dans l'interaction qu'il apparaît et la classe de l'objet qui envoie doit avoir une capacité de coordination dans cette interaction. Le cadre dans ce cas serait de calculer les croyances de la satisfiabilité du critère de signification par les messages qui violent la règle et classe les incohérences causées par chacun de ces messages dans l'ordre décroissant des croyances calculées.

Robinson et Pawlowski [126] suggèrent l'utilisation de deux mesures simples comme les estimations de l'impact des déclarations d'exigences contradictoires, à savoir l'exigence de «contention» et «conflit potentiel moyen". Les déclarations des exigences profondes, qui peuvent être exprimées dans DealScribe, sont liés les uns aux autres comme très contradictoires, conflictuels, neutre, porteur et très porteur. L'affirmation d'une déclaration d'exigence est calculée comme le rapport entre le nombre des relations très conflictuelles ou contradictoires sur le nombre total de relations que cette déclaration a avec d'autres énoncés de besoins. Le conflit potentiel moyen d'une déclaration est évalué comme la moyenne des probabilités subjectives de conflit qui ont été associés à toutes les relations conflictuelles et très contradictoires qui ont été revendiqués pour elle. Robinson et Pawlowski affirment que la mesure de contention a été trouvée pour être très efficace dans le classement des exigences contradictoires en termes d'importance et de tenter leur résolution dans l'ordre dérivé. Kotonya et Sommerville [154] dans leur méthode de Vord attendent également les parties prenantes à fournir des poids qui indiquent l'ordre d'importance de leurs modèles d'exigences. Ces poids sont par la suite utilisés pour établir l'importance des conflits entre ces exigences.

Un résumé des hypothèses, des principaux aspects positifs et des limites des différentes approches du diagnostic des incohérences est donné dans le tableau 7.

Approche	Hypothèses	Avantages	Limitations
QC-Logic (Hunter & Nuseibeh, **[141]**)	• Appliqué aux modèles de logiciels officiels • Formules dans les modèles doivent être étiquetés	• Identification automatique des sources possibles d'incohérences	• Coûteux en calcul
S-expressions (Spanoudakis & Kassis, **[142]**)	• Modèles exprimés en UML • Les intervenants précisent les critères pour diagnostiquer l'importance des incohérences	• Distinctions fine de signification sur la base de raisonnement avec une sémantique bien définie • Calculs relativement peu coûteux	• Pas possible de différencier l'importance des violations des différentes règles de cohérence
DealScribe (Robinson & Pawlowski, **[126]**)	• Modèles de besoins exprimés dans un langage propriétaire ciblée • Sur la base de probabilités subjectives de conflits identifiés par les parties prenantes	• Efficace dans le classement des exigences contradictoires	• Évolutivité en raison de la nécessité de fournir des probabilités subjectives de tous les conflits
VORD (Kotonya & Sommerville, **[154]**)	• Modèles d'exigences informelles • Sur la base des poids indiquant l'importance des exigences	• Applicable aux modèles des exigences informelles	• La différenciation d'importance des exigences n'est pas toujours possible

Tableau 7 : Hypothèses, avantages et limitations des techniques de diagnostic des incohérences

2.3.5 Traitement des incohérences

Le traitement des incohérences est préoccupé par les questions de savoir comment faire face à des incohérences, quels sont les impacts et les conséquences de manières spécifiques pour traiter les incohérences, et quand pour faire face à des incohérences. Les incohérences peuvent être manipulées par des actions qui peuvent être prises dans certaines conditions. Les actions qui doivent être prises, dépendent du type d'incompatibilité [155]. Les actions peuvent modifier les modèles, les relations de chevauchement ou les règles pour restaurer, améliorer les incohérences ou informer les parties prenantes sur les incohérences et procéder à des tris d'analyse qui serait sûr d'une motivation supplémentaire à partir des modèles sans cependant changer les modèles, les chevauchements et les règles. Les actions de l'ancien genre sont appelées les actions de changement et les actions de ce dernier type sont appelées les actions de non-changement. Les actions de changement peuvent être divisées en actions de résolution partielles et complètes. Les premières sont des actions qui améliorent les incohérences, mais ne les résolvent pas entièrement. Ces dernières sont des actions qui résolvent les incohérences. Ces deux types d'actions peuvent être automatiquement exécutés ou exécutés seulement si elles sont sélectionnées par les parties prenantes.

2.3.5.1 Actions de changement

La plupart des techniques classées dans ce groupe utilisent l'interaction humaine pour soutenir la manipulation de l'incohérence [110], [128], [136], [106], [123], [141]. Ces techniques attendent les intervenants pour définir et sélectionner les actions de manipulation en évaluant les incohérences, et exécuter ces actions.

2.3.5.2 Actions de non-changement

Les incohérences ne doivent pas nécessairement être résolues ou améliorées (voir [156], [157] pour une perspective similaire en ce qui concerne les incohérences dans les bases de données spécifiées dans la logique temporelle). Les actions qui peuvent être prises en réponse à une incohérence peut être une référence à un utilisateur ou une tentative d'obtenir plus de données ou une sorte d'une analyse complémentaire sur les modèles.

Un résumé des hypothèses, des principaux aspects positifs et des limites des différentes approches de manipulation des incohérences est donné dans le tableau 8.

Approche	Hypothèses	Avantages	Limitations
Synoptique (Easterbrook, **[110]**)	• Les intervenants sont appelés à définir et sélectionner des actions de manipulation	• Liberté complète aux parties prenantes • Soutien pour les modèles informels	• Les incohérences sont traitées par les parties prenantes • Pas de support pour générer des actions de manutention
Système OZ (Robinson & Fickas, **[128]**)	• Utilisation d'ensemble de stratégies de domaines indépendantes pour générer les actions de manipulation	• Identification des actions de résolution	• Interaction humaine élevée
Méthode de réconciliation (Spanoudakis & Finkelstein, **[136]**)	• Utilisez des métriques de distance pour indiquer le type et l'ampleur des incohérences dans la proposition d'actions de manutention	• Les parties prenantes ont le contrôle de l'activité de manutention • Soutien à la résolution partielle • Génération automatique des actions de manipulation	• Pas de support pour l'automatisation de l'application de mesures
KAOS (van Lamsweerde et al. , **[106]**)	• L'utilisation de modèles de résolution de divergence	• Les parties prenantes ont le contrôle de l'activité de manutention • sémantique sonores	• Seuls certains types de divergences peuvent être manipulés
Framework de ViewPoint (Finkelstein et al., **[123]**)	• Les parties prenantes peuvent préciser les actions de manipulation à prendre dans les violations des règles spécifiques	• Automatique partielle de la pleine résolution	• Les parties prenantes ne sont pas en plein contrôle du processus, mais peuvent désactiver des règles de traitement
QC-Logic (Hunter & Nuseibeh, **[141]**)	• Analyse pour identifier les parties au maximum cohérentes des modèles qui sont sûrs de continuer le raisonnement	• Analyse utile en cas de résolutions partielles • Sémantique sonores	• Manuel d'identification des actions connexes • coûteux en calcul

Tableau 8 : Hypothèses, avantages et limitations des techniques de manipulation des incohérences

2.3.6 Localisation

Cette activité concerne l'enregistrement des informations sur le processus de gestion de la cohérence. Les types d'informations qui sont suivis par différentes techniques et les structures utilisées pour le faire sont décrits par [123], [117], [126], [129], [136].

Un résumé des hypothèses, des principaux aspects positifs et des limites des diverses techniques dans le suivi du processus de gestion de l'incohérence est donné dans le Tableau 9.

Approche	Hypothèses	Avantages	Limitations
Framework de ViewPoint (Finkelstein et al., **[123]**)	• Modèles exprimés selon le modèle ViewPoint instancié	• Enregistrement automatique de toutes les informations relatives à la détection et le traitement des incohérences	• Fortement liée au processus de développement de ViewPoint • Peut générer une grande quantité d'informations
conformité aux normes (Emmerich et al, **[117]**)	• Modèles des exigences informels exprimés dans le format propriétaire	• Génère un rapport de tous les éléments du modèle qui violent les règles de cohérence	• Ne pas suivre l'ensemble du processus de gestion de l'incohérence
DealScribe (Robinson & Pawlowski, **[126]**)	• Modèle des exigences orienté but exprimé dans le format propriétaire	• Contrôle automatique de l'histoire de la conformité des modèles	
QARCC (boehm et In, **[129]**)	• Modèles informels	• Fournit une structure pour les conflits d'enregistrement, leur traitement et les options de gestion de rechange envisagées	• Le dossier doit être créé manuellement par les parties prenantes
Méthode de réconciliation (Spanoudakis & Finkelstein, **[136]**)	• Modèles exprimés en UML	• Tient un registre de l'adoption du modèle intégré du processus de réconciliation	• Peut générer une grande quantité d'informations

Tableau 9 : Hypothèses, avantages et limitations des techniques de localisation

2.3.7 Spécification et l'application des politiques de gestion d'incohérence

La politique de gestion de l'incohérence précise les techniques qui pourraient être utilisées pour mener à bien les activités de recouvrement et de détection d'incohérence, le diagnostic et l'évaluation de l'impact des incohérences, et la manipulation des incohérences. La variété des voies possibles qui peuvent être utilisés pour mener à bien chacune de ces activités de gestion de l'incohérence doivent avoir renforcé ce point. Chacun de ces moyens est applicable sous certaines conditions, on travaille mieux dans les paramètres qui satisfont ses propres hypothèses et a ses propres avantages et inconvénients. Ainsi, les acteurs sont confrontés à de nombreuses options à choisir à partir et ont besoin de

conseils pour la prise de décisions mutuellement cohérentes qui répondent le mieux aux objectifs de l'ensemble du processus.

Dans cette section, nous discutons le soutien qui est disponible pour la spécification et le suivi d'une politique de gestion de l'incohérence par les méthodes et les techniques que nous avons interrogées. Plus précisément, nous étudions si les techniques et méthodes examinées :

a) ont une spécification de processus de gestion de la contradiction implicite ou explicite et dans ce dernier cas, comment ce processus est spécifié

b) permettre la modification de leur processus de gestion d'incohérence par les parties prenantes

c) mises en œuvre leurs processus de gestion d'incohérence ou de guider les acteurs dans leur promulgation

d) intègrent leur processus de gestion des incohérences dans un processus plus large de développement de logiciels ou de la méthode

Un résumé des hypothèses, des principaux aspects positifs et des limites des diverses techniques de spécification et d'application d'un processus de gestion de l'incohérence est donné dans le tableau 10.

Approche	Hypothèses	Avantages	Limitations
Framework de ViewPoint (Finkelstein et al., **[123]**)	• Le processus de gestion de l'incohérence fait partie de l'ensemble du processus de développement de ViewPoint • Le processus est spécifié par les règles de la forme *Si <Situation> Alors <action>*	• Modèle explicite et modifiables du processus • Promulgation décentralisée du processus	• Nécessite la compatibilité entre les modèles de processus dans différents modèles de logiciels
conformité aux normes (Emmerich et al, **[117]**)	• Utilisez des modèles de processus explicites appelé «politiques»	• Modèles explicites et modifiables du processus • Couvre la détection d'incohérence, le diagnostic et le traitement	• Processus n'est pas intégré dans un processus plus large de développement de logiciels ou de la méthode
DealScribe (Robinson & Pawlowski, **[126]**)	• Les parties prenantes peuvent définir des objectifs, des actions pour suivre les objectifs, des opérations pour traiter les violations et les conditions pour invoquer les opérations	• Modèles explicites et modifiables du processus • Couvre la détection et le traitement des incohérences	• Processus n'est pas intégré dans un processus plus large de développement de logiciels ou de la méthode
QARCC (boehm et In, **[129]**)	• L'utilisation de processus implicite	• Le processus ne soit pas appliqué • Le processus fait partie du modèle WinWin spirale de développement de logiciels	• modèle implicite
Méthode de réconciliation (Spanoudakis & Finkelstein, **[136]**)	• Spécification explicite du processus de gestion d'incohérence en utilisant la méthode de modélisation de processus logiciel «contextuelle» • L'exécution et le suivi du processus par un moteur de processus d'adoption	• Modèle explicite du processus qui n'est pas appliquée • Couvre l'identification du chevauchement, du diagnostic de détection, et des incohérences de manutention	• Processus n'est pas intégré dans un processus plus large de développement de logiciels ou de la méthode
Système OZ (Robinson & Fickas, **[128]**)	• L'utilisation de processus implicite	• Le processus ne soit pas appliqué • Couvre l'identification de chevauchement, de la détection d'incohérence et de manutention	• Modèle implicite du processus • Processus n'est pas intégré dans un processus plus large de développement de logiciels ou de la méthode

Tableau 10 : Hypothèses, avantages et limitations des techniques du processus de spécification et d'application

2.4 Conclusion

Dans ce chapitre, nous avons présenté une étude sur les travaux de recherche qui ont été menés pour faire face à la problématique de la gestion des incohérences des artefacts logiciels. Cette étude a été présentée selon un cadre conceptuel qui considère la gestion de l'incohérence comme un processus qui intègre des activités de développement pour la détection des chevauchements et des incohérences entre les artefacts logiciels, le diagnostic et le traitement des incohérences, le suivi de l'information, la spécification et le suivi de la manière exacte de réalisation de chacune de ces activités.

Nous avons essayé de surveiller tous les travaux de recherche qui ont été portés à notre attention, identifier des problèmes qui se posent dans le processus de gestion des incohérences et qui ont été traités par ces travaux, qui sont les principales contributions qui ont été faites, et quelles sont les questions qui semblent être ouvertes à de nouvelles recherches pour le moment.

Notre conclusion générale de cette étude est que la gestion de l'incohérence des artefacts logiciels a été très active et continue d'évoluer rapidement dans le domaine du génie logiciel. Les recherches actuelles ont certainement apporté des contributions importantes à la clarification des questions qui se posent dans ce domaine et ont livré des techniques qui répondent à un grand nombre de ces questions. Cependant, il doit être apprécié que la gestion de l’incohérence ne soit pas un problème facile. En effet, il doit faire face à des artefacts logiciels hétérogènes, construites par différents processus de développement de logiciels pour décrire les systèmes dans différents domaines. La plupart des techniques qui ont été développées visent à traiter seulement les artefacts homogènes (même langage et même phase de développement). À cette fin, nous estimons que plus de recherches sont nécessaires pour répondre aux questions que nous avons discutées précédemment.

CHAPITRE 3

Approche de construction et de gestion de cohérence

Objectif

Nous présentons, dans ce chapitre, notre approche que nous avons proposée pour faire face aux différentes problématiques exposées dans cette recherche, et nous décrivons en détail, comment nous avons attaqué les problèmes, quelles méthodes et instruments nous avons employés et comment nous avons procédé aux résultats obtenus.

Sommaire

4.1 Introduction ... 105

4.2 Architecture globale du prototype de validation ... 106

4.3 Le module « artifactsBuilder » ... 109

4.3.1 Liste des taches d'extraction ... 110

4.3.2 Nouvelle tâche d'extraction ... 111

4.4 Le module « activityMonitor » ... 112

4.5 Le module « dataRepopsitoryManager » ... 113

4.6 Le module « consistencyRulesBuilder » ... 114

4.6.1 Liste des règles de cohérence ... 114

4.6.2 Nouvelle règle de cohérence ... 115

4.7 Le module « checkEngine » ... 117

4.8 Conclusion ... 118

3.1 Introduction

En 1995, Michael Jackson a décrit avec précision le génie logiciel en tant que discipline de la description [158]. Les ingénieurs en logiciel utilisent de nombreuses descriptions, y compris les modèles d'analyse, spécifications, conception, code source, guides utilisateur, plans de test, demandes de modification, guides de style, plannings et modèles de processus. Étant donné que ces descriptions peuvent être évoluées dans le temps à travers la participation et la collaboration de nombreux ingénieurs tout au long du processus de développement, la création et le maintien de la cohérence entre les descriptions est un problème difficile pour plusieurs raisons :

- Les descriptions varient considérablement dans leurs formalité et leur précision ; Parfois, des descriptions incomplètes ou imprécises sont utilisées seulement pour mieux comprendre ; à d'autres moments, des descriptions détaillées et précises sont essentielles pour assurer l'exactitude, la sûreté et la sécurité, etc.
- Les descriptions individuelles peuvent être eux-mêmes mal formés ou auto-contradictoire ;
- Les descriptions évoluent tout au long du cycle de vie, et à des rythmes différents
- La vérification de la cohérence d'un grand nombre, arbitraire de descriptions est très coûteuse en termes de ressources informatiques.

En 2000 Bashar et autres Ont utilisé le terme incohérence pour désigner toute situation dans laquelle un ensemble de descriptions n'obéit pas à une relation qui doit tenir entre eux [159]. La relation entre les descriptions peut être exprimée comme une règle de cohérence sur laquelle les descriptions peuvent être vérifiées. Dans la pratique actuelle, certaines règles peuvent être saisies dans les descriptions du processus de développement, d'autres peuvent être intégrés dans les outils de développement. Cependant, la majorité de ces règles ne sont pas saisies n'importe où.

Une approche systématique de la gestion de l'incohérence peut aider à résoudre un ensemble de ces problèmes [159]. L'incohérence attire l'attention sur les zones contenant des problèmes, ce qui signifie que vous pouvez utiliser l'incohérence comme un outil pour :

- améliorer la compréhension commune de l'équipe de développement.
- diriger le processus d'extraction des exigences.
- aider à la vérification et à la validation.

En 2001, Spanoudakis et Zisman ont défini six activités de gestion de l'incohérence qui devraient être entreprises [109]. La première activité qui est la détection de l'incohérence est d'un intérêt particulier, car elle définit le fondement de l'ensemble du processus et elle est menée pour vérifier les violations des règles de cohérence sur les modèles de logiciels. Compte tenu de cette activité, deux familles d'approches ont été identifiées :

- **Approche basée sur la logique** : Elle se caractérise par l'utilisation d'une technique d'inférence formelle pour calculer les incohérences de modèles.

- **Approche de vérification de modèle** : déploie des algorithmes dédiés de vérification des modèles qui sont bien adaptés pour détecter les incohérences de comportement spécifiques, mais ne sont pas bien adaptés à d'autres types d'incohérences.

L'approche dite de « gestion de la cohérence » considère qu'il est impossible d'assurer la cohérence globale de tous les artefacts logiciels à tout moment. N'importe quel artefact peut être temporairement incohérent. Les incohérences temporaires seront corrigées le moment venu par l'équipe de développement. La principale problématique de cette approche réside dans le suivi des incohérences. Il faut en effet pouvoir détecter l'introduction de nouvelles incohérences ainsi que la suppression d'incohérences existantes lors des modifications successives faites par les développeurs sur les artefacts ; et ce, sans gêner le déroulement du processus de développement.

Il est important de souligner le fait que ces approches actuelles ne permettent en général que de travailler sur des artefacts homogènes (documents spécifiques d'objets modèles UML par exemple) [15], [26], ou utilisent des formats pivots (comme XML) pour masquer l'hétérogénéité [18]. En outre, ils ne peuvent généralement pas faire face à l'évolution des différents artefacts hétérogènes.

Problème

Un logiciel est composé d'une variété d'objets, qui sont des produits de différentes activités impliquées dans le processus de développement. Dans la pratique, les artefacts logiciels évoluent à des rythmes différents et des modifications appliquées à un artefact ne peuvent pas nécessairement se répercuter sur les autres objets connexes.

