REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université El Hadj Lakhdar – BATNA

Faculté des Sciences et des Sciences de l'ingénieur

Département d'Informatique

N° d'ordre :……
Série :……

Mémoire
présenté en vue de l'obtention du diplôme

Magister en Informatique

Option: **Système d'Information et Connaissance**

SUJET DU MÉMOIRE :

Rétro-ingénierie des modèles comportementaux d'UML 2

Présenté le : 28 /09 /2010

Par : **Wahab ROUAGAT**

Composition du jury :

Dr. Brahim BELATTAR	Président	(Maître de conférences à l'université de Batna).
Dr. Tewfik ZIADI	Rapporteur	(Maître de conférences à l'Université de Paris 6 France)
Dr. Allaoua CHAOUI	Rapporteur	(Maitre de conférences à l'université de Constantine).
Pr. Nacerdine ZAROUR	Examinateur	(Professeur à l'Université de Constantine).
Dr. Ezeddine BILAMI	Examinateur	(Maitre de conférences à l'université de Batna).

Dédicace

La louange est à Allah Le Clément et Le Miséricordieux et que la prière et le salut de mon Seigneur soient sur son Prophète et son Serviteur Mohammed (ﷺ).

Je dédie cet humble travail :

A mes très chers parents, faible témoignage de ma reconnaissance pour leurs inestimables efforts consentis dans l'unique souci de ma réussite et de mon bien être ;

A mes frères, mes sœurs, et tous les membres de la famille ROUAGAT et MIMI ;

A ma chérie ma fidèle fiancée ;

A tous mes amis et à toutes personnes ayant contribué de près ou de loin à la réalisation de ce mémoire ;

A Mr. Haroun CHENCHOUNI.

Remerciements

Au terme de ce modeste travail, je tiens à exprimer ma gratitude et présenter mes vifs remerciements à tous ceux qui ont participé de près ou de loin à sa réalisation. Je rends un hommage particulier :

- Au Dr. Tewfik ZIADI, du Laboratoire d'Informatique (Université de Paris 6, France) : Celui qui a toujours apprécié mes initiatives et qui m'a encouragé à atteindre mes objectifs.
- Au Dr. Allaoua CHAOUI de l'université de Constantine, celui qui m'a aidé dans la réalisation de ce travail, notamment par ses conseils très utiles.
- Ma gratitude va également aux honorables membres de jury qui ont bien voulu prendre le soin de lire ce manuscrit et de juger ce travail :
    - Dr. Brahim BELATTAR (Maître de conférences à l'université de Batna)
    - Pr. Nacerdine ZAROUR (Professeur à l'Université de Constantine).
    - Dr. Ezeddine BILAMI (Maitre de conférences à l'université de Batna).
- A toute personne m'a enseigné, depuis le primaire jusqu'à l'université. Merci à tous ceux qui ont participé à la réalisation de ce travail.
- A tous les collègues, enseignants-chercheurs, responsables et staff administratif du département d'Informatique de l'Université de Batna, en particulier Mr. Boubakeur AZOUI.

Table de matière

TABLE DE MATIÈRE **1**
LISTE DES FIGURES **4**
INTRODUCTION **6**
PARTIE 1 : ÉTAT DE L'ART **9**
CHAPITRE 1 : RÉTRO-INGÉNIERIE **10**
1. Introduction 10
2. Origine et Définition de la rétro-ingénierie 10
3. Apports de la rétro-ingénierie logiciel 11
4. Étapes de la rétro-ingénierie 12
4.1 Collection de données 12
4.1.1 Techniques de collection d'informations 12
4.2 Extraction d'informations pertinentes 13
4.3 Visualisation 14
5. La rétro-ingénierie des logiciels orientés objet 15
6. Conclusion 16
CHAPITRE 2 : LANGAGE DE MODÉLISATION UNIFIÉ (UML) **17**
1. Introduction 17
2. Historique d'UML 17
3. La Méta-Modélisation 18
4. Les Diagrammes d'UML2 19
4.1 Les machines à état 20
4.2 Les diagrammes de séquence 21
4.2.1 Les interactions 21
4.2.2 Les mécanismes de composition 22
5. Conclusion 26
CHAPITRE 3 RÉTRO-INGÉNIERIE DES MODÈLES COMPORTEMENTAUX **27**

1. Introduction 27
2. Etat de l'art sur la rétro-ingénierie des modèles comportementaux d'UML 27
2.1 Travaux de la rétro-ingénierie des machines à états 27
2.2 Travaux de la rétro-ingénierie des diagrammes de séquence d'UML 28
2.2.1 Travaux basés sur l'analyse statique 28
2.2.2 Travaux basés sur l'analyse dynamique 29
2.2.3 Travaux basés sur l'analyse hybride 34
2.3 Discussion et conclusion 35
PARTIE 2 : CONTRIBUTION 38
CHAPITRE 4 : APPROCHE INCRÉMENTALE 39
1. Introduction 39
2. Exemple pour illustration 39
3. Approche incrémentale 40
3.1 La Collection de traces 41
3.2 Construction incrémentale du diagramme de séquence 45
3.2.1 Détection des opérateurs 48
a) Fragment combiné avec l'opérateur "loop" 48
b) Fragment combiné avec l'opérateur "opt" 51
c) Fragment combiné avec l'opérateur "alt" 53
d) Fragment combiné avec l'opérateur "seq" 56
4. Conclusion 57
CHAPITRE 5 IMPLÉMENTATION 58
1. Outils d'implémentation 58
1.1 Outils de collection de traces 58
1.2 Stockage des données 61
1.3 API UML2 63
2. Application de l'approche sur l'exemple *Vente* 64

2.1 Cas d'utilisation 64

CONCLUSION ET PERSPECTIVES.....70

1. Évaluation 70

2. Discussion des résultats.....70

3. Perspectives 71

RÉFERENCES.....72

ANNEXES.....76

(1) Code de détection des fragments combinés de type "loop".....76

(2) Code source de l'application "Vente".....77

Liste des figures

FIG.1.1. MODÈLE D'ARCHITECTURE DES OUTILS PAR [3] 11

FIG. 1.2. ÉTAPES PRINCIPALES DE LA RÉTRO-INGÉNIERIE 12

FIG. 1.3. GRAPHE D'INTERACTION GÉNÉRÉ AVEC PROGRAM EXPLORER [44] 15

FIG. 2.1. L'ARCHITECTURE À QUATRE NIVEAUX DE L'OMG 19

FIG. 2.2. EXEMPLE DE MACHINE À ÉTAT 21

FIG. 2.3. UNE PARTIE DU MÉTA-MODÈLE DES DIAGRAMMES DE SÉQUENCE D'UML2 22

FIG. 2.4. EXEMPLE D'UN DS DANS UML2.0 ET SES CONCEPTS 23

FIG. 2.5. EXEMPLE DES RÉFÉRENCES VERS LES DS DANS UML2 23

FIG. 2.6. EXEMPLE D'UN DIAGRAMME DE SÉQUENCE COMBINÉ 25

FIG. 2.7. EXEMPLE D'UN DIAGRAMME DE VUE D'ENSEMBLE D'INTERACTION DANS UML2. 25

FIG. 3.1. CFG ET SON DIAGRAMME DE SÉQUENCE CORRESPOND [1] 29

FIG. 3.2. DS APRÈS L'EXÉCUTION DE DEUX ITÉRATIONS D'UNE BOUCLE [7] 30

FIG. 3.3. ARCHITECTURE DE JAVAVIS [7] 31

FIG. 3.4. MÉTA-MODÈLE DE DIAGRAMME DE SCÉNARIO BRIAND ET AL. [12] 32

FIG. 3.5. MÉTA-MODÈLE DE TRACE BRIAND ET AL. [12] 32

FIG. 3.6. APPARIEMENT ENTRE TRACE ET SÉQUENCE MÉTA-MODÈLES BRIAND ET AL. [12] 33

FIG. 4.1. LA CONSTRUCTION INCRÉMENTALE DU DIAGRAMME DE SÉQUENCE 41

FIG. 4.2. TRACES DE L'APPLICATION VENTE AVEC LES DONNÉES D'ENTRÉE DE CHAQUE TRACE 42

FIG. 4.3. EXEMPLE DE TRACE 43

FIG. 4.4. LA DÉCOMPOSITION DES TRACES EN BLOCS 48

FIG. 4.5. DS DE *VENTE* APRÈS APPLICATION DE L'ALGORITHME DÉTECTION DE LOOP. 50

FIG. 4.6. DS DE *VENTE* APRÈS L'APPLICATION DE L'ALGORITHME DÉTECTION DE OPT 53

FIG. 4.7. DS FINAL DE L'APPLICATION *VENTE* 57

FIG. 5.1. CODE SOURCE DE L'EXEMPLE LOOP.JAVA 58

FIG. 5.2. VERSION NON INSTRUMENTÉ DE LA MÉTHODE M1(). 59

FIG. 5.3. VERSION INSTRUMENTÉ DE LA MÉTHODE M1(). 59

FIG. 5.4. TRACE 2 DE L'APPLICATION VENTE GÉNÉRÉE PAR L'OUTIL MODEC 60

FIG. 5.5. TRACE2 ADAPTÉ DE L'APPLICATION VENTE 61

FIG. 5.6. LE MODÈLE RELATIONNEL INTERMÉDIAIRE 62

FIG. 5.7. VUE INITIALE DE LA TABLE MESS_TR 65

FIG. 5.8. VUE FINALE DE LA TABLE MESS_TR 66

FIG. 5.9. VUE FINALE DE LA TABLE COMBINED_FRAGMENTS.....67
FIG. 5.10. VUE FINALE DE LA TABLE OPERANDS 67
FIG. 5.11. LE DIAGRAMME DE SÉQUENCE GÉNÉRÉ AFFICHÉ PAR UN ÉDITEUR ECLIPSE 68
FIG. 5.12. UNE PARTIE DU DS REPRÉSENTE UN FRAGMENT COMBINÉ SOUS FORME D'UN DOCUMENT UML.....69

Introduction

Les systèmes logiciels sont devenus très importants dans notre vie quotidienne. Ils contrôlent les machines de payement, les instruments médicales, les systèmes de télécommunication et plusieurs d'autres systèmes. A cause de leur intérêt et leur vaste domaine d'application, tout comportement non désiré ou erreur de fonctionnement peut engendrer des pertes d'argent, arrêts de services et des risques de vies. La fiabilité de logiciels devient donc un facteur suprême. D'autre part, l'évolution de besoins et le changement continu de contextes de fonctionnement nécessitent la mise à jour et la maintenance courante de ces systèmes.

Dans le cycle de vie de développement des applications logicielles, l'étape de maintenance est l'étape la plus critique, et elle consomme environ 90% de l'ensemble des ressources utilisées [5]. La tache principale de la maintenance est de comprendre l'application en analysant sa structure et l'organisation de ses composants afin de fixer les problèmes et/ou d'ajouter des nouvelles caractéristiques [5]. Utiliser directement le code source pour comprendre une application est une tache difficile pour trois raisons principales : 1) les applications sont de plus en plus complexes et leur code source est dispersé sur des milliers voir des millions de lignes de code. 2) les applications, principalement les anciennes (connues par l'appellation « legacy systems »), sont souvent codées dans des langages de programmations obsolètes peu connus actuellement par les mainteneurs[1]. 3) la maintenance est souvent réalisée par des individus qui n'ont pas été impliqués dans le codage de l'application. Ainsi, les mainteneurs ont besoin de disposer d'une représentation abstraite permettant de gérer la complexité des applications. D'autre part, il est important que cette représentation soit décrite dans un langage « universel » connu par un très grand nombre de mainteneurs.

La rétro-ingénierie logicielle est le champ du génie logiciel permettant de répondre aux besoins des mainteneurs. Elle est définie comme le domaine qui a pour but d'analyser le code du logiciel et de le représenter automatiquement sous une forme abstraite pour que sa

[1] Nous utilisons dans le reste de ce document le « mainteneur » pour désigner toute personne impliquée dans la tache de maintenance.

maintenance et sa compréhension soient plus faciles [6]. La rétro-ingénierie des modèles UML s'intéresse à représenter le logiciel sous forme de modèles UML. En effet, Le langage unifié de modélisation UML (Unified Modeling Language) propose un ensemble de notations (appelés diagrammes UML[1]) pour représenter d'une manière abstraite les différents aspects d'une application. Les deux aspects principaux considérés dans UML sont *l'aspect structurel* et *l'aspect comportemental*. L'aspect structurel décrit l'architecture statique de l'application. Les diagrammes de classes, de composants sont des exemples de diagrammes UML permettant de représenter l'aspect structurel des applications. L'aspect comportemental décrit principalement les interactions entre les objets pour réaliser les fonctionnalités de l'application. Les diagrammes de séquence et les machines à états sont les principaux diagrammes comportementaux d'UML.

La rétro-ingénierie des modèles UML a attiré beaucoup d'attentions ces dernières années. D'une part au sein des outils CASE UML où cette fonctionnalité est intégrée dans la quasi-totalité des outils UML [63]. D'autre part au sein de la communauté académique où plusieurs travaux de recherche se sont intéressés à ce sujet. Cependant, et comme nous allons voir par la suite les travaux existants, plusieurs constats d'insuffisance sont dégagés :

1. La majorité des travaux existants, autour de la rétro-ingénierie des modèles UML, s'intéresse seulement à l'aspect structurel notamment les diagrammes de classes.
2. Le peu existant qui s'intéresse à l'aspect comportemental et principalement aux diagrammes de séquence ne génère que des modèles correspondants à un seul scénario[2] particulier d'exécution du système.
3. La majorité des travaux existants ne supporte pas la nouvelle version des diagrammes de séquence dans UML2 et en particulier les opérateurs d'interaction récemment introduits.
4. Les solutions proposées sont très liées aux langages et plateformes dans lesquels se tournent les systèmes étudiés.

Dans ce travail, nous proposons en premier lieu un état de l'art autour du domaine de la rétro-ingénierie logicielle. Nous nous sommes principalement intéressés à la rétro-ingénierie des diagrammes de séquence d'UML. Nous présentons une étude comparative des travaux existants dans ce contexte. Par la suite, nous proposons de revisiter ce problème de

[1] Dans le reste de ce document nous utilisons conjointement « diagramme UML » et « modèle UML » pour désigner la même chose.

[2] Le mot scénario désigne l'ensemble de données d'entrée choisi dans une instance d'exécution d'un système.

rétro-ingénierie des diagrammes de séquence d'UML2 en produisant une nouvelle approche. Cette approche est basée sur une analyse dynamique du système et prend à l'entrée un ensemble de traces collectées dans les différentes sessions d'exécutions du système considéré. Nous avons développé des heuristiques qui permettent de détecter les différents types d'opérateurs d'interaction et construire un diagramme de séquence d'UML représentant le comportement global du système étudié. Notre approche est implémentée dans un prototype outil et validée sur une étude de cas.

Le reste de ce document est divisé en deux partie : 1) État de l'art 2) Contribution. Dans la première partie nous présenterons en premier lieu le domaine de la rétro-ingénierie et les travaux existants dans le contexte des modèles comportementaux d'UML. La deuxième partie va présenter notre approche en détaillant ses différentes étapes.

Partie 1 : État de l’art

Chapitre 1 : Rétro-ingénierie

1. Introduction

Dans ce chapitre nous explorons la rétro-ingénierie en présentant ses différents domaines d'application dans la littérature. Nous nous sommes intéressés particulièrement à la rétro-ingénierie liés au domaine du génie logiciel. Nous déterminons ses apports et ses étapes principales, ses approches existantes et les différents niveaux d'abstraction visés par la rétro-ingénierie orientée objet.

2. Origine et Définition de la rétro-ingénierie

D'après [4], la rétro-ingénierie revient aux années de la révolution industrielle. Elle est semblable à la recherche scientifique où les chercheurs essayent d'inventer des plans et des cartes aux phénomènes naturels. La seule différence est que la rétro-ingénierie concerne les objets créés par l'homme (des appareils mécanique, des composants électronique, des programmes informatique, …). D'une façon générale, la rétro-ingénierie est le processus d'extraction de connaissances ou de concevoir des représentations à partir de tout ce qui est construit par l'homme.

Dans le contexte de l'ingénierie logicielle, le terme rétro-ingénierie a été défini en 1990 par Chikofsky et Cross dans [3] comme le processus d'analyse d'un système pour (1) identifier les composants du système et leurs interrelations, et (2) créer des représentations du logiciel dans une autre forme ou dans un niveau d'abstraction plus élevé. La rétro-ingénierie a été traditionnellement vue comme un processus en deux étapes : l'extraction des informations et l'abstraction. L'étape d'extraction des informations analyse les artefacts du logiciel pour recueillir des données brutes, tandis que l'étape d'abstraction crée des vues et documents orientés utilisateur à partir des données brutes [25]. Chikofsky et Cross ont présenté une structure de base pour les outils implémentant la rétro-ingénierie (voir Fig.1.1.). Le logiciel étudié est analysé et les résultats de cette analyse sont stockés dans une base d'informations. Par la suite, ces informations sont utilisées pour produire différentes vues du logiciel, comme les diagrammes, les rapports, et les métriques.

.

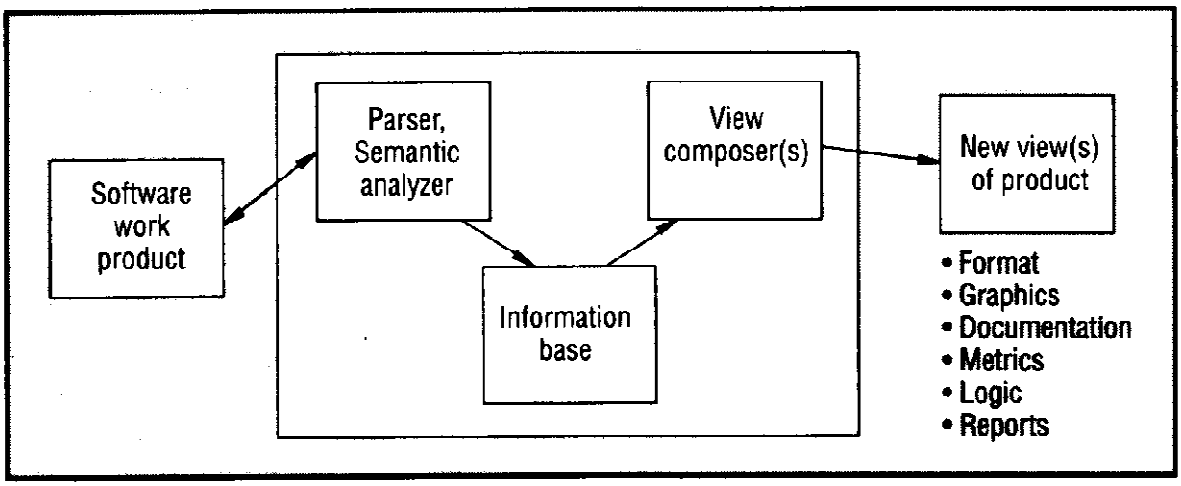

Fig.1.1. Modèle d'architecture des outils par [3]

3. Apports de la rétro-ingénierie logiciel

La rétro-ingénierie est très utile aux développeurs des logiciels. En effet, ils l'utilisent comme un moyen de coopération et collaboration autour des logiciels qui ne disposent pas (ou qui disposent partiellement) de la documentation. Ils peuvent l'utiliser aussi pour étudier et améliorer les logiciels concurrents.

De plus, la rétro-ingénierie permet d'évaluer la qualité et la robustesse d'un logiciel. Elle est également le moyen essentiel dans les différentes taches de l'étape de maintenance qui représente une étape très importante dans le cycle de vie de logiciel. En effet, une enquête effectuée aux USA en 1986 auprès de 55 entreprises a révélé que 53% du budget total d'un logiciel est affecté à la maintenance [27]. La rétro-ingénierie facilite les taches de maintenances car elle permet de proposer aux personnes intervenant dans la maintenance des représentations plus abstraites qui leur permet de comprendre le logiciel pour le tenir à jour en fonction des nouveaux besoins des utilisateurs.

En plus de faciliter la maintenance de logiciels, six autres objectifs de la rétro-ingénierie ont été identifié dans [3] :

- Faire face à la complexité.
- Générer différentes vues.
- Recouvrir des informations perdues.
- Détecter l'effet de bord.
- Synthétiser des abstractions de haut niveau.
- Faciliter la réutilisation.

4. Étapes de la rétro-ingénierie

Le schéma générale de la rétro-ingénierie comme il a été identifié dans [3] se compose de trois étapes principales est illustré dans la Figure Fig. 1.2. : 1) collection de données, 2) extraction d'informations pertinentes, 3) visualisation. Nous décrivons dans les sections suivantes chacune de ces trois étapes.

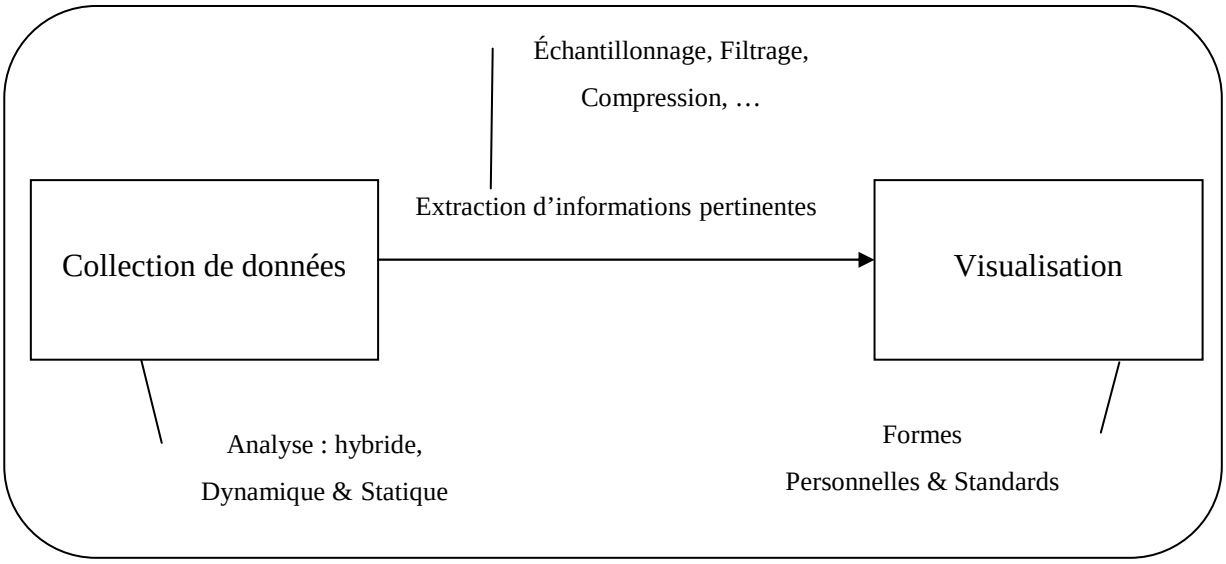

Fig. 1.2. Étapes principales de la rétro-ingénierie

4.1 Collection de données

Cette étape consiste à analyser le système pour obtenir les données nécessaires à la compréhension de ses différents aspects structurels et comportementaux. Il existe plusieurs techniques de collection d'informations qui peuvent être classées en trois catégories : analyse statique, analyse dynamique et analyse hybride.

