UNIVERSITÉ DE MONTRÉAL

CONCEPTS EXTRACTION FROM EXECUTION TRACES

SOUMAYA MEDINI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
NOVEMBRE 2014

CONCEPTS EXTRACTION FROM EXECUTION TRACES

*This dissertation is dedicated to my son Iyess,
who is 18 months old in spite of his mother spending
so much time away from him working on this dissertation.*

ACKNOWLEDGMENTS

First and foremost, I thank Allah (God) for providing me the health, patience, and knowledge to complete my dissertation.

I would like to thank my supervisor, Dr. Giuliano Antoniol for his advices and inspiration throughout this research. His immense knowledge in Software Engineering, and great skills to explain things simply and clearly make him a great advisor. He could not imagine how much I have learned from him. I also thank him for keeping his door always open for helpful conversation.

I would like also to thank my co-supervisor, Dr. Yann-Gaël Guéhéneuc , whose support, encouragement and advices, made my thesis work possible. I have learned much from him about presenting ideas and collaborating with other colleagues. He helped me whenever I needed his assistance.

A special thanks to Dr. Wahab Hamou-Lhadj, Dr. Bram Adams and Dr. Hanifa Boucheneb for taking the time to read this dissertation and to serve on my thesis committee.

My appreciation goes to Dr. Massimiliano Di Penta and Dr. Paolo Tonella for their feedback during the various phases of this dissertation.

I am very thankful to all my colleagues of SOCCERLab and Ptidej teams for their helpful discussions. I would like to express my gratitude to all the members of department of computing and software engineering at the École Polytechnique de Montréal. I am indebted to all the students who participated in our experiment.

I would like also to thank my sister, Salma, for her continuous support along the way. A special thanks to my husband, Aymen, for always believing in me and giving me the encouragement to complete this thesis.

Last but not the least, an enormous thanks to my parents, Bacha and Faouzia. I cannot thank them enough for all the sacrifices they made during every phase of this dissertation.

RÉSUMÉ

L'identification de concepts est l'activité qui permet de trouver et localiser l'implémentation d'une fonctionnalité d'un logiciel dans le code source. L'identification de concepts permet d'aider les développeurs à comprendre les programmes et de minimiser l'effort de maintenance et d'évolution des logiciels. Dans la littérature, plusieurs approches statiques, dynamiques et hybrides pour l'identification des concepts ont été proposées. Les deux types statiques et dynamiques ont des avantages et des inconvénients et se complètent mutuellement en approches hybrides. Par conséquent, de nombreux travaux récents ont porté sur des approches hybrides pour améliorer les performances en terme de temps et de précision du processus d'identification de concepts. De plus, les traces d'exécution sont souvent trop larges (en termes de nombre de méthodes invoquées) et elles ne peuvent pas être utilisées directement par les développeurs pour les activités de maintenance.

Dans cette thèse, nous proposons d'extraire l'ensemble des concepts des traces d'exécution en se basant sur des approches hybrides. En effet durant la maintenance d'un logiciel, les developpeurs cherchent à trouver et à comprendre le(s) segment(s) qui implémente(nt) le concept à maintenir au lieu d'analyser en détails toute la trace d'exécution. L'extraction de concepts facilite les tâches de maintenance et d'évolution des logiciels en guidant les développeurs sur les segments qui implémentent les concepts à maintenir et ainsi réduire le nombre de méthodes à analyser.

Nous proposons une approche basée sur la programmation dynamique qui divise la trace d'exécution en segments qui représentent des concepts. Chaque segment implémente un et un seul concept et est défini comme une liste ordonnée des méthodes invoquées, c'est-à-dire une partie de la trace d'exécution. Un concept peut être implémenté par un ou plusieurs segments.

Ensuite, nous proposons une nouvelle approche (SCAN) pour attacher des étiquettes aux segments de la trace d'exécution. Nous utilisons la recherche d'information (IR) pour extraire une étiquette formée par des mots clés qui définissent le concept implémenté par un segment. Les étiquettes permettent aux développeurs d'avoir une idée du concept implémenté par les méthodes du segment et de choisir les segments qui implémentent les concepts à maintenir.

Les segments qui implémentent les concepts à maintenir peuvent être de larges tailles en terme de nombre de méthodes invoquées et ainsi difficiles à comprendre. Nous proposons de diminuer la taille des segments en gardant juste les plus importantes méthodes invoquées. Nous réalisons des expériences pour évaluer si des participants produisent des étiquettes différentes lorsqu'on leur fournit une quantité différente d'informations sur les segments. Nous montrons qu'on conserve 50% ou plus des termes des étiquettes fournies par les participants tout en réduisant considérablement la quantité d'informations, jusqu'à 92% des segments, que les participants doivent traiter pour comprendre un segment.

Enfin, nous étudions la précision et le rappel des étiquettes générées automatiquement par SCAN. Nous montrons que SCAN attribue automatiquement des étiquettes avec une précision moyenne de 69% et de un rappel moyen de 63%, par rapport aux étiquettes manuelles produites par au moins deux participants.

L'approche SCAN propose aussi l'identification des relations entre les segments d'une même trace d'exécution. Ces relations fournissent une présentation globale et de haut niveau des concepts misent en œuvre dans une trace d'exécution. Ceci permet aux développeurs de comprendre la trace d'exécution en découvrant les méthodes et invocations communes entre les segments. Nous montrons que SCAN identifie les relations entre les segments avec une précision supérieure à 75% dans la plupart des logiciels étudiés.

À la fin de cette thèse, nous étudions l'utilité de la segmentation automatique des traces d'exécution et l'affectation des étiquettes durant les tâches d'identification des concepts. Nous prouvons que SCAN est une technique qui supporte les tâches d'identification de concepts. Nous démontrons que l'extraction de l'ensemble des concepts des traces d'exécution présentée dans cette thèse guide les développeurs vers les segments qui implémentent les concepts à maintenir et ainsi réduire le nombre de méthodes à analyser.

ABSTRACT

Concept location is the task of locating and identifying concepts into code region. Concept location is fundamental to program comprehension, software maintenance, and evolution. Different static, dynamic, and hybrid approaches for concept location exist in the literature. Both static and dynamic approaches have advantages and limitations and they complement each other. Therefore, recent works focused on hybrid approaches to improve the performance in time as well as the accuracy of the concept location process. In addition, execution traces are often overly large (in terms of method calls) and they cannot be used directly by developers for program comprehension activities, in general, and concept location, in particular.

In this dissertation, we extract the set of concepts exercised in an execution trace based on hybrid approaches. Indeed, during maintenance tasks, developers generally seek to identify and understand some segments of the trace that implement concepts of interest rather than to analyse in-depth the entire execution trace. Concept location facilitates maintenance tasks by guiding developers towards segments that implement concepts to maintain and reducing the number of methods to investigate using execution traces.

We propose an approach built upon a dynamic programming algorithm to split an execution trace into segments representing concepts. A segment implements one concept and it is defined as an ordered list of the invoked methods, i.e., a part of the execution trace. A concept may be implemented by one or more segments.

Then, we propose SCAN, an approach to assign labels to the identified segments. We uses information retrieval methods to extract labels that consist of a set of words defining the concept implemented by a segment. The labels allow developers to have a global idea of the concept implemented by the segment and identify the segments implementing the concept to maintain.

Although the segments implementing the concept to maintain are smaller than the execution traces, some of them are still very large (in terms of method calls). It is difficult to understand a segment with a large size. To help developers to understand a very large segment, we propose to characterise a segment using only the most relevant method calls.

Then, we perform an experiment to evaluate the performances of SCAN approach. We investigate whether participants produce different labels when provided with different amount of information on a segment. We show that 50% or more of the terms of labels provided by participants are preserved while drastically reducing, up to 92%, the amount of information that participants must process to understand a segment.

Finally, we study the precision and recall of labels that are automatically generated by

SCAN. We show that SCAN automatically assigns labels with an average precision and recall of 69% and 63%, respectively, when compared to manual labels produced by merging the labels of at least two participants.

SCAN also identifies the relations among execution trace segments. These relations provide a high-level presentation of the concepts implemented in an execution trace. The latter allows developers to understand the execution trace content by discovering commonalities between segments. Results show also that SCAN identifies relations among segments with an overall precision greater than 75% in the majority of the programs.

Finally, we evaluate the usefulness of the automatic segmentation of execution traces and assigning labels in the context of concept location. We show that SCAN support concept location tasks if used as a standalone technique. The obtained results guide developers on segments that implement the concepts to maintain and thus reduce the number of methods to analyse.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ABBREVATIONS

AI	**A**rtificial **I**ntelligence
ASDG	**A**bstract **S**ystem **D**ependency **G**raph
DP	**D**ynamic **P**rogramming
FCA	**F**ormal **C**oncept **A**nalysis
GA	**G**enetic **A**lgorithm
GUI	**G**raphical **U**ser **I**nterface
HPC	**H**igh **P**erformance **C**omputing
IDE	**I**nteractive **D**evelopment **E**nvironment
IDF	**I**nverse **D**ocument **F**requency
IR	**I**nformation **R**etrieval
LSI	**L**atent **S**emantic **I**ndexing
LDA	**L**atent **D**irichlet **A**llocation
NLP	**N**atural **L**anguage **P**rocessing
RLE	**R**un **L**ength **E**ncoding
SBSE	**S**earch **B**ased **S**oftware **E**ngineering
SVD	**S**ingular **V**alue **D**ecomposition
TF	**T**erm **F**requency
TWI	**T**wo **W**ay **I**mpact
WTWI	**W**eighted **T**wo **W**ay **I**mpact

CHAPTER 1

Introduction

Program comprehension is an important preliminary activity that may require half of the effort devoted to software maintenance and evolution (Dehaghani et Hajrahimi, 2013). The first step to understand a program is to identify which concept this program implements. Concept location is an important task during program comprehension. Several researchers proposed concept location approaches from execution traces. These approaches used different techniques to locate concept in source code and–or execution traces, e.g., Antoniol and Guéhéneuc (2006) proposed an epidemiological metaphor to analyse source code; Poshyvanyk *et al.* (2007) used latent-semantic indexing (LSI) to locate concept in source code and execution traces; Rohatgi *et al.* (2008) used graph dependency ranking on static and dynamic data; Asadi *et al.* (2010a; 2010b) proposed a hybrid approach to identify the concepts by segmenting exection trace based on a genetic algorithm; Pirzadeh and Hamou-Lhadj (2011) studied psychology laws describing how the human brain groups similar methods in execution traces; Shafiee (2013) introduced a new trace visualisation technique to represent the execution phases invoked in a trace. The proposed approaches cannot gives a global idea of the concept implemented in a segment by providing labels (i.e., a set of terms) describing the relevant information of a segment. None of these approaches helps developers to understand execution traces by identifying the relations between execution trace segments.

We believe that extracting concepts from execution traces helps developers by reducing the number of methods that they must investigate using trace segments compared to analysing the entire trace during maintenance tasks.

Indeed, developers generally must understand some segments of the trace that implement concepts of interest rather than to analyse in-depth the entire execution trace. Extracting concepts from execution traces facilitates maintenance tasks by guiding developers to segments that implement the concepts to maintain. Our conjecture is that a high-level presentation of the concepts implemented in an execution trace allows developers to understand the execution trace content.

In this dissertation, to reduce the complexity of analysing execution traces, we automatically split them into meaningful segments, each representing a concept. A segment is defined as a set of successive method calls, i.e., a part of the execution trace. A concept is implemented by one or more segments. Then, we propose SCAN, an approach to assign labels to the identified segments. The labels allow developers to have an idea of the concept implemented by the segment and help them to identify the segments implementing the concept to maintain. We identify the relations among trace segments to provide a high-level presentation of the concepts implemented in an execution trace and to allow developers to understand the concepts implemented in the execution trace.

This thesis aims to identify concepts and facilitate the analysis of large execution traces for maintenance tasks. The proposed techniques in this thesis (1) provide developers with trace segments composed of method calls; (2) to assign labels to the identified segments representing the most important terms that characterise the concepts of the segments; (3) to find the relations among the segments to give developers a high-level view of the execution traces. Finally, we show the usefulness of the proposed techniques in practice.

The organisation of this chapter is as follows: We present first the definitions of the main notions used in this dissertation in Section 1.1. Then, we present the motivations of our dissertation in Section 1.2. We present the contributions of the dissertation in Section 1.3. Finally, in Section 1.4, we describe the organisation of the rest of the dissertation.

1.1 Definitions

In this section, we present the basic definitions used in this dissertation. Concept location aims at identifying concepts and locating them within code regions or, more generally, into segments (Kozaczynski *et al.*, 1992; Biggerstaff *et al.*, 1993). A feature is defined as a user-observable functionnality of the program and hence a concept is more general than a feature (e.g., "Draw circle" is a feature of JHotDraw program) (Dapeng *et al.*, 2007). A concept represents a functionality of a program that is accessible to developers (e.g., "Save automatically a circle" is a concept of JHotDraw program) (Biggerstaff *et al.*, 1993). The distinction between a concept and a feature is often unclear and a concept is sometimes referred to a feature in the litterature (Chen et Rajlich, 2000; Asadi *et al.*, 2010b; Poshyvanyk *et al.*, 2013). In this thesis, we use the term concept. A functionality is an operation provided by a program and accessible to the user. Source code is a set of instructions written by a developer using a computer language. A failure is the action exercising an unwanted funtionnality of a program. An execution trace is represented as a sequence of methods called during the execution of a scenario. Execution traces are generally very large and the called methods likely relate to multiple concepts of the program. We define a segment as a set of successive method calls representing a concept, i.e., a portion of an execution trace. A concept is implemented by one or more segments. The label of a segment is a set of words describing the possible concept activated by the segment. Two or more segments implementing the same concept are part of the same phase such as the segments S1 and S6 in Figure 1.1 implements the same concept C1 and then are part of the same phase P1. A phase implements a concept and thus represents one or more segments. Finally, we abstract repeated sequences of phases (e.g., P1, P2 and P3 in Figure 1.1) into macro-phases (e.g., Marco-Phase1 in Figure 1.1). A macro-phase implements a set of concepts (e.g., C1, C2, and C3).

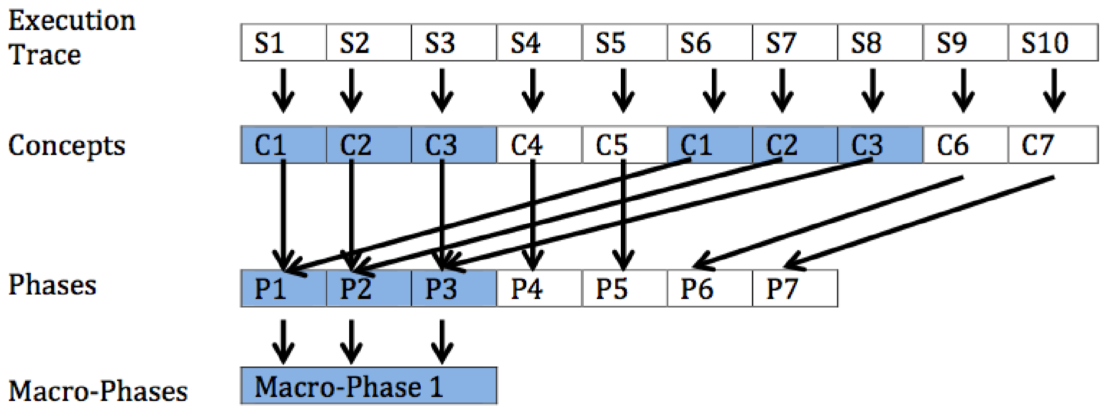


Figure 1.1 Overview of segments relations.

1.2 Motivation and Problem Statement

A typical scenario in which concept location takes part is as follows:

1 Let us suppose that a failure has been observed in a program under certain execution conditions

2 Unfortunately, such execution conditions are hard to reproduce

3 But one execution trace was saved during such a failure.

Developers then face the difficult and demanding task of analysing the one execution trace of the program to identify in the trace the set(s) of methods producing the unwanted functionality.

Some approaches (Wilde *et al.*, 1992; Wilde et Scully, 1995) rely on multiple execution traces of a program but sometimes only one trace is available. For example, when a bug occurs, it may be difficult to reproduce the same execution scenario. For this reason, we are interested to identify concepts using one or possibly more execution traces.

Cornelissen *et al.* (Cornelissen *et al.*, 2009) present a systematic survey of 176 articles from the last decade on program comprehension through dynamic analysis. They found the first article on program comprehension through dynamic analysis dates back to 1972 where

Biermann builds finite state machines from execution traces (Biermann, 1972). Despite the advantages of dynamic analysis approaches, there are also known drawbacks, one of which is *scalability* (Cornelissen *et al.*, 2009): "The scalability of dynamic analysis due to the large amounts of data that may be introduced in dynamic analysis, affecting performance, storage, and the cognitive load humans can deal with." Indeed, execution traces are a precious source of information but they can be overly large and noisy. For example, the trace corresponding to the simple scenario "Draw a rectangle" in JHotDraw v5.1 contains almost 3,000 method calls. Hence, the problem is that execution traces might not be of immediate support to developers for identifying the set of methods to maintain. To address scalability issues, some approaches propose to compact execution traces (e.g., (Reiss et Renieris, 2001), (Hamou-Lhadj et Lethbridge, 2006)), build high-level behavioral models (e.g., (Hamou-Lhadj *et al.*, 2005), (Safyallah et Sartipi, 2006)), extract dynamic slices (e.g., (Agrawal *et al.*, 1993), (Zhang *et al.*, 2003)), and segment execution traces (e.g., (Asadi *et al.*, 2010b), (Pirzadeh et Hamou-Lhadj, 2011)). None of the proposed approaches guide developers towards segments that implements the concepts to maintain by labeling such segments and identifying relations between segments.

Furthermore, concept location approaches typically use static and–or dynamic information extracted from the source code of a program or from some execution traces to relate method calls to concepts. Both static and dynamic approaches have some limitations. Dynamic approaches most often rely on multiple executions of the programs, i.e., multiple execution traces. Static approaches can rarely identify methods contributing to a specific execution scenario. Recent works focus on hybrid approaches integrating static and dynamic information to improve the performance in time as well as precision and recall of the concept location process (Antoniol et Guéhéneuc, 2005; Poshyvanyk *et al.*, 2007; Rohatgi *et al.*, 2008; Asadi *et al.*, 2010b). Thus, we focus on a hybrid approach to identify concepts.

1.3 Research Contributions

The contributions of this thesis are as follows:

– An execution traces segmentation approach that splits execution traces into segments using conceptual cohesion and coupling based on a dynamic programming (DP) algorithm. The proposed trace segmentation allows developers to focus on segments to maintain instead of analysing the entire execution trace and thus facilitate their maintenance tasks.

– Labeling execution traces segments using information retrieval techniques. The labels allow developers to understand the concept implemented by the segment and guide them towards segments implementing the concept to maintain. We investigate SCAN capability to label segments and select the most important methods of a segment.

– Identifying relations among execution traces segments using Formal Concept Analysis (FCA), which we also evaluate empirically.
– A study of the usefulness of an automatic trace segmentation and labeling in the context of concept location. We aim to assess whether SCAN supports concept location tasks if used as a standalone technique in practice.

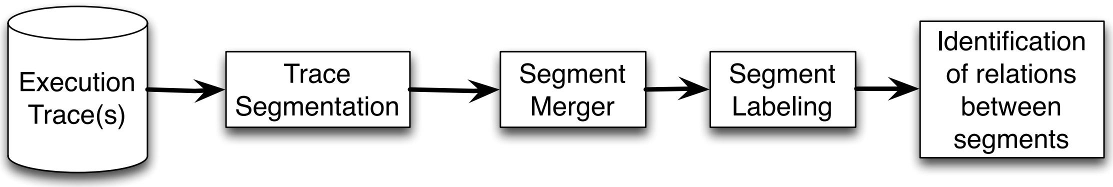


Figure 1.2 Overview of SCAN.

Figure 1.2 provides a high-level view of SCAN. It consists of the following main blocks:

1. Trace segmentation to split traces into cohesive segments.
2. Segment merger to merge similar segments using the Jaccard measure on terms extracted from the segments.
3. Segments labeling to assign labels to segments using an IR-based approach.
4. Identification of relations between segments using FCA.

1.3.1 Execution Trace Segmentation

Execution traces are very large and thus very difficult to explore and understand (Cornelissen *et al.*, 2009). In addition, developers generally are interested to understand some parts of the trace that implement the concept of interest rather than to analyse in-depth the entire execution trace. The proposed trace segmentation simplify the comprehension of large execution traces. We aim to split execution trace into cohesive and decoupled fragments of the trace. The cohesion and coupling computations are used in previous works for segments identification and we assume that using the same computations will be helpful (Asadi *et al.*, 2010b). Differently to the previous approaches based on genetic algorithms (GA), our approach can compute the exact solution to the trace segmentation problem. We use two programs, JHotDraw and ArgoUML, to show that our approach improves the accuracy of Asadi *et al.* results. Results show that our approach significantly out-performs the previous approach in terms of the optimum segmentation score vs. fitness function and the times required to produce the segmentations. We simplify the comprehension of large execution traces by representing them as execution trace segments. However, developers still must understand the concept implemented by each segment to identify the segments that implement the concepts to maintain. For this reason, we propose to assign labels to the identified segments.

1.3.2 Segments Labeling

We propose SCAN (Segment Concept AssigNer) an approach to assign labels to sequences of methods in execution traces. The assigned label provides relevant information on the concept implemented by each segment to help developers understand the concept implemented by each segment.

We perform a manual validation by one person on several traces of both JHotDraw and ArgoUML to evaluate the accuracy and effectiveness of assigning meaningful sets of words representative of the concepts implemented in segments. Results show that SCAN is successful in assigning labels very similar to manually-defined labels and that these labels actually correspond to the concepts encountered in the segment based on documentation, source code, and method execution.

A manually labeled segments by one person may bias our evaluation of labels generated by SCAN. To cope with this limitation, we perform an experiment aiming at verifying SCAN capability to select the most important methods of a segment. We ask 31 participants to label segments in the traces of six Java programs (ArgoUML, JHotDraw, Mars, Maze, Neuroph, and Pooka). To evaluate the accuracy of SCAN to label segments, we compare the labels of the trace segments generated by SCAN with respect to the labels produced by the participants. Results show the ability of SCAN to accurately reduce the size of segments and assign labels to the segments. Segments labeling provides a high-level presentation of the concepts implemented in each segment. Yet, developers would also benefit of the relations between segments in the entire execution trace to identify the segments that implement the concepts to maintain. For this reason, we propose to identify relations among trace segments.

1.3.3 Segments Relations Identification

We propose to identify the different relations among segments to provide a high-level presentation of the concepts implemented in the execution trace to developers and guide them towards the segments that implement the concepts to maintain. We identify three types of relations among segments: same phase, sub/super-phase, and macro-phase. We use formal concept analysis (FCA) to discover commonalities between segments. We perform an experiment to evaluate SCAN capability to identify relations among segments using 31 participants and six Java programs (ArgoUML, JHotDraw, Mars, Maze, Neuroph, and Pooka). Results show the ability of SCAN to accurately identify relations between segments. These results show the accuracy of SCAN, which is a prerequisite to studying its usefulness.

1.3.4 SCAN Usefulness

We study the usefulness of SCAN automatic trace segmentation and labeling in the context of performing a concept location task. We evaluate the usefulness of SCAN for two Java programs: JabRef and muCommander. We assess whether SCAN can be used to reduce the burden of developers when identifying the set of methods impacted by a concept, once a concept location technique identifies these methods. Our hypothesis is that methods related to a concept should be contained in one or few segments. Hence, to analyse the concept impact set, a developer could only focus on one or few segments instead of looking at the entire execution trace. In addition, we also want to investigate whether, instead of relying on concept location techniques, SCAN can be used as a standalone technique to automatically identify segments relevant for a query. We show that SCAN has alone the potential to be useful during concept location because it groups gold set methods in only two segments in general. We thus show that SCAN supports concept location tasks if used as a standalone technique.

1.4 Organisation of Dissertation

The rest of this dissertation is organised as follows:

Chapter 2 - Background This chapter presents a brief background of thechniques and approaches useful to understand this dissertation. First, we define the information retrieval (IR) techniques that we use in this dissertation and we briefly illustrate the IR processing with an example.

Second, we present IR performance measures, i.e., precision, recall, and Jaccard index used in this dissertation. We define the search-based optimization algorithms e.g., Genetic Algorithm (GA) and Dynamic Programming (DP) algorithm used in this dissertation. Then, we introduce Formal Concept Analysis (FCA) that we use to identify the relations among execution trace segments and illustrate the algorithm by an example. Finally, we explain the statistical tests used to assess the performances of the proposed approaches.

Chapter 3 - Related Work This chapter presents the research areas that are related to our dissertation. The chapter starts by briefly presenting state-of-the-art concept location approaches: static, dynamic and hybrid approaches. Finally, summarisation of source code works related to our dissertation are presented.

Chapter 4 - Trace Segmentation This chapter starts by introducing execution trace segments. An approach for segmenting a trace into execution segments using dynamic programming is then presented. The chapter proceeds by discussing different steps of our proposed trace segmentation approach. Evaluation of the trace segmentation approach is then presented on two Java programs: JHotDraw and ArgoUML.

Chapter 5 - Segments Labeling This chapter starts by introducing the different steps of SCAN approach to assign labels to the identified trace segments. In this chapter, we perform a qualitative as well as a quantitative analysis in which only one participant assign manual labels. The chapter proceeds by improving this evaluation by an experiment to assess SCAN capability to select the most important methods of a segment and compare the resulting labels against labels provided by 31 participants. We assess also the quality of the generated labels generated by SCAN compared to the manual labels provided by the participants.

Chapter 6 - Segments Relations This chapter starts by introducing SCAN approach to identify different relations among trace segments. In this chapter, we perform a qualitative analysis to validate the relations identified automatically by only one participant. The chapter proceeds by improving this evaluation with an experiment to assess SCAN capability to identify relations among segments in comparison to 31 participants. We verify the relations among the execution trace segments identified by SCAN compared to the relations provided by the participants.

Chapter 7 - SCAN Usefulness In this chapter we evaluate whether SCAN is useful for concept location tasks—when an issue request and an execution trace are available—in addition to concept location techniques or as an alternative. The results of the evaluation of the usefulness of SCAN for two Java programs: JabRef and muCommander are then presented.

Chapter 8 - Conclusion This chapter revisits the main contributions of this dissertation. The chapter continues by describing opportunities for future research.

Appendix A: It provides the list of publications published during my Ph.D.