Conséquence

Cette évolution différentielle des artefacts logiciels peut entraîner des problèmes de synchronisation et d'incohérence entre les objets. Les objets obsolètes constituent également des obstacles à la maintenance d'une évolution efficace d'un système logiciel. Bien que le développement incrémental fournis une solution plus souple pour le traitement des changements [160], la cohérence des artefacts

est souvent négligée : les différentes représentations de logiciels passent par des étapes de raffinement sans tenir compte de tous les objets dépendants, comme le processus qui n'applique pas les liens d'artefacts.

Solution

Nous avons conçu et construit CMAC, un outil qui répond aux différentes questions en utilisant un mécanisme basé sur les traces de construction des artefacts logiciels. CMAC détecte la présence ou l'absence d'incohérence entre les artefacts. Les incohérences sont spécifiées par des règles logiques portant sur les traces de construction. Cet outil permet la définition de règles d'incohérence dites méthodologiques offrant des gains de performances très intéressants. L'outil est conçu pour notifier les développeurs quand les artefacts deviennent non synchronisés. Notre objectif est de fournir aux développeurs les informations nécessaires pour veiller à ce que tous les artefacts d'un système logiciel sont cohérents et complètes tout au long du processus de développement.

La meilleure façon de construire un tel système serait d'avoir une définition formelle pour tous types d’artefacts à l'aide d'une représentation unifiée, puis d'assurer la cohérence logique entre ces artefacts. La diversité des types d'artefacts (les exigences, les spécifications, et même la conception des artefacts et les cas de test) rend cette tâche difficile, voire impossible d’assurer la cohérence générale et significative entre tous les artefacts.

Notre outil CMAC fonctionne en plusieurs phases :

1. Il permet aux développeurs de déclarer les types d’artefacts concernés par l’extraction.
2. Il extrait les informations pertinentes du logiciel en se basant sur les types d’artefacts déclarés par les développeurs et puis, les stocker dans une base de données.
3. Il permet aux développeurs de renseigner les règles de cohérence entre les artefacts mémorisés.
4. Il utilise les informations enregistrées dans la base de données (Artefacts et règles de cohérence) pour vérifier la validité de chacune de ces règles.
5. Il présente les résultats de ces vérifications aux développeurs afin qu'ils puissent résoudre les incohérences.

L’objectif de ce chapitre est de présenter notre approche que nous avons proposée pour faire face aux différentes problématiques exposées dans cette recherche, et de décrire en détail, comment nous avons attaqué les problèmes, quelles méthodes et instruments nous avons employé et comment

nous avons procédé aux résultats obtenus.

Le chapitre comporte deux grandes sections, la première section expose notre approche de construction des artefacts, tandis que la deuxième présente notre mécanisme de gestion d'incohérence.

3.2 Travaux connexes

Diverses solutions ont été proposées dans les domaines de la gestion de la cohérence et de la traçabilité, qui aident l'analyse de l''impact des changements [161], pour soutenir l'évolution cohérente des artefacts logiciels [162] [163], [164], [165], [109]. Cependant, un certain nombre de défis reste à relever, c'est que les approches actuelles ne fournissent que des solutions partielles: elles sont restrictives en termes d'artefacts pris en charge et ils ne répondent pas à tous les aspects de la cohérence des artefacts.

Conceptuellement, l'approche la plus simple pour assurer la cohérence des différents aspects de logiciel est de les combiner dans un seul langage de programmation. Plusieurs environnements tels que l'environnement de Xerox Cedar Mesa [166] et Common Lisp [167] ont combiné la documentation avec le code. Ces efforts ont abouti à la programmation littéraire [168], [169] et, plus récemment, à l'utilisation de javadoc et ses conventions correspondantes. Les environnements comme Visual Studio combinent le code et la conception de l'interface utilisateur. Les partisans de l'UML proposent d'écrire des systèmes complets dans leurs Framework, ce qui en fait un langage de programmation qui combine la conception avec le code. Batory [170] soulève cette idée au niveau des modules qui encapsulent le code, la documentation et d'autres dimensions ; toutefois, ceux-ci doivent tous composer à travers le même mécanisme. Ce n'est pas seulement très restrictif, mais on ne sait pas, par exemple, comment composer le texte de la même façon que nous composons le code. D'autres travaux récents se penchent sur l'impact de l'évolution de code, mais ne tiennent pas compte des autres dimensions [171].

Il y a un certain nombre de systèmes qui vérifient la cohérence des aspects uniques du système logiciels. Lint [172] et les successeurs tels que CCEL [173] et LCLint [174] effectuent la vérification statique des programmes. Les vérificateurs de styles tels que la suite d'outils de *Parasoft*[9] ou le

9 http://www.parasoft.com/products.jsp

projet de *checkstyle*[10] effectuent la vérification de style et de convention des programmes. Les systèmes tels que *ViewIntegra* [175] et *xlinkit* [176] ont été utilisés pour vérifier la cohérence des diagrammes UML. Il y a aussi un large éventail d'outils pour faire la couverture de test, des langues comme *Eiffel*[11] qui inclut les spécifications vérifiables dans le code, et des systèmes tels que *Flavors* [177] qui font la vérification statique des spécifications externes. Les environnements comme *Eclipse*[12] et *NetBeans*[13] fournissent un langage de style et de contrôle d'utilisation, y compris la vérification des documents de la source.

Le travail le plus proche de ces derniers à la nôtre est l'approche de *xlinkit* [19] appliquée au génie logiciel. *Xlinkit* fournit la capacité générale de vérifier la cohérence de plusieurs documents XML. Les documents XML peuvent être spécifiés soit directement ou peuvent être obtenus à partir d'autres artefacts. Les contraintes utilisent un langage de requête XML basé sur *XPath*[14] et *XLink*[15]. Le système actuel est capable de traiter de très gros documents à l'aide d'une représentation sur le disque et il est capable de faire un contrôle incrémental limité de contraintes en surveillant les parties de l'arbre XML qui ont changé.

Notre approche est différente à plusieurs égards. **Tout d'abord**, plutôt que d'utiliser XML et *XPath*, nous utilisons un Framework et des requêtes SQL relationnelles. Cela fournit un langage de requête plus puissant et élimine la nécessité de traiter des documents volumineux. **Deuxièmement**, notre système présente une approche semi-automatique d’expression des artefacts hétérogènes et permet aux responsables de l’évolution de spécifier une seule fois les niveaux des artefacts concernés par l’extraction au lieu de passer sur tous les documents du logiciel et obliger les développeurs à chaque fois de convertir les documents en XML pour vérifier leurs cohérences. *Xlinkit* exige à chaque modification la génération des fichiers XML et la mise à jour incrémentale des contraintes au niveau des règles de cohérences. **Troisièmement**, notre système gère une gamme plus large des artefacts logiciels voir tous les types des artefacts, à la fois statiques et dynamiques. Par exemple, nous traitons les cas de test et de la couverture, des diagrammes d'interaction UML et spécifications comportementales. **Enfin**, notre système fonctionne dans les environnements de programmation existants, outils de programmation existants, et les méthodologies existantes, et ce, sans aucune intervention du programmateur.

10 http://checkstyle.sourceforge.net
11 https://www.eiffel.com
12 https://www.eclipse.org
13 https://netbeans.org
14 http://www.w3schools.com/xpath
15 http://www.w3.org/TR/xlink

3.3 Vue d'ensemble de l'approche

Le travail proposé comprend la conception, la mise en œuvre et l'évaluation d'une plateforme pour la gestion de la cohérence des artefacts logiciels hétérogènes. La nouveauté de la solution provient de son approche uniforme et extensible d'extraction de tous types d'artefacts vers une représentation unifiée dans une base de données pour résoudre la question de la cohérence des artefacts.

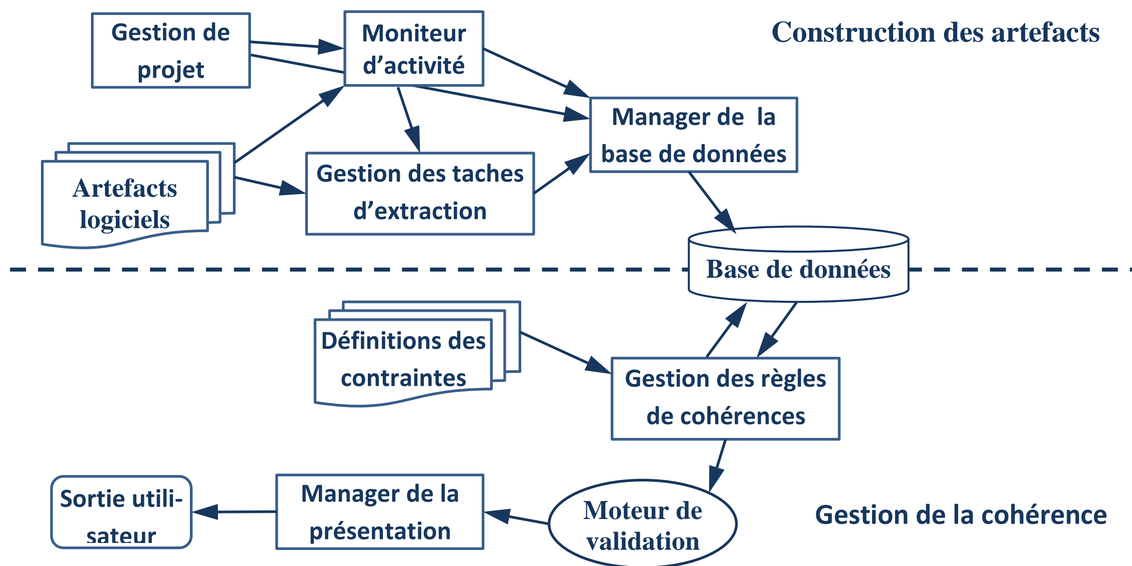

Figure 5 : Architecture fonctionnelle de l'approche

La plateforme globale est constituée des différents composants présentés dans la figure 5. Ces composants peuvent être divisés en deux parties : la première partie gère l'extraction des informations nécessaires auprès des artefacts source tandis que la seconde partie utilise cette information pour évaluer, valider les règles de cohérences et afficher des informations sur les résultats de cette validation à l'utilisateur final.

Les artefacts sont extraits par un ensemble des taches d'extractions. La création des taches d'extraction passe par deux étapes : définir le niveau des artefacts par type de fichier concerné par l'extraction (par l'utilisateur), lier le niveau sélectionné avec un outil d'extraction existant ou bien avec un nouveau (créé par l'utilisateur).

Il y avait plusieurs problèmes qui devaient être résolus ici. Il s'agit notamment :

- **Problème 1 :** *Quels sont les artefacts associés à un projet de logiciel particulier ?* Pour répondre à cette question, nous avons fourni un gestionnaire de projet qui permet à l'utilisateur de définir un projet en termes de répertoires et les fichiers, en spécifiant le chemin d'accès associé.

- **Problème 2 :** *Quelles sont les informations requis dans les artefacts ?* Cela a nécessité une compréhension de la sémantique souvent vague des artefacts logiciels. Ici, nous n'enregistrons que le minimum d'information par défaut, et nous permettons aux utilisateurs d'ajouter autant d'informations requis en suivant un mécanisme évolutif et simple par niveau et type de fichier.

- **Problème 3 :** *Comment relier l'information entre les différents extracteurs, différentes dimensions, et des moments différents ?* pour répondre à cette question nous avons mis en place un méta-modèle unifié de représentation et d'enregistrement des différents types d'artefacts à travers un gestionnaire de base de données partagé.

- **Problème 4 :** *Comment faire face au problème d'hétérogénéité ? comment unifier l'information extraite depuis plusieurs types d'artefacts ?* Nous avons développé des outils d'extraction spécifiques par niveau et type de fichier, et pour les artefacts plus complexes tels que les artefacts composés, les fichiers textes et les documents de spécification des règles de métiers, nous avons adopté une approche qui consiste à les définir de façon manuelle par l'expert chargé de l'évolution.

Après avoir des artefacts unifiés, recueillis et déposés dans la base de données, l'outil utilise ces artefacts pour évaluer et valider les règles de cohérences avant d'afficher les résultats à l'utilisateur final. Ce travail est effectué par trois composants, un gestionnaire de règles de cohérence qui est en charge de créer, mettre à jour et stocker les règles dans la base de données; un moteur de validation qui utilise les informations enregistrées précédemment pour valider les règles de cohérence sur les artefacts ; et un gestionnaire de présentation qui utilise les résultats du moteur de validation pour transmettre les informations appropriées pour le développeur.

Encore une fois, il y avait plusieurs problèmes clés qui doivent être abordés afin de faire ce travail efficacement. Il s'agit notamment :

- **Problème 5 :** *Comment définir des règles de cohérence ?* Nous avons choisi de définir les règles sous forme des opérations logiques entres les valeurs des artefacts déjà extraits dans la base de données. Pour des règles simples (celle définit entre deux artefacts par exemple), nous avons proposé de les définir manuellement à travers une interface utilisa-

teur, et pour des règles plus complexes les développeurs doivent passer par l'option du langage à travers la même interface proposée.

- **Problème 6 :** *Comment détecter et vérifier l'incohérence ?* Nous avons développé dans le cadre de la validation de notre approche un moteur de validation des règles de cohérence enregistrées dans la base de données. Notre approche consiste à convertir les différentes règles enregistrées à des traitements en langage java, et en sortie de l'exécution le programme mis à jours l'historique des états de chaque règle et affiche les messages spécifiques en cas d'incohérence.

3.4 La construction des artefacts logiciels hétérogènes

Plusieurs approches de construction des artefacts logiciels ont été mises en œuvre. Elles utilisent généralement des systèmes de requêtes sur des graphes et des systèmes de réécritures de graphes. Depuis plusieurs décennies déjà les compilateurs ont utilisé les structures d'arbres pour représenter les programmes. Cependant, les arbres sont limités en matière de représentation de différents artefacts logiciels. Le concept de graphe, plus particulièrement celui des graphes typés et attribués [178], est un moyen générique, semi-automatique, et plus adéquat que les arbres pour représenter la structure des artefacts ainsi que des informations supplémentaires les concernant (à travers les attributs).

3.4.1 Les artefacts logiciels

2.3.1.1 La définition des artefacts logiciels

Un artefact est un élément d'information qui est produit, modifié ou utilisé par un processus. Les artefacts sont les produits tangibles du projet, qui sont créés où utilisés, tout en travaillant vers le produit final. Les artefacts sont utilisés comme entrée par les développeurs pour effectuer une activité; et elles sont le résultat ou la sortie de ces activités [179]. Les artefacts peuvent prendre diverses figures ou formes. Le tableau 11 liste quelques types d'artefacts communs avec des exemples.

Type d'artefact	Exemples
Document	Les cas d'affaire, le document d'architecture logicielle, les spécifications des exigences du logiciel (SRS), la description de la conception de logiciel (SDD), le plan de test, etc.
Diagrammes	Le diagramme de séquence, le diagramme d'état, le diagramme d'activité, le diagramme de collaboration, le modèle entité-relation, le diagramme des classes, les cas d'utilisation, etc.
Code	Le code source, les scripts de la base de données, les fichiers batch, les pages web, les fichiers de configuration XML, etc.
Exécutable	Les programmes, les scripts, les plugins.
Résultats	Les résultats des tests fonctionnels, les résultats des tests unitaires, le rapport des anomalies, etc.
Autres	Les données, les prototypes, etc.

Tableau 11 : Artefacts logiciels communs

Les systèmes logiciels sont développés selon des processus par phases, dans lesquels la complexité de génie logiciel est abordée par le biais des activités de raffinement ultérieurs [180]. Ces phases guident le développement de sa première conception à la réalisation et à la maintenance [181]. Chaque phase de développement du logiciel produit plusieurs artefacts. Par exemple le document des spécifications des exigences du logiciel (SRS) est créé au cours de la phase d'analyse des besoins ; la description de la conception du logiciel (SDD), les cas d'utilisation, les diagrammes de séquence, entre autres sont créés pendant la phase de conception ; et le code source, les fichiers de configuration, les exécutables et la documentation de l'API lors de la phase de la mise en œuvre. Ces artefacts sont étroitement liés comme le montre la figure 6. L'artefact de résultat d'une phase est l'entrée pour la phase suivante. Toutefois, un processus par phases ne garantit pas la traçabilité de comment les exigences évoluent dans la conception et la conception dans le code [180] et ni la cohérence des artefacts créés dans chaque phase.

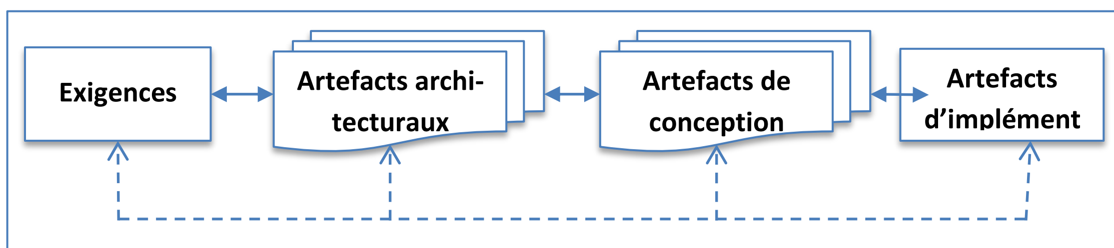

Figure 6 : Relation entre les artefacts logiciels

La figure 6 montre la relation entre artefacts. Les lignes continues représentent une relation

directe entre les artefacts. La ligne pointillée montre la relation indirecte entre les artefacts. Lorsqu'en raison de l'évolution des circonstances, le logiciel a besoin d'être adapté, la relation entre les besoins et la conception ou la conception et le code peut facilement se briser et les artefacts deviennent incompatibles. Par exemple, une exigence spécifique est convertie en une conception architecturale d'artefact. Ensuite, un élément de programme (morceau de code source) satisfait cette décision de conception et donc l'exigence spécifique. Si une exigence nécessite un petit changement, il doit probablement se refléter dans l'artefact de conception et dans l'élément de programme. Une situation similaire se produit quand un problème est détecté et corrigé dans le code. Cette action peut affecter la conception architecturale provoquant que les besoins satisfaits par la décision de conception peuvent ne pas être remplis.

4.3.1.1 La classification des artefacts logiciels

La classification ou hiérarchisation des artefacts procède d'une stratégie de simplification du processus d'extraction des artefacts. Nous retenons deux principaux critères de classification nous permettant ainsi de définir deux niveaux (ou dimensions). Ce sont les niveaux d'abstraction et de granularité [182]. Il est important de noter que des artefacts peuvent avoir le même niveau d'abstraction sans avoir le même niveau de granularité. Dans le cadre du paradigme objet, par exemple, une variable d'instance (champ d'une classe) et une méthode ont le même niveau d'abstraction mais pas le même niveau de granularité (une méthode de granularité plus "grosse" qu'une variable d'instance). De la même manière des artefacts peuvent avoir le même niveau de granularité mais des niveaux d'abstraction différents. Ainsi, une classe a le même niveau de granularité qu'un template ou patron de classe mais pas le même niveau d'abstraction (un template a un niveau d'abstraction supérieur à celui d'une classe).

