Université de Montréal

Dérivation de diagrammes de séquence UML compactes à partir de traces d'exécution en se basant des heuristiques

par
Houssem Aloulou

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

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

Juillet, 2015

RÉSUMÉ

Nous proposons une approche d'extraction des diagrammes de séquence à partir de programmes orientés objets en combinant l'analyse statique et dynamique. Notre objectif est d'extraire des diagrammes compacts mais contenant le plus d'informations possible pour faciliter la compréhension du comportement d'un programme. Pour cette finalité, nous avons défini un ensemble d'heuristiques pour filtrer les événements d'exécution les moins importants et extraire les structures de contrôles comme les boucles et la récursivité. Nous groupons aussi les objets en nous basant sur leurs types respectifs. Pour tenir compte des variations d'un même scénario, notre approche utilise plusieurs traces d'exécution et les aligne pour couvrir le plus possible le comportement du programme. Notre approche a été évaluée sur un système de simulation d'ATM. L'étude de cas montre que notre approche produit des diagrammes de séquence concis et informatifs.

Mots clés: Compréhension de programme, retro-ingénierie, diagrammes de séquences UML, traces d'exécutions, réduction de traces, alignement de trace.

ABSTRACT

We propose an approach for extracting sequence diagrams from object-oriented programs by combining static and dynamic analysis. Our objective is to derive compact yet informative diagrams to ease the understanding of a program's behavior. To this end, we define a set of heuristics to filter unimportant execution events and abstract control structures such as loops and recursions. We also group objects based on their respective types. To cover variations of the same behavior, our approach considers and aligns multiple execution traces. Our approach was evaluated on an ATM Banking simulation system. The case study shows that our approach produces concise, yet informative sequence diagrams.

Keywords: Program comprehension, Reverse Engineering, UML Sequence Diagram, Execution Traces, Trace reducing, Traces Alignment.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

À mes chers parents Youssef & Rafia

À mes chers Yasmine & Omar

REMERCIEMENTS

Je ne peux pas laisser passer l'occasion de la présentation de ce rapport sans exprimer mes remerciements et ma gratitude à tous ceux qui ont bien voulu apporter l'assistance nécessaire au bon déroulement de ce projet.

J'adresse mes plus vifs remerciements à mon encadrant, Houari Sahraoui, pour l'attention et l'intérêt qu'il a porté à mon travail et pour ses multiples conseils et orientations tout au long de l'élaboration de ce projet.

Je tiens aussi à témoigner ma reconnaissance envers tous mes collègues de travail du GEODES pour toutes les discussions autour d'un café et d'une beigne.

Je remercie aussi le gouvernement de la Tunisie et CRSNG pour leur support financier.

CHAPITRE 1

INTRODUCTION

1.1 Contexte

La compréhension du comportement d'une grande application informatique est une tâche cruciale avant de commencer sa maintenance ou sa mise à jour. La consultation de la documentation de conception d'un logiciel permet au développeur de comprendre rapidement son fonctionnement et son comportement. Néanmoins, plusieurs raisons font que les changements apportés au code, au cours du cycle de vie du système, ne sont pas traduits dans la documentation de la conception de l'application. Cela est généralement dû à la complexité de la tâche de mise à jour de la documentation de conception et la nécessité de terminer la maintenance d'un programme dans les plus brefs délais. Ainsi, l'écart entre l'application et sa documentation ne cesse de s'accroître avec le temps et après chaque opération de mise à jour ou de maintenance. En conséquence, la documentation de conception du logiciel ne reflète plus convenablement le comportement du système après un certain nombre d'opérations de maintenance. Cette situation rend impossible à un développeur de comprendre le comportement de l'application.

La rétro-ingénierie des diagrammes dynamiques, comme le diagramme de séquence, est une solution de choix qui permet de reproduire automatiquement la documentation de conception d'un logiciel pour faciliter sa compréhension et sa maintenance [8]. Nous utiliserons la rétro-ingénierie afin d'extraire une information concise et précise à partir des milliers, voire des millions, d'événements générés durant l'exécution de l'application.

1.2 Problématique

Pour modifier le code, le programmeur doit à priori comprendre le comportement du système à partir de la documentation. Si cette dernière n'est pas à jour, alors le programmeur se trouverait face à un grand problème. Selon plusieurs études, jusqu'à 60%

de l'effort et du temps des travaux de maintenance et d'entretien sont consacrés à la compréhension du logiciel [12, 13]. Pour cette raison, il est essentiel d'exploiter des techniques et des outils qui facilitent la tâche de compréhension des logiciels.

Une des approches connues pour leur efficacité dans de tels cas est la rétro-ingénierie. Dans cette perspective, plusieurs travaux de recherche ont été proposés. La plupart des travaux se sont basés sur l'analyse statique des applications. L'analyse statique avec les contributions récentes, comme *l'exécution symbolique* [20], ont une bonne précision dans la capture du comportement d'un programme. En effet, l'analyse statique permet la description de toutes les exécutions possibles du programme sans pour autant l'exécuter et s'occupe de l'identification des propriétés structurelles d'une application[52].

Néanmoins, l'utilisation extensive des caractéristiques dynamiques par les applications orientées objet modernes, tels que le polymorphisme et le chargement dynamique de classes, peut compromettre les points forts de l'analyse statique [53]. Contrairement à cette dernière, l'analyse dynamique permet de repérer facilement ces caractéristiques. En effet, l'analyse dynamique suit le système lors de l'exécution d'un scénario. C'est l'analyse des propriétés d'un programme en exécution [1]. L'analyse dynamique permet de dériver des propriétés qui sont valables pour une ou plusieurs exécutions en examinant le comportement d'un programme. Elle permet par exemple de détecter les violations de propriétés ainsi que de fournir aux programmeurs des informations utiles sur le comportement du programme.

Cependant, l'analyse dynamique présente deux problèmes majeurs. D'une part, une trace d'exécution, obtenue par analyse dynamique, ne reflète qu'une exécution spécifique d'un scénario. Ainsi, afin de comprendre tous les comportements possibles du système, il est nécessaire d'obtenir un ensemble de traces d'exécution représentatives pour un cas d'utilisation spécifique.

D'autre part, les traces d'exécution contiennent un grand nombre d'événements pouvant aller de quelques milliers à quelques millions d'événements pour les grands systèmes informatiques. La grande majorité des événements produits ne sont pas pertinents pour une compréhension de haut niveau du comportement d'un programme. Nous avons constaté que le "mapping" direct de ces traces d'exécution en diagramme de séquence

produit d'énormes diagrammes. Il sera alors impossible de lire et comprendre un très grand diagramme de séquence écrit sur un support dont la surface est égale à celle d'un mur d'une chambre...

Il sera alors intéressant de trouver une méthode qui permet d'élaguer les événements qui n'ont pas d'importance dans la compréhension du comportement du programme. Aussi, il sera d'une grande utilité de trouver les événements "identiques" qui se répètent dans une trace d'exécution. À l'issue de ces simplifications il nous sera possible de reconstruire des diagrammes d'UML, comme le diagramme de séquence, afin de faciliter la compréhension du comportement du programme.

1.3 Objectifs

Typiquement, chaque trace d'exécution est le reflet d'une exécution spécifique d'un scénario qui ne donne qu'une vue partielle du comportement du système. Les données primitives contenues dans les traces d'exécution ne peuvent pas être facilement traitées et comprises vu le grand nombre de traces et la quantité énorme de données dans chacune d'elles.

Dans le cadre de ce travail de recherche, nous proposons une approche pour l'abstraction et la compréhension du comportement des programmes orientés objets. Nous combinons l'exploitation de l'analyse dynamique et l'analyse statique.

Notre objectif étant la rétro-ingénierie de diagrammes dynamiques et de comportement d'UML pour brosser un portrait le plus complet possible du comportement du système. Nous abordons le problème d'explosion de la taille des traces d'exécution en définissant un ensemble d'heuristiques pour éliminer les événements peu ou pas importants et en dérivant les structures de contrôle complexes. La préoccupation de réduction concerne aussi le nombre d'objets impliqués dans les exécutions en raisonnant sur leurs types. En outre, nous contournons le problème de généralisation du comportement par l'examen et l'alignement de nombreuses traces d'exécution pour avoir une seule trace d'exécution résultante qui regroupe le comportement de plusieurs traces initiales.

L'implantation de notre approche permettra de faciliter la maintenance et la re-documentation

des applications dont la documentation disponible ne reflète plus la réalité.

Notre approche se déroule sur quatre phases. La première phase consiste à définir un ensemble d'exécutions qui permettent d'observer le comportement que nous voulons documenter afin d'obtenir une vue la plus complète possible du comportement du système. Au cours de la deuxième phase, nous collectons plusieurs traces d'exécution pour chaque scénario. Les traces d'exécution générées pour un même scénario doivent être combinées et réduites. C'est l'objectif de la troisième phase de notre approche qui comporte deux sous-phases pouvant être interverties. Pour la première sous-phase, nous proposons un algorithme pour l'alignement de plusieurs traces d'exécution appartenant à un même scénario. Pour la deuxième sous-phase, nous proposons un deuxième algorithme basé sur des heuristiques pour trouver les possibilités de réduction des traces d'exécution. La quatrième phase consiste à générer un diagramme de séquence d'UML à partir de la trace d'exécution résultante.

Notre approche a été évaluée sur le système de simulation d'un ATM [9]. Après la sélection des cas d'utilisation avec les différents scénarios, nous générons les traces d'exécution sur lesquelles nous appliquons les stratégies de réduction et d'alignement. L'étude de cas a montré que notre approche produit des diagrammes de séquence concis, mais informatifs.

Ce mémoire est organisé comme suit. Dans le deuxième chapitre, nous présentons un état de l'art sur les travaux réalisés dans le même contexte de ce présent travail. Ensuite, nous détaillons notre approche dans le troisième chapitre. Dans la conclusion générale, un résumé de notre travail et les améliorations possibles sur notre approche sont présentées.

CHAPITRE 2

ÉTAT DE L'ART

2.1 Introduction

D'une façon générale, la rétro-ingénierie dans le domaine du génie logiciel est le processus d'analyse d'un système pour 1) identifier ses composants et les relations entre eux et 2) créer des représentations du système sous d'autres formes dans un plus haut niveau d'abstraction pour comprendre son fonctionnement [11]. Le terme "*compréhension d'un programme*" a été défini par Biggerstaff et al. dans [2] comme suit : "*Une personne comprend un programme lorsqu'il est en mesure d'expliquer le programme, sa structure, son comportement, ses effets sur le fonctionnement de son contexte et ses relations avec son domaine d'application*". Dans ce chapitre, nous commençerons dans la première ppartie par mettre l'accent sur la terminologie, les méthodes et les théories qui seront exploitées dans notre travail de recherche. Nous représenterons ensuite un état d'art sur les principales approches et algorithmes qui ont été proposés pour la réduction et l'alignement des traces d'exécution.

Ainsi, la première partie de ce chapitre sera consacrée pour la présentation des diagrammes dynamiques et de comportements d'UML. Ensuite, nous comparons l'analyse statique et l'analyse dynamique pour motiver le choix de cette dernière. Puis, nous décrivons les principales caractéristiques d'une trace d'exécution et les outils permettant de l'extraire. Par la suite, nous exposerons les principales caractéristiques de la rétro-ingénierie et de l'alignement des traces d'exécution.

2.2 Diagrammes dynamiques et de comportement d'UML

Le langage de modélisation unifié (Unified Modeling Language, UML [34]) est la spécification de l'OMG[1] (Object Management Group [22]) la plus utilisée.

[1] L'OMG est un organisme international, ouvert, à but non lucratif, créé depuis 1989. L'objectif de son apparition est de standardiser les modèles objet. L'OMG est à la base des standards UML, CORBA, MDA, etc.

UML est un langage graphique de modélisation orienté objet permettant de spécifier, construire, visualiser et documenter les composantes d'un système logiciel [34]. UML est devenu un standard *de facto*. La deuxième version (UML 2.0) constitue une évolution importante. En effet, la dernière version stable[2] d'UML diffusée par l'OMG depuis mai 2010 contient 13 diagrammes qui se complètent en vue de faciliter la modélisation d'un projet informatique au cours de son cycle de développement et durant tout son cycle de vie.

Nous relevons deux groupes principaux de diagrammes : d'une part nous trouvons les diagrammes statiques (appelés aussi structurels), qui montrent une vue statique du logiciel, et d'autre part, il y a les diagrammes dynamiques et de comportements, qui montrent une vue dynamique du logiciel [19].

Les diagrammes statiques d'UML sont utilisés en phase d'analyse qui précède la phase d'implémentation. Ils permettent de montrer la structure statique des éléments existants (par exemple les classes, les types, etc.) soit leur structure interne et leurs relations. Les diagrammes dynamiques et de comportements d'UML présentent quant à eux le caractère temporel de l'information, soit son aspect dynamique.

Par conséquent, il ne sera pas possible de comprendre et de suivre le comportement du système au cours de son exécution avec les diagrammes statiques d'UML . Pour cela, nous nous intéressons, dans ce qui suit, à la rétro-ingénierie des diagrammes dynamiques et de comportements d'UML.

2.2.1 Diagrammes dynamiques

Les diagrammes dynamiques représentent une vision microscopique du fonctionnement du système. Ils servent à mettre en évidence les relations temporelles inter-objets. Les diagrammes dynamiques d'UML permettent de décrire le système de plusieurs angles et ainsi accroître la sémantique qu'un graphe peut déceler.

L'utilisation des diagrammes dynamiques permet de décrire les interactions entre les objets, spécifier les états et leurs changements et préciser le déroulement des actions de contrôle et des événements [34].

[2] Apparition d'une version Beta UML 2.4 depuis mars 2011

Les diagrammes dynamiques sont : le diagramme de séquence, le diagramme d'interaction, le diagramme de communication et le diagramme de temps. Dans ce qui suit, nous présenterons succinctement ces diagrammes en donnant plus d'importance à la description du diagramme de séquence, car c'est le diagramme qui nous intéresse dans ce travail de recherche.

- Diagramme de séquence : il est considéré comme un des diagrammes d'UML les plus importants car il permet de comprendre le fonctionnement du logiciel. Il montre, selon un ordre chronologique, d'une part les interactions entre les objets et l'extérieur du système au cours d'un scénario bien déterminé et d'autre part les interactions entre les objets à l'intérieur du système. Dans les premières phases du cycle de développement, un diagramme de séquence est exploité pour illustrer des cas d'utilisation et offre en plus un excellent moyen pour communiquer les aspects dynamiques d'un système.

  La figure 2.1 représente un exemple d'un diagramme de séquence permettant de calculer le prix d'achat d'un produit dans une ligne d'une commande.

  Dans ce diagramme de séquence les objets sont représentés par des rectangles comme par exemple *Command :Com* et *Command Line :liCom*. Chaque objet possède une ligne de vie et communique avec les autres objets par des messages. Par exemple, l'objet *Command :Com* envoi le message *getQuantity* vers l'objet *Command Line :liCom* qui lui répond par un message de retour pointillé contenant la réponse à sa requête. Les messages dans le diagramme de séquence sont présentés dans un ordre chronologique allant du haut vers le bas. Ainsi, le temps y est représenté explicitement par la dimension verticale du diagramme.