CHAPTER 2

Background

This chapter provides the details of the main techniques, i.e., Information Retrieval (IR) techniques, search-based optimization algorithms, Formal Concept Analysis (FCA), and statistical tests, which are used in this dissertation. First, we briefly define IR techniques, i.e., VSM and LSI. We explain also the different steps of the IR process by using an example. Second, we present IR performance measures, i.e., precision, recall, and Jaccard index used in this dissertation. We define the search-based optimization algorithms e.g., Genetic Algorithm (GA) and Dynamic Programming (DP) algorithm used in this dissertation. Then, we explain the Formal Concept Analysis, which we use to identify relations among segments of the same execution trace and illustrate the algorithm by an example. Finally, we explain the statistical tests used to assess the performances of the proposed approaches.

2.1 IR Techniques

In this dissertation, we use some IR techniques, in particular VSM (De Lucia *et al.*, 2012; Le *et al.*, 2013) and LSI (Marcus *et al.*, 2004; Poshyvanyk *et al.*, 2007) to identify concepts in execution traces. Both techniques use term-by-document matrices. To build these matrices we choose the well-known $TF\text{-}IDF$ weighting scheme (Baeza-Yates et Ribeiro-Neto, 1999). In the following, we explain the IR techniques and weighting in details and then we present an example of IR process.

2.1.1 Vector Space Model (VSM)

Information Retrieval (IR) is a family of techniques for searching information within documents. Vector Space Model (VSM) is an IR technique used in software engineering for concept location. VSM represents documents as vectors in the space of all the terms of the documents of the corpus. In a VSM, documents are represented by a term-by-document matrix, i.e., m $\times$ n matrix, where m is the number of terms and n is the number of documents. The values of the matrix cells represent the weights of the terms in a document. After the generation of the term-by-document matrix, we calculate the similarity value between each pair of documents.

The similarity value between two documents is computed as the cosine of the angle (θ) between the two vectors (e.g., vectors D and Q) of these two documents. Cosine values are in

$[-1, 1]$ but similarity between documents cannot be negative. Thus, we discard the negative values (i.e., use zero). The similarity between two documents is calculated as:

$$Similarity(D, Q) = \frac{D \ \times Q}{||D|| \ \times ||Q||} = \frac{\sum_{t_i} w_{t_{iD}} \times w_{t_{iQ}}}{\sqrt{\sum_{t_i \in D} w_{t_{iD}}^2} \times \sqrt{\sum_{t_i \in Q} w_{t_{iQ}}^2}} \tag{2.1}$$

where t_{iD} is the weight of the ith term in the vector D, and t_{iQ} is the weight of the ith term in the vector Q. The smaller the vector angle, the higher similarity between two documents.

2.1.2 Term Weighting

To represent the documents as vectors of terms, each term is assigned a weight. Different schemes for weighting terms have been proposed in literature (Poshyvanyk *et al.*, 2007; De Lucia *et al.*, 2012). Widely used weighting schemes are based on two main factors:

– TF (Term Frequency) indicates the occurrence of the term in one document

– IDF (Inverse Document Frequency) indicates the importance of the term in the corpus.

TF is calculated as the number of occurrences of the term in the document divided by the occurrences of all the terms of the document.

$$TF_{(i,j)} = \frac{n_{(i,j)}}{\sum_k n_{(k,j)}} \tag{2.2}$$

where k is the number of terms in the document and $n_{(i,j)}$ is the number of occurrences of the term t_i in the document D_j.

TF is a term weighting, it does not include the distribution of terms through the documents. If a term appears multiple times in a single or multiple documents then the IR technique would recommend that document as relevant to a query. However, multiple occurrences of a term does not guarantees that it is an important term. For this reason, Jones (1972) proposed IDF to reduce the weight of a term that appears in several documents. The IDF of a term is computed using the following equation:

$$IDF_i = log(\frac{|N|}{|d : t_i \in D|}) \tag{2.3}$$

where $|N|$ is the number of all documents in the corpus and $d : t_i \in D$ is the number of documents that contains the term t_i.

$TF\text{-}IDF$ is computed using the following equation:

$$(TF\text{-}IDF)_{(i,j)} = TF_{(i,j)} \times IDF_i \tag{2.4}$$

where $TF_{(i,j)}$ is the term-frequency of the term t_i in the document D_j, and IDF_i is the inverse document frequency of the term t_i.

2.1.3 Latent Semantic Indexing

VSM does not address the synonymy and polysemy problems and relations among terms (Deerwester *et al.*, 1990). For example, one document having a term "car" and another document having a term "vehicle" are non-similar documents using VSM. To address this limitation, LSI identifies relations among terms and documents. The first step of LSI is to transform the corpus of documents into a term-document matrix A as explained in 2.1.1. Latent Semantic Indexing uses a mathematical technique called Singular Value Decomposition (SVD) to decompose the matrix into the product of three other matrices (Deerwester *et al.*, 1990): T, S, and V where T is the term-concept vector matrix, S is the diagonal matrix ordered by weight values and V is a concept-document matrix. The relation between the four matrices is as follows:

$$A = T \times S \times V \tag{2.5}$$

LSI orders the matrix S by size and then set to zero all the values after the first largest k value. Thus, deleting the zero rows and columns of S and corresponding columns of T and rows of V would produce the following reduced matrix:

$$A \approx A_k = T_k \times S_k \times V_k \tag{2.6}$$

where the matrix A_k is approximately equal to A. The choice of the k value is critical and still an open issue in the literature (Deerwester *et al.*, 1990; Marcus *et al.*, 2004). We should choose a k value that is large enough to accommodate all the data structures, but also small enough to discard unimportant details in the data.

2.1.4 IR Process Example

We explain the process of VSM with an example. Given the following documents:
Document 1 (D1): I love fish.
Document 2 (D2): I eat meat and I do not eat fish.
Document 3 (D3): No fish, no meat, I eat vegetables.
If we have the following query: **Query**: I do not eat fish.

A vector space model is represented as a matrix in which we have the vectors of the documents shown in Table 2.1 and the vector of the query represented in Table 2.2. The values represent the weight of each term in the documents. We present the results of the computation of weights using TF and $TF\text{-}IDF$. We want to produce the vector space model for these documents. After deleting the punctuations from the documents, the terms of our vector space model are: <I>, <love>, <fish>, <eat>, <meat>, <do>, <not>, <no>, <vegetables>.

Table 2.1 Document-Term Matrix of the example

	I	**love**	**fish**	**eat**	**meat**	**do**	**not**	**no**	**vegetables**
D1	1	1	1	0	0	0	0	0	0
D2	2	0	1	2	1	1	1	0	0
D3	1	0	1	1	1	0	0	2	1

Each row of Table 2.1 represent a vector of weights of the terms in the corresponding document D$_i$. The weight represent the number of occurrences of the term in the document. We construct the vector of the query as shown in Table 2.2.

Table 2.2 Query-Term Vector of the example

	I	**love**	**fish**	**eat**	**meat**	**do**	**not**	**no**	**vegetables**
Query	1	0	1	1	0	1	1	0	0

Tables 2.3 and 2.4 presents the results of the computation of TF and $TF\text{-}IDF$, respectively, of all the terms for the same example. When we compare the matrices of the Tables 2.3 and 2.4, we observe that, using $TF\text{-}IDF$, the weights of the terms existing in all documents is zero as desired because in general, these terms are non significant terms for a particular document with respect to others.

To find the document satisfying the query, we calculate the similarity between each document and the query. To calculate the similarity between two documents, we calculate the cosine of the angle (θ) between the two vectors (vectors d_i and vector q) of these two documents as defined in Equation 2.1. The similarity values between the documents and the query are presented in Table 2.5 based on TF and $TF\text{-}IDF$ weights where the list of the documents is sorted by similarity values. Document D2 is the most similar document to the query while document D1 is the least similar to the query. Because $TF\text{-}IDF$ reduces the weight of the terms <I> and <fish> that exists in all the documents, it increases the similarity between document D1 and the query but decreases the similarity between the documents D3 and the query.

Table 2.3 TF Matrix of the example

	I	**love**	**fish**	**eat**	**meat**	**do**	**not**	**no**	**vegetables**
D1	1/3	1/3	1/3	0	0	0	0	0	0
D2	2/8	0	1/8	2/8	1/8	1/8	1/8	0	0
D3	1/7	0	1/7	1/7	1/7	0	0	2/7	1/7

Table 2.4 Document-Term Matrix using TF-IDF of the example

	I	**love**	**fish**	**eat**	**meat**	**do**	**not**	**no**	**vegetables**
D1	0	0.16	0	0	0	0	0	0	0
D2	0	0	0	0.04	0.02	0.06	0.02	0	0
D3	0	0	0	0.02	0.02	0	0	0.5	0.07

Table 2.5 Sorted relevant documents using TF and TF-IDF weights.

	TF	TF-IDF
D2	0.9	0.98
D3	0.75	0.08
D1	0.52	0

2.2 IR Performance Measures

In the following, we describe the metrics used to compute the accuracy of our results.

2.2.1 False Positives and Negatives

We use the numbers of false positives and false negatives to evaluate the accuracy of the generated labels (i.e., set of words describing the concept). False positive is defined as number of relevant terms not retrieved and false negative represents the number of retrieved terms that are not relevant.

2.2.2 Jaccard Index, Precision, and Recall

The Jaccard index is used to compare the similarity and diversity of sets of segments. It has values in the interval $[0, 1]$:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The Jaccard index is defined as the size of the intersection between two sets A and B devided by the size of the union of A and B. We use Jaccard similarity to calculate the overlap between segments.

We use also two well-known IR metrics, precision and recall, to evaluate the accuracy of our results. Both measures have values in the interval $[0, 1]$. Precision is defined as the number of relevant documents retrieved divided by the number of retrieved documents by an approach. If all the recovered documents are correct, the precision value will be 1. Recall is defined as the relevant documents retrieved divided by the number of relevant documents. If all the relevant documents are retrieved, the recall value will be 1.

$$Precision = \frac{|\ \{\text{relevant documents}\} \cap \{\text{retrieved documents}\}\ |}{|\ \{\text{retrieved documents}\}\ |}$$

$$Recall = \frac{|\ \{\text{relevant documents}\} \cap \{\text{retrieved documents}\}\ |}{|\ \{\text{relevant documents}\}\ |}$$

2.2.3 F-Measure

The precision and recall are two independent metrics to measure two different accuracy concepts. F-measure is a summary measure of precision and recall and is used to compare the results of different approaches.

F-measure is computed as:

$$F = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

2.3 Search-based Optimization Algorithms

In the following, we define the search-based optimization algorithms, e.g., Genetic Algorithm (GA) and Dynamic Programming (DP) algorithm used in this dissertation.

2.3.1 Genetic Algorithm

Genetic algorithms are evolutionary algorithms. In a genetic algorithm, an initial population of individuals that are generated randomly to an optimization problem is evolved toward better solutions. The number of individuals of the initial population depends on the nature of the problem and is the same for all generations. In each iteration, some of the individuals of the current generation are selected (parents) to generate the next generation. The selected parents are chosen using a predefined fitness function. The fitness function selects good quality parents because the parents have more chance to produce good quality descendants according to the fitness function. The selected parents are modified to form a new generation using two genetic operators: crossover and mutation. The new generation is expected to increase the average fitness. The generation process continues until reaching a terminating critera. A very low diversity of the solutions or a maximum number of iterations can be considered as a stopping criteria.

Crossover

Crossover is a genetic operator that takes two parents and produce two new chromosomes (offsprings). Several types of crossover exist, such as one-point, two-point, and uniform. The one-point crossover operator selects randomly one split point within the parents chromosomes and splits them at this point. Then, two offsprings are generated by exchanging the tails of the parents chromosomes. An example of one-point crossover is presented in Figure 2.1.

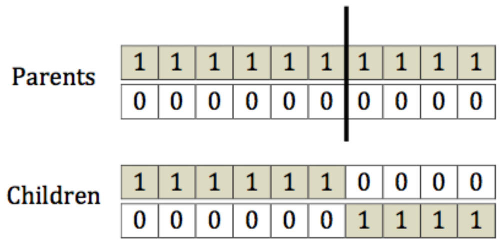


Figure 2.1 One-Point Crossover Example.

The two-point crossover operator selects randomly two split points within the parents chromosomes and splits them at these points. Then, two offsprings are generated by exchanging the genes located between these two split points. An example of two-point crossover is presented in Figure 2.2.

Finally, the uniform crossover operator mixes the genes of parents chromosomes with each other. For each gene of the first child, it flips a coin to decide from which parent this child

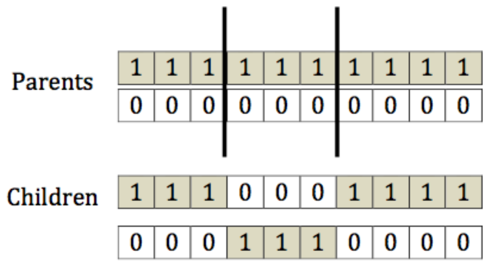


Figure 2.2 Two-Point Crossover Example.

should inherits this gene. Then, the gene of the other parent will be assigned to the second child. An example of uniform crossover is presented in Figure 2.3.

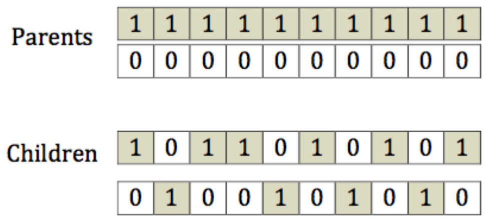


Figure 2.3 Uniform Crossover Example.

Mutation

Mutation alters some gene values to generate a new chromosome. An example of applying a mutation operator is illustrated in Figure 2.4.

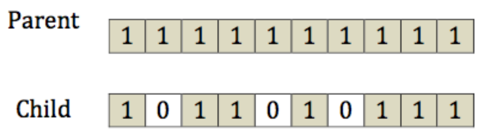


Figure 2.4 Mutation Example.

Several types of mutation exist, such as flip-bit, boundary, and uniform. A flip-bit mutation inverts the value of the selected gene, i.e., it replaces a 0 with a 1 and vice-versa. A boundary mutation replace the value of the selected gene by the lower or the upper bound defined for that gene. The change by the upper or the lower bound is chosen randomly. A uniform mutation replaces the value of a selected gene with a random value selected between the user defined lower and upper bounds for that gene.

2.3.2 Dynamic Programming Algorithm

Dynamic Programming (DP) is a technique to solve search and optimization problems with overlapping sub-problems and an optimal substructure. It is based on the divide-and-conquer strategy where a problem is divided into a set of sub-problems, recursively solved, and where the solution of the original problem is obtained by combining the solutions of the sub-problems (Bellman et Dreyfus, 1962; Cormen *et al.*, 1990). Sub-problems are overlapping if the solving of a (sub-)problem depends on the solutions of two or more other sub-problems, e.g., the computation of the Fibonacci numbers. The sequence F_n of Fibonacci numbers is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$, such as the numbers in the following integer sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...

The original problem must have a particular structure to be solved by DP. First, it must be possible to recursively break it down into sub-problems up to some elementary problem easily solved; second, it must be possible to express the solution of the original problem in term of the solutions of the sub-problems; and, third, the Bellman's principle of optimality must be applicable. Bellman's principle of optimality states that: An optimal solution has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal solution with regard to the state resulting from the first decision (Bellman et Dreyfus, 1962).

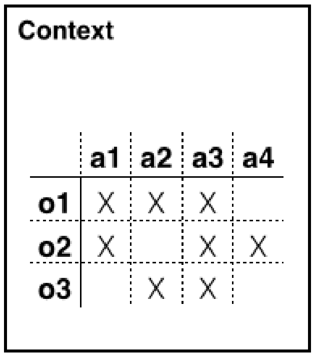
An important aspect of dynamic programming is that the solutions of sub-problems are saved to avoid computing them again later on. The solution of one sub-problem may be used multiple times to solve several larger sub-problems. Thus, for some problems, dynamic programming algorithms are more efficient than classical recursive algorithms.

2.4 Formal Concept Analysis (FCA)

Formal Concept Analysis (FCA) (Ville, 1999) is a technique to group *objects* that have common *attributes*. The starting point for FCA is a *context* (O, A, P), i.e., a set of objects O, a set of attributes A, and a binary relation among objects and attributes P, stating which attributes are possessed by which objects. A *FCA concept* is a maximal collection of objects that have common attributes, i.e., a grouping of all the objects that share a set of attributes. More formally, for a set of objects $X \subseteq O$, a set of attributes $Y \subseteq A$, and the binary relation between them P, a FCA concept is the pair (X, Y) such that:

$$X \mapsto X' = \{a \in A \mid \forall o \in X : (o, a) \in P\}$$
$$Y \mapsto Y' = \{o \in O \mid \forall a \in Y : (o, a) \in P\}$$

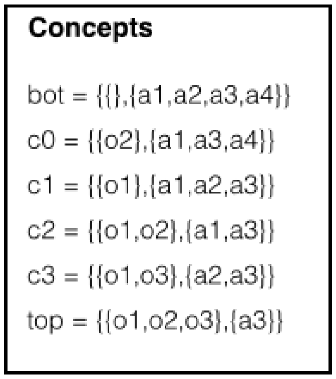
where $X' = Y$ and $Y' = X$. X' is the set of attributes common to all objects in X and Y' is the set of objects possessing all attributes in Y. X is the *extent* of the concept and Y is its *intent*. The extent X of a concept is obtained by collecting all objects reachable from (X, Y) down to the bottom node (e.g., the node *bot* in the Figure 2.7). The intent Y is obtained following the opposite direction, i.e., from (X, Y) to the top node (e.g., the node *top* in the Figure 2.7), and by collecting all reachable attributes. The concepts define a lattice.



Context

	a1	a2	a3	a4
o1	X	X	X	
o2	X		X	X
o3		X	X	

Figure 2.5 Context of the example.



Concepts

bot = {{},{a1,a2,a3,a4}}

c0 = {{o2},{a1,a3,a4}}

c1 = {{o1},{a1,a2,a3}}

c2 = {{o1,o2},{a1,a3}}

c3 = {{o1,o3},{a2,a3}}

top = {{o1,o2,o3},{a3}}

Figure 2.6 Extent and intent of the concepts of the example.

In this dissertation we use the general bottom-up algorithm described by Siff and Reps (1999). Given an example of three objects o1, o2, and o3 and four attributes a1, a2, a3, and a4, Figure 2.5 shows which objects are considered to have which attributes. The binary relations between attributes and objects are also given in Figure 2.5. For example, (o1, a2) is a binary relation but (o2, a2) is not.

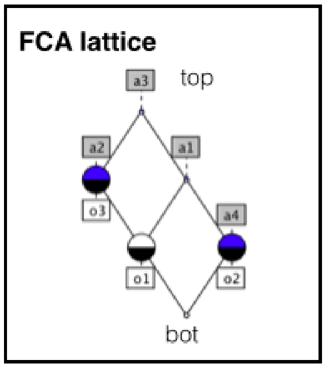


Figure 2.7 Concept lattice of the example.

To build a FCA lattice the bottom-up algorithm first computes the bottom element of the concept lattice. Next, it computes atomic concepts. Atomic concepts are the smallest concepts with an extent containing each object treated as a singleton set, such as $c0$ in Figure 2.6. Then, the algorithm closes the set of atomic concepts under join. Initially, a work-list is formed containing all pairs of atomic concepts (c', c) where $c \nsubseteq c'$ and $c' \nsubseteq c$. While the work-list is not empty, the algorithm removes the element $(c0, c1)$ from the work-list and computes $c'' = c0 \cup c1$. If c'' is a concept that is not yet discovered, then it adds all pairs of concepts (c'', c) to the work-list, where $c \nsubseteq c''$ and $c'' \nsubseteq c$. This process is repeated until the work-list is empty. The result is shown in Figure 2.6. It shows the computed concepts as well as the extent and the intent of each concept of the example. To visualize the lattice we use Concept Explorer [1] (Yevtushenko, 2000). The concept lattice of the example is shown in Figure 2.7. Each node in the lattice represents a concept. Blue (black) filled upper (lower) semi-circle indicates that an attribute (object) is attached to the concept.

2.5 Statistical Hypothesis Testing

In this dissertation, we use statistical hypothesis testing to compare the performance of an approach to another. To perform a statistical test, first, we analyse the distribution of our data to choose an appropriate statistical test. Second, we establish a null hypothesis that we want to reject. We define α the significance level of the test to reject the null hypothesis. Finally, we perform the selected statistical test to compute the probability value, i.e., p-value. We compare the p-value and α and we reject the null hypothesis if the p-value is less than α. The null hypothesis is rejected if the result value of the statistical test is below α and an alternate hypothesis is accepted. In this dissertation, we perform two statistical tests

1. http://conexp.sourceforge.net/

(Wilcoxon paired test and Permutation test). These tests assess whether the obtained results are statistically significant or not. In addition, we measure the magnitude of the differences between two approaches.

2.5.1 Statistical Tests

In the following, we present the statistical tests that we use in this dissertation.

Wilcoxon Paired Test

The Wilcoxon paired test is a non-parametric statistical hypothesis test and an alternative to the two-sample student's t-test. It evaluate whether the difference between two related samples or repeated measurements on a single sample is significantly (Wohlin *et al.*, 2000).

Permutation Test

The permutation test (Baker, 1995) is a non-parametric alternative to the two-way ANalysis Of VAriance (ANOVA). Different from ANOVA, it does not require the data to be normally distributed. It builds the data distributions and compares the distributions by computing all possible values of the statistical test while rearranging the labels (representing the various factors being considered) of the data points.

2.5.2 Effect Size Measure

Other than studying the significant differences between studied approaches, we also use effect-size measure to analyse the magnitude of the difference between two approaches.

Cliff's Delta

We compute the magnitude of the differences using the non-parametric effect-size Cliff's δ measure (Grissom et Kim, 2005), which, for dependent samples is defined as the probability that a randomly-selected member of one sample A is better than a randomly-selected member of the second sample B, minus the reverse probability. The effect size δ is considered small for $0.148 \leq \delta < 0.33$, medium for $0.33 \leq \delta < 0.474$, and large for $\delta \geq 0.474$ (Grissom et Kim, 2005).

CHAPTER 3

Related Work

In this chapter, we present existing concept location approaches related to this dissertation. Concept location approaches typically use static and–or dynamic information to relate method calls to concepts. In addition, we describe source code summarisation techniques because we summarise trace segments using labels to provide developers with a global idea of the concept implemented by each segment.

3.1 Concept Location

The litterature proposed concept location techniques using static analysis, dynamic analysis, and their combination.

3.1.1 Static Approaches

Static approaches relied on information statically collected from the program under analysis, such as a source code.

Anquetil and Lethbridge (1998) proposed an approach to locate concepts by extracting abbreviations from file names. The authors hypothesised that abbreviations extraction from file names provided high-level abstractions that could support design activities and were more helpful for developers than statement-level information. The file name was considered as an abbreviation of a concept. To verify the accuracy of the results, they decomposed manually some file names and extracted a representative abbreviation. They observed that using only file names gived poor results. For this reason, they used other sources of information: comments, identifiers, and abbreviations of English words. Adding these sources of information, they guided file names decomposition and observed better results. We shared with Anquetil and Lethbridge the use of source code but we consider a different granularity of source code: they used program files and we analyse program methods. While Anquetil and Lethbridge extracted the abbreviations from file names, they were not able to associate these abbreviations with concepts. Anquetil and Lethbridge did not group files representing a same concept while we propose a trace segmentation approach that also groups methods implementing a same concept.

Chen and Rajilich (2000) developed an approach to locate concepts using only graph dependency. They studied search scenarios for concept location using extracted Abstract

System Dependency Graph (ASDG) of the components in a program (i.e., functions and global variables). A search graph is a part of the ASDG and is composed of the components visited by a developer. The first step of the approach was to locate a starting component (e.g., main() function or a randomly chosen component or a component with similar name to the concept). Second, for each component, a developer explored the source code, dependence graph, and documentation to understand the component and to decide whether it was related or unrelated to the concept. Different strategies were used to create search graph: bottom-up and top-down. The top-down strategie expanded the search graph from the function main() until reaching the desired functionnality. The bottom-up strategie expanded the search graph through calling functions. We shared with Chen and Rajilich the use of source code lo locate concepts but our purpose and the granularity of the source code were different. Chen and Rajilich extracted dependencies using methods and variables but we analyse methods because generally for maintance tasks, a developer must change some methods in the program. While our approach identifies automatically the segments of the concepts; using Chen and Rajilich approach, a developer should choose a starting point component, guide the search graph, and check if she reached all the components of the concept.

Marcus *et al.* (2004) proposed a technique to identify the part of source code that implement a specific concept. They analysed the source code using Latent Semantic Indexing (LSI) to find semantic similarities between user queries and documents to locate concepts of interest in the source code. A document represented a declaration block, a function, or a “.h file”. Their results were compared to two other approaches: Chen and Rajilich (2000) approach and the traditional *grep*-based method. Their approach was as easy and flexible to use as grep and provided better results. In addition, they identified some components of a concept that were missed by the dependence-graph search approach (Chen et Rajlich, 2000). The proposed approach identified terms and identifiers from the source code related to a given term or a set of terms within the context of the program. Thus, they were able to automatically generate queries starting with one or more terms. The results of the automatically generated queries were comparable with the queries formulated manually by the users. As Marcus *et al.*, our approach strongly relies on the textual content of the program source code but our purpose is different. While they applied LSI to identify documents implementing a concept of interest, we use LSI to split execution trace into meaningful segments, each representing a concept.

Bassett and Kraft (2013) focused on a static Latent Dirichlet Allocation-based concept location technique. They compared the accuracy of the results of an LDA-based concept location technique using 16 weighting schemes (configurations) for structural term weighting. The purpose of this comparison was to determine whether structural term weighting could improve the performance of an LDA-based concept location technique. The results showed that some configurations of structural term weighting provide better performance than uniform term weighting. The results proved that increasing the weights of terms originating from method names tended to improve accuracy but increasing the weights of terms originating from method calls tended to decrease accuracy. Bassett and Kraft suggested to use a multiplier of eight for terms extracted from method names and a multiplier of one for terms extracted from method calls. Terms derived from method names could be given more importance than terms derived from local variable names. These results prove and justify our choice to extract labels from method names. We share also with Bassett and Kraft the use of text retrieval models to locate concepts but they applied structural term weighting (i.e., latent Dirichlet allocation) while we use latent semantic indexing (LSI) to compute execution trace segmentation.

3.1.2 Dynamic Approaches

While static approaches use only source code, dynamic approaches use one or more execution traces to locate concepts in the source code.