Pour une meilleure compréhension des applications logicielles, nous avons considéré que tous les artefacts, quel que soit leur type, issus des différents types de fichiers et sont assimilables à des graphes typés [178]. Un Artefact est alors composé d'éléments (artefacts). Chaque élément est typé. Il peut porter des valeurs pour des attributs et peut référencer d'autres éléments. Toutes les attributs et les références sont elles aussi typées. Les artefacts sont liés entre eux à des niveaux abstro-granulaires différents [182]. L'ensemble des artefacts peuvent être représentés selon la hiérarchisation en niveaux par le triple $< \sum_{Ft}, \sum_{Lv}, \sum_{art} >$ où $\sum_{Ft}$ et l'ensemble de tous les types de fichiers, $\sum_{Lv}$ est l'ensemble de tous les niveaux. Le $k^{ème}$ artefact du $j^{ème}$ niveau du $i^{ème}$ type de fichier est représenté par le triple $<Ft_i, Lv_j, art_k>$. Par exemple, $<Ft_{Java}, Lv_{Class},$ *Calculator* $>$ dénote le fait que *Calculator* est un artefact appartenant au niveau classes dans un fichier de type Java.

Modéliser les relations inter-artefacts est une tâche complexe et très importante. Nous considérons trois types de relations (figure 7) [43]. Ce sont :

1. **Les relations inter-fichiers :** ces relations relient des artefacts appartenant à deux fichiers différents du système. C'est le cas par exemple de la relation entre une classe UML et une classe Java qui l'implémente ou bien la relation entre classe java et le descripteur de déploiement (fichier xml).

2. **Les relations horizontales :** elles représentent différentes sortes de liens sémantiques appartenant au même fichier et qui relient des artefacts d'un même niveau abstro-granulaires. C'est notamment le cas de la relation d'appel entre deux méthodes ou de la relation d'héritage entre deux classes, etc.

3. **Les relations verticales :** elles relient deux artefacts appartenant au même fichier et à des niveaux abstro-granulaires différents. Un exemple de ce type de relation est celle qui relie une classe à ces attributs, une méthode à son corps ou un bloc aux instructions qui le composent, etc.

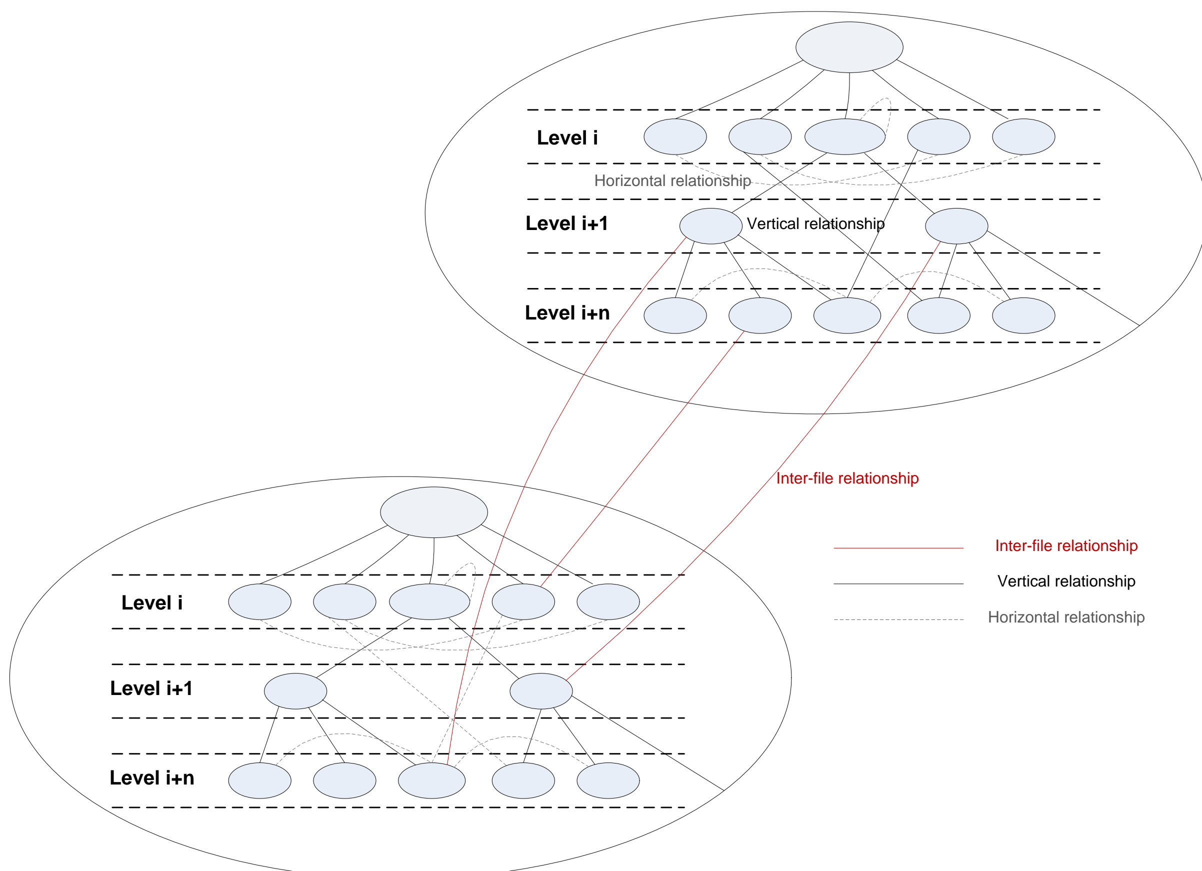

Figure 7 : Classification des Artefacts logiciels par type de fichier

3.4.2 La surveillance des changements

Les artefacts sont extraits des plusieurs types de fichiers dans les logiciels en utilisant un réseau de différents outils d'extraction spécifiques par type de fichier et niveau abstro-granulaire des artefacts. La composante « *Moniteur d'activité* » détermine périodiquement si des fichiers ont changé et puis exécute automatiquement les taches d'extraction appropriées ou bien enregistre l'information dans la base de données en attendant le planificateur des taches (dépend du paramétrage).

Des changements sont détectés et mis à jour au niveau du fichier. Autrement dit, lorsque le moniteur d'activité détecte que le fichier représentant un ensemble d'artefacts particuliers a changé, toutes les informations de tous les artefacts logiciels appartenant à ce fichier, doivent être mises à jour dans le système. Le niveau de fichier semble être le plus approprié pour la mise à jour de ces informations, car il est facile de détecter le changement d'un fichier et plus facile d'écrire des extracteurs qui travaillent sur un fichier tout à la fois. Bien qu'il soit plus commun pour seulement une petite quantité d'informations dans le fichier est réellement changée, de détecter et extraire touts les renseignements tout en maintenant la cohérence avec le reste de la base de données aurait fait tous les extracteurs beaucoup plus complexes et plus lents. Nous avons opté à la place de mettre la charge de la détermination de la mise à jour incrémentale réelle dans le gestionnaire de base de données et de garder les extracteurs simple et rapide.

3.4.3 Le Méta-model de représentation unifié des artefacts

La plateforme que nous avons développée analyse et extrait des artefacts hétérogènes par différentes sortes d'extracteurs que nous avons définis par type de fichier et niveau d'artefact [42]. Chacun de ces extracteurs permet d'enregistrer les artefacts identifiés de façon homogène et unifiée dans le méta-model U2MHA proposé dans la figure 8 [43].

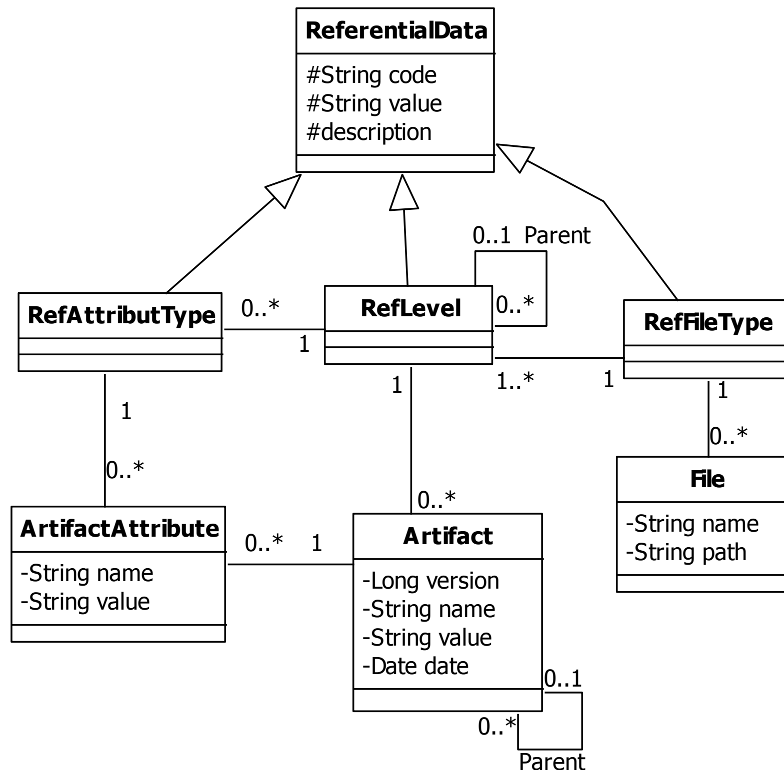

Figure 8 : U2MHA : Méta-model unifié des artefacts hétérogènes

Nous avons essayé de proposer un méta-model simple et évolutif pour représenter tous types d'artefacts. Nous avons considéré que tous les artefacts sont enregistrés dans les fichiers. Chaque type de fichiers doit avoir des niveaux abstro-granulaire hiérarchiques (exemple du fichier java, niveau classe qui contient des méthodes dont on peut trouver des paramètres etc). Les artefacts peuvent être représentés aussi d'une manière hiérarchique, un artefact peut avoir des éléments (artefacts), chaque élément peut avoir d'autres éléments (artefacts), ainsi de suite. Chaque artefact appartient à un niveau et peut avoir d'autres attributs par niveau qui peuvent être définis manuellement par les développeurs.

Des algorithmes spécifiques ont été définis par type de fichier et niveau d'artefacts et d'autres ajoutés manuellement par les experts en suivant l'approche adoptée. Des listeners[16] adaptés par niveau sont à l'écoute des différentes opérations de construction (création, modification, suppression) afin de suivre l'évolution de chaque artefact dans le méta-modèle de construction U2MHApour un meilleur contrôle de l'évolution du logiciel.

Les résultats de chacun des extracteurs sont une liste de commandes au gestionnaire de base de données. Le gestionnaire de base de données elle-même a des principales responsabilités. Il doit

16 **http://en.wikipedia.org/wiki/Listener**

d'abord traiter progressivement les commandes issues des différents extracteurs d'information, ajouter et supprimer des lignes dans la base de données. Deuxièmement, gérer la qualité et la performance des traitements, voire supprimer toutes les anciennes informations associées à un artefact particulier et les remplacer par les nouvelles, etc.

3.4.4 La gestion des taches d’extraction

Il est important de souligner le fait que les approches actuelles proposent d’extraire les données soit en parcourant la totalité des artefacts avec un écoute sur les opérations de construction des développeurs puis les stocker dans une base de données [26], ou bien à travers la conversion de tous les artefacts en XML avant d’appliquer les règles de cohérences [18]. L’extraction incrémentale de tous les artefacts ainsi que l’écoute sur les différentes opérations de construction est un processus très lourd et exigeant en termes de ressources informatique pour l’installer dans un environnement de développement. Dans le cas de notre approche et dans le cadre d’améliorer la performance de notre plateforme, nous proposons un mécanisme d’extraction permettant aux utilisateurs qui sont souvent les développeurs du système logiciel de gérer manuellement les taches d’extraction (ajout, suppression, modification et lancement) afin de sortir l’information utile et au moment choisi.

Une tâche d’extraction représente un enregistrement des paramètres d’extraction (une ligne dans la base de données) des artefacts par type de fichier et niveau abstro-granulaire, effectué par les développeurs à travers l’application web développée (formulaire) qui fait partie de notre plateforme. Une tache peut être démarrée manuellement par l’utilisateur de l’application (Action bouton) comme elle peut être lancée à travers des évènements en écoute (listeners) sur les opérations de construction des développeurs ou bien sous forme des batch programmés. Ces options sont à disposition des développeurs à travers l’interface d’administration de l’application web.

La figure 9 présente le processus de création et de mise à jour des taches d’extraction.

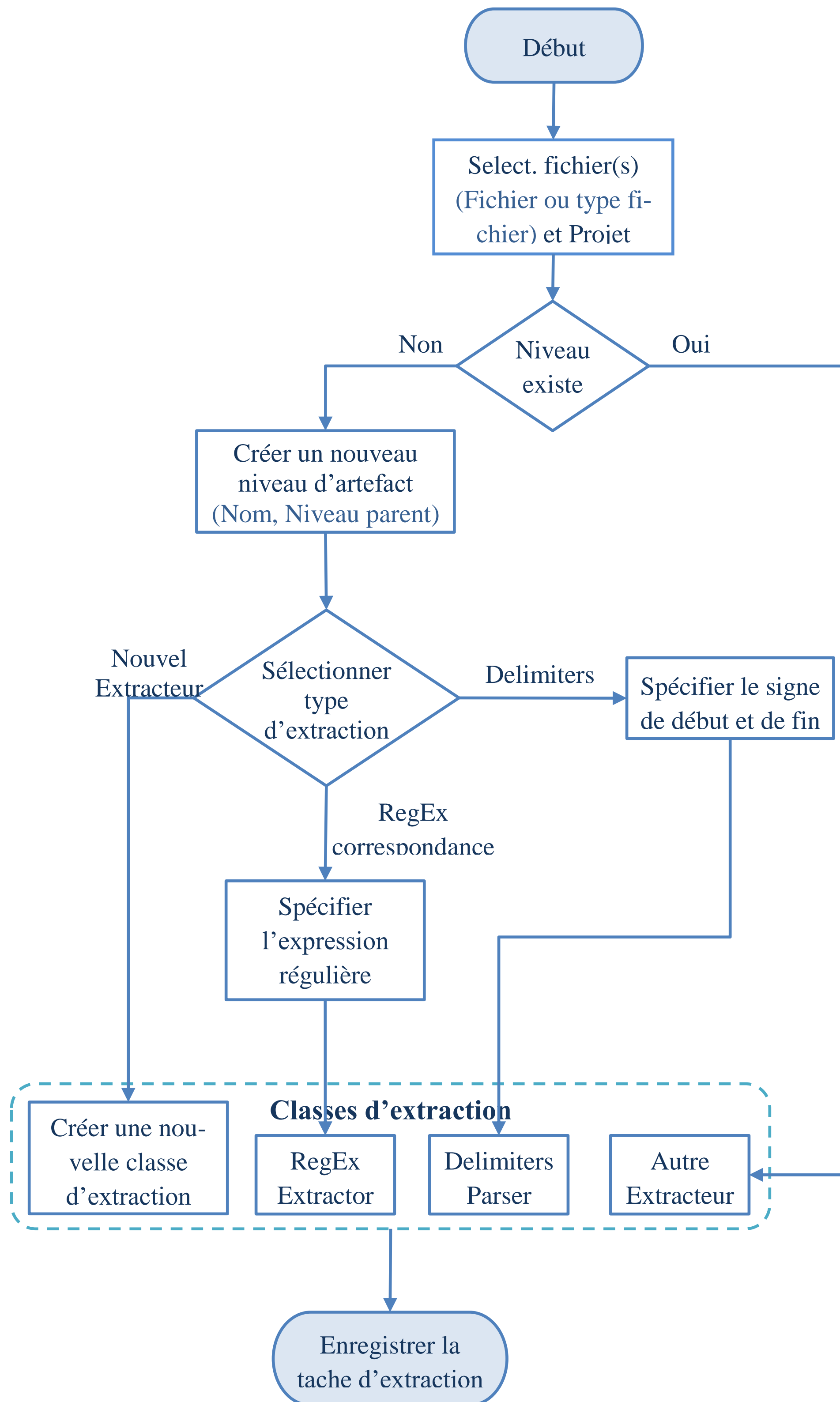

Figure 9 : Processus de création des taches d'extraction

Notre plateforme web offre aux développeurs des écrans interactifs pour gérer les taches

d'extraction des artefacts (ajout, modification, suppression et lancement). L'utilisateur doit choisir les fichier(s) désignés par l'extraction (un fichier spécifique ou bien plusieurs par type de fichier), puis il peut choisir un niveau abstro-granulaire existant (utilisant des classes d'extraction livrées avec l'application) ou bien créer un nouveau. Pour créer un nouveau niveau abstro-granulaire l'utilisateur doit choisir parmi les trois types d'extraction :

- **Par les délimiteurs :** permet l'extraction des artefacts encadrés par les signes (tokens) de début et de fin spécifiés par l'utilisateur, ce type d'extraction utilise une classe générique « *DelimitersExtractor* ».
- **RegEx** : pour l'extraction des artefacts respectant l'expression régulière spécifiée par l'utilisateur, ce type d'extraction utilise une classe générique « *RegExExtractor* ».
- **Nouvelle classe d'extraction** : Donne la possibilité aux développeurs de mentionner une nouvelle classe d'extraction spécifique à leur besoin. Pour les artefacts plus complexes, nous adoptons une approche qui permet aux développeurs de créer une nouvelle classe d'extraction spécifique et de mentionner son nom lors de la création d'un nouveau niveau abstro-granulaire, cette nouvelle classe d'exatraction doit hériter de la classe générique des extracteurs appelé « *AbstractExtractor* » et après il suffit de redéfinir et personnaliser la méthode d'extraction générique « *extractArtefacts* » (voir la figure 10).

```
public class NewExtractor extends AbstractExtractor {

    /**
     * Constructeur par défaut
     */
    public NewExtractor() {
    }

    /**
     * {@inheritDoc}
     */
    public List<Artefact> extractArtefacts(File pFile) {
        // Liste des artefacts à retourner
        List<Artefact> result = null;

        // 1. Parser librement le fichier

        // 2. Alimenter la liste des artefacts à retourner

        // 3. Retourner le résultat
        return result;
    }
}
```

Figure 10 : Exemple du code java d'une nouvelle classe d'extraction

La figure 11 représente le méta-model EJ2M de la gestion des taches d'extractions créées par

les utilisateurs à travers l'application web.

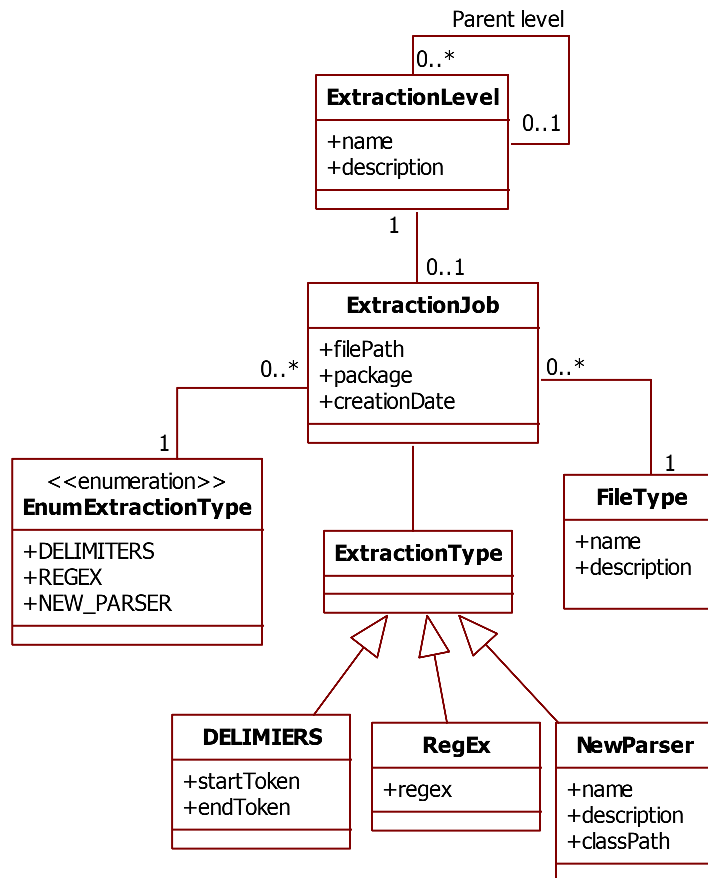

Figure 11 : EJ2M : Méta-model des taches d'extraction

Le résultat du lancement des taches créées se traduit par une représentation unifiée de tous les artefacts logiciels demandés dans un référentiel homogène (Figure 8). Tout ceci s'effectue bien entendu à travers une application web extensible développée au sein de la plateforme Eclipse. En d'autres termes le développeur dispose d'une représentation uniforme de tous les artefacts hétérogènes sans quitter son environnement de travail habituel et il est donc possible de mener en même temps des évolutions sur la plateforme et une activité d'analyse dans le but d'améliorer la qualité et la cohérence des données tout au long du processus de développement. Notre approche permet également aux utilisateurs d'ajouter des nouveaux parseurs pour prendre en charge des fichiers plus complexes et de personnaliser l'extraction par l'ajout et la modification des taches traitant d'autres types de fichiers pour une gestion plus spécifique de l'extraction.