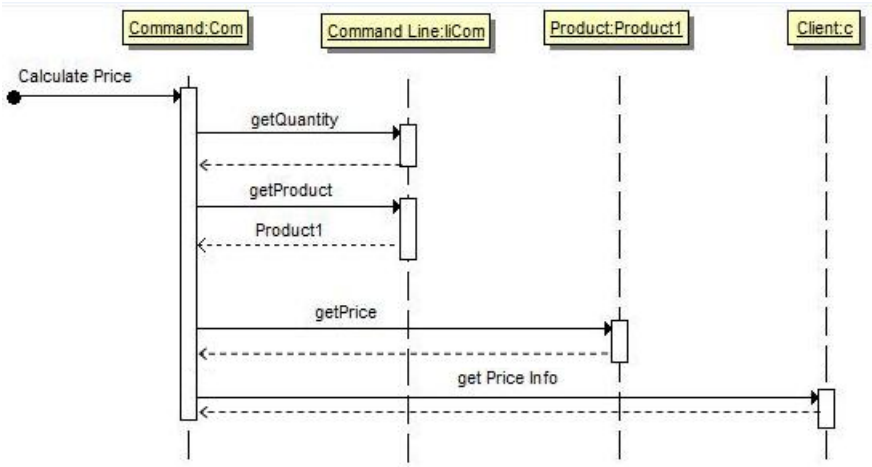


Figure 2.1 – Exemple d'un diagramme de séquence

- Diagramme d'interaction (diagramme d'interactivité) : il permet de donner une vue d'ensemble du flux de contrôle en donnant une représentation de l'organisation spatiale des participants de l'interaction. Ce diagramme permet le déclenchement d'un événement en fonction des états du système et de modéliser les comportements parallélisables.

- Diagramme de communication : appelé aussi diagramme de collaboration, il montre des instances de classes, leurs relations ainsi que l'échange de messages entre elles. Ce diagramme se concentre sur l'organisation structurelle des objets expéditeurs et destinataires de messages.

- Diagramme de temps : il montre le changement de l'état ou de la condition d'une instance d'un classificateur ou d'un rôle dans le temps. Le diagramme de temps est utilisé pour explorer le comportement des objets d'un système durant une période.

2.2.2 Diagrammes de comportement (Behavioral Diagrams)

Les diagrammes de comportement servent à mettre en évidence la représentation du comportement de chaque objet sous forme d'un automate [34]. En effet, ces diagrammes permettent de décrire le comportement d'un élément de modélisation, par exemple un objet, une interaction (cf. diagramme de collaborations), un cas d'utilisation, un acteur,

un sous-système, une opération ou une méthode. Les diagrammes de comportement sont les suivants :

- Diagramme de cas d'utilisation : il montre des cas d'utilisation, des acteurs ainsi que leurs relations pour donner une vision globale du comportement fonctionnel d'un logiciel. Un cas d'utilisation représente l'interdépendance entre un utilisateur, humain ou machine, et un système. Dans un tel diagramme, les utilisateurs sont appelés acteurs (actors), ils interagissent avec les cas d'utilisation (use cases).

- Diagramme d'activité : il sert principalement à modéliser des processus métiers ou à décrire des opérations complexes, en incluant des flots de données. Il présente une vision macroscopique et temporelle du système modélisé sous une forme proche de l'organigramme. Il permet de modéliser un processus interactif pour un système donné et d'exprimer une dimension temporelle sur une partie du modèle, à partir de diagrammes de classes ou de cas d'utilisation.

- Diagramme d'états-transitions : appelé aussi diagramme de machines à états, il décrit les états d'un objet, aussi bien que les transitions entre les états. Le diagramme d'états-transitions décrit le comportement interne d'un seul objet à l'aide d'un automate à états finis. Il présente les séquences possibles d'états et d'actions qu'une instance de classe peut subir/traiter au cours de son cycle de vie en réaction à des événements discrets. Il offre une vision complète et non ambiguë de l'ensemble des comportements de l'élément auquel il est attaché.

2.2.3 Améliorations de UML2.0

Nous visons à appliquer quelques unes des nouveautés de la version UML 2.0 qui a donné plusieurs solutions aux limites des versions antérieures telles que la quantité d'information dans un même diagramme, la quantité de diagrammes à réaliser pour visualiser le comportement global du système et la difficulté de la mise à jour lors d'un changement au niveau des exigences. Par exemple, pour les diagrammes de séquence, il existe plusieurs constructions qui permettent une meilleure visualisation du comportement du système :

- Fragments combinés (combined fragment) : ils sont utilisés pour grouper un ensemble de messages. Les fragments combinés permettent de visualiser les flux conditionnels dans un diagramme de séquence. Ils permettent d'avoir des diagrammes de séquences compactes.

  Un fragment combiné est défini par *un opérateur d'ineraction* et *des opérandes* :

  Un opérateur d'interaction est représenté par un rectangle dont le coin supérieur gauche indique le type de la combinaison.

  Les opérandes d'un opérateur d'interaction sont séparés par une ligne pointillée. Les conditions de choix des opérandes sont données par des expressions booléennes entre crochets.

  Il existe douze opérateurs définis dans la notation UML 2.0 [49]. Le tableau 2.I présente ces opérateurs d'interaction :

  La figure 2.2 représente un exemple de l'utilisation d'un fragment combiné de type "alt" (alternatives) pour l'encaissement d'un chèque. Le diagramme de séquence contient trois objets *Bank*, *Check* et *CheckingAccount*. le message *getBalance* permet de vérifier la disponibilité du montant à encaisser. Deux cas alors se présentent : Le montant nécessaire est disponible ou insuffisant. C'est avec le fragment combiné *alt* que ces deux conditions sont vérifiées. Dans le premier cas, les messages *addDebitTransaction* et *storePhotoOfCheck* sont envoyés à l'objet *CheckingAccount*. Dans le cas contraire, la transaction ne pourra pas être réalisée, ainsi, c'est seulement les messages dans la partie de *else* qui est exécutée.

- Références (interaction occurrence) : ce sont des pointeurs ou des raccourcis vers un autre diagramme de séquence existant. Ils permettent de factoriser des parties de comportement utilisées dans plusieurs scénarios.

- Continuations (continuation symbol) : ils permettent de définir des branchements. Ils sont utilisés souvent avec des fragments combinés. Dans d'autres langages, ce type de construction est aussi connu sous le nom de "Label".

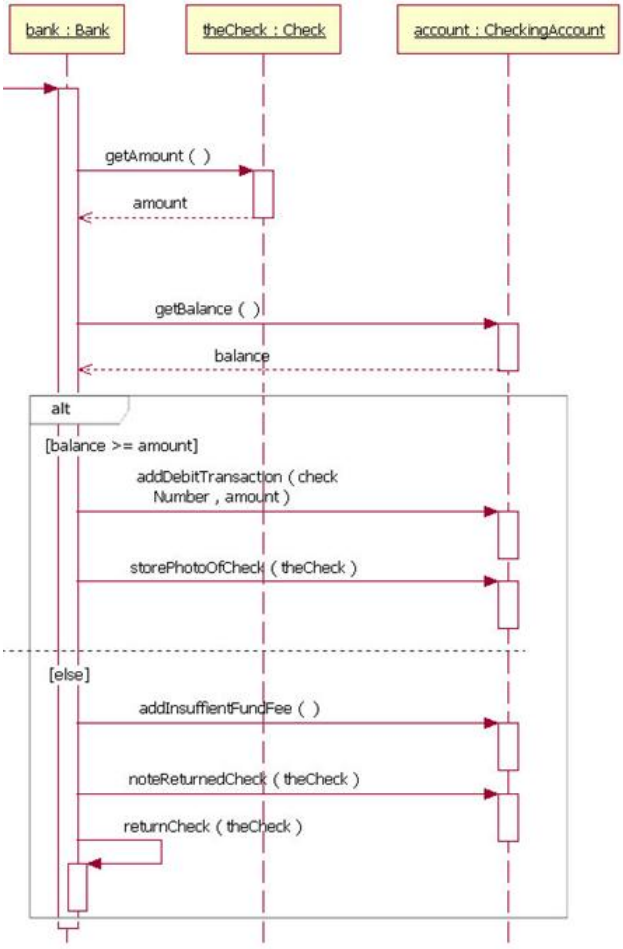


Figure 2.2 – Exemple de l'utilisation du fragment combiné "alt"

Opérateurs par fonction	*Opérateur*	*Description*
opérateur de choix et de boucle	"alt" (alternative)	Représente deux comportements possibles : c'est en quelque sorte l'équivalent du SI...ALORS...SINON
opérateur de choix et de boucle	"opt" (option)	Fragment optionnel : représente un comportement qui peut se produire... ou pas
opérateur de choix et de boucle	"break"	Représente l'interruption du flot "normal" des interactions par des exceptions
opérateur de choix et de boucle	"loop"	Décrit un ensemble d'interactions qui s'exécutent en boucle
opérateurs contrôlant l'envoi en parallèle de messages	"par" (parallel) and "critical" (critical region)	Représente des interactions ayant lieu en parallèle pour "par" et une section critique pour "critical"
opérateurs contrôlant l'envoi de messages	"ignore"	Indique qu'il existe des messages qui ne sont pas présents dans le fragment combiné
opérateurs contrôlant l'envoi de messages	"consider"	Désigne les interactions à prendre en compte dans la séquence
opérateurs contrôlant l'envoi de messages	"assert" (assertion)	Indique que le fragment combiné est une assertion
opérateurs contrôlant l'envoi de messages	"neg" (negative)	désigne un ensemble d'interactions invalides
opérateurs fixant l'ordre d'envoi des messages	"seq" (weak sequencing)	notifie que les interactions qui s'opèrent entre des entités indépendantes n'ont pas d'ordre particulier
opérateurs fixant l'ordre d'envoi des messages	"strict" (strict sequencing)	impose l'ordre décrit sur le diagramme

Tableau 2.I – Opérateurs d'interaction par fonction des fragments combinés

- Décomposition hiérarchique (part decomposition) : ils assurent une description *top-down* du système à réaliser. L'idée est de commencer la présentation par un diagramme de haut niveau (les utilisateurs + le système) pour chaque fonctionnalité et ensuite raffiner chaque diagramme pour décrire les interactions entre sous-systèmes.

2.3 Analyse statique vs dynamique

La distinction entre l'analyse statique et l'analyse dynamique réside dans le moment de l'analyse du programme en question.

2.3.1 Analyse statique

L'analyse statique explore en avance et dans un temps fini un programme sans l'exécuter. Elle permet, entre autres, de repérer des erreurs de programmation ou de conception [53]. L'analyse statique extrait les informations qui seront utiles pour la transformation de programmes ou des preuves d'exactitude du programme.

L'analyse statique d'un programme est sûre et finie. Elle représente aussi un calcul approximatif de la sémantique du code du programme.

- Sûre (sound) : compatible avec la sémantique concrète de l'exécution.
- finie : quel que soit le programme et sa sémantique approchée, l'analyse se termine.

Le plus grand avantage de l'utilisation de l'analyse statique c'est qu'elle augmente la confiance quant à l'exactitude du système en question. Aussi, il ne sera pas trop tard pour corriger les bugs observés lors de l'analyse. Ceci explique le fait que l'analyse statique est la méthode la plus utilisée pour la vérification des systèmes critiques. Néanmoins, ces caractéristiques non pas d'influence sur nos choix puisque nous voulons faire la rétro-ingénierie de systèmes orientés objets, qui tournent déjà, en vue de comprendre leurs comportements et les re-documenter. En plus les programmes modernes utilisent intensivement les caractéristiques des langages orienté objet, difficile de détecter avec l'analyse statique comme le polymorphisme, le chargement et la génération dynamique de classes, la réflexion, etc.

Pour ces raisons, plusieurs équipes de recherche ont adopté des approches basées sur l'analyse dynamique pour l'extraction des traces d'exécution dans les programmes.

2.3.2 Analyse dynamique

L'analyse dynamique est basée sur la collecte des données lors de l'exécution d'un programme [41]. Dans [1], Ball définit l'analyse dynamique comme "*l'analyse des propriétés d'un logiciel en exécution*". L'utilisation de l'analyse dynamique pour la compréhension des programmes est une des activités de recherche qui ont suscité le plus

d'intérêt des chercheurs durant ces dernières années.

L'analyse dynamique est précise car elle ne nécessite pas des abstractions ou des approximations. Elle examine le comportement exact du système au cours de son exécution. Pour avoir de bons résultats lors de l'analyse dynamique, il faut varier les entrées pour avoir une idée la plus complète possible du fonctionnement général du système.

L'exploitation de l'analyse dynamique pour la compréhension de programmes a plusieurs avantages :

- La précision à l'égard du comportement réel de l'application, par exemple, dans le contexte de logiciels orientés objet qui utilisent le polymorphisme avec "liaison tardive" (late binding).

- La définition d'un scénario d'exécution, de telle sorte que seulement la partie du système qui intéresse le programmeur sera analysée, permet de gagner le temps en se basant sur une stratégie axée sur les objectifs.

Néanmoins, l'analyse dynamique présente aussi quelques limites :

- L'incomplétude inhérente à l'analyse dynamique puisque les traces ou les comportements enregistrés représentent seulement une petite fraction de l'application. Pour remédier à cette limite, la solution est de choisir le plus possible de scénarios distincts.

- La difficulté de déterminer quels scénarios exécuter pour déclencher les parties du programme qui intéressent le programmeur.

- Le passage à l'échelle de l'analyse dynamique en raison de la grande quantité de données qui peuvent être introduites par l'analyse dynamique.

Afin de faire face à ces limites, de nombreuses techniques proposent des abstractions ou des heuristiques, permettant de grouper des points du programme ou des points de l'exécution qui partagent certaines propriétés. Dans de tels cas, un compromis doit être fait entre le rappel (est-ce qu'il nous manque des points pertinents du programme ?) et la précision (est-ce que les points du programme que nous avons choisis aident vraiment à la compréhension de ce programme ?).

2.4 Traces d'exécution

La génération des traces d'exécution est réalisée automatiquement lors de l'exécution de l'application afin de préserver des informations à propos de l'exécution. Les traces d'exécution générées sont sauvegardées dans un fichier. Un fichier de trace contient plusieurs lignes dont chacune représente un événement. Par exemple :

- La méthode appelante et la méthode appelée,
- L'objet appelant et l'objet appelé (l'adresse mémoire et le nom de la classe),
- Les structures conditionnelles,
- Les structures itératives ainsi que le type et la condition d'arrêt.

Les traces d'exécution prennent généralement la forme d'un *arbre d'appels dynamiques*. Chaque noeud de cet arbre, à l'exception du noeud racine, représente une seule invocation de la méthode appelée. Chaque arc représente un appel entre une méthode appelante et une méthode appelée. Si une méthode m est appelée n fois, alors il existe n arcs vers m dans l'arbre d'appel dynamique. Aussi, la précision est favorisée sur l'espace de stockage lorsqu'il y a un très grand nombre d'événements, par exemple un appel d'une méthode à l'intérieur d'une boucle.

Il existe différentes techniques pour la génération des traces d'exécution [26]. Une technique répandue est basée sur l'instrumentation du code source. Elle consiste à insérer des instructions dans des endroits appropriés du code original. Par exemple, pour la génération de traces d'exécution, les instructions d'instrumentation sont insérées dans chaque entrée et optionnellement chaque sortie de chaque méthode. D'autres techniques pour collecter les informations de l'exécution d'une application existent comme les outils de contrôle du système, l'instrumentation de la machine virtuelle de Java (JVM) et l'utilisation des débogueurs personnalisés.

La compréhension d'un programme à travers les traces d'exécution générées n'est pas une tâche facile puisque les traces sont trop grandes pour être comprises directement. Une expérimentation menée par Reiss a montré, qu'en moyenne, il y a un gigaoctet de

trace pour chaque deux secondes du code C/C++ exécuté et pour chaque dix secondes du code en Java [41].