Wilde *et al.* (1992; 1995) used test cases to produce execution traces; concepts location was performed by comparing two sets of traces: one in which the concepts were executed and another without concepts. Using probabilistic and deterministic formulations, they identified methods that were only invoked in the execution traces that implement the concepts. The latter approach was evaluated on small C programs and a small number of scenarios. The results showed that this approach was useful to indicate the parts of source code where developers should look. They conduct a study (Wilde et Scully, 1995) with professional participants and showed that this technique could be adopted by professional developers. This approach was useful to detect the parts of source code that uniquely belong to some particular concepts but it did not guarantee to find all the source code parts that participate in implementing all concepts. In addition, the approach could not guarantee the identification all of the related artifacts for a concept. Thus, the results of this technique presented only a starting point. Same as Wilde *et al.* (1992; 1995), we use execution traces to locate concepts. However, we need at least two execution traces (i.e., with and another without the concepts of interest exercised) to apply Wilde *et al.* approach while our approach is suitable if only one execution trace is available.

Eisenbarth *et al.* (2001b; 2001a) presented a technique for generating concept component maps using dynamic analysis. They applied concept analysis to reveal relations between concepts and components as well as to concept relations to a set of execution traces implementing different scenarios. Given execution traces resulting from various usage scenarios (covering a set of concepts of a program), Eisenbarth *et al.* applied concept analysis to reveal relations between concepts and components. Based on the resulting concept lattice, a concept component map was derived and used to identify components of interest given a particular concept. They performed a case study on C programs to show that their approach generated concept component maps and grouped related concepts in the concept lattice. As Eisenbarth *et al.*, we apply concept analysis to discover commonalities among concepts but they considered functions and procedures names as objects and we use terms in methods signatures as objects. Our work is close to this work but using their approach a developer must analyse a concept lattice to discover relations while our approach identifies automatically relations. A large concept lattice would be difficult to analyse to identify concept relations and thus, using Eisenbarth *et al.* approach, a developer would hardly make the effort to discover these relations. In addition, to obtain these relations using their approach, a set of scenarios must be prepared where each scenario exploit preferably only one relevant concept. However, using our approach, only one scenario is needed. Finally, their technique was not suited for concept that are only internally visible. Internal concepts (i.e., non user-observable) could only be detected by looking at the source, because it is not clear how to invoke them from the program and how to derive from an execution trace whether these concepts are present or not. Our approach is suitable for any type of concept (user-observable or not).

Hamou-Lhadj *et al.* (2002) proposed an approach to remove the repeated instances of patterns of events from a trace and keep only one of such instance. First, they filter contiguous repetitions of events and then find and filter non-contiguous repetitions. Thus, they generate a compressed trace without repetitions. Hamou-Lhadj *et al.* (2006) extended their trace summarisation approach by measuring whether an event is considered an utility event or not. The proposed metric ranks the system components according to whether they implement key system concepts or are related to implementation details. Their trace summarisation approach is based on the removal of implementation details such as utilities from execution traces. It also removes also constructors, destructors, accessers, nested classes, and methods related to programming languages libraries because they do not implement key system concepts. According to the developers of the studied system, the generated summaries of the execution traces are adequate high-level representations of the main interactions of the traces. Hamou-Lhadj *et al.* (2002; 2006) proposed approaches to generate a summary of execution traces by removing utility events. In contrast, we propose to split a trace into cohesive segments and then extract relevant information from each segment to assign meaningful labels to each segment.

Eisenberg (2005) proposed a concept location approach by using test cases. Eisenberg developed a tool to locate concepts in three steps. First, a developer must devise a test suite.

The developer provided the tool with concepts mapping that, as much as possible, grouped all test cases that collectively and comprehensively exhibited all parts of the concepts of interest; this is the exhibiting test set. Next, the tool performed execution trace analysis and produced execution trace by executing the exhibiting test set. Then, for each test execution, all method calls and their call depth were stored by the tool. Each method caller and callee of the execution trace was extracted and ranked based on their relevance to the concepts. The rank was based on three heuristics: multiplicity, specialisation, and depth. Multiplicity was the score of the number of times a method is called in one test compared to the number of times it is exercised in all other test sets. Specialisation represented how much a method was only executed by some concepts and no others. A method exercised by many different test sets was considered as likely unrelated to any concept (i.e., utility method). Depth gived how a test set exhibits directly some concepts compared to all other test sets. Eisenberg expected that, for a well-designed and well-partitioned test suite, a test set will exhibit the behavior of the conepts of interest in the most direct manner. They associated directness with call depth. Eisenberg presented a study to show that the approach could be useful for concept location. Same as Eisenberg (2005), we use execution traces to locate concepts but our aim is different, i.e., identify cohesive segments relevant for a concept rather than to identify single methods related to a concept. In addition, we require a set of execution traces (e.g., exhibiting test set) to apply Eisenberg approach but our approach is applicable even if only one execution trace is available.

Safyallah and Sartipi (2006) introduced an approach applying a data mining technique on the execution traces. This approach analyses a set of execution traces collected from a set of concept-specific scenarios using sequential pattern mining. They identified execution patterns, which were frequent continuous fragments of execution traces. A frequently occurring sequence of transactions (i.e., a pattern) was defined as a sequence supported by a user-specified minimum number of customer-sequences (i.e., MinSupport of this pattern). The results of this approach were a set of execution patterns corresponding to a given concept. They performed a case study on a medium size C program and showed that they identified the methods specific to a given concept. This technique identified the relevant parts of the execution trace to reduce the complexity to analyse large execution traces. They extended their approach (Sartipi et Safyallah, 2010) using concept lattice to locate the common group of functions of the execution traces. The advantage of using the concept lattice was to identifying the related concepts in the source code. They evaluated their approach on two C program and showed that they identified the methods specific to a given concept as well as common functions. Same as Safyallah and Sartipi (2010; 2006), we use concept lattice however our purpose is different. While, they used it to locate the common group of functions of the execution traces, we use it to identify relations among execution trace segments and locate the common concepts of the execution traces.

Kuhn *et al.* (2006) drew an analogy between dynamic analysis and signal processing. They proposed an approach to reduce the huge volume of data in traces by grouping sequences of events. First, they transformed method calls into time series by representing the nesting levels of the calls by points. Then, they grouped sequences of events based on the amounts of changes in their nesting levels. Finally, they applied several filters, such as a minimal nesting level threshold. Their approach also allows to visualise large numbers of calls in multiple traces on a single screen. It identifies the similarity between trace signals and arranges them accordingly on the screen. It thus summarises traces by reducing the length of the trace signal by 50% to 90% while preserving relevant information. Their approach displays the different groups of sequences of events in traces but it do not present the concepts implemented by each group. In contrast, we propose to split a trace into cohesive segments and to assign labels to describe the concepts implemented by each segment.

Cornelissen *et al.* (2008) proposed an assessment methodology to evaluate and compare trace reduction techniques. The authors selected four trace abstraction techniques found in the literature, which they evaluated and compared using a test set of seven large execution traces. They proposed an assessment methodology based on a common context, common evaluation criteria, and a common test set to ensure that these techniques could be properly compared. They found that the abstraction techniques performed well in terms of reducing trace size but were less useful in preserving high-level and medium-level information contained in the execution traces. Thus, the abstracted traces in their study were not representative of the original traces. The main challenge when reducing the volume of data of execution traces is to preserve relevant information. To overcome this limitation, in this dissertation, we present an approach to reduce the size of segments based on *TF-IDF* and we show that the reduced segments are representative of the original segments.

Our work is close to the work of Pirzadeh and Hamou-Lhadj (2011) who divided execution traces into segments corresponding to the program's main execution phases (e.g., initializing variables, performing a specific computation, etc.). Their algorithm for phase detection was inspired by psychology laws describing how the human brain grouped similar items. Potential execution phases were identified by applying the similarity and continuity gravitational schemes. The similarity scheme reduced the distance between same method calls, where distance was defined using a mapping from the execution order of the method calls to an interval scale, i.e., ruler distance. The continuity scheme reduced the distance between method calls in higher nested levels and the previous method calls. Applying the schemes may result in a rearrangement of the methods calls compared to the original trace, i.e., the order of execution was not preserved. Pirzadeh and Hamou-Lhadj then used the K-means clustering algorithm to group potential phases thus identifying the execution phases. In contrast with this work of Pirzadeh and Hamou-Lhadj (2011), SCAN preserves the order of method calls when segmenting an execution trace.

Pirzadeh *et al.* (2011b) proposed a trace sampling framework. They used stratified sampling to obtain traces of reduced size with respect to the original trace by distributing the desired characteristics of an execution trace similarly in both the sampled and the original trace. They used random sampling techniques to generate sampled execution traces. However, random sampling may generate samples that are not representative of the original trace. They extended their approach (Pirzadeh *et al.*, 2011a) by extracting higher-level views that characterise the relevant information about execution traces. They proposed a technique called content prioritisation to weight the trace elements of each phase and keep the most representative elements of a phase. The first step of the proposed approach is to remove utility methods from the execution phases to reduce the noise in the data. Second, they applied a weighting function to weigh methods of a phase according to their relevance. The higher the weight, the more representative the method. They selected the most representative methods in a phase from the obtained list of ranked methods. Pirzadeh *et al.* approach (2011a) extracted higher-level views that characterise the relevant information of execution segments in terms of method calls while we characterise execution segments by relevant terms. While they also used the weighted elements to detect similar phases, we use FCA to identify the relations among trace segments based on the extracted terms.

3.1.3 Hybrid Approaches

Both static and dynamic techniques have some limitations. Dynamic techniques most often rely on multiple executions of the programs, i.e., multiple execution traces. Execution traces are often very large, cumbersome to manipulate and furthermore they may contain uninteresting events from the developers' point of view. Static analyses can rarely identify methods contributing to a specific execution scenario. They often fail to properly capture a program behavior. Hybrid approaches have been introduced to overcome the limitations of dynamic and static approaches. For these reasons, researches developed hybrid approaches to identify concepts.

Antoniol and Guéhéneuc (2005) presented a hybrid approach for concept location and reported its results on real-life, large, object-oriented, multi-threaded programs. They used knowledge filtering and probabilistic ranking to overcome the difficulties of getting rid of uninteresting events. The approach was improved (Antoniol et Guéhéneuc, 2006) by using the notion of disease epidemiology. Antoniol and Guéhéneuc (2006) proposed an epidemiological metaphor to analyse source code to assist program understanding tasks in large multithreaded object-oriented programs. They identified the microarchitectures implementing some concepts of interest and highlighted the variables, classes, functions, and methods activated when exercising a concept. They produced also a ranked list of methods participating in a concept to support maintenance and program understanding tasks. They used case studies to assessed the usefulness of their approach. They compared their process of concept location with previous works by Eisenbarth *et al.* (2003) and showed that their epidemiological metaphor dramatically decreased the number of methods in comparison to Eisenbarth *et al.* works and the size of the microarchitecture implementing the concepts of interest. We share with Antoniol and Guéhéneuc the use of static (i.e., source code) and dynamic (i.e., execution traces) analysis but our purpose is different. While Antoniol and Guéhéneuc proposed an approach to identify the microarchitectures implementing some concepts of interest and to rank methods of the source code based on their similarity to the concepts of interest, we propose an approach to identify the trace segments implementing the concepts of interest.

Liu *et al.* (2007) proposed concept location technique named SIngle Trace and Information Retrieval (SITIR). They combined the information from the execution trace and from the identifiers and comments of the source code. Concept location using SITIR required little domain or program specific knowledge. It consisted of four steps: formulating and executing a single scenario, formulating the query, ranking the executed methods, and examining the results. The first step of concept location using SITIR is to formulate a scenario that capture the concept of interest. Using SITIR, we should have a single execution trace execising one concept. Second, a developer should selected a set of terms describing the concept to find. Given this set of terms and an execution trace, the approach proposed by Liu *et al.* ranked methods of the source code that appear in the execution trace based on their textual similarity with this set of words. They hypothesised that once the starting point of the modification was known, the developer could identify the other methods that would be impacted by any change related to the concept of interest. They presented two different case studies. In the first case study, they used SITIR to locate three concepts associated with change requests. In the second case study, they compared the results obtained by SITIR with two other approaches, namely Probabilistic Ranking Of Methods based on Execution Scenarios and Information Retrieval (PROMESIR) (Poshyvanyk *et al.*, 2007) and Scenario-based Probabilistic Ranking (SPR) (Antoniol et Guéhéneuc, 2006). The results of the first case study showed that, in most cases, the relevant methods to the located concepts were ranked in the top ten. The results of the second case study confirmed that SITIR outperformed LSI and SPR in locating bug related concepts and were very close to PROMESIR results. We share with Liu *et al.* the combination of dynamic analysis (i.e., execution trace) and textual analysis based on Latent

Semantic Indexing (LSI) but our goal is different. While Liu *et al.* proposed an approach to rank methods of the source code that appear in the execution trace based on their textual similarity with the change request, we propose an approach to identify the relevant segments.

Rohatgi *et al.* (2008) presented a hybrid approach for concept location. They used dynamic analysis to generate an execution trace by exercising a concept of interest. Then, they proposed two impact analysis metrics to rank the extracted classes. Both metrics were based on a static class-dependency graph built from the classes invoked in the execution trace. They ranked the classes based on identifying the impact of a class modification on the rest of the program. The first proposed metric, TWI (Two Way Impact), showed very good results and the second proposed metric, WTWI (Weighted Two Way Impact), improved the results of the first metric by adding information from the program architecture. The ranking mechanism guided developers to locate methods implementing a concept of interest, without the need for a deep understanding of the program. We share with Rohatgi *et al.* the analysis of source code and execution traces but our purpose is different. While Rohatgi *et al.* proposed an approach to rank methods by relevance to the concept of interest, we propose approaches to split the execution trace into segments, assign label to each segment describing the implemented concept, and identify relations among trace segments.

Asadi *et al.* (2010b) presented a concept location approach using genetic algorithm. They identified concepts by finding cohesive and decoupled fragments in a trace. They used a textual analysis of the source code using LSI. Although, they found that genetic algorithm identified concepts with high precision, the fitness function of their approach has a polynomial evaluation cost and was computationally intensive. A run of their approach on a trace of thousands of methods may require several hours of computation on a standard PC. Consequently, they extended their work (Asadi *et al.*, 2010a) to reduce computation times by parallelising the genetic algorithm over a standard network. They developed four distributed architectures and compared their performances. Although they decreased of computation time up to 140 times, their approach was still taking hours for some execution traces. Furthermore, it was based on metaheuristic search and thus each run may produced a different concept assignment. In this dissertation, we propose an approach using dynamic programming to overcome the limitations of Asadi *et al.* approach.

Poshyvanyk *et al.* (2013) used Formal Concept Analysis (FCA) and LSI to locate concepts in source code corresponding to a textual description, e.g., description of a concept or a bug report. Given a query, i.e., a summary of the description, source code elements are ranked using LSI. From the top-most elements in the ranked list, the top-most descriptive terms are selected and a concept lattice is built. As Poshyvanyk *et al.* did, our approach uses LSI and FCA. It also uses both static and dynamic information whereas the previous authors used static information only. When building FCA lattices, we operated at a higher level of abstraction, i.e., in our case objects are segments of execution traces rather than methods. This is because our aim is different, i.e., identify cohesive segments relevant for a concept rather than to identify single methods related to a concept. Finally, we used relations among FCA concepts in the lattice to automatically identify relations among segments.

3.2 Source Code Summarisation

Software artifact summarisation consists of techniques extracting short descriptions from software artifacts to help developers during program comprehension. There are different approaches to summarise source code. Some of them use heuristics to extract structured or natural-language summaries (Sridhara *et al.*, 2010, 2011a) whereas others use IR techniques to extract relevant keywords representing program artifacts (Haiduc *et al.*, 2010a,b; De Lucia *et al.*, 2012).

Sridhara *et al.* (2010) proposed a novel technique to automatically generate comments for Java methods. They used the signature and the body of a method to generate a descriptive natural language summary of the method. After the automatic generation of a summary, developers verified the accuracy of the generated summary with the method source code. They judged that the generated summaries were accurate and reasonably concise. The work was extended (Sridhara *et al.*, 2011a) by using a classification of code into fragments, to generate a natural language description of "actions" related to each fragment. The authors identified three types of fragments: sequence fragments, conditional fragments and loop fragments. Haiduc *et al.* (2010a; 2010b) applied and combined several automatic summarisation techniques. In a reported case study, they found that a combination of techniques making use of the position of terms in a program and traceability recovery techniques capture the meaning of methods and classes better than any other of the studied techniques. In addition, an experiment conducted with four developers revealed that the summaries produced using this combination are accurate, reasonably concise, and do not miss important information. We share with Sridhara *et al.* and Haiduc *et al.* the goal of representing artifacts with shorter descriptions. The main difference between labeling source code artifacts and labeling execution traces is that execution traces must be (1) pruned, else utility methods and event handlers dominate the creation of labels thus producing meaningless labels, and (2) segmented, else labels would not be meaningful to developers.

De Lucia *et al.* (2012) experimented the use of different IR techniques to extract keywords from source code artifacts. They compared the labeling obtained using a Vector Space Model (VSM) and TF or $TF\text{-}IDF$ weighting schemes with those of more complex techniques, such as LSI. They used a manual labeling performed by 17 students as oracle against which to compare the various techniques. They found that simpler indexing techniques, such as VSM, outperform LSI, whose results were better only for larger artifacts in which LSI clustering capabilities help to reduce the noise. As De Lucia *et al.* (2012), we use LSI to compute the exact splitting of execution trace and we use a VSM (properly complemented with some heuristics to remove noise) to label execution trace segments.

Wang *et al.* (2014) proposed an approach that automatically segments source code methods into meaningful blocks for the purpose of automatic blank line insertion . The approach used the program structure and identifiers to identify consecutive statements that logically implemented a high-level action. Examples of meaningful blocks were a sequence of statement belonging to the same syntactic category (e.g., method call, variable declaration), a sequence of statements related through data flow, and a sequence of statements grouped in a while loop including the immediately preceding statements initializing variables that control the condition. Wang *et al.* also defined a statement-pair similarity measure to segment syntactical blocks; the measure is based on the program identifiers—the use of words that constitute them and naming conventions. Syntactical blocks were defined with a sliding window of three statements. For each three consecutive statements, there were a segmentation if the similarity between the first two and the last two statements was different. Wang *et al.* studied how developers inserted blank lines to define heuristics that automatically mimiced their behavior. We cannot use a similar approach as execution traces are not manually written. In contrast to the sliding window used by Wang *et al.*, we start with one method and we keep adding the following methods one by one as long as the fitness function is improved; we segment when adding a statement decreases the value of the fitness function.

3.3 Summary

Developers generally are interested to understand some parts of the trace that implement concepts of interest rather than to analyse in-depth the entire execution trace. For this reason, several approaches tried to split execution traces into segments (Asadi *et al.*, 2010a,b; Pirzadeh et Hamou-Lhadj, 2011). Asadi *et al.* (2010b; 2010a) identified concepts by finding cohesive and decoupled fragments in a trace using genetic algorithm. Although they found that genetic algorithm identify concepts with high precision, their approach is computationally intensive and could not be applied to traces of thousands of methods. Furthermore, it was based on metaheuristic search and thus each run could produce a different concept assignment. To address these limitations, our trace segmentation approach directly extends Asadi *et al.* works (2010b; 2010a) by reformulating the trace segmentation problem as a DP problem. We split the execution trace into segments similarly to what also Pirzadeh and Hamou-Lhadj (2011) did, although based on a different approach, incorporating conceptual similarity in a search-based optimization technique (Asadi *et al.*, 2010b). Pirzadeh and Hamou-Lhadj (2011) divided execution traces into segments corresponding to the program's main execution phases. They only considered exact naming when evaluating the similarity between methods and they did not take into account other types of information, such as the source code of methods.

In addition, the extraction of the essence of the information of the segments is helpful to developers to understand the concept implemented by the segment. For this reason, several approaches characterised execution trace segments with relevant information (Pirzadeh *et al.*, 2011b,a), extracted structured or natural-language summaries (Sridhara *et al.*, 2010, 2011a) for program artifacts, and others labeled program artifacts (Haiduc *et al.*, 2010a,b; De Lucia *et al.*, 2012). However, none of the proposed approaches label trace segments. Labeling segments allow developers to have an idea of the concepts implemented by the segments and guide them towards segments implementing the concepts to maintain.

Our work is close to Eisenbarth *et al.* (2001b; 2001a) works, which presented semi-automatic approaches in which a developer should analyse a concept lattice to discover relations among concepts but our approach identify automatically relations. In addition, their technique was not suited for concepts that are only internally visible while our approach is suitable for any type of concepts (user-observable or not). The relations among trace segments provide a high-level presentation of the concepts implemented in an execution trace and allow developers to understand the concepts implemented in the execution trace.

CHAPTER 4

Trace Segmentation

Developers generally are interested to understand some parts of the trace that implement the concepts of interest rather than to analyse in-depth the entire execution trace. To reduce the complexity of analysing execution traces Asadi *et al.* (2010b; 2010a) identified concepts by finding cohesive and decoupled fragments in a trace using a genetic algorithm. Although, they found that the genetic algorithm identifies concepts with high precision, their approach is computationally intensive. Furthermore, it is based on metaheuristic search and thus each run may produce a different concept assignment. To address these limitations, we automatically split exection traces into meaningful segments, each representing a concept. To do so, we reformulate the trace segmentation problem as a dynamic programming problem. This chapter describe the new problem formulation and the algorithmic details of our approach. We then compare and discuss the results of the performances of dynamic programming with those of a genetic algorithm.

4.1 Trace Segmentation Approach

The execution trace segmentation approach consists of five steps. First, a program is instrumented. Second, the program is exercised to collect execution traces. Third, the collected traces are compacted to reduce the search space that must be explored to identify concepts. Fourth, each method of the program is represented by means of the text that it contains. Finally, a search-based optimisation technique is used to identify, within execution traces, sequences of method invocations that are related to a concept, this latter step will be detailed in the section 4.1.2 here we simply report the computed fitness function.

4.1.1 Trace Segmentation Problem

This section summarises essential details of a previous trace segmentation approach (Asadi *et al.*, 2010a,b), which problem we reformulate as a dynamic programming problem. Therefore, the five steps of the two approaches are identical, with the only difference that the trace segmentation was previously performed using a GA algorithm and that we describe the use of DP in Section 4.1.2.

Step 1 and 2 – Program Instrumentation and Trace Collection First, a program under study is instrumented using the *instrumentor* of MoDeC to collect traces of its execution under some scenarios. MoDeC is an external tool to extract and model sequence diagrams from Java programs (Ng *et al.*, 2010), implemented using the Apache BCEL bytecode transformation library [1]. The tool also allows to manually label parts of the traces during executions of the instrumented programs, which we did to produce our oracle. In this dissertation MoDeC is simply used to collect and manually tag traces.

Step 3 – Pruning and Compacting Traces Usually, execution traces contain methods invoked in most scenarios, e.g., methods related to logging or Graphical User Interface (GUI) events. Yet, it is unlikely that such invocations are related to any particular concept, i.e., they are utility methods. We build the distribution of method invocation frequency and prune out methods having an invocation frequency greater than $Q3 + 2 \times IQR$, where $Q3$ is the third quartile (75% percentile) of the invocation frequency distribution and IQR is the interquartile range because these methods do not provide useful information when segmenting traces and locating concepts.

Execution trace contains repetitions of method calls, for example $m1(); m1(); m1();$ or $m1(); m2(); m1(); m2();$. Since the repetition does not define a new concept we remove the repetitions using the Run Length Encoding (RLE) algorithm and we just keep one occurrence of any repetition. We compact any sub-sequences of method invocations having an arbitrary length. The examples would become $m1()$ and $m1(); m2()$, respectively.

We compact the traces using a Run Length Encoding (RLE) algorithm to remove repetitions of method invocations. We still apply the RLE compaction to compare segments obtained with the DP approach with those obtained using the GA approach when segmenting the same traces.

Step 4 – Textual Analysis of Method Source Code Trace segmentation aims at grouping together subsequent method invocations that form conceptually cohesive groups. The conceptual cohesion among method is computed using the Conceptual Cohesion metric defined by Marcus *et al.* (2008).

We first extract a set of terms from each method by tokenizing the method source code and comments, removing out special characters, programming language keywords, and terms belonging to a stop-word list for the English language. We split compound identifiers separated by Camel Case, e.g., getBook is split into get and book. Then, we perform stemming using a Porter stemmer (Porter, 1980). We then index the obtained terms using the $TF\text{-}IDF$

1. http://jakarta.apache.org/bcel/

indexing mechanisms (Baeza-Yates et Ribeiro-Neto, 1999). We obtain a term–document matrix, and where documents are all methods of all classes belonging to the program under study and where terms are all the terms extracted (and split) from the method source code. Finally, we apply Latent Semantic Indexing (LSI) (Deerwester *et al.*, 1990) to reduce the term–document matrix into a concept–document [2] matrix, choosing, as in previous works (Asadi *et al.*, 2010a,b), a LSI subspace size equal to 50.

Step 5 – Trace Splitting through Optimization Techniques Since the execution traces are very large and the execution trace segmentation solution must be found in large search spaces. Due to the potentially large size of the search space we need to apply some optimization techniques to segment the obtained trace. Applying an optimization technique requires a representation of the trace and of a trace segmentation and a means to evaluate the quality of a trace segmentation, i.e., a fitness function. In the following paragraphs, we reuse where possible previous notations and definitions (Asadi *et al.*, 2010b) for the sake of simplicity.

The fitness function drives the optimization technique to produce a (near) optimal segmentation of a trace into segments likely to relate to some concepts. It relies on the software design principles of cohesion and coupling, already adopted in the past to identify modules in programs (Mitchell et Mancoridis, 2006), although we use conceptual (i.e., textual) cohesion and coupling measures (Marcus *et al.*, 2008; Poshyvanyk et Marcus, 2006), rather than structural cohesion and coupling measures.

Segment cohesion (COH) is the average (textual) similarity between the source code any pair of methods invoked in a given segment l. It is computed using the formulas in Equation 4.1 where $begin(l)$ is the position of the first method invocation of the l^{th} segment and $end(l)$ the position of the last method invocation in that segment. The similarity σ between methods m_i and m_j is computed using the cosine similarity measure over the LSI matrix from the previous step. COH is the average of the similarity (Marcus *et al.*, 2008; Poshyvanyk et Marcus, 2006) of all pairs of methods in a segment.

Segment coupling (COU) is the average similarity between a segment l and all other segments in the trace, computed using Equation 4.2, where N is the trace length. It represents, for a given segment, the average similarity between methods in that segment and those in different ones.