4.1.1 Techniques de collection d'informations

1) L'analyse statique. Dans ce type de techniques, les informations sont collectées directement auprès du code compilé ou du code source d'un programme sans l'exécuter. Les informations peuvent être recueillies dans le cadre statique, par le contrôle des flux de données ou par l'insertion d'annotations dans le code pour marquer les points d'intérêt et de guider cette analyse. Un exemple d'outils qui suivent ce type d'analyse, comporte les désassembleurs qui permettent d'obtenir à partir de code machine binaire un texte dans un langage assembleur, et les dé-compilateurs qui essayent de produire des codes sources de langages de haut-niveau à partir des codes binaires.

2) L'analyse dynamique. Selon [18], l'analyse dynamique est l'analyse des propriétés d'un programme en cours d'exécution. L'importance de l'analyse dynamique est dérivée de ses caractéristiques essentielles qui sont :

1) *La précision des informations* : Avec l'analyse dynamique, il est possible de collecter précisément les informations pertinentes qui ne comprennent que des comportements possibles.
2) *Dépendance aux entrées des programmes* : l'analyse dynamique est caractérisée par le fait que ses résultats sont liés aux données d'entrées et de sorties de programme.. Cela permet d'aider les développeurs à comprendre le logiciel en étudiant comment il change son comportement en fonction des différentes entrées et sorties [26].

Il existe plusieurs techniques d'analyse dynamique pour la collection des données. Parmi ces techniques nous citons : les outils de contrôle du système, l'instrumentation du code source, l'instrumentation du byte code (code intermédiaire), l'instrumentation de la machine virtuelle et l'utilisation des débogueurs personnalisés [5].

Par ce type d'analyse la collection des informations se traduit par la génération des traces. Une trace montre les valeurs des variables du programme, l'état de la pile d'exécution, les occurrences d'appels des méthodes, l'état des threads ou toute autre information d'exécution [26].

3) Analyse hybride. L'analyse hybride permet de combiner conjointement l'analyse statique et l'analyse dynamique pour la collection des données. Cela permet de profiter des avantages des deux types d'analyse. En effet**,** les informations collectées par l'analyse statique peuvent fournir une connaissance générale sur le comportement du système et l'analyse dynamique permet de l'améliorer.

4.2 Extraction d'informations pertinentes

L'étape précédente permet de collecter des données brutes et de grand volume. Donc, Il devient nécessaire de nettoyer ces données avant qu'elles soient utilisées dans l'étape de visualisation, c'est le rôle de l'étape d'extraction d'informations pertinentes. Particulièrement dans l'analyse dynamique, Une trace de grand volume pose un problème majeur qui est la perte de l'information utile dans la totalité d'informations collectées. Par conséquence, plusieurs techniques de l'extraction d'informations pertinentes ont été développées telles que l'échantillonnage et la technique de cacher des composants [23] :

- **L'échantillonnage**. L'échantillonnage est un moyen intéressant de réduire la taille de la trace, il a été utilisé dans AVID [45]. Il s'agit de choisir seulement un échantillon de la trace de l'analyse au lieu d'utiliser toute la trace. Cependant, trouver les bons paramètres d'échantillonnage n'est pas une tâche facile, et même si certains paramètres vont bien pour comprendre une caractéristique, il n'est pas évident qu'ils soient utiles avec une autre.
- **Cacher de Composants**. Une autre façon de réduire la taille de la trace est de cacher certaines de ses composants. Par exemple, l'analyste peut décider de cacher toutes les invocations d'une méthode spécifique.

4.3 Visualisation

La visualisation est une étape cruciale pour la rétro-ingénierie. La façon dont l'information est présentée aux développeurs ou aux mainteneurs affecte fortement l'utilité de l'étape d'analyse ou les techniques d'extraction d'informations pertinentes. Il existe deux choix de visualisation, soit d'utiliser des diagrammes bien définis et largement adoptés (standards) comme ceux d'UML, ou d'utiliser des notations propre à l'analyste (Personnelle) où la visualisation constitue le principe d'une technique de rétro-ingénierie [25].

Des exemples d'outils capables de visualiser l'information extraite statiquement incluent l'outil Rigi [43], CodeCrawler [41], et sv3D [42]. Certains de ces outils, comme Rigi, visent à montrer des vues d'architecture. Autres outils, comme sv3D, fournissent des techniques de visualisation en 3D de paramètres d'artefacts du logiciel. L'outil CodeCrawler combine la capacité de montrer les entités des logiciels et leurs relations, avec la possibilité de visualiser les paramètres du logiciel en utilisant les vues poly-métriques, qui montrent les différents paramètres en utilisant la largeur, la longueur et la couleur. Certains outils de visualisation, tels que Program Explorer [44], permet de visualiser des informations dynamiques (e.g. les interactions entre les objets) en les modélisant sous forme d'un graphe orienté appelé graphe d'interaction (voir Fig. 1.3.). Les nœuds représentent les objets et les arcs représentent les méthodes appelées. Les arcs sont étiquetés par les temps d'appel et retour des méthodes.

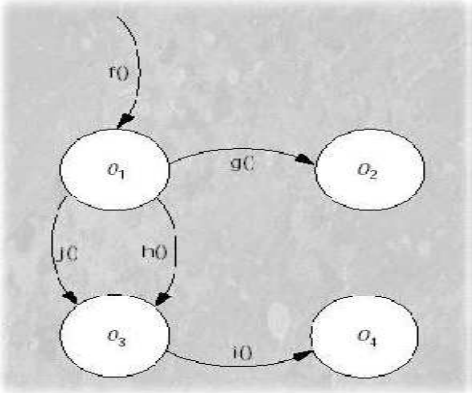

Fig. 1.3. Graphe d'interaction généré avec Program Explorer [44]

5. La rétro-ingénierie des logiciels orientés objet

La compréhension des d'applications orientées-objet est une tâche difficile qui se doit de surmonter les particularités de ce paradigme. En effet, même si ses particularités sont les points forts de ce type de programmation, elles élèvent l'analyse et la compréhension des applications orientées-objet à un autre niveau de difficulté. Le cas du polymorphisme et la liaison dynamique rendent les traditionnels outils d'analyse inadéquats.

L'objectif essentiel de la rétro-ingénierie orientée-objet est de voir le système par une vue descriptive dans le niveau d'abstraction adéquat [6]. Pour mieux comprendre les fonctionnalités d'un logiciel il est très nécessaire de choisir le bon niveau d'abstraction [23] :

- **Niveau instruction**

Ce niveau inclut l'exécution de chaque instruction du code. La majorité des outils ne supporte pas cette vue à l'exception des débogueurs. Ce niveau d'abstraction va bien avec les activités spécifiques de maintenance comme la fixation des bugs.

- **Niveau communication inter-thread**

Ce niveau s'occupe de visionner les interactions entre les threads du système. Peu sont les outils qui prisent en charge ce type d'interaction, et si c'est le cas, souvent les autres niveaux sont omis comme dans Jinsight [1O].

- **Niveau objet**

Ce niveau concerne la visualisation des interactions des méthodes entre les objets. Ce niveau peut être utile pour détecter les fuites de mémoire et autres étranglements de performance. La majorité des outils supporte ce niveau.

- **Niveau classe**

Dans ce niveau, les objets de la même classe sont substitués par le nom de leurs classes. Ce niveau, qui est supporté par la majorité des outils, va bien avec les activités qui

requièrent de haut-niveau de compréhension du comportement du système, tels que le recouvrement de la documentation, et la compréhension de quelles classes implémentent une caractéristique particulière.

- **Niveau architectural**

Ce niveau consiste à grouper des classes dans des clusters et montrer comment les composants du système inter-actent les uns les autres.

6. Conclusion

L'activité de la rétro-ingénierie est très importante car elle permet de comprendre un système existant en produisant des artefacts d'un niveau d'abstraction supérieur au code. Dans ce chapitre nous avons présenté les apports de la rétro-ingénierie et nous avons étudié ses étapes principales.

À cause de sa présence importante dans l'industrie, le standard UML et ses modèles étaient l'un des plus importants domaines d'application de la rétro-ingénierie des systèmes orientés-objet. Dans le chapitre suivant nous allons présenter le langage de modélisation unifié (UML) en se concentrant sur les modèles comportementaux. Par la suite, le chapitre 3 présente une étude détaillé autour de la rétro-ingénierie des modèles comportementaux d'UML.

Chapitre 2 : Langage de Modélisation Unifié (UML)

1. Introduction

La description de la programmation par objets a fait ressortir l'étendue du travail conceptuel nécessaire : définition des classes, de leurs relations, des attributs et méthodes, des interfaces etc.

Pour programmer une application, il ne convient pas de se lancer tête baissée dans l'écriture du code : il faut d'abord organiser ses idées, les documenter, puis organiser la réalisation en définissant les modules et étapes de la réalisation. C'est cette démarche antérieure à l'écriture que l'on appelle modélisation, son produit est un modèle [27].

Un modèle est une représentation abstraite et simplifiée (i.e. qui exclut certains détails), d'une entité (phénomène, processus, système, etc.) du monde réel envie de le décrire, de l'expliquer ou de le prévoir. Modèle est synonyme de théorie, mais avec une connotation pratique : un modèle, c'est une théorie orientée vers l'action qu'elle doit servir. Concrètement, un modèle permet de réduire la complexité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative. Il reflète ce que le concepteur croit important pour la compréhension et la prédiction du phénomène modélisé. Les limites du phénomène modélisé dépendent des objectifs du modèle.

Dans ce chapitre nous présentons le langage unifié de modélisation UML avec ses notions et notations les plus importantes en montrant leurs faveurs aux intérêts de la modélisation des systèmes.

2. Historique d'UML

Dans les années 90, une multitude de méthodes ont été proposées afin de renforcer la spécification et la documentation de logiciel (Booch, OMT, OOSE, méthodes orientées objet, et autres). Cette diversité a engendré un problème d'interopérabilité entre les outils de modélisation qui a ralenti l'adoption des méthodes de développement de logiciels basés sur les modèles. Dans ce contexte troublé par différents langages de modélisation, Grady Booch, Ivar Jacobson et Jim Rumbaugh ont proposé le langage unifié de modélisation (UML - Unified Modeling Language) [28]. UML a été conçu pour unifier différentes notations graphiques de modélisation, être le plus général possible et être extensible afin de prendre en charge des contextes non prévus. UML est basé sur les fondements de l'approche objets, c'est

un langage visuel dédié à la modélisation et c'est un support fondamental pour générer des programmes.

La première version d'UML a été publiée en 1997 par l'OMG. Depuis, UML est devenue la référence pour la création de modèles pour l'analyse et la conception de logiciels.

3. La Méta-Modélisation

Un métamodèle est une spécification utilisée pour créer des modèles conformes (à cette spécification), par exemple, le métamodèle d'UML.

Depuis ses premières versions, le standard UML est caractérisé par sa sémantique définie par une approche de méta-modélisation. Un méta-modèle est la définition des constructions et des règles de création des modèles. Le méta-modèle d'UML définit donc la structure que doit respecter tout modèle UML. Le méta-modèle d'UML1.x est défini dans un seul document, cependant le standard UML2 est maintenant divisé en deux documents : UML2 Infrastructure [29] et UML2 Superstructure [28]. UML Infrastructure décrit les constructions fondamentales utilisées pour la définition d'UML2 sous forme d'une librairie d'infrastructure (Infrastructure Library). UML Superstructure réutilise et raffine la librairie d'infrastructure et définit le méta-modèle proprement dit, vu par les utilisateurs.

L'approche de méta-modélisation adoptée par l'OMG est connue comme une hiérarchie à quatre niveaux [36] (voir la Figure 2.1) :

Niveau méta-méta-modèle (M3). M3 est le niveau méta-méta-modèle, il définit le langage de spécification du méta-modèle. Le MOF (Meta Object Facility) est un exemple d'un méta-méta-modèle.

Niveau méta-modèle (M2). M2 est le niveau méta-modèle. Le méta-modèle d'UML se situe à ce niveau et il est spécifié en utilisant le MOF, c.à.d. les concepts du méta-modèle d'UML sont des instances des concepts de MOF. La Figure 2.1 montre deux méta-classes du méta-modèle UML : Class et Association.

Niveau modèle (M1). M1 correspond au niveau des modèles UML des utilisateurs. Les concepts d'un modèle UML sont des instances des concepts du méta-modèle UML. La Figure 2.1 montre un extrait de diagramme de classes pour une application bancaire contenant deux classes Account et Customer liées par une association UML. Les deux classes sont des instances de la méta-classe Class et le lien est une instance de la méta-classe Association du méta-modèle UML.

Niveau objets (M0). M0 correspond au niveau des objets à l'exécution. Il s'agit ici de deux objets must et acc des instances des deux classes Customer et Account respectivement.

Le méta-modèle d'UML est décrit en utilisant une partie de la notation d'UML lui même. Les concepts suivants sont utilisés :

– Les classes d'UML, pour décrire les méta-classes.

– Les attributs, pour décrire les propriétés attachées à une méta-classe.

– Les associations, pour décrire des liens entre les méta-classes.

– Les paquetages (packages), pour regrouper les méta-classses par domaine.

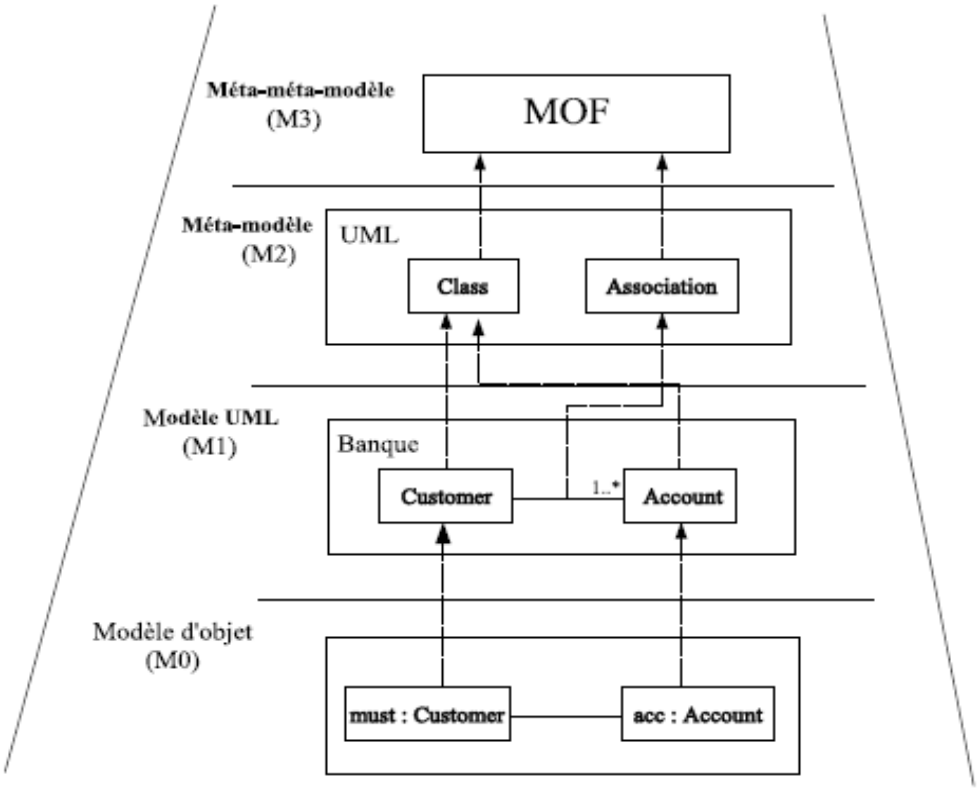

Fig. 2.1. L'architecture à quatre niveaux de l'OMG

4. Les Diagrammes d'UML2

La notation UML est décrite sous forme d'un ensemble de diagrammes. La première génération d'UML (UML1.x), définit neuf diagrammes pour la documentation et la spécification des logiciels. Dans UML2.0 Superstructure [28], quatre nouveaux diagrammes ont été ajoutés : il s'agit des diagrammes de structure composite (Composite structure diagrams), les diagrammes de paquetages (Packages diagrams), les diagrammes de vue

d'ensemble d'interaction (Interaction overview diagrams) et les diagrammes de synchronisation (Timing diagrams). Ils sont regroupés dans deux classes principales :

Diagrammes statiques. Regroupe les diagrammes de classes, les diagrammes d'objets, les diagrammes de structure composite, les diagrammes de composants, les diagrammes de déploiement, et les diagrammes de paquetages.

Diagrammes comportementaux. Regroupe Les diagrammes de séquence, les diagrammes de communication (nouvelle appellation des diagrammes de collaboration d'UML1.x), les diagrammes d'activités, les machines à états, les diagrammes de vue d'ensemble d'interaction, et les diagrammes de synchronisation.

Les travaux existants autours de la rétro-ingénierie des modèles comportementaux d'UML s'intéressent particulièrement aux deux types de modèles à savoir : les diagrammes de séquence et les machines à états. Ainsi, nous présentons en détail ces deux types de modèles dans les sections suivantes.

4.1 Les machines à état

Les scénarios tels que les diagrammes de séquence d'UML décrivent la coopération de plusieurs objets pour réaliser une fonctionnalité. Cependant la spécification du comportement d'un seul objet est le domaine des techniques de modélisation orientées états. Les machines à états et ses variantes comme les statecharts de Harel [62] sont des exemples de ces techniques.

Une machine à états d'UML (une variante des statecharts de David Harel) est généralement associée à une classe particulière. Elle décrit le comportement complet de tous les objets instances de la classe. Le comportement d'un objet est décrit par un ensemble d'états et de transitions. La Figure 2.2 montre les concepts de base d'une machine à état d'UML. Les états décrivent certaines conditions sur l'objet et les transitions indiquent les réactions possibles de l'objet avec le respect des conditions de l'état courant et les événements qui déclenchent ces réactions. Une transition est définie par :

– Un événement déclencheur de la transition.

– Une condition garde de la transition.

– Une action exécutée lorsque la transition est tirée.

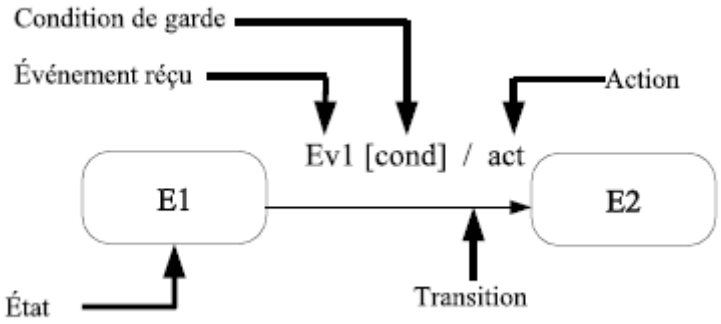

Fig. 2.2. Exemple de machine à état

4.2 Les diagrammes de séquence

Montrent les interactions entre les objets dont la représentation se concentre sur la séquence des interactions selon un point de vue temporel. Dans ce qui suit, nous présentons en détail ce type de diagramme car il est l'objet de notre recherche.

Le standard UML2 a apporté une refonte majeure aux diagrammes de séquence. Il est maintenant possible de décrire et spécifier les exigences sous forme d'un ensemble de diagrammes de séquence de base et par la suite de les composer en utilisant des opérateurs pour obtenir des scénarios plus complexes. Aussi, les diagrammes de séquence dans UML2 sont maintenant considérés comme des collections partiellement ordonnées d'événements (au lieu des collections ordonnées de messages dans UML1.x), ce qui introduit la concurrence et l'asynchronisme et permet la définition de comportements plus complexes [36].

4.2.1 Les interactions

Les diagrammes de séquence dans UML2 [29] sont définis dans la section Interaction de UML2 Superstructure. La Figure 2.3 résume cette section. La méta-classe Interaction spécifie une unité de comportement décrivant l'échange d'information entre un ensemble d'objets dans un diagramme de séquence. La méta-classe Lifeline spécifie un objet dans une interaction. InteractionFragment est une partie d'interaction. La composition entre une interaction et InteractionFragment spécifie qu'une interaction peut englober un ensemble de sous-interactions. CombinedFragment définit une composition d'un ensemble de InteractionOperand en utilisant un opérateur d'interaction (nous discutons plus en détail la composition des diagrammes de séquence ci-dessous). Chaque message est défini par deux événements : un événement spécifiant son émission et un autre spécifiant sa réception. L'événement d'émission d'un message précède toujours l'événement de sa réception [29].

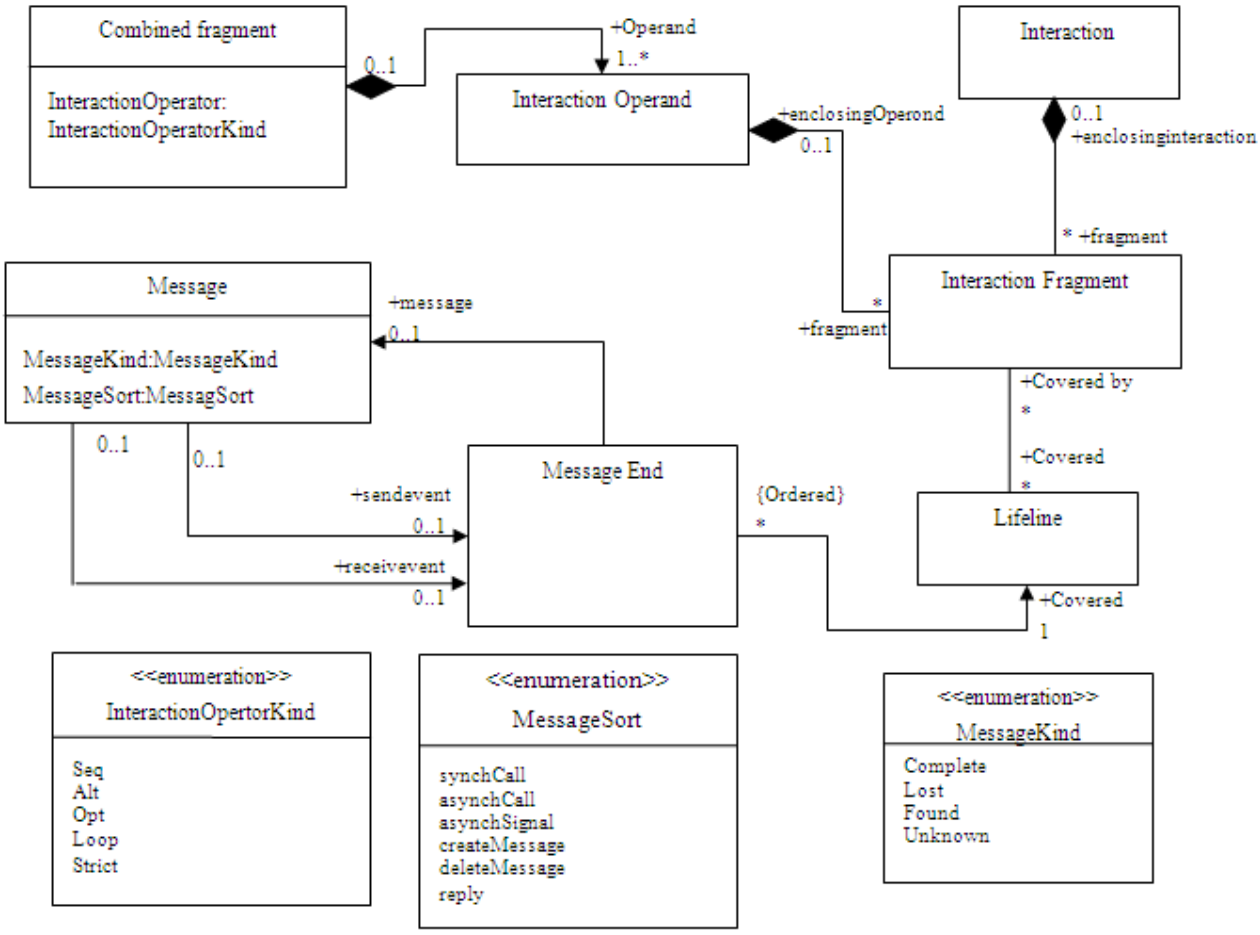

Fig. 2.3. Une partie du méta-modèle des diagrammes de séquence d'UML2

Une nouvelle notation des diagrammes de séquence a été introduite dans UML2. Un diagramme de séquence d'UML2 est maintenant décrit par un cadre rectangulaire libellé par le mot clé sd (pour Sequence Diagram) dans le coin gauche supérieur. La Figure 2.4 montre la notation d'un diagramme de séquence simple dans UML2.