2.5 Rétro-ingénierie des diagrammes dynamiques et de comportement d'UML

Dans le contexte du génie logiciel, le terme rétro-ingénierie a été défini par Chikofsky et Cross dans [11] comme *le processus d'analyse d'un système pour identifier ses composants et leurs interrelations et créer des représentations du logiciel dans une autre forme ou dans un niveau d'abstraction plus élevé*. Ainsi, la rétro-ingénierie consiste à transformer un code vers un modèle pour augmenter le niveau d'abstraction afin de mieux comprendre et analyser un logiciel.

La rétro-ingénierie a été traditionnellement vue comme un processus en deux étapes : (1) l'extraction des informations, en analysant les "artefacts" du logiciel pour recueillir des données brutes, et (2) l'abstraction, qui consiste à créer des vues et des documents orientés utilisateur à partir des données brutes.

Plusieurs travaux de recherches ont été proposés pour la rétro-ingénierie des structures statiques (comme par exemple les diagrammes de classes) en se basant sur l'analyse statique des systèmes [32]. Ainsi, de nombreux outils fiables et efficaces de modélisation UML pour les structures statiques existent déjà sur le marché [4, 40] à l'exception de quelques défis se rapportant par exemple aux associations, aux relations d'agrégation et de composition.

Plusieurs travaux de recherche ont ciblé le problème de la rétro-ingénierie des diagrammes dynamiques d'UML. Quelques-uns de ces travaux se sont basés sur l'utilisation de l'analyse statique alors que d'autres ont utilisé l'analyse dynamique.

2.5.1 Retro-ingénierie par analyse statique

Un travail intéressant basé sur l'analyse statique est celui de Rountev et al. [43]. Dans leur travail, ils ont extrait des fragments du diagramme de séquence d'UML à partir des *graphes de flot de contrôle* des méthodes individuelles. Ces graphes sont ensuite décomposés en des sous-graphes en assurant la différenciation entre les fragments. La

troisième phase consiste à construire le diagramme de séquence. Enfin, une phase qui vise à rendre le diagramme de séquence construit plus compréhensible et plus clair par l'application d'une série de transformations.

Dans la même perspective, mais pour un autre type de diagrammes, Kollman et Gogolla [33] ont proposés une technique d'extraction automatique des diagrammes de collaboration, qui sont conceptuellement similaires aux diagrammes de séquence. Dans leur approche, ils ont mappé le code d'un programme vers une instance d'un méta-modèle. Ensuite, ils ont appliqué un ensemble de règles de transformations pour dériver les diagrammes de collaboration. Pour obtenir des diagrammes compacts, l'accent est mis sur une méthode spécifique comme point d'entrée.

De même que dans le travail précédent, Tonella et Potrich [48] ont utilisé l'analyse des flux de contrôle pour dériver les diagrammes d'interaction pour les programmes écrits en langage C++.

Bien que les trois contributions ci-dessus dérivent des diagrammes qui sont génériques pour toutes les exécutions, elles sont limitées pour deux raisons : premièrement, les diagrammes peuvent être trop grands et manquent d'abstraction. Se concentrer sur une seule méthode réduit la taille mais ne permet pas d'extraire le comportement pour plusieurs scénarios d'un même cas d'utilisation. La seconde limitation est reliée aux approximations réalisées pour les caractéristiques dynamiques, comme le polymorphisme et le chargement dynamique, qui sont largement utilisés. De telles approximations peuvent affecter la compréhension du comportement du programme [54].

2.5.2 Rétro-ingénierie par analyse dynamique

Pour faire face aux limites de l'analyse statique, plusieurs chercheurs ont proposé des approches basées sur l'analyse des traces d'exécution pour la compréhension du comportement des programmes.

Taniguchi et al. [47] ont proposé une méthode pour la génération de diagrammes de séquence compactés à partir des traces d'exécution collectées lors de l'exécution d'une application orientée objet. Leur méthode se compose de quatre règles de compactage pour générer automatiquement des diagrammes de séquence.

La réduction de la taille du diagramme généré est réalisée par abstraction des patrons de répétition. Dans ce travail, les traces sont représentées par un arbre dont chaque noeud est un appel de méthode. Les diagrammes de séquence sont compactés à travers l'identification et la réduction des séquences d'appels répétitifs et les séquences d'appels récursifs. Les auteurs ont proposé quatre règles de compactage qui visent des comportements spécifiques.

- Règle 1 : elle détecte la répétition de la même structure d'appel de méthodes et la compacte. En d'autres termes, cette règle considère deux sous arbres comme similaires si les sous arbres ont la même structure et leurs noeuds ont les mêmes méthodes et objets. Règle 1 utilise la première structure comme représentant et ensuite calcule le nombre de répétitions pour l'ajouter sous forme d'une étiquette à côté du diagramme de séquence.

  Une représentation schématique de cette règle est présentée dans la figure 2.3.

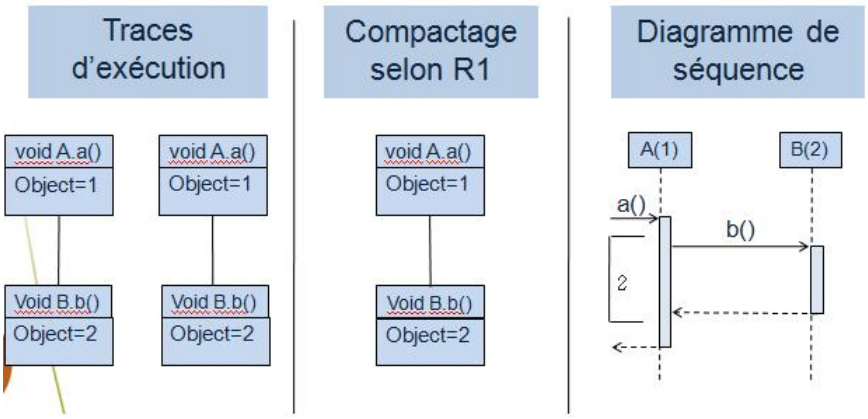


Figure 2.3 – R1 : Deux sous arbre complètement identiques

- Règle 2 : elle permet d'avoir des objets différents lorsqu'on a des structures d'appels de méthodes qui se répètent, mais qui ont des identifiants différents. On fait l'union entre les deux traces.

  La figure 2.4 présente cette règle.

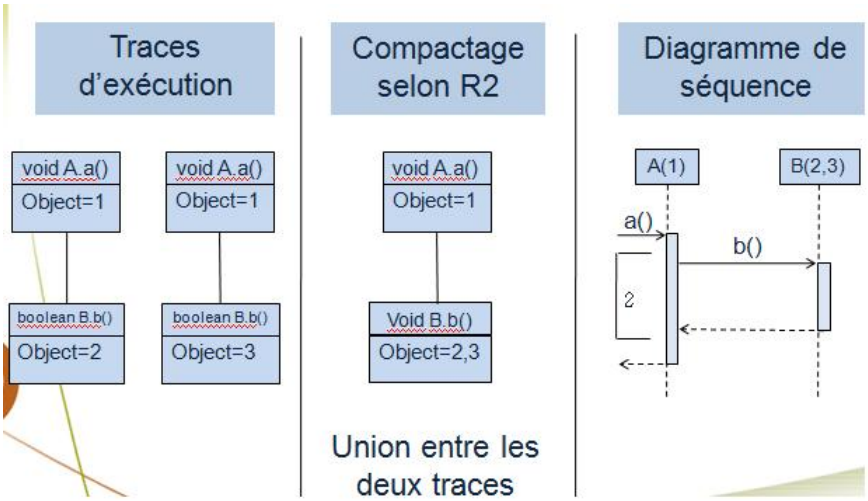


Figure 2.4 – R2 : Permettre des objets différents

- Règle 3 : elle compare deux arbres d'appels qui se diffèrent légèrement par quelques appels de méthodes, comme le montre la colonne "Traces d'exécution" de la figure 2.5. La Règle 3 permet donc de trouver les appels qui se trouvent dans l'un des deux arbres d'appels et absents de l'autre arbre d'appels. La deuxième colonne "Compactage selon R3" représente l'arbre obtenu après application de la règle 3 et en se basant sur le plus grand arbre. Une étiquette avec un point d'interrogation est ajoutée pour signaler que l'appel se trouve seulement dans l'une des deux arbres.

- Règle 4 : cette règle détecte les appels récursifs de méthodes pour réduire la taille de la structure. Deux sous arbres identiques utilisant chacune des appels récursifs sont représentés par un seul arbre grâce à cette quatrième règle de compactage.

  La figure 2.6 présente la règle 4.

Le problème dans la méthode de Taniguchi est qu'elle ne considère que quelques cas particuliers pour le compactage des traces d'exécution représentés par les quatre règles. À l'issue d'une expérience qui a réalisé en utilisant sa méthode, le meilleur ratio de compactage obtenu a été égal à 0.85%. Mais les diagrammes obtenus contiennent toujours des milliers de messages.

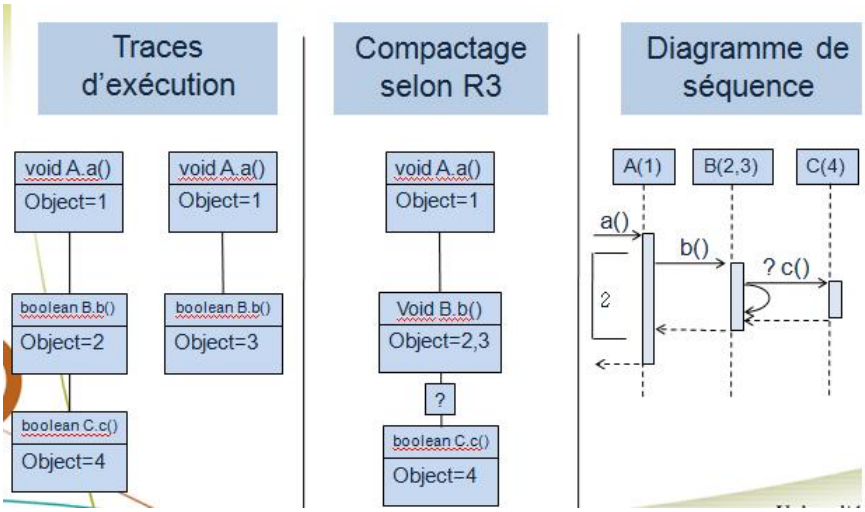


Figure 2.5 – R3 : Manque d'appels de méthodes

Delamare et al. [16] ont étendu le travail précédent en identifiant d'autres structures de contrôles. Leur approche se compose de deux étapes. Lors de la première étape, un diagramme de séquence simple est généré contenant seulement les appels de méthodes pour chaque exécution. La deuxième étape permet ensuite de tracer des diagrammes de séquence de haut niveau qui englobe le comportement général en combinant les diagrammes de séquence générés lors de la première étape et en essayant d'identifier les fragments "alt" et "loop" de UML2.0.

D'autres travaux comme [15, 30, 42] ont proposé des approches permettant de générer des diagrammes de collaboration d'UML réduits à partir des traces d'exécution du système. L'idée est d'identifier dans les traces les patrons qui se répètent pour aider à reconnaître les concepts importants. En effet, dans [42], Richner et al. proposent une approche pour récupérer les collaborations et les rôles à partir des traces d'exécution. Ils utilisent le "pattern matching" pour identifier les séquences d'exécution similaires dans les traces d'exécution d'un même scénario. Cette idée leur a permis d'aborder le problème de passage à l'échelle (scalability) puisque un développeur ne sera pas affronté à toutes les informations de la trace d'exécution, mais plutôt à des patrons de collaboration.

Dans [25], Hamou-Lhadj et al. ont proposé une technique semi-automatique pour

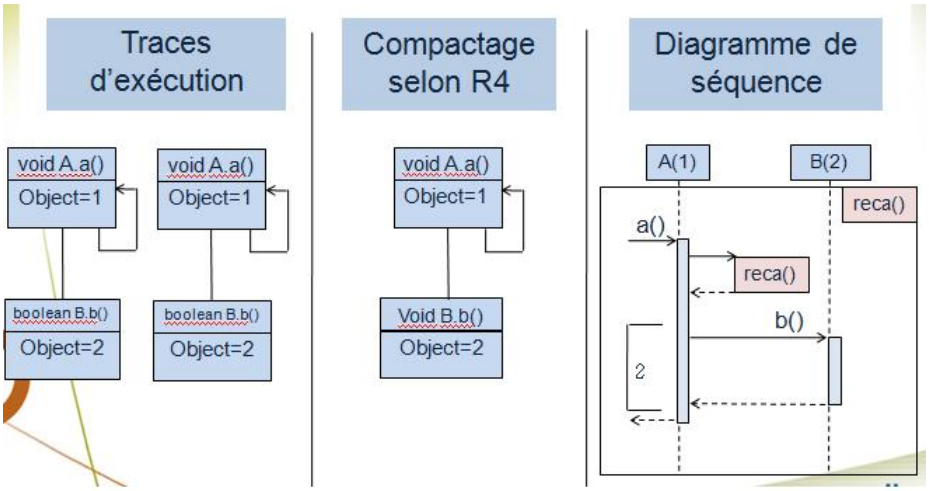


Figure 2.6 – R4 : Compactage des appels récursifs

résumer le contenu des traces d'exécution. Dans cette approche, ils ont imité le concept de résumé des textes. La méthode proposée prend des traces d'exécution en entrée et génère son résumé en sortie. Le résumé résultant des traces peut être par la suite converti en un diagramme de séquence d'UML et utilisé par les développeurs pour comprendre le comportement du système.

Briand et al. dans [7, 8] ont proposé une approche pour la rétro-ingénierie des diagrammes de séquence à partir des traces d'exécution. Ils utilisent l'ingénierie dirigée par les modèles pour transformer les traces d'exécution en diagrammes de séquence. Ils ont défini deux méta-modèles. Le premier méta-modèle décrit les diagrammes de séquence et le deuxième décrit les traces d'exécution. Les traces d'exécution sont générées par l'instrumentation du code source. 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. Le mapping est illustré par trois règles de transformation qui sont écrites en OCL (Object Constraint Language). Ces règles permettent d'identifier les messages "appels de méthodes", les messages "return" et les messages itératifs pour le modèle de diagramme de séquence. Le diagramme de séquence obtenu condense la trace d'exécution, en présentant une séquence de messages répétée une seule fois, et ajoute plus d'informations, en reportant la condition sous laquelle un appel est effectué dans le diagramme de

scénario.
L'une des principales contributions de cette approche est la manipulation de la concurrence. Mais, ces travaux se limitent à une seule exécution du système. Le résultat obtenu est donc un diagramme qui présente le comportement d'un scénario particulier. De plus, les imbrications de structures de contrôles ne sont pas supportées et toutes les classes sont instrumentées car il n'y a pas une technique de sélection partielle. Là encore, cette approche ne permet pas de générer des schémas compacts et abstraits.

Dans [23], les auteurs ont proposé une approche d'analyse dynamique pour la rétro-ingénierie des diagrammes de séquence et des machines à états. L'approche est définie en trois étapes. La première étape permet la génération des diagrammes de séquence à partir des fichiers de traces générés par l'outil d'analyse dynamique Caffeine [24]. Ensuite, la deuxième étape consiste à composer les diagrammes de séquence de base. Enfin, une fois les diagrammes de séquence sont générés et composés, ils produisent automatiquement des machines à états.