Thus, we compute the quality of the segmentation of a trace split into K segments using the fitness function (fit) defined in Equation 4.3, which balances segment cohesion and their

2. In LSI “concept” refers to orthonormal dimensions of the LSI space, while in the rest of the dissertation “concept” means some abstraction relevant to developers.

coupling with other segments in the split trace.

$$
\begin{aligned}
COH_l &= \frac{\sum_{i=begin(l)}^{end(l)-1} \sum_{j=i+1}^{end(l)} \sigma(m_i, m_j)}{(end(l) - begin(l) + 1) \times \frac{(end(l)-begin(l))}{2}} &\quad (4.1)\\
COU_l &= \frac{\sum_{i=begin(l)}^{end(l)} \sum_{j=1, j<begin(l) \text{ } or \text{ } j>end(l)}^{N} \sigma(m_i, m_j)}{(N - (end(l) - begin(l) + 1)) \times (end(l) - begin(l) + 1)} &\quad (4.2)\\
fit(segmentation) &= \frac{1}{K} \times \sum_{i=1}^{K} \frac{COH_i}{COU_i + 1} &\quad (4.3)
\end{aligned}
$$

4.1.2 Search-based Optimization Technique

Now we use previous notations and definitions to describe the use of a GA algorithm to segment traces and the reformulation of the trace segmentation problem as a dynamic programming problem.

Trace Segmentation using a Genetic Algorithm

Asadi *et al.* (2010a; 2010b) represent a problem solution, i.e., a trace segmentation, as a bit-string as long as the execution trace in number of method invocations. Each method invocation is represented as a “0”, except the last method invocation in a segment, which is represented as a “1”. For example, the bit-string $\underbrace{00010010001}_{11}$ represents a trace containing 11 method invocations and split into three segments: the first four method invocations, the next three, and the last four.

Now, the mutation, crossover, and selection operators, used by a GA to segment traces are described (Asadi *et al.*, 2010a,b). The mutation operator randomly chooses one bit in the trace representation and flips it over. Flipping a “0” into a “1” means splitting an existing segment into two segments, while flipping a “1” into a “0” means merging two consecutive segments. The crossover operator is the standard 2-points crossover. Given two individuals, two random positions x, y, with $x < y$, are chosen in one individual’s bit-string and the bits from x to y are swapped between the two individuals to create a new offspring. The selection operator is the roulette-wheel selection. Asadi *et al.* use a simple GA with no elitism, i.e., it does not guarantee to retain best individuals across subsequent generations.

Trace Segmentation using Dynamic Programming

DP technique is divided into three steps: first, recursively breaking the problem down into sub-problems; second, expressing the solution of the original problem in term of the solutions of the sub-problems; and finally applying the Bellman's principle of optimality.

For our trace segmentation problem, we interpret the obove steps as follows. When computing a trace segmentation, at a given intermediate method invocation in the trace and for a given number of segments ending with that invocation, only the best among those possible partial splits, will be, possibly, part of the final optimal solution. Thus, we must record only the best fitness for any segmentation and we must expand only the corresponding best segment to include more method invocation, possibly including the entire trace.

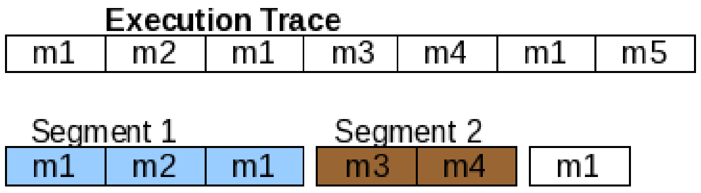


Figure 4.1 Example of execution trace segmentation.

Let's consider the example of the trace segmentation in the Figure 4.1. The existing solution is that we compose the first five methods into two segments: Segment 1 composed of the first three methods of the trace and Segment 2 included the fourth and the fifth methods. When extending the existing solution two things can happen: either a third segment is added starting from the method "m1" or the method "m1" is attached to Segment 2. Given the trace of the example in the Figure 4.1, suppose we compute and store all possible optimal splits of a trace into two segments. The sub-trace of the first five method invocations, we compute its optimal (in terms of fitness function) split into two segments. Clearly there are several ways to split the five methods into two segments, however we only consider the best in term of fitness function. The same can be done for a sub-trace of length six and seven. When we reach the end of the trace we will have the best segmentation on the given trace into two parts. When computing the segmentation into three parts, there is no need to redo all computations. For example, three segments ending at position seven can be computed in terms of two segments ending at any previous position (e.g., position five), and forming a third segment with the remaining methods. Thus, a possible solution consists of the two segments ending at position five, plus a segment of length two.

More formally, let $\mathcal{A} = \{1, 2, \ldots, n\}$ be an alphabet of n symbols, i.e., method invocations, and $T[1 \ldots N]$ be an array of method invocations of $\mathcal{A}$, i.e., an execution trace. Given an interval $T[p \ldots q]$ $(1 \leq p \leq q \leq N)$ of $T[1 \ldots N]$, as explained in Section 4.1.1, we compute

COH as the average similarity between the elements of $T[p \ldots q]$ and the interval coupling, COU, as the average similarity between any element of $T[p \ldots q]$ (methods between p and q) and any element of $T[1 \ldots N]-T[p \ldots q]$. We compute the *score of an interval* as COH/COU.

A segmentation S of $T[1 \ldots L](L \leq N)$ is a partition S of $T[1 \ldots L]$ in k_S intervals: $S = \{T[1 \ldots a_1], T[a_1 + 1 \ldots a_2] \ldots T[a_{k-1} + 1 \ldots a_k = N]\}$. We denote such a segmentation by $(a_0 = 0, a_1, \ldots, a_{k_S} = L)$. We then define the segmentation score (e.g., fitness) of an array as the average score of its intervals. Therefore, the *trace segmentation problem* consists to find a segmentation of $T[1 \ldots N]$ maximizing the score fit, as defined in Section 4.1.1.

We introduce the definitions D1–D4 to explain our DP approach:

(D1) $A(p, q) = \Sigma_{i=p}^{q-1} \Sigma_{j=i+1}^{q} \sigma(i, j)$
(D2) $B(p, q) = \Sigma_{i=p}^{q} \Sigma_{j=1 \ldots N(j \notin [p,q])} \sigma(i, j)$
(D3) $f(p, q) = \frac{2 \times (N-(q-p+1))}{(q-p)} \times \frac{A(p,q)}{B(p,q)}$
(D4) $fit(k, L) = max_{\{(a_i)_{i=0..k}: a_0=0, a_i<a_{i+1}, a_k=L\}} \Sigma_{i=1..k} f(a_{i-1} + 1, a_i)$

We notice that the COH and COU of an interval $T[p \ldots q]$ correspond to $\frac{2 \times A(p,q)}{(q-p)\times(q-p+1)}$ and $\frac{B(p,q)}{(N-(q-p+1))\times(q-p+1)}$, respectively. Thus $f(p, q)$ represents the score of the interval $T[p \ldots q]$. It also represents the contribution of the interval to a solution and $fit(k, L)$ corresponds to the maximum score of a (k, L)-segmentation, i.e., a segmentation of $T[1 \ldots L]$ in k intervals. Therefore, the optimum segmentation score is $max_{k=1}^{N/2} \frac{fit(k,N)}{k}$.

If we consider a solution ending at p (sub-trace $T[1 \ldots p]$) and made up by k segments, then its score is $fit(k, p)$ and we have multiple optimum segmentations: one for each possible k in $1 < k < p/2$. When we extend the sub-trace to q, $T[1 \ldots p \ldots q]$ and given a solution made up of k segments ending in p, we seek the solution $fit(k + 1, q)$ into $max_{p=k \ldots q}(fit(k, p) + f(p + 1, q))$, where $1 \leq k < q \leq N$. If we pre-compute and store $fit(k, p)$ in a table, we do not need to recompute the expensive COH and COU every time to evaluate $fit(k + 1, q)$. However, we still must compute $f(p + 1, q)$ for every sub-problems and we perform this computation efficiently using the following definitions:

(D5) $\Delta(p, q) = \Sigma_{i=p}^{q-1} \sigma(T[i], T[q])$
(D6) $\Theta(p) = \Sigma_{i=1..N(i \neq p)} \sigma(T[i], T[p])$

It can be proved that $\Delta(p, q) = \Delta(p + 1, q) + \sigma(T[p], T[q])$ and, thus, $A(p, q) = A(p, q - 1) + \Delta(p, q)$ and $B(p, q+1) = B(p, q) + \Theta(q+1) - 2 \times \Delta(p, q+1)$ and thus we can recursively update $A(p, q)$ and $B(p, q+1)$. We choose $q = p+1$, which means that we extend the current solution one method at the time from left-to-right and that $A(p, q)$ becomes $A(p, p + 1)$ and

$B(p, q+1)$ becomes $B(p, p+2)$, which we can pre-compute (from previous values) and stored into two arrays.

To conclude, we can compute $fit(k+1, p+1)$ using $fit(k, i)$ and the sum of the values of $f(i+1, p+1)$, which we can compute by dividing $A(i+1, p+1)$ by $B(i+1, p+1)$, both already pre-computed. The DP approach is thus fast because it goes left-to-right and reuses as much as possible of previous computation.

We show below the pseudo-code of (a basic version of) the algorithm at the core of the DP approach.

Algorithm DP split
Input:
integers n and N, matrix of similarities $Sim[1..n][1..n]$, array $T[1..N]$
Output: matrix of fitness values $fit[1..N][1..N]$

```
1.  For L=1..N do
2.      Theta := comp_theta(L)
3.      Delta := 0
4.      A[L] := 0
5.      B[L] := Theta
6.      For p=L-1..1 do
7.          Delta := Delta + Sim[T[p]][T[L]]
8.          A[p] := A[p-1] + Delta
9.          B[p] := B[p-1] + Theta − 2 × Delta
10. For L=1..N do
11.     fit[1][L] := comp_f(1,L)
12.     For k=2..L do
13.         F_max := 0
14.         For p=k..L-1 do
15.             F_max:=max(F_max, fit[k-1][p] + comp_f(p+1))
16.         fit[k][L] := F_max
17. Return fit
```

where the input matrices $Sim[1..n][1..n]$ and $T[1..N]$ contain the similarities between methods and the trace encoding, respectively. The function *comp_f*() computes the value of f based on definition $D3$ and *comp_theta* recursively evaluates $\Theta(p)$. The function *comp_f*(p) and the function *comp_theta*(p) are computed as follows:

```
1.  Function comp_f(int $p$)
2.      Return $(2 * (N - (q - p + 1)))/(q - p)) * (A[p]/B[p])$
3.  Function comp_theta(int $p$)
```

```
4     Res := 0
5.    **For** i=1..p-1 **do**
6.        Res := Res+simil(i,p)
7.    **For** i=p+1..N **do**
8.        Res := Res+simil(i,p)
9.    **Return** Res
```

The most expensive part of the algorithm are the nested loops at lines 10, 12, and 14. The algorithm, in this basic formulation, has a complexity of $\mathcal{O}(N^3)$, which is also the (worst case) complexity of the evaluation of the GA fitness function as both COH and COU have worst case complexity of $\mathcal{O}(N^2)$ and in the worst case must be evaluated for $N/2$ segments. Thus, a single step of the GA approach equates the entire calculation of the DP approach.

4.2 Case Study

This section reports an empirical study comparing the GA approach proposed by Asadi *et al.* (2010b) with our novel DP approach. The goal of this study is to analyse the performances of the trace segmentation approaches based on GA and DP with the purpose of evaluating their capability to identify meaningful concepts in traces. The quality focus is the accuracy and completeness of the identified concepts. The perspective is that of researchers who want to evaluate which of the two techniques (GA or DP) better solves the trace segmentation problem. The context consists of two trace segmentation approaches, one based on GA and one on DP, and of the same execution traces used in previous work (Asadi *et al.*, 2010b) and extracted from two open-source programs, ArgoUML and JHotDraw.

ArgoUML[3] is an open-source UML modelling tool with advanced concepts, such as reverse engineering and code generation. The ArgoUML project started in September 2000 and is still active. *ArgoUML* has been widely studied and used in various research works. We analysed *ArgoUML* release 0.19.8, which contains 1,230 classes in about 113 KLOC. *JHotDraw*[4] is a Java framework for drawing 2D graphics. JHotDraw started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. Similarly to *ArgoUML*, *JHotDraw* has been widely used in various research works due to its structure (based on extensive adoption of design patterns) and documentation. We analysed release 5.1, which consists of 155 classes in about 8 KLOC.

Table 4.1 summarises the programs statistics. We generated traces by exercising various scenarios in the two programs. Table 4.2 summarises the scenarios and shows that the

3. http://argouml.tigris.org
4. http://www.jhotdraw.org

Table 4.1 Statistics of the two programs.

Programs	**NOC**	**KLOC**	**Release Dates**
ArgoUML v0.18.1	1,267	203	30/04/05
JHotDraw v5.4b2	413	45	1/02/04

Table 4.2 Statistics of the collected traces.

Programs	**Scenarios**	**Original Sizes**	**Cleaned Sizes**	**Compacted Sizes**
ArgoUML	Start, Create note, Stop	34,746	821	588
	Start, Create class, Create note, Stop	64,947	1,066	764
JHotDraw	Start, Draw rectangle, Stop	6,668	447	240
	Start, Add text, Draw rectangle, Stop	13,841	753	361
	Start, Draw rectangle, Cut rectangle, Stop	11,215	1,206	414
	Start, Spawn window, Draw circle, Stop	16,366	670	433

generated traces include from 6,000 to almost 65,000 method invocations. The compacted traces include from 240 up to more than 750 method invocations.

This study aims at answering the three following research questions:

- **RQ1.** *How do the performances of the GA and DP approaches compare in terms of fitness values, convergence times, and number of segments?* This research question analyses whether DP approach outperfoms a GA approach. Because our goal is to find the trace segmentation with the best fitness function value, we compare the fitness function values of GA and DP approaches. The DP approach is proposed to overcome the scalability problem of GA approach, so we compare the computation times of both approaches.
- **RQ2.** *How do the GA and DP approaches perform in terms of overlaps between the automatic segmentation and the manually-built oracle, i.e., recall?* This research question evaluates whether the identified segments are meaningful and representing concepts with respect to the manually-built segments. We compare the overlap between the manually-built segments and the segments identified by both approaches.
- **RQ3.** *How do the precision values of the GA and DP approaches compare when splitting execution traces?* This research question investigates how precise are the GA and DP approaches to identify segments representing a concept in comparison to the manually-built segments.

The GA approach is implemented using the *Java GA Lib*[5] library. Asadi *et al.* use a simple GA with no elitism, i.e., it does not guarantee to retain best individuals across subsequent generations. They set the population size to 200 individuals and a number of

5. http://sourceforge.net/projects/javagalib/

generations of 2,000 for shorter traces (those of JHotDraw) and 3,000 for longer ones (those of ArgoUML). The crossover probability is set to 70% and the mutation to 5%, which are values used in many GA applications.

The DP approach scans the trace from left-to-right building the exact solution and in its current formulation does not have any configuration parameter.

In previous work, the results of the GA approach were reported for for multiple (10) runs of the algorithm to account for the nondeterministic nature of the technique. We only report the results of the DP approach for one of its run per traces because it is by nature deterministic and multiple runs would produce exactly the same results. Also, we compare DP results with the best result regarding fitness function values achieved among the 10 GA runs.

To address **RQ1**, we compare the value of the fitness function reached by the GA approach with the value of the segmentation score obtained by the DP approach. The values of the fitness function and segmentation score do not say anything about the quality of the obtained solutions. Yet, we compare these values to assess, given a representation and a fitness function/segmentation score, which of the GA or DP approach obtain the best value. We also compare the execution times of the GA and DP approaches. We finally report the number of segments that the two approaches create for each execution trace.

For **RQ2**, we compare the overlap between a manually-built oracle and segments identified by the GA and DP approaches. We build an oracle by manually assigning a concept to trace segments—using the tagging concept of the instrumentor tool—while executing the instrumented programs. Given the segments determined by the tags in the oracle and given the segments obtained by an execution of either of the approaches, we compute the Jaccard overlap (Jaccard, 1901) between each manually-tagged segment in the oracle and the closest, most similar segment obtained automatically. Let us consider a (compacted) trace composed of N method invocations $T \equiv m_1, \ldots m_N$ and partitioned in k segments $s_1 \ldots s_k$. For each segment s_x, we compute the maximum overlap between s_x and the manually-tagged segments so_y as $max(Jaccard(s_x, so_y)), y \in \{1 \ldots k\}$ where:

$$Jaccard(s_x, so_y) = \frac{|s_x \cap so_y|}{|s_x \cup s_y|}$$

For **RQ3**, we evaluate (and compare) the precision of both the GA and DP approaches in terms of precision, which is defined as follows:

$$Precision(s_x, so_y) = \frac{|s_x \cap so_y|}{|s_y|}$$

where s_x is a segment obtained by an automatic approach (GA or DP) and so_y is a segment in the corresponding trace of the oracle.

For **RQ1**, **RQ2**, and **RQ3**, we statistically compare results obtained with the GA and DP approaches using the non-parametric, paired Wilcoxon test. We also compute the magnitude of the differences using the non-parametric effect-size Cliff's δ measure (Grissom et Kim, 2005), which, for dependent samples, as in our study, is defined as the probability that a randomly-selected member of one sample DP has a higher response than a randomly-selected member of the second sample GA, minus the reverse probability:

$$\delta = \frac{\left|\text{DP}^i > \text{GA}^j\right| - \left|\text{GA}^j > \text{DP}^i\right|}{|\text{DP}|\,|\text{GA}|}$$

The effect size δ is considered small for $0.148 \le \delta < 0.33$, medium for $0.33 \le \delta < 0.474$ and large for $\delta \ge 0.474$ (Grissom et Kim, 2005).

4.3 Results and Discussions

4.3.1 Results

This section reports the results of the empirical study.

Table 4.3 Number of segments, values of fitness function/segmentation score, and times required by the GA and DP approaches.

Program	**Scenario**	**# of Segments**		**Fitness**		**Time (s)**	
		GA	**DP**	**GA**	**DP**	**GA**	**DP**
ArgoUML	(1)	24	13	0.54	0.58	7,080	2.13
	(2)	73	19	0.52	0.60	10,800	4.33
JHotDraw	(1)	17	21	0.39	0.67	2,040	0.13
	(2)	21	21	0.38	0.69	1,260	0.64
	(3)	56	20	0.46	0.72	1,200	0.86
	(4)	63	26	0.34	0.69	240	1.00

Regarding **RQ1**, Table 4.3 summarises the obtained results using both the GA and DP approaches, in terms of (1) number of segments in which the traces were split, (2) achieved values of fitness function/segmentation score, and (3) times needed to complete the segmentations (in seconds). The DP approach tends to segment the trace in less segments than the GA one, with the exception of Scenario (1) of JHotDraw, composed of one concept only and for which the DP approach creates 21 segments whereas GA creates only 17 segments, and of Scenario (2) of JHotDraw, for which the number of segments is 21 for both approaches. The difference of the number of segments is not statistically significant (p-value=0.10), although Cliff's δ effect size is high (1.16) and in favor of the GA approach.

Looking at the values of the fitness function/segmentation score, the DP approach always produces better values than the GA one. The Wilxocon test indicates that the difference is statistically significant (p-value=0.03) and the Cliff's δ effect size is high (0.76): the DP approach performs significantly better than the GA approach, given the representations described in Section 4.1.1. The better convergence of the DP also explains the smaller number of segments obtained; that is, DP is able to converge to better solutions that—according to the fitness function of equation (4.3)—favor a smaller number of segments.

Finally, the convergence times of the GA approach are by far higher than that of the DP one: from several minutes or hours (for ArgoUML) to seconds. The difference between the GA and DP approaches is statistically significant (p-value=0.03) and the effect size high (1.05).

We thus answer **RQ1**: How do the performances of the GA and DP approaches compare in terms of fitness values, convergence times, and number of segments? The DP approach out-performs the GA approach by stating that in terms of fitness values, convergence time, and number of segments.

Table 4.4 Jaccard overlaps and precision values between segments identified by the GA and DP approaches.

Program	**Scenario**	**Concept**	**Jaccard**		**Precision**	
			GA	**DP**	**GA**	**DP**
ArgoUML	(1)	Create Note	0.33	0.87	1.00	0.99
	(2)	Create Class	0.26	0.53	1.00	1.00
	(2)	Create Note	0.34	0.56	1.00	1.00
JHotDraw	(1)	Draw Rectangle	0.90	0.75	0.90	1.00
	(2)	Add Text	0.31	0.33	0.36	0.39
	(2)	Draw Rectangle	0.62	0.52	0.62	1.00
	(3)	Draw Rectangle	0.74	0.24	0.79	0.24
	(3)	Cut Rectangle	0.22	0.31	1.00	1.00
	(4)	Draw Circle	0.82	0.82	0.82	1.00
	(4)	Spawn window	0.42	0.44	1.00	1.00

To address **RQ2**, we evaluate the Jaccard overlap between the manually-identified segments corresponding to each concept of the execution scenarios and the segments obtained using the GA and DP approaches. Columns 4 and 5 of Table 4.4 report the results. Jaccard scores are always higher for the GA approach than for the DP one, with the only exception of the *Draw Rectangle* concept in JHotDraw, for which the Wilcoxon paired test indicates that there is no significant difference between Jaccard scores (p-value=0.56). The obtained Cliff's δ (0.11) is small, although slightly in favor of the DP approach. We thus answer **RQ2** by stating that in terms of overlap, segments obtained with the GA and DP approaches do not significantly differ and the DP approach has thus a recall similar to that of the GA one..

We thus answer **RQ2**: How do the GA and DP approaches perform in terms of overlaps between the automatic segmentation and the manually-built oracle, i.e., recall? DP is similar to the GA in terms of overlaps between automatic segmentation and manually-built oracle.

Regarding **RQ3**, Columns 6 and 7 of Table 4.4 compare the precision values obtained using the GA and DP approaches. Consistently with results reported in previous work (Asadi *et al.*, 2010b), precision is almost always higher than 80%, with some exceptions, in particular the *Add Text* and *Draw Rectangle* concepts of JHotDraw. There is only one case for which the DP approach exhibits a lower precision than the GA one: for the *Draw Rectangle* concept of JHotDraw (Scenario 3) where the DP approach has a precision of 0.24 whereas the GA one has a precision of 0.79. Yet, in general, the Wilcoxon paired test indicates no significant differences between the GA and DP approaches (p-value=0.52) and the Cliff's δ (0.04) indicates a negligible difference between the two approaches. In conclusion, we answer **RQ3** by stating that the precision obtained using the DP approach does not significantly differ from the one obtained using the GA approach.

We thus answer **RQ3**: How do the precision values of the GA and DP approaches compare when splitting execution traces? The precision of DP is similar to the GA one when comparing automatic segmentation and the manually-built oracle.

4.3.2 Discussions

Qualitative Analysis

We analyse in depth the segmentation results to understand how our approach splits the traces into segments. The Jaccard score is lower for JHotDraw thus we focus on the cases where the Jaccard score is low. For other cases, because they are consistent with the oracle, we claim that the segments are meaningful.

The concept *Cut rectangle* of JHotDraw was tagged as a sequence of 172 method invocations. However, only 55 of these methods were grouped together by the DP approach. We analysed this sequence and observed that it is related to (1) add the rectangle content to the clipboard and (2) modify the properties of the drawn rectangle to appear as "cut" in the painter. The remaining sequence of 117 (= 172 - 55) method invocations was splited in many small segments in which GUI events were interleaved. So, we claim that the DP approach produced meaningful segments.

Moreover, in the scenario 3 of JHotDraw, the concepts *Cut rectangle* and *Draw rectangle* are implemented using similar sequence of method invocations (i.e., add the rectangle content to the clipboard and modify the properties of the drawn rectangle). Because these concepts are executed one after the other, our DP approach splits the trace into segments dissimilar to the ones from the oracle. Consequently, the overlap and precision of *Draw rectangle* was 24%, which were very high in the scenario 1 with 75% of overlap and 100% of precision.

The concept *Add text* of JHotDraw presented also a low overlap. This lower overlap does not mean that our approach was unable to successfully identify the concepts. In fact, the concept *Add text* was created by adapting a textual-editing concept as a shape-drawing concept, using the Adapter design pattern. This adaption of the concepts generate non-cohesive sequences of methods. Thus, our algorithm splits these sequences in small segments resulting in low overlaps.

Finally, we highlighted the capability of the DP approach to split execution traces into cohesive segments despite the low Jaccard overlap with respect to the oracle. Indeed, the extensive use of design patterns in JHotDraw explains the lower results when compared to those obtained for ArgoUML. Inheritance and design patterns lead to create many method invocations not directly related to a concept, but supporting the implementation of this concept. Consequently, these method invocations were present in different segments related to different concepts.

DP approach Scalability

To evaluate which of the two techniques (GA or DP) better solves the trace segmentation problem, we compared the segmentation results of the same execution traces used in a previous work (Asadi *et al.*, 2010b) and extracted from two open-source programs, ArgoUML and JHotDraw. However, the sizes of the studied traces are from 240 to less than 800 method invocations. To understand the scalability of the DP approach, we studied its computation time to split large traces into segments. Thus, we generated traces by exercising various scenarios in the two programs: ArgoUML and JHotDraw. Table 4.5 summarises the generated traces, which range from 28,000 to almost 3,000,000 method invocations. The compacted traces include from 1,000 up to more than 50,000 method invocations. Table 4.5 shows also the computation times spent by the DP approach to split these traces into segments, which are from 5 seconds to about 29 hours. Figure 4.2 presents the computation times spent by the

DP approach as a function of the sizes of the studied traces. We observe in Figure 4.2 that the increase in the computation times is most dramatic from 25,000 method invocations. We conclude that the computation times exponentially increase with trace sizes. This increase could be due to the limited memory available on the computer running the experiments, resulting in memory swapping. In average, splitting a compacted trace of about 38,000 methods took about one day. To make the DP approach appealing, we must improve scalability in time to handle huge traces and to obtain results in a reasonable amount of time.