4.2.2 Les mécanismes de composition

Comme nous l'avons mentionné ci-dessus, la grande nouveauté dans la nouvelle génération des diagrammes de séquence d'UML2 est la possibilité de composer les diagrammes de séquence pour spécifier des comportements plus complexes. La composition des diagrammes de séquence d'UML2 est introduite par le moyen de deux mécanismes de base : les références vers les DS et les opérateurs d'interactions.

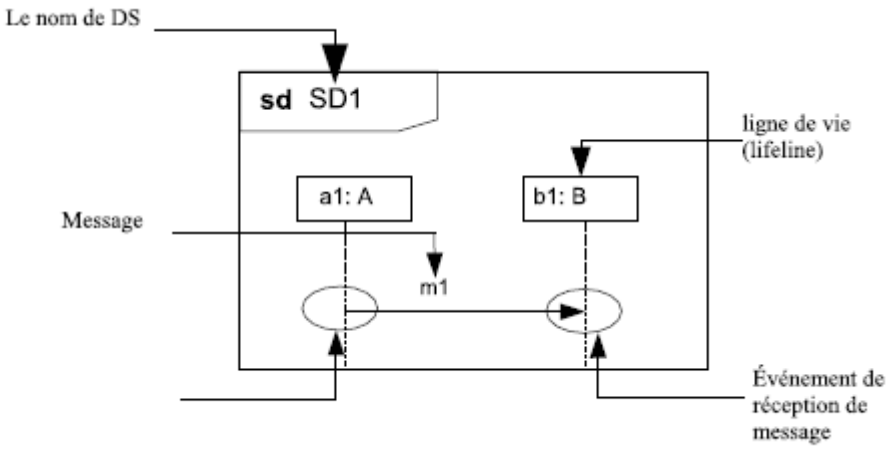

Fig. 2.4. Exemple d'un DS dans UML2.0 et ses concepts

Les références vers les DS. UML2 introduit la notion d'occurrence d'interaction pour permettre à un diagramme de séquence de référencer un autre diagramme de séquence. Une occurrence d'interaction est en quelque sorte un raccourci [29] vers une copie du contenu d'une interaction. Le diagramme de séquence SD2 dans la Figure 2.5 référence le diagramme de séquence SD1. La référence vers un diagramme de séquence est décrite par un cadre rectangulaire libellé par le mot clé ref dans le coin gauche supérieur ; le nom de DS référencé est spécifié au milieu du cadre.

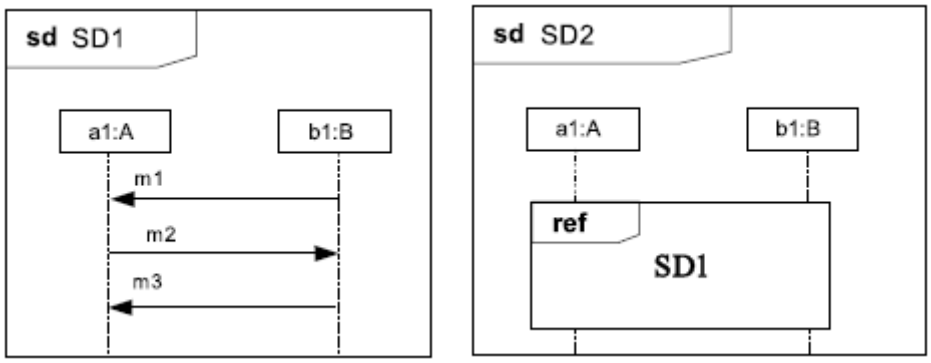

Fig. 2.5. Exemple des références vers les DS dans UML2

Les opérateurs d'interaction. Les DS d'UML2 peuvent être composés en utilisant un ensemble d'opérateurs appelés Opérateur d'interaction. Les opérateurs de composition possibles dans UML2 sont définis dans le méta-modèle par l'énumération InteractionOperatorKind (voir Figure 2.3). Un opérateur d'interaction est associé à une interaction combinée (CombinedFragment -voir Figure 2.3) et il compose un ensemble d'opérandes d'interaction (InteractionOperand - voir la Figure 2.3). Nous décrivons dans ce qui suit les opérateurs fondamentaux :

– La séquence (seq). L'opérateur de séquence seq spécifie une séquence faible entre les comportements des opérandes [28]. La composition séquentielle faible signifie que les événements situés dans le premier DS sur une ligne de vie particulière doivent être exécutés avant les événements de deuxième DS situés sur la même ligne de vie [28]. Mais seq n'impose pas de synchronisation entre des lignes de vie différentes.

– L'alternative (alt). L'opérateur alt définit un choix entre les comportements d'un ensemble de DS.

– L'itération (loop). L'opérateur loop spécifie l'itération de comportement d'un DS.

– La séquence forte (strict). L'opérateur strict spécifie une composition séquentielle forte. Au contraire de l'opérateur seq, où l'ordre est seulement imposé sur les événements sur la même ligne de vie, l'opérateur strict spécifie que tous les événements (quelle que soit leur localisation) du premier DS doivent être exécutés avant les événements du deuxième DS.

– La composition parallèle (par). L'opérateur par spécifie que les événements des opérandes peuvent être exécutés en parallèle, c.à.d. dans n'importe quel ordre ou en concurrence.

– Option (opt). L'opérateur opt spécifie un choix entre le comportement d'un DS ou un comportement vide. Cet opérateur est équivalent à une alternative où le deuxième opérande est une interaction vide (sans événements).

La composition des diagrammes de séquence d'UML2 est décrite dans des diagrammes de séquence combinés. La Figure 2.6 montre un exemple de diagramme de séquence combiné qui référence et compose trois diagrammes de séquence : SD1, SD2 et SD3. Les opérateurs de composition dans les diagrammes de séquence combinés sont décrits par des cadres rectangulaires avec un coin gauche supérieur labellisé par l'opérateur (seq, alt, loop..etc). Les opérandes pour les opérateurs de séquence, d'alternative, et de parallélisme sont séparés par des lignes horizontales discontinues.

La composition séquentielle peut être aussi implicitement donnée par l'ordre relatif de référence aux diagrammes de séquence. Par exemple le DS CombinedSD dans la Figure 2.6,

est équivalent à l'expression : loop (SD1 seq (SD2 alt SD3)). UML2.0 propose aussi une autre notation pour la spécification de la composition des diagrammes de séquence dans ce qui est appelé les diagrammes de vues d'ensemble d'interactions (qui sont en quelque sorte des diagrammes d'activités dont les activités sont des DS). La Figure 2.7 montre un exemple de diagramme de vues d'ensemble d'interactions ; il est équivalent à l'expression : SD1 seq (SD2 alt SD3).

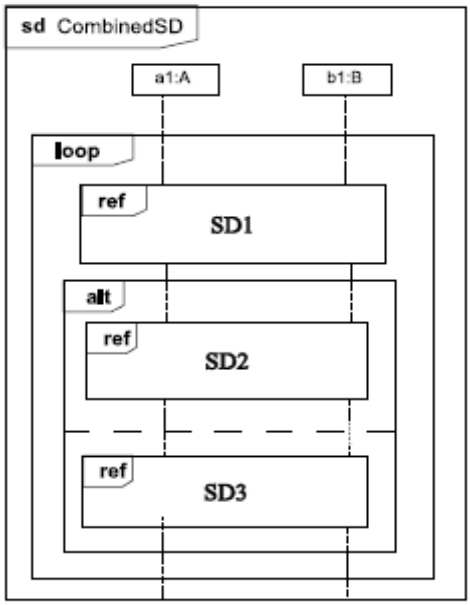

Fig. 2.6. Exemple d'un diagramme de séquence combiné

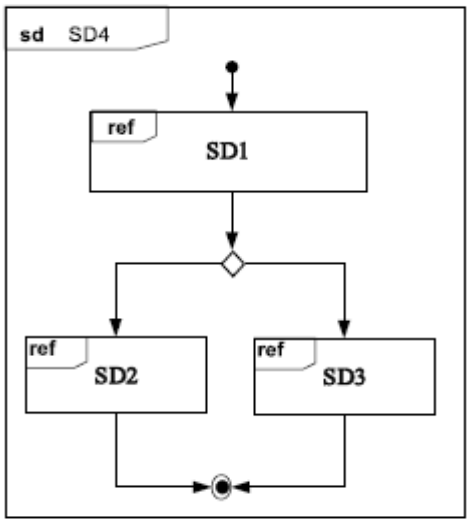

Fig. 2.7. Exemple d'un diagramme de vue d'ensemble d'interaction dans UML2

5. Conclusion

Notre choix des modèles d'UML pour la rétro-ingénierie est motivé par la prospérité remarquable acquise par ce langage dans les domaines de modélisation et développement des logiciels et son évolution continue. Ainsi de son intégrité, complicité, et la possibilité d'échange et communication disposée aux développeurs.

Comme nous l'avons indiqué ci-devant dans l'introduction, les modèles UML sont classés en deux catégories : les modèles statiques qui décrivent la structure et les modèles dynamiques qui décrivent le comportement. Une bonne compréhension d'un système ne peut être achevée que par l'inclusion des deux aspects statique et dynamique, néanmoins peu sont les travaux de la rétro-ingénierie qui s'occupent des modèles comportementaux.

Dans le chapitre suivant nous allons entamer la rétro-ingénierie des modèles comportementaux.

Chapitre 3 Rétro-ingénierie des modèles comportementaux

1. Introduction

Dans ce chapitre nous nous intéressons aux travaux de la rétro-ingénierie des modèles comportementaux d'UML. Comme nous l'avons présenté au chapitre 2, les deux principaux types de modèles comportementaux d'UML sont les machines à états et les diagrammes de séquence. Nous présentons principalement les travaux concernant les machines à état et les diagrammes de séquence. Ces deux notations d'UML sont les plus intéressantes dans les recherches existantes. Dans la section suivante nous présentons un état de l'art qui est divisé en deux parties. La première partie présente les travaux liés aux machines à état. Cependant, la deuxième partie est consacrée à une étude détaillée concernant la rétro-ingénierie des diagrammes de séquence. Notre étude liée aux diagrammes de séquence est plus détaillés que celle concernant les machines à états car notre sujet de recherche s'intéresse en particulier aux diagrammes de séquence d'UML. Nous concluions ce chapitre par la discussion des limites des travaux étudiés et en fixant le cadre générale de notre approche que nous proposons dans la seconde partie de cette thèse.

2. Etat de l'art sur la rétro-ingénierie des modèles comportementaux d'UML

2.1 Travaux de la rétro-ingénierie des machines à états

Plusieurs travaux de la rétro-ingénierie de machines à état ont été proposés [55, 56, 54, 57, 59].

Whaley et al dans [56] ont utilisé le code source (analyse statique) en tant que base pour la rétro-ingénierie des machines à état, mais ils ont augmenté leur analyse avec des traces dynamiques du système. Cette approche demande une certaine forme d'analyse de la structure du code source en fonction de ses données et contrôle de flux.

L'idée de construire des machines à état présentant le comportement d'un programme à partir des exemples d'exécution (i.e des traces) a été proposée initialement par Biermann et Feldman dans [57]. Ils ont proposé un algorithme appelé K-tails qui construit une machine à états à partir de chaque trace en premier temps. Par la suite, ces machines à états vont être fusionnées en se basant sur la similitude de leur comportement. Un autre algorithme appelé QSM (Query-driven State Merging) [58] similaire à k-tail et se base aussi sur la fusion des états pour construire une machine à états globale.

Walkinshaw et al dans le travail [59] ont proposé une technique qui utilise l'algorithme QSM pour la rétro-ingénierie d'une machine à état à partir d'un ensemble de traces.

Dans [26] Duarte a utilisé une combinaison des informations statiques et dynamiques pour l'extraction des modèles LTS (Labelled Transition Systems) à partir de code source Java, suivant l'approche hybride. Les informations recueillies statiquement sont liées au système de contrôle de flux. Pour la partie dynamique, Duarte a utilisé conjointement les informations de trace et les valeurs des variables du programme.

Yuan et al dans [61] proposent une autre méthode d'extraction de machines à état pour un système appelé Brastra. Cette approche permet l'extraction automatique des machines à états à partir des exécutions des tests unitaire.

2.2 Travaux de la rétro-ingénierie des diagrammes de séquence d'UML

Dans cette section nous présentons une étude détaillée des travaux existants autours de la rétro-ingénierie des diagrammes de séquence d'UML. Les travaux existants sont présentés selon la technique de collection de données (voir section 4.1.1 dans le chapitre 1) utilisée, à savoir : analyse statique, analyse dynamique et analyse hybride.

2.2.1 Travaux basés sur l'analyse statique

Parmi les travaux existants nous citons le travail de Rountev et al dans [1] qui propose un algorithme pour la rétro-ingénierie des diagrammes de séquence d'UML 2.0 par l'analyse statique du flux de contrôle de code source java. Ils ont proposé un mapping entre le graphe des flux de contrôle général (CFG) et les diagrammes de séquence d'UML. L'objectif ayant été de représenter le comportement itératif et conditionnel inter-procédural du système sous forme d'un diagramme de séquence. La figure Fig. 3.1 illustre un exemple du mapping entre le CFG et le diagramme de séquence d'UML 2.

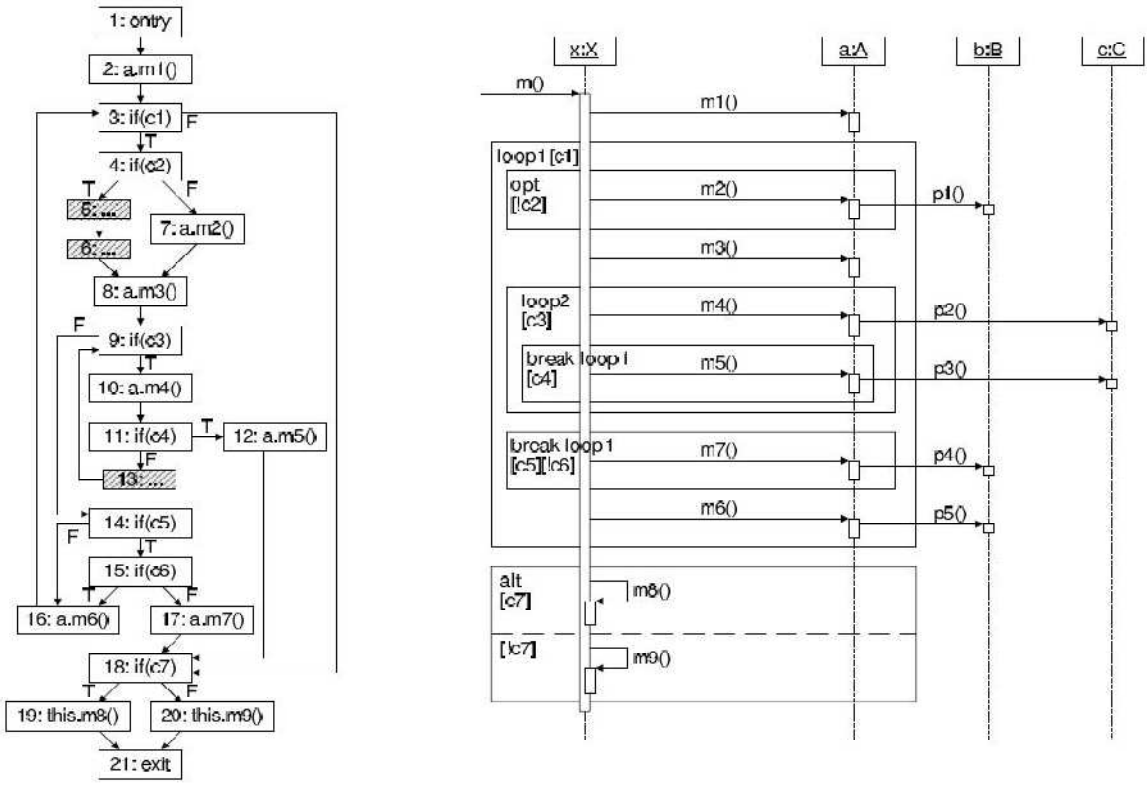

Fig. 3.1. CFG et son diagramme de séquence correspond [1]

L'analyse statique permet de construire un diagramme de séquence avec les fragments combinées : alt, loop etc., cependant les diagrammes de séquence obtenus ne montrent que les interactions entre les classes et ils ne permettent pas au développeur de déterminer combien d'objets d'une classe peuvent exister à l'exécution, ni de dire combien d'appels de méthodes peuvent effectuer entre ces objets.

2.2.2 Travaux basés sur l'analyse dynamique

La majorité des travaux existants utilise les techniques d'analyse dynamique pour la collection de données. Dans la suite de cette section nous présentons une étude détaillée de ces travaux.

Dans [5], Janice a proposé une méthode pour la rétro-ingénierie des diagrammes de séquence en exécutant le système pour un seul cas d'utilisation. L'approche utilise l'instrumentation du code intermédiaire pour construire les traces.

L'instrumentation se concentre sur les messages de création d'objet (les constructeurs), les messages d'appel des méthodes et les contrôles des flux de données de types itératif et alternatif.

Oechsle et Schmitt dans [7] ont développé l'outil JAVAVIS qui permet aux utilisateurs de comprendre ce qui se passe lors de l'exécution d'un programme Java. L'outil est basé sur l'interface de débogage Java (JDI) pour visualiser le comportement dynamique

sous forme d'un diagramme d'objet et de séquence (voir Fig. 3.2.) JAVAVIS se compose structurellement de deux parties essentielles (voir fig. 3.3): l'interface de débogage Java (JDI) qui se charge du contrôle, et le noyau Vivaldi qui permet de générer les diagrammes.

Dans [10] De Pauw et al. ont conçu un outil appelé Jinsight qui a pour but de visualiser plusieurs facettes du comportement du système en se basant sur des traces d'exécution. La technique d'instrumentation de la machine virtuelle a été utilisée pour collecter ces traces. L'outil Jinsight implémente une technique d'extraction de patrons qui a été proposée pour simplifier les vues par élimination des répétitions des messages.

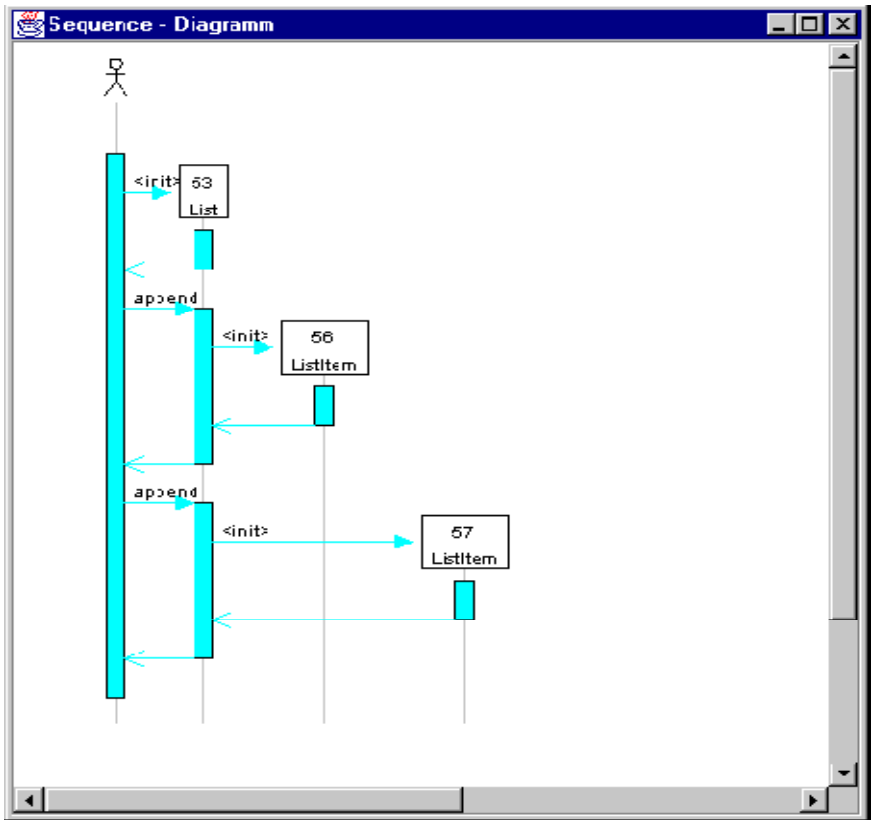

Fig. 3.2. DS après l'exécution de deux itérations d'une boucle [7]

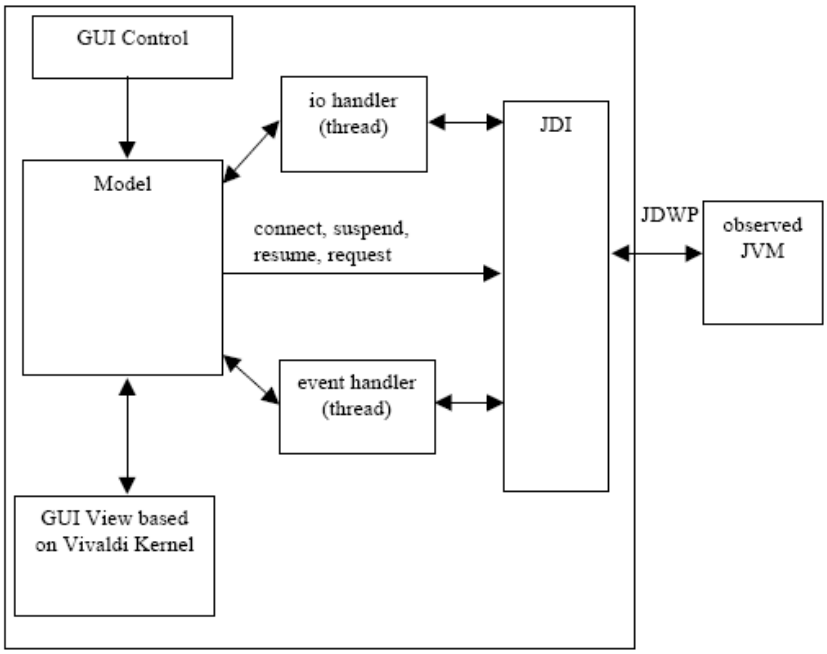

Fig. 3.3. Architecture de JAVAVIS [7]

Briand et al dans [11] ont proposé une méthode pour la rétro-ingénierie des diagrammes de séquence à partir des traces d'exécution. Ils ont défini deux méta-modèles : le premier méta-modèle décrit les diagrammes de séquence, et le dernier décrit les traces d'exécution. Un mapping entre les deux méta-modèles est proposé pour montrer la façon de dériver un diagramme de scénario à partir d'une trace d'exécution. Ce mapping, illustré dans la figure fig. 3.5., est défini par un ensemble de contraintes consistantes d'OCL (Object Constraint Language). Les traces sont générées par l'instrumentation du code source. Le diagramme de séquence obtenu condense la trace d'un côté (la séquence des messages répétée n'apparaîtra qu'une seule fois avec une condition de répétition), et ajoute plus d'informations d'un autre (la condition sous laquelle un appel est effectué est reportée dans le diagramme de scénario). Parmi les limites de ce travail est que son contexte est restreint sur une seule exécution du système qui correspond à un seul cas d'utilisation. Le résultat obtenu est donc un diagramme de séquence incomplet appelé diagramme de scénario qui dénote seulement ce qui se passe dans un seul scénario particulier. Aussi, les imbrications ne sont plus supportées et toutes les classes sont instrumentées car il n'y a pas de technique de sélection partielle.