Prenant l'exemple suivant, une class java « Calculator.java » (Figure 12) est référencé dans le fichier de déploiement « app-context.xml » (Figure 13), ce fichier de déploiement contient aussi des

références vers des paramètres dans un fichier des propriétés « build.properties » (Figure 14).

```java
package ma.organization.calculation;

public class Calculator {
    private int operand1;
    private int operand2;

    public Calculator(int pOperand1, int pOperand2) {
        this.operand1 = pOperand1;
        this.operand2 = pOperand2;
    }
    // ....
}
```

Figure 12 : Exemple du fichier java "Calculator.java"

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <bean id="calculator" class="ma.organization.calculation.Calculator" >
        <constructor-arg value="${organization.setting.operand1}" />
        <constructor-arg value="${organization.setting.operand2}" />
    </bean>
</beans>
```

Figure 13 : Exemple u fichier XML "app-context.xml"

```
# Default parameters
organization.setting.operand1=25
organization.setting.operand2=75
```

Figure 14 : Exemple du fichier properties "build.properties"

Supposant que l'ingénieur de développement veut superviser la cohérence entre :

- La déclaration des beans dans le descripteur de déploiement et l'existence de ces classes déclarées dans le même chemin spécifié.
- Les attributs des beans déclarés dans le descripteur de déploiement et les attributs de la classe elle même
- Les références utilisées dans le fichier de déploiement et l'existence de ces paramètres dans le fichier des propriétés.

Les tableaux (Tableau 12) et (Tableau 13) représentent respectivement les différents niveaux et artefacts qui peuvent êtres extraits de l'exemple précédent afin de répondre aux règles de cohérences.

Value	Parent	File
package		java
javaclass	package	java
attribut	javaclass	java
constructor	javaclass	java
parameter	construc-tor	java
xmlbean		xml
xmlclass	bean	xml
constructor-arg	bean	Xml
paramKey		properties
paramValue	paramKey	properties

Tableau 12 : Les niveaux abstro-granulaire de l'exemple

Value	Parent	RefLevel
ma.organization.calculation		package
Calculator	package	javaclass
operand1	Class	attribut
Operand2	Class	attribut
Calculator	Class	constructor
pOperand1	Calculator	parameter
pOperand2	Calculator	parameter
bean		xmlbean
ma.organization.calculation.Calculator	bean	xmlclass
${organization.setting.operand1}	bean	Constructor-arg
${organization.setting.operand1}	bean	Constructor-arg
organization.setting.operand1		paramKey
organization.setting.operand1		paramValue

Tableau 13 : Les artefacts extraits de l'exemple

3.5 La gestion de la cohérence des artefacts logiciels

La gestion de la cohérence entre les différents artefacts en génie logiciel a été reconnue comme essentiel pour de nombreuses années [183], [184], [185]. Dans l'ingénierie des exigences, la

gestion de la cohérence entre les spécifications des exigences formelles et les modèles d'architecture et de design a été étudiée [175], [186]. De même, plusieurs approches ont été développées pour tenter de déterminer les incohérences entre les descriptions en langage naturel d'exigences et de modèles formalisés des besoins [183], [187]. Certaines techniques ont été développées pour soutenir la correction des incohérences telles que l'utilisation des opérations de réparation [188]. Les incohérences détectées peuvent parfois nécessiter une correction immédiate. Vivre avec l'incohérence permet la gestion des incohérences dans le temps et offre plus de souplesse dans le processus de développement [155]. La correction des incohérences et le soutien de l'outil approprié pour détecter l'incohérence sont difficiles à gérer [189].

Clairement, la gestion de l'incohérence est une activité à multiples facettes qui est de plus en plus reconnue comme essentielle à la bonne gestion de développement de logiciels. En fait, on pourrait dire qu'en réalité, les praticiens du développement logiciel vivent et travaillent avec des informations inconsistantes sur une base quotidienne. Une approche est donc nécessaire au sein de laquelle une variété d'activités de gestion de l'incohérence peut être effectuée. Une telle approche devrait faciliter l'interfonctionnement d'une combinaison, normalement isolés, des techniques, des mécanismes et des outils de détection, d'identification (diagnostic / classification) et la manipulation des incohérences.

Nous proposons une approche fondée sur la logique de la gestion de l'incohérence. En résumé, nous détectons des incohérences logiques basées sur les traces de construction (en traduisant les différents types de connaissances sur le développement (tous les types d'artefacts) à une représentation unifiée sur une base de données). Actuellement, nous avons concentré nos travaux plutôt sur la détection d'incohérence et d'identification, et nous avons laissé la manipulation pour les utilisateurs de ces outils.

3.5.1 La cohérence des artefacts

2.5.1.1 L'incohérence : définition de base

Nous utilisons le terme incohérence pour désigner toute situation dans laquelle un ensemble de descriptions ne respecte pas une relation qui est prescrite de tenir entre eux [103]. Une condition préalable à l'incohérence est que les descriptions en question ont une zone de chevauchement [104]. Une relation entre descriptions peut être exprimée comme une règle de cohérence, avec laquelle les descriptions peuvent être vérifiées. Dans la pratique actuelle, certaines de ces règles de cohérence sont capturés dans différents documents du projet, d'autres sont intégrés dans les outils, et certains ne sont pas pris n'importe où.

Cette définition de l'incohérence est plus large : il englobe de nombreux types d'incohérence qui se produisent dans le développement de logiciels. En particulier, la notion de l'incohérence logique est comprise dans notre définition, où la relation qui devrait tenir est celle qui ne devrait pas être possible de dériver une contradiction depuis un ensemble de propositions. La définition de l'incohérence de cette façon offre une certaine souplesse, car il ne nous lie pas à une notation particulière et nous permet d'envisager de nombreuses formes de l'incohérence tout au long du processus de développement. Ainsi, par exemple, la vérification des descriptions contre des règles de cohérence peuvent révéler des conflits des parties prenantes [105], des objectifs divergents [106], des failles dans un système en fonctionnement [107], et des écarts de processus de développement documentés [108].

2.5.1.2 Les causes de l'incohérence

L'incohérence est une partie inévitable du processus de développement de logiciels. Même dans les processus de développement le plus bien défini, géré et optimisé, les exigences du système sont souvent incertaines ou contradictoires, des solutions de conception alternatives existent, et des erreurs dans la mise en œuvre se posent.

L'étape d'ingénierie des exigences de développement est particulièrement illustrative de ces incohérences. Lors de l'acquisition des besoins, les exigences des clients sont souvent fragmentaires et incertaines. Pour les grands projets, en particulier, un certain nombre de responsables peut avoir des exigences incompatible, voire contradictoires. Dans de nombreux cas les clients peuvent même pas être certains de leurs propres besoins, et le travail d'un ingénieur des exigences est en partie à susciter et clarifier ces besoins. Le cahier des charges produit à la suite d'un tel processus de spécification et d'analyse n'est cependant pas statique : il continue à évoluer que de nouveaux besoins sont ajoutés et des conflits identifiés sont résolus.

Ainsi, il existe un large éventail de causes possibles d'incohérences dans le développement de logiciels. Beaucoup d'entre eux sont dus à l'hétérogénéité des produits en cours de développement (par exemple, les systèmes de déployer différentes technologies) et la multiplicité des acteurs et / ou les participants au développement impliqués dans le processus de développement. Des contradictions entre plusieurs participants de développement en raison de :

- Ils détiennent des points de vue différents.
- Ils parlent des langues différentes.

- Ils déploient des stratégies de développements (méthodes) différentes.
- Ils traitent des stades de développement différents.
- Ils ont partiellement, totalement ou non-chevauchement des domaines d'intérêt.
- Ils veulent atteindre des objectifs techniques, économiques et/ou politiques différents.

Depuis, les incohérences peuvent se produire dans les processus de développement et les produits logiciels pour une multitude de raisons, nous adoptons une simple et unique définition de ce qui constitue en fait une incohérence :

« Une incohérence se produit si et seulement si une (cohérence) règle a été brisée »

Cette règle décrit explicitement une certaine forme de relation ou de fait qui est nécessaire pour tenir. Dans des travaux précédents, nous avons examiné trois utilisations de ces règles de cohérence. Ils peuvent décrire les relations syntaxiques entre les artefacts de développement prévues par une méthode de développement, qui est aussi une façon de décrire les relations sémantiques entre les artefacts produits par cette méthode [103]. Ils peuvent également être utilisés pour prescrire les relations entre les sous-processus dans un processus de développement global, qui est aussi un moyen de coordonner les activités des développeurs de déployer différentes stratégies de développement [122]. Enfin, ils peuvent être utilisés pour décrire les relations définies par l'utilisateur qui émergent comme le développement d'une spécification de produit logiciel [139], [190]. Ceci est utile pour capturer des relations ontologiques entre les produits d'un processus de développement (par exemple, deux développeurs spécifiant un système de bibliothèque peuvent utiliser le terme «utilisateur» et «emprunteur» pour désigner la même personne).

Réduire une incohérence pour la rupture d'une règle facilite la détection et l'identification subséquente («diagnostic») d'incohérences dans les spécifications, et c’est un outil utile pour la gestion d'autres «problèmes» qui se posent au cours du développement de logiciel. Par exemple, si nous traitons le conflit comme l'interférence des artefacts d'une partie provoquée par les actions d'une autre partie [190], alors nous pouvons utiliser l’incohérence comme un outil pour détecter plusieurs conflits. De même, si nous définissons une erreur comme une action qui serait reconnu comme une erreur par son auteur (par exemple, une faute de frappe), alors nous pouvons détecter des erreurs qui se manifestent comme des incohérences.

Hagensen et Kristensen ont adopté une stratégie similaire dans leur examen explicite de la «perspective de cohérence" dans le développement de logiciels [191]. Ils ont proposé un Framework qui met l'accent sur les structures de représentation de l'information («descriptions») et les relations

entre ces structures. La cohérence des descriptions est définie comme des relations entre les interprétations des descriptions. Les techniques de manipulation de la cohérence sont modélisées en termes des légendes, des interprétations et des relations.

2.5.1.3 La détection et l'identification de l'incohérence

Une fois la cohérence a été définie en termes de règles explicites, l'incohérence peut être détectée automatiquement par le contrôle de ces règles. Par exemple, un contrôleur de type peut vérifier si une instance ou variable conforme ou non à sa définition de type. De même, un analyseur peut vérifier une phrase conforme ou non aux règles syntaxiques définies par la grammaire. Les inférences simples dans la logique classique peuvent également être utilisées pour détecter des incohérences logiques résultant de trop ou peu d'information. Par exemple, une contradiction (l'affirmation simultanée, ou déduction, des deux X et ¬X) peut être détectée de cette façon.

D'autres types d'incompatibilité sont plus difficiles à détecter. Un conflit entre deux participants de développement ne peut se manifester comme une contradiction tant que le développement a eu lieu (prise de la source d'origine de l'incohérence difficile à identifier). En outre, ce qui actuellement constitue une incohérence depuis une perspective d'un participant ne peut pas être le cas d'une autre perspective. Un exemple de ceci est une "incohérence" dans la déclaration de revenus d'une personne. Cette incohérence peut effectivement être une pièce «souhaitable» de l'information du point de vue d'un inspecteur des impôts !

Une des difficultés dans le traitement efficace des incohérences, même après qu'elles ont été détectées avec succès, c'est que le *genre* d'incohérence détectée doit aussi être identifié. Pour remédier à cette difficulté, une certaine forme de classification de l'incohérence peut être utile dans le diagnostic de leur source et de leur cause. Le système CONMAN (gestion de configuration) [192], par exemple, tente à classer la cohérence dans les programmes dans l'un des six types afin de faciliter la manipulation ultérieure de l'incohérence. Ce sont :

- **La cohérence complète :** où un système satisfait les règles que le langage de programmation précise pour les programmes juridiques (dans la mesure où ils peuvent être contrôlés avant l'exécution) ;
- **La cohérence type :** où un système satisfait la vérification des règles de type statique du langage de programmation ;

- **La cohérence version :** où un système est construit en utilisant une version de manière exacte de chaque fichier logique du code source ;
- **la cohérence dérivation :** où un système est opérationnellement équivalent à un système cohérent de version ;
- **La cohérence lien :** où chaque unité de compilation est exempt des erreurs de type statique, et chaque référence symbolique entre les unités de compilation est de type sécuritaire selon les règles du langage de programmation ;
- **cohérence accessible** : où tout le code et les données qui pourraient être accessibles ou exécutés en invoquant le système à travers l'un de ses points d'entrée sont en sécurité. CONMAN contrôle tous les six types de cohérence automatiquement, puis réagit différemment selon le type d'incohérence détectée. Les six types d'incohérence sont très spécifiques à la configuration des applications de gestion, mais leur identification est fondamentale pour la manipulation d'incohérence efficace dans ce cadre.

2.5.1.4 La manipulation de l'incohérence

La manipulation de l'incohérence [109] est une notion d'activités et de techniques qui visent à faire face aux incohérences dans le développement logiciel. Selon les types de langues et des abstractions utilisées dans le développement, des techniques très différentes de manipulation d'incohérence ont été développées. En général, on peut distinguer entre les *actions de changement* et les *actions de non-changement* et la décision générale soit *tolérer* ou *résoudre une incohérence*. La manipulation des incohérences comprend l'identification de ces actions ainsi que l'évaluation de leurs coûts et l'évaluation des risques de ne pas résoudre une incohérence. Dans le cas de notre approche, nous avons été limités à la gestion et la détection des incohérences, et nous avons laissé la manipulation aux travaux futurs.

3.5.2 Gestionnaire des règles de cohérence

Compte tenu des données sur les différentes dimensions d'un système logiciel, la partie suivante de l'outil définit gère et présente les règles de cohérence qui assurent que les différents artefacts restent cohérents en parallèle avec l'évolution du logiciel. La première étape consiste à définir ici ce

que signifie la compatibilité entre deux artefacts logiciels. En général, cela signifie qu'un détail syntaxique ou sémantique défini dans l'un des artefacts est représenté de manière appropriée dans l'autre artefact. Notre outil utilise une règle pour tenir compte de cette association [44].

CMAC assure la gestion de ces règles de cohérences en utilisant une base de données pour enregistrer les différentes opérations de changement et de s'en servir par le moteur de validation pour détecter les différentes violations de ces règles [45]. Il donne la possibilité aux développeurs de définir manuellement à travers une interface interactive et extensible les différentes règles logiques de cohérences entre les artefacts enregistrés dans la même base de données par l'outil d'extraction présenté précédemment.

3.5.2.1 La classification de la règle de cohérence

Dans cette section, nous classons les règles de cohérence. Le but de cette classification est de fournir une base pour l'évaluation de la pertinence du langage de règles de CMAC.

Nous discutons maintenant les classes des règles que nous avons proposées pour gérer l'incohérence. Ces classes vont être traitées par un moteur de validation pour identifier les différentes violations présentent entre les différents artefacts logiciels. Les développeurs peuvent toujours apporter des modifications à notre outil CMAC grâce à son architecture modulaire évolutive afin de supporter d'autres classes de règles et répondre aux exigences du client concernant la cohérence des artefacts de leur système.

3.5.2.1.1 Les règles simples

- **Existence :** les règles d'existence imposent qu'un certain artefact doit exister dans un autre artefact ou une liste des artefacts.
- **Egalité :** La règle précise que ce que deux artefacts doivent avoir la même valeur.
- **Unicité :** Les règles d'unicité imposent que, dans un ensemble de valeurs, chaque valeur doit apparaître exactement une seule fois.
- **Comparaison :** Ça concerne les opérateurs habituelles $<$, $>$, $\leq$ et $\geq$. Dans notre cas, nous devons prendre en compte tous les types des artefacts (les dates et les chaines de caractères par exemple) et non seulement des nombres entiers ou réels.

3.5.2.1.2 Les règles complexes