La visualisation interactive a été utilisée par De Pauw et al. [37] pour produire des vues de la trace avec des patrons d'exécution. L'objectif principal de cette technique est de permettre de naviguer dans une très grande trace d'exécution.

Parmi les principales limites dont souffre la rétro-ingénierie des diagrammes dynamiques d'UML, nous citons la grande quantité d'information qu'il faut réduire pour améliorer la lisibilité et la compréhension des diagrammes d'UML générés. Comme nous venons de présenter dans cette section, des travaux ont essayé de cibler cette problématique en réduisant les diagrammes UML produit et en essayant de trouver des solutions en utilisant des techniques de filtrage pour cibler seulement les informations pertinentes. Néanmoins, nous remarquons que plusieurs approches se basent sur une seule trace d'exécution pour faire la rétro-ingénierie. Cela est justifié par la difficulté de combiner plusieurs traces d'exécution. De cette façon, ces approches n'offrent pas une vue complète des cas d'utilisation vu l'existence de structures conditionnelles dans le code... En plus, la majorité de ces approches ont pris seulement des cas particuliers et n'ont pas proposé des solutions génériques qui permettent d'avoir une solution automatique pour tous les cas de figure qui nécessitent des réductions dans les traces d'exécu-

tion. Nous remarquons aussi l'absence de solutions pour des problèmes comme le cas de polymorphisme qui surviennent lors d'une boucle d'appels sur un type abstrait. Un exemple concret est celui d'un message dans un diagramme de séquence qui est envoyé trois fois par le même objet vers trois objets différents. Par contre, en analysant la structure, on trouve que les trois classes de ces objets héritent d'une même classe abstraite. Une telle idée pourrait optimiser davantage le diagramme de séquence généré.

2.5.3 Réduction des traces d'exécution

L'idée de réduction de la taille des traces d'exécution pour comprendre les comportements des logiciels n'est pas nécessairement reliée à l'extraction des diagrammes de séquence. Dans [50] et [10], par exemple, les auteurs ont proposé des stratégies d'échantillonnage pour sélectionner des sous-ensembles d'événements dans une trace d'exécution. Dans la même perspective, Hamou-Lhadj et al. proposent dans [27] une approche pour résumer des traces d'exécution en s'inspirant des méthodes de résumé de textes.

D'autres techniques consistent à cacher quelques éléments de la trace d'exécution pour réduire sa taille, comme les constructeurs, les getters, les setters, etc. [14, 55], éliminer les auto-appels [28], limiter la hauteur d'un arbre d'appels [29] ou élaguer les appels non importants de méthodes [3].

Dans notre approche, nous réutilisons quelques-unes de ces heuristiques de réduction, les améliorons et les combinons avec notre méthode de réduction du nombre d'objets.

D'autres travaux de recherche ont appliqué la méthode dite *de localisation de fonctions* pour identifier l'emplacement dans une trace d'exécution qui implémente des fonctionnalités dans une application [18], [17] et [39].

Pour produire un mapping des composants les plus importants à partir des traces d'exécution, l'analyse de concept est utilisée pour révéler les liens qui relient les caractéristiques et les composants. Une analyse des concepts est réalisée pour dévoiler la relation entre les caractéristiques et les composants via le mapping réalisé.

Une technique novatrice qui divise automatiquement le contenu d'une trace d'exécution en segments de trace plus petits et plus significatifs et correspondant aux phases

principales d'exécution du programme ont été proposés dans les travaux de [38] et [51]. Ces phases permettent de construire une vue de haut niveau de la trace d'exécution dans le but de simplifier l'exploration de grandes traces. Elles permettent aux ingénieurs de logiciels de parcourir la trace en se concentrant sur ses phases d'exécution à la place d'un flux de simples événements. Cette méthode est intéressante pour comprendre le comportement général d'un cas d'utilisation sans avoir des détails sur le fonctionnement interne.

Les approches basées sur l'analyse dynamique manquent de connaissances contextuelles pour faire la distinction entre les informations pertinentes et les détails d'implémentation.

2.6 Alignement des traces d'exécution

Tous les travaux de recherche cités dans les sections précédentes utilisent une seule trace d'exécution à la fois pour la compréhension d'un cas d'utilisation du système. Cependant, ceci ne permet pas de comprendre convenablement son comportement. Par conséquent, une idée plus sophistiquée consiste à utiliser plus qu'une seule trace d'exécution à la fois. Toutes les traces d'exécution appartiennent à des scénarios différents d'un même cas d'utilisation.
Néanmoins, en ayant plusieurs traces pour un même cas d'utilisation, nous serons face à une très grande quantité d'information. Ceci entraîne beaucoup de difficultés dans le traitement des informations recueillies.

Parmi les solutions les plus novatrices figure l'alignement de plusieurs traces d'exécution. Peu de travaux ont utilisé l'alignement des traces d'exécution.

Pour considérer plusieurs scénarios d'exécution, Grati et al. . [21] ont proposé une approche permettant d'avoir les opportunités d'alignement de plusieurs traces d'exécution à travers un environnement de visualisation. Le système enregistre les décisions d'alignement de l'analyste et les mappe sous forme d'un diagramme de séquence. Lors du processus d'interaction avec l'utilisateur, ce dernier peut à tout moment enlever des appels de méthodes ou d'objets qui ne sont pas utiles pour la compréhension du com-

portement, ce qui permet de réduire la taille des diagrammes obtenus. Cette approche nécessite que l'analyste ait une bonne connaissance du fonctionnement du programme, ce qui n'est pas toujours le cas.

Silva et al. dans [44] ont aligné les traces d'exécution déjà résumée pour séparer les parties communes des parties spécifiques. Bose et Van Der Aalst [5] a proposé une approche d'alignement récursif de plusieurs traces dans un but de compréhension et de diagnostic du comportement.

Johnson et al. ont proposé dans [31] une approche d'alignement de traces découpées en se basant sur l'indexage de l'exécution. La trace résultante permet d'identifier les différents flux et valeurs dans les exécutions.

L'alignement des traces d'exécution profite des travaux de recherche en bio-informatique qui s'intéressent à l'alignement des séquences d'ADN ou d'ARN. Appliquée au génie logiciel, cette technique peut servir pour aider à identifier les comportements identiques et les différences entre des exécutions différentes d'un même scénario.

Bose et al. [6] proposent une définition formelle de l'alignement des traces d'exécution T = $\{T_1, T_2, ..., T_n\}$ comme un mapping de l'ensemble des traces dans T vers un ensemble de traces T' = $\{T'_1, T'_2, ..., T'_n\}$ où l'ensemble T' englobe tous les éléments de l'ensemble T plus les gaps ajoutés lors de l'alignement afin d'améliorer la similarité :

- $|T'_1| = |T'_2| = ... = |T'_n|$ = m, avec "m" est la longueur de chaque trace d'exécution après l'ajout des gaps.
- en supprimant tous les gaps, on pourra revenir de T' à T.

Un alignement de deux traces d'exécution peut être représenté par une matrice rectangulaire A. Il peut y avoir plusieurs possibilités d'alignements pour un ensemble donné de traces. La longueur "m" de l'alignement doit être comprise entre la longueur maximale dans les traces T et la somme des longueurs de toutes les traces dans T.

Il existe deux types d'alignement : l'alignement par pairs (pairwise alignment) et l'alignement multiple (multiple alignment).

L'alignement de traces d'exécution par pairs est le type d'alignement qui se fait entre deux traces d'un même scénario. L'alignement entre deux traces, T_1 et T_2, peut être consi-

déré comme une transformation de la trace T_1 vers la trace T_2 ou inversement à travers un ensemble d'opérations d'édition appliquées itérativement aux deux traces. Le nombre des alignements possibles pour deux traces étant trop grand. Il faut donc utiliser des algorithmes permettant de trouver le meilleur alignement en calculant à chaque itération un score pour l'alignement. Ces algorithmes se basent généralement sur la programmation dynamique [3]. Parmi ces algorithmes, nous citons Needleman-Wunsch [35] et Waterman-Smith [46]. Nous reviendrons plus en détail sur l'algorithme Needleman-Wunsch dans le chapitre suivant.

Le deuxième type d'alignement des traces d'exécution est l'alignement multiple qui peut prendre plus que deux traces à la fois. Comme l'alignement par pairs, il se base aussi sur la programmation dynamique. L'un des mécanismes de scoring les plus connus de l'alignement multiple de séquences est la méthode de somme des pairs (sum-of-pairs : SP). Néanmoins, cette méthode est peu utilisée en raison de son temps de calcul et espace de stockage exponentiel.

Toutefois, le traitement des traces d'exécution ne se fait pas seulement avec les algorithmes de la bio-informatique. Il existe des travaux qui ont proposé d'autres méthodes pour la réduction de la taille des traces d'exécution. Nous reviendrons sur ces méthodes dans la section suivante.

[3] La programmation dynamique résout des problèmes en combinant des solutions de sous-problèmes. L'idée est de mémoriser les résultats de calculs intermédiaires qui seront probablement répétés.

CHAPITRE 3

APPROCHE

3.1 Introduction

Dans le cadre de ce travail de recherche, nous proposons une approche pour la rétro-ingénierie de diagrammes de séquence d'UML en se basant sur des traces d'exécution générées via notre traceur d'exécution. Nous exposerons dans ce chapitre les différentes étapes de notre approche. Elle commence par l'identification d'un ensemble de scénarios pour un cas d'utilisation. Ensuite, l'exécution de ces sénarios permetteront à notre traceurs de générer les traces d'exécution. L'alignement et la réduction des traces d'exécution permetteront d'avoir des traces d'exécution réduites qui englobe le plus possible des informations contenues dans un cas d'utilisation. À la fin, un diagramme de séquence est généré à partir de la trace d'exécution résultante des opérations d'alignement et de réduction.

3.2 Approche

La rétro-ingénierie des diagrammes dynamiques et de comportement se réalise pour plusieurs raisons, telles que l'analyse, la compréhension et la documentation du code. Notre travail est motivé par la re-documentation d'une application existante.

Notre approche extrait un diagramme de séquence à partir d'un ensemble de traces d'exécution appartenant à un même *cas d'utilisation*, c'est-à-dire une tâche réalisée par un usager utilisant l'application. Pour un cas d'utilisation il existe plusieurs *scénarios* possibles. Chaque scénario représente une exécution particulière du cas d'utilisation. Par exemple, pour une application d'ATM, " faire un retrait " peut être représenté par plusieurs scénarios possibles, comme : *fonds suffisants*, *fonds insuffisants* et *limite maximale quotidienne de retrait atteinte*.

Notre approche comprend quatre étapes, comme illustré dans la figure 3.1. Elle commence par l'identification d'un ensemble de scénarios pour un cas d'utilisation donné.

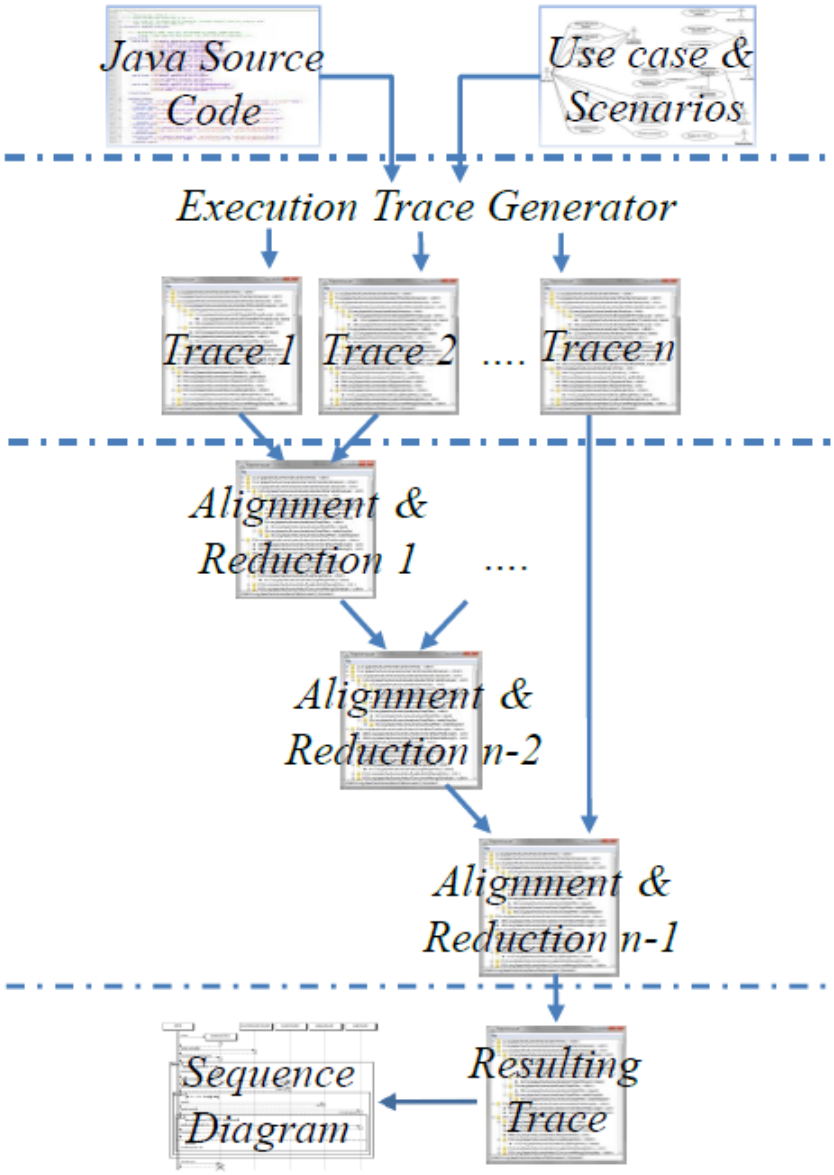


Figure 3.1 – Aperçu de l'approche

Une trace d'exécution est collectée pour chaque scénario en utilisant le JVMTI (Java Virtual Machine Tool Interface). Les traces d'exécution stockent les événements d'entrées et de sorties des méthodes.

Nous collectons un enregistrement détaillé pour chaque invocation de méthode. Chacune des lignes de la trace d'exécution contient les éléments suivants :

- Classe de définition de la méthode (*className*).
- Nom de la méthode (*methodName(c)*).
- Signature de la méthode (*MethSig(c)*) : englobe les types des paramètres en entrée de la méthode et le type retour.
- Identificateur unique de l'objet receveur (*IdR(c)*).
- Type dynamique de l'objet receveur (*DynType(c)*).
- Site d'appel dans la méthode appelante (*callSite(c)*).
- Type déclaré de l'objet receveur dans le site d'appel (*DecType(c)*).
- Temps effectif d'exécution de la méthode (*effecExecTime(c)*), excluant le temps d'exécution des méthodes qu'elle appelle.
- Vie d'une méthode (*LifeTime*), incluant le temps d'exécution de toutes les méthodes qu'elle appelle.
- Le nombre des appels directs réalisés par la méthode invoquée (*NSubCalls(c)*).

Dans la troisième étape, chaque trace est *réduite* en utilisant plusieurs heuristiques et abstractions. Nous effectuons deux types de réduction :

- Une réduction *horizontale* dont le but est d'éliminer les événements " les moins importants " de la trace. La réduction horizontale inclut aussi la détection des fragments répétés comme les boucles et les récursions. L'objectif est donc de réduire davantage la taille du diagramme résultant.

- Une réduction *verticale* dont le but est de réduire le nombre d'objets.