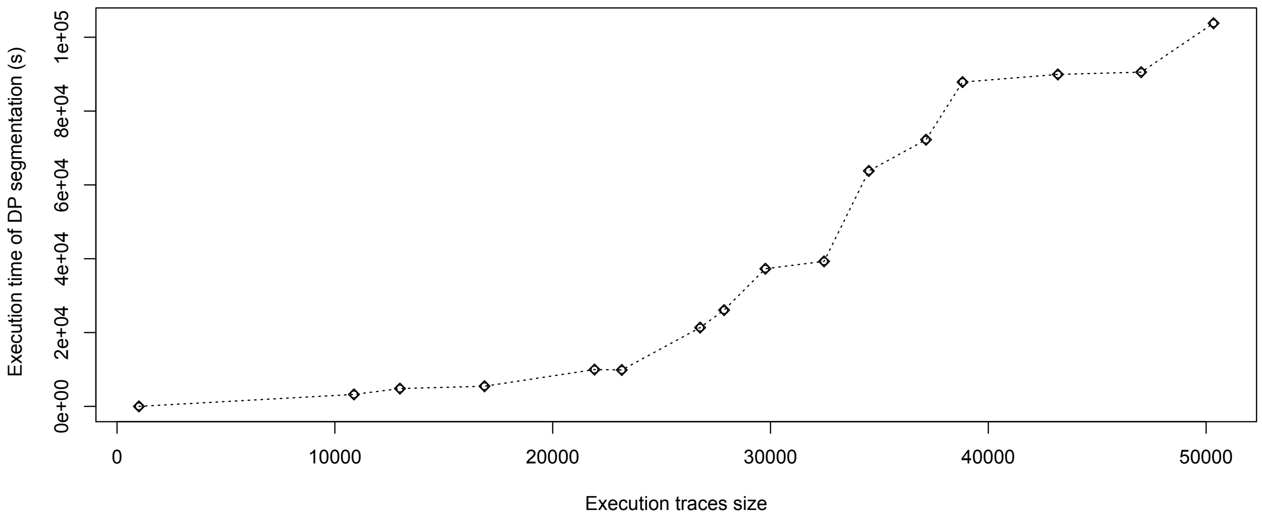


Figure 4.2 Computation Times of DP approach.

4.3.3 Threats to Validity

We now discuss the threats to the validity of our empirical study.

Construct validity Threats to construct validity concern the relation between theory and observation. We cannot compare the times required by the GA and DP approaches to achieve the same fitness value/segmentation score because the DP approach always reaches, by construction, the global optimum while the GA approach does not. Moreover, even if the achieved fitness values and segmentation scores are different, we showed that the DP approach is able to reach a better score in a shorter time.

Table 4.5 Statistics of the collected traces.

Programs	Original Sizes	Compacted Sizes	Time (s)
ArgoUML	924,645	10,877	3,244
	458,504	16,870	5,460
	302,349	23,174	9,869
	1,231,732	27,869	26,086
	1,918,062	32,459	39,287
	2,934,261	34,517	63,796
	1,054,994	38,817	87,854
	613,413	47,016	90,544
	162,577	50,341	103,786
JHotDraw	27,894	1,004	5
	351,810	12,987	4,827
	370,189	21,920	9,981
	287,984	26,769	21,350
	243,313	29,765	37,292
	464,729	37,140	72,238
	688,526	43,193	89,926

Internal validity Threats to internal validity concern confounding factors that could affect our results. These could be due to the presence, in the execution traces, of extra method invocations related to GUI events or other program events. The frequency-based pruning explained in Step 3 of Section 4.1.1 mitigates this threat.

Conclusion validity Threats to conclusion validity concern the relationship between treatment and outcome. We statistically compared the performances of the GA and DP approaches using the non-parametric Wilcoxon paired test and used the non-parametric Cliff's δ effect size measure.

4.4 Summary

In this chapter we reformulate the trace segmentation problem as a DP problem and, specifically, as a particular case of the string splitting problem. We showed that we can benefit from the overlapping sub-problems and an optimal substructure of the string splitting problem to reuse computed scores of intervals and segmentation scores and, thus, to obtain dramatic gains in performances without loss in precision and recall. Indeed, differently from a GA approach, the DP approach reuses pre-computed cohesion and coupling values among subsequent segments of an execution trace, which is not possible using genetic algorithms, due to their very nature.

We empirically compared the DP and GA approaches, using the same data set from previous work (Asadi *et al.*, 2010a,b). Our empirical study consisted in the execution traces from ArgoUML and JHotDraw, which were previously used to validate the GA approach. Results indicated that the DP approach can achieve results similar to the GA approach in terms of precision and recall when its segmentation is compared with a manually-built oracle. They also show that the DP approach has significantly better results in terms of the optimum segmentation score vs. fitness function. More important, results showed that the DP approach significantly out-performed the GA approach in terms of the times required to produce the segmentations: where the GA approach would take several minutes, even hours; the DP approach just takes a few seconds.

We introduced the DP approach to overcome the GA limitations in terms of computations time and fitness function value for trace segmentation. This trace segmentation allows developers to focus on segments to maintain instead of analysing the entire execution trace.

CHAPTER 5

Labeling Segments

We split execution traces into segments allowing developers to focus on segments to maintain instead of analysing an entire execution trace. However, most segments are still too large and difficult to understand and thus we do not guide a developer to the segments implementing the concepts to maintain. To solve this problem, we propose to label segments using a set of terms describing the concepts implemented by segments and thus facilitate maintenance tasks. In this chapter, we present an approach inspired from previous works on program summarisation (Haiduc *et al.*, 2010b; Sridhara *et al.*, 2010, 2011a,b) to assign meaningful labels by applying Information Retrieval (IR) techniques to terms extracted from method signatures. We then compare the quality of the obtained labels with manually-assigned labels obtained by 1) inspecting the source code, 2) checking the available documentation, and 3) performing a step-by-step analysis of the execution traces. We perform a qualitative as well as a quantitative analysis in which only one participant assign manual labels. Then, we improve this evaluation by an experiment to assess SCAN capability to select the most important methods of a segment and compare the resulting labels against labels provided by 31 participants.

5.1 Labeling Trace Segments Approach

SCAN accepts as input one or more execution traces obtained by exercising some scenarios in a program. Such execution traces can be obtained by executing the scenarios for which the developer is interested to perform concept location. As depicted in Figure 1.2 in Chapter 1, it consists in a series of four steps.

In Step 1, SCAN analyses an execution trace to identify segments by finding cohesive and decoupled fragments of the trace. The trace segmentation approach based on dynamic programming is presented in Chapter 4. The identified segments represents a higher-level view of the trace to allow developers to explore execution trace segments instead of low-level events (i.e., method calls).

Then, in Step 2, SCAN merges similar segments using the Jaccard measure on terms extracted from the segments. After that, in Step 3, it uses an IR-based approach to label segments. Finally, in Step 4, it uses FCA to identify relations among segments.

In the following, we provide details of Steps 2 and 3.

5.1.1 Segmentation Merger

Modern programming languages such as C, C++ or Java provide mechanisms to handle multi-threading, to take advantage of multi-processors, multi-core CPUs and in general parallel hardware. In a multi-threading application multiple threads co-exist within the context of the same process. Process threads share resources but are executed in parallel and independently. Multi-threading on one hand makes an application run faster; on the other hand there is no way to predict the order in which individual threads are executed. This means that even for a simple scenario, the sequence of methods in a trace may or may not be the same upon multiple executions of the same scenario. Consider for example, ArgoUML, a Java application used to validate SCAN. ArgoUML has an auto-save functionality, but the user has no way to decide, or define, when the auto-saving thread is activated. If involving the auto-save functionality, two executions of the same scenario will therefore produce different sequences of methods and thus potentially different, though, consistent, segmentations.

SCAN merges similar segments using the Jaccard measure on terms extracted from the segments. The aim of the merging is to recognize similarities among segments belonging to multiple execution traces and merge these segments. Indeed, we expect to have a great number of common segments between multiple execution traces of a same scenario or, even, of different scenarios even though the ordering of the segments and some other segments may be different among traces due to thread interleaving, variations in application inputs (some of which cannot be fully controlled while instrumenting the programs to collect traces), or variations in machine-load conditions. We thus suppose that highly similar segments in different traces contribute to the same concept, regardless of the specific thread interleaving or trace region in which they occur. SCAN merges segments from different execution traces of a same scenario only.

Let $S = (s_1, \ldots, s_n)$ and $Z = (z_1, \ldots, z_m)$ be two segmentations of two traces, i.e., two sequences of segments. For each s_i, SCAN computes all similarities $\sigma(s_i, z_j)$ and keeps pairs above a given threshold. The similarity between two segments is computed as the Jaccard coefficient between the segments terms, extracted from the term–document matrices of the segments. (The Jaccard coefficient for two sets A and B is defined as the ratio between the intersection $A \cap B$ and the union $A \cup B$.) The higher the number of terms in common between two segments, the higher the similarity. Once SCAN has identified all pairs of similar segments, it generates a synthetic trace containing n segments. Each segment is the result of a (possibly multiple) union of s_i and z_j where $\sigma(s_i, z_j)$ is above the threshold. SCAN attempts to find both one-to-one as well as one-to-many relations among the segments. It keeps track of the pairs of merged segments to allow mapping the information computed in subsequent steps back to the original segments.

Consider for example a trace Z split into four segments $Z = (z_1, z_2, z_3, z_4)$ and a threshold equal to 70%. First, SCAN computes the similarities between s_1 and the segments of the trace Z as follows $\sigma(s_1, z_1) = 0.8$, $\sigma(s_1, z_2) = 0.2$, $\sigma(s_1, z_3) = 0.6$, and $\sigma(s_1, z_4) = 0.9$. Then SCAN generates a synthetic segment s'_1 by merging the segments s_1, z_1, and z_4 since $\sigma(s_1, z_1)$ and $\sigma(s_1, z_4)$ are greater than 70%.

SCAN expects a (reasonably) high similarity between segments to merge two segments but this similarity is not necessarily close to one. Indeed, two segments might deserve to be merged even though their similarity is lower: let us suppose that one of the two segments is contained in the second segment as a sub-segment. In this situation, the segment similarity may not be very high. Therefore, the threshold should also not be very high. Using a lower threshold does not compromise the accuracy of the merging because the computed segments are ensured to be cohesive and decoupled, as explained in Chapter 4 and, consequently, the algorithm has no incentive in putting together non-cohesive methods.

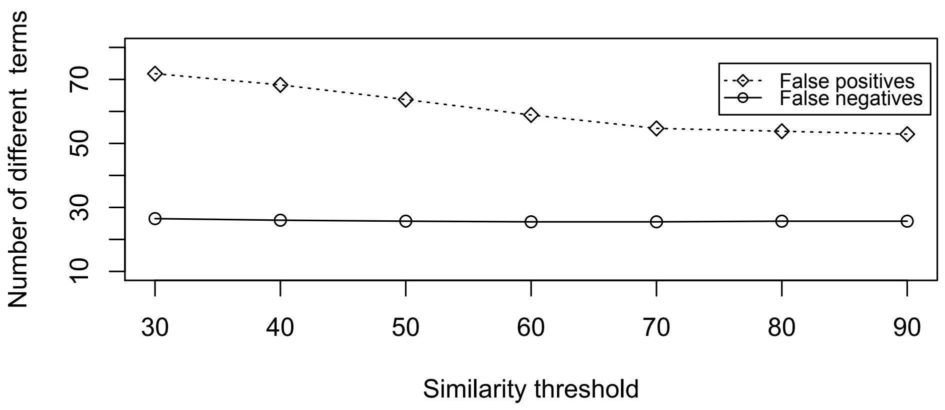


Figure 5.1 False positives and negatives according to different threshold values.

To determine a threshold providing a good accuracy when SCAN merges segments, we performed an experiment using five scenarios for ArgoUML and JHotDraw and their 11 corresponding execution traces. We varied the threshold values from 30% to 90% and evaluated the accuracy of the merged segments using the labels that SCAN generates for these merged segments and the manual labels. Our hypothesis is that the variation of the false positives and negatives is due to the merging of segments. Thus, when segments that pertain to the same concept are merged the number of false positives and negatives will be lower compared to when segments pertaining to different concepts are merged. The solid line in Figure 5.1 shows the average number of terms in the manual labels that did not appear in the automatic labels, i.e., false negatives. The average is between 26.5 for a threshold at 30% and 25.7 at 90%. The minimum value is equal to 25.5, at 60% and 70%. The values are very close thus the threshold does not significantly affect the merged segments in terms of false negatives. The dotted line in Figure 5.1 shows the average number of terms in the automatic labels that did not appear in the manual labels, i.e., false positives. The average is between 71.8 for a threshold at 30% and 52.9 at 90%. Using 30% as a similarity threshold, the number of different terms is more important because SCAN merges segments that pertain to different concepts. False positive values are more stable between 70% and 90%, between 54.7 and 52.9. Thus, we choose the value 70%. If only one segmentation is provided the merger trivially produces as output the same input segmentation.

5.1.2 Relevant Term Identification

This step represents the core of the proposed approach, and aims at labeling segments. The first issue when labeling segments is the choice of the most appropriate source of information. Each segment may contain thousands of method calls from hundreds of different methods; the problem is select a subset of terms hopefully relevant for the functionality implemented by the segment sequence of method calls. Several strategies are possible. First and foremost, since we are interested in linguistic information, we have to decide if method bodies should be considered or not. If method bodies are considered one has to decide if both comments and identifiers contribute to assigning labels or if only one of the two suffices. A second strategy is to disregard the method body and just concentrate on the signatures. However, a previous study (De Lucia *et al.*, 2012) reported that lexicon from method signatures provide more meaningful terms when labeling software artifacts than other sources. Moreover, developers often pay more attention to method signature when understanding source code. Consequently, we decided to use only terms contained in the signatures of invoked methods and their parameters.

In this step, SCAN uses an IR-based approach to label segments. Given a trace segmentation $S = (s_1, \ldots, s_n)$, SCAN extracts the signatures of all the called methods in each identified segment s_i. Then, it models the segments as a set of documents, uses Vector Space Model (VSM) to represent segments as vectors of terms and computes for each term $t_l \in s_i$ the $TF\text{-}IDF$ weighting scheme (Baeza-Yates et Ribeiro-Neto, 1999). Specifically, $TF\text{-}IDF$ provides a measure of the relevance of the terms for a segment, rewarding terms having a high term frequency in a segment (high TF) and appearing in few segments (high IDF). We make the hypothesis that a term appearing often in a particular segment but not in other segments provides important information for that segment.

SCAN ranks the terms in segments by their $TF\text{-}IDF$ values and keeps the topmost ranked terms. The number of retained terms must be a compromise between a succinct and a verbose description. Several possible strategies are foreseeable to select the top-ranked terms. First, it is possible to retain a maximum percentage (e.g., top 10%) of the terms that have the highest ranking; second, a gap-based strategy is applicable (i.e., retaining all terms up to when the difference between two subsequent terms in the ranked list is above a certain percentage of gap); and third, one could choose a fixed number of topmost terms. In this work, we adopt the latter strategy and we found that considering the topmost 10 terms represents a reasonable compromise, which yields meaningful segment labels. Note that this value represents the maximum number of terms, thus, segments containing very few methods may be labeled by SCAN with fewer terms.

The terms of the labels are extracted from method signatures because the studied traces are generated as ordered lists of methods. However, our approach is applicable to any other traces such as log traces after applying the same pre-processing (i.e., pruning and removing English stop words).

5.2 Preliminary Evaluation

The *goal* of this study is to evaluate SCAN, with the *purpose* of assessing its capabilities to label segments. The *quality focus* is the comprehension of execution traces. Maintainers have to perform this task during program understanding. The *context* consists of execution traces collected from two Java programs, JHotDraw and ArgoUML. In this evaluation, we use *JHotDraw* release 5.1 and *ArgoUML* release 0.19.8. For both programs, we collected execution traces for different scenarios. Specifically, we reuse some of the scenarios previously used to validate trace segmentation (Asadi *et al.*, 2010b; Medini *et al.*, 2011), plus we add some more, based on the knowledge we gain about this application. Tables 5.1 and 5.2 reports details about the exercised scenarios and the collected traces. In the following, we refer to each scenario with a brief English sentence such as "Draw Ellipse, Delete Ellipse". We imply that, when the scenario is executed, other than the two concepts (drawing an ellipse and deleting it), also application start-up and shut-down are executed.

The study aims at answering the following research question:

RQ1. *How effective is SCAN in assigning labels to segments?* This research question verifies whether the assigned labels corresponds to the concept encountered in a segment and thus help developers to understand the concept implemented by each segment. To address **RQ1**, we manually built labels for segments and validate the SCAN results. We then compare manually built labels with the ones produced by SCAN by computing precision and recall (Frakes et Baeza-Yates, 1992) for each segment i of a scenario j:

$$Precision_{i,j} = \frac{|M_{i,j} \cap S_{i,j}|}{|S_{i,j}|}$$

$$Recall_{i,j} = \frac{|M_{i,j} \cap S_{i,j}|}{|M_{i,j}|}$$

where $M_{i,j}$ is the set of words contained in the manually generated label for segment i of scenario j and, similarly, $S_{i,j}$ is the set of words produced by SCAN. Note that, before computing precision and recall, we preprocess the manual labels similarly to what done when producing labels automatically. Specifically, (i) we split compound words (using camel case and underscore heuristics), (ii) we remove English stop words, and (iii) we perform Porter stemming.

5.2.1 Results

In the following, we report results aimed at addressing the research question, presenting, for the sake of clarity, all results for JHotDraw first, and then all results for ArgoUML.

JHotDraw

Table 5.3 shows SCAN generated labels in the first column and the manual labels in the second column for one of the JHotDraw scenarios. The top part of Table 5.4 reports descriptive statistics (first and third quartile, median, mean and standard deviation) of precision and recall. It can be noticed that, for instance, the mean Precision varies between 0.56 of "Draw Rectangle, Draw Eclipse" and 0.65 of "Draw Rectangle, Delete Rectangle", while the mean recall is stable around 0.81-0.82. Hence, for JHotDraw the automatic labeling performs relatively well, also considering that such results are perfectly in line with performances of automatically labeling of source code artifacts (De Lucia *et al.*, 2012), which we argue are easier to label than execution traces.

To better understand the rationale of the identified segments and check the meaningfulness of the provided labels, we performed a fine-grained analysis of the segments. By exploring the content of each segment of the trace described in Table 5.3, we found, for example, that Segment 1 contains methods that start the application (menu and icons creation). Segment 2 to Segment 24 correspond to phases needed to prepare canvas for creating and adding a figure to it. Furthermore, Segment 2, Segment 4 and Segment 24 contain methods to execute the "draw figure" command. Differently from the others, Segment 19 contains methods involved in bringing the selected figure to front and to send the other figures to back. Segment 20 contains methods needed to create box and figure locations. For segments between 24 and 33,

Table 5.1 Statistics of JHotDraw collected traces.

Programs	**Scenarios**	**Original Sizes**	**Compacted Sizes**	**Number of Segments**
JHotDraw	Draw Rectangle (1)	15,706	930	54
	Draw Rectangle (2)	4,850	555	35
	Draw Rectangle, Delete Rectangle (1)	5,960	554	32
	Draw Rectangle, Delete Rectangle (2)	5,960	554	32
	Draw Ellipse (1)	4,545	556	36
	Draw Ellipse (2)	5,252	562	33
	Draw Ellipse, Delete Ellipse (1)	10,760	953	53
	Draw Ellipse, Delete Ellipse (2)	17,931	1,433	74
	Draw Rectangle, Draw Ellipse (1)	10,908	864	23
	Draw Rectangle, Draw Ellipse (2)	17,471	1,096	46
	Draw Rectangle, Draw Ellipse (3)	8,790	690	30

Table 5.2 Statistics of ArgoUML collected traces.

Programs	**Scenarios**	**Original Sizes**	**Compacted Sizes**	**Number of Segments**
ArgoUML	New Class (1)	82,579	2,785	22
	New Class (2)	60,853	2,239	19
	New Package(1)	13,115	800	15
	New Package (2)	21,423	1,642	19
	New Class, New Package (1)	38,940	1,220	13
	New Class, New Package (2)	50,650	1,146	13
	New Class, New Package (3)	36,408	1,251	12

we found that each of these segments corresponds to deletion and removal of figures, change listeners and events. Similar results have been obtained for the other scenarios.

ArgoUML

Table 5.2 reports information about traces and identified segments for ArgoUML. As for JHotDraw, we compared the automatically generated labels with the ones produced manually. Table 5.5 shows, for the ArgoUML scenario "New Class", automatic and manually generated labels for the identified trace segments. For ArgoUML, the performance analysis of the comparison between manually produced labels and labels produced by SCAN (reported in the bottom part of Table 5.4) reveals that performances are relatively lower than those obtained for JHotDraw. In particular, the mean precision ranges between 0.36 of "New Package" and 0.40 of "New Class", while the mean recall ranges between 0.48 of "New Class, New Package" and 0.64 of "New Class". The lower performances can be explained by the ArgoUML lexicon which is not as good as the JHotDraw one (JHotDraw was designed for pedagogical purposes, i.e., to show the usage of design patterns, hence source code artifacts are carefully named).

Table 5.3 SCAN generated and manual labels for the JHotDraw trace “Draw Rectangle, Delete Rectangle”.

Segment Number	**Automatic Label**	**Manual Labels**
1	draw iconkit creat palett text tool button line imag icon	Create drawing palette button tool and create icons kit.
2	draw cut transfer figur command view	Execute draw figure command.
3	draw menu copi shortcut past add command transfer duplic view	Add a command with the given shortcut to the menu.
4	draw transfer delet figur command view	Execute draw figure command.
5	ungroup draw group command view	Command to group and ungroup the selection into a group figure.
6	draw back send bring command front view	Create a command to bring to front and send to back the selected figures from others.
7	applic draw creat menu align command	Application menu creation and draw command.
8	applic draw graphic java palett menu button paint command tool	Draw command and get the selected botton from the menu palette tool.
9	add figur chang listen	Add a figure change listener.
10	add figur multicast intern chang event listen	Add a figure change event.
11	box decor anim display figur	Display the box and the borders of the figure.
12	figur set initi attribut	Initialize figure attributes.
13	figur empti size	Verify if the figure size is empty.
14	draw tool view editor standard	Draw the standard drawing tool.
15	draw applic tool set button	Set the tool of the editor.
16	graphic execut button revers enumer paint command tool select view	Execute command to paint the selected graphic button.
17	delet command execut duplic	Execute command to delete the duplicated selection.
18	ungroup group command execut	Execute command to group and ungroup the selection into a group figure.
19	execut back send bring command front	Execute command to bring to front and send to back the selected figures from others.
20	box relat locat handl west doubl kit north east south	Handle the locations and display the box of the figure.
21	unlock view unfreez draw standard	Unfreezes the view by releasing the drawing lock.
22	draw tool standard key press event view	Handling the key events in the drawing view.
23	draw transfer delet figur command view	Execute draw figure command.
24	draw remov request standard delet figur chang bounc event select	Delete the event from the selection.
25	box decor anim display figur	Display the box and the borders of the figure.
26	remov intern figur multicast chang event listen	Remove the figure change event.
27	releas decor figur	Release the figure decorator.
28	remov figur chang listen	Remove the figure change listener.
29	decor peel remov intern figur multicast releas chang event listen	Remove the figure change listener.
30	remov figur chang listen	Remove the figure change listener.
31	remov intern figur multicast chang event listen	Remove the figure change event.
32	draw enabl execut standard command key element check select view	Execute command to check enabled elements key from the selection.

Table 5.4 Descriptive statistics of precision and recall when comparing SCAN labels with manually-produced labels.

JHotDraw										
Scenario	Precision					Recall				
	Q1	median	Q3	mean	σ	Q1	median	Q3	mean	σ
Draw Rectangle	0.50	0.60	0.83	0.64	0.25	0.75	0.83	1.00	0.81	0.20
Draw Rectangle, Delete Rectangle	0.50	0.60	0.72	0.65	0.21	0.70	0.80	1.00	0.82	0.15
Draw Rectangle, Draw Eclipse	0.40	0.60	0.70	0.56	0.22	0.67	0.80	1.00	0.81	0.19

ArgoUML										
Scenario	Precision					Recall				
	Q1	median	Q3	mean	σ	Q1	median	Q3	mean	σ
New Class	0.29	0.40	0.50	0.40	0.13	0.50	0.67	0.75	0.64	0.14
New Package	0.29	0.33	0.50	0.36	0.17	0.50	0.50	0.71	0.54	0.21
New Class, New Package	0.20	0.33	0.50	0.38	0.24	0.25	0.50	0.67	0.48	0.20

We performed an in-depth analysis by exploring the content of each segment of the traces.

By manually inspecting code and documentation of ArgoUML, as well as the Cookbook for developers (Tolke *et al.*, 2004), we found that Segment 1 contains methods for system start-up: Setup the project and implement factory and helper interfaces that control the lifetime and properties of elements in the repository. Segment 2 to Segment 7 correspond to “prepare creation” and “addition” of a new UML Class. For example, Segment 2 and Segment 3 contain methods to generate the module identification key. Segment 4 contains methods to create a class and define parameters. Similar results have been obtained for the other scenarios.

In summary, we can claim that SCAN is able to assign labels in most of cases similar or representative to manually defined labels and that these labels actually correspond to the concepts encountered in the segments based on our manual inspection of code, documentation and executions.

5.2.2 Discussions

Quantitative results might be read as indicators of poor performance of the label assignment algorithm, with the recall/precision around 50% and above. As mentioned above, the achieved performance are in line with those obtained when comparing automatically generated and manually generated labels for source code artifacts (De Lucia *et al.*, 2012); moreover, the obtained results confirm that also for execution traces a simple labeling based on lexicon extracted from method signature is enough.

Also, if we complement the quantitative data with the qualitative investigation performed on the automatically labeled segments, we can conclude that this level of similarity between

Table 5.5 SCAN generated and manual labels for the ArgoUML trace “New Class”.

Segment Number	**Automatic Label**	**Manual Labels**
1	event member helper notat manag project diagram implement model factori	Add project member and implement factory and helper interfaces
2	modul java display key generat	Display the module identification key
3	modul java key generat	Generate the module identification key
4	oper type facad classifi meta mdr associ class impl paramet	Create class and define parameters
5	vertex state meta mdr type impl	Display the state vertex
6	composit state meta synch mdr type impl	Display SynchSTate and composite state
7	member helper notat factori project diagram chang model event manag	Manage diagram events changes
8	modul java display key generat	Display the module identification key
9	modul java key generat	Generate the module identification key
10	oper type facad classifi meta mdr associ class impl paramet	Create class and define parameters
11	vertex state meta mdr type impl	Display the state vertex
12	composit state meta synch mdr type impl	Display SynchSTate and composite state
13	event member helper notat manag project diagram implement model factori	Add project member, implement factory and helper interfaces
14	modul java display key generat	Display the module identification key
15	modul java key generat	Generate the module identification key
16	oper type facad classifi meta mdr associ class impl paramet	Create class and define parameters
17	vertex state meta mdr type impl	Display the state vertex
18	composit state meta synch mdr type impl	Display SynchSTate and composite state
19	notat helper facad event pump mdr model chang impl listen	Add model event change listener

automatic and manual label sets is definitely adequate to support program understanding tasks. This is because we expect that developer with some knowledge about the application would find it relatively easy to distill the relevant concepts from the automatic labels, even if such labels contain some noise and overlap only by 50% with the manually produced labels. For instance, consider the label set produced for Segment 1 of JHotDraw (see Table 5.3): it is relatively easy for someone having a (even limited) application knowledge to recognize the terms *creat draw palett button tool iconkit* as key terms for the implemented concept. Even though the manually produced label is longer and more explanatory (*Create drawing palette button tool and create icons kit*), the terms selected from the automatically produced label represent a very good and crisp summary of it. Similar considerations can be applied for Segment 18 of ArgoUML, where *synch composite state* are a meaningful summary for *Display SynchState* and *composite state.*

5.3 Labeling Segments Evaluation

In a preliminary evaluation, we manually labeled segments which may bias our evaluation of the labels generated by SCAN. In addition, we applied our approach on traces from two different programs, further studies on larger traces and more complex programs are needed to better demonstrate SCAN accuracy in assigning labels representative of concepts implemented by the segments. For this reason, we conduct a study to analyse the ability of SCAN to accurately reduce the size of segments and identify labels from segments.