Dans la pratique, la plupart des règles sont des combinaisons de tout ce qui précède. Nous avons trouvé des exemples où les règles simples doivent être combinées en utilisant les opérateurs logiques (AND, OR, NOT), pour vérifier si un artefact est inclus dans un ensemble d'autres artefacts et vérifie si les ensembles sont égaux, etc. Dans le cas de notre approche, nous proposant un langage spécifique basé sur Java permettant aux développeurs de rédiger manuellement la règle de cohérence.

3.5.2.1.3 Les règles exotiques

Ces contraintes ne peuvent pas être valablement exprimées avec les opérateurs indiqués ci-dessus et nécessitent le développement de nouveaux opérateurs. Dans notre outil, en plus qu'il dispose d'un langage spécifique pour écrire les règles complexes, son architecture est extensible à soutenir d'autres opérateurs spécifiques pour enrichir la base de notre module de définition des règles de cohérence.

3.5.2.2 La définition des règles de cohérence

Cette section présente notre mécanisme uniforme de définition des règles de cohérence (moteur de règles) pour faire face à l'incohérence des artefacts quel que soit leurs types.

Pour les règles de cohérence, nous avons proposé de les définir sous forme de relations entre les artefacts définis dans le méta-modèle de construction (figure 8). Dans le cas de notre approche, nous permettons aux utilisateurs (qui sont souvent les chargés de l'évolution) de définir les dépendances entre les artefacts logiciels. Ces relations vont être représentées sous forme de formules logiques définies par l'ingénieur de développement à travers notre outil de validation et puis enregistrées dans la base de données. Ces données vont être utilisées par le moteur de validation pour détecter la violation de ces règles. Pour des relations plus complexes nous adoptons un langage spécifique qui consiste à les définir de façon manuelle par l'expert chargé de l'évolution.

3.5.2.2.2 Le méta-modèle des règles de cohérences

La figure présente U2MACR : notre méta-model de représentation unifié des différents artefacts extraits du système logiciel, ainsi que les règles de cohérence définies par les développeurs pour détecter les incohérences de ces artefacts et suivre l'impact de leur évolution (détecter les violations des règles méthodologiques).

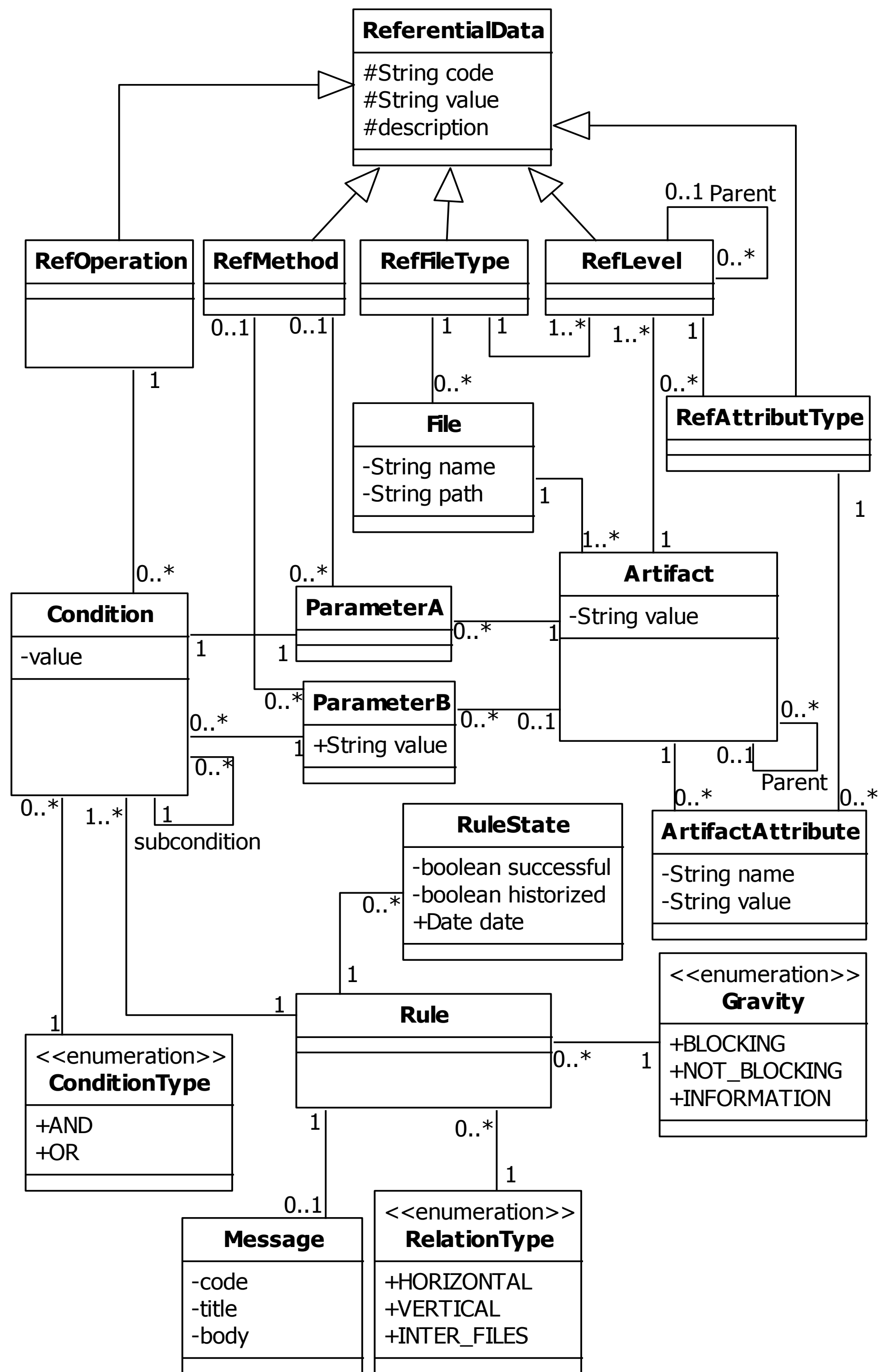

Figure 15 : U2MACR - Méta-Modèle des artefacts et des règles de cohérences

Nous avons considéré qu'une règle de cohérence « ***Rule*** » est composée de plusieurs conditions « ***Condition*** ».

Une condition peut être soit une composition de plusieurs sous condition « **Condition** » ou bien sous forme de deux Paramètres « ***ParameterA & B***» et une opération « ***RefOperation*** ».

« *ParameterA*» peut avoir la valeur exacte de l'artefact ou bien après l'application de la méthode « ***RefMethode*** ».

« *ParameterB*» même principe que le paramètre A, sauf qu'il peut avoir aussi une valeur saisie par l'utilisateur.

« *RefMethod*» ce sont des méthodes spécifiques qui peuvent être appliquées sur les paramètres (par exemple : StringToInt, NumberOfChart, ect)

L'opération « **RefOperation** » peut avoir la valeur des signes suivantes (>, <, ==, equals, etc), généralement tous les signes utilisés dans l'instruction « *if* » du langage java.

(Par exemple : (param1 > 3), dont le signe « > » représente l'opération, param1 est la valeur du ParameterA et 3 la valeur du ParameterB).

L'utilisateur peut spécifier un message d'erreur en cas d'incohérence comme il peut préciser le niveau de gravité pour chaque règle de cohérence (en cas de « *BLOCKING* » l'application affiche des erreurs de compilation).

« ***RuleState*** » contenant un historique des états de chaque règle (*successful* and *not succesful*), ainsi que l'état courante (*historized : false*), cette table est alimentée à chaque vérification par le moteur de validation.

3.5.2.2.3 Le langage des règles de cohérence

Cette section présente notre langage de règles basé sur les artefacts extraits du système logiciel et enregistrés dans la base de données. Nous présentons une base formelle simple pour la langue et nous formalisons notre exemple de règle.

Une règle de cohérence est généralement écrite sous la forme suivante :

IF ***conditions*** Then ***actions(A)*** *Else* ***actions(B)***

La condition peut être un ensemble d'autres conditions :

$$\textit{condition} = \sum_{\$condition}$$

Ou bien peut avoir la forme suivante :

Condition *= $ConditionType ($ParameterA $RefOperation $ParameterB)*

- ***$ConditionType*** = {OR | AND} : peut avoir deux valeurs OR ou AND.
- ***$RefOperation*** peut avoir les symboles suivants *:{<, >, =, ==, !=, equals, …}* presque tous les symboles utilisés dans les conditions if du langage Java.
- ***$ParameterA*** *= {($Artefact | $RefMethode($Artefact) :* égale la valeur d'un artefact soit simple ou bien après application d'une méthode spécifique (exemple : conversion d'une chaine de caractère à une valeur numérique).
- ***$ParameterB*** = {*Value* | *$Artefact | $RefMethode($Artefact)} :* même principe que *$ParameterA*, sauf que *$ParameterB* peut prendre une valeur saisie par le développeur.

"actions(A)" peut avoir les taches suivantes :

actions(A) *= {(update curent $RuleState) & (add new $RuleState)}*

Si les conditions de la règle sont vraies alors :

1. mettre à jour l'état courante *$RuleState(historized:false)* de la règle en modifiant la valeur de l'attribut *historized* à *true*, puis
2. ajouter un nouveau objet *$RuleState* avec *successful:true*, *historized :false* et *date :new Date().*

"actions(A)" peut avoir les taches suivantes :

actions(B) *= {(update curent $RuleState) & (add new $RuleState) & (display Messages)}*

Si les conditions de la règle sont fausses alors :

1. Mettre à jour l'état courante *$RuleState(historized:false)* de la règle en modifiant la valeur de l'attribut *historized* à *true*,

2. Ajouter un nouveau objet *$RuleState* avec *successful:false*, *historized :false* et *date :new Date(),*

3. Si *$Gravity :blocking* → Arrêter le traitement et afficher le message spécifique (ou message générique si le développeur n'a pas spécifié un message d'erreur) dans le console sinon si *$Gravity :{not blocking|information}* Afficher le message dans le console sans bloquer le traitement.

3.5.3 Moteur de validation

Nous avons mis en place un module dans notre outil CMAC appelé « moteur de validation » responsable de la vérification automatique de l'incohérence. Le programme est développé avec le langage java, et principe clé c'est de convertir les règles définies dans le méta-model à des requêtes écrites en langage Java. Il se base sur les artefacts extraits et les règles de cohérence définies par le développeur afin de sortir les différents messages spécifiques dans la console et mettre à jour la base de données par le résultat de validation.

Le prototype développé donne la possibilité d'exécuter une seule règle à la fois ou bien paramétrer le lancement au démarrage ou bien sous forme de batch. Son architecture modulable permet aux développeurs d'évoluer le module pour répondre aux différentes exigences des parties prenantes.

3.6 Conclusion

Dans cette recherche, nous avons présenté une approche de gestion des cohérences des artefacts hétérogènes pour contrôler l'évolution des logiciels. Cette approche comprend un méta-model unifié de représentation des différents artefacts hétérogènes, ainsi un formalisme uniforme de spécification des règles de cohérence méthodologiques pour soutenir l'ingénierie de l'information dans les projets de développement de logiciels. Et nous avons validé notre approche en construisant une plateforme modulaire et extensible permettant la construction uniforme des artefacts hétérogènes, la définition des règles méthodologiques et la détection des violations de ces règles de cohérence.

Grâce à son architecture modulaire, CMAC sert de bonne base pour mettre en place toute une chaîne d'outils de traitement au-dessus de celui-ci. De nombreuses tâches dans le cadre du déve-

loppement de logiciels et l'ingénierie inverse exigent non seulement de récupérer des informations qui nous intéressent, mais aussi de traiter d'une certaines règles particulières. Des tâches comme le prétraitement, refactoring, validation de la cohérence, etc, tous exigent un outil d'extraction personnalisé. Par exemple, bloquer la modification de toute entité avant de mettre à jour la classe dans le model UML. Cet exemple indique que CMAC est une bonne base pour la mise en œuvre des outils et des langages destinés à l'évolution du génie logiciel.

CHAPITRE 4

Prototype de validation

Objectif

Dans ce chapitre, nous présentons le détail de développement de notre plateforme plate-forme appelée CMAC « *Consistency Management based on Artefacts Construction* » en détaillant les différents modules qui constituent son architecture.

Sommaire

4.1 Introduction 105

4.2 Architecture globale du prototype de validation 106

4.3 Le module « artifactsBuilder » 109

4.3.1 Liste des taches d'extraction 110

4.3.2 Nouvelle tâche d'extraction 111

4.4 Le module « activityMonitor » 112

4.5 Le module « dataRepopsitoryManager » 113

4.6 Le module « consistencyRulesBuilder » 114

4.6.1 Liste des règles de cohérence 114

4.6.2 Nouvelle règle de cohérence 115

4.7 Le module « checkEngine » 117

4.8 Conclusion 118

5.2 Perspectives des travaux 123

4.1 Introduction

Pour une validation opérationnelle de l'ensemble des contributions de nos travaux proposés pour la gestion de la cohérence des artefacts logiciels hétérogènes dans l'évolution logiciel, nous avons mis en œuvre une plate-forme intégrée et hébergeant différents modules et composants implémentant chacun un des aspects de l'évolution évoqués tout au long de ce manuscrit.

Notre plate-forme ainsi que tous les modules qui la constituent ont été implémentés avec le langage Java[17] par l'IDE (Integrated Development Environment) Eclipse[18] sous forme d'une application web[19] partagée et évolutive. La particularité et l'objectif central de l'utilisation de Java est que les logiciels écrits dans ce langage doivent être très facilement portables sur plusieurs systèmes d'exploitation tels que UNIX, Windows, Mac OS ou GNU/Linux, avec peu ou pas de modifications. Le choix d'Eclipse a été dicté par le fait que c'est un IDE open source intéressant un nombre de plus en plus croissant de développeurs à travers le monde [193]. Ceci est, constitue à notre sens le meilleur moyen de diffuser nos implémentations au sein de la communauté des développeurs pour qui elles ont été conçues en premier lieu.

Cela, nous semble-t-il, devrait favoriser l'utilisation puis le test de nos implémentations et donc leur amélioration continue. C'est aussi une bonne opportunité de trouver des partenaires testant nos outils sur des projets de tailles réelles.

Dans ce chapitre, nous présentons notre plate-forme appelée CMAC « *Consistency Management based on Artefacts Construction* » en détaillant les différents modules qui constituent son architecture. Les principaux modules sont les suivants :

1) **Module** semi-automatisé d'extraction et de consultation des artefacts hétérogènes par différentes sortes de parseurs (Parsers). Nous avons implémenté des parseurs par niveau abstro-granulaire et par type de fichier (class, attribut, Bean, scénarios, …) et des librairies en java permettant d'extraire les artefacts de façon semi-automatique. Pour les artefacts plus complexes tels que les artefacts composés, les fichiers textes et les documents de spécification des règles de métiers, nous adoptons une approche qui consiste à les définir de façon manuelle par l'expert chargé de l'évolution. Ça veut dire que le référentiel d'artefacts logiciels extraits par différents parseurs et différents outils d'analyse statique et également par des informations fournies par les

17 www.oracle.com/technetwork/java/
18 http://www.eclipse.org/
19 http://fr.wikipedia.org/wiki/Application_web

différents experts ou acteurs du développement (architectes, spécialistes qualité, etc.).

2) **Module** de surveillance des différentes opérations de construction des développeurs afin de mettre à jour le référentiel des artefacts.

3) **Module** d'enregistrement et de représentation unifiée de ces artefacts de façon homogène dans le méta-model U2MHA proposé dans les chapitres précédents.

4) **Module** de définition et de gestion des règles de cohérence méthodologiques basées sur les artefacts extraits dans la base de données.

5) **Module** de validation et de détection des différentes violations des règles spécifiées.

Nous avons implémenté également dans le cadre de notre plateforme une application web, dans le but de présenter des interfaces utilisateurs pour les différents modules de notre plateforme.

Ce chapitre est organisé comme suit : la première section résume l'architecture et la conception générale de notre prototype de validation ainsi que le détail de chacun des modules qui composent notre plate-forme. La deuxième section présente les résultats de notre approche, tandis que la troisième conclut le chapitre et discute brièvement les perspectives du développement de notre prototype.

4.2 Architecture globale du prototype de validation

Notre plate-forme appelée CMAC « *Consistency Management based on Artefact Construction* » permet à la fois de construire et d'unifier les artefacts logiciels hétérogènes et enfin de vérifier la cohérence et la conformité des artefacts résultant du processus de construction.

CMAC est construit à l'aide d'Eclipse. Eclipse est une plate-forme open source largement utilisée pour construire des plateformes ouvertes et extensibles de développement constituées d'outils destinés à la construction, au déploiement et à la gestion de logiciels en incluant toutes les phases du cycle de vie de développement. Eclipse fournit un ensemble d'outils assistant le développeur dans chaque étape du développement, de l'analyse au déploiement. Dans notre travail, nous nous sommes intéressés à Maven[20]. Maven est un outil « open source » d'Apache Jakarta. Il permet de faciliter et d'automatiser la gestion et la construction d'un projet java. Maven utilise une approche déclarative, où

20 http://maven.apache.org

le contenu et la structure du projet sont décrits par Maven, Cela aide à mettre en place des standards de développements au niveau d'un projet et réduit le temps nécessaire pour écrire et maintenir les scripts de construction.

CMAC est une plateforme modulaire et évolutive, composée de 5 modules principaux :

1) ***artifactsBuilder :*** c'est un module de gestion des taches de construction des artefacts logiciels, permettant aux utilisateurs de définir et gérer l'extraction des artefacts depuis les différents types de fichiers et niveaux sélectionnés, afin de les stocker dans une base de données unifiée.

2) ***activityMonitor :*** c'est un module de surveillance de différentes activités de modification des développeurs sur les artefacts logiciels, permettant d'écouter les différentes opérations de construction (Ajout, modification et suppression) effectuées par les développeurs sur les fichiers supervisés, afin d'appeler les taches de construction concernées, dans le but de mettre à jour le référentiel des artefacts.

3) ***dataRepositoryManager :*** c'est un module de gestion de la communication avec le référentiel des artefacts et des règles de gestion (ajout, modification, recherche et suppression, etc.), permettant de centraliser la communication (façade) avec la base de données.

4) ***consistencyRulesBuilder :*** c'est un module de gestion des règles de cohérence entre les artefacts logiciels enregistrés, permettant aux utilisateurs de définir et gérer les différentes règles de cohérences méthodologiques entre les artefacts unifiés dans le référentiel.

5) ***checkEngine :*** c'est un moteur de validation chargé de détecter les incohérences. Il analyse les règles de cohérence enregistrées dans la base de données (conversion vers le langage spécifique avant l'exécution) et produit un rapport de détection des incohérences.

L'architecture modulaire globale montrant l'interaction de ces modules est schématisée par la figure 16. Nous discuterons par la suite les détails de l'implémentation de ces cinq modules.

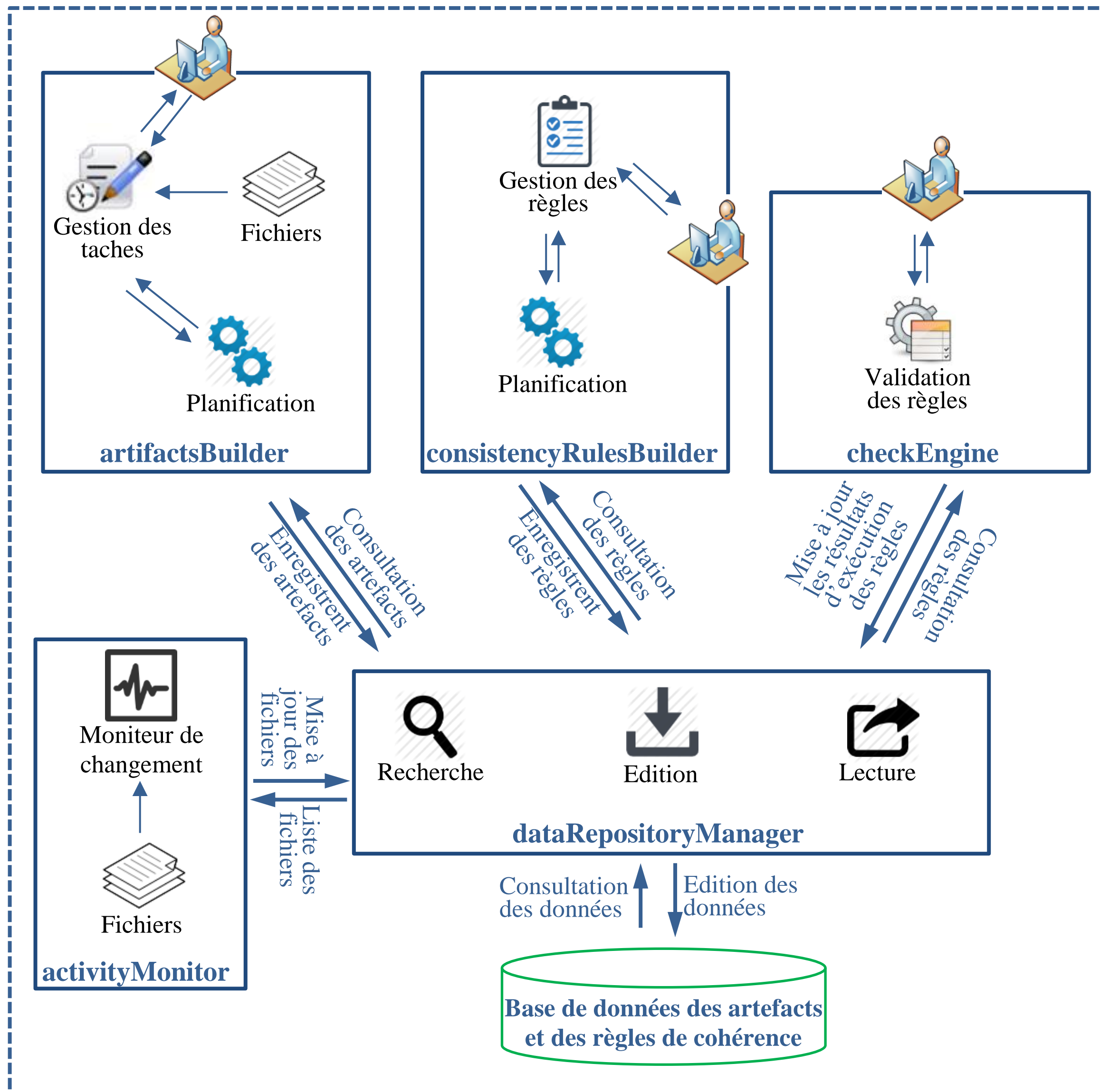

Figure 16 : Architecture modulaire globale de CMAC

Nous avons développé tous les modules en respectant la structure standard de Maven, et la figure 17 montre une vue Eclipse des différents modules de notre plateforme CMAC.

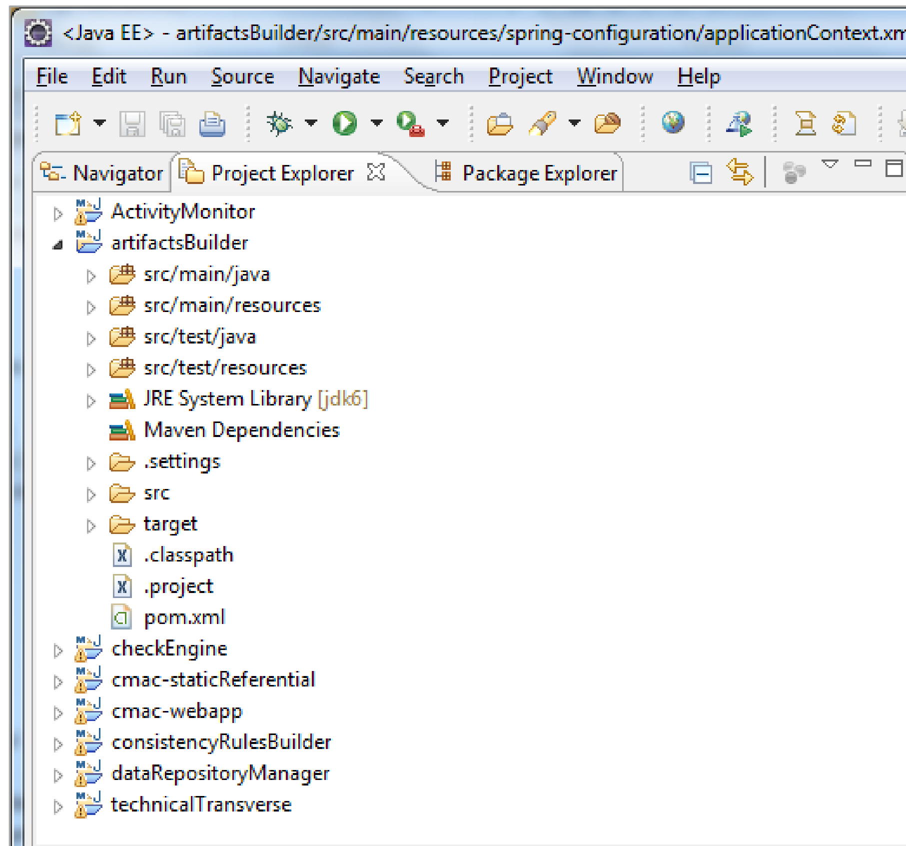

Figure 17 : Vue Eclipse de tous les modules de CMAC

4.3 Le module « artifactsBuilder »

« *artifactsBuilder* » permet aux utilisateurs d'ajouter et gérer d'une manière transparente (interface web) les taches d'extraction des artefacts. Pour créer une tache d'extraction il faut spécifier le type de fichier et le niveau abstro-granulaire des artefacts et préciser la classe d'extraction spécifique ou générique. Les extracteurs dans le cas de notre approche est une stratégie incluant des implémentations spécifiques utilisant les différents parseurs (selon le type de fichier à analyser) pour générer les différents artefacts logiciels et leurs relations tels que spécifiés par notre modèle U2MHA. Ce mécanisme a été implémenté en utilisant le design pattern Strategy[21]. Le pattern Strategy est un patron de conception (design pattern) de type comportemental grâce auquel des algorithmes peuvent être sélectionnés à la volée au cours du temps d'exécution selon certaines conditions. Le but d'utilisation de Strategy c'est de permettre aux experts de l'évolution de personnaliser ou bien créer un nouveau code

21 http://en.wikipedia.org/wiki/Strategy_pattern