Dans [12] Briand et al. ont complété leur travail [11] avec prise en charge des systèmes distribués, mais avec un peu de différence où ils se sont basés sur l'instrumentation du code intermédiaire (byte code). Ainsi qu'ils ont instrumenté le code source pour détecter les flux de control itératifs et conditionnels. La même stratégie de mapping entre les deux

méta-modèles en définissant des règles décrites dans OCL est poursuivi comme il est indiqué dans fig. 3.4, fig. 3.5, et fig. 3.6.

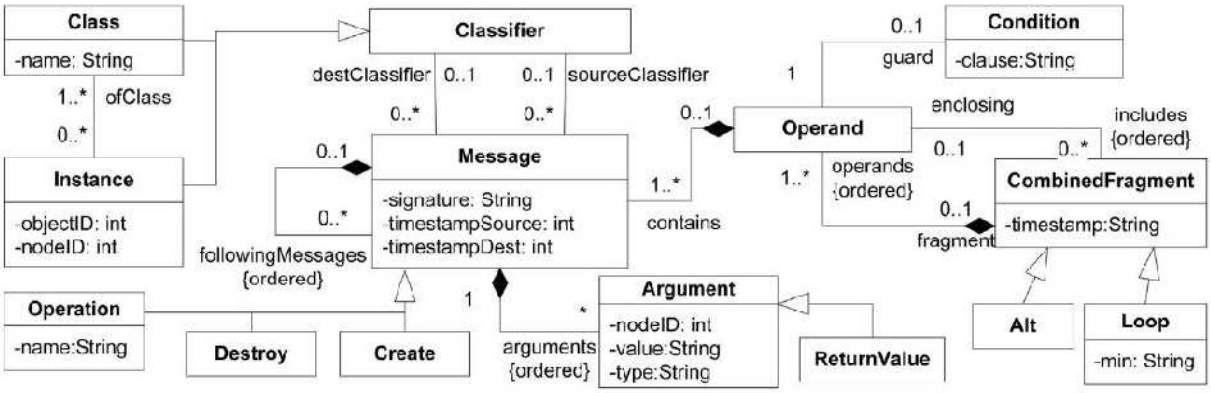

Fig. 3.4. Méta-modèle de diagramme de scénario Briand et al. [12]

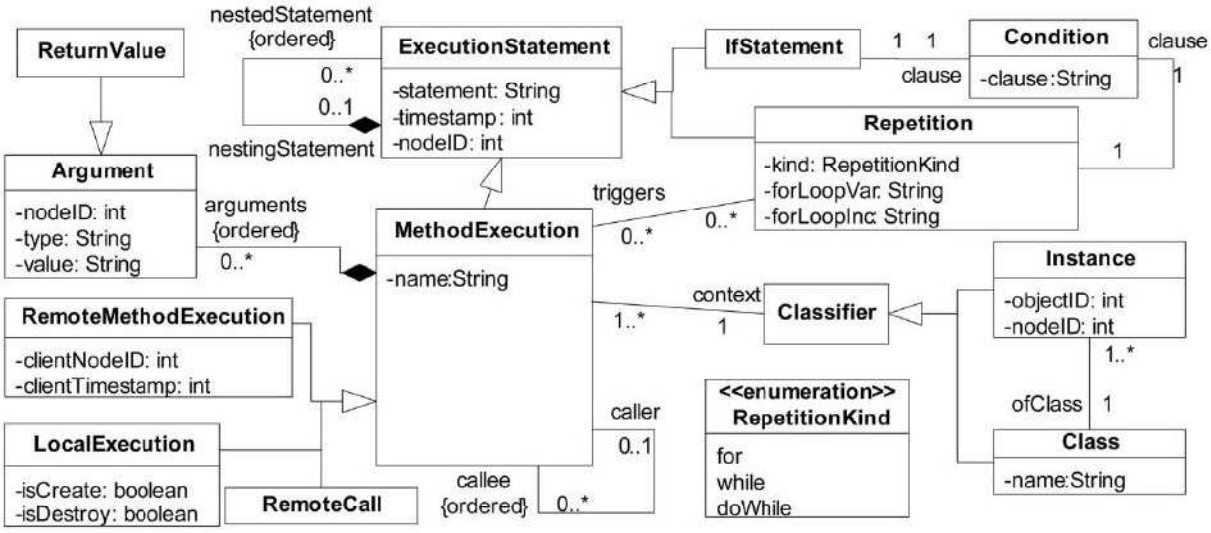

Fig. 3.5. Méta-modèle de trace Briand et al. [12]

```
1 T::MethodExecution.allInstances->forAll(  me1: T::MethodExecution,
2                                           me2: T::MethodExecution |
3   CheckMapping.mapping(me1,me2)->notEmpty() implies
4   S::Message.allInstances()->exists(mes: S::Message |
5     //comparing timestamps
6     mes.timestampSource = me1.timestamp
7     and mes.timestampDest = me2.timestamp
8     //the context of me1 is the source of message mes
9     and CheckMapping.mapContextClassifier(me1.context, mes.sourceClassifier)
10    //the context of me2 is the target of message mes
11    and CheckMapping.mapContextClassifier(me2.context, mes.destClassifier)
12    //compare arguments (matching entire sequences)
13    and CheckMapping.mapExecutionMessageArgs(me2, mes)
14    //mapping the message guard
15    and
16      if (mapping(me1,me2).nestingStatement.oclIsTypeOf(MethodExecution)) then
17        mes.operand->isEmpty()
18      else // the nesting statement type is either IfStatement or Repetition
19        mes.operand.guard.clause =
20             mapping(me1,me2).nestingStatement.clause.clause
21      endif
22    //mapping the exact message type, along with message name and signature
23    and mes.signature = me2.statement
24    and mes.name = me2.name
25    and me2.isCreate = true implies mes.oclIsTypeOf(S::Create)
26    and me2.isDestroy = true implies mes.oclIsTypeOf(S::Destroy)
27    and not (me2.isCreate = true or me2.isDestroy = true) implies
28             mes.oclIsTypeOf(S::Operation)
29  ) // S::Message.allInstances->exists
30)
```

Fig. 3.6. Appariement entre trace et séquence méta-modèles Briand et al. [12]

Guéhéneuc et Ziadi dans [13] ont proposé une approche d'analyse dynamique pour la rétro-ingénierie des modèles comportementaux d'UML 2.0 (diagrammes de séquence et machines à états). L'approche est définie en trois étapes :

Etape 1. Construire des diagrammes de séquence de base à partir de fichiers de traces générés par l'outil d'analyse dynamique Caffeine [38].

Etape 2. La composition de diagrammes de séquence de base.

Etape 3. La synthèse des diagrammes d'états.

Une fois les diagrammes de séquence auraient été générés, ils produisent des machines à états automatiquement en utilisant une méthode existante [60] pour la synthèse de machines à états. Une analyse de haut niveau concernant le contrôle de conformité et l'identification des patrons a été aussi proposée.

Dans [16], la technique de débogage a été utilisée dans le développement d'un outil d'analyse dynamique appelé JavaTracer. Cet outil sauvegarde des traces lors de l'exécution d'un programme Java. Il met à la faveur de l'utilisateur de choisir les classes et les méthodes à tracer. L'outil JavaTracer pourvoit des détails sur les événements d'exécution sous forme de

messages en indiquant l'objet appelant, l'objet appelé, et la méthode exécutée pour en produire des diagrammes de séquence.

De plus des travaux présentés, il existe certains travaux qui se sont intéressés à extraire des informations plus utiles et pertinentes à partir des traces d'exécution :

Richner et Ducasse dans [8] ont présenté une approche pour recouvrir les collaborations dans un logiciel implémenté dans un langage orienté objet. L'approche prend en entrée une trace d'exécution puis elle condense ces informations en représentant le comportement du programme par des patrons de collaboration. Enfin, elle présente ces informations aux développeurs en termes de classe expéditrice, classe réceptrice, méthode invoquée et patrons de collaboration.

Dans [20] Hamou-Lhadj a proposé une technique d'analyse de traces dynamiques en résumant leur contenu. La technique prend une trace d'exécution à l'entrée et génère son résumé en sortie. L'auteur a inspiré des techniques existantes de résumé de textes. Ces techniques permettent de représenter un texte par ces idées principales en évitant les détails.

Kuhn et al dans [21] ont proposé une approche d'analyse des traces d'exécution des caractéristiques (fonctions) de logiciels en utilisant la méthode de recherche d'information LSI (Latent semantic indexing). Cette analyse a pour but d'identifier les similarités entre les caractéristiques en se basant sur les traces, ainsi de classifier les classes selon leurs fréquences dans les traces.

2.2.3 Travaux basés sur l'analyse hybride

Une tentative d'utilisation de cette approche est présentée dans [39], où les informations statiques sont analysées afin de confirmer l'existence d'invariants déduits en se basant sur les informations dynamiques. Les auteurs ont réussi à démontrer l'utilité de compléter les informations dynamiques par des informations statiques. Toutefois, leur approche applique les deux types d'analyses, par conséquence, elle nécessite l'utilisation de deux outils différents: l'un pour déduire les invariants des informations dynamiques et l'autre pour vérifier ces invariants auprès des informations statiques.

Dans [6], Systa a considéré les deux aspects statique et dynamique, le premier pour modéliser la structure statique du système ciblé, tant que le dernier pour modéliser le comportement d'exécution. Un environnement expérimental appelé Shimba a été construit pour supporter la rétro-ingénierie des systèmes logiciels Java en visant les deux aspects structurel et comportemental. Les informations statiques sont extraites à partir des fichiers de classes de code intermédiaire Java (Java bytecode), par contre les informations dynamiques

d'évènements sont générées après l'exécution du système sous un débogueur JDK adapté baptisé JDebugger. Le prototype Shimba intègre deux outils existants Rigi [43] et SCED. Après leur extraction, les informations statiques sont visualisées dans l'environnement Rigi, cependant, l'outil de modélisation dynamique SCED est utilisé pour la visualisation des informations dynamique sous forme de diagramme de scénarios. Grace aux techniques de JDebugger, Shimba a supporté le contrôle des flux, ce qui permet à l'utilisateur d'apercevoir les différentes structures conditionnelles Java (if, while, ..). L'environnement Shimba implique à l'utilisateur de détecter les éléments qui implémentent la fonction souhaitée. Cela n'est pas très pratique dans de nombreuses situations.

Antoniol et al dans [14] ont développé un outil d'analyse des applications web appelé Wanda. Une approche de quatre étapes a été proposée :

1) Analyse et instrumentation.
2) L'extraction des informations
3) L'abstraction des modèles
4) Présentation des résultats.

Les auteurs de [14] ont combiné les deux types de méthodes d'analyse : statique et dynamique en se basant fortement sur la dernière qui s'exprime par l'analyse de:

- Pages web HTML et PHP (instrumentation).
- Les variables d'environnement http.
- Les Cookies, les entêtes, et les fonctions de gestion de sessions
- Les bases de données et fichiers d'E /S.
- Les invocations des services web.
- Les accès aux bibliothèques.

A la fin de cette analyse, plusieurs types de diagrammes d'UML vont être générés dont : diagramme de composant, diagramme de déploiement, diagramme de séquence, et diagramme de classe.

2.3 Discussion et conclusion

Le tableau 3.1 récapitule les travaux étudiés dans ce chapitre. Pour chaque travail, nous décrivons le type de la méthode d'analyse de système utilisée, la technique de collection de données considérée et le niveau d'abstraction visé.

Travail	Méthode d'analyse suivie			Multi trace	Niveau d'abstraction visé		Technique de collection de données		Opérateurs d'interaction supportés
	statique	dynamique	hybride		objet	classe	Débogage	instrument	
Rountev et al. [1]	X			/	X				Opt, loop, break, alt
Janice [5]		X		Non	X			X byte code	Loop, alt
Oechsle et Schmitt [7]		X		Non	X		X		/
DePauw et al. [10]		X		Non				X machine virtuelle	/
Briand et al. [11]		X		Non	X			X code source	loop
Briand et al. [12]		X		Non	X			X byte code	Loop, alt
Guéhéneuc et al. [13]		X		Oui				X byte code	/
Wendehals [16]		X		Non	X		X		/
Richner et Ducasse [8]		X		Non		X			/
Nimmer et al. [39]			X	Non					/
Systa [6]			X	Non	X		X		/
Antoniol et al. [14]			X	Non				X code source	/

Tableau 3.1 Travaux de la rétro-ingénierie des diagrammes de séquence UML

En étudiant de près les travaux existants, plusieurs limites peuvent être identifiées que nous résumons autour des points suivants :

La majorité des travaux existants ne considère qu'une seule trace d'exécution du système en entrée dans la rétro-ingénierie des diagrammes de séquence. Cela est justifié en particulier par la difficulté de faire combiner plusieurs traces d'exécution du même système [5]. En effet, il est difficile de savoir quand deux occurrences d'une action dans deux traces différentes correspondent à la même action spécifique dans le code.

Tenant de l'étude des travaux existants, nous proposons de revisiter le problème de la rétro-ingénierie des diagrammes de séquence d'UML par une nouvelle approche. Notre approche utilise la technique d'analyse dynamique de code, en particulier nous proposons d'analyser les traces d'exécution d'un système pour en construire des diagrammes de séquence d'UML2 avec une prise en compte des opérateurs d'interactions.

PARTIE 2 : Contribution

Chapitre 4 : Approche Incrémentale

1. Introduction

Comme nous l'avons souligné dans le chapitre précédent, la majorité des travaux existants dans le contexte de la rétro-ingénierie des diagrammes de séquence d'UML permet de générer principalement des diagrammes de séquence en se basant sur un seul scénario d'exécution du système. Donc, Les diagrammes de séquence obtenus ne correspondent qu'à une seule trace, par conséquent, ils ne donnent qu'une vision partielle du comportement du système.

Dans ce chapitre, nous proposons une nouvelle approche revisitant le problème de rétro-ingénierie des diagrammes de séquence d'UML2 à partir d'un système logiciel. Contrairement aux travaux existants, notre approche considère plusieurs traces d'exécution. Dans ce qui suit, nous présentons en détail notre approche et nous l'illustrons sur un simple exemple qui concerne un système de vente classique.

2. Exemple pour illustration

Pour illustrer les différentes étapes de notre approche nous allons utiliser l'application *Vente*. Il s'agit d'une simple application développé en Java, dont le code est montré dans l'annexe, permettant à des vendeurs de créer des ventes d'articles. Pour réaliser cette création, le vendeur envoie un ordre de création d'une nouvelle vente et il peut par la suite ajouter des articles et calculer la somme. L'ajout des articles et le calcul de la somme peut être répété autant de fois que le nombre d'articles commandés. Après, soit un bon de livraison ou une facture doit être établie, pour qu'elle soit signée par le vendeur. Enfin, la création d'un bon de paiement est l'objet de choix pour le client. La figure Fig. 4.7. illustre le diagramme de séquence de cette application. Nous distinguons les classes suivantes :

La classe *Vendor* représente le vendeur qui initialise l'activité de vente avec les paramètres suivants :

nbr_article : désigne le nombre d'articles de la vente courante.

isInvoice : indique si une facture doit être établie ou non.

isPayslip : indique si un bon de paiement doit être établi ou non.

Les données avec lesquelles sont initialisés ces paramètres sont eux même les données d'entrée qui précisent le comportement de l'application dans chaque session d'exécution.

La classe *Vendor* contient également les méthodes *signInvoice*() , *signDelivery*() , *signPayslip*() qui représentent le visa des paperasses établies par le vendeur.

La classe *sale* qui représente la vente contient la méthode *newSale*(), celle-ci crée une nouvelle instance de vente, ainsi contient la méthode *addArticle*() qui ajoute les articles commandés.

La classe de calcule *Calcul* contient la méthode *calculAmount*(`float newValue, float oldValue`) qui fait calculer la nouvelle somme de la vente après l'ajout d'un nouveau article.

La classe *Delivery* qui s'occupe aux bons de livraison contient la méthode *getDelivery*(), cette dernière établit un bon de livraison pour la vente s'il est commandé par le client.

La classe *Invoice* qui s'occupe aux factures contient la méthode *getInvoice*(), cette dernière établit une facture pour la vente si la facture n'est pas été commandé par le client.

La classe *Payslip* qui s'occupe aux bons de paiement contient la méthode *getPaysliph*(), cette dernière établit un bon de payement pour la vente s'il est commandé par le client.

3. Approche incrémentale

Notre approche est illustrée dans la figure Fig. 4.1. Elle suit le schéma général des approches du reverse-engineering (cf. chapitre 1) et elle se compose principalement de deux étapes, dont la première concerne la collection de traces. Ces traces sont enregistrées en exécutant le système instrumenté[1] plusieurs fois pour couvrir les scénarios possibles. La deuxième étape est la construction incrémentale d'un diagramme de séquence complet[2], elle consiste à appliquer progressivement des heuristiques sur l'ensemble d'information collectées dans l'étape précédente. Dans ce qui suit nous allons voir en détail ces deux étapes en illustrant chaque étape sur l'exemple de vente.

[1] On utilise un outil d'instrumentation pour obtenir une version instrumenté du système ciblé.

[2] Un diagramme de séquence complet représente le comportement général d'un système pas seulement le comportement du système dans un scénario particulier.

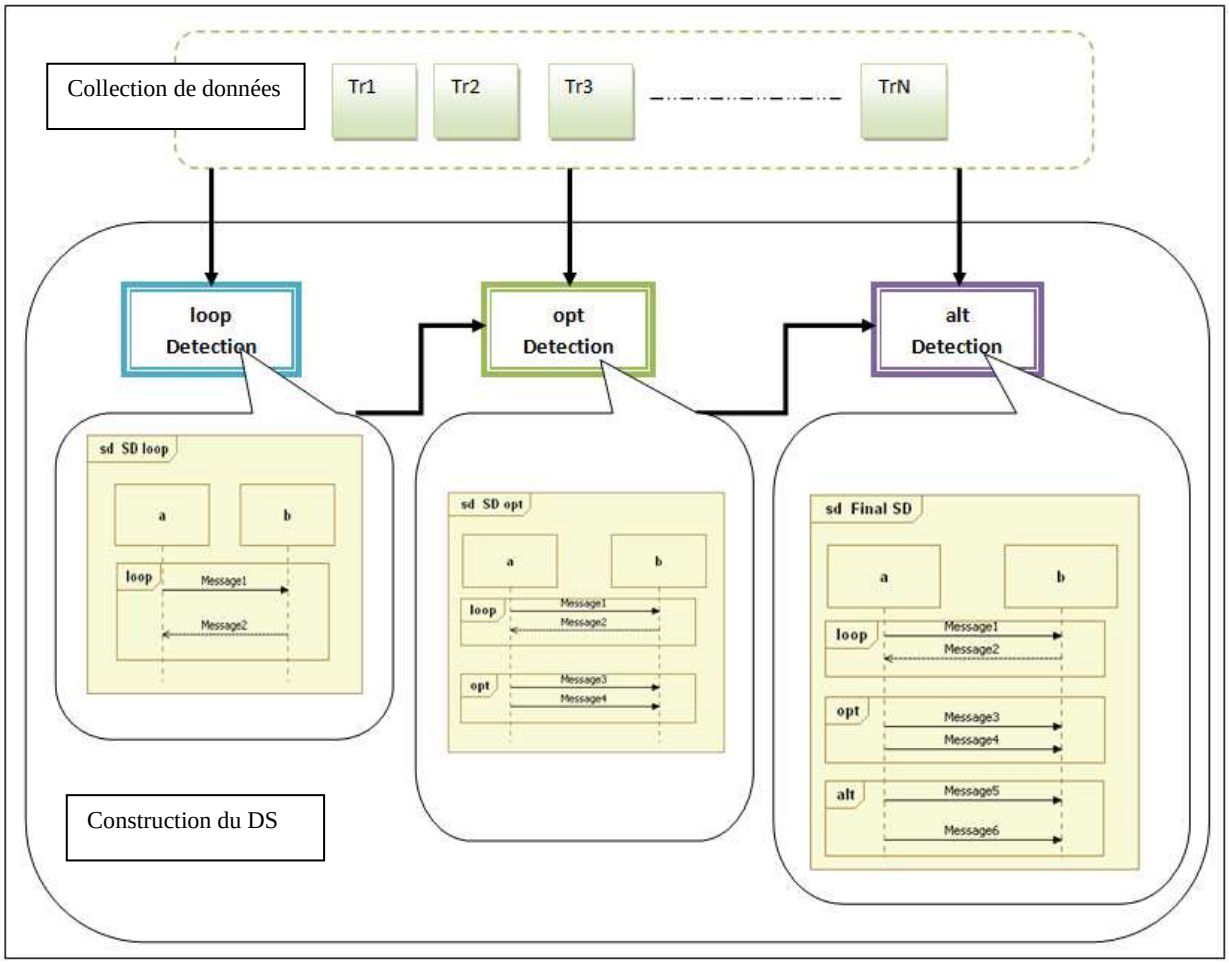

Fig. 4.1. La construction Incrémentale du diagramme de séquence

3.1 La Collection de traces

L'objectif de la première étape est de collecter les traces d'exécution du système logiciel dont nous voudrions construire des diagrammes de séquence en appliquant la rétro-ingénierie. Comme il est présenté dans le chapitre 1, la technique adéquate pour la collection de traces est l'analyse dynamique. Dans notre approche nous avons choisi d'utiliser un outil existant d'analyse dynamique permettant la collection de traces pour les systèmes logiciels Java. Cet outil est appelé MoDec[5].

La collection des traces d'exécution du système nécessite de l'exécuter plusieurs fois en changeant les données d'entrée. En effet, le comportement du système est fortement lié à son environnement et en particulier les données introduites par l'utilisateur pour initialiser les variables spécifiques du système. Ainsi, une session d'exécution ne suffit pas pour identifier le comportement général du système. En effet, les interactions alternatives et itératives ne peuvent être détectées qu'en précisant les valeurs appropriées des conditions "if, else if" et "while, for, repeat ...". Il faut donc répéter cette phase autant de fois que le nombre de

différentes entrées du système pour collecter les traces pertinentes. Cependant exécuter le système pour couvrir toutes les entrées possibles du système est irréaliste car la plage des valeurs de données est souvent infinie. Dans notre approche nous avons posé l'hypothèse de l'existence d'un utilisateur expert qui connaît les différentes valeurs d'entrées du système permettant d'obtenir les traces les plus pertinentes et de couvrir toutes les fonctionnalités possibles du système.