Ensuite, les traces d'exécution sont *alignées* pour produire une seule représentation du comportement associé au cas d'utilisation.

Finalement, un diagramme de séquence UML est généré à partir de la trace d'exécution finale (après les réductions et l'alignement) après sa conversion. [1].

Les sections suivantes présentent d'une façon détaillée les étapes de réduction et d'alignement.

3.3 Réduction des traces d'exécution

Généralement, les traces d'exécution sont constituées de plusieurs milliers (voires millions) d'événements et de centaines (voire milliers) d'objets. Ainsi, il est impossible d'afficher la trace dans un diagramme de séquence UML, et par conséquent, c'est compliqué pour un programmeur de comprendre le comportement de l'application, surtout lorsqu'il ne connaît pas les détails d'implémentation du système.

Dans notre travail, nous avons choisi de détecter tous les événements déclenchés par l'exécution du système à l'exception des appels de méthodes des librairies de Java. En plus, nous avons choisi de représenter la trace d'exécution originale sous la forme d'un arbre d'appel pour assurer son parcours et son changement (réduction de la trace) dans un temps raisonnable.

La trace d'exécution est représentée par un arbre d'appels dynamiques. Chaque noeud représente le début d'exécution d'une méthode. Si une méthode "e1" appelle une méthode "e2", alors nous représentons un lien du noeud représentant " e1 " vers son enfant " e2 ". Un événement de sortie d'une méthode est représenté par la remontée d'un niveau dans la chaîne d'appels de l'arbre. Un parcours en profondeur de l'arbre d'appels dynamique permet d'avoir la trace d'exécution originale.

Dans la sous-section suivante, nous présentons notre *approche de réduction* composée de deux étapes que nous avons appelées *reduction horizontale* et *reduction verticale*.

[1] La conversion est réalisée sous un format reconnu par l'outil d'affichage Quick Sequence Diagram Editor http ://sdedit.sourceforge.net/

3.3.1 Réduction du nombre d'événements (Réduction horizontale)

L'objectif de la réduction horizontale est de diminuer significativement le nombre d'événements dans la trace sans toucher aux appels des méthodes les plus importantes à la compréhension du comportement du programme. Cet objectif est atteint grâce à l'utilisation de la méthode d'élagage (pruning) des appels les moins significatifs dans la trace d'exécution et aussi par la détection de quelques *fragments combinés d'UML* comme les boucles.

3.3.1.1 Élagage (Pruning)

Dans cette partie, nous avons étendu le travail de Bohnet et al. dans [3] et nous lui avons apporté aussi quelques adaptations. Leur technique utilise une classification des événements de la trace d'exécution en se basant sur une expérimentation. Cette dernière est partie des observations réalisées sur des programmeurs. Leurs tâches étaient de localiser des parties du programme qui correspondent à l'exécution de quelques fonctionnalités.

Dans leur travail de recherche, Bohnet et al. se sont basés sur une expérimentation qui leur a permis de déduire que les appels de méthodes dans un arbre d'appels n'ont pas tous la même importance. Ils ont alors classé les appels de méthodes en 4 types comme suit :

- Les appels les moins significatifs (less significant calls) : sont représentés par des appels de méthodes qui n'appellent pas d'autres méthodes, c'est à dire des feuilles de l'arbre d'appels, ou bien des appels de méthodes qui déclenchent peu de méthodes et dont le temps d'exécution est "trop court". Ainsi, ces appels de méthodes ne contiennent vraiment pas d'informations qui permettent de mieux comprendre le programme, d'où la nécessité de les enlever de la trace d'exécution.

- Les appels de délégation (Delegating calls) : ce sont des appels de méthodes dans l'arbre d'appels dont l'objectif est de distribuer les tâches aux autres méthodes qu'elles appellent. Les appels de délégation se trouvent au deuxième niveau de

l'arbre d'appels et donc viennent directement sous sa racine. Les appels de délégation n'ont pas une importance et sont supprimées de l'arbre d'appels.

- Les appels de coordination (Coordinating calls) : ont une très grande importance dans la trace d'exécution puisqu'ils déclenchent l'exécution de méthodes respectant deux paramètres seuils : le seuil du nombre de sous appels d'une méthode *TNSubCalls* et le seuil du temps d'exécution d'une méthode *TeffecExecTime*. Dans la pratique, l'implémentation d'une application est généralement décomposée en modules. Les appels de coordination semblent être le résultat de cette pratique. Ils sont donc cruciaux lors de l'exploration des traces et pour essayer de comprendre le comportement du système qui est capturé par la trace, ce qui fait que ces appels seront conservés dans la trace d'exécution.

- Les appels de travail (Working calls) : sont des méthodes qui ne déclenchent pas l'exécution d'autres appels significatifs mais qui ont une durée d'exécution relativement importante. La durée d'exécution d'une méthode est un indicateur de son importance, pour cela qu'on a décidé de garder de telle méthode dans la trace d'exécution.

Nous avons utilisé un algorithme basé sur des heuristiques pour identifier automatiquement les appels importants de méthodes dans la trace d'exécution. Pour cela, nous nous sommes basés sur le nombre d'appels déclenchés par un appel d'une méthode "c " (*NSubCalls(c)*) et sa durée d'exécution (*effecExecTime(c)*).

L'algorithme est contrôlé par deux paramètres seuils reliés par la condition logique "ET" :

- *TNSubCalls* qui représente le nombre minimal de sous appels pour qu'une méthode soit considérée comme importante.

- *TeffecExecTime* qui représente la durée d'exécution minimale d'une méthode pour qu'elle soit considérée comme importante.

En conséquence, ces deux seuils permettent de déterminer les *appels de coordination (coordinating calls)* et les *appels de travail (working calls)* pour les conserver dans la

trace d'exécution et élaguer tous les autres appels de méthodes non importantes. Nous fixons les deux seuils en utilisant un fichier journal contenant toutes les informations de la trace d'exécution. Ce fichier aide le programmeur à décider les valeurs à utiliser pour les seuils et pour savoir à peut près les méthodes qui seront supprimées de la trace d'exécution. Dans certains cas, les noms des paquetages, des classes et des méthodes peuvent être d'une grande utilité dans le choix des valeurs des deux seuils.

Le nombre d'appels significatifs *(NbSigSubCalls(c))* déclenchés directement par un appel "c" est égal au nombre d'enfants qui ont soit $NSubCalls \geq TNSubCalls$ et/ou $effecExecTime \geq TeffecExecTime$. Si le nombre de $NbSigSubCalls(c) \geq 2$ alors " c " est un appel de coordination. Sinon, si $NbSigSubCalls(c) = 0$ et ($NSubCalls \geq TNSubCalls$ ou $effecExecTime \geq TeffecExecTime$) alors " c " est un appel de travail.

Les séquences d'appels moins significatives (less significant calls) entre un appel de coordination et un appel de travail sont fusionnées dans un seul *appel d'agrégation*.

3.3.1.2 Localiser les fragments combinés d'UML

La localisation de quelques fragments combinés d'UML dans la trace d'exécution est très utile pour réduire considérablement la taille et donc pour améliorer la compréhensibilité du diagramme de séquence résultant. Pour se faire, notre traceur retourne dans chaque événement d'entrée " c " le *site d'appel (call site)* de l'appel du parent (callSite(c)). Nous proposons deux algorithmes qui localisent *les boucles* et *la récursivité*. Tous les deux utilisent la recherche en profondeur dans l'arbre d'appels dynamique.

a. Localisation des boucles : Plusieurs noeuds frères S={n_1, n_2, ..., n_n} peuvent être fusionnés en une boucle si le motif composé du quadruplet { *className*, *methodName*, *callSite* et *IdR*} est répété au moins deux fois. Dans ce cas, tous les noeuds répétés sont fusionnés dans un seul noeud. Une boucle peut contenir des appels à plusieurs méthodes. Les événements correspondants dans l'arbre d'appel de la trace d'exécution contiennent des répétitions de ces méthodes. En plus, si un appel d'une méthode est guidé par une condition, alors il n'apparaîtra pas dans tous les cas. Cette situation rend difficile la détection de ces boucles. Cependant, il existe plusieurs méthodes qui

traitent le problème de la plus longue chaîne commune [45].

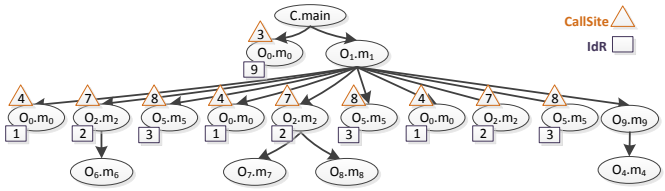
Nous nous sommes inspirés de la méthode de l'arbre des suffixes compacts pour déterminer la plus longue chaîne commune. Chaque sous-chaîne commune est une séquence de quadruplets composé chacun de (className, methodName, callSite, IdR). Deux noeuds n_1 et n_2 sont égaux si $\textit{className}(n_1) = \textit{className}(n_2)$ et $\textit{methodName}(n_1) = \textit{methodName}(n_2)$ et $\textit{IdR}(n_1) = \textit{IdR}(n_2)$ et $\textit{callSite}(n_1) = \textit{callSite}(n_2)$. Lorsque l'algorithme trouve le motif représentant la sous-chaîne commune la plus longue, il fusionne les noeuds similaires en un seul noeud.

Le noeud résultant de la fusion des noeuds frères similaires et représentant la boucle contient entre autres :

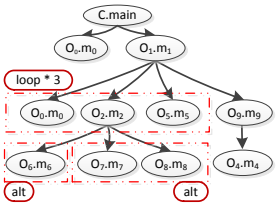
- L'information qui indique le noeud de début et le noeud de fin de la boucle et le nombre de répétitions de la boucle.
- Chacun des champs *effecExecTime* et *LifeTime* est égal à la somme des champs correspondants des noeuds fusionnés.
- *NSubCalls* est égale à la somme des valeurs dans les champs correspondants dans les noeuds fusionnés.
- Chaque enfant des noeuds fusionnés sera un enfant du noeud résultant. Si une branche de l'arbre d'appel prenant comme source un noeud de la boucle ne se répète pas dans les noeuds fusionnés alors il s'agit d'une alternative (alt)

La figure 3.2 représente l'application de notre algorithme sur un arbre d'appel dynamique représentant un petit exemple de la trace d'exécution d'une application ATM de banque. Comme nous pouvons le constater, le nombre de noeuds (événements) a été considérablement réduit et par conséquent la compréhensibilité du comportement de l'application a été améliorée.

b. Localisation de la récursivité : Un ensemble de noeuds S={n_1, n_2, ...,n_n} peuvent être fusionnés dans une récursion si :



(a) Arbre d'appel avant la détection des boucles



(b) Arbre d'appel après la detection de la boucle

Figure 3.2 – Localisation des boucles

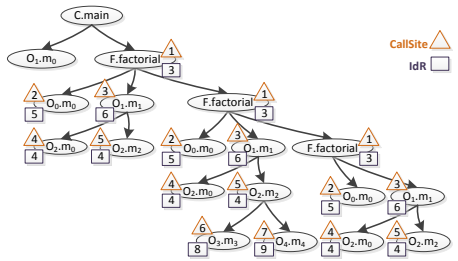
- n_1 est le parent de n_2 et n_2 est le parent de n_3 et n_i est le parent de n_{i+1}. (avec i $<$ n-1) et n_{n-1} est le parent de n_n.
- Pour chaque noeud n_2 et son parent n_1, $className(n_1) = className(n_2)$, $methodName(n_1) = methodName(n_2)$, $IdR(n_1) = IdR(n_2)$ et $callSite(n_1) = callSite(n_2)$.

Les enfants des noeuds fusionnés seront les enfants du noeud résultant. S'il y a des enfants qui apparaîssent dans un noeud et n'apparaissent pas dans le second, alors ceci signifie qu'il y a une condition qui est vraie pour un des événements fusionnés et non pas pour les autres. Dans ce cas, nous utilisons le fragment combiné d'UML " alt ".

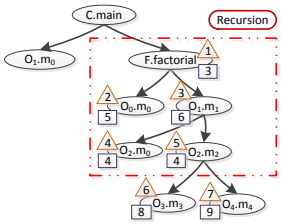
La figure 3.3 représente un cas d'école représentant la détection d'une récursion.

3.3.1.3 Combinaison de la localisation des fragments d'UML et de l'élagage

La localisation des fragments combinés d'UML et l'élagage sont deux méthodes intéressantes pour réduire de la taille de la trace d'exécution. Dans notre approche, nous



(a) Arbre d'appel avant la détection de la récursivité



(b) Arbre d'appel après la détection de la récursivité

Figure 3.3 – Localisation des structures récursives

combinons ces deux techniques en commençant tout d'abord par la localisation des fragments combinés suivie de l'élagage. La trace d'exécution résultante ne contient que les événements les plus importants et ne retient qu'une seule copie de plusieurs événements successifs. Nous appelons cette combinaison *la réduction horizontale* car elle permet de réduire le nombre d'événements du diagramme de séquence résultant.

3.3.2 Réduction du nombre d'objets (réduction verticale)

En plus de la réduction du nombre d'événements obtenu dans la section précédente grâce à l'élagage, nous avons aussi exploré la possibilité d'utiliser les types dynamiques (DynType) ou les types déclarés (DecType) à la place d'utiliser le type où la méthode a été définie (className).

Pour distinguer entre les trois types que nous avons dans la trace d'exécution, prenons un petit exemple dans la figure 3.4. Si nous avons trois classes " A ", " B " et " C " où la classe " C " hérite de la classe " B " et la classe " B " hérite de la classe " A ". Supposons que la classe " B " contient une méthode " m1() ". Si nous déclarons " a " de type " A " et instancions " a " comme " new C() ", alors " a.m1() " a la classe " A " comme le *type déclaré* (DecType), " C " comme le *type dynamique* (DynType) et " B " la classe qui contient la définition de la méthode (className).

Le choix d'utiliser le type déclaré ou du type dynamique, à la place du nom de la

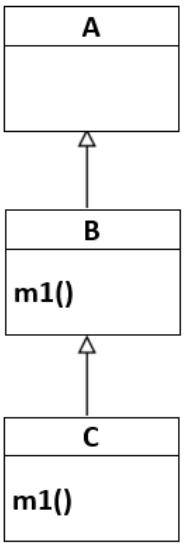


Figure 3.4 – ClassName-DecType-DynType

classe, découle de la nécessité de se rapprocher le plus possible de la conception du développeur de l'application. Dans ce cas, nous aurons une meilleure compréhension du comportement du système. En effet, le type déclaré nous permettra de savoir l'intention du programmeur, c'est-à-dire l'objet qu'il utilise pour appeler la méthode.

3.3.2.1 Réduire le nombre de types dynamiques

Nous trions les événements en entrée de la trace d'exécution selon les types déclarés. Ensuite, pour chaque type déclaré (DecType), s'il y a deux ou plus de types dynamiques (DynType) correspondants, alors nous modifions le type déclaré seulement dans le contexte local et non pas pour toutes les apparitions de ces types dynamiques. Nous appelons ce type de réduction *DecTyp_DynTyp_1_n* et nous utilisons dans ce cas *DynType* comme objets du diagramme de séquence résultant.

3.3.2.2 Réduire le nombre des types déclarés

Contrairement, nous trions dans ce cas les événements en entrée de la trace d'exécution selon les types dynamiques. Ensuite, pour chaque type dynamique (DynType),

si nous trouvons au moins deux types déclarés (DecType) correspondant, nous modifions les types déclarés par le type dynamique seulement dans le contexte local et non pas pour toutes les apparitions de ces types déclarés. Nous appelons ce type de réduction *DynTyp_DecTyp_1_n* et nous utilisons dans ce cas *DecType* comme objets dans le diagramme de séquence résultant.