spécifique d'extraction couvrant d'autres types de fichiers. Le résultat de l'exécution du module *« artifactsBuilder »*, en d'autres termes les artefacts générés (enregistrés dans le référentiel des artefacts U2MHA), est utilisé soit pour la recherche et la consultation d'artefacts logiciels par le module CMAC-*Webapp* ou bien par d'autres outils d'analyse et de gestion de la cohérence de données.

Les parseurs utilisés par les classes d'extraction permettent d'analyser les documents sources de divers type de fichiers et langage de programmation (Java, C, C++, Perl, PHP, COBOLE, text, xml, Doc, ect). Des parseurs ont été développés à l'aide de Java Compiler Compiler[22] (*JavaCC*) et d'autres parseurs existent dans le marché de l'open source. *JavaCC* est un compilateur de compilateurs et un générateur d'analyseurs lexicaux entièrement écrit en JAVA et intégré la plate-forme Eclipse. Il permet d'interpréter la grammaire d'un langage donné et d'en générer un parseur. Cela nous a simplifié la possibilité de rajouter de nouveaux parseurs pour d'autres types de fichiers au fur et à mesure de l'avancement du développement de notre prototype. Nous avons donc développé de nombreux parseurs incluant également un parseur de code source et d'autres. A partir des fichiers source d'entrée, les parseurs sont utilisés par le module *« artifactsBuilder »* pour générer les différents artefacts logiciels et leurs relations tels que spécifiés par notre modèle U2MHA.

Les tâches créées peuvent être lancées manuellement par les utilisateurs, comme elles peuvent être programmées pour un lancement périodique sous forme des Batchs[23], en s'appuyant principalement sur *« Spring Batch[24] »*. Spring-Batch est un Framework développé en collaboration par SpringSource et Accenture, c'est un Framework adapté à l'écriture des batchs (batch : programme automatisé réalisant un ensemble de traitements sur un volume de données).

4.3.1 Liste des taches d'extraction

La figure 18 présente une capture de l'écran que nous avons développé pour consulter la liste des tâches déjà créées avec les informations nécessaires comme le niveau d'artefact et le statut de la tâche d'extraction. Cet écran donne la possibilité de lancer une tâche directement en cliquant sur l'action de lancement, comme il offre des actions permettant de supprimer une tâche ou bien accéder à la page de modification et de création d'une nouvelle tâche.

22 http://en.wikipedia.org/wiki/JavaCC
23 http://fr.wikipedia.org/wiki/Traitement_par_lots
24 http://projects.spring.io/spring-batch/

Figure 18 : Ecran de gestion des tâches d'extraction

4.3.2 Nouvelle tâche d’extraction

Afin de créer et programmer une tâche d’extraction et d’enregistrer des artefacts logiciels, nous avons utilisé l’écran de création de notre prototype. La capture d’écran de la figure 19 montre un exemple de création d’une nouvelle tâche d’extraction.

- **Nom de la tâche** : Un nom significatif de la tâche d’extraction.
- **Description de la tâche :** Une description fonctionnelle de la tâche d’extraction.
- **Liste des fichiers :** Si oui, une zone va apparaître pour saisir la liste des fichiers concernés par l’extraction des artefacts (contenant les artefacts à extraire).
- **Type de fichier :** Si oui, une zone va apparaître permettant de choisir le type des fichiers qui vont être la source d’extraction.
- **Niveau d’artefacts :** Optimiser la tâche d’extraction par la sélection d’un niveau existant ou bien choisir de créer un nouveau. Si l’option « créer un nouveau » est sélectionnée, une zone va apparaître pour créer un nouveau niveau en saisissant le nom le niveau parent et la classe d’extraction liée au nouveau niveau. L’utilisateur peut utiliser une

classe d'extraction existante ou bien créer une nouvelle classe spécifique en suivant le mécanisme définit.

Figure 19 : Ecran de création de la tâche d'extraction

4.4 Le module « activityMonitor »

Les artefacts sont extraits des plusieurs types de fichiers dans les logiciels en utilisant un réseau de différents outils d'extraction spécifiques par type de fichier et niveau abstro-granulaire des artefacts. Le module« *activityMonitor* » détermine périodiquement si des fichiers ont changé et puis exécute automatiquement les taches d'extraction appropriées ou bien enregistre l'information dans la base de données en attendant le planificateur des taches (dépend du paramétrage).

Des changements sont détectés et mis à jour au niveau du fichier. Autrement dit, lorsque le moniteur d'activité détecte que le fichier représentant un ensemble d'artefacts particuliers a changé, toutes les informations de tous les artefacts logiciels appartenant à ce fichier, doivent être mises à jour dans le référentiel. Le niveau de fichier semble être le plus approprié pour la mise à jour de ces informations, car il est facile de détecter le changement d'un fichier et plus facile d'écrire des extracteurs

qui travaillent sur un fichier tout à la fois. Bien qu'il soit plus commun pour seulement une petite quantité d'informations dans le fichier est réellement changée, de détecter et extraire touts les renseignements tout en maintenant la cohérence avec le reste de la base de données aurait fait tous les extracteurs beaucoup plus complexes et plus lents. Nous avons opté à la place de mettre la charge de la détermination de la mise à jour incrémentale réelle dans le gestionnaire de base de données et de garder les extracteurs simple et rapide.

Il faut bien noter que les approches actuelles, pour gérer la détection et la mise à jour incrémentale des changements, ré-exécute la tâche de construction de tous les artefacts logiciels en parcourant la totalité des fichiers et en maintenant la mise à jours de la base de données, ce qui rend la tâche beaucoup plus complexe et plus lente. C'est pour cela, nous avons opté à mettre en place le module de surveillance des changements *« activityMonitor »* par fichier et permettant en plus aux utilisateurs de sélectionner les fichiers concernés.

Nous avons développé le module sous forme de Batch. Le batch peut être programmé pour surveiller périodiquement le changement des fichiers enregistrés dans la base de données. Et pour détecter le changement, nous avons utilisé la bibliothèque *« java.io »* en comparant la date de la dernière modification (la méthode `file.lastModified()`) d'un fichier avec celle enregistrée dans la base de données.

4.5 Le module « dataRepopsitoryManager »

« dataRepositoryManager » représente le noyau de notre plateforme, c'est un module de gestion de la communication avec le référentiel des artefacts et des règles de gestion (ajout, modification, recherche et suppression, etc.), permettant de centraliser la communication (façade) avec la base de données. Nous avons opté à externaliser et centraliser l'accès à la base de données dans le but de sécuriser et réutiliser les fonctionnalités dans tous les autres modules.

Nous avons développé ce module en s'appuyant principalement sur JPA[25] (Java Persistence API). JPA est une interface de programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java, elle permet de définir très facilement, et précisément des objets métiers, qui pourront servir d'interface entre la base de données et l'application.

25 http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

4.6 Le module « consistencyRulesBuilder »

Le principe du constructeur des règles de cohérence *« consistencyRulesBuilder »* est très simple, c'est une interface à travers CMAC permettant aux utilisateurs de définir les règles sous forme des opérations logiques entres les valeurs des artefacts déjà extraits par le module « artifactBuilder ».

Pour des règles simples (celles définies entre deux artefacts par exemple), nous avons proposé de les définir manuellement à travers une page web interactive proposée dans notre outil. Pour des règles plus complexes, les développeurs peuvent soit garder la même règle logique en développant une classe d'extraction personnalisée pour trouver à la fin les artefacts à comparer, tout en utilisant un parseur adapté, et puis lier cette classe à la tâche d'extraction en passant par la même page web de l'outil ou bien, utilisant un langage spécifique basé sur java tout en spécifiant les variables (id artefacts) entre les symboles « &{ } » et les méthodes entre les crochets « #[ ] ».

Les deux manières de déclaration des règles de cohérence délivrent en sortie des données enregistrées dans un modèle de données unifié.

4.6.1 Liste des règles de cohérence

La figure 20 présente une capture de l'écran que nous avons développé pour consulter la liste des règles de cohérence déjà créées avec les informations nécessaires comme le type et le statut de la règle de cohérence méthodologique. Cet écran donne la possibilité de lancer la validation d'une règle directement en cliquant sur l'action de lancement comme il offre des actions permettant de supprimer une règle ou bien accéder à la page de modification et de création d'une nouvelle règle.

CMAC - Liste des règles de cohérence

La gestion des règles de cohérence ...

Nouvelle règle de cohérence

Show 10 Rows

Numéro	Description	Type de règle	Statut	Actions
0001	La cohérence entre les classes UML et JAVA	EQUAL	New	
0002	La cohérence entre les attributs UML et JAVA	EQUAL	New	
0003	La Declaration des beans JAVA dans les descripteurs de déploiement	EXIST IN	New	
0004	L'unicité des declarations des beans dans les fichiers XML	UNIQUE	Consistent	
0005	L'unicité des declarations des paramètres properties	UNIQUE	Inconsistent	
0006	Les règles de gestion entre JAVA et DOC	COMPLEX	Inconsistent	
0007	les classes entre Java et DOC	EXIST IN	New	
0008	les classes entre UML et DOC	EXIST IN	New	
0009	Les tests unitaires des méthodes	EXIST IN	New	
0010	Le nombre de lignes des méthodes limité	LE	New	

Showing 1 to 10 of 11 entries

Figure 20 : Ecran de gestion des règles decohérence

4.6.2 Nouvelle règle de cohérence

Afin de créer, enregistrer et programmer le lancement d'une règle de cohérence, nous avons utilisé l'écran de création de notre prototype. Les captures d'écran des figures 21 et 22 montre respectivement deux scénarios de création d'une nouvelle règle de cohérence. La première figure montre un scénario de création d'une règle de cohérence méthodologique simple et guidée par cette page. La deuxième montre un deuxième scénario concernant l'utilisation d'un langage spécifique pour la création d'une règle de cohérence complexe.

- **Nom de la règle** : Un nom significatif de la règle de cohérence.
- **Description de la règle :** Une description fonctionnelle de la règle de cohérence.
- **Règle méthodologique :** Si oui, une zone va apparaître facilitant la saisie de la règle méthodologique.
- **Règle spécifique :** Si oui, une zone de texte va apparaître permettant de saisir un code spécifique respectant notre mécanisme proposé précédemment.
- **Paramètre A :** fait référence vers le premier artefact de la règle.
- **Paramètre B :** fait référence vers le deuxième artefact de la règle.

- **Méthode A :** le nom de la méthode à appliquer sur l'artefact référencé dans « paramètre A » afin d'utiliser le résultat dans la règle.
- **Méthode B :** le nom de la méthode à appliquer sur l'artefact référencé dans « paramètre B » afin d'utiliser le résultat dans la règle.
- **Action :** C'est l'action utilisée pour valider la cohérence des artefacts.
- **Planification de lancement :** La méthode utiliser pour lancer la règle de cohérence, soit manuellement, et dans ce cas l'utilisateur prend la charge de lancer **manuellement** la règle depuis l'écran « Liste des règle de cohérence ». ou bien automatique à travers les batch dont l'utilisateur doit saisir la fréquence de lancement en utilisant l'expression Cron[26].
- **Message spécifique :** L'utilisateur peut toujours saisir un message spécifique dans le cas d'incohérence.

Figure 21 : Ecran du premier scénario de création de la règle de cohérence

[26] https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm

Figure 22 : Ecran du deuxième scénario de création de la règle de cohérence

4.7 Le module « checkEngine »

« *checkEngine* » est un moteur de validation chargé de détecter les incohérences. Il analyse les règles de cohérence enregistrées dans la base de données (conversion vers le langage spécifique avant l'exécution) et produit un rapport de détection des incohérences.

Nous avons choisi de construire notre moteur de validation en langage java. Notre approche consiste à convertir les différentes règles enregistrées à des requêtes en langage java (section 3.5.2.2.2), et à la sortie de l'exécution de ces conditions le programme mis à jours l'historique des états de chaque règle et affiche les messages spécifiques en cas d'incohérence.

«*checkEngine* » offre aussi une page de paramétrage permettant aux utilisateurs de modifier la configuration par défaut et qui concerne surtout le mode de lancement du moteur de validation (manuelle ou bien périodique).

4.8 Conclusion

Dans ce chapitre nous avons décrit, la conception et les fonctionnalités de la plate-forme que nous avons développées et déployées pour la validation de nos contributions pour la construction et la gestion de la cohérence des artefacts logiciels. Cette plate-forme a été développée sous forme d'un ensemble de modules avec Maven et Eclipse et des Frameworks open sources pour une meilleure évolution et réutilisation de ces fonctionnalités.

Dans ce chapitre nous avons d'abord décrit l'architecture globale de la plate-forme qui est constituée de cinq principaux modules qui sont « *artifactsBuilder* », « *consistencyRulesBuilder* », « *activityMonitor* », « *checkEngine* » et « *dataRepositoryManager*». Nous avons donc détaillé les fonctionnalités de chacun de ces modules.

Actuellement, notre travail de développement porte à la fois sur l'accomplissement et l'achèvement du langage spécifique de définition des règles de cohérence, et les parseurs d'extraction des fichiers complexes.

Nos travaux d'implémentation concernent aussi le développement de quelques composants techniques et d'un module web couvrant les fonctionnalités de la couche présentation pour tous les modules développés.

CHAPITRE 5

Conclusion générale et perspectives

Objectif

Nous présentons, dans ce chapitre, le bilan de nos deux contributions majeures de nos travaux de recherche. Après avoir porté un regard critique sur les sujets que nous avons évoqués dans ce manuscrit, nous présenterons les perspectives de nos travaux de recherche qui couvrent d'autres voies de recherche comme extension de la solution que nous proposons.

Sommaire

5.1 Bilan des contributions 121

5.1.1 Contribution à la construction des artefacts logiciels 122

5.1.2 Contribution à la gestion de la cohérence des artefacts logiciels 122

5.2 Perspectives des travaux 123

5.1 Bilan des contributions

Le travail présenté dans cette thèse traite la gestion de la cohérence des artefacts logiciels. L'objectif principal étant, de permettre, d'une part, la construction des artefacts hétérogènes depuis les différents types de fichiers du système logiciel vers une représentation unifiée dans une base de données et d'autre part, d'offrir un mécanisme uniforme qui permet de définir et de valider des règles de cohérence entre les différents artefacts logiciels construits. Afin de s'atteler à cette tâche, nous avons proposé de découper la problématique globale en deux axes principaux de recherche : (1) la construction des artefacts logiciels, (2) la gestion de la cohérence des artefacts logiciels.

Après avoir dressé un état de l'art des différentes approches et techniques les plus largement connues dans le domaine de la gestion de la cohérence, nous avons proposé dans un premier volet de nos travaux, une approche unifiée de construction des artefacts logiciels basée sur une classification par niveaux d'abstraction et de granularité pour simplifier le processus d'extraction des artefacts logiciels. Dans un deuxième volet de nos travaux de recherche, et afin de gérer la cohérence de tous les artefacts, nous avons proposé un mécanisme uniforme basé sur la logique pour la détection des incohérences des artefacts logiciels hétérogènes. Cette démarche consiste à définir et valider des règles de cohérence méthodologiques entre les différents artefacts afin de détecter et afficher les incohérences aux utilisateurs tout au lents du processus de développement.

Pour valider notre approche CMAC, nous avons développé une plateforme web pour une meilleure utilisation auprès de la communauté des développeurs. Grâce à son architecture modulaire, CMAC sert de bonne base pour mettre en place toute une chaîne d'outils de traitement au-dessus de celui-ci. De nombreuses tâches dans le cadre du développement de logiciels et de l'ingénierie inverse exigent non seulement de récupérer des informations qui nous intéressent, mais aussi de traiter d'une certaines règles particulières. Des tâches comme le prétraitement, refactoring, validation de la cohérence, etc., exigent un outil d'extraction personnalisé. Par exemple, bloquer la modification de toute entité avant de mettre à jour la classe dans le model UML. Cet exemple indique que CMAC est une bonne base pour la mise en œuvre des outils et des langages destinés à l'évolution du génie logiciel.

5.1.1 Contribution à la construction des artefacts logiciels

La première contribution de cette thèse a porté sur la construction et la représentation unifiée des artefacts logiciels. En effet, nous avons proposé une approche qui se base sur une classification par niveaux abstro-granulaire pour extraire les artefacts logiciels hétérogènes.

Notre démarche, que nous avons d'ailleurs formalisée, débute par la définition et la gestion des tâches d'extraction des artefacts. La tâche d'extraction peut être crée et planifiée par les utilisateurs qui sont souvent les développeurs d'applications. L'objectif étant de fournir un moyen pour extraire que les artefacts utilisés dans la formulation des règles de cohérence au lieu de proposer un programme d'extraction de tous les artefacts du système logiciel, qui peut avoir des impacts de performance très importants (le cas des travaux de recherches actuels). En second lieu, nous avons proposé un méta-model unifié pour sauvegarder tous les artefacts résultants des différentes tâches d'extraction.