Dans le cas de l'exemple d'illustration *Vente*, nous avons constaté que trois sessions d'exécution sont suffisantes pour observer les comportements possibles. Cela en changeant les données d'entrée dans chaque exécution. La figure Fig. 4.2 montre les traces obtenues dans chaque session et leurs correspondantes données d'entrée.

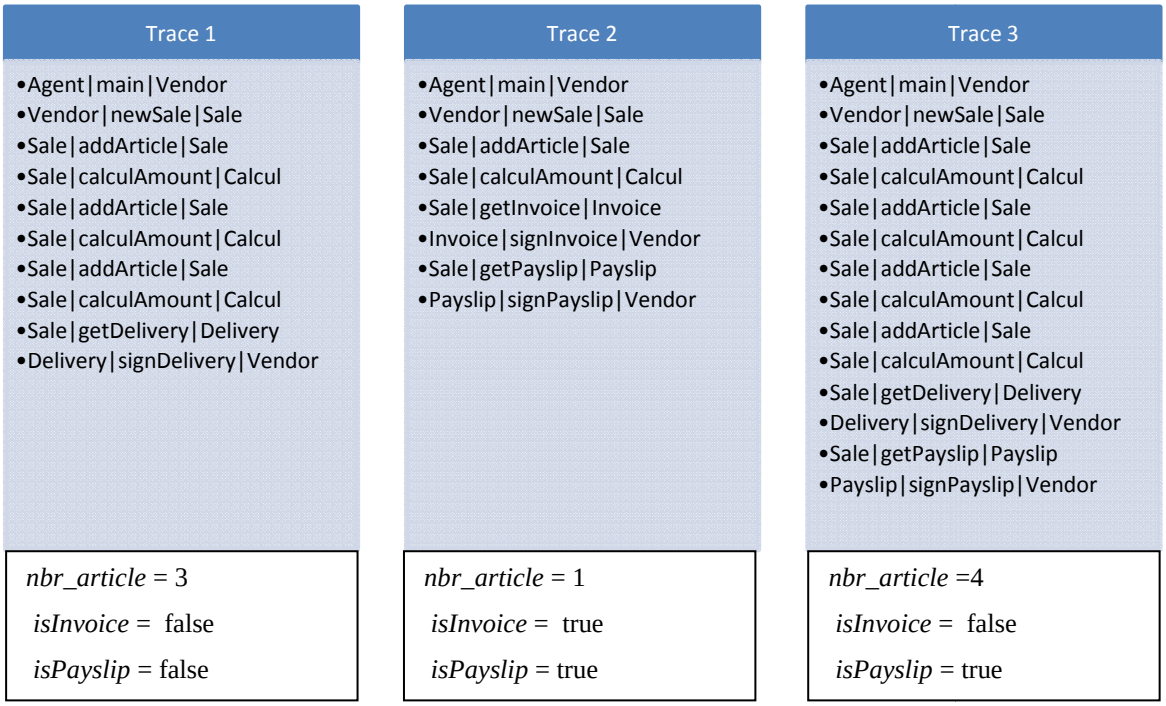

Fig. 4.2. Traces de l'application Vente avec les données d'entrée de chaque trace

La Trace 1 par exemple est obtenue en exécutant le système du vente avec les variables d'entrée *: nbr_article = 1, inInvoice = false et isPayslip = false*. Chaque trace est une séquence de statements ou chaque statement montre l'échange de message entre deux objets du système. Dans ce qui reste, nous présentons une formalisation des traces que nous manipulerons par la suite pour générer les diagrammes de séquence.

Trace

Nous appelons *trace* la suite d'événements d'appel de méthodes enregistrés après une session d'exécution du système étudié. Ainsi, nous appelons *Statement* tout un événement enregistré dans une *trace*.

Définition 1.

Une STATEMENT *est un tuple* ST=‹sender :*SENDER, METHOD, receiver :RCEIVER*› *où :*

(i) *SENDER est la classe de l'objet sender qui invoque la méthode ;*
(ii) *RECEIVER est la classe de l'objet receiver qui exécute la méthode ;*
(iii) *METHOD est la méthode.*

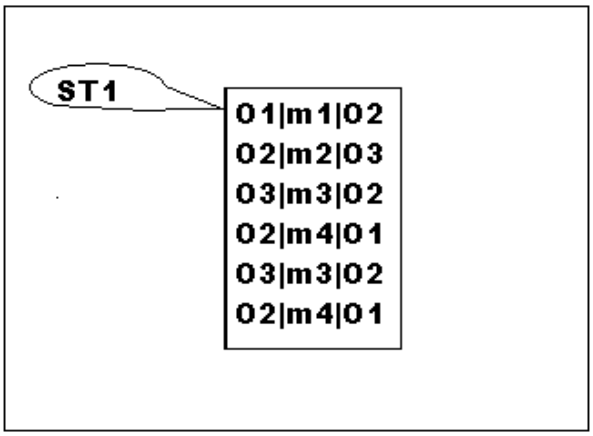

Fig. 4.3. Exemple de Trace

Définition 2. *Une* TRACE *est un tuple TR=‹ST, L, <› où :*

(i) *ST est l'ensemble de STATEMENT;*
(ii) *L ∈ N est la taille de la trace où N est l'ensemble des entiers positifs;*
(iii) *< : est une relation d'ordre total dans ST.*

Exemple 1

La figure Fig. 4.3. montre un exemple de trace dont :

St1=‹O1, m1, O2› ∈ STATEMENT *où sender =O1, receiver =O2, et METHOD =m1.*

Exemple 2

Soit Tr ∈ TR un exemple de trace illustré dans la figure Fig. 4.3. nous avons :

Tr= {St1, St2, St3, St4, St5, St6} où St1=‹O1, m1, O2›, St2=‹O2, m2, O3›, St3=‹O3, m3, O2›, St4=‹O2, m4, O1›, St5=‹O3, m3, O2›, St6=‹O2, m4, O1›.

Des fonctions pour la manipulation des traces

Dans ce qui suit nous définissons l'ensemble des fonctions que nous allons utiliser plus tard dans nos algorithmes :

Soit Tr ∈TR une trace et St1, St2 ∈ ST sont deux STATEMENTs de Tr, et l sa taille :

1) *St1 < ST2 ssi St1 est situé avant St2 ;*
2) RANG *est la fonction qui renvoie l'ordre d'une STATEMENT dans une trace :*

$$RANG: ST \times TR \rightarrow N$$

$$RANG(st, Tr) = x$$

N est l'ensemble d'entiers positifs,

3) SubTrace *est la fonction qui renvoie une sous-trace d'une trace :*

$$SubTrace: TR \times N \times N \rightarrow TR$$

$$SubTrace(Tr, x, y) = subTr$$

où x est la position du début de $subTr$ et y est la position de sa fin.

4) FirstStatement *est la fonction qui rétrouve la première STATEMENT d'une trace :*

$$FirstStatement: TR \rightarrow ST$$

$$FirstStatement(Tr) = St$$

L'équivalence suivante est correcte : $RANG(St, Tr) = 1 \iff FirstStatement(Tr) = St$

5) LastStatement *est la fonction qui rétrouve la dernière STATEMENT d'une trace :*

$$LastStatement: TR \rightarrow ST$$

$$LastStatement(Tr) = St$$

L'équivalence suivante est correcte : $RANG(St, Tr) = l \iff LastStatement(Tr) = St$

6) Successor *est la fonction qui renvoie la Statement qui succède une Statement donnée :*

$$Successor: ST \times TR \rightarrow ST$$

$$Successor(Stx, Tr) = Sty$$

Les équivalences suivantes sont correctes :

$$RANG(St, Tr) = l \Leftrightarrow Successor(St, Tr) = Null$$

$$Successor(Stx, Tr) = Sty \iff RANG(Sty, Tr) = RANG(Stx, Tr) + 1$$

7) Predecessor *est la fonction qui renvoie la Statement qui précède une Statement donnée :*

$$\text{Predecessor} : ST \times TR \rightarrow ST$$

$$\text{Predecessor}\ (Stx, Tr) = Sty$$

Les équivalences suivantes sont correctes :

$$RANG(St, Tr) = 1 \Leftrightarrow \text{Predecessor } (St, Tr) = Null$$

$$\text{Predecessor } (Stx, Tr) = Sty \Leftrightarrow RANG(Sty, Tr) = RANG(Stx, Tr) - 1$$

8) SuccessorSet *est la fonction qui renvoie l'ensemble des Statements qui succède une Statement donnée :*

$$SuccessorSet: ST \times TR \rightarrow TR$$

$$SuccessorSet(St, Tr) = subTr$$

Les équivalences suivantes sont correctes :

$$RANG(St, Tr) = l \Leftrightarrow SuccessorSet(St, Tr) = \emptyset$$

$$SuccessorSet(Stx, Tr) = Sty + SuccessorSet(Sty, Tr) \dots (1)$$

$$(1) \Leftrightarrow RANG(Sty, Tr) = RANG(Stx, Tr) + 1$$

9) PredecessorSet *est la fonction qui renvoie l'ensemble des Statements qui précède une* Statement *donnée :*

$$PredecessorSet: ST \times TR \rightarrow TR$$

$$PredecessorSet(St, Tr) = subTr$$

Les équivalences suivantes sont correctes :

$$RANG(St, Tr) = 1 \Leftrightarrow PredecessorSet(St, Tr) = \emptyset$$

$$PredecessorSet(Stx, Tr) = Sty + PredecessorSet(Sty, Tr) \dots (1)$$

$$(1) \Leftrightarrow RANG(Sty, Tr) = RANG(Stx, Tr) - 1$$

10) isEquivalent *est la fonction qui vérifie si deux sous-traces sont équivalentes ou non* :

$$isEquivalent: TR \times TR \rightarrow Boolean$$

$$isEquivalent(subTr1, subTr2) = true | false$$

Soient : subTr1 et subTr2 deux sous-traces :

$$isEquivalent\ (subTr1, subTr2) = true\ ssi:$$

$$\forall\ St1 \in subTr1, \quad St2 \in subTr2\ si\ RANG(St1, subTr1) = RANG(St2, subTr2) \Longrightarrow St1 = St2$$

3.2 Construction incrémentale du diagramme de séquence

La deuxième étape de notre approche consiste à générer un diagramme de séquence UML représentant le comportement du système spécifié dans les traces obtenues dans la première étape. Les diagrammes de séquence que nous souhaitons obtenir sont des instances du méta-modèle d'UML 2.1.2 (cf. chapitre 2). Ces diagrammes peuvent être construits en utilisant la correspondance entre les concepts du méta-modèle et les éléments des traces. Chaque objet qui apparaît dans la trace sera transformé en un *Lifeline* dans le diagramme de

séquence et les méthodes exécutées seront transformées en *Messages*. Toutefois, la question cruciale dans cette phase, concerne l'identification des fragments combinés. Autrement dit, comment peut-on détecter les opérateurs d'interaction : loop, opt, et alt, à partir des traces?

La solution que nous proposons pour détecter les opérateurs d'interactions consiste à construire un diagramme de séquence d'une manière incrémentale étape par étape, tel qu'il est décrit dans la figure Fig. 4.1. Cette construction est basée sur l'identification des séquences d'événements répétées qui sont les fragments combinés avec l'opérateur loop, et en se basant sur les divergences entre les traces pour identifier les fragments combinés avec les opérateurs 'opt' et 'alt'. Le diagramme de séquence doit évoluer progressivement jusqu'à la fin du traitement. En effet, le diagramme de séquence obtenu à partir de la première itération est enrichi par la deuxième, et ainsi de suite.

Avant de présenter les algorithmes permettant de construire le diagramme de séquence, à partir des traces nous présentons ci-dessous une formalisation des diagrammes de séquence d'UML et les fonctions permettant leurs manipulations.

Diagramme de séquence

Définition 3. *Un* BasicSequenceDiagram *est un tuple* BasicSD=‹*M, L, <=*› *où :*

(i) M est un ensemble de message. Un message est défini par : m= (sender, name, receiver) où : sender, receiver ∈ L ;

(ii) L est l'ensemble de lifelines participants dans le diagramme de séquence ;

(iii) <= est une relation d'ordre partiel dans M.

Définition 4. *Un* CombinedFragment *est un tuple* CF=‹*Operands, Operator*› *où :*

(i) Operands est un ensemble de BasicSequenceDiagram ;

(ii) Operator est une énumération : seq, opt, alt, loop.

Définition 5. *Un* SequenceDiagram *est un tuple* SD=‹*CFs, <*› *où :*

(i) CFs est un ensemble de CombinedFragment ;

(ii) < est une relation d'ordre total dans CFs.

Des fonctions pour la manipulation des diagrammes de séquence

Dans ce qui suit nous définissons l'ensemble des fonctions que nous allons utiliser dans nos algorithmes :

1) SetOperand (messages) est la fonction qui assigne un ensemble de messages à un *BasicSequenceDiagram* .
2) GetOperand (message) est la fonction qui retourne le nom du *BasicSequenceDiagram* assigné à un message.
3) newOperand(name) est la fonction qui crée un basicSequenceDiagram dans un CombinedFragment.
4) newCombinedFragment(name, operator) est la fonction qui crée un combinedFragment dans un sequenceDiagram.

Nous utilisons dans nos algorithmes de détection des opérateurs d'interaction la notion de « bloc ». Ci-dessous, nous présentons cette notion de bloc ainsi qu'un ensemble de fonctions.

Bloc

Notre approche actuelle s'intéresse à des applications sans imbrications (i.e. pas de structure itérative ou conditionnelle imbriqué dans une autre). Les suites de **statements** d'une trace peuvent être regroupées en blocs en fonction de leurs comportements communs. Le comportement général d'un système peut être vu donc comme une séquence de blocs (voir figure Fig. 4.4.), semblable à la séquence des fragments combinés dans le méta-modèle du diagramme de séquence d'UML2. Les blocs sont considérés comme des conteneurs de **statements**, et ils respectent le même ordre dans toutes les traces. Le contenu d'un bloc peut être différent d'une trace à une autre en fonction de la nature de son comportement. Un bloc peut être vide dans une trace en fonction des conditions structurelles (comportement optionnel ou itératif), et il doit contenir des **statements** dans toutes les traces (comportement permanent ou alternatif) (voir figure Fig. 4.4.).

En général, un SequenceDiagram se compose d'une séquence de **blocs**, où chaque **bloc** peut contenir une séquence de **messages** (*BasicSequenceDiagram*). Comme les Traces représentent le même comportement du SequenceDiagram, ils respectent la même séquence de **blocs**, où chaque **bloc** peut contenir une suite de Statements. Une Statement d'une Trace ressemble à un message dans un SequenceDiagram, donc, on parle d'un bloc de Statements

ou d'un bloc de messages pour désigne la même chose. Nous introduisons ci-dessous un ensemble de fonctions manipulant les blocs :

1) GetBloc (Statement) est *la fonction qui retourne l'ordre du* bloc *d'une* Statement.
2) GetBlocSubTrace(bloc, Trace) *est la fonction qui retourne les* Statements *d'un* bloc *dans une* Trace.
3) isEmptyBloc(bloc, Trace) *est la fonction qui vérifie qu'un* bloc *est vide dans une* Trace.
4) isExistStatement(Statement, bloc, Trace) *est la fonction qui vérifie qu'une* Statement *existe dans un* bloc *d'une* Trace.
5) GetBlocOperand(bloc, Trace) est *la fonction qui retourne le nom du* BasicSequenceDiagram *des messages d'un bloc.*

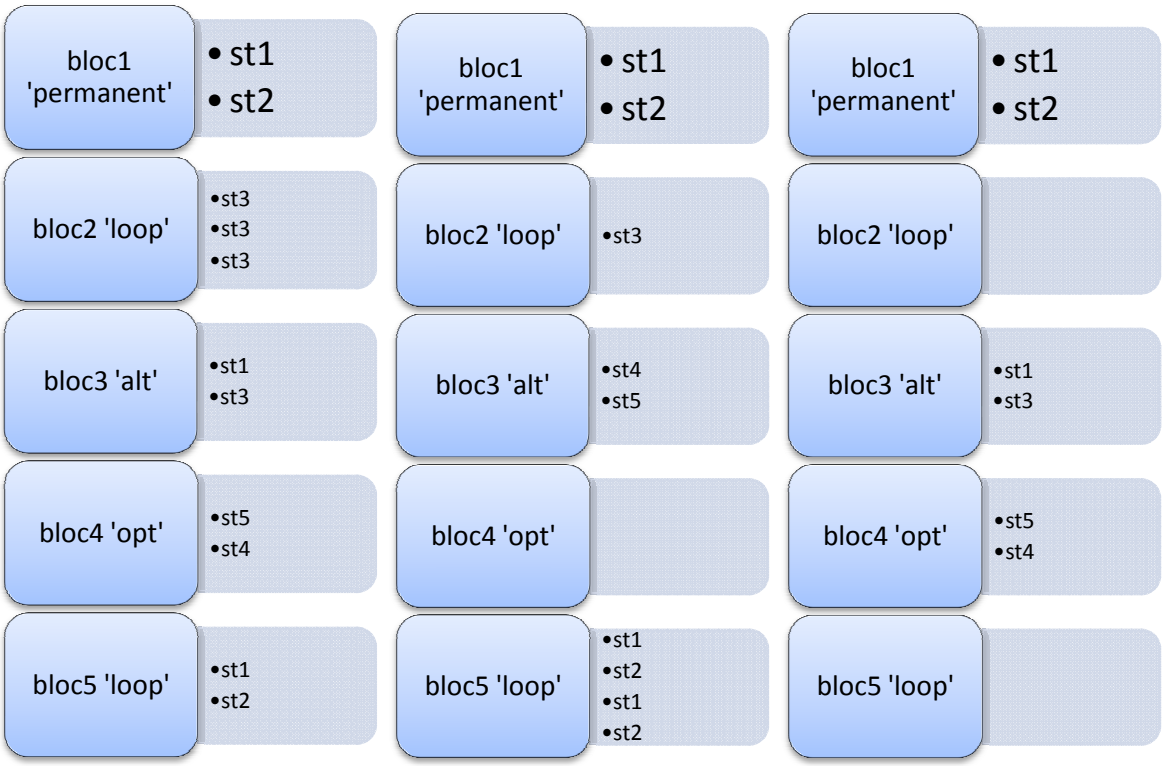

Fig. 4.4. La décomposition des traces en blocs

3.2.1 Détection des opérateurs

a) Fragment combiné avec l'opérateur "loop"

Hypothèse. Comme nous l'avons présenté dans le chapitre 2 (UML), l'opérateur d'interaction « loop » permet de spécifier une itération (i.e. boucle) d'un ou de plusieurs messages. L'hypothèse que nous utilisons pour détecter les fragments combinés avec l'opérateur « loop » est basée donc sur la présence d'une répétition d'une séquence de statements dans

les traces considérées. En effet, pour chaque trace d'exécution, le fait qu'on voit la même séquence de statements continument (plus qu'une fois), nous constatons l'existence d'un fragment combiné de type "loop".

Algorithme. L'algorithme que nous proposons pour détecter l'opérateur « loop » appelé « *Algo1» et illustré ci-dessous. Il* prend en entrée une trace et un SequenceDiagram qui est initialement vide. Cet algorithme doit être appliqué sur toute trace. Enfin, on obtient comme résultat un SequenceDiagram avec des CombinedFragments d'opérateur "loop". Le principe de cet algorithme est basé sur le parcours séquentiel de toutes les statements. Pour chaque Statement (4, 13), on cherche sa première occurrence (7, 8), si une occurrence est trouvée (14) on vérifie l'équivalence des successeurs de la Statement avec les successeurs de son occurrence (15). Si l'équivalence est correcte on adapte le SequenceDiagram (16, 21) et on cherche les itérations éventuelles de la séquence détectée (22, 28).

Algo1;

Inputs : Tr=(M,T,<) Є *Trace*, SD=(Fc,<) Є **SequenceDiagram**
Outputs : SD avec fc=(F,Op) Є **CombinedFragment** where Op = loop
Variables i :*integer*; m, succ , next: *Statement; detected: Boolean; Tr1:* **Trace***;*

```
1:  Begin
2:  SD:=null;
3:  m:= FirstStatement (Tr) ;
4:  For i :=1 to T do begin
5:      detected:=false;
6:      succ= successor (m, Tr);
7:      If (m=succ) then
8:          Detected:= true
9:      Else begin
10:         succ:= successor(succ,Tr);
11:         If (succ = null) go to 14:;
12:         Else go to 7:;
13:     end
14:     If (detected) then begin
15:        If (isEquivalent(successorSet(succ,Tr),predecessorSet(succ,Tr))) then begin
16:           SD.newCombinedfragment(cf,loop);
17:           cf. newOperand(opd) ;
18:           opd. SetOperand(succ+ successorSet (succ,Tr)) ;
19:           opd. SetOperand(m+ predecessorSet(succ,Tr)) ;
20:           next := successor(LastStatement (successorSet(succ,Tr)),Tr) ;
21:           Tr1:= successorSet (next,Tr);
22:           While (Tr1<> Φ) then
23:              If (isEquivalent(successorSet (succ,Tr),Tr1) then;begin
24:                 opd. SetOperand(next + successorSet (next,Tr));
25:                 next := successor(LastStatement (successorSet (next)),Tr) ;
26:                 Tr1:= successorSet (next,Tr) ;
```

```
27:                end
28:                Else begin m:=next; go to 31:; end
29:           End
30:           Else begin succ:= successor(succ,Tr); go to 7:;end
31:       End
32:       Else m :=successor(m,Tr);
33:  End
34:  end Algo 1
```

Discussion. Nous devons exécuter cette routine pour chaque trace, et le plus grand nombre de traces analysées implique le plus grand nombre de CombinedFragments d'opérateur "loop" détectés.

Prenons cet exemple :

```
For (int i=0; i<variable; i++){
        O1.M1();
        O2.M2();
}
```

Ce code ressemble à un CombinedFragment d'opérateur "loop", mais il ne peut pas être détecté dans une session d'exécution où « variable=0 ». Si « variable=0 » dans toutes les sessions d'exécution, ce CombinedFragment ne peut jamais être détecté, ce qu'explique le besoin d'aide auprès l'expert du système étudié.

Illustration. Après avoir appliqué ce premier algorithme sur les traces de l'exemple de Vente, nous avons obtenu le diagramme de séquence illustré dans la Figure Fig. 4.5. Ce diagramme de séquence comporte les CombinedFragments de type loop.

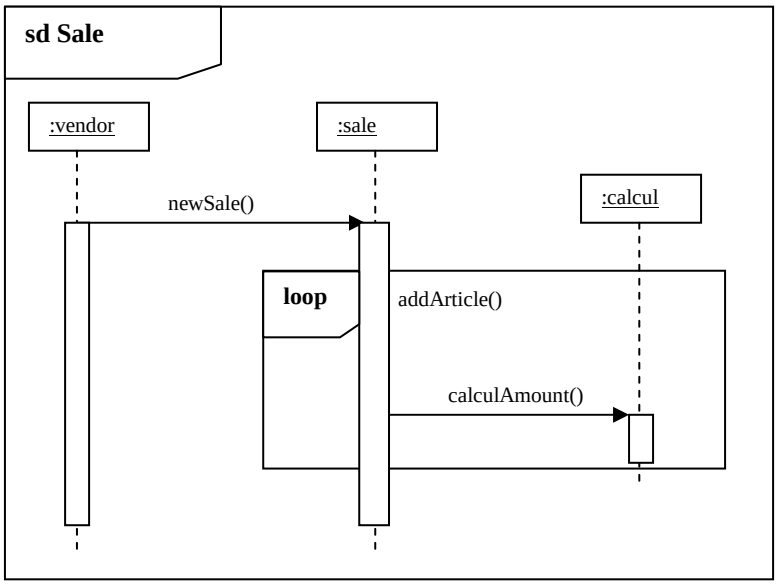

Fig. 4.5. DS de *Vente* après application de l'algorithme détection de loop.

b) Fragment combiné avec l'opérateur "opt"

Hypothèse. L'opérateur d'interaction « opt » permet de spécifier dans un diagramme de séquence qu'un ou plusieurs messages sont optionnels. L'hypothèse que nous avons adopté pour détecter cet opérateur consiste à parcourir tous les blocs et si une statement est absente dans un bloc d'une trace on va mettre cette statements dans un CombinedFragment avec un opérateur « opt ».