3.3.3 Combinaison de la réduction verticale et la réduction horizontale

Comme nous l'avons présenté dans la section 3.3.1, nous avons choisi de réaliser l'élagage et la localisation des fragments combinés d'UML. Pour réduire le nombre d'objets, nous avons choisi d'utiliser soit les types déclarés soit les types dynamiques. L'ordre de combinaison de la réduction verticale et la réduction horizontale n'est pas important. La trace d'exécution finale sera le résultat de la combinaison de ces deux types de réduction.

3.3.4 Abstraction

En plus de l'application des méthodes de réductions énoncées ci-haut, nous avons aussi considéré quelques autres hypothèses pour la manipulation des événements dans la trace d'exécution. Nous avons choisi d'éliminer les événements qui correspondent aux *appels de méthodes statiques* et aux *appels de classes abstraites* car ils ne doivent pas être présentés dans un diagramme de séquence d'UML. En plus, nous avons supprimé les objets créés et qui ne sont pas utilisés, c'est-à-dire qu'il n y'a dans la trace aucun événement qui utilise cet objet. Ceci provient dans le cas où le programmeur de l'application a instancié un objet mais il ne l'a pas utilisé par la suite, ou bien si l'objet ne reçoit que les messages de la provenance des constructeurs (<init>). Ce dernier cas peut se produire après l'élagage en utilisant les heuristiques qui suppriment les événements non importants et ne laisse que les constructeurs.

3.4 Alignement des traces d'exécution

Dans cette phase, l'objectif est d'avoir une seule trace d'exécution qui combine tous les comportements des traces d'exécution représentant chacune un scénario différent, mais appartenant toutes au même cas d'utilisation. Nous appelons cette méthode l'alignement des traces d'exécution. L'alignement doit être réalisé selon des règles et des métriques bien définies pour assurer une cohérence temporelle.

Nous avons défini un algorithme d'alignement des traces d'exécution qui prend en entrée plusieurs traces d'exécution et donne en sortie une trace d'exécution alignée qui contient le comportement de toutes les traces d'exécution alignées. Cet algorithme identifie les fragments d'implémentation communs et les situations spécifiques.

L'alignement consiste en deux étapes successives. Il commence avec la définition d'un algorithme permettant de réaliser un alignement exact entre deux traces d'exécution appelé *correspondance des messages* (message matching) suivi par l'identification des opportunités additionnelles d'alignement basées sur la *correspondance des objets* (object matching). Quand il faut aligner plus que deux traces d'exécution, l'alignement des traces est effectué graduellement par deux traces à la fois. Pour le premier alignement, il sera réalisé entre deux traces initiales, et ensuite, il sera effectué entre la trace résultante de l'alignement et une autre trace d'exécution non encore alignée. Ce processus est répété jusqu'à l'alignement des toutes les traces d'exécution correspondantes à différents scénarios du même cas d'utilisation. Chaque message de la trace d'exécution résultante contient un label qui indique le (ou les) nom(s) de(s) la trace(s) d'exécution originale(s).

3.4.1 Correspondance des messages (Message matching)

Puisque les traces d'exécution sont représentées par des arbres d'appel dynamiques, notre algorithme d'alignement est basé sur un processus récursif où deux arbres correspondants à deux traces sont comparés niveau par niveau. Pour chaque niveau, deux noeuds sont comparés si leurs parents respectifs ont déjà été alignés. Deux noeuds sont alignés s'ils ont le même *className* et *methodName*. Notre algorithme commence par

la comparaison des racines des deux arbres de traces. Si deux noeuds sont alignés alors leurs enfants seront par la suite comparés. L'algorithme est appliqué récursivement pour chaque pair de noeuds alignés.

Pour comparer deux séquences de noeuds de deux traces d'exécution différentes dont leurs parents respectifs ont été alignés, nous appliquons l'algorithme de Needleman-Wunsch [36] qui est un algorithme pour l'alignement de séquence utilisé à l'origine en bio-informatique. Il permet un alignement global maximal de deux chaines de caractères basé sur la programmation dynamique. Il permet de trouver le score d'alignement maximal entre deux chaines de caractères. L'idée de la programmation dynamique est de construire un alignement optimal en utilisant des solutions optimales dans des séquences plus petites, c'est-à-dire stocker les résultats des calculs intermédiaires susceptibles d'être répétées.

L'algorithme de Needleman-Wunsch permet l'introduction des "écarts" pour améliorer la similarité entre les séquences. Nous avons choisi de ne pas attribuer des pénalités dans le cas d'addition d'écarts. Une pénalité égale à 1 va être assignée dans le cas de concordance (matching). Ainsi, le principe de pondération utilisé est le suivant :

- *Initialisation* : Elle consiste à créer une matrice où le nombre de colonnes est égal au nombre d'éléments dans la première séquence et le nombre de lignes est égal au nombre d'éléments dans la deuxième séquence. Pour illustration, prenons un petit exemple.

  Prenons l'exemple de deux exécutions "T1" et "T2" d'un même scénario (Figure 3.5). Les lettres en majuscule représentent les noms des classes, les lettres en minuscule représentent les noms des méthodes et les nombres représentent les adresses mémoires (pour l'exemple ci-dessous, les adresses utilisées ne sont pas représentatives).

  Les arbres d'appels dynamiques correspondants respectivement à "T1" et "T2" sont représentés dans 3.6 par "A1" et "A2".

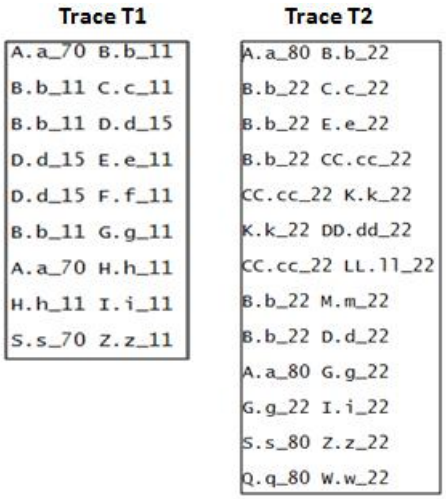


Figure 3.5 – Exemple de deux traces d'exécution

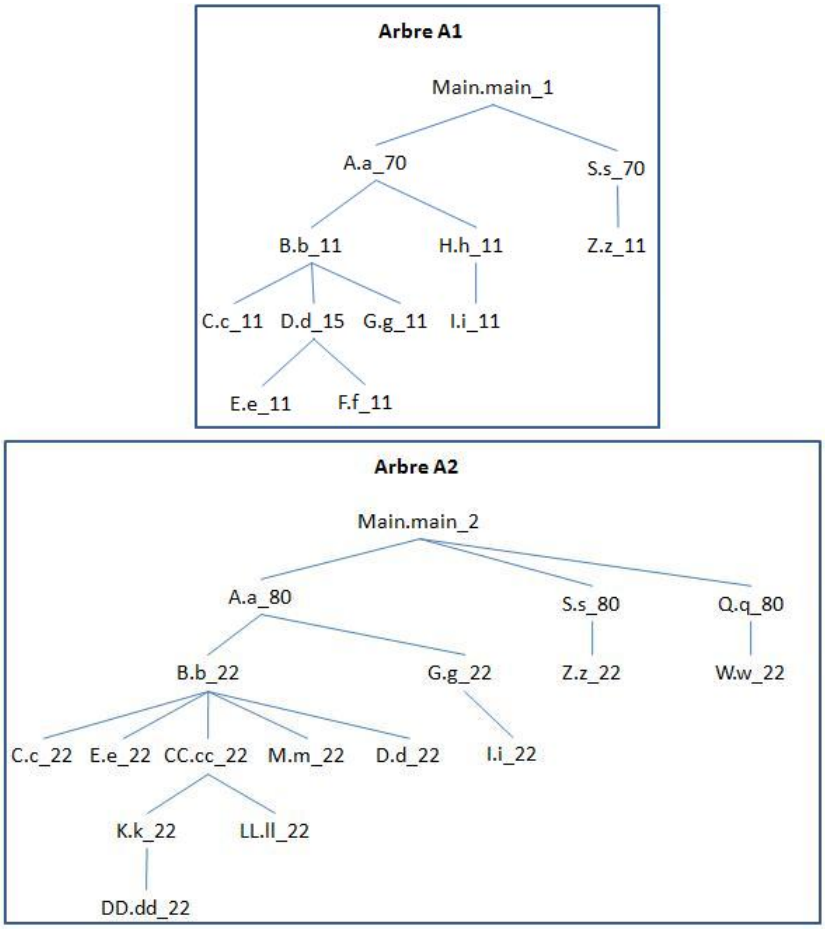


Figure 3.6 – Arbres d'appels dynamiques correspondants à T1 et T2

Considérons deux séquences à aligner :

seq1 : C.c_11 D.d_15 G.g_11 et

seq2 : C.c_22 E.e_22 CC.cc_22 M.m_22 D.d_22 (où chaque élément représente className.methodName_IdR)

Ainsi, la matrice d'initialisation serait comme suit (Tableau 3.I) :

i/j	***-***	***C.c_11***	***D.d_15***	***G.g_11***
-	0	0	0	0
C.c_22	0	-	-	-
E.e_22	0	-	-	-
CC.cc_22	0	-	-	-
M.m_22	0	-	-	-
D.d_22	0	-	-	-

Tableau 3.I – Initialisation de la matrice d'alignement pour Needleman-Wunsch

- *Calcule du score et remplissage de la matrice* : Pour chaque cellule $M_{i,j}$ de la matrice M, le score est calculé comme suit :

  $M_{i,j}$ = Max $\{M_{i-1,j-1} + S_{i,j} ; M_{i,j-1} + w ; M_{i-1,j} + w\}$

  $S_{i,j}$ définit le score de correspondance de deux caractères " i " et " j ". Lorsque un écart (gap) *w* est inséré avant " i " (respectivement j), il encourt une pénalité d'écart(i,_) (respectivement écart(_,j)).

  Ainsi, pour calculer $M_{i,j}$ il faut avoir calculé $M_{i-1,j}$, $M_{i,j-1}$ et $M_{i-1,j-1}$. Comme chaque séquence commence par le résidu de concordance S1 et que nous avons posé que w=0. Donc : $M_{1,1}$ = Max[$M_{0,0+1}$, $M_{1,0+0}$, $M_{0,1+0}$] = Max[1, 0, 0] = 1. Nous pouvons donc inscrire un "1" dans $M_{1,1}$.

  Le processus de calcul est poursuivi jusqu'au remplissage de toute la matrice. Le tableau 3.II montre la matrice résultante après son remplissage. Son score étant égal à "2".

  L'utilisation de la programmation dynamique permet de réduire le temps de calcul pour la comparaison de séquence d'exponentiel (O(k^n)) à un temps quadratique (O(m*n)), avec "m" et "n" les longueurs respectives des deux séquences à comparer). En effet, la table contient "m*n" valeurs dont chacune s'effectue en un temps

i/j	***-***	***C.c_11***	***D.d_15***	***G.g_11***
-	0	0	0	0
C.c_22	0	1.0	1.0	1.0
E.e_22	0	1.0	1.0	1.0
CC.cc_22	0	1.0	1.0	1.0
M.m_22	0	1.0	1.0	1.0
D.d_22	0	1.0	2.0	2.0

Tableau 3.II – Matrice d'alignement pour Needleman-Wunsch après son remplissage

constant.

Cette étape nous a permis de savoir que le score d'alignement maximal pour les deux séquences est 2. Cette information nous donne les meilleurs scores parmi une collection de séquences à comparer à la cible. Mais pour connaître un alignement optimal qui donne ce score, il faut suivre la méthode suivante :

- *Détermination de l'alignement optimal* : L'outil que nous avons développé pour l'alignement de deux séquences permet d'avoir l'alignement optimal actuel. En effet, il prend la case qui contient le score maximal, qui est $M_{m,n}$ (la cellule en bas à droite) et le compare à ses voisins.

  Nous avons appelé un recul à gauche dans la matrice par "Ouest" ou "O", un déplacement en haut par "Nord" ou "N" et un déplacement par "Nord-Ouest" ou "NO" pour un recul vers la cellule du coin entre les deux cellules "O" et "N". Ainsi, nous pouvons constater que :

  - un déplacement "NO" correspond à un matching entre les deux éléments comparés,
  - un déplacement "N" signifie l'ajout d'un gap dans la séquence horizontale,
  - un déplacement "O" signifie l'ajout d'un gap dans la séquence verticale.

  L'algorithme de Needleman-Wunsch est répété pour tous les niveaux des deux arbres à aligner. Une fois tous les alignements réalisés, nous aurons l'arbre d'appels dynamiques résultants. La figure 3.7 montre l'arbre résultant pour notre exemple.

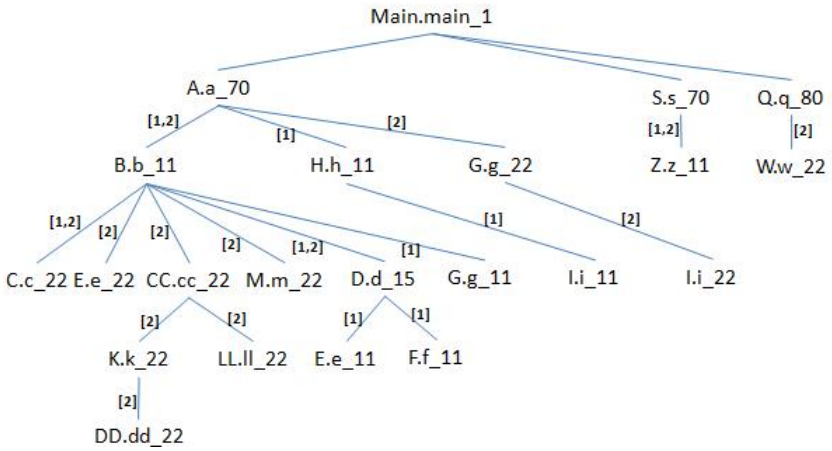


Figure 3.7 – Arbre d'appels dynamiques résultant de l'alignement de T1 et T2

Les étiquettes sur les liens entre les noeuds sont utilisées pour faciliter la compréhension de l'alignement entre les deux arbres "T1" et "T2". L'étiquette [1] signifie que le noeud en dessous provient de l'arbre "T1" suite à l'ajout d'un gap dans l'arbre "T2", l'étiquette [2] signifie que le noeud en dessous provient de l'arbre "T2" suite à l'ajout d'un gap dans l'arbre "T1", l'étiquette [1,2] signifie qu'il y'avait eu un matching entre les deux noeuds ce qui a permis d'avoir un seul noeud en dessous du noeud actuel.

Enfin, notre outil génère automatiquement un fichier qui contient la trace résultante de l'alignement des deux traces d'exécution.

La figure 3.III montre l'application de l'algorithme de Needleman-Wunsch pour l'exemple choisi. La couleur grise représente l'alignement optimal. Le calcul de l'alignement est réalisé de la droite vers la gauche.

Ainsi, l'alignement optimal nécessite l'insertion de quatre écarts (gap) dont chacun est représenté par " - ". Deux pairs de noeuds sont fusionnés : {C.c_11, C.c_22} et {D.d_15, D.d_22}.