Cependant, une des limites actuelles de notre approche de construction des artefacts consiste à ne pas prendre en considération un grand nombre de types de fichiers et nécessite l'intervention des développeurs pour créer des parseurs et des extracteurs spécifiques en suivant le mécanisme que nous avons défini pour ajouter des nouveaux niveaux d'extraction. D'ailleurs, ceci est l'une de nos perspectives, que nous présentons dans la section suivante.

5.1.2 Contribution à la gestion de la cohérence des artefacts logiciels

Lors de l'évolution des artefacts logiciels, il s'avère, parfois, que les modifications apportées à un artefact ont un impact sur d'autres artefacts logiciels. En effet, il est donc primordial de mettre en œuvre des mécanismes pour la détection de l'incohérence des artefacts logiciels hétérogènes durant toutes les phases du développement logiciel.

Dans notre deuxième contribution, nous avons proposé un mécanisme uniforme basé sur la logique pour la détection des incohérences des artefacts logiciels. Ce mécanisme permet aux utilisateurs de créer et suivre des règles de cohérence basées sur des formules logiques, et expose un moteur de validation des règles pour la détection des incohérences.

Enfin, et pour valider notre approche, nous avons proposé l'élaboration d'un prototype de validation modulaire développé par le langage Java dans Eclipse. Cet outil est une application web

proposant des pages interactives pour la création et le suivi des règles de cohérence ainsi que la validation pour détecter les différentes violations.

Cependant, une des limites actuelles de notre approche de gestion de la cohérence consiste à ne pas prendre en considération certains types de relation de cohérence utilisés dans la définition des règles de cohérence, et l'utilisateur est obligé de passer par le langage spécifique que nous avons proposé et qui nécessite des compétences dans le développement java et des connaissances approfondies sur notre approche proposée. D'ailleurs, ceci est l'une de nos perspectives, que nous présentons dans la section suivante.

5.2 Perspectives des travaux

Comme une première perspective de nos travaux de recherche concernant la construction des artefacts logiciels, nous envisageons de développer plus de parseurs et extracteurs spécifiques pour couvrir un grand nombre de types de fichiers dans notre outil d'extraction semi-automatique des artefacts logiciels. Nous envisageons également de définir des templates personnalisés pour les fichiers complexes comme les fichiers textes et les fichiers doc, pour faciliter et automatiser l'extraction.

Dans un futur proche, et afin de compléter notre approche de gestion de cohérence, nous envisageons d'évoluer notre langage spécifique que nous avons proposé pour définir des règles de cohérence plus complexes (qui ne peuvent pas être définies par une simple condition logique), et pour faire, nous allons enrichir notre langage par d'autres types de relations de cohérence et par d'autres méthodes de conversion et de comparaison.

Nous travaillons actuellement sur l'évolution de notre plateforme CMAC qui peut faciliter la gestion de la cohérence entre les artefacts hétérogènes et servir de bonne base à la mise en œuvre des outils et des langages destinés à l'évolution du génie logiciel. Enfin, nous souhaitions intégrer notre approche dans plusieurs outils de refactoring, de mapping objet et d'industrialisation de développement, citant par exemple l'outil « *Checkstyle* », « *Dozer* » et « *Jenkins* ». Ceci est, constitue à notre sens le meilleur moyen pour diffuser nos implémentations au sein de la communauté des développeurs. Cela, devrait favoriser l'utilisation puis le test de nos implémentations et donc leur amélioration continue. C'est aussi une bonne opportunité de trouver des partenaires testant nos outils sur des projets de tailles réelles.

Bibliographie

[1] J. Buckley, T. Mens, M. Zenger, A. Rashid and G. Kniesel, "Towards a taxonomy of software change," *Journal on Software Maintenance and Evolution: Research and Practice,* vol. 17, pp. 309-332, 2005.

[2] A. Capiluppi, M. Morisio and P. Lago, "Evolution of understandability in OSS projects," in *the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, 2004.

[3] M. Lehman and L. Belady, "Program Evolution Processes of Software Change," *London Academic Press,* 1985.

[4] A. Hassan and R. Holt, "Predicting change propagation in software systems," in *20th IEEE International Conference on Software Maintenance (ICSM'04)*, 2004.

[5] T. Zimmermann, P. Weißgerber, S. Diehl and A. Zeller, "Mining version histories to guide software changes," in *26th International Conference on Software Engineering (ICSE 2004)*, 2004.

[6] T. Ball and S. Eick, "Software visualization in the large," *IEEE Computer,* pp. 33-43, 1996.

[7] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM Symposium on Software Visualization*, 2003.

[8] M. Fischer, M. Pinzger and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, 2003.

[9] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," in *Proceedings of LMO 2002 (Langages et Mod`eles `a Objets*, 2002.

[10] F. Van Rysselberghe and S. Demeyer, "Studying software evolution information by visualizing the change history," in *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004.

[11] X. Wu, A. Murray, M.-A. Storey and R. Lintern, "A reverse engineering approach to support software maintenance: Version control knowledge extraction," in *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, 2004.

[12] I. Sommerville, Software Engineering, sixth ed., Addison-Wesley, 2000.

[13] C. T.A, "Program understanding : challenge for the 1990s," *IBM Systems Journal,* vol. 28, no. 2, pp. 294-306, 1989.

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

[15] A. Egyed, "Fixing Inconsistencies in UML Design Models," in *Proceedings of the 29th international conference on Software Engineering*, 2007.

[16] X. Blanc, I. Mounier, A. Mougeno and T. Mens, "Incremental Detection of Model Inconsistencies based on Model Operations," in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering*, 2009.

[17] S. Caffiau, P. Girard, L. Guittet and X. Blanc, "Vérification de cohérence entre modèles de tâches et de dialogue en conception centrée-utilisateur," *Revue des sciences et technologies de l'information,* vol. 16, no. 5, pp. 9-41, 2011.

[18] M. Eichberg, M. Mezini, K. Ostermann and T. Schâfer, "XIRC: A Kernel for Cross-Artifact Information Engineering in Software Development Environments," in *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004.

[19] C. Nentwich, L. Capra, W. Emmerich and A. Finkelsteiin, "xlinkit: A Consistency Checking and Smart Link Generation Service," *ACM Transactions on Internet Technology,* vol. 2, no. 2, pp. 151-185, 2002.

[20] H. Gall, M. Jazayeri, R. R. Klösch and G. Trausmuth, "Software Evolution Observations Based on Product Release History," in *THE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE*, 1997.

[21] M. D'Ambros and M. Lanza, "Reverse Engineering with Logical Coupling," in *Proceedings of the13th Working Conference on Reverse Engineering*, 2006.

[22] M. Lungu, L. Michele, T. Girba et R. Heeck, «Reverse Engineering Super-Repositories,» chez *14th Working Conference on Reverse Engineering*, 2007.

[23] H. Kagdi, S. Yusuf and J. I. Ma- letic, "Mining sequences of changed-files from version histories," in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006.

[24] J. Rumbaugh, I. Jacobson and G. Booch, "Unified Modeling Language Reference Manual (2nd Edition)," Pearson Higher Education, 2004.

[25] F. Van Rysselberghe and S. Demeyer, "Studying Software Evolution Information by Visualizing the Change History," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004.

[26] X. Blanc, I. Mounier, A. Mougeno and T. Mens, "Detecting model inconsistency through operation-based model construction," in *Proceedings of the 30th international conference on Software engineering*, 2008.

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

[28] G. Antoniol, G. Canfora, G. Casazza and A. De Lucia, "Maintaining traceability links during object-oriented software evolution," *Software : Practice and Experience,* vol. 31, no. 4, pp. 331-355, 2001.

[29] G. Antoniol and Y.-G. Guéhéneuc, "Feature identification : An epidemiological metaphor," *IEEE Transactions on Software Engineering (TSE),* vol. 32, no. 9, pp. 627-641, 2006.

[30] M. Jazayeri, "On architectural stability and evolution," in *da-Europe '02 : Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, London, 2002.

[31] M. Kim, D. Notkin and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *ICSE '07 : Proceedings of the 29th international conference on Software Engineering*, Washington, 2007.

[32] S. Kpodjedo, F. Ricca, P. Galinier and G. Antoniol, "Recovering the evolution stable part using an ecgm algorithm : Is there a tunnel in mozilla ?," in *CSMR '09 : Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, Washington, 2009.

[33] D. Mandelin, D. Kimelman and D. Yellin, "A bayesian approach to diagram matching with application to architectural models," in *ICSE '06 : Proceedings of the 28th international conference on Software engineering*, New York, 2006.

[34] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *IWPC '02 : Proceedings of the 10th International Workshop on Program Comprehension*, Washington, 2002.

[35] Z. Xing, "Analyzing the evolutionary history of the logical design of object-oriented software," *IEEE Transactions on Software Engineering,* vol. 31, no. 10, pp. 850-868, 2005.

[36] Z. Xing and E. Stroulia, "Understanding class evolution in object-oriented software," in *International Conference on Program Comprehension*, 2004.

[37] Z. Xing and E. Stroulia, "Umldiff : an algorithm for object-oriented design differencing," in *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, New York, 2005.

[38] L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering,* vol. 31, no. 2, pp. 166-181, 2005.

[39] T. Zimmermann, P. Weibgerber, S. Diehl and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[40] S. Bouktif, Y.-G. Guéhéneuc and G. Antoniol, "Extracting change-patterns from CVS repositories," in *Susan Elliott Sim and Massimiliano Di Penta, editors, Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, 2006.

[41] J. . B. Kruskal and M. Liberman, "The symmetric time-warping problem : From continuous to discrete," in *David Sanko® and Joseph B. Kruskal, editors, Time Warps, String Edits, and Macromolecules : The Theory and Practice of Sequence Copmparison*, Addison-Wesley, 1983.

[42] M. ZEKKAOUI et A. FENNAN, «HETEROGENEOUS ARTIFACTS CONSTRUCTION FOR SOFTWARE EVOLUTION CONTROL,» chez *the ICSCE 2014 : XII International Conference on Software and Computer Engineering*, Paris, 2014.

[43] M. ZEKKAOUI et A. FENNAN, «Semi-Automated Construction Mechanism of Heterogeneous Artifacts,» *International Journal of Computer Technology and Applications,* vol. 5, n° %15, pp. 1696-1702, 2014.

[44] M. ZEKKAOUI et A. FENNAN, «UNIFIED APPROACH FOR BUILDING HETEROGENEOUS ARTIFACTS AND CONSISTENCY RULES,» chez *the International Conference on Intelligent Information and Network Technology*, Settat, 2013.

[45] M. ZEKKAOUI et A. FENNAN, «UNIFIED APPROACH FOR BUILDING HETEROGENEOUS ARTIFACTS AND CONSISTENCY RULES,» *the Journal of Emerging Technologies in Web Intelligence,* vol. 6, n° %11, pp. 26-31, 2014.

[46] K. H. Bennett, "Software evolution: past, present and future," *Information and Software Technology,* vol. 38, no. 11, pp. 673-680, 1996.

[47] M. M. Lehman, "On understanding laws, evolution, and conversation in the large-program life cycle," *journal of Systems and Software,* vol. 68, no. 9, pp. 1060-1076, 1980.

[48] K. H. Bennett and V. . T. Rajlich, "The staged model of the software lifecycle: A new perspective on software evolution," 2013.

[49] W. S. Humphrey, "The software engineering process : definition and scope," in *the 4th international software process workshop on Representing and enacting the software process*, New York, 1988.

[50] ANSI/IEEE 1002, *IEEE Standard Taxonomy for Software,* IEEE, 1987.

[51] P. Zave, "Classification of research efforts in requirements engineering," *ACM Computing Surveys,* vol. 29, no. 4, pp. 315-321, 1997.

[52] B. Nuseibeh and S. Easterbrook, "Requirements engineering : a roadmap," in *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, New York, 2000.

[53] A. MacKenzie and S. Monk, "From Cards to Code: How Extreme Programming Re-Embodies Programming as a Collective Practice," vol. 13, no. 1, pp. 91-117, 2004.

[54] L. Gasser, W. Scacchi, G. Ripoche and B. Penne, "Understanding Continuous Design in F/OSS Projects," in *the 16th International Conference on Software & Systems Engineering and their Applications*, 2003.

[55] G. . T. HEINEMAN and W. T. COUNCILL, Component-based software engineering : putting the pieces together, Boston, MA: Addison-Wesley Longman Publishing Co, 2001.

[56] R. LAND et I. CRNKOVIC, «Existing approaches to software integration – and a challenge for the future,» chez *Proceedings of Software Engineering Research and Practice in Sweden (SERPS)*, 2004.

[57] A. A. TEREKHOV et C. VERHOEF, «The realities of language conversions,» *IEEE Software,* vol. 17, n° %16, pp. 111-124, 2000.

[58] J. DOMINGUE et P. MULHOLLAND, «Fostering debugging communities on the web,» *Communications of the ACM,* vol. 44, n° %14, pp. 65-71, 1997.

[59] E. S. RAYMOND, The Cathedral and the Bazaar : Musing on Linux and Open Source by an Accidental Revolutionary, Sabastopol: O'Reilly and Associates, 2001.

[60] K. GOFFIN, "Customer support distribution channels : five exploratory case studies," Cranfield School of Management, 1998.

[61] A. van RIEL, V. LILJANDER, J. LEMMINK et S. STREUK, «Boost customer loyalty with online support : the case of mobile telecomms providers,» *International Journal of Internet Marketing and Advertising (IJIMA),* vol. 1, n° %11, pp. 4-23, 2004.

[62] V. COOPER, S. LICHTENSTEIN and R. SMITH, "Knowledge transfer in enterprise information technology support using web-based self-service systems," vol. 1, no. 2, pp. 145-170, 2006.

[63] M. M. LEHMAN, "Programs, life cycles, and laws of software evolution," in *Proceedings of the IEEE*, 1076.

[64] K. H. BENNETT et V. T. RAJLICH, «Software maintenance and evolution : a roadmap,» chez *In ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, New York, 2000.

[65] O. PEYMAN, "Decentralized software evolution," in *Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1)*, 1998.

[66] M. M. LEHMAN et J. F. RAMIL, «Software evolution : background, theory, practice,» *Inf. Process. Lett.,* vol. 88, n° %11-2, pp. 33-44, 2003.

[67] M. M. LEHMAN et J. F. RAMIL, Software Evolution and Feedback : Theory and Practice, John Wiley, 2006.

[68] L. OSTERWEIL, «Strategic directions in software quality,» *ACM Computing Surveys,* vol. 28, n° %14, pp. 738-750, 1996.

[69] D. R. WALLACE et R. U. FUJII, «Software verification and validation : an overview,» *IEEE Software,* vol. 6, n° %13, pp. 10-17, 1989.

[70] D. JACKSON et M. RINARD, «Software analysis : a roadmap,» chez *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, New York, 2000.

[71] M. CIOLKOWSKI, O. LAITENBERGER et S. BIFFL, «Software reviews : The state of the practice,» *IEEE Software,* vol. 20, n° %16, pp. 46-51, 2003.

[72] S. SANKARAN, «Code Reviews,» 2006. [En ligne]. Available: http://today.java.net/pub/a/today/2006/08/17/code-reviews.html.

[73] M. J. HARROLD, «Testing : a roadmap,» chez *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, New York, 2000.

[74] L. OSTERWEIL, «Strategic directions in software quality,» *ACM Computing Surveys,* vol. 28, n° %14, pp. 738-750, 1996.

[75] J. ESTUBLIER, «Software configuration management : a roadmap,» chez *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, New York, 2000.

[76] W. W. ROYCE, «Managing the development of large software systems,» chez *Proceedings of WesCon*, 1970.

[77] A. STROHMEIER, «Cycle de vie du logiciel. École Polytechnique Fédérale de Lausanne, Laboratoire de Génie Logiciel, Département d'Informatique,» 2000.

[78] C. LARMAN et V. R. BASILI, «Iterative and incremental developments : a brief history.,» *Computer,* vol. 36, n° %16, pp. 47-56, 2003.

[79] D. L. PARNAS et P. C. CLEMENTS, «A rational design process : How and why to fake it,» *IEEE Transactions on Software Engineering,* vol. 12, n° %12, pp. 251-257, 1986.

[80] F. P. BROOKS, The Mythical Man-Month : Essays on Software Engineering, Addison-Wesley Publishers, 1995.

[81] G. CUGOLA et C. GHEZZI, «Software processes : a retrospective and a path to the future,» *Soft-*

ware process - Improvement and practice, vol. 4, n° %13, 1998.

[82] J. MCDERMID et K. RIPKEN, Life Cycle Support in the ADA Environment, New York: Cambridge University Press, 1984.

[83] P. ROOK, «Controlling software projects. Software Engineering Journal,» vol. 1, n° %11, pp. 7-16, 1986.

[84] THE STANDISH GROUP, «CHAOS MANIFESTO 2013,» THE STANDISH GROUP, 2003.

[85] M. M. Lehman et L. A. Belady, «Program Evolution: Processes of Software Change,» Academic Press, 1985. [En ligne]. Available: ftp://ftp.umh.ac.be/pub/ftp infofs/1985/ProgramEvolution.pdf.

[86] M. M. Lehman, D. E. Perry et J. Ramil, «Implications of evolution metrics on software maintenance.,» chez *Proc. of the 1998 IEEE Intl. Conference on Software Maintenance*, 1998.

[87] M. M. Lehman et J. F. Ramil, «Software evolution and feedback: Theory and practice,» chez *N. H. Madhavji, J. Fernandez-Ramil, and D. E. Perry, editors, Software Evolution*, 2006.