Algorithme. L'algorithme « Algo2» ci-dessous permet de détecter l'opérateur d'interaction « opt ». Il prend en entrée toutes les traces et le diagramme de séquence obtenu après l'application de l'Algorithme « Algo1 » i.e., un diagramme de séquence avec seulement des opérateurs « loop » et il renvoie comme résultat un SequenceDiagram avec CombinedFragments d'opérateurs "loop, opt". Pour chaque Statement appartenant à l'interaction principale 'Global_SD' (6,7), en commençant par la première (5), on confirme l'existence de ses occurrences dans le reste des traces au même bloc (8, 10), et si cela est confirmé alors on adapte le type du bloc de la trace courante au type indiqué par les autres traces (12, 16). Dans le cas contraire (i.e. absence d'occurrences au moins dans une trace), on marque le bloc comme optionnel (18).

Algo2:

Inputs : Tr1,Tr2, …,TrN=(M,T,<) Є **Trace,** SD=(Fc,<) Є **SequenceDiagram** avec fc=(F,Op) Є **CombinedFragment** where Op = loop

Outputs : SD=(Fc,<) Є **SequenceDiagram** avec fc=(F,Op) Є **CombinedFragment** where Op = <loop, opt>.

Variables i,j,k, bloc :*integer*; m: *Statement; oper,operOpt:BasicSequenceDiagram;*

```
1:  Begin
2:  SD.newCombinedfragment(cf,opt);
3:  cf. newOperand(operOpt) ;
4:  for k:=1 to N do // N is the number of traces
5:      m:= FirstStatement (Trk) ;
6:      For i :=1 to T do begin  // T is the number of Statements of Trace
7:          If (GetOperand(m)='Global_SD') then  begin  // Global_SD is the default name of the
    BasicSequenceDiagram
8:              bloc:=GetBloc(m);       // find the bloc order of current Statement
9:              for j:=1 to N<>k do begin    // N<>k do not compare Trace with itself
10:                 if(isExistStatement(m, bloc, Trj)) then begin // check existence of current
    statement in other trace at the same bloc
11:                     Oper:= GetBlocOperand(bloc,Trj);
12:                     if (oper<>'Global_SD') then  begin
13:                         oper.SetOperand(GetBlocSubTrace(bloc, Trk));
14:                         m:= successor(LastStatement(GetBlocSubTrace(bloc, Trk)), Trk);
15:                         i:=RANG(m,Trk)-1; j:=N+2;
```

```
16:                         end
17:                     end else begin
18:                             operOpt.SetOperand(m);
19:                             m:= successor(m,Trk); j:=N+2;
20:                         end
21:                 end
22:                 if (j=N) then
23:                    m:= successor(m,Trk);
24:         end
25:  end
26:  end algo2
```

À ce point tous les messages optionnels sont détectés inclus ceux qui sont réellement alternatifs, parce que tout message alternatif est optionnel et pas l'inverse. En outre, le modèle commun de la séquence des blocs des traces est juste entrain d'être déterminé, il contient des blocs de types "loop", "permanent", et "opt".

Il est prévu que les messages optionnels qui se cachent et apparaissent ensemble, soient appartenir au même CombinedFragment alternatif ou optionnel.

Discussion. Comme le cas des interactions itératives, si une condition n'a pas été réalisée dans toutes les traces, alors le CombinedFragment optionnel correspondant ne serait jamais détecté.

Illustrations. Après l'application de ce deuxième algorithme le précédent diagramme de séquence de vente est enrichi par toutes les CombinedFragments de type "opt" comme il est décrit dans la figure Fig. 4.6.

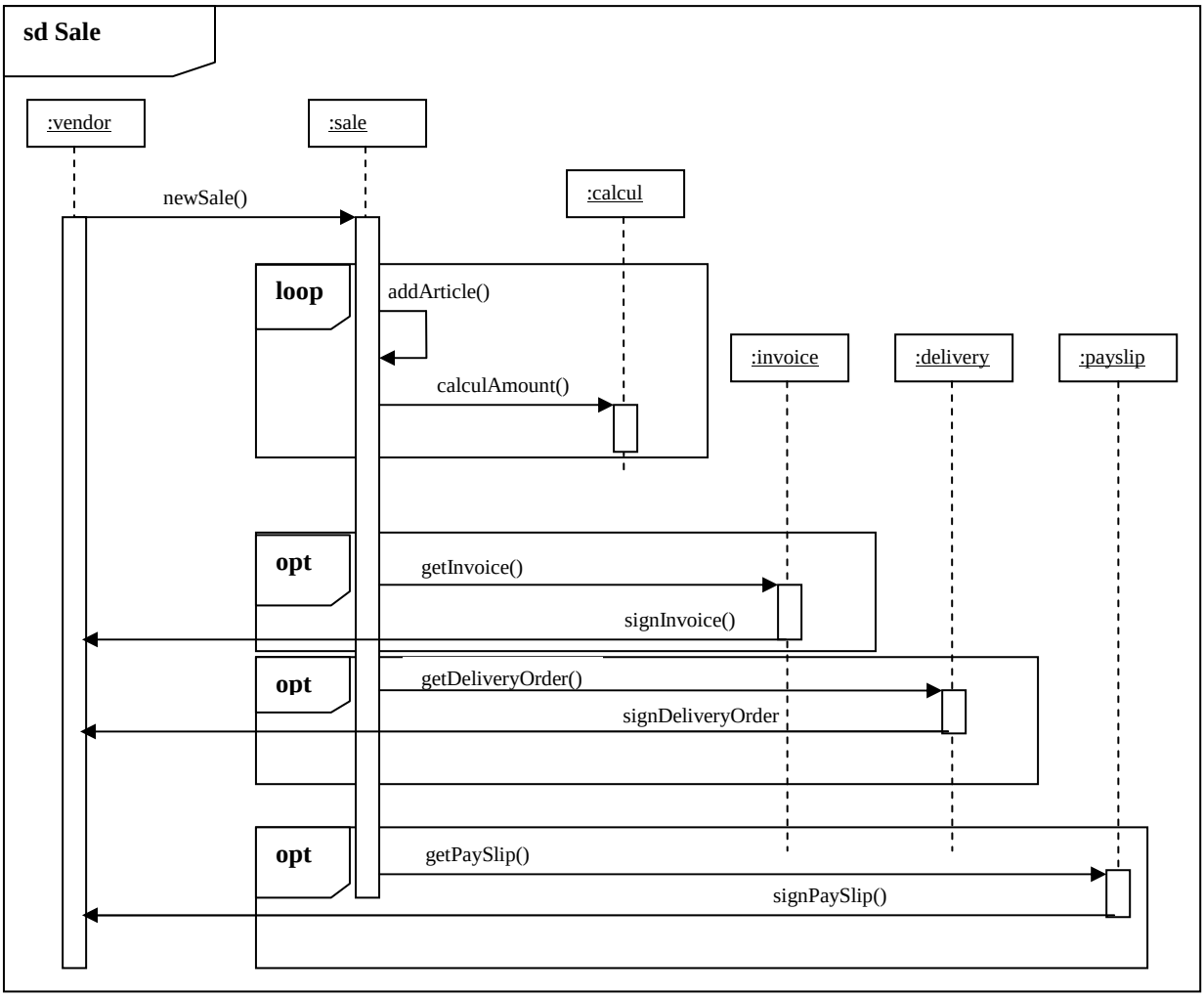

Fig. 4.6. DS de *Vente* après l'application de l'algorithme détection de opt

c) Fragment combiné avec l'opérateur "alt"

Hypothèse. Le cas de l'opérateur "alt" est le cas le plus complexe, car il dépend fortement des divergences entre les différentes traces. A ce stage-là notre SequenceDiagram contient des messages ordonnés certains appartenant aux opérandes de type "loop", d'autres sont permanentes, et le reste est optionnel. Pour détecter l'opérateur d'interaction "alt", nous avons établi quelques règles pour confirmer la présence d'un bloc alternative :

a) Tous les statements apparaissant dans le même bloc d'une trace ne sont pas des alternatives (elles peuvent être alternatives à une autre séquence de statements).

b) Chaque statement d'un bloc d'une trace est candidate d'être alternative aux autres statements absentes de son bloc.

c) Chaque bloc dans toutes les traces doit contenir exactement une parmi les séquences de statements alternatives.

d) Le comportement alternatif qui contient plus d'éléments (i.e., operands) est sélectionné comme le CombinedFragment de type "alt" pour le bloc.

En appliquant ces règles, on peut filtrer les blocs optionnels qui sont réellement alternatifs et on les adapte.

Avant d'entamer l'algorithme de détection de CombinedFragments de type « alt » nous expliquons d'abord les deux procédures **longestAltChain** et **decomposeChain** :

La chaine alternative la plus longue (longestAltChain).

Selon la règle **b)** nous composons la chaîne alternative la plus longue, c'est le cas où chaque statement d'un bloc est alternative aux autres statements :

Soit {mes***1***, mes***2***, .., mes***k***} l'ensemble des statements d'un bloc dans toutes les traces, alors la chaine alternative la plus longue est "mes***1***+ mes***2***+ ..+ mes***k***"
où '**+**' signifie alternatif.

La décomposition de la chaine (decomposeChain).

Si une chaîne alternative ne respecte pas la règle **a),** cette chaîne doit être décomposée à un nombre spécifique de sous-chaines qui respectent la règle **a)**. La **chaîne alternative principale** doit être racinée, par la suite les nouvelles **sous-chaînes alternatives** vont être construites en se basant sur la **chaîne racine**. Cette dernière peut être construite en éliminant les statements présentes dans le bloc de la trace en cours de traitement :

<root-chain> = <old-chain> – <trace bloc's statements>
En respectant les règles **a)** et **b),** on peut construire un nombre de sous-chaînes égale au nombre des combinaisons sans répétition construites en utilisant les statements du bloc de la trace courante:

$$\mathbf{NbrChains} = \sum_{\mathbf{i=1}}^{\mathbf{j}} \frac{\boldsymbol{j}!}{(\boldsymbol{j}-\boldsymbol{i})!\,\boldsymbol{i}!}$$

j: *nombre de* statements *du bloc de la trace courante.*

À partir de chaque combinaison de statements on construit une nouvelle chaîne alternative en se basant sur la chaîne racine comme suit :

<sub-chain> = <root-chain> + <combined-Chain >

Exemple: Soit l'ensemble de statements d'un bloc d'une trace {m1, m2, m3} alors : NbrChains = 7 et
sub-chains = {<<root-chain> + m1>, <<root-chain> + m2>, <<root-chain> + m3>,
<<root-chain> + m1m2>, <<root-chain> + m1m3>, <<root-chain> + m2m3>,
<<root-chain> + m1m2m3>}.

Algorithme. L'algorithme « *Algo3*» prend en entrée toutes les traces initiales et un SequenceDiagram avec CombinedFragments d'opérateurs "loop, opt" qui est le résultat de l'application des algorithmes « Algo1 » et « Algo2 ». Il renvoie comme résultat un SequenceDiagram avec CombinedFragments d'opérateurs "loop, opt, alt". Pour détecter des CombinedFragments d'opérateur "alt", on doit assembler toutes les statements de chaque bloc optionnel dans toutes les traces, puis on construit, en utilisant ces statements, **la chaîne alternative la plus longue** (11). Cette chaîne va être comparée avec les statements du bloc dans la première trace, qui soit ***l'ignorer*** **totalement** (si la règle **c)** n'est pas respectée) (12,15), ou **ils *supposent* sa décomposition** (18), alors respectivement**,** on supprime (16) ou ***on*** décompose (19) la chaîne. Dans le cas contraire où la chaîne est acceptée par les statements du bloc, on la maintient. Les chaines récemment composées et celles qui sont maintenues vont être comparées avec les statements du bloc des autres traces. Enfin, on applique la règle **d)** pour créer les CombinedFragment d'opérateur "alt" (25, 26).

Algo 3 :

Inputs : Tr1,Tr2, …,TrN=(M,T,<) ∈ **Trace,** SD=(Fc,<) ∈ **SequenceDiagram** avec fc=(F,Op) ∈ **CombinedFragment** where Op = <loop, opt>.
Outputs : SD=(Fc,<) ∈ **SequenceDiagram** avec fc=(F,Op) ∈ **CombinedFragment** where Op = <loop, opt,alt>.
Variables i,j,k, bloc :*integer*; m: *Statement; oper,operOpt:BasicSequenceDiagram; StatementSet :set of Statements;*
chainSet,subChainSet: set of chains, altCF: array of **CombinedFragment;**

```
1:  Begin
2:  For k:=1 to N do // N is the number of traces
3:      m:= FirstStatement (Trk) ;
4:      for i :=1 to T do begin  // T is the number of Statements of Trace
5:          if (GetOperand(m)='operOpt') then begin  // operOpt is an operand of type opt
6:              bloc:=GetBloc(m);       // find the bloc order of current Statement
7:              StatementSet:=Φ; chainSet:=Φ;
8:              For j:=1 to N do begin
9:                  StatementSet:= StatementSet+GetBlocSubTrace(bloc, Trj);
10:             end
11:             chain:= longestAltChain(StatementSet); chainSet:= chainSet+chain;
12:             for j:=1 to N do begin
13:                 subChainSet:=Φ;
14:                 for (every: chain ϵ chainSet) do begin
15:                     if (isIgnored(chain, GetBlocSubTrace(bloc, Trj))) then
16:                         deleteFromChainSet(chain);
17:                     else
18:                         if (chainRequireDecomposition) then begin
19:                             decomposeChain(chain, subChainSet);
20:                             deleteFromChainSet(chain);
21:                         end
```

```
22:                         end
23:                         chainSet:=chainSet+subChainSet;
24:                 end
25:                 altCF[bloc]:= SD.newCombinedfragment(ChooseLongestAltChain(chainSet),alt);
26:                 altCF[bloc]. newOperand(oper) ;
27:                 for j:=1 to N do begin
28:                     oper.SetOperand(GetBlocSubTrace(bloc, Trj));
29:                 m:= successor(LastStatement(GetBlocSubTrace(bloc, Trk)), Trk);
30:                 i:=RANG(m,Trk)-1;
31:             end
32:         end
33:  end
34:  end Algo 3
```

Cet algorithme commence par enfermer toutes les séquences alternatives possibles, puis il emploie une méthode d'exclusion itérative en respectant les combinaisons de statements des blocs dans chaque trace. Finalement, on obtient soit zéro, un, ou plusieurs chaînes alternatives appropriées à toutes les combinaisons de statements du bloc dans toutes les traces, automatiquement la chaine la plus longue est choisie (règle **d)**).

Nous rappelons que l'ordre d'application de ces trois algorithmes est très important, et si on change cet ordre, les résultats ne seront plus achevés.

Illustrations. Après l'application de ce dernier algorithme le précédent diagramme de séquence de vente est enrichi par toutes les CombinedFragments de type "alt" la figure Fig.4.7. montre le diagramme de séquence final de l'application Vente, celui-ci comprend les trois types de CombinedFragment à savoir : loop, opt et alt.

d) Fragment combiné avec l'opérateur "seq"

Comme nous avons indiqué ci-dessus dans la section 4.2.2 du chapitre II, le diagramme de séquence d'UML2 supporte deux types d'opérateurs de séquence, dont : 1) l'opérateur "seq" (pour la séquence faible) et 2) l'opérateur "strict". Tous les CombinedFragments qui sont détectés par nos algorithmes (d'opérateur "opt", "alt" ou "loop"), sont en séquence de type "seq" selon l'ordre de leurs messages tel qu'il est montré dans la figure Fig. 4.7.

Finalement le diagramme de séquence est complètement construit. Il nous reste que l'exploiter en l'exposant aux utilisateurs par des techniques de visualisation, ou par des outils UML (cf. Fig 5.12.).

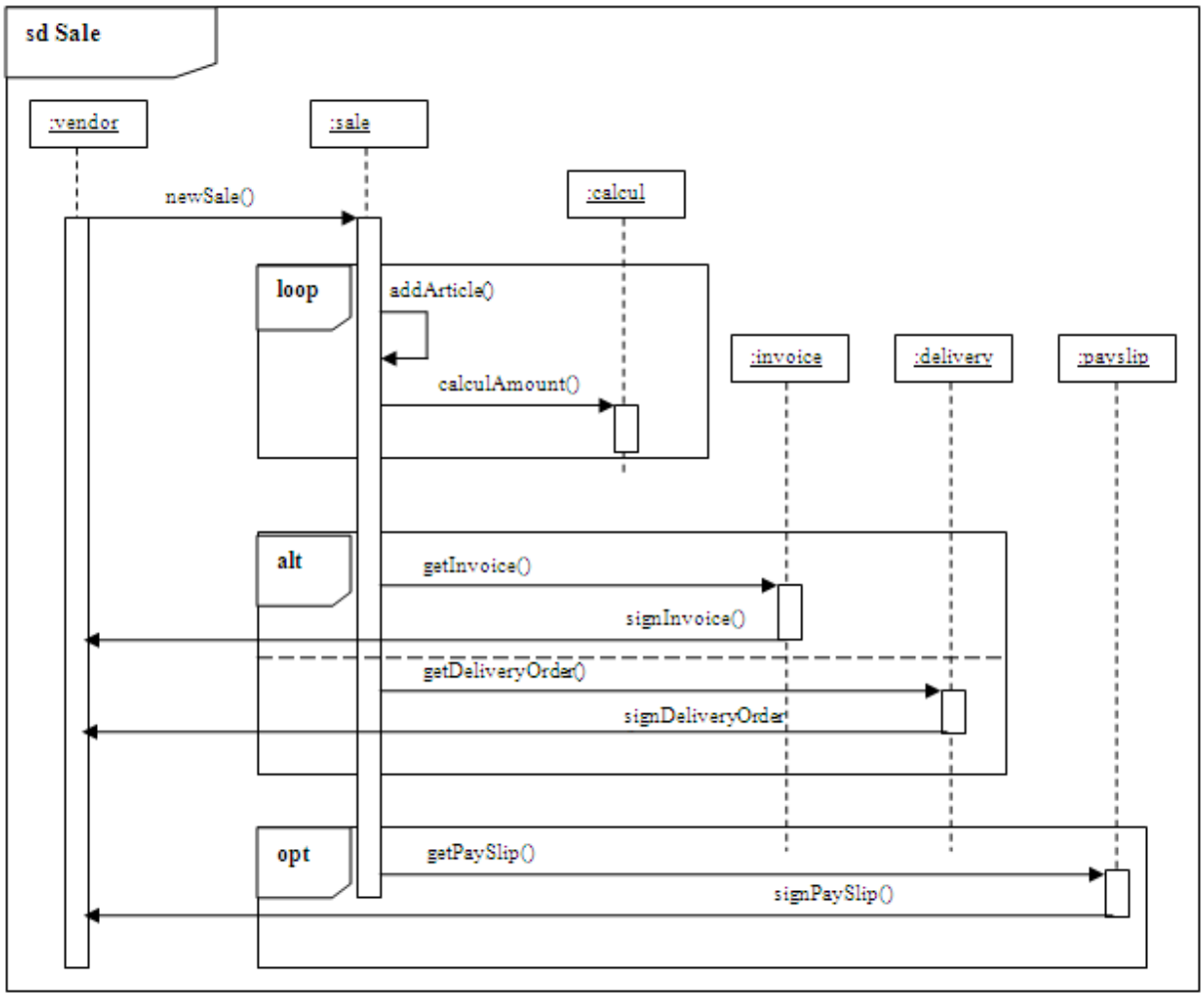

Fig. 4.7. DS final de l'application *Vente*

4. Conclusion

Dans ce chapitre nous avons présenté en détail notre approche de la rétro-ingénierie des diagrammes de séquence d'UML2 à partir d'un système logiciel. Elle se résume essentiellement dans deux étapes :

1. Collection des traces : en instrumentant le logiciel ciblé puis on l'exécute autant de fois que le nombre de scénarios possibles.
2. Construction du diagramme de séquence complet : en appliquant itérativement les heuristiques et traitements de détection des opérateurs "loop, opt, alt" dans l'ordre précisé.

Dans le chapitre prochain nous allons voir les détails de l'implémentation de l'approche présentée.

Chapitre 5 Implémentation

L'objectif de ce chapitre est de présenter les détails d'implémentation de l'approche incrémentale, où nous décrivons les outils, langages et plateformes utilisées. Ainsi, nous présentons les vues progressives de notre solution par rapport à l'exemple d'illustration *Vente*.

1. Outils d'implémentation

Nous avons choisi le langage Java et les bases de données Access pour implémenter notre solution.

1.1 Outils de collection de traces

Parmi plusieurs outils de collection de traces pour les systèmes logiciels Java, nous avons choisi celui de Janice [5] appelé MoDec. Celui-ci est un outil d'instrumentation de code intermédiaire Java (java bytecode) (voir fig. 5.1., fig. 5.2., fig. 5.3.), donc il a comme avantage de ne pas toucher le code source ce qui donne la possibilité de modifier le code et de relancer l'instrumentation à nouveau. Une autre option très intéressante est également fournie par l'outil MoDec où l'utilisateur peut choisir parmi les classes du programme celles qu'il veut faire suivre et instrumenter.

```
public static void m1 () {
  a++;
}

public static void main (String args []){
  for(int counter = 0 ; counter < 5; counter++){
    a = 0;
    while (a < 2){
      if(counter == 1)
        break ;
      else if (a < 2)
        m1();
    }
    if(counter == 2)
      return ;
  }
}
```

Fig. 5.1. Code source de l'exemple Loop.java

```
public static void m1()
Code(max_stack = 2, max_locals = 0, code_length = 9)
0:    getstatic                          Loops.a I (2)
3:    iconst_1
4:    iadd
5:    putstatic                          Loops.a I (2)
8:    return
```

Fig. 5.2. Version non instrumenté de la méthode m1().

```
public static void m1()
Code(max_stack = 3, max_locals = 1, code_length = 56)
0:    ldc "ControlStructure.trace" (27)
2:    ldc "METHOD entry Loops.public static void m1() CALLER " (59)
4:    invokestatic tool.instrumentation.LogToFile.write …
7:    new <java.lang.Throwable> (47)
10:   dup
11:   invokespecial java.lang.Throwable.<init> ()V (48)
14:   invokevirtual java.lang.Throwable.getStackTrace () …
17:   astore_0
18:   ldc "ControlStructure.trace" (27)
20:   aload_0
21:   iconst_1
22:   aaload
23:   invokestatic tool.instrumentation.LogToFile.write …
26:   invokestatic Loops.m1$impl ()V (61)
29:   ldc "ControlStructure.trace" (27)
31:   ldc "METHOD exit Loops.public static void m1() CALLER " (63)
33:   invokestatic tool.instrumentation.LogToFile.write …
36:   new <java.lang.Throwable> (47)
39:   dup
40:   invokespecial java.lang.Throwable.<init> ()V (48)
43:   invokevirtual java.lang.Throwable.getStackTrace ()…
46:   astore_0
47:   ldc "ControlStructure.trace" (27)
49:   aload_0
50:   iconst_1
51:   aaload
52:   invokestatictool.instrumentation.LogToFile.write …
55:   return
```

Fig. 5.3. Version instrumenté de la méthode m1().