C.c_11	–	–	–	D.d_15	G.g_11
C.c_22	E.e_22	CC.cc_22	M.m_22	D.d_22	–

i/j	-	C.c_11	D.d_15	G.g_11
-	0	0	0	0
C.c_22	0	1.0	1.0	1.0
E.e_22	0	1.0	1.0	1.0
CC.cc_22	0	1.0	1.0	1.0
M.m_22	0	1.0	1.0	1.0
D.d_22	0	1.0	2.0	2.0

Tableau 3.III – Application de l'algorithme de Needleman-Wunsch

3.4.2 Correspondance des objets (object matching)

L'algorithme de correspondance permet, en plus de réduire le nombre de messages, de réduire le nombre d'objets chaque fois que nous avons deux objets de deux traces d'exécutions différentes qui ont les mêmes messages en entrée et en sortie et qui peuvent être alignés.

L'objectif est d'avoir un diagramme de séquence compacte via la correspondance des objets. Ainsi, nous fusionnons deux objets de deux traces d'exécution différentes même s'il existe des messages de l'un ou l'autre des deux objets qui ne peuvent pas être alignés. Ceci arrive lorsque les deux objets ont le même chemin de création, c'est-à-dire la même séquence de parents dans l'arbre d'appel dynamique.

3.5 Combinaison de l'alignement et de la réduction

Nos approches de réduction et d'alignement sont complémentaires. D'une part, la réduction permet d'avoir des traces d'exécution compactes. Chaque trace d'exécution représente un scénario spécifique. Toutes les traces appartiennent au même cas d'utilisation. D'autre part, le processus d'alignement permet d'avoir un diagramme de séquence plus informatif qui montre le comportement de deux ou plusieurs traces d'exécution.

CHAPITRE 4

ÉTUDE DE CAS

4.1 Introduction

Pour illustrer l'efficience de l'utilisation de notre approche, nous avons choisi l'application *ATM Banking Simulation System* [1]. Ce choix a été motivé par plusieurs raisons : D'une part, c'est une application qui est relativement bien documentée avec des diagrammes d'UML, incluant des diagrammes de séquence, et dont le code source en Java est aussi disponible. D'autre part, plusieurs travaux similaires comme [9, 21] l'ont aussi utilisé pour la validation de leurs approches. Pour ces raisons, il nous sera possible de comparer les diagrammes de séquence générés par notre approche avec ceux de la documentation.

4.2 Étude de cas

Nous visons, dans cette étude de cas, à répondre à deux questions de recherche :

- Jusqu'à quel point on est capable de réduire une trace d'exécution.
- Jusqu'à quel point ce qui reste de la trace d'exécution, représenté par un diagramme de séquence, est informatif.

4.2.1 ATM Banking simulation

ATM Banking est une application Java composée de 38 classes et de 6 packages. Puisque le système de simulation d'ATM utilise un seul thread, l'application de notre approche permet de générer un seul diagramme de séquence qui inclut tout le comportement du système. La figure 4.1 montre un diagramme de séquence de haut niveau d'une session. Chaque transaction est détaillée dans un diagramme de séquence/collaboration séparé.

[1] `http://math-cs.gordon.edu/local/courses/cs211/ATMExample/`

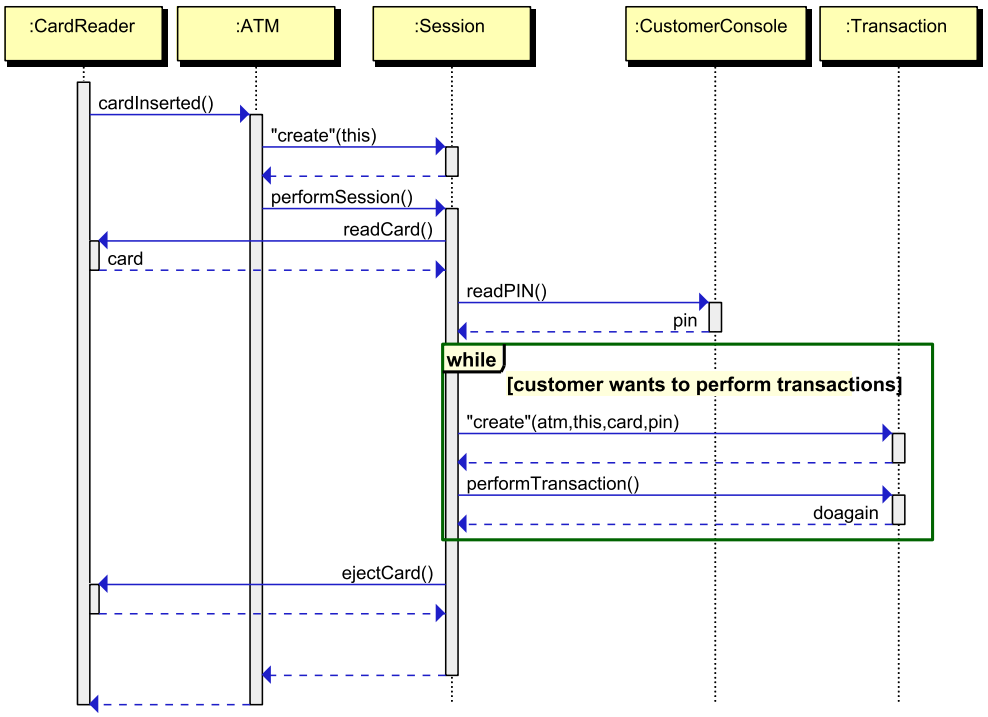


Figure 4.1 – ATM Session sequence diagram

Plusieurs cas d'utilisation ont été proposés dans la documentation. Pour illustrer notre approche, considérons le cas d'utilisation de retrait.

4.2.1.1 Définition des scénarios

Pour considérer les variations dans le cas d'utilisation de retrait, deux scénarios de cas d'utilisation sont proposés : le scénario principal avec un retrait normal (appelé *Scn-A*) et un scénario alternatif avec un retrait qui excède la quantité d'argent disponible dans le distributeur d'argent ATM (appelé *Scn-B*). L'exécution des deux scénarios est réalisée via l'interface graphique du simulateur de l'interface de l'ATM. Les étapes du premier scénario sont les suivantes :

a. Lancer l'application simulateur d'ATM et passer à la position it ON,

b. Entrer le montant de l'argent disponible dans le distributeur d'argent (5 billets de 20$),

c. Cliquer sur le bouton *"Insert ATM Card"* et "entrer le *numéro de la carte (card number)"* et le code *PIN*,

d. Choisir le type de la transaction *"retrait (Withdrawal)"*,

e. Sélectionner *checking* pour vérifier le compte duquel nous voulons retirer l'argent,

f. Entrer 40$ pour le montant à retirer du compte,

g. Répondre par (*NON*) à la question *"voulez-vous réaliser une autre transaction ?"*,

h. Cliquer sur le bouton *"prendre le reçu (Take receipt)"* et

i. passer à la position d'arrêt *"OFF"* pour arrêter le système.

Le scénario *Scn-B* a les mêmes étapes que *Scn-A*. La seule différence est que le montant à retirer excède le montant disponible dans le distributeur d'argents. Ensuite, l'étape " f " est remplacée par les étapes suivantes :

f1. Entrer 200$ pour le montant à retirer du compte,

f2. Un message d'erreur s'affiche (*montant excède l'argent disponible (Amount exceeds the available money). , Choisir un autre montant.*),

f3. Entrer 60$ pour le montant à retirer du compte,

Pour répondre à la première question de recherche, *jusqu'à quel point on est capable de réduire une trace d'exécution*, nous avons utilisé plusieurs types de réduction. Dans ce qui suit, nous présentrons les résultats obtenus grâce à l'application de ces réductions et à leur combinaison.

4.2.1.2 Réduction horizontale

Comme discuté dans la section 3.3.1, la réduction horizontale des messages consiste à élaguer les messages les moins importants et aussi à localiser des fragments UML.

L'élagage (pruning) de messages utilise deux paramètres qui sont le nombre d'appels et le temps effectif d'exécution. Les résultats qui seront présentées ici ont été obtenus avec un seuil égal à 2 pour le nombre de sous appels (*TNSubCalls=2*) et un seuil égal à 0.01µs pour le temps effectif d'exécution (*TeffecExecTime=0.01*µ*s*).

Après inspection des diagrammes de séquence générés après l'application des heuristiques de réduction sur notre exemple, nous avons remarqué que le nombre de messages a été réduit de 77% alors que le nombre d'objets a été réduit de 32% par rapport au nombre de messages et d'objets initiaux respectivement.

4.2.1.3 Combiner la réduction horizontale et la réduction verticale

Après application sur notre étude de cas, nous avons conclu que la réduction verticale permet de réaliser des réductions supplémentaires du nombre d'objets dans la trace d'exécution à peu près égal à 6%. Nous obtenons des traces d'exécution plus compactes lorsque nous combinons la réduction horizontale et verticale. En effet, le nombre d'objets dans les deux scénarios *Scn-A* et *Scn-B* a diminué de 31% par rapport aux résultats obtenus en appliquant la réduction horizontale seulement. Aussi, le nombre de message a diminué de à peu près de 60%.

4.2.1.4 Alignement

Le nombre de messages dans la trace d'exécution alignée comparé au nombre total de messages (somme des messages de Scn-A et de Scn-B) a été réduit de 46.03% en raison de la correspondance de messages. Le nombre d'objets a été réduit de 66.67% grâce à la correspondance des objets qui ont le même chemin de création.

Malgré les résultats encourageants de l'alignement et le fait qu'il permet d'assurer une meilleure compréhension de l'utilisation du cas d'utilisation exécuté, il existe encore

trop de messages non importants dans le diagramme de séquence aligné. D'où l'importance de combiner notre approche d'alignement et notre approche de réduction.

4.2.1.5 Combinaison de l'alignement et de la réduction

Les résultats quantitatifs obtenus de la combinaison de l'alignement et de la réduction montrent que dans le diagramme de séquence résultant, le nombre d'objets a été réduit de 76.54%.

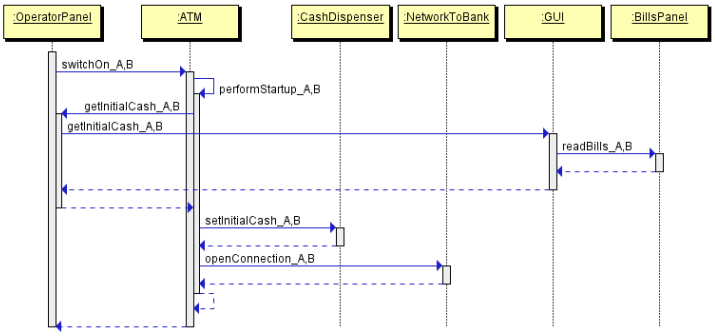
De plus, le nombre de messages dans la trace d'exécution alignée et réduite est aux alentours de 90%.

La deuxième question de recherche, *jusqu'à quel point ce qui reste de la trace d'exécution, représenté par un diagramme de séquence, est informatif*, permet de valider les diagrammes de séquence générés par notre approche. Nous nous sommes basés sur les diagrammes de séquence fournit dans la documentation du système de simulation d'ATM. Chacun des diagrammes de la documentation montre une fonctionnalité.

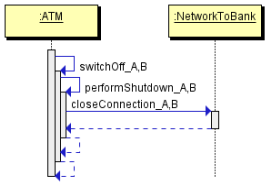
Pour un but de présentation, nous avons décomposé manuellement le diagramme de séquence généré en plusieurs diagrammes de séquence. Chacun des diagrammes correspond à un des quatre diagrammes de séquence proposés dans la documentation du système de simulation d'ATM.

Les figures 4.2a et 4.2b montrent des coupes du diagramme de séquence généré par notre approche. Ils correspondent respectivement au démarrage et à l'arrêt du système de simulation d'ATM. Ces deux diagrammes de séquence ressemblent beaucoup aux diagrammes de séquence correspondant de la documentation moyennant quelques différences mineures. En effet, la différence est que notre diagramme de séquence contient, en plus, les objets *GUI* et *BillsPanel* et les messages *getInitialCash* et *ReadBills*. Nous croyons que l'existence de ces informations supplémentaire est avantageuse pour le programmeur pour l'informer de l'interaction avec l'interface graphique.

La figure 4.3 suivante expose la partie présentant une *session*. Une comparaison manuelle entre le diagramme de séquence de la documentation et celui généré par notre méthode montre que notre diagramme encapsule tous les événements cruciaux proposés par le concepteur et contient en plus d'autres événements de nature fonctionnelle.



(a) Diagramme de séquence de démarrage du système



(b) Diagramme de séquence d'arrêt du système

Figure 4.2 – Diagrammes de séquence de démarrage et d'arrêt du système

Ces événements sont *readCardNumber* et *readMenuChoice*. Chacun d'eux fournit une information supplémentaire sur le comportement du système lors de l'ouverture d'une session.

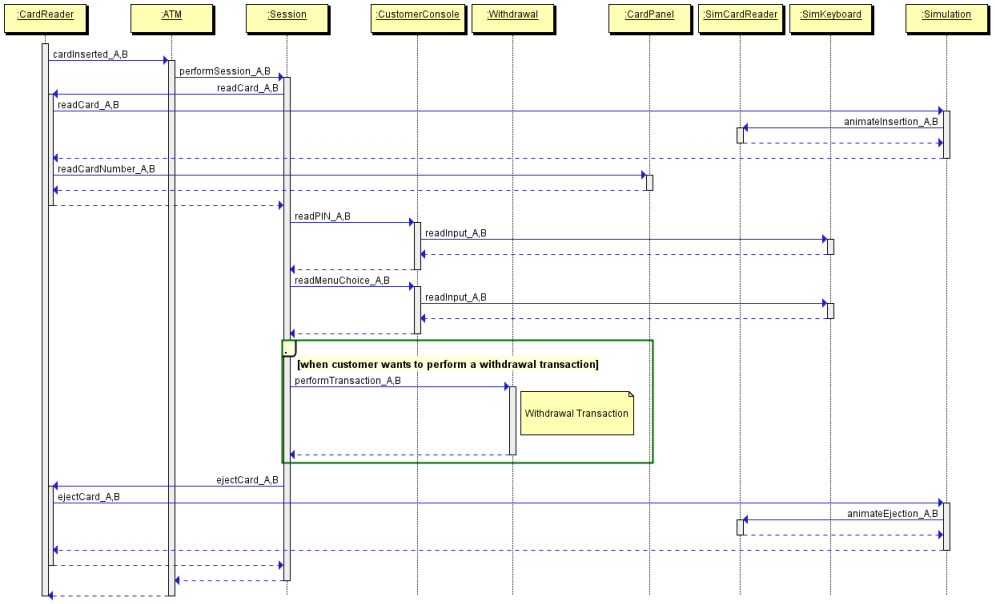


Figure 4.3 – Digramme de séquence d'une session

Dans notre approche, nous n'avons pas tenu en compte des types abstraits et nous avons considéré seulement les types définis. Par conséquent, l'objet *Transaction* qui est un type abstrait dans l'application n'apparaît pas dans notre diagramme de séquence

généré. En fait, notre diagramme de séquence contient à la place de l'objet *transaction*, l'objet définit correspondant, qui est *Withdrawal* (retrait) dans les scénarios *Scn-A* et *Scn-B* choisis dans cette expérience.

Le diagramme de séquence de retrait (Withdrawal) généré est représenté dans la figure 4.4 suivante. Pour le valider, nous avons utilisé le diagramme de séquence de transaction et le diagramme de collaboration de transaction de retrait de la documentation. La comparaison montre que notre diagramme de séquence contient tous les objets et messages des diagrammes correspondant dans la documentation et contient en plus les objets de simulation de l'interaction de l'utilisateur avec l'interface graphique de l'application.