[88] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry et W. M. Turski, «Metrics and laws of software evolution—The nineties view.,» chez *Proc. of the Fourth Intl. Software Metrics Symposium*, 1997.

[89] J. Capers , «The economics of software maintenance in the twenty first century,» 2006.

[90] D. L. Parnas, «Software Aging,» chez *Proceedings of ICSE 1994*, 1994.

[91] D. E. Perry et A. L. Wolf, «Foundations for the Study of Software Architecture,» *ACM SIGSOFT Software Engineering Notes,* vol. 17, n° %14, 1992.

[92] C. B. Jaktman, L. Leaney et M. Liu, «Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study,» chez *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, 1999.

[93] E. B. Swanson, «The dimensions of maintenance,» chez *proceedings of the 2nd international conference on software engineering*, Los Alamitos, 1976.

[94] F. P. Brooks, «No Silver Bullet,» *IEEE Computer,* vol. 20, n° %14, pp. 10-19, 1987.

[95] D. L. Parnas, «Software aging,» chez *Proceedings of 16th International Conference on Software Engineering*, 1994.

[96] J. D. Kipe, «A framework for characterisation of the degree of integration of software tools,» *Journal of Systems Integration,* vol. 4, pp. 5-32, 1994.

[97] S. Meyers, «Difficulties in Integrating Multiview Editing Environments,» *IEEE Software,* vol. 8, n° %11, pp. 49-57, 1991.

[98] S. P. Reiss, «Connecting Tools Using Message Passing in the Field Environment,» *IEEE Software,* vol. 7, n° %17, pp. 57-66, 1990.

[99] J. C. Grundy, J. G. Hosking, S. Fenwick et W. B. Mugridge, «Connecting the pieces, in Visual Object-Oriented Programming,» Manning/Prentice-Hall, 1995.

[100] M. Ratcliffe, C. Wang, R. J. Gautier et B. R. Whittle, «Dora - a structure oriented environment generator,» *IEEE Software Engineering Journal,* vol. 7, n° %13, pp. 84-190, 1992.

[101] J. C. Grundy, J. G. Hosking et W. B. Mugridge, «Supporting flexible consistency management via discrete change description propagation,» *Software - Practice & Experience,* vol. 26, n° %19, pp. 1053-1083, 1996.

[102] A. Finkelstein, G. Spanoudakis et D. Till, «Managing Interference,» chez *Proceedings of the Sigsoft '96 Workshops – Viewpoints '*, 1996.

[103] B. Nuseibeh, J. Kramer et A. C. W. Finkelstein, «A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification,» *IEEE Transactions on Software Engineering,* vol. 20, n° %110, pp. 760-773, 1994.

[104] G. Spanoudakis, A. C. W. Finkelstein et D. Till, «Overlaps in Requirements Engineering,» *Automated Software Engineering,* vol. 6, n° %12, pp. 171-198, 1999.

[105] W. N. Robinson, «Negotiation Behaviour During Multiple Agent Specification: A Need for Automated Conflict Resolution,» chez *Proceedings, 12th International Conference on Software Engineering (ICSE-12)*, 1990.

[106] A. van Lamsweerde et E. Letier, «Integrating Obstacles in Goal-Driven Requirements Engineering,» chez *Proceedings 20th International Conference on Software Engineering (ICSE-20)*, Kyoto, 1998.

[107] B. Littlewood, «Learning to Live with Uncertainty in our Software,» chez *Proceedings, 2nd International Symposium on Software Metrics*, London, 1994.

[108] G. Cugola, «Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models,» *IEEE Transactions on Software Engineering,* vol. 24, n° %111, pp. 982-1001, 1998.

[109] G. Spanoudakis et A. Zisman, Inconsistency management in software engineering: Survey and open research issues, Handbook of Software Engineering and Knowledge Engineering, 2001.

[110] S. Easterbrook, «Handling Conflict between Domain Descriptions with Computer-Supported Negotiation,» *Knowledge Acquisition,* vol. 3, pp. 255-289, 1991.

[111] J. Leite et P. A. Freeman, «Requirements Validation through Viewpoint Resolution,» *IEEE Transactions on Software Engineering,* vol. 12, n° %112, pp. 1253-1269, 1991.

[112] H. Delugach, Analyzing Multiple Views Of Software Requirements, Conceptual Structures: Current Research and Practice, 1992.

[113] P. Zave et M. Jackson, «Conjunction as Composition,» *ACM Transactions on Software Engineering and Methodology,* vol. 2, n° %14, pp. 379-411, 1993.

[114] M. Jackson, «The meaning of requirements,» *Annals of Software Engineering,* vol. 3, pp. 5-21, 1997.

[115] E. Boiten, J. Derrick, H. Bowman et M. Steen, «Constructive Consistency Checking for Partial Specification in Z,» *Science of Computer Programming,* vol. 35, n° %11, pp. 29-75, 1999.

[116] W. Robinson, «I Didn't Know My Requirements were Consistent until I Talked to My Analyst,» chez *Proceedings of 19th International Conference on Software Engineering (ICSE-97)*, Boston, 1997.

[117] W. Emmerich, F. Finkelstein, C. Montangero, S. Antonelli et S. Armitage, «Managing Standards Comliance,» *IEEE Transactions on Software Engineering,* vol. 25, n° %16, pp. 836-851, 1999.

[118] C. Heitmeyer, B. Labaw et D. Kiskis, «Consistency Checking of SCR-Style Requirements Specifications,» chez *Proceedings of the 2nd International Symposium on Requirements Engineering (RE '95)*, 1995.

[119] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin et J. D. Reese, «Model Checking Large Software Specifications,» *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 24, n° %17, pp. 498-520, 1998.

[120] S. Clarke, J. Murphy et M. Roantree, «Composition of UML Design Models: A Tool to Support the Resolution of Conflicts,» chez *Proceedings of the Int. Conference on Object-Oriented Information Systems*, 1998.

[121] K. Knight, «Unification: A Multidisciplinary Survey,» *ACM Computing Surveys,* vol. 21, n° %11, pp. 93-124, 1989.

[122] S. Easterbrook, A. Finkelstein, J. Kramer et B. Nuseibeh, «Co-Ordinating Distributed ViewPoints: the anatomy of a consistency check,» *International Journal on Concurrent Engineering: Research & Applications,* vol. 2, n° %13, pp. 209-222, 1994.

[123] A. Finkelstein, D. Gabbay, A. Hunter et J. Kramer, «Inconsistency Handling In Multi-Perspective Specifications,» *IEEE Transactions on Software Engineering,* vol. 20, n° %18, pp. 569-578, 1994.

[124] M. Ainsworth, S. Riddle et P. Wallis, «Formal Validation of Viewpoint Specifications,» *IEE Soft-*

ware Engineering Journal, vol. 11, n° %11, pp. 58-66, 1996.

[125] H. Nissen, M. Jeusfeld, M. Jarke, G. Zemanek et H. Huber, «Managing Multiple Requirements Perspectives with Metamodels,» IEEE Software, 1996, pp. 37-47.

[126] W. Robinson et S. Pawlowski, Managing Requirements Inconsistency with Development Goal Monitors, IEEE Transactions on Software Engineer, 1999.

[127] W. Robinson, «Interactive Decision Support for Requirements Negotiation,» *Concurrent Engineering: Research & Applications,* vol. 2, pp. 237-252, 1994.

[128] W. Robinson et S. Fickas, «Supporting Multiple Perspective Requirements Engineering,» chez *Proceedings of the 1st International Conference on Requirements Engineering (ICRE 94)*, 1994.

[129] B. Boehm et H. In, «Identifying Quality Requirements Conflicts,» pp. 25-35, 1996.

[130] T. Gruber, «Towards Principles for the Design of Shared Ontologies Used for Knowledge Sharing,» chez *Proceedings of International Workshop on Formal Ontology*, 1993.

[131] N. Guarino, «The Ontological Level,» chez *Philosophy and the Cognitive Sciences*, 1994.

[132] J. Fiadeiro et T. Maibaum, «Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality,» chez *Proceedings of the Symposium on the Foundations of Software Engineering (FSE 95)*, 1995.

[133] H. Bowman, J. Derrick, P. Linington et M. Steen, «Cross-viewpoint Consistency in Open Distributed Processing,» *IEE Software Engineering Journal,* vol. 11, n° %111, pp. 44-57, 1996.

[134] J. Goguen et S. Ginali, «A Categorical Approach to General Systems Theory,» chez *Applied General Systems Research*, 1978.

[135] S. Easterbrook, J. Callahan et V. Wiels, «V & V Through Inconsistency Tracking and Analysis,» chez *Proceedings of International Woskshop on Software Specification and Design*, Kyoto, 1998.

[136] G. Spanoudakis et A. Finkelstein, «Reconciling requirements: a method for managing interference, inconsistency and conflict,» *Annals of Software Engineering,* vol. 3, n° %11, pp. 433-457, 1997.

[137] M. Maiden, P. Assenova, P. Constantopoulos, M. Jarke, P. Johanneson, H. W. Nissen, G. Spanoudakis et A. G. Sutcliffe, «Computational Mechanisms for Distributed Requirements Engineering,» chez *Proceedings of the 7th International Conference on Software Engineering & Knowledge Engineering (SEKE '95)*, Pitsburg, 1995.

[138] C. Papadimitriou et K. Steiglitz, Combinatorial Optimisation: Algorithms and Complexity, Englewood Cliffs: Prentice-Hall, 1982.

[139] S. Easterbrook et B. Nuseibeh, «Inconsistency Management in an Evolving Specification,» chez *Proc. of 2nd Int. Symposium on Requirements Engineering (RE 95)*, New York, 1995.

[140] L. A. van et E. Letier, «Handling Obstacles in Goal-Oriented Requirements Engineering,» *IEEE Transcactions on Software Engineering,* vol. 6, 2000.

[141] A. Hunter et B. Nuseibeh, «Managing Inconsistent Specifications: Reasoning, Analysis and Action,» *ACM Transactions on Software Engineering and Methodology,* vol. 7, n° %14, pp. 335-367, 1998.

[142] G. Spanoudakis et K. Kassis, «An Evidential Framework for Diagnosing the Significance of Inconsistencies in UML Models,» chez *Proceedings of the International Conference on Software: Theory and Practice*, Bejing, 2000.

[143] H. Bowman, J. Derrick, P. Linington et M. Steen, «Cross-viewpoint Consistency in Open Distributed Processing,» *IEE Software Engineering Journal,* vol. 11, n° %111, pp. 44-57, 1996.

[144] N. A. M. Maiden, P. Assenova, P. Constantopoulos, M. Jarke, H. W. Nissen, G. Spanoudakis et A. G. Sutcliffe, «Computational Mechanisms for Distributed Requirements Engineering,» chez *Proceedings of the 7th International Conference on Software Engineering & Knowledge Engineering (SEKE '95), Pitsburg*, 1995.

[145] J. Mylopoulos, «Telos: Representing Knowledge about Information Systems,» *ACM Transactions on Information Systems,* pp. 325-362, 1990.

[146] OMG, «OMG Unified Modelling Language Specification, V. 1.3a,» OMG, 1999.

[147] L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.

[148] J. Holzmann, «The model checker SPIN,» *IEEE Transactions on Software Engineering,* vol. 23, n° %15, pp. 279-295, 1997.

[149] S. Easterbrook, «Model Checking Software Specifications: An Experience Report,» NASA/WVU Software Research Laboratory, 1997.

[150] R. Bharadwaj et C. Heitmeyer, «Model Checking Complete Requirements Specifications Using Abstraction,» *Automated Software Engineering,* vol. 6, pp. 37-68, 1999.

[151] M. Glinz, «An Integrated Formal Model of Scenarios Based on Statecharts,» chez *Proceedings of the 5th European Software Engineering Conference (ESEC '95)*, 1995.

[152] G. Spanoudakis et A. Finkelstein, «A Semi-automatic process of Identifying Overlaps and Inconsistencies between Requirement Specifications,» chez *Proceedings of the 5th International Conference on Object-Oriented Information Systems (OOIS 98)*, 1998.

[153] A. Zisman, W. Emmerich et A. and Finkelstein, «Using XML to Specify Consistency Rules for Distributed Documents,» chez *the 10th International Workshop on Software Specification and Design (IWWSD-10)*, 2000.

[154] G. Kotonya et I. Sommerville, «Requierements Engineering with Viewpoints,» *Software Engineering Journal,* vol. 11, n° %11, pp. 5-18, 1999.

[155] B. Nuseibeh, S. Easterbrook et A. Russo, «Leveraging Inconsistency in Software Development,» *Computer,* vol. 33, pp. 24-29, 2000.

[156] D. Gabbay et A. Hunter, «Making Inconsistency Respectable 1: A Logical framework for inconsistency in reasoning,» chez *Foundations of Artificial Intelligence Research* , 1991.

[157] D. Gabbay et A. Hunter, «Making Inconsistency Respectable: Part 2 - Meta-level handling of inconsistency,» chez *Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU'93)*, 1993.

[158] M. Jackson, Software Requirements & Specifications: a lexicon of practice, principles and prejudices, Wokingham: ACM Press, 1995.

[159] N. Bashar, E. Steve et R. Alessandra, «Leveraging Inconsistency in Software Development,» *IEEE Computer,* vol. 33, n° %14, pp. 24-29, 2000.

[160] C. LARMAN, Applying UML and Patterns, Prentice Hall, 2001.

[161] A. D. S. IBRAHIM, M. MUNRO et A. AND DERAMAN, «A requirements traceability to support change impact analysis,» *Asean Journal of Information Technology,* vol. 3, n° %11, pp. 1-12, 2005.

[162] A. D. LUCIA, F. FASANO, R. OLIVETO et G. TORTORA, «Fine-grained management of software artefacts: the adams,» *Software: Practice and Experience,* vol. 40, n° %111, p. 1007–1034, 2010.

[163] S. P. REISS, «Incremental maintenance of software artifacts,» *IEEE Transactions on Software Engineering,* vol. 32, n° %19, p. 682–697, 2006.

[164] S. WINKLER et J. PILGRIM, «A survey of traceability in requirements engineering and model-driven development,» *Software & Systems Modeling,* vol. 9, n° %14, p. 529–565, 2009.

[165] T. OLSSON et J. GRUNDY, «Supporting traceability and inconsistency management between software artefacts,» chez *International Conference on Software Engineering and Applications*, 2002.

[166] W. Teitelman, «A tour through Cedar,» *IEEE Software,* vol. 1, n° %12, pp. 44-73, 1984.

[167] L. S. Guy, Common Lisp: the Language, Bedford: Digital Press, 1990.

[168] B. Childs, «Literate programming, a practitioner's view,» chez *Proceedings of the 1991 annual meeting of the Tex User's Group*, 1991.

[169] D. E. Knuth, «Literate programming,» *The Computer Journal,* vol. 27, n° %12, pp. 97-111, 1984.

[170] D. Batory, D. Brant, M. Gibson et M. Nolen, «ExCIS: an integration of domain-specific languages and feature-oriented programming,» chez *Workshops on New Visions for Software Design and Productivity: Research and Applications*, 2001.

[171] B. G. Ryder et F. Tip, «Change impact analysis for object-oriented programs,» chez *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[172] D. M. Ritchie, S. Johnson, M. Lesk et B. Kernighan, «The C programming language,» *bell systems technical journal,* vol. 57, n° %16, pp. 1991-2020, 1978.

[173] C. Duby, S. Meyers et S. Reiss, «CCEL: a metalanguage for C++,» chez *Proceeding of Second Usenix C++ Conference*, 1992.

[174] D. Evans, J. Guttag, J. Horning et Y. M. Tan, «LCLint: a tool for using specifications to check code,» *Software Engineering Notes,* vol. 19, n° %15, pp. 87-96, 1994.

[175] A. Egyed, «Scalable consistency checking between diagrams -- the ViewIntegra approach,» chez *the 16th IEEE International Conference on Automated Software Engineering*, 2001.

[176] C. Gryce, A. Finkelstein et C. Nentwich, «Lightweight checking for UML based software development,» chez *Workshop on Consisteny Problems in UML-based Software Development*, 2002.

[177] J. M. Cobleigh, L. A. Clarke et L. J. Osterweil, «FLAVERS: A finite state verification technique for software systems,» *IBM Systems Journal,* vol. 41, n° %11, pp. 140-165, 2002.

[178] E. Hartmut, P. Ulrike et T. Gabriele, «Fundamental Theory for Typed Attributed Graph Transformation, Graph Transformations,» chez *Second International Conference, ICGT*, 2004.

[179] Rational Software, Best practices for software development teams, A Rational Software Corporation White Paper, 1998.

[180] G. Antoniol, B. Caprile, A. Potrich et P. Tonella, «Design-code traceability for object-oriented systems,» *Annals of Software Engineering,* vol. 9, n° %110, pp. 35-58, 2000.

[181] M. Looise, Inter-level consistency checking between requirements and design artifacts, University of Twente, 2008.

[182] A. Ahmad, H. Basson, L. Deruelle and M. Bouneffa, "A knowledge-based framework for software evolution control," in *INFORSID' 09 : 27th Conference on Informatique des organisation et*

systèmes d'information et de décision, Toulouse, 2009.

[183] A. Kozlenkov et A. Zisman, «Are their design specifications consistent with our requirements ?,» chez *Proceedings. IEEE Joint International Conference on Requirements Engineering*, 2002.

[184] W. L. Poon et A. Finkelstein, «Consistency management for multiple perspective software development,» chez *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops San Francisco*, California, 1996.

[185] A. Finkelstein, «A Foolish Consistency: Technical Challenges in Consistency Management,» chez *Database and Expert Systems Applications*, 2000.

[186] C. Nentwich, W. Emmerich, A. Finkelstein et E. Ellmer, «Flexible consistency checking,» *ACM Transactions on Software Engineering and Methodology,* vol. 12, pp. 28-63, 2003.

[187] L. G. Gnesi, G. Trentanni, F. Fabbrini et M. Fusani, «An automatic tool for the analysis of natural language requirements,» *International Journal of Computer Systems Science & Engineering,* vol. 20, pp. 53-61, 2005.

[188] N. Christian, E. Wolfgang et F. Anthony, «Consistency management with repair actions,» chez *Proceedings of the 25th International Conference on Software Engineering Portland, Oregon: IEEE Computer Society*, 2003.

[189] J. Grundy, J. Hosking et W. B. Mugridge, «Inconsistency management for multiple-view software development environments,» *IEEE Transactions on Software Engineering,* vol. 24, pp. 960-981, 1998.

[190] S. Easterbrook et B. Nuseibeh, «Using ViewPoints for Inconsistency Management,» *Software Engineering Journal,* vol. 11, n° %11, pp. 31-43, 1996.

[191] T. M. Hagensen et B. B. Kristensen, «Consistency in Software System Development: Framework, Model, Techniques & Tools,» *Software Engineering Notes,* vol. 17, n° %15, pp. 58-67, 1992.

[192] R. W. Schwanke et G. E. Kaiser, «Living With Inconsistency in Large Systems,» chez *Proceeding of the International Workshop on Software Version and Configuration Control*, Grassau, 1988.

[193] G. Goth, «Beware the March of this IDE: Eclipse is overshadowing other tool technologies,» *IEEE Software,* vol. 22, n° %14, pp. 108-111, 2005.

[194] A. Ahmad, H. Basson and M. Bouneffa, "Rule-Based Approach for Software Evolution Management," in *IEEE APSSC 2009 : IEEE Asia- Pacific Services Computing Conference*, 2009.