5.3.1 Study Set Up

We describe the set up of the experimental evaluation. It presents the objects, i.e., execution traces of six Java programs, and the participants, i.e., 31 students and professionals that participated in the experiment.

Objects

The objects of our evaluation are execution traces collected from six Java programs belonging to different domains. ArgoUML and JHotDraw are described in Section 4.2 in Chapter 4. Mars[1] is a simulator for the MIPS assembly language. It also includes a lightweight interactive development environment (IDE) for programming in MIPS. Maze[2] is a micro-mouse maze editor and simulator. It provides statistics on the comparisons of different mazes and

1. http://courses.missouristate.edu/kenvollmar/mars/index.htm
2. http://code.google.com/p/maze-solver/

Artificial Intelligence (AI) algorithms. Neuroph[3] is a lightweight Java neural network framework to develop common neural network architectures. It includes a library of neural network concepts and a user-interface to create, train, and save networks. Pooka[4] is an email client written in Java, using Swing and JavaMail. It supports IMAP, POP3, and Unix-style mailbox folders. It also has support for encryption using PGP and S/MIME.

Table 5.6 Program characteristics.

Programs	LOCs	Trace	Trace size
ArgoUML v0.19.8	163K	New class new package	36K
JHotDraw v5.1	8K	Draw rectangle delete rectangle	6K
Mars v4.3	32K	Screen magnifier	673K
Maze r186	9K	Micro mouse	1,075K
Neuroph v2.1.0	10K	Kohonen visualizer	75K
Pooka v2.0	44K	New account new e-mail	36K
		Create folder open folder	23K
Total for the 6 programs		7 traces	

Table 5.7 Segments characteristics.

Programs - Traces	# of segments	Number of method calls					
		Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
ArgoUML - New class new package	12	2	2	2	104	37	714
JHotDraw - Draw rectangle delete rectangle	32	2	2	2	17	3	183
Mars - Screen magnifier	30	2	2	2	11	3	167
Maze - Micro mouse	75	2	2	2	30	3	999
Neuroph - Kohonen visualizer	4	2	2	4	8	10	23
Pooka - New account new e-mail	60	2	2	2	33	3	1,038
Pooka - Create folder open folder	18	2	2	2	78	5	957
Overall	231	2	2	2	34	3	1,038

The choice of these six programs was of convenience, driven by the availability of documentation to ease the participants' task of labeling the segments and by the possibility to generate traces related to different execution scenarios. Table 5.6 summarises characteristics of the programs, i.e., their sizes (in terms of lines of code), short descriptions of the scenarios used to generate the execution traces, and the sizes of the traces (in terms of number of executed events, i.e.,constructor and method calls). We used one scenario per program, except for Pooka, for which we used two scenarios. Table 5.7 reports descriptive statistics about the numbers and sizes of the segments that SCAN identifies in the execution traces.

3. http://neuroph.sourceforge.net/index.html
4. http://www.suberic.net/pooka/

Participants

Table 5.8 Participants characteristics.

	# of Participants	Years of programming experience		
		Min.	Median	Max.
Students	23	3	7	14
Professionals	8	4	9	14
Overall	31	3	7	14

The experiment involved a total of 31 participants. Eight of them were professionals, i.e., developers, researchers, or postdoctoral research fellows, and the others were students, i.e., Ph.D., M.Sc., or B.Sc. Table 5.8 provides descriptive statistics of the participants' programming experience. All participants were volunteers.

None of the participants is an original developer of the object programs. This lack of knowledge could decrease the participants' ability to properly comprehend the traces. However, developers of large software programs may not be familiar with the entire program and thus would have been subject to the same threat. To cope with this threat, we ask more than one participant to perform the same task. Because none of the participants know the programs, we do not have to take into account possible differences between "novices" or "experts" that could influence our results (Soh *et al.*, 2012).

5.3.2 Experimental Design and Analysis

The evaluation aims at answering the following research question:

– **RQ2:** *How do the labels of the trace segments produced by the participants change when providing them different amount of information?* This research question is formulated to verify whether the relevant methods characterising a segment are reasonably concise to describe the related concepts and thus help developers to understand the concept implemented by each segment.

  To address this research question, we investigate whether providing participants with the list of the most relevant methods in a segment is sufficient to produce labels. We rank methods by relevance according to their $TF\text{-}IDF$ (Baeza-Yates et Ribeiro-Neto, 1999), i.e., methods frequently invoked in the particular segment but not in other segments.

– **RQ3:** *How do the labels of the trace segments produced by the participants compare to the labels generated by SCAN?* This research question is formulated to verify whether the assigned labels summarise the concepts encountered in segments and thus help developers to understand the concept implemented by each segment. To address this research question, we evaluate the performances of SCAN when labeling segments. Similarly to De Lucia *et al.* (De Lucia *et al.*, 2012) when evaluating the labeling of software artifacts, we aggregate labels produced by the participants and compare the sets of most frequent terms with the labels automatically produced by SCAN.

Table 5.9 Segments used to evaluate the filtering using *TF-IDF*.

Programs	Traces	Segments IDs	Full segments sizes	# of unique methods
ArgoUML	New class new package	s5	92	49
JHotDraw	Draw rectangle delete rectangle	s1	183	77
		s20	69	41
Mars	Screen magnifier	s1	167	160
		s22	93	82
Maze	Micro mouse	s4	142	71
		s45	102	36
Pooka	New account new e-mail	s52	131	91
	Create folder open folder	s1	88	67
Overall	9 segments		1,067	674

In the following, we describe the evaluation design and procedure followed to answer the two research questions.

RQ2: How do the labels of the trace segments produced by the participants change when providing them different amount of information?

When the size of a segment (in terms of its numbers of method calls) is large, it is difficult to understand. To reduce the time and effort for understanding a segment, we characterise a segment using only the calls to 5 or 15 different, unique methods. Note that a method can be called more than once in a segment. We selected the values in a way to have one small and one medium versions of the segment (5 and 15 respectively). The small version reduces the number of methods to understand substantially but may result in loss of relevant information. The medium version of the segment is likely to better preserve the relevant information but at the expense of the larger number of methods that one must understand.

To address **RQ2**, we compare the labels produced by the participants when showing them three versions of a same segment: full, i.e., the segment in its original size; medium, i.e., a subset of the segment reduced to the calls to only 15 unique methods; and small, i.e., a subset of the segment reduced to the calls to only 5 unique methods. We select the unique methods using the top-most ranked methods according to the *TF-IDF* weighting scheme (Baeza-Yates et Ribeiro-Neto, 1999). A segment subset is obtained by removing all method calls other than the top 5 or 15 from the original segment. The order of the method calls are preserved.

The experiment is designed as follows. We select nine segments (belonging to different programs) whose full sizes, i.e., numbers of method calls, is between 50 and 200. We set an upper limit to control the participants' effort. We set a lower limit to ensure that the medium and small subsets of the segments are still meaningful and do not reduce only to a couple of methods. Table 5.9 shows the segments used for this part of the experiment, their original sizes, and the numbers of unique methods.

We group participants into three groups, G1, G2, and G3. We assign each version of a segment to a different group. For example, to G1, we assign the subset of the top 5 unique method calls of segment *s5* of ArgoUML, "New class new package", representing a total of 12 method calls as some methods are called more than once. To G2, we assign the subset of the top 15 unique method calls of the same segment resulting in a total of 31 method calls. To G3, we give the segment in its original version (92 method calls to 49 unique methods). Participants belonging to each group label an equal number of small, medium, and full segments.

We evaluate the filtering approach, i.e., the approach of reducing the size of segments, from two aspects: (1) the degree to which information is preserved and (2) the degree to which it preserves the agreement between participants, as explained in the following.

Preservation of Information. We use the labels produced by the participants working on the full segments as oracle to assess the preservation of information in the medium and small subsets of the segments. Thus, to evaluate the preservation of information, we compute the intersection between terms provided by participants working with medium and small segments and those produced with the full segments. The greater the intersection between the reduced (medium and small) and full versions of the segments, the higher the recall.

Preservation of Agreement among Participants. We consider the number of terms on which participants agree to be representative of a segment and of the degree of agreement among participants. Again, we use as oracle the labels produced by the participants working on full segments. To evaluate the preservation of agreement among participants, we use a two-way permutation test (Baker, 1995) to verify if the degree of agreement is significantly influenced by (1) the number of participants considered to compute the agreement, (2) the size of the segment subset given to the participants (i.e., full segment, medium and small subsets), and (3) the interaction between the number of participants and the segment size. Thus, we investigate if the agreement decreases when a larger number of participants provide labels (because different participants may provide different labels) and the agreement changes when providing participants with a different amount of information, i.e., full or reduced versions of the segments. We use also a two-way permutation test (Baker, 1995) to verify if the degree of agreement is significantly influenced by the experience of participants considered to compute the agreement. We investigate if the agreement changes when participants have high or low experience. The mean of the experience of participant is seven years. We consider highly experienced participants who have an experience period of more than seven years and low experienced participants who have a duration of experience less than seven years of experience.

We used the implementation of a permutation test available in the *lmPerm* R package. We have set the number of iterations of the permutation test to 500,000. The permutation test does sample permutations of combinations of factor levels and, therefore, multiple runs of the test may produce different results. We made sure to choose a high number of iterations so that results did not vary over multiple executions of the test. When performing the test, we assume a significance level $\alpha = 0.05$.

To assess the agreement among participants, we follow a rule to decide when two terms are considered equivalent. In Chapter 5, we ruled that there exist an agreement between two participants on a term if both participants provide two terms sharing the same stem. In the following, we extend this rule to synonyms (e.g., *shape* and *figure*), terms holding a hypernym/hyponym relation (e.g., *display* and *screen*), and terms holding a holonym/meronym relation (e.g., *point* and *location*). We use WordNet (Miller, 1995) to obtain these taxonomic relations among terms and we will refer to terms sharing those relations as *synsets*.

We considered different options to select the numbers of participants that must choose a synset to consider this synset representative of a segment. For example, we could consider a synset if at least one participant chooses it. Such a choice is equivalent to considering the union of all terms proposed by all participants. A minimum of two participants would mean that we consider only synsets chosen by at least two participants. The number of required participants can grow until it reaches the total number of participants, which would mean that a synset must be chosen by all participants to be considered representative. To avoid making choice that could bias the results of the experiment, we consider the entire range of possible values in our experiment.

RQ3: How do the labels of the trace segments produced by the participants compare to the labels generated by SCAN?

To evaluate the labels produced automatically by SCAN, we first build an oracle consisting of 210 segments labeled manually by the participants. The inclusion criteria was that these segments must include less than 100 method calls to ease the participants' labeling tasks, because larger segments could have been more difficult for participants to understand. Each segment is labeled by at least one participant. More than half of the segments (116) are labeled by two participants. Then, we evaluate SCAN by computing the precision and recall of its automatically-generated labels with respect to the labels provided by the participants.

Table 5.10 Examples of labels provided by SCAN and the participants.

	Labels	SCAN	
		Precision	Recall
SCAN	figure, listener, add, internal, multicaster, event, change		
Participants - intersection	figure, event, change	43%	100%
Participants - union	trigger, figure, event, change, listener, multicaster, manage, add	86%	75%

To explain how we performed the evaluation of the labels generated by SCAN, let us consider the automatic label produced by SCAN for a segment of JHotDraw and the corresponding manual labels provided by the participants, shown in Table 5.10. The two possible operators to combine the manual labels are intersection and union. The first operator (i.e., intersection) considers a synset to be relevant if both participants suggested it. The second operator (i.e., union) considers a synset to be relevant if at least one of the participants suggested it. We observe that, depending on the operator, the precision and recall of SCAN varies. When the more conservative operator (intersection) is chosen, the number of synsets in the manual oracle significantly decreases thus resulting in higher recall but lower precision. Union provides a balance between the two measures. We show results for union and intersection.

Term Frequency. On the same basis, we evaluate the labels produced using term frequency (TF). We compare 210 segments labeled manually by the participants and the labels produced based on tf using the inclusion and union criterias. Then, we evaluate the produced labels using tf by computing the precision and recall of these labels with respect to the labels provided by the participants.

5.3.3 Experiment Results and Discussions

This section reports the results of our experimental evaluation to address the research questions formulated in Section 5.3.2.

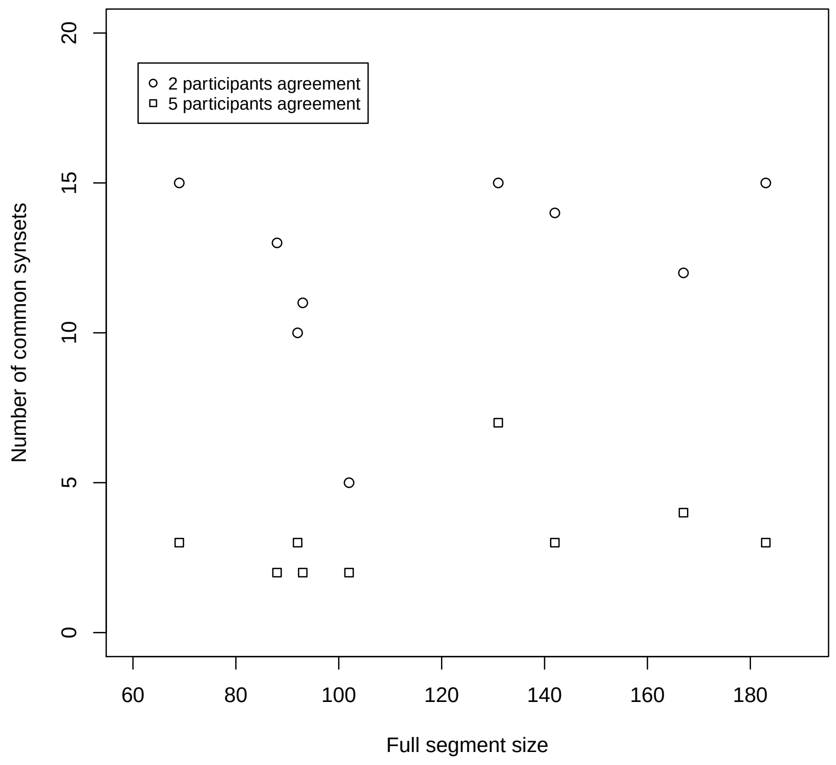


Figure 5.2 Agreement among participants for full segments.

RQ2: How do the labels of the trace segments produced by the participants change when providing them different amount of information?

Before analysing the participants' labels with different subsets of the segments, we assess the quality of the participants' labels with full segments, i.e., when using all information available. When the sizes of segments are large, the agreement among participants could be low due to the overwhelming amount of method calls, the complexity of the segments, or other factors, resulting in a random selection of terms. Figure 5.2 shows the number of synsets on which participants agree as a function of the sizes of the segments in their full version. To simplify the figure, we only show the cases of two and five participants. The figure shows that there is no linear relation between agreement and sizes, but rather a constant relation. This constant relation is confirmed by building a linear regression model and observing that the size of the segments is never a significant variable.

Table 5.11 Precision (P), Recall (R), and F-Measure (F) on the synsets of labels when comparing small and medium subsets versus full segments.

Segment	Small versus Full						Medium versus Full					
	2 participants			5 participants			2 participants			5 participants		
	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)
ArgoUML - New class new package: Segment 5	62	50	55	100	67	80	50	50	50	50	100	67
JHotDraw - Draw rectangle delete rectangle: Segment 1	62	33	43	0	0	0	73	53	61	20	33	25
JHotDraw - Draw rectangle delete rectangle: Segment 20	78	47	59	67	67	67	88	47	61	67	67	67
Mars - Screen magnifier: Segment 1	44	33	38	67	50	57	62	42	50	25	25	25
Mars - Screen magnifier: Segment 22	38	45	41	0	0	0	57	73	64	20	50	29
Maze - Micro mouse: Segment 4	30	21	25	40	67	50	60	43	50	67	67	67
Maze - Micro mouse: Segment 45	57	80	67	50	100	67	80	80	80	100	100	100
Pooka - New account new e-mail: Segment 52	64	60	62	50	14	22	82	60	69	67	29	40
Pooka - Create folder open folder: Segment 1	33	31	32	25	50	33	64	54	59	33	50	40
Overall	52	44	47	44	46	42	68	56	60	50	58	51

Table 5.12 Examples of labels produced by SCAN and participants.

Examples of labels with low accuracy		
Segment	SCAN label	Participants label
Maze - s9	info description cell template maze model size page painter icon	creating new maze setting its properties creating paths
Maze - s19	count controller step robot	model steps history move done turn
Neuroph - s1	network neural components tree project application neurons component easy folder view	initiate Kohonen view train randomize
Neuroph - s4	draw visualizer kohonen frame application neurons easy	return the view of the current Kohonen
Examples of labels with high accuracy		
Segment	SCAN label	Participants label
Mars - s5	key dump stroke file memory action venus save icon	dump save file item menu action create memory
JHotDraw - s21	unlock standard unfreeze drawing view	unfreeze unlock standard view drawing
Pooka NewAccountNewMail - s6	network connection add listener item manager change	connection manager change item listener add network
Pooka CreateOpenFolder - s15	item connection network	network connection item id

Preservation of Information. To analyse the amount of information lost when reducing the sizes of segments, we calculate the precision and recall of the labels produced with the reduced segments, i.e., the small and medium subsets of the segments, with respect to the labels produced with the full segments. Table 5.11 shows the results. We vary the number of participants considered for agreement and we show results for two and five participants [5].

We observe that, with the increase of the minimum number of participants, the variation of both precision and recall increases too. Thus, a smaller number of participants results in a smaller standard deviation of precision and recall across different segments. The larger standard deviation when increasing the number of participants is due to the small number of synsets in the manual labels of the segments. Thus, the size of the manual oracle decreases when the number of participants increases, which impacts negatively the evaluation of the precision of SCAN because the number of terms in the automatic labels is constant.

Considering the mean values of precision and recall and varying the numbers of participant between two and five, precision for small subsets varies between 44% and 52% and recall varies between 44% and 46%. For medium subsets of the segments, the mean values for precision and recall are greater and vary between 50% and 68% for precision and between 56% and 58% for recall. These results show that we can significantly reduce the number of methods that participants must use to understand a segment, i.e., on average 92% for small and 76% for medium subsets, while keeping about half of the synsets that would appear if the segment was not reduced in size.

Considering small and medium subsets, we loose about half of the information, which explains the difference between labels produced using, on the one hand, medium and small segment, and, on the other hand, labels produced using the full segments.

Participants analysing reduced segments have less information and thus may tend to provide more details regarding the key concepts. However, participants understanding the full segments must provide more effort to extract the key concepts; they may produce more concise labels concerning the key concepts while trying to reach as many concepts as possible.

Preservation of Agreement among Participants. Figure 5.3 shows the mean value and the standard deviation of agreements among participants for the nine segments, considering their full versions, as well as their reduced versions, i.e., small and medium subsets. The only notable decrease in the number of synsets on which participants agree happens when two participants consider a synset as representative for a segment. We tested whether the

5. When six participants is the minimum number required to consider a synset as representative for a segment, the resulting labels for some of the analysed segments are empty.

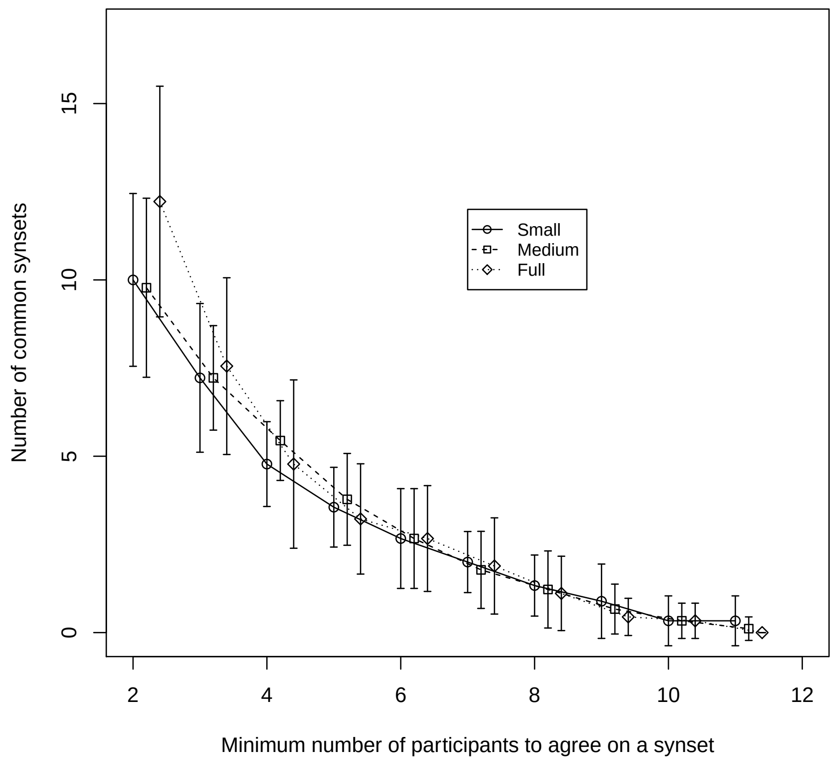


Figure 5.3 Agreement among participants.

agreement was influenced by (1) the number of participants, (2) the size of the provided segment subset (full, medium, small), and (3) their interactions. Results of the permutation test, shown in Table 5.13, indicate that, while there is a significant difference of agreement when considering a different number of participants, as expected, the sizes of the subsets and their interactions with the number of participants do not significantly influence the agreement. Results of the permutation test, shown in Table 5.14, indicate that the experience of the participants do not significantly influence the agreement.

We thus answer **RQ2**: How do the labels of the trace segments produced by the participants change when providing them different amount of information? Small and medium subsets of segments preserve 50% or more of the synsets provided by participants while drastically reducing the amount of information that participants must process to understand a segment. The reduction of size with respect to the original size of the segment is on average 92% for small subsets and 76% for medium subsets.

Table 5.13 Results of two-way permutation test of agreement by number of participants and size of the segment subset (full, medium, small).

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Participants	1	2448.7364	2448.7364	500000	<**0.0001**
Size	2	0.8222	0.4111	156267	0.8915
Participants:Size	2	11.3515	5.6758	500000	0.2062
Residuals	264	968.0566	3.6669		

Table 5.14 Results of two-way permutation test of agreement by experience (high, low).

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Experience	1	1.68	1.6806	65	0.6154
Residuals	70	500.64	7.1520		

RQ3: How do the labels of the trace segments produced by the participants compare to the labels generated by SCAN?

Figure 5.4 shows violin plots for the number of synsets in the manually-labeled segments. Violin plots (Hintze et Nelson, 1998) combine boxplots and kernel density functions, thus showing the shape of a distribution. The dot inside a violin plot represents the median; a thick line is drawn between the lower and upper quartiles; a thin line is drawn between the lower and upper tails. We observe from Figure 5.4 that segments that have been labeled by two participants have a median of 3 common synsets when considering intersection, but with a large proportion of values being concentrated at 2 synsets. We consider this number of common synsets to be a reasonable agreement provided that the median for segments that have been labeled by one participant only is at 5.5, again with high concentration in lower values—5. When considering the union of the synsets the median is higher at 7.

Table 5.15 Precision (P) and Recall (R) of automatic labels assigned by SCAN compared to oracle built by participants.

Program	1 participant only		2 participants intersection		2 participants union	
	P	R	P	R	P	R
ArgoUML	-	-	27%	50%	48%	37%
JHotDraw	-	-	53%	91%	82%	62%
Mars	60%	100%	48%	97%	65%	64%
Maze	60%	53%	18%	92%	32%	45%
Neuroph	28%	48%	-	-	-	-
Pooka	85%	82%	43%	93%	74%	73%
Averages	58%	71%	38%	85%	60%	56%

Table 5.15 reports the results of evaluating the automatic labels when considering both

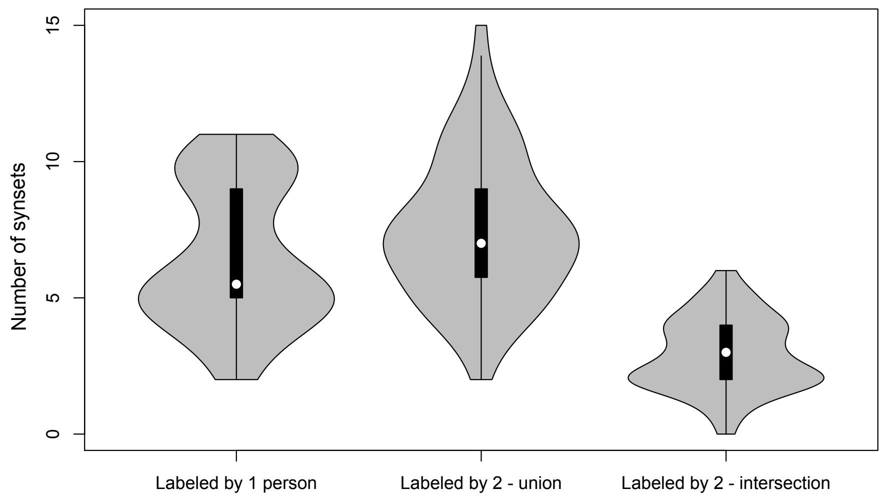


Figure 5.4 Number of synsets in manual labels.

operators. When considering all programs, the average values for precision and recall for segments labeled by one participant are 58% and 71%, respectively. For segments labeled by two participants, precision and recall values are 60% and 56% on average, respectively. Finally, when we consider the intersection of synsets for segments labeled by two participants, the recall is high, 85% on average, but precision can be as low as 18% with an average of 38%. This low precision is partially due to the lower number of synsets in the manually-labeled segments compared to the number of terms in the labels generated by SCAN. As shown in Figure 5.4, the median for the number of synsets in manual labels is three when intersection is considered. However, SCAN generates 10 terms for any label.