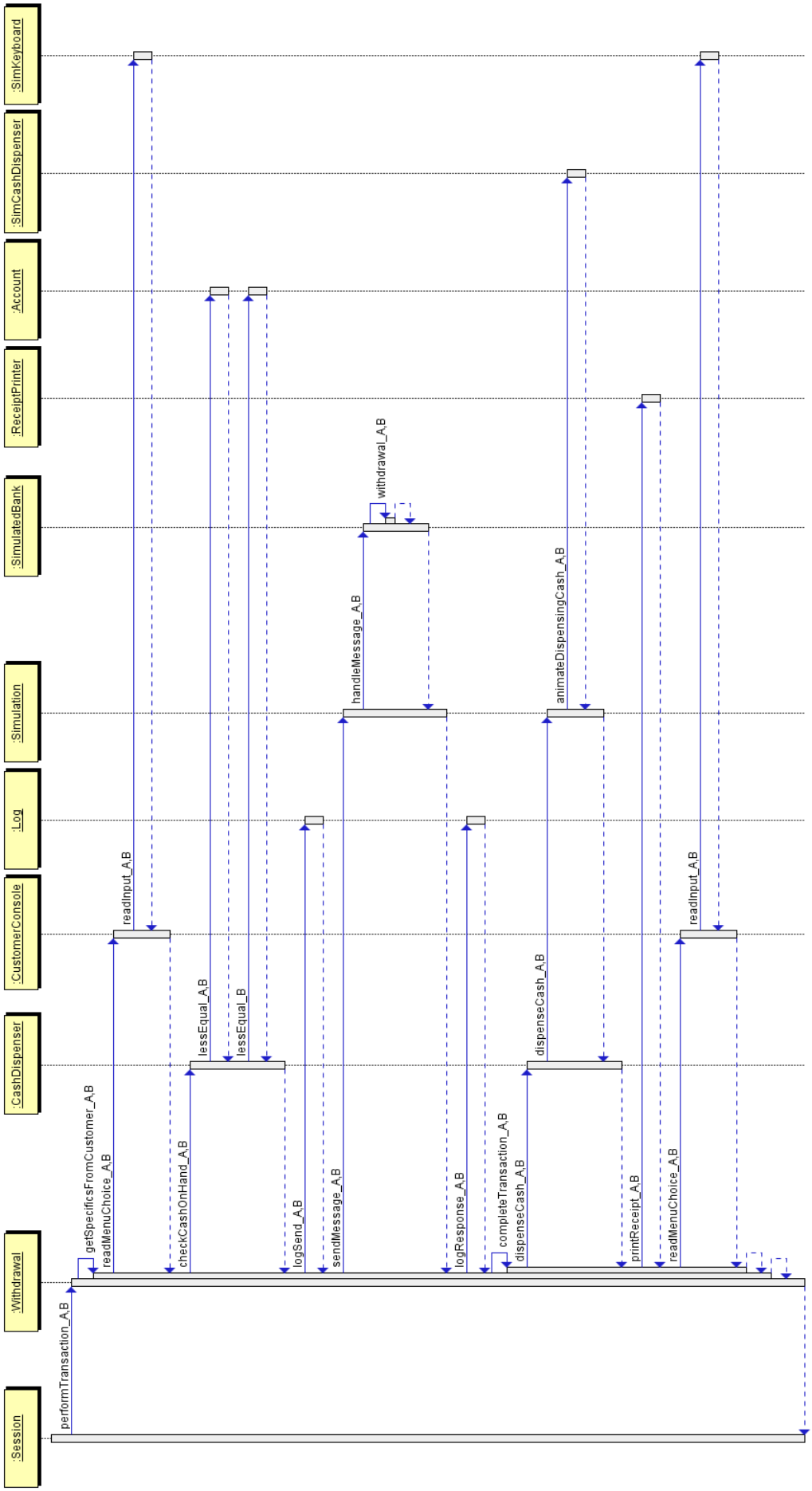


Figure 4.4 – Diagramme de séquence de transaction de retrait (Withdrawal)

Selon la validation précédente sur le système de simulation d'ATM bancaires, nous pouvons conclure que nos heuristiques permettent de générer des diagrammes de séquences compactes et expressifs. La comparaison avec les diagrammes de la documentation montre que nous sommes trop près de la conception du système.

CHAPITRE 5

CONCLUSION

Dans ce travail de recherche, nous avons proposé une approche basée sur des heuristiques, pour la génération automatique, par rétro-ingénierie, des diagrammes de séquences à partir des traces d'exécution afin de comprendre le comportement des programmes orientés objet. Notre approche combine l'utilisation de l'analyse dynamique et de l'analyse statique des programmes.

Notre méthode est composée de quatre étapes : Nous commençons par l'identification d'un ensemble de scénarios pour un cas d'utilisation donné. Ensuite, l'exécution de chaque scénario identifié est accompagnée par la génération d'une trace d'exécution via notre outil de génération de traces qui utilise le JVMTI. Dans la troisième étape, les traces d'un même scénario sont réduites en nous basant sur nos heuristiques d'abstractions. Ces traces sont par la suite alignées en une seule trace d'exécution qui englobe tout le comportement du scénario. Finalement, un diagramme de séquence UML est généré à partir de la trace d'exécution finale.

Pour tester notre approche, nous avons utilisé une application, de simulation d'un guichet bancaire ATM, bien documentée pour mener une étude comparative entre les diagrammes de séquence générés par notre approche et les diagrammes de séquence dans la documentation. Les résultats ont montré que l'application de nos heuristiques permet d'avoir des diagrammes de séquence qui représentent le plus fidèlement possible le comportement du cas d'utilisation exécuté. Notre approche facilite considérablement la tâche des programmeurs qui veulent comprendre, maintenir ou mettre à jour des programmes et qui n'ont pas une documentation à jour.

Même si les premiers résultats semblent prometteurs, de nombreuses améliorations sont envisagées. Dans ce travail de recherche, nous avons implémenté un premier prototype pour faire les tests. Il est possible de l'étendre par un framework de visualisation interactif qui permet de voir en temps réel le diagramme de séquence résultant en faisant varier les valeurs des seuils utilisés dans nos heuristiques.

À moyen terme, nous prévoyons étendre notre approche pour permettre la rétro-ingénierie d'autres diagrammes dynamiques d'UML (comme les diagrammes d'états, les diagrammes d'interaction, les diagrammes de collaboration, etc.). Nous pensons que la rétro-ingénierie de ces diagrammes permettra d'améliorer encore plus la compréhension du comportement du programme et permettra aussi d'explorer d'autres attributs et caractéristiques qui ne peuvent pas être présentés par un diagramme de séquence.

BIBLIOGRAPHIE

[1] T. Ball. The concept of dynamic analysis. Dans *European Software Engineering Conference, Springer-Verlag*, pages 216–234, 1999.

[2] T. J. Biggerstaff, B. G. Mitbander et D. Webster. The concept assignment problem in program understanding. Dans *Proceedings of the 15th international conference on Software Engineering*, 1993.

[3] Johannes Bohnet, Martin Koeleman et Jürgen Döllner. Visualizing massively pruned execution traces to facilitate trace exploration. Dans *VISSOFT*, pages 57–64, 2009.

[4] Borland. Together. URL `www.borland.com/together`.

[5] R. P. Jagadeesh Chandra Bose et Wil M. P. van der Aalst. Trace alignment in process mining : opportunities for process diagnostics. Dans *Proceedings of the 8th international conference on Business process management*, BPM'10, pages 227–242, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15617-7, 978-3-642-15617-5.

[6] R. P. Jagadeesh Chandra Bose et Wil M. P. van der Aalst. Trace alignment in process mining : opportunities for process diagnostics. Dans *Proceedings of the 8th international conference on Business process management*, BPM'10, pages 227–242, 2010.

[7] L. C. Briand, Y. Labiche et Y. Miao. Towards the reverse engineering of UML sequence diagrams. WCRE '03, 2003.

[8] Lionel C. Briand, Yvan Labiche et Johanne Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[9] Lionel C. Briand, Yvan Labiche et Y. Miao. Towards the reverse engineering of uml sequence diagrams. Dans *10th Working Conference on Reverse Engineering*

(WCRE 2003), 13-16 November 2003, Victoria, Canada, pages 57–66. IEEE Computer Society, 2003.

[10] Andrew Chan, Reid Holmes, Gail C. Murphy et Annie T. T. Ying. Scaling an object-oriented system execution visualizer through sampling. Dans *in Proc. 11th Int. Workshop on Program Comprehension*, pages 237–244, 2003.

[11] E. J. Chikofsky et J. H. Cross II. Reverse engineering and design recovery : A taxonomy. *IEEE Software*, 1990.

[12] T. A. Corbi. Program understanding : challenge for the 1990's. *IBM Systems Journal*, 28:294–306, 1989.

[13] B. Cornelissen, A. Zaidman et A. Van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, pages 341–355, 2011.

[14] Bas Cornelissen, Arie Van Deursen, Leon Moonen et Andy Zaidman. Visualizing testsuites to aid in software understanding. Dans *In Proc. 11th European Conf. on Software Maintenance and Reengineering (CSMR),pages 213âĂŞ222. IEEE,2007*.

[15] W. De Pauw, D. Lorenz, J. Vlissides et M. Wegman. Execution patterns in object-oriented visualization. Dans *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*, pages 16–16, 1998.

[16] Romain Delamare, Benoit Baudry et Yves Le Traon. Reverse-engineering of uml 2.0 sequence diagrams from execution traces. Dans *Proceedings of the workshop on Object-Oriented Reengineering at ECOOP 06*, Nantes, France, juillet 2006.

[17] Bogdan Dit, Meghan Revelle, Malcom Gethers et Denys Poshyvanyk. Feature location in source code : a taxonomy and survey. *Journal of Software : Evolution and Process*, 25(1):53–95, 2013.

[18] Andrew David Eisenberg. Dynamic feature traces : Finding features in unfamiliar code. Dans *In Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 337–346. IEEE Computer Society Press, 2005.

[19] M. Fowler. *UML Distilled : A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional ; 3 edition, 2003.

[20] C. Ghezzi, M. Jazayeri et D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991. ISBN 9780138204327.

[21] Hassen Grati, Houari Sahraoui et Pierre Poulin. Extracting sequence diagrams from execution traces using interactive visualization. Dans *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 87–96, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4123-5.

[22] Object Management Group. URL `http://www.omg.org`.

[23] Y.-G. Guéhéneuc. Automated reverse-engineering of UML v2.0 dynamic models. Dans *proceedings of the 6 th ECOOP Workshop on Object-Oriented Reengineering*, 2005.

[24] Y.-G. Guéhéneuc, R. Douence et N. Jussien. No Java without caffeine : A tool for dynamic analysis of Java programs. Dans *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 117 – 126, 2002.

[25] A. Hamou-Lhadj et T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. Dans *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 181–190, 2006.

[26] A. Hamou-Lhadj et Timothy C. Lethbridge. Understanding the complexity embedded in large routine call traces with a focus on program comprehension tasks. 2010.

[27] Abdelwahab Hamou-Lhadj et Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. Dans *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2601-2.

[28] Abdelwahab Hamou-Lhadj et Timothy C. Lethbridge. A survey of trace exploration tools and techniques. Dans *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '04, pages 42–55. IBM Press, 2004.

[29] Abdelwahab Hamou-Lhadj, Timothy C. Lethbridge et Lianjiang Fu. Challenges and requirements for an effective trace exploration tool. *International Conference on Program Comprehension*, page 70, 2004. ISSN 1092-8138.

[30] D. Jerding et S. Rugaber. Using visualization for architectural localization and extraction. Dans *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 56–, 1997.

[31] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud et Dawn Song. Differential slicing : Identifying causal execution differences for security applications. Dans *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 347–362, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4402-1.

[32] R. Kollman, P. Selonen, E. Stroulia, T. Systä et A. Zundorf. A study on the current state of the art in tool-supported UML-based static reverse engineering. Dans *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, 2002.

[33] Ralf Kollmann et Martin Gogolla. Capturing dynamic program behavior with uml collaboration diagrams. Dans *Proceedings of the Fifth European Conference on*

Software Maintenance and Reengineering, CSMR '01, pages 58–, Washington, DC, USA, 2001. IEEE Computer Society.

[34] Unified Modeling Language. URL `http://www.uml.org`.

[35] Saul B. Needleman et Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[36] Saul B. Needleman et Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[37] Wim De Pauw, David Lorenz, John Vlissides et Mark Wegman. Execution patterns in object-oriented visualization. Dans *In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS âĂŹ98*, pages 219–234, 1998.

[38] Heidar Pirzadeh, Akanksha Agarwal et Abdelwahab Hamou-Lhadj. An approach for detecting execution phases of a system for the purpose of program comprehension. Dans *Proceedings of the 2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications*, SERA '10, pages 207–214, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4075-7.

[39] Denys Poshyvanyk, Malcom Gethers et Andrian Marcus. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol.*, 21(4):23 :1–23 :34, février 2013. ISSN 1049-331X.

[40] Rational. Rational test realtime. URL `www.rational.com/products/testrt`.

[41] S. P. Reiss et M. Renieris. Encoding program executions. 2001.

[42] T. Richner et S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. Dans *In Proceedings of ICSM 2002 (International Conference on Software Maintenance*, pages 34–43, 2002.

[43] Atanas Rountev. Static control-flow analysis for reverse engineering of uml sequence diagrams. Dans *In Proc. 6th Workshop on Program Analysis for Software Tools and Engineering (PASTEï£¡05*, pages 96–102. ACM Press, 2005.

[44] Luciana Lourdes Silva, Klerisson Ribeiro Paixao, Sandra de Amo et Marcelo de Almeida Maia. On the use of execution trace alignment for driving perfective changes. Dans *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 221–230, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4343-7.

[45] Steven Skiena. *The Algorithm Design Manual (2. ed.)*. Springer, 2008.

[46] T. Smith et M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[47] Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto et Katsuro Inoue. Extracting sequence diagram from execution trace of java program. Dans *In Proc. International Workshop on Principles of Software Evolution (IWPSE'05*, pages 148–151, 2005.

[48] Paolo Tonella et Alessandra Potrich. Reverse engineering of the interaction diagrams from c++ code. Dans *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 159–, Washington, DC, USA, 2003. IEEE Computer Society.

[49] UML2.0. URL `http://laurent-audibert.developpez.com/Cours-UML/?page=diagrammes-interaction`.

[50] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-benson, Darin Wright, Darin Swanson et Jeremy Isaak. Visualizing dynamic software system information through high-level models, 1998.

[51] Yui Watanabe, Takashi Ishio et Katsuro Inoue. Feature-level phase detection for execution trace using object cache. Dans *Proceedings of the 2008 international workshop on dynamic analysis : held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 8–14, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-054-8.

[52] Ba Wichmann, Aa. Canning, D. L. Clutterbuck, L A Winsborrow, N. J. Ward et D. W. R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 1995.

[53] A. Zaidman et S. Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution*, 20, November 2008.

[54] Andy Zaidman et Serge Demeyer. Automatic identification of key classes in a software system using webmining techniques. *J. Softw. Maint. Evol.*, 20(6):387–417, novembre 2008. ISSN 1532-060X.

[55] Iyad Zayour et Timothy C. Lethbridge. A cognitive and user centric based approach for reverse engineering tool design. Dans *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '00, pages 16–. IBM Press, 2000.