Après l'instrumentation du bytecode par MoDec, et en lançant l'exécution du programme instrumenté un fichier de trace va être généré (voir fig. 5.4.). Ce fichier contient tous les événements de construction et de destruction des objets des classes instrumentées, ainsi les événements d'appel et de retour de ses méthodes invoquées sont aussi enregistrés. La forme de traces collectées peut différer d'un outil à un autre, ce qui nous a obligés de développer un adaptateur qui réorganise les traces dans une nouvelle forme adaptée où chaque

trace se compose d'une suite de statements, et chaque statement dans une ligne de la trace se compose de l'objet appelant, la méthode exécutée, et l'objet appelé. Le rôle de l'adaptateur est de restructurer la trace dans une forme appropriée aux traitements de l'étape de la construction du diagramme. Un exemple de cette forme est illustré dans la figure Fig. 5.5.

Trace 2

- METHOD entry public static void main(String[] args) CALLEE Vendor -1
- CONSTRUCTOR entry public void <init>() CALLEE Sale 1333195
- CONSTRUCTOR exit public void <init>() CALLEE Sale 1333195
- METHOD entry public void newSale(int nbr_article, boolean isInvoice, boolean isPayslip) CALLEE Sale 1333195
- METHOD entry public static float addArticle() CALLEE Sale -2
- METHOD exit public static float addArticle() CALLEE Sale -2
- CONSTRUCTOR entry public void <init>() CALLEE Calcul 10730689
- CONSTRUCTOR exit public void <init>() CALLEE Calcul 10730689
- METHOD entry public float calculAmount(float newValue, float oldValue) CALLEE Calcul 10730689
- METHOD exit public float calculAmount(float newValue, float oldValue) CALLEE Calcul 10730689
- CONSTRUCTOR entry public void <init>() CALLEE Invoice 29181730
- CONSTRUCTOR exit public void <init>() CALLEE Invoice 29181730
- METHOD entry public void getInvoice() CALLEE Invoice 29181730
- METHOD entry public static void signInvoice() CALLEE Vendor -2
- METHOD exit public static void signInvoice() CALLEE Vendor -2
- METHOD exit public void getInvoice() CALLEE Invoice 29181730
- CONSTRUCTOR entry public void <init>() CALLEE Payslip 15980197
- CONSTRUCTOR exit public void <init>() CALLEE Payslip 15980197
- METHOD entry public void getPayslip() CALLEE Payslip 15980197
- METHOD entry public static void signPayslip() CALLEE Vendor -2
- METHOD exit public static void signPayslip() CALLEE Vendor -2
- METHOD exit public void getPayslip() CALLEE Payslip 15980197
- METHOD exit public void newSale(int nbr_article, boolean isInvoice, boolean isPayslip) CALLEE Sale 1333195
- METHOD exit public static void main(String[] args) CALLEE Vendor -1

Fig. 5.4. Trace 2 de l'application Vente générée par l'outil MoDec

Fig. 5.5. trace2 adapté de l'application Vente

1.2 Stockage des données

Nous avons représenté les instances du Méta-Modèle du diagramme de séquence d'UML2 par un modèle de base de données relationnel (voir la figure Fig. 5.6), ce modèle est un intermédiaire entre l'ensemble de fichiers de traces (entrée) et les outils de visualisation et de communication (sortie). En effet, nous reposons sur les capacités du langage SQL pour fournir des informations au modèle relationnel, traiter et analyser ses informations, et également pour extraire le diagramme de séquence dans une forme adéquate (par exemple XMI [53]) aux outils UML2. La figure Fig. 5.6 montre les tables principales qui stockent les informations tout au long des périodes d'évolution du diagramme de séquence. Dans ce travail initial nous nous sommes concentrés sur les éléments indispensables d'un diagramme de séquence. Cependant, notre modèle relationnel reste ouvert pour des nouvelles modifications structurelles. Cet avantage nous permet de faire des évolutions futures pour supporter d'autres éléments du méta-modèle UML. Notre base de données utilisée est composée des tables suivantes :

Table *Mess_Tr* : cette table stocke le contenu des traces au début sans aucune modification. Le champ *Tr_file* contient le numéro du fichier de trace, *Ord_mess* stocke l'ordre du message tel qu'il apparaît dans le fichier de trace. Le champ *Cde_mess* contient un code affecté pour identifier les messages et leurs occurrences dans tous les fichiers de traces. *Obj_Sd, Obj_Rcv,* et *Mthd* sont respectivement l'objet appelant, l'objet appelé, et la méthode invoquée. *Int_fr* et

operand sont utilisés dans la suite des traitements pour sauvegarder respectivement le fragment combiné et l'opérande auxquels appartient un message. Nous pouvons identifier chaque occurrence d'un message dans tous les traces par le numéro de la trace *Tr_file* et son ordre *Ord_mess.*

Table *Operands* : cette table stocke tous les opérandes recouverts tout au long du processus de construction du diagramme. Le champ *level* indique le niveau d'imbrication, *rang* contient l'ordre de l'opérande détecté. *name_op* contient le nom attribué à l'opérande, ce nom se compose de suffixe "OP" plus les codes des messages appartenant à cet opérande séparés par des points. Le champ Combined_f réfère au nom de fragment combiné auquel appartient cet opérande.

Table *Combined_fragment* : dans cette table nous conservons tous les fragments combinés détectés. Le champ level indique le niveau d'imbrication, rang contient l'ordre de fragment combiné dans le diagramme de séquence, cet ordre ressemble à celui de l'opérateur seq dans le méta-modèle du diagramme de séquence. name_cf contient le nom attribué au fragment combiné, ce nom se compose du suffixe "SD" plus les codes des messages appartenant à cet fragment. Le champ operator indique le type d'opérateur du fragment combiné : loop, opt, ou alt. Tandis que le champ operand réfère à l'opérande auquel appartient ce fragment combiné dans le cas des imbrications.

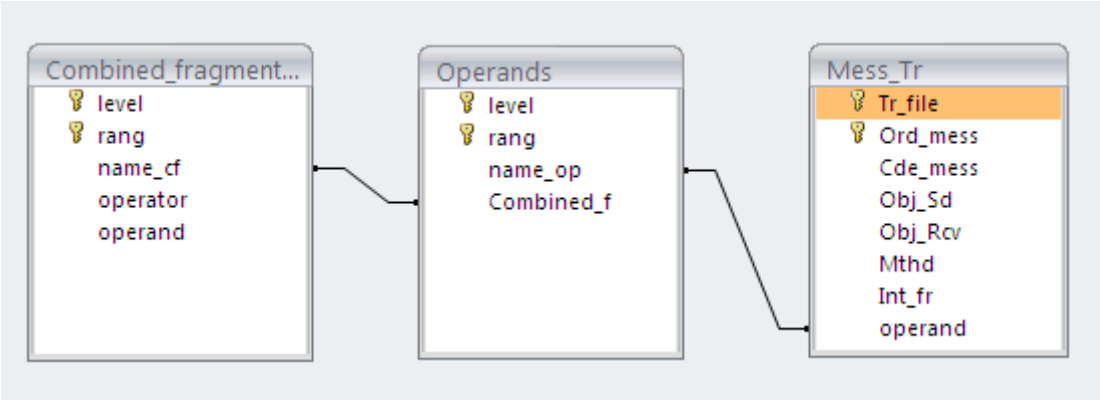

Fig. 5.6. Le Modèle relationnel intermédiaire

A la fin de l'application des heuristiques de construction du diagramme de séquence le modèle relationnel va contenir dans ses tables toutes les informations du diagramme de séquence dans une forme structurée et facile d'être interrogée. La table Combined_fragment contient tous les fragments combinés, ces fragments représentent le comportement général du logiciel étudié. La table operands contient tous les opérandes des fragments combinés stockés

dans la table Combined_fragment. Quant à la table Mess_Tr, elle garde toujours les informations des objets en interaction et ses méthodes.

Pour extraire ces informations vers une représentation standard et exploitable par les outils UML2. Nous nous sommes basés sur les requêtes SQL pour piloter la transformation des diagrammes de séquence au format respectant le standard UML. Cette transformation est réalisée en utilisant les primitives de l'API UML2 que nous allons présenter dans ce qui suit.

1.3 API UML2

L'API UML2 est une bibliothèque Java qui contient l'ensemble des classes nécessaires à la création et à la gestion automatique de l'ensemble des modèles d'UML2 en respectant les règles et la structure du méta-modèle UML2. Cette API permet également de sauvegarder ces modèles dans des fichiers de format standard de l'intercommunication XMI.

Exemples

Pour créer un modèle UML :

```
Model myModel = UMLFactory.eINSTANCE.createModel();
```

Pour créer une interaction UML et initialiser son nom :

```
Interaction
    interaction=UMLFactory.eINSTANCE.createInteraction();
interaction.setName("myIntercation");
```

Pour créer un opérande, le relier à l'interaction, et initialiser son nom :

```
InteractionOperand
    operand=UMLFactory.eINSTANCE.createInteractionOperand();
operand.setEnclosingInteraction(interaction);
operand.setName("OP_0");
```

Nous avons implémenté un programme qui se comporte comme un pilote de ces instructions de code afin de générer un diagramme de séquence respectant le standard UML2.

2. Application de l'approche sur l'exemple *Vente*

2.1 Cas d'utilisation

Pour prouver l'efficacité de notre approche, nous avons choisi le processus de vente comme un cas d'utilisation. Nous signalons que la contrainte des imbrications a rendu très difficile de trouver une application réelle pour valider notre approche.

Dans ce qui suit nous présentons l'application détaillée de notre approche sur l'exemple de Vente.

Etape 1 : l'étape de collection de traces

Nous avons obtenu trois différentes traces générées par l'outil *MoDec*, après trois sessions d'exécution de l'application Vente (voir Fig. 5.4). Dans ces trois sessions nous avons suivi trois scénarios d'exécution en fournissant différentes combinaisons de valeurs d'entrée. Et nous sommes conscients que ces trois scénarios d'exécutions sont assez suffisants pour recouvrir le comportement général de l'application Vente.

Nous nous sommes intéressés aux interactions entre classes, et nous avons substitué les objets de la même classe par le nom de leur classe. La figure Fig. 4.2 montre les trois traces après le filtrage des données et l'adaptation de la structure des traces pour être plus simple à analyser. A la fin de cette étape, le contenu des trois traces est chargé dans la table Mess_Tr, avec quelques informations supplémentaires. Dans cet état initial, l'ordre des messages est maintenu, un code identificateur est attribué aux messages, et tous les messages sont considérés initialement appartenant à l'interaction principale Global_SD tel qu'il est montré dans la figure Fig. 5.7.

Etape 2 : l'étape d'analyse et construction du diagramme de séquence

La figure 5.8 affiche le contenu de la table ***Mess_Tr*** à la fin de cette étape. On peut voir que le contenu de certaines lignes est modifié, précisément des champs *int_fr* et *operand*.

Mess_Tr							
Tr_file	Ord_mess	Cde_mess	Obj_Sd	Obj_Rcv	Mthd	Int_fr	operand
1	1	1	Agent	Vendor	main	Global_SD	
1	2	2	Vendor	Sale	newSale	Global_SD	
1	3	3	Sale	Sale	addArticle	Global_SD	
1	4	4	Sale	Calcul	calculAmount	Global_SD	
1	5	3	Sale	Sale	addArticle	Global_SD	
1	6	4	Sale	Calcul	calculAmount	Global_SD	
1	7	3	Sale	Sale	addArticle	Global_SD	
1	8	4	Sale	Calcul	calculAmount	Global_SD	
1	9	5	Sale	Delivery	getDelivery	Global_SD	
1	10	6	Delivery	Vendor	signDelivery	Global_SD	
2	1	1	Agent	Vendor	main	Global_SD	
2	2	2	Vendor	Sale	newSale	Global_SD	
2	3	3	Sale	Sale	addArticle	Global_SD	
2	4	4	Sale	Calcul	calculAmount	Global_SD	
2	5	7	Sale	Invoice	getInvoice	Global_SD	
2	6	8	Invoice	Vendor	signInvoice	Global_SD	
2	7	9	Sale	Payslip	getPayslip	Global_SD	
2	8	10	Payslip	Vendor	signPayslip	Global_SD	
3	1	1	Agent	Vendor	main	Global_SD	
3	2	2	Vendor	Sale	newSale	Global_SD	
3	3	3	Sale	Sale	addArticle	Global_SD	
3	4	4	Sale	Calcul	calculAmount	Global_SD	
3	5	3	Sale	Sale	addArticle	Global_SD	
3	6	4	Sale	Calcul	calculAmount	Global_SD	
3	7	3	Sale	Sale	addArticle	Global_SD	
3	8	4	Sale	Calcul	calculAmount	Global_SD	
3	9	3	Sale	Sale	addArticle	Global_SD	
3	10	4	Sale	Calcul	calculAmount	Global_SD	
3	11	5	Sale	Delivery	getDelivery	Global_SD	
3	12	6	Delivery	Vendor	signDelivery	Global_SD	
3	13	9	Sale	Payslip	getPayslip	Global_SD	
3	14	10	Payslip	Vendor	signPayslip	Global_SD	

Fig. 5.7. Vue initiale de la table Mess_Tr

Mess_Tr							
Tr_file	Ord_mess	Cde_mess	Obj_Sd	Obj_Rcv	Mthd	Int_fr	operand
1	1	1	Agent	Vendor	main	Global_SD	
1	2	2	Vendor	Sale	newSale	Global_SD	
1	3	3	Sale	Sale	addArticle	SD3.4	OP3.4
1	4	4	Sale	Calcul	calculAmount	SD3.4	OP3.4
1	5	3	Sale	Sale	addArticle	SD3.4	OP3.4
1	6	4	Sale	Calcul	calculAmount	SD3.4	OP3.4
1	7	3	Sale	Sale	addArticle	SD3.4	OP3.4
1	8	4	Sale	Calcul	calculAmount	SD3.4	OP3.4
1	9	5	Sale	Delivery	getDelivery	SDA5.6+7.8	OPA5.6
1	10	6	Delivery	Vendor	signDelivery	SDA5.6+7.8	OPA5.6
2	1	1	Agent	Vendor	main	Global_SD	
2	2	2	Vendor	Sale	newSale	Global_SD	
2	3	3	Sale	Sale	addArticle	SD3.4	OP3.4
2	4	4	Sale	Calcul	calculAmount	SD3.4	OP3.4
2	5	7	Sale	Invoice	getInvoice	SDA5.6+7.8	OPA7.8
2	6	8	Invoice	Vendor	signInvoice	SDA5.6+7.8	OPA7.8
2	7	9	Sale	Payslip	getPayslip	SDO9.10	OPO9.10
2	8	10	Payslip	Vendor	signPayslip	SDO9.10	OPO9.10
3	1	1	Agent	Vendor	main	Global_SD	
3	2	2	Vendor	Sale	newSale	Global_SD	
3	3	3	Sale	Sale	addArticle	SD3.4	OP3.4
3	4	4	Sale	Calcul	calculAmount	SD3.4	OP3.4
3	5	3	Sale	Sale	addArticle	SD3.4	OP3.4
3	6	4	Sale	Calcul	calculAmount	SD3.4	OP3.4
3	7	3	Sale	Sale	addArticle	SD3.4	OP3.4
3	8	4	Sale	Calcul	calculAmount	SD3.4	OP3.4
3	9	3	Sale	Sale	addArticle	SD3.4	OP3.4
3	10	4	Sale	Calcul	calculAmount	SD3.4	OP3.4
3	11	5	Sale	Delivery	getDelivery	SDA5.6+7.8	OPA5.6
3	12	6	Delivery	Vendor	signDelivery	SDA5.6+7.8	OPA5.6
3	13	9	Sale	Payslip	getPayslip	SDO9.10	OPO9.10
3	14	10	Payslip	Vendor	signPayslip	SDO9.10	OPO9.10

Fig. 5.8. Vue finale de la table Mess_Tr

Dans la figure Fig. 5.9, la table Combined_fragments contient les fragments combinés détectés dans cette étape. Cette table indique l'ordre, ou bien, la séquence des fragments combinés dans le champ *rang*. Dans cette table, Le fragment **Global_SD1** et **Gllobal_SD2** représente l'ordre des messages permanents respectivement 1 et 2 par rapport aux fragments

combinés. Dans la version actuelle d'UML, il n'existe pas des fragments combinés de type permanent, toutefois, nous avons signalé que nous avons besoin de telle représentation.

SD3.4: un fragment combiné avec l'opérateur "loop" contient deux messages représentés par les codes: '3' et '4'.

SDA5.6+7.8: un fragment combiné avec l'opérateur "alt" contient deux opérandes : '5.6' et '7.8'.

SDO9.10 : un fragment combiné avec l'opérateur "opt" contient deux messages représentés par les codes : '9' et '10'.

La table operands (voir Fig. 5.10) montre les opérandes des différents fragments combinés, nous pouvons connaître le fragment combiné de chaque opérandes en consultant les valeurs du champ Combined_f : **OP3.4** : un opérande d'un fragment combiné itératif. **OPA5.6** : un opérande d'un fragment combiné alternatif dont appartiennent deux messages : '5' et '6'.

OPO9.10 : un opérande d'un fragment combiné optionnel.

Combined_fragments				
level	rang	name_cf	operator	operand
0	0	Global_SD	seq	
1	1	Global_SD1	per	
1	2	Global_SD2	per	
1	3	SD3.4	loop	
1	4	SDA5.6+7.8	alt	
1	5	SDO9.10	opt	

Fig. 5.9. Vue finale de la table Combined_fragments

Operands			
level	rang	name_op	Combined_f
1	1	OP3.4	SD3.4
1	3	OPA5.6	SDA5.6+7.8
1	4	OPA7.8	SDA5.6+7.8
1	5	OPO9.10	SDO9.10

Fig. 5.10. Vue finale de la table Operands

Etape 3 : l'étape d'extraction du diagramme de séquence

Nous avons utilisé des requêtes SQL pour extraire les informations nécessaires à la visualisation ou l'intercommunication du diagramme de séquence. En utilisant les primitives de l'API UML2, nous avons construit le diagramme de séquence correspondant à l'application *Vente*. La figure Fig. 5.11 montre une vue du diagramme de séquence dans un

éditeur Eclipse. Ainsi, la figure Fig. 5.12 présente une partie du document UML généré. Cette partie représente un fragment combiné, de type "alt", qui contient deux opérandes : OPA5.6 et OPA7.8. Les diagrammes de séquence dans la figure Fig. 5.11 et dans la figure Fig. 4.7 représentent le même comportement général de l'application *Vente*, avec seulement deux différences qui sont l'objet Agent et la méthode main (l'agent qui lance l'exécution de l'application par sa méthode initiale main).

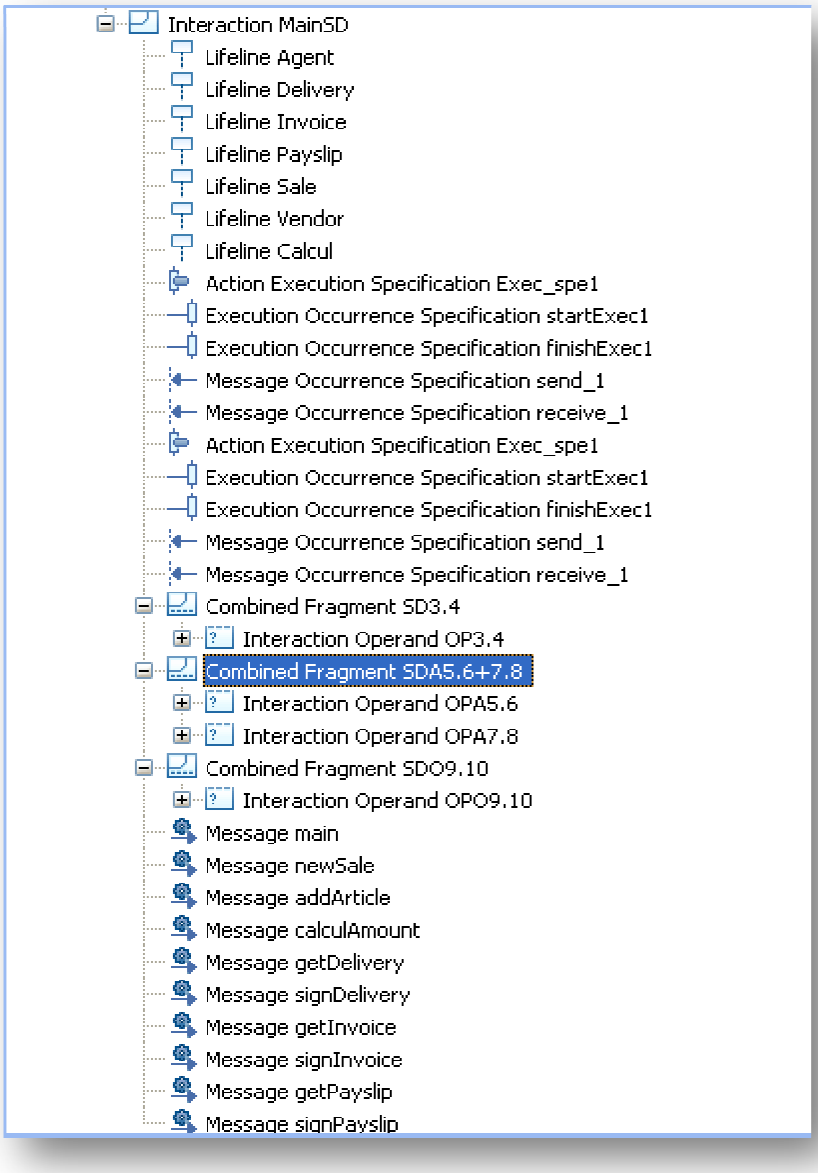

Fig. 5.11. Le Diagramme de séquence généré affiché par un éditeur Eclipse

```
<fragment xmi:type="uml:CombinedFragment" xmi:id="_zyrdfmJnEd6iXaV5GJoiXQ"
name="SDA5.6+7.8" interactionOperator="alt">
      <operand xmi:id="_zyrdf2JnEd6iXaV5GJoiXQ" name="OPA5.6">
        <fragment xmi:type="uml:ActionExecutionSpecification"
xmi:id="_zyrdgGJnEd6iXaV5GJoiXQ" name="Exec_spe1" start="_zyrdgWJnEd6iXaV5GJoiXQ"
finish="_zyrdgmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification"
xmi:id="_zyrdgWJnEd6iXaV5GJoiXQ" name="StartExec_1" covered="_zyrdYmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification"
xmi:id="_zyrdgmJnEd6iXaV5GJoiXQ" name="FinishExec_1" covered="_zyrdYmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification"
xmi:id="_zyrdg2JnEd6iXaV5GJoiXQ" name="send_1" covered="_zyrdZWJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification"
xmi:id="_zyrdhGJnEd6iXaV5GJoiXQ" name="receive_1" covered="_zyrdYmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ActionExecutionSpecification"
xmi:id="_zyrdhWJnEd6iXaV5GJoiXQ" name="Exec_spe2" start="_zyrdhmJnEd6iXaV5GJoiXQ"
finish="_zyrdh2JnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification"
xmi:id="_zyrdhmJnEd6iXaV5GJoiXQ" name="StartExec_2" covered="_zyrdZmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification"
xmi:id="_zyrdh2JnEd6iXaV5GJoiXQ" name="FinishExec_2" covered="_zyrdZmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification"
xmi:id="_zyrdiGJnEd6iXaV5GJoiXQ" name="send_2" covered="_zyrdYmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification"
xmi:id="_zyrdiWJnEd6iXaV5GJoiXQ" name="receive_2" covered="_zyrdZmJnEd6iXaV5GJoiXQ"/>
      </operand>
      <operand xmi:id="_zyrdimJnEd6iXaV5GJoiXQ" name="OPA7.8">
        <fragment xmi:type="uml:ActionExecutionSpecification"
xmi:id="_zyrdi2JnEd6iXaV5GJoiXQ" name="Exec_spe4" start="_zyrdjGJnEd6iXaV5GJoiXQ"
finish="_zyrdjWJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification"
xmi:id="_zyrdjGJnEd6iXaV5GJoiXQ" name="StartExec_4" covered="_zyrdY2JnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification"
xmi:id="_zyrdjWJnEd6iXaV5GJoiXQ" name="FinishExec_4" covered="_zyrdY2JnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification"
xmi:id="_zyrdjmJnEd6iXaV5GJoiXQ" name="send_4" covered="_zyrdZWJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification"
xmi:id="_zyrdj2JnEd6iXaV5GJoiXQ" name="receive_4" covered="_zyrdY2JnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ActionExecutionSpecification"
xmi:id="_zyrdkGJnEd6iXaV5GJoiXQ" name="Exec_spe5" start="_zyrdkWJnEd6iXaV5GJoiXQ"
finish="_zyrdkmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification"
xmi:id="_zyrdkWJnEd6iXaV5GJoiXQ" name="StartExec_5" covered="_zyrdZmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification"
xmi:id="_zyrdkmJnEd6iXaV5GJoiXQ" name="FinishExec_5" covered="_zyrdZmJnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification"
xmi:id="_zyrdk2JnEd6iXaV5GJoiXQ" name="send_5" covered="_zyrdY2JnEd6iXaV5GJoiXQ"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification"
xmi:id="_zyrdlGJnEd6iXaV5GJoiXQ" name="receive_5" covered="_zyrdZmJnEd6iXaV5GJoiXQ"/>
      </operand>
    </fragment>
```