Table 5.12 shows examples of labels assigned by SCAN and the corresponding labels assigned by the participants. The first part of the table shows labels for which both precision and recall are low ($\leq 40\%$). The second part shows cases where SCAN has a precision and recall greater than 75%.

Term Frequency. Table 5.16 reports the results of evaluating the automatic labels using term frequency when considering both operators. When considering all programs, the average of precision and recall for segments labeled by one participant are the same as for SCAN labels. For segments labeled by two participants, overall the average of precision and recall for term frequency are lower than the average of precision and recall for SCAN when we consider the intersection. For segments labeled by two participants, the average of recall for term frequency is higher than the average of recall for SCAN and the average of precision for term frequency is the same as the average of precision for SCAN when we consider the union. We analyse in depth the labels produced using term frrequency, labels generated by SCAN and labels produced by participants and we conclude that the increase of the precision and recall for Maze program is due to the term *maze*. The term *maze* is selected as a top term in the majority of the segments of the execution trace of Maze program using term frequency but using SCAN (i.e., $TF\text{-}IDF$) the term *maze* is selected in some segments. Table 5.17 shows an example of label of the segment s50 of Maze program assigned using term frequency, SCAN label and the corresponding labels assigned by two participants (i.e., P1 and P2). One of the two participants believe that the term *maze* is significant for the segment s50 of Maze program but the second participant did not mention it. For this reason, precison and recall for term frequency is higher than precison and recall using SCAN for Maze program when we consider the union. Table 5.17 shows also that term frequency filter some relevant terms that SCAN and participants identify as relevant terms for segments such as the term *explore*. Thus, we could conclude that SCAN base on $TF\text{-}IDF$ gives better results than using term frequency to assign labels to execution trace segments.

Table 5.16 Precision (P) and Recall (R) of labels assigned using term frequency compared to oracle built by participants.

Program	1 participant only		2 participants intersection		2 participants union	
	P	R	P	R	P	R
ArgoUML	-	-	27%	50%	48%	37%
JHotDraw	-	-	53%	91%	82%	62%
Mars	60%	100%	48%	97%	65%	64%
Maze	60%	53%	15%	75%	34%	52%
Neuroph	28%	48%	-	-	-	-
Pooka	85%	82%	42%	92%	73%	72%
Averages	58%	71%	37%	81%	60%	58%

Table 5.17 Example of labels of the segment s50 of Maze program produced using term frequency and participants.

Term frequency label	SCAN label	Participants label
cell location wall robot current floodfill master direction maze model	current direction wall cell explored master location robot floodfill model	P1: move direction explore robot
		P2: check wall explore direction draw maze
		Intersection: direction explore
		Union: move robot check wall explore direction draw maze

We thus answer **RQ3**: How do the labels of the trace segments produced by the participants compare to the labels generated by SCAN? SCAN automatically assigns labels with an average precision and recall of 69% and 63%, respectively, when compared to manual labels produced by merging the labels of two participants using union.

5.3.4 Threats to Validity

This section discusses the threats to the validity of our evaluation and explains how we tried to mitigate them when possible.

Construct validity Construct validity concerns the relation between theory and observations. Our theory is that participants consistently understand execution traces and, thus, that an automated approach can accurately segment execution traces and label trace segments. In the preliminary evaluation, to compare manual and automatic labels, we validated the terms having the same stem but we did not consider that manual and automatic terms could be synonymous. We considered other type of relations among manual and automatic terms in Section 5.3. Moreover, for the preliminary evaluation, we manually labeled segments ourselves which may bias our evaluation of the labels generated by SCAN. For this reason, we performed a study with participants to validate SCAN labeling segments in Section 5.3.

Threats to the construct validity of our evaluation could mainly be due to our evaluation of the capability of SCAN to label segments. For the former, we compare the automatically generated labels with the manually generated labels in **RQ3**. To limit bias due to subjectiveness, we also report results obtained by applying union or intersection over labels produced by multiple participants.

The participants of the experiment are not the original developers of the studied programs. To address this threat to validity, we asked more than one participant to manually label the same segment. Note also that developers of large software programs may not be familiar with the entire program and thus would have been subject to the same threat.

In **RQ2**, we show that using an approach based on $TF\text{-}IDF$ to identify terms for labeling segments makes sense, as labels produced by participants when using reduced segment subsets with the most frequently invoked methods are not significantly different from those obtained when having the full segments available.

The terms of the labels are extracted from method signatures that represent low-level information. We extract the relevant terms describing the concepts implemented by segments and thus we are switching from a low-level description that contains all the terms extracted from method signatures to a high-level description that contains only the relevant information of the extracted terms.

Internal Validity The internal validity of an evaluation is the extent to which a treatment changes the dependent variable. Threats to internal validity could be due to the presence, in the execution traces, of extra method invocations related to GUI events or other program events. Also, the order of invocation in different executions may depend on multi-threading. This may affect $TF\text{-}IDF$ values and could produce different results in terms of relevant information. The frequency-based pruning and the analysis of different execution trace instances for one scenario mitigate these threats.

The internal validity of our empirical evaluation could be threatened by our choice of the traces to segment and label as well as related thresholds (e.g., the threshold used to merge two segments). We mitigated this threat by using different traces obtained from executing different scenarios on different programs. Also, participants confirmed the precision and recall of the segment labels.

External Validity Although in the preliminary evaluation we applied our approach on traces from two different programs, further studies on larger traces and more complex programs are needed, especially to better demonstrate SCAN accuracy in assigning labels representative of concepts implemented by trace segments. For this reason we performed a study in Section 5.3 using different programs. Our choices reduce the threat to the external validity of our empirical evaluation. As explained in Section 5.3.1, participants involved in the evaluation of the performances of SCAN are not original developers of the analysed programs, hence results might be different when considering people having a better knowledge of the programs.

Conclusion validity Conclusion validity threats deal with the relation between the treatment and the outcome. Wherever appropriate, we use statistical tests to support our claims. Specifically, for **RQ2**, we use permutation test, which is a non-parametric alternative to ANOVA, hence it does not require data to be normally distributed.

5.4 Summary

Our tool SCAN aims at supporting developers to discover concepts in segments of execution traces by assigning sets of words to each segment. SCAN has been conceived and developed with the trace segmentation approach presented in Asadi *et al.* (2010b); Medini *et al.* (2011) in mind. However, it is not tied to any specific trace segmentation approach. We presented a preliminary evaluation investigating the accuracy and effectiveness of SCAN in assigning meaningful sets of words representative of the concepts implemented in segments. We performed a manual validation on several traces of both JHotDraw and ArgoUML, two known Java programs, often used as a benchmark in software engineering research. JHotDraw and ArgoUML are small enough to allow manual validation, still they are real programs of non trivial size. ArgoUML in particular is a real world application with a large user community and is actively maintained by several developers. We performed both a qualitative and a quantitative validation aiming at verifying the relation between manually defined labels and segment labels automatically generated by SCAN. Quantitative analysis shows different ranges of similarities between manual and automatic labels. Manual inspection of several examples of the automatically produced label sets indicates that these are quite informative and useful to reconstruct the target concepts associated with each segments. So the relatively low similarity values should not be interpreted as poor performance. On the contrary, our qualitative analysis indicates that such performance is sufficient for manual concept assignment and phase recognition. In summary, we can claim that SCAN was successful in assigning labels very similar to manually defined labels and that these labels actually correspond to the concepts encountered in the segment based on source code documentation and method execution. Moreover, in the preliminary evaluation, we manually labeled segments ourselves, which may bias our evaluation of the labels generated by SCAN. Consequently, we conducted a study aiming at analysing the ability of SCAN to accurately reduce the size of segments and identify labels.

In this study, we ask 31 participants (professionals and students) to assign labels to segments extracted from six Java programs (ArgoUML, JHotDraw, Mars, Maze, Neuroph, and Pooka). First (**RQ2**), we investigate whether providing the participants with the most relevant methods only is sufficient to understand segments; we compare the quality of the labels and participant agreement with those obtained when full segments are used to produce labels. Then (**RQ3**), we compare manually-produced labels for 210 segments with those produced by SCAN. Results of the empirical evaluation confirmed the ability of SCAN to select the most representative methods of a segment, thus reducing on average 92% of the information that participants must process while guaranteeing that close to 50% of the knowledge is preserved (**RQ2**)—the labels produced by participants when analysing the reduced segments contained 50% of the information of the labels produced from the original segments. Results also showed that SCAN can automatically label segments with 69% precision and 63% recall when compared to the manual labels produced by the participants (**RQ3**).

Results showed that SCAN was successful in assigning labels very similar to labels manually-defined by participants and that these labels actually correspond to the concepts encountered in the segment based on source code, documentation, and method execution. Results also showed that we provide the relevant information on the concept implemented by each segment, helping developers to understand the concept implemented by each segment.

CHAPTER 6

Relating Segments

We split execution traces into segments allowing developers to focus on segments to maintain instead of analysing the entire execution trace. Segment labeling provides a description of the concepts implemented in each segment. However, developers would benefit to know the relations among segments and concepts. We propose to identify different relations between segments to help developers identify the segments that implement the concepts to maintain.

In this chapter, after defining the approach relating segments in details, we perform a qualitative analysis to validate the relations identified automatically by only one participant. Then, we improve this evaluation with an experiment to assess SCAN capability to identify relations among segments in comparison to 31 participants. We describe the design of the experiment and how we analyse results. Finally, we present and discuss the results of the evalutation.

6.1 Relating Segments Approach

As depicted in Figure 1.2 in Chapter 1, SCAN consists in a series of four steps. In Step 1, SCAN uses execution traces to identify segments by finding cohesive and decoupled fragments of the trace. The trace segmentaion approach based on dynamic programming is presented in Chapter 4. Then, in Step 2, it merges similar segments using the Jaccard measure on terms extracted from the segments. After that, in Step 3, it uses an IR-based approach to label segments. Step 2 and 3 are presented in Chapter 5. This chapter presents Step 4, which uses FCA to identify relations among segments.

While we expect the labels produced in Step 3 to fully describe the concept implemented by a segment, they do not help developers to relate segments in a same trace with one another. For example, segments with identical labels may appear multiple times, in different trace regions. Furthermore, two segments may share many terms, which could possibly indicate the existence of a higher-level concept common to both segments. To discover such relations among segments, SCAN uses Formal Concept Analysis (FCA) and highlights commonalities and differences among segments by identifying terms shared between multiple segment labels and terms that are specific to particular segment labels.

As described in Section 2.4 in Chapter 2, FCA groups *objects* that have common *attributes*. In SCAN, objects are segments and attributes are the terms of the segments labels. The binary relation states which term is included in which label. A FCA concept is a cohesive set of segments sharing some terms in their labels. Figure 6.1 shows an example of a FCA lattice for the ArgoUML scenario "add a new class" in which each node represents a formal concept (X, Y). SCAN uses the lattice to identify relations among segments as explained in the following.

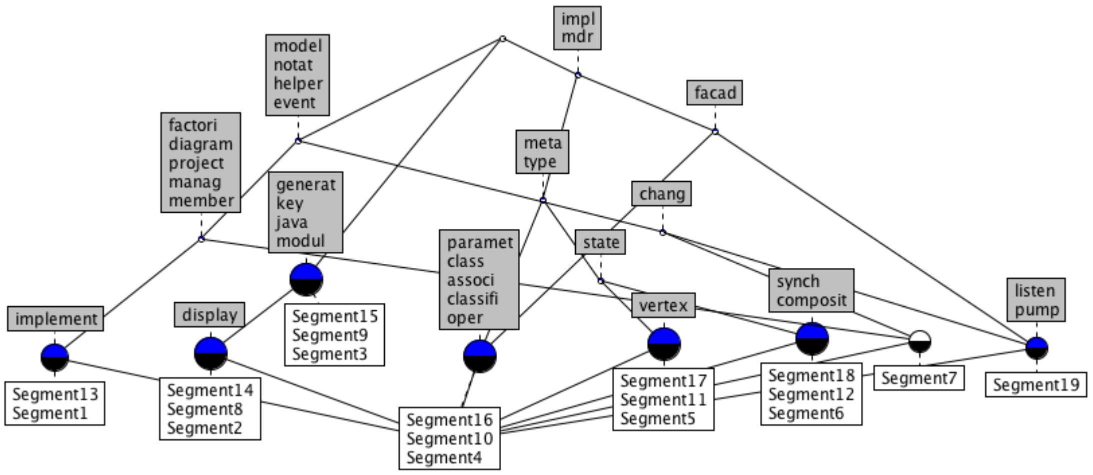


Figure 6.1 ArgoUML FCA lattice for the scenario "add a new class".

Types of Relations. By applying FCA on the segments and terms from their labels and analysing the resulting lattices, we identified the following relations among segments: same phase, sub/super concept, and macro-phase.

Two distinct segments sharing the same relevant terms are considered to activate the same concept, thus forming a phase. For example, in Figure 6.1, SCAN identifies Segments 2, 8, and 14 as part of the same phase because these segments belong to the same concept. These three segments share the same terms and actually activate the same concept.

Sub-concept relation exists when a set of segment(s) activate part of a concept of another set of segment(s). SCAN identifies a sub-concept relations between two segments when relevant terms in the label of one segment are a superset of the terms of the label of another segment, i.e., by selecting the intent of a concept of interest in the lattice. For example, in Figure 6.1, Segments 2, 8, and 14 share the terms "*generat, key, java, and modul*" with the Segments 3, 9, and 15 and thus are a sub-concept of these segments. Conversely, a super-concept relation exists when terms in the label of one segment are a subset of the terms of another.

A macro-phase is the result of the abstraction of repeated sequences of identical phases, which activates a set of concepts. SCAN identifies macro-phases by finding repeating sequences of FCA concepts. For example, in Figure 6.1, there are several phases such as: Phase 2: Segments 2, 8, and 14; Phase 3: Segments 3, 9, and 15; Phase 4: Segments 4, 10, and 16; Phase 5: Segments 5, 11, and 17; and Phase 6: Segments 6, 12, and 18. A segment activates the phase that it belongs to, thus, for example, Segment 2 activates Phase 2. The next segment in the trace, Segment 3, activates Phase 3. In the same way segments 4, 5, and 6 activate respectively Phases 4, 5, and 6. Thus, the sequence of Segments 2 to 6 activates the sequence of Phases 2 to 6. However, the same sequence of phases is also activated with the sequence of Segments 8 to 12 as well as with the sequence of Segments 14 to 18. Thus, the three sequences of Segments 2 to 6, Segments 8 to 12, and Segments 14 to 18, activate the same concepts, i.e., activate the concepts of Phases 2 to 6. SCAN abstracts those five phases and identifies the macro-phase containing Phases 2 to 6.

Sequence Diagram. The FCA lattice shown in Figure 6.1 can be used by developers in the more familiar form of a UML sequence diagram. To obtain a sequence diagram from the FCA lattice, segments are considered in the order in which they appear in the execution trace. Each segment is associated with its most specific FCA concepts in the FCA lattice. Methods are activated in the sequence diagram for each FCA attribute of the segment-specific FCA concepts. The topmost reachable FCA attributes are activated first and all FCA attributes in the sub-FCA concepts are activated as nested operations.

A portion of the sequence diagram for the FCA lattice in Figure 6.1 is shown in Figure 6.2 (generated using PlantUML[1]). The sequence diagram shown in Figure 6.2 contains the same information available from the FCA lattice, but the sequential ordering of the called method makes it easier to read and understand for developers.

Segment 1 is associated with a FCA concept that has three super-FCA concepts, two of which are annotated with FCA concept-specific attributes. Starting from the topmost annotated FCA concept, the following methods are activated in the sequence diagram: *"model, notat, ..."*, *"factori, diagram, .."*, and *"implement"*. Similarly, Segment 2 activates *"generate, key, ..."*, which has a nested activation labeled *"display"*, while Segment 3 activates only *"generate, key, ..."*. For the sake of clarity, only a portion of the method calls in Segment 4 are shown.

1. http://plantuml.sourceforge.net/

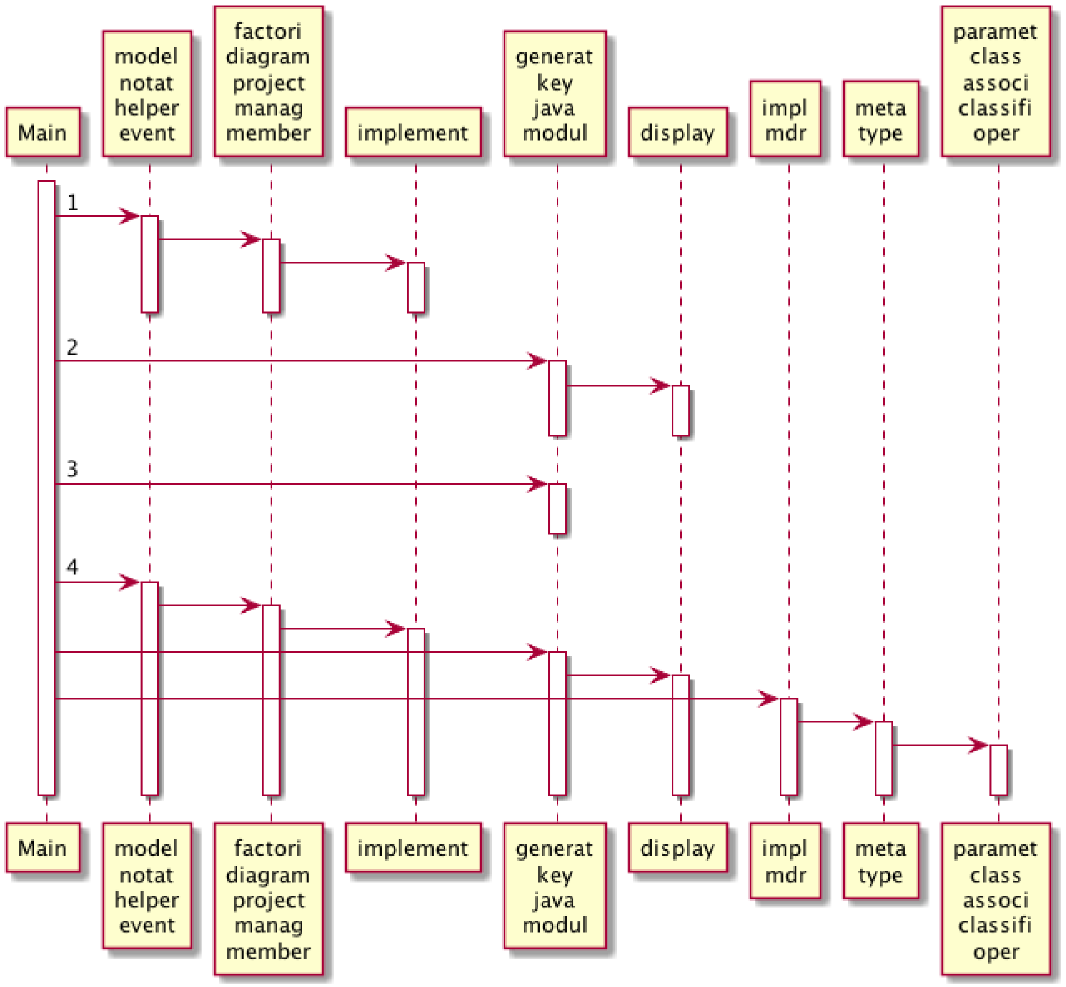


Figure 6.2 ArgoUML sequence diagram derived from the FCA lattice for the scenario “add a new class”.

6.2 Preliminary Evaluation

We aim to evaluate SCAN capabilities to identify relations among segments. In this evaluation, we reuse the scenarios previously used to validate segments labels reported in Tables 5.1 and 5.2 in Chapter 5.

The study aims at answering the following research question:

RQ1. *Does SCAN help to discover relations between segments? Does it help to discover the macro-phases in a trace?* This research question is formulated to verify the efficiency of SCAN when relating segments. To address this research question, we analyse the lattice produced by FCA to identify relations between different segments.

6.2.1 Results and Discussions

In the following, we report results aimed at addressing **RQ1**. We exploited FCA to identify linguistically overlapping segments. In other words, segments having the same or shared labels implement similar or related concepts. By looking at Figure 6.3 we can notice that, for example, segments 4 and 23 are identical and implement the same concept. This was confirmed by manual inspection of the source code. A developer can therefore use lattice information to infer relations between segments and identify segments implementing the same concept. We can also notice that sometimes a computation phase, represented as an FCA concept, is contained in a more abstract one. For example, in Figure 6.3 segments 28 and 30 are contained in a super-concept of the concept containing segments 26 and 31. In fact, they all share some labels (*listen, change, remove, figure*), but the latter segments (26, 31) have their own specific labels (*intern, multicast*).

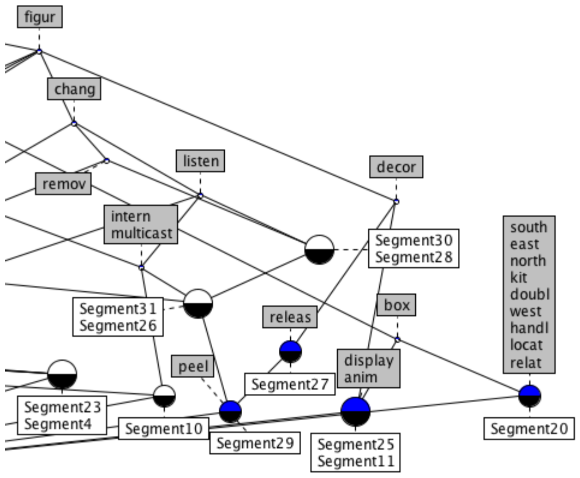


Figure 6.3 Excerpt of the JHotDraw FCA lattice for the scenario “Draw Rectangle, Delete Rectangle”.

Figure 6.1 shows the FCA lattice for the execution trace of the scenario “New Class”. As for JHotDraw, also for ArgoUML FCA helps to highlight relations between segments. For example, segments 4, 10 and 16 implement the same concept. The concept containing segments 3, 9 and 15 is a sub-concept of the one containing segments 2, 8 and 14 and in fact it points to higher level concepts (*generate key java module*), while the super-concept includes segments specific of the *display* functionality.

To identify macro-phases in a trace, we consider relations between cohesive sets of segments, regarded as execution phases. A macro-phase is built by repeated segments in a trace. For example, in Figure 6.1, segments 2, 3, 4, 5 and 6 define an execution phase on the trace and this phase is repeated two times: first with segments 8, 9, 10, 11 and 12, and then with segments 14, 15, 16, 17 and 18. The rest of the segments are also converted to an execution phases.

Qualitative results indicate that the automatically-produced labels, organised into a concept lattice where similar or identical segments are grouped together, are extremely useful to understand commonalities and differences between segments and to extract a view where macro-phases can be labeled by the terms associated with the super-concepts of the involved segments. Cohesive sets of similar segments can be identified in the concept lattice. Such sets, in turn, define macro-phases, that labeled with super-concept terms. The temporal ordering of the segments involved in different macro-phases suggests the temporal organization of the recognized phases. We think this has huge potential in supporting comprehension of complex execution scenarios for large programs.

Flow Diagram of Phases. After defining the phases we can draw a higher level flow diagram of phases with labels as shown in Figure 6.4, using the temporal relations between phases. The "New Class" scenario, generating 19 segments, can be summarised into four macro-phases. The first phase deals with the program startup, this is followed by activity needed to create class and properties (e.g., state, composite, etc). The third phase is devoted to managing diagram events and, finally, the last phase models *add events* and *model changes*.

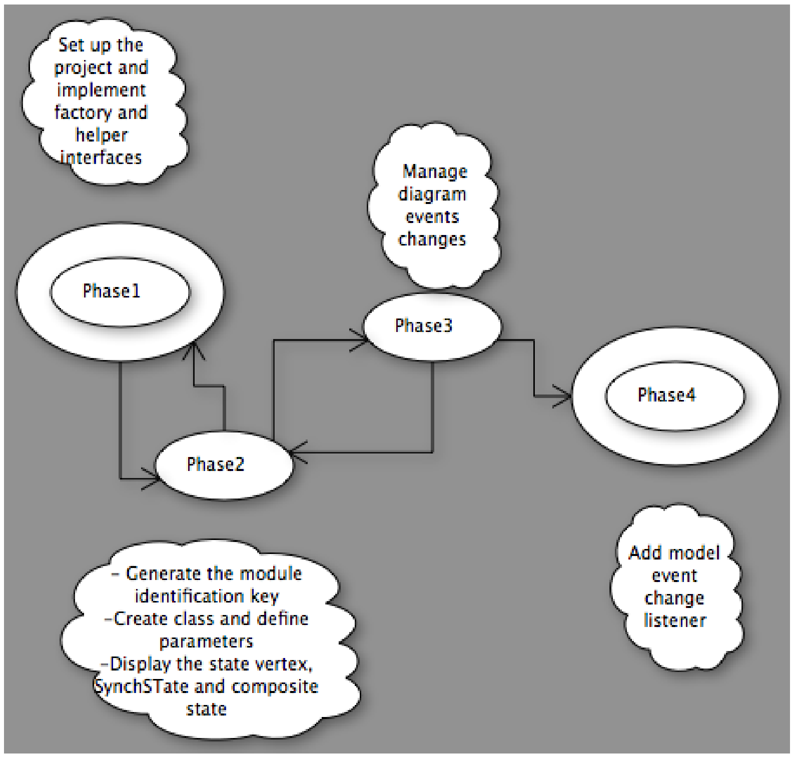

Set up the project and implement factory and helper interfaces
Manage diagram events changes
Phase1
Phase3
Phase4
Phase2
- Generate the module identification key
-Create class and define parameters
-Display the state vertex, SynchSTate and composite state
Add model event change listener

Figure 6.4 Flow diagram of phases for the scenario "New Class".

6.3 Relating Segments Evaluation

In the preliminary evaluation, a developer should analyse a concept lattice to discover relations. However a large concept lattice is difficult to analyse to identify concept relations. For this reason, we extend our approach to identify automatically relations among trace segments. Moreover, in the preliminary evaluation, we validated the relations among segments ourselves, which may bias our evaluation. For this reason, we conduct a study in which we reuse the set up of the experimental evaluation of labeling segments detailed in 5.3.1 in Chapter 5. Table 5.6 summarises characteristics of the programs, i.e., their sizes (in terms of lines of code), a short description of the scenarios used to generate the execution traces, and the sizes of the traces (in terms of number of executed events, i.e., constructor and method calls). Table 5.8 provides descriptive statistics of the participants' programming experience.