Fig. 5.12. Une partie du DS représente un fragment combiné sous forme d'un document UML

Conclusion et perspectives

1. Évaluation

Dans ce travail nous avons revisité le domaine de la rétro-ingénierie des diagrammes de séquence d'UML2. En premier temps, nous avons réalisé un large état de l'art sur ce domaine et autour des travaux existants. Par la suite, nous avons proposé une nouvelle approche permettant de construire un diagramme de séquence à partir de plusieurs traces d'exécution du système. Notre approche a suivi le schéma général de la rétro-ingénierie et elle se base sur deux étapes principales : la collection des traces et la construction des modèles. Pour la collection des traces, nous n'avons pas construit un nouvel outil de collection de trace, car il existe énormément d'outils efficaces dans la littérature correspondant aux plusieurs langages et plateformes [38, 52, 46, 50, 47, 48, 49, 51]. Selon l'environnement du système ciblé nous pouvons choisir l'outil de traçage adéquat, nous réagissons par une petite adaptation dans la partie « collection de données ». De cette façon nous assurons une indépendance vis-à-vis les langages et les plateformes.

Pour la construction des diagrammes de séquence, nous avons proposé trois algorithmes principaux permettant de détecter les trois opérateurs d'interaction principaux d'UML2 à savoir « loop », « opt » et « alt ». Ces algorithmes nous ont permet de construire un diagramme de séquence global reprenant les comportements de toutes les traces considérées. Enfin, nous avons utilisé l'API UML2 comme un moyen pour construire des diagrammes de séquence qui peuvent être visualisé et manipulés par les outils UML2.

2. Discussion des résultats

L'approche que nous avons proposée a quelques avantages par rapports aux travaux existants que nous pouvons les résumer dans les points suivants :

- Notre solution permet de construire un diagramme de séquence complet qui représente le comportement général d'un système, pas seulement un diagramme de séquence qui représente un comportement partiel du système. Cette différence est en particulier engendrée par le fait que nous avons considéré plusieurs traces d'exécution en entrée et pas seulement une.
- La solution proposée est indépendante de langages et plateformes du système étudié (cf. la discussion ci-dessus).

- Assurer la portabilité des diagrammes extraits et la possibilité de les traiter automatiquement.
- Notre approche supporte complètement le standard UML2 et ses résultats peuvent être exploités par les outils UML2.

Cependant, notre approche à l'état actuel a quelques inconvénients :

- Nous n'avons pas encore supporté les applications avec imbrications.
- Il est aussi difficile d'avoir en entrée des traces qui représentent tous les scénarios possibles des applications étudiés.

3. Perspectives

Comme nous l'avons souligné ci-dessus, l'inconvénient majeur de notre approche concerne le non support des blocs imbriqués. Le problème principal réside dans la façon de délimiter les blocs dans les différentes traces. Nous souhaitons traiter cette limite dans un travail futur.

Il nous reste aussi d'entamer la détection des fragments combinés avec les opérateurs break, par, etc. et de faire supporter le reste des notations du diagramme de séquence d'UML tels que : "gates", "continuants", "interactionUse, etc.

Un autre axe de recherche future est d'extraire d'autres types de diagrammes d'UML tels que les diagrammes d'états et les diagrammes de communication, soit en utilisant des méthodes de transformation ou bien en utilisant d'autres techniques.

Nous pensons que par la méthode hybride on peut atteindre les bons résultats, puisque l'utilisation des deux méthodes statique et dynamique engendre une bonne quantité d'informations, nous planifions de trouver une combinaison adéquate de ces deux sources d'informations pour une meilleure rétro-ingénierie des modèles UML.

Réferences

[1] A. Rountev, O. Volgin and M. Reddoch. Static control flow analysis for reverse engineering of UML SD. *Workshop on Program Analysis for Software Tools and Engineering*. 2005.

[2] P. Tonella and A. Potrich. Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers. In *Proceedings of the International Conference on Software Maintenance*, pages 376–385, Firenze, Italy, 2001. IEEE Computer Society.

[3] E. Chikofsky and J. I. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[4] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Publishing, Inc. ISBN-10: 0-7645-7481-7. 2005.

[5] J. Ka-Yee Ng. Identification of Behavioral and Creational Design Patterns through Dynamic Analysis. *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 34-42, October 2007.

[6] T. Systa. *Static and dynamic reverse engineering techniques For Java Software Systems*. Ph.D Thesis, University of Tampere, Dept. of Computer and Information Sciences, Report A-2000-4, 2000.

[7] R. Oechsle and T. Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). *Software Visualization*, pp. 672-675. 2002.

[8] T. Richner and S. Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. Software Maintenance, 2002. *Proc. International Conference Software Maintenance (ICSM'02)*, pp. 34 – 43, 2002.

[9] W. De Pauw, D. Kimelman, and J. M. Vlissides. Modeling object-oriented program execution. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, 821: 163–182. Springer-Verlag, July 1994.

[10] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang. Visualizing the Execution of Java Programs. *Revised Lectures on Software Visualization, International Seminar*. pp. 151–162, 2001.

[11] L. C. Briand, Y. Labiche, Y. Miao. Towards the Reverse Engineering of UML Sequence Diagrams. *Proceedings of the 10th Working Conference on Reverse Engineering*. 2003.

[12] L. C. Briand, Y. Labiche, J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *Software Engineering, IEEE Transactions on*, 32(9): 642-663, Sept. 2006.

[13] Y.-G. Guéhéneuc, T. Ziadi. Automated Reverse-engineering of UML v2.0 Dynamic Models. *Proc. ECOOP Workshop Object-Oriented Reengineering*, 2005.

[14] G. Antoniol, M. Di Penta and M. Zazzara. Understanding Web Applications through Dynamic Analysis. *Program Comprehension, 2004. Proceeding 12th IEEE International Workshop*.

[15] R. Koschke. What architects should know about reverse engineering and Reengineering. Software Architecture, 2005. WICSA 2005. *5th Working IEEE/IFIP Conference*.

[16] L. Wendehals. Selective Tracer for Java Programs. *Proceedings of the 2nd International Fujaba Days 2004*, Darmstadt, Germany. September 2004.

[17] D. Cooper, B. Khoo, B. R. von Konsky, M. Robey. Java Implementation Verification Using Reverse Engineering. *Proceedings of the 27th Australasian conference on Computer science*, pp. 203-211, January 01, 2004, Dunedin, New Zealand.

[18] T. Ball. The Concept of Dynamic Analysis. *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, Toulouse, France. 1999.
[19] M. Van Hilst and E. B. Fernandez. Reverse Engineering to Detect Security Patterns in Code. *Proccedings the 1st International Workshop on Software Patterns and Quality (SPAQu'07), co-located with 14th Asia-Pacific Software Engineering Conference (APSEC'07)*, Nagoya, Japan, December 3, 2007.
[20] A. Hamou-Lhadj. The Concept of Trace Summarization. *Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis co-located with IEEE WCRE*, Pittsburg, USA, 2005.
[21] A. Kuhn, O. Greevy and T. Gîrba. Applying Semantic Analysis to Feature Execution Traces. *Proceedings IEEE Workshop on Program Comprehension through Dynamic Analysis (PCODA 2005). IEEE Computer Society Press*: Los Alamitos CA, 2005.
[22] H. Lee, C. Yoo. A Form Driven Object-Oriented Reverse Engineering Methodology. *Information Systems*, 25 (3): 235–259, May 2000.
[23] A. Hamou-Lhadj, T. C. Lethbridge. A Survey of Trace Exploration Tools and Techniques. *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, p.42-55, October 04-07, 2004, Markham, Ontario, Canada.
[24] E. Stroulia, T. Systa. Dynamic Analysis for Reverse Engineering and Program Understanding. *ACM SIGAPP Applied Computing Review*, 10(1): 8–17, 2002.
[25] G. Canfora and M. Di Penta. New Frontiers of Reverse Engineering. *International Conference on Software Engineering, 2007 Future of Software Engineering*, pp. 326-341, May 2007.
[26] L. M. Duarte. *Behaviour Model Extraction using Context Information*. Ph.D. Thesis, Department of Computing, Imperial College London, December 2007.
[27] L. Audibert. *UML2*. Éd. 2007-2008.
[28] OMG. *Unified Modeling Language (OMG UML), Superstructure*. V2.1.2. November 2007.
[29] OMG. *Unified Modeling Language (OMG UML), Infrastructure*. V2.1.2. November 2007.
[30] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing Dynamic Software System Information through High-level Models. *ACM SIGPLAN Notices*, 33(10): 271–283, October 1998.
[31] N. Shi and A. Ronald. Reverse Engineering of Design Patterns for High Performance Computing. In *Proceedings of the 2005 Workshop on Patterns in High Performance Computing*, 2005.
[32] L. Angyal, L. Lengyel, H. Charaf. An Overview of the State-of-The-Art Reverse Engineering Techniques. *Proceedings of 7th International Symposium of Hungarian Researchers on Computational Intelligence*, 2006.
[33] ILogix. Rhapsody, the Rhapsody case tool. http://www.ilogix.com.
[34] Rational. Rose, the Rational Rose case tool. http://www.rational.com.
[35] Object International. TogetherJ, the TogetherJ case tool. http://www.togethersoft.com.
[36] T. Ziadi. *Manipulation de Lignes de Produits en UML*. Ph.D. Thesis, Université de Rennes 1, 2004.
[37] B. Cornelissen, A. van Deursen, L. Moonen, et A. Zaidman. Visualizing Testsuites to Aid in Software Understanding. *Proceedings 11th Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Amsterdam, Netherlands, 2007; 213–222.
[38] Y.-G. Gueheneuc, R. Douence, N. Jussien. No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs. *Proceedings of the IEEE 17th international Conference on Automated Software Engineering*, Sept., 2002. Page, 117.

[39] J. W. Nimmer et M. D. Ernst. Automatic Generation of Program Specifications. *Intl Symp. on Software Testing and Analysis*, pp. 232–242, Rome, Italy, July 2002.
[40] J. Niere, M. Meyer, and L. Wendehals. *User-driven adaption in rule-based pattern recognition*. Technical report, tr-ri-04-249, University of Paderborn, Paderborn, Germany, June 2004.
[41] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Trans. Software Eng*., 29(9):782–795, 2003.
[42] J. I. Maletic, A. Marcus, and L. Feng. Source Viewer 3D (sv3D) - a framework for software visualization. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, May 3-10, 2003, Portland, Oregon, USA, pp. 812–813, 2003.
[43] K. Wong, S. Tilley, H. A. Muller, and M. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1): 46–54, Jan. 1995.
[44] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of the Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 342–357, 1995.
[45] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Swanson, J. Isaak. Visualizing Dynamic Software System Information through High-level Models. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 271–283, British Columbia, Canada, 1998.
[46] University of Oregon, TAU: Tuning and Analysis Utilities, http://www.cs.uoregon.edu/research/paracomp/tau/, 1999.
[47] A. Chawla and A. Orso. A Generic Instrumentation Framework for Collecting Dynamic Information. *SIGSOFT Software Engineering Notes, Section: Workshop on Empirical Research in Software Testing*. *ACM Press*, 29(5):1–4, September 2004.
[48] Sun Microsystems. Java Platform Debugger Architecture(JPDA). Online at http://java.sun.com/products/jpda/index.jsp. Last visited: September 2005.
[49] B. Lewis. Recording Events to Analyze Programs. In *Object-Oriented Technology. ECOOP 2003 Workshop Reader. Lecture notes on computer science (LNCS 3013)*, Springer, July 2003.
[50] L. Wendehals. Tool Demonstration: Selective Tracer for Java Programs. In *Proc. of the 12th Working Conference on Reverse Engineering, Pittsburgh*, Pennsylvania, USA, November 2005.
[51] A. Vasconcelos, R. Cepêda, C. Werner. An Approach to Program Comprehension through Reverse Engineering of Complementary Software Views. *Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis (PCODA'05)* 2005.
[52] F. Fleurey. JTracor. Available at : http://franck.fleurey.free.fr/JTracor/index.htm
[53] M. Alanen and I. Porres. Model Interchange Using OMG Standards. Software Engineering and Advanced Applications, 2005. *31st EuroMicro Conference* on 30 Aug.-3 Sept. 2005.
[54] N. Walkinshaw, K. Bogdanov, and M. Holcombe. Identifying state transitions and their functions in source code. In *Testing: Academic and Industrial Conference (TAIC PART'06)*, pages 49–58. IEEE Computer Society, 2006.
[55] L. M. Duarte, J. Kramer, and S. Uchitel. Model extraction using context information. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of Lecture Notes in Computer Science, pages 380–394. Springer, 2006.

[56] J. Whaley, M. Martin, and M. Lam. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.
[57] A. W. Biermann and J. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computers*, 21:592–597, 1972.
[58] D. Pierre, L. Bernard, D. Christophe, V. L. Axel. The QSM Algorithm and its Application to Software Behavior Model Induction. *Applied Artificial Intelligence*, 22 (1):77-115, 2008.
[59] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pp.209–218, Washington, DC, USA, 2007. IEEE Computer Society.
[60] T. Ziadi, L. Helouet, and J.-M. Jezequel. Revisiting statechart synthesis with an algebraic approach. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pp. 242–251, Washington, DC, USA, 2004. IEEE Computer Society.
[61] H. Yuan, T. Xie, and E. Martin. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pp. 835–838, New York, NY, USA, 2006. ACM.
[62] D. Harel. Statecharts, A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, 1987.
[63] http://www.uml.org/#Links-Tools, 2010.

Annexes

(1) Code de détection des fragments combinés de type "loop"

```
static void cf_construct(Connection Contact) throws SQLException{
      Statement Requet=Contact.createStatement();
      ResultSet re=Requet.executeQuery("select max(tr_file) from Mess_tr
;");
      int tr_file=1;
      if (re.next())
            tr_file= re.getInt(1);
      re.close();
      for (int tr=1;tr<=tr_file;tr++){
            Statement Requete1=Contact.createStatement();
            ResultSet res= Requete1.executeQuery("select cde_mess, ord_mess
from Mess_tr where tr_file="+tr+" order by ord_mess;");
            Statement Requete2=Contact.createStatement();
            ResultSet res2= Requete2.executeQuery("select count(*)from
Mess_tr where tr_file="+tr+";");
            if (res2.next()){
                  if (res.next()){
                        int taille= (int) res2.getLong(1);
                        int []tab = new int [taille+2];
                        System.out.println("taille = : "+taille+" tr
="+tr);
                        int j=1;
                        //Copy cde_mess into an array:
                        for (j=0;j<taille;j++){
                              tab[j]=res.getInt(1);
                              System.out.println("ok : "+tab[j]);
                              res.next();
                        }
                        // find int_frag:
                        int suit_loop=0;
                        int L=1,cf=0;
                        int h=0;
                        for (j=1;j<taille;j++){                    //loop
until the end of messages sequence
                              if (tab[j]!=tab[j-L]){               //detect
the eventual fragment begin
                                    L++;

                                    if (j==taille-1){
                                          L=1;
                                          h++;
                                          j=h;

                                          }
                              }
                              else {
      //Begin detected
                                          suit_loop=0;
                                          String intf="";
                                          int m=0;
                                          int c=0;
                                          int f=0;
                                          for (m=j+1;m<=j+L;m++){ //loop to
check the rest of fragment
```

```
                                                if (m==j+L){
      //this is a loop fragment

                                                if (suit_loop==0){
                                                for (int n1=j; n1<m; n1++) {
                                                      intf+= tab[n1];
                                                }
                                                cf++;
                                                }
                                                else{
                                                System.out.println(" loop
suit: ");
                                                }
                                                if (tab[m]==tab[j]){
      //detect the eventual suit of loops
                                                      j=m;
                                                      suit_loop=1;
                                                }
                                                else
                                                c=1;

                                                }
                                                else
                                                if (tab[m]!=tab[m-L]){  //
no fragment detected: continue
                                                      f=j;
                                                m=f+L+2;
                                          }
                              }
                              if (c==1){  //move to the rest after the currant fragment
                                    L=1;
                                    j=m-1;
                                    h=m-1;
                                    System.out.println("Parsed messages number: "+j);
                              }
                        }
      }
      }
      }
      }
}
```

(2) Code source de l'application "Vente"

```
public class Vendor {

      /**
       * @param args
       */
      public static void main(String[] args) {
            int nbr_article=4;
            boolean isInvoice=false;
            boolean isPayslip=true;
            Sale sale=new Sale();
            sale.newSale(nbr_article, isInvoice,isPayslip);

      }
```

```
	public static void signInvoice(){
		System.out.println("Invoice signed");
	}
	public static void signDelivery(){
		System.out.println("Delivery signed");
	}
	public static void signPayslip(){
		System.out.println("Payslip signed");
	}
}
public class Sale {
	public void newSale(int nbr_article, boolean isInvoice, boolean
isPayslip){
	float oldValue=0;
	System.out.println("New sale created");
	for(int i=1;i<=nbr_article;i++){
		float newValue=addArticle();
		Calcul calcul= new Calcul();
		oldValue=calcul.calculAmount(newValue, oldValue);
	}
	if(isInvoice){
		Invoice invoice= new Invoice();
		invoice.getInvoice();
	}
	else {
		Delivery delivery= new Delivery();
		delivery.getDelivery();
	}
	if(isPayslip){
		Payslip payslip=new Payslip();
		payslip.getPayslip();
	}
}
	public static float addArticle(){
	System.out.println("New article added");
	return 1000;
}
}
public class Calcul {
	//
	/*
	*This method add new amount to the old amount
	*
	*/
	public float calculAmount(float newValue, float oldValue){
		System.out.println("Amount calculated");
		float amount=newValue+oldValue;
		return amount;
	}

}
public class Delivery {
	public void getDelivery(){
		System.out.println("Delivery printed");
		Vendor.signDelivery();
	}
}
public class Invoice {
	public void getInvoice(){
	System.out.println("Invoice printed");
```

```
		Vendor.signInvoice();
}
}
public class Payslip {
		public void getPayslip(){
			System.out.println("Payslip printed");
			Vendor.signPayslip();
		}
}
```

ABSTRACT
Reverse Engineering of UML 2 Dynamic Models

Many legacy systems are developed without providing the necessary documentation for future maintenance and evolution. In such cases; maintainers and developers will spend more time and will make a great effort (more cost) in system structure and behavior understanding phase. Despite the great amount of system comprehension step; results are error-prone even if maintainers are themselves the inner developers. The reverse engineering aims to alleviate developers and maintainers in their work by restraining cost and increasing system comprehension precision; and to present results in a human understandable manner. This study proposes a new approach for UML dynamic models reverse engineering; in particular sequence diagrams; by concentrating on the new notions introduced in the resent versions of UML such as combined fragments. We have followed the dynamic method to obtain information on the system. Our approach is composed of two main parts, which are: 1) traces collection and 2) sequence diagram construction. We have presented an overview of the reverse engineering and related works; as well as; our approach implementation details are explained. Furthermore; we have been successful in testing our solution on a modest application, which encourages us to expect new achievements in the future.

Keywords: reverse engineering, UML sequence diagram, behavioral models, static analysis, dynamic analysis.

RÉSUMÉ
Rétro-ingénierie des modèles comportementaux d'UML 2

De nombreux anciens systèmes sont développés sans fournir la documentation nécessaire pour leur maintenance et leur évolution futures. Dans tels cas, les mainteneurs et les développeurs passent plus de temps et fournissent un grand effort (coût en plus) dans la phase de compréhension du comportement et de la structure du système. Quelle que soit le coût de l'étape de compréhension du système, les résultats peuvent être erronés, même si les mainteneurs sont eux-mêmes les premiers développeurs. La rétro-ingénierie vise à faciliter le travail des développeurs et des mainteneurs par la réduction du coût d'une part et d'accroître la précision de compréhension du système et de présenter les résultats d'une manière compréhensible d'une autre part. Dans cette étude, nous avons proposé une nouvelle approche pour la rétro-ingénierie des modèles comportementaux d'UML, en particulier les diagrammes de séquence, en se concentrant sur les nouvelles notions introduites dans les versions récentes d'UML, tels que les fragments combinés. Nous avons suivi la méthode dynamique pour obtenir des informations sur le système. Notre approche se compose de deux parties principales qui sont : 1) la collection de traces et 2) la construction du diagramme de séquence. Nous avons présenté une vue d'ensemble sur le domaine de la rétro-ingénierie et les travaux existants, ainsi, les détails d'implémentation de notre approche sont expliqués. En outre, nous avons réussi le test de notre solution sur une application modeste, ce qui nous pousse à réaliser de nouvelles recherches dans ce domaine dans le futur.

Mots clés : rétro-ingénierie, diagramme de séquence d'UML, modèles comportementaux, analyse statique, analyse dynamique.