6.3.1 Experimental Design and Analysis

The evaluation aims at answering the following research question: **RQ2:** *To what extent does SCAN correctly identify relations among segments?* This research question is formulated to verify the efficiency of relating segments and thus providing an accurate high-level description of the concepts implemented in an execution trace. This research question assess the use of FCA by scan to identify relations among segments. We ask participants to assess the relations among segments identified by SCAN.

In the following, we describe the evaluation design and procedure followed to answer the research questions.

To address **RQ2**, we ask the participants to validate all relations among segments identified by SCAN. We do not ask participants to manually identify the relations among segments for two reasons. First, identifying the relations requires to compare each segment in a trace with any other segments, taking into account the possible reordering of method calls as well as inclusions. Thus, such a task would have been very demanding for the participants. Second, identifying these relations is not a task commonly undertaken by participants and, thus, its results would have been of a quality inferior to that of the labeling task, which participants perform implicitly or explicitly when understanding a segment. Participants validating the relations between segments use the full segments to understand and label the segments. Next, they use the labels that they just produced and the comprehension they have gained to validate the possible relations. We provide definitions for the different types of relations between segments: same concept (phase) or sub/super concept. For macro-phases, we ask the participants to validate all the phases of a given macro-phase. If all phases participating in a sequence of SCAN phases is validated by the participants then the macro-phase is also considered valid by construction.

We report the accuracy, i.e., the percentage of relations that SCAN has correctly identified as vetted by the participants. We also report separately the accuracy for SCAN capability to identify super/sub-concept relations because of the participants' difficulty to distinguish sub-, super-, and same-concept as illustrated by the following example. Table 6.1 shows two segments from JHotDraw labeled by SCAN and by two participants, Participant I and Participant J. Both SCAN and Participant I identify Segment 9 as activating a sub-concept of the concept activated by Segment 10. However, according to Participant J, the two segments activate the same concept. This example shows that distinguishing between sub/super concept and same concept is difficult. Therefore, when presenting the results, we report the percentages of agreements between the participants and SCAN with and without distinguishing between sup/super concept and same concept.

Table 6.1 Example of relations among segments.

	Segments	Labels	Relations
SCAN	9 10	listener, add, change, figure figure, listener, add, internal, multicaster, event, change	sub/super concept
Participant I	9 10	composite, figure, trigger, event manage, figure, change, event, trigger	sub/super concept
Participant J	9 10	abstract, figure, change, add, listener figure, change, event, multicaster, add, listener	same concept

6.3.2 Experiment Results and Discussions

This section reports the results of our experimental evaluation to address **RQ2**.

Figure 6.5 shows an excerpt of the Pooka FCA lattice for the scenario "New account new e-mail" and Table 6.2 shows some of the relations identified by SCAN for this scenario. For example, SCAN identifies that Segments 16, 28, and 41 form a phase, i.e., Phase 16 in Table 6.2, as they all activate the same concept, which is loading the state through the wizard editor pane. SCAN automatically labels this phase as "load state wizard editor pane".

Hence, Segment 16 activates Phase 16. The next segment in the trace, Segment 17, activates Phase 17. In the same way Segments 18, 19, and 20 activate respectively Phases 18, 19, and 20. Thus, the sequence of Segments 16 to 20 activate the sequence of Phases 16 to 20. However, the same sequence of phases is also activated with the sequence of Segments 28 to 32 as well as with the sequence of Segments 41 to 45. Thus, the three sequences of Segments 16 to 20, Segments 28 to 32, and Segments 41 to 45, activate the same concepts, i.e., activate the concepts of Phases 16 to 20. Thus, SCAN identifies a macro-phase from the repeated execution of Phase 16 $\rightarrow$ Phase 17 $\rightarrow$ Phase 18 $\rightarrow$ Phase 19 $\rightarrow$ Phase 20.

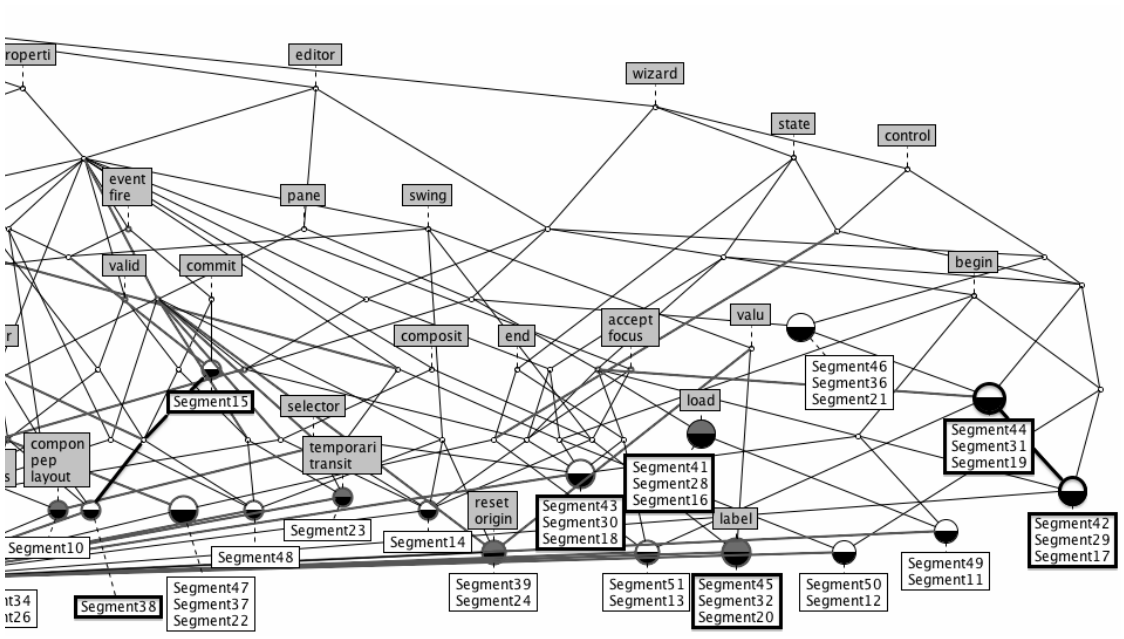


Figure 6.5 Excerpt of the Pooka FCA lattice for the scenario "New account new e-mail".

Considering the automatic labels produced by SCAN, we observe that Phase 17 and Phase 19 activate concepts pertaining to the state of the wizard editor pane. However, Phase 17 is more specific as it is concerned with beginning states. SCAN identifies a sub/super-concept relation between Phase 17 and Phase 19, as shown in Figure 6.5. SCAN also reports that Segment 38 has a super-concept relation with Segment 15. The former raises different property events, including property committing events, in common with Segment 15.

Table 6.3 reports the number of relations identified by SCAN in the six programs (we do not report numbers for Neuroph, as no relation was identified among its segments), with and without the numbers of sub/super relations. SCAN identifies 100 relations: 59 sub/super, 7 macro-phase, and 34 same concept relations. An agreement between SCAN and the participants occurs when the same relation is identified by SCAN and at least one of the participants. We do not show results when both participants agree, as the number of cases in which participants disagree is low (six and eight relations in case of no distinction and distinction, respectively).

We can conclude that, depending on whether we distinguish sub/super relations or not, the overall accuracy of SCAN in identifying relations between segments is 96% and 63%, respectively. When evaluating relations with distinction, the precision of SCAN is greater than 75% in the majority of the programs. The two exceptions are ArgoUML and Mars, and for both of these programs, the proportion of detected sub-concept relations is extremely high with respect to other relation, i.e., 100% and 82% for ArgoUML and Mars, respectively.

Table 6.2 Examples of relations detected by SCAN for Pooka, scenario "New account new e-mail".

Phases	**Segments in the Phase**	**SCAN Labels**
Phase 16	Segments 16, 28, and 41	load state wizard editor pane
Phase 17	Segments 17, 29, and 42	wizard state controller editor pane beginning
Phase 18	Segments 18, 30, and 43	set end wizard focus accept state editor pane property beginning
Phase 19	Segments 19, 31, and 44	controller wizard state pane editor
Phase 20	Segments 20, 32, and 45	composite focus accept label editor swing property
Macro-phases	**Involved Phases**	**Number of Repetitions**
Macro-phase 1	Phases 16, 17, 18, 19, and 20	The sequence of phases is repeated 3 times
Sub-concepts	**Segments/Phases involved**	**Details**
Sub-concept 5	Phase 17 activates a sub-concept of Phase 19	Both activate concepts regarding state of the wizard editor pane. Phase 17 is specific to beginning state.
Sub-concept 10	Segment 38 activates a sub-concept of Segment 15	Both fire property committing events. Segment 38 fires additional property events.

Table 6.3 Evaluation of the automatic relations.

Programs	Relations identified by SCAN	Sub/Super concept relations	Agreements with participant(s) without distinction btw. sub/super relations	Agreements with participant(s) with distinction btw. sub/super relations
ArgoUML	6	6	100%	33%
JHotDraw	9	5	100%	100%
Mars	22	18	100%	9%
Maze	12	1	100%	83%
Pooka	51	29	82%	78%
Total	100	59	96%	63%

For ArgoUML, the majority of the detected relations involve class `MetaTypesMDRImpl`, which retrieves objects that represent the different UML types. SCAN labels Segment 2 as "*mdr meta impl types*", which is the general concept implemented by the class. SCAN labels the rest of the segments by specifying additional terms, e.g.,"*composite state meta impl synch mdr types*" for Segment 11. Thus, Segment 2 is identified as the super-concept of five other segments, as SCAN considers them as addressing more specific concepts. Both participants labeling ArgoUML traces produced similar label for Segment 2, e.g., "*implementation dependent UML class type*" but also for the rest of the segments and, thus, considered all segments to implement the same concept. We observe a similar phenomenon for Mars, as shown in Table 6.3, i.e., when no distinction is made participants agree 100% with the identified relations.

We thus answer **RQ2**: To what extent does SCAN correctly identify relations among segments? SCAN identifies relations among segments with an overall precision of 63% and a precision greater than 75% in the majority of the programs.

6.3.3 Threats to Validity

Although our approach performed well, there are some aspects that can impact its efficiency.

Construct validity Construct validity concerns the relation between theory and observations. Our theory is that participants consistently understand trace segments and, thus, that an automated approach can accurately relate execution trace segments. Threats to the construct validity of our evaluation could mainly be due to our evaluation of the capability of SCAN to identify relations between them. To limit bias, we report results obtained by multiple participants. The participants of the experiment are not the original developers of the studied programs. To address this threat to validity, we asked more than one participant to validate the same segment relation. Note also that developers of large software programs may not be familiar with the entire program and thus would have been subject to the same threat. We asked participants to validate relations among segments and, because they do not know how our approach works, their bias is limited.

Internal Validity The internal validity of an evaluation is the extent to which a treatment changes the dependent variable. The internal validity of our evaluation could be threatened by our choice of the segments to relate. We mitigated this threat by using segments from different traces obtained from executing different scenarios on different programs. Also, participants confirmed the precision and recall of the relations among the segments.

External Validity The external validity of an evaluation relates to the extent to which we can generalize its results. In the preliminary evaluation, we applied our approach on traces from two different programs, further studies on larger traces and more complex programs were needed. We performed a study in Section 6.3. Our choices reduce the threat to the external validity of our empirical evaluation. As explained in Section 5.3.1, participants involved in the evaluation are not original developers of the analysed programs, hence results might be different when considering people having a better knowledge of the programs.

6.4 Summary

Our tool SCAN aims at supporting developers to discover concepts in segments of execution traces by identifying relations among segments via formal concept analysis. It also supports grouping segments into macro-phases. We presented a preliminary evaluation investigating the accuracy and effectiveness of SCAN in identifying relations among segments. We performed a manual validation on several traces of both JHotDraw and ArgoUML. We performed a qualitative validation to verify the relations among segments generated by SCAN. Results shows that SCAN detect relations between segments, to easily identify repeated computational phases and to abstract them into macro-phases execution traces. In the preliminary evaluation, a developer should analyse a concept lattice to discover relations. However, for a large concept lattice it would be difficult to analyse and to identify concept relations. For this reason, we extended our approach to identify automatically relations among segments. Moreover, in the preliminary evaluation, we validated relations among segments ourselves which may bias our evaluation. Thus, we conducted a further study in which we asked 31 participants (professionals and students) to validate relations among segments extracted from six Java programs (ArgoUML, JHotDraw, Mars, Maze, Neuroph, and Pooka). We use SCAN to identify relations among segments and ask participants to validate them. Results of the empirical evaluation confirm that participants agreed at 63% with the identified relations among segments identified by SCAN (**RQ2**).

Results confirmed that SCAN detects automatically relations between segments, accurately, to help identify repeated computational phases and to abstract them into macro-phases. Results showed also that SCAN can provide a high-level representation of the concepts implemented in the entire execution trace by identifying different relations between segments and abstracting them into macro-phase. We evaluated the accuracy of SCAN with participants but a key evaluation point must be on its usefulness.

CHAPTER 7

SCAN Usefulness Evaluation

During maintenance, developers generally are interested to understand some segments of a trace that implement concepts of interest rather than to analyse in depth the entire execution trace. We assess the usefulness of SCAN to group these concepts of interest in few segments and thus ensure that our trace segmentation approach facilitates maintenance tasks. In addition, we assess the usefulness of SCAN to guide developers towards segments that implement the concepts to maintain and reduce the number of methods to investigate. We also present the evaluation of the usefulness of SCAN through an empirical study. Finally, we report and discuss the results.

7.1 Applying SCAN to Support Concept Location

According to Dit *et al.* (2013b) concept location is "*one of the most frequent maintenance activities undertaken by developers because it is a part of the incremental change process (Rajlich et Gosavi, 2004)*". Given the importance of concept location in the context of software maintenance tasks, we further explore how trace segmentation and labeling performed by SCAN can be used to support concept location to help developers in their everyday activity.

Consider a concept location techniques that uses dynamic analysis, as the Single Trace and Information Retrieval approach (SITIR) proposed by Liu *et al.* (2007), which combines dynamic analysis and textual analysis based on Latent Semantic Indexing (LSI). Given a change request—e.g., bug description—and an execution trace, the approach proposed by Liu *et al.* (2007) ranks methods of the source code that appear in the execution trace based on their textual similarity with the change request—e.g., the bug description or title. It is important to point out that concept location techniques aim at finding a starting point of the modification, i.e., the "seed"—a method in the source code that is relevant for the change request and where developers will start the necessary modifications to implement the change request. The motivation for that is because, once the seed is known, the developer can identify the other methods that would be impacted by any change related to such a concept, e.g., impacted by a bug fixing activity.

For this reason, the effectiveness of a concept location technique is evaluated in terms of the number of methods in the ranked list produced by the technique that a developer has to scrutinize before reaching any method belonging to the impact set of the concept. Such a method would be the seed for a modification. In order to perform this kind of evaluation, we require the availability of a gold set, i.e., the set of all methods (and those methods only) that a developer should modify in order to fix a bug. The lower the number of methods to explore before finding the seed, the better the technique.

In this context, we are interested to evaluate whether SCAN can be used to reduce the burden of developers when identifying the set of methods impacted by a concept, once a concept location technique identifies one of these methods (i.e., the seed). The conjecture is that methods related to a concept should be contained in one or few segments. Hence, to analyse the concept impact set, a developer could only focus on one or few segments instead of looking at the entire execution trace. In addition to that, we also want to investigate whether, instead of relying on concept location techniques, SCAN can be used as a standalone technique to automatically identify segments relevant for a query.

7.1.1 Typical Scenario

Figure 7.1 shows an example of a bug report for JabRef and the top 5 ranked methods produced by SITIR. In a typical scenario the developer assigned to implement the changes will start by looking into the first method of the ranked list—i.e., method `isiAuthorsConvert(String)` defined in *IsiImporter*—by trying to understand the source code and–or execution trace. The execution trace in this particular case consists of $13,616$ method calls.

To ease the analysis of the execution trace, a developer can use SCAN to segment it. Figure 7.1 shows one segment of this trace—the segment containing the top 1 ranked method provided by the concept location technique. The segment shows the methods in their order of execution, thus method `isiAuthorsConvert(String)` occurs two times (in positions 2 and 16) as it was executed twice. Since SCAN's segmentation is guided by the textual cohesion between methods, segment 4 can be regarded as the smallest, highly cohesive part of the trace activating the problematic concept that provides the context in which the top 1 ranked method is executed. In other words, rather than considering the entire execution trace, one may limit the context of a method to the segment in which it appears. From the methods of segment 4, one can quickly grasp that the author conversion is indeed performed in the context of importing a bibliographical entry. A further analysis of the methods contained in segment 4 reveals that the segment also contains a couple of other gold set methods—appearing at positions 1, 3, 4, 6, 7, 17, 18, 20, and 21. Indeed, the developer fixing the wrong author import also fixed other related problems in the import of an ISI entry—the parsing of month and pages in particular. Thus, we conjecture that methods relevant to a change request are grouped in few segments. If this is indeed the case, SCAN can be used to reduce the analysis of the trace to the analysis of few segments, i.e., reduce the number of methods to be analysed. This assumes that a concept location technique is used to guide the search and is able to retrieve the segments containing the gold set methods.

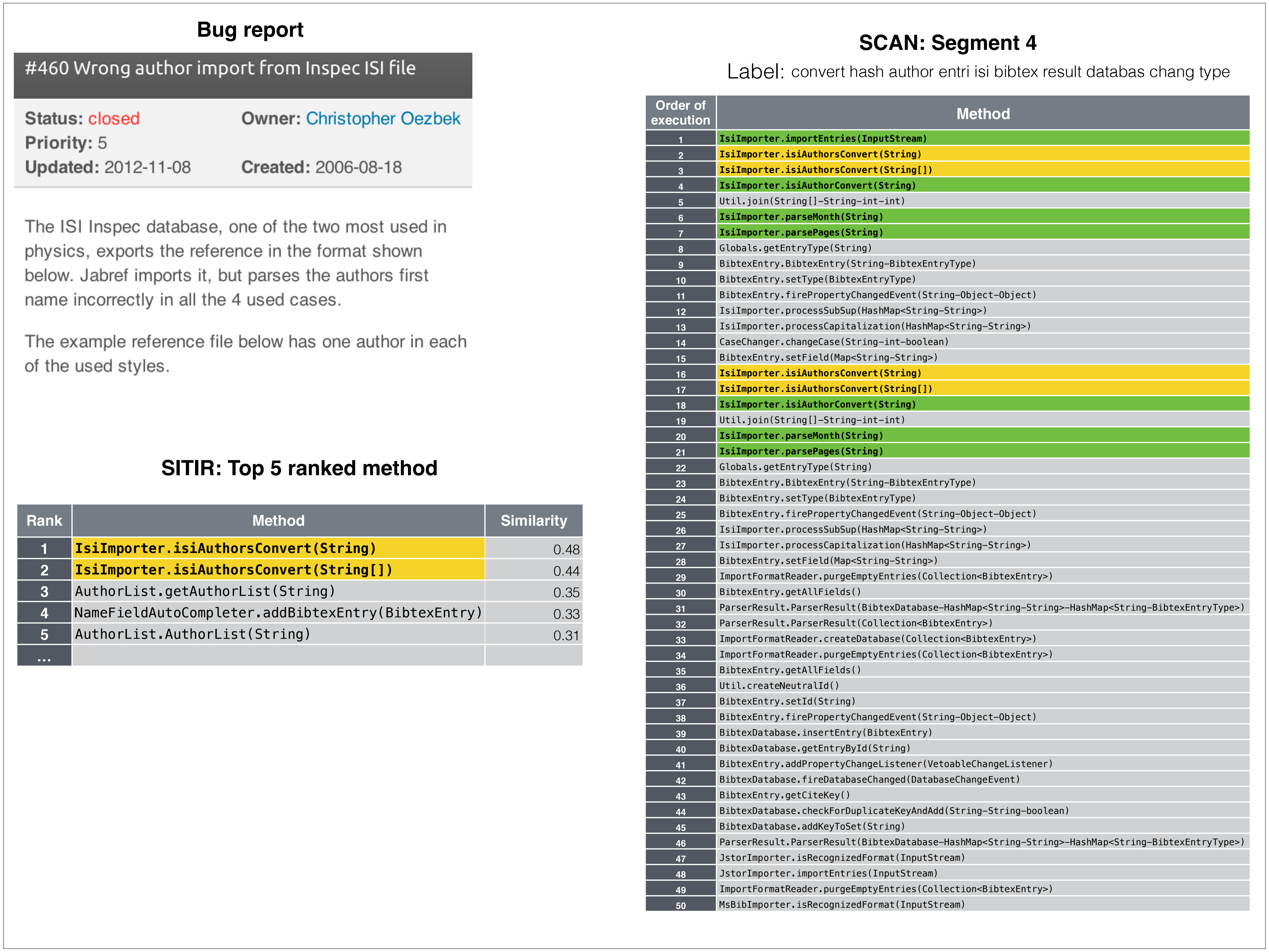
However, when no concept location technique is available to guide the search, SCAN can also be used to retrieve the segments that contain relevant methods by using the FCA lattice produced in the earlier stage and the title of the issue report as a query to guide the search. Figure 7.2 shows the partial FCA lattice corresponding to the trace of the example shown in Figure 7.1. We include the query as part of the set of objects and use the terms of the query to retrieve the segments it is related to. In other words, in order to identify the relevant segments, we look for the segments that share terms with the query. Thus, we start from the node representing the query and following the paths towards the top node the first two reachable nodes connect Query with Segment 3 (as they share the words *"isi"* and *"inspec"*) and Query with Segment 4 (as they share the words *"isi"* and *"author"*). Those are the segments containing the methods from the gold set. In general, the closer we are to the top node, the more segments we collect as we are less restrictive on the terms that those segments must contain. The closer we are to the Query node, the less segments we collect as we impose greater number of terms to be shared between them. Our conjecture is that SCAN can be used to automatically identify the segments containing the relevant methods. We also hypothesize that the automatically retrieved segments reduce the number of methods to be analysed, compared to the analysis of the entire execution trace.

7.1.2 Empirical Study Definition and Planning

In the rest of this section we describe the study that we performed to evaluate the usefulness of SCAN to support concept location. The goal of the study is to assess the usefulness of SCAN for developers with the purpose of showing its usefulness when performing concept location tasks as a complement to a concept location technique or as a standalone technique. The quality focus is the possible effort reduction achieved when using SCAN, due to the smaller number of methods a developer should scrutinize. The perspective is of researchers interested in providing support to program comprehension by labeling and relating segments in execution traces.

Study Set-Up

This section details the study set-up, specifically describing the datasets that we use, i.e., the execution traces and gold set methods of selected issue reports for two Java programs.



Bug report

#460 Wrong author import from Inspec ISI file

Status: closed **Owner:** Christopher Oezbek
Priority: 5
Updated: 2012-11-08 **Created:** 2006-08-18

The ISI Inspec database, one of the two most used in physics, exports the reference in the format shown below. Jabref imports it, but parses the authors first name incorrectly in all the 4 used cases.

The example reference file below has one author in each of the used styles.

SITIR: Top 5 ranked method

Rank	Method	Similarity
1	**IsiImporter.isiAuthorsConvert(String)**	0.48
2	**IsiImporter.isiAuthorsConvert(String[])**	0.44
3	AuthorList.getAuthorList(String)	0.35
4	NameFieldAutoCompleter.addBibtexEntry(BibtexEntry)	0.33
5	AuthorList.AuthorList(String)	0.31
...		

SCAN: Segment 4

Label: convert hash author entri isi bibtex result databas chang type

Order of execution	Method
1	**IsiImporter.importEntries(InputStream)**
2	**IsiImporter.isiAuthorsConvert(String)**
3	**IsiImporter.isiAuthorsConvert(String[])**
4	**IsiImporter.isiAuthorConvert(String)**
5	Util.join(String[]-String-int-int)
6	**IsiImporter.parseMonth(String)**
7	**IsiImporter.parsePages(String)**
8	Globals.getEntryType(String)
9	BibtexEntry.BibtexEntry(String-BibtexEntryType)
10	BibtexEntry.setType(BibtexEntryType)
11	BibtexEntry.firePropertyChangedEvent(String-Object-Object)
12	IsiImporter.processSubSup(HashMap<String-String>)
13	IsiImporter.processCapitalization(HashMap<String-String>)
14	CaseChanger.changeCase(String-int-boolean)
15	BibtexEntry.setField(Map<String-String>)
16	**IsiImporter.isiAuthorsConvert(String)**
17	**IsiImporter.isiAuthorsConvert(String[])**
18	**IsiImporter.isiAuthorConvert(String)**
19	Util.join(String[]-String-int-int)
20	**IsiImporter.parseMonth(String)**
21	**IsiImporter.parsePages(String)**
22	Globals.getEntryType(String)
23	BibtexEntry.BibtexEntry(String-BibtexEntryType)
24	BibtexEntry.setType(BibtexEntryType)
25	BibtexEntry.firePropertyChangedEvent(String-Object-Object)
26	IsiImporter.processSubSup(HashMap<String-String>)
27	IsiImporter.processCapitalization(HashMap<String-String>)
28	BibtexEntry.setField(Map<String-String>)
29	ImportFormatReader.purgeEmptyEntries(Collection<BibtexEntry>)
30	BibtexEntry.getAllFields()
31	ParserResult.ParserResult(BibtexDatabase-HashMap<String-String>-HashMap<String-BibtexEntryType>)
32	ParserResult.ParserResult(Collection<BibtexEntry>)
33	ImportFormatReader.createDatabase(Collection<BibtexEntry>)
34	ImportFormatReader.purgeEmptyEntries(Collection<BibtexEntry>)
35	BibtexEntry.getAllFields()
36	Util.createNeutralId()
37	BibtexEntry.setId(String)
38	BibtexEntry.firePropertyChangedEvent(String-Object-Object)
39	BibtexDatabase.insertEntry(BibtexEntry)
40	BibtexDatabase.getEntryById(String)
41	BibtexEntry.addPropertyChangeListener(VetoableChangeListener)
42	BibtexDatabase.fireDatabaseChanged(DatabaseChangeEvent)
43	BibtexEntry.getCiteKey()
44	BibtexDatabase.checkForDuplicateKeyAndAdd(String-String-boolean)
45	BibtexDatabase.addKeyToSet(String)
46	ParserResult.ParserResult(BibtexDatabase-HashMap<String-String>-HashMap<String-BibtexEntryType>)
47	JstorImporter.isRecognizedFormat(InputStream)
48	JstorImporter.importEntries(InputStream)
49	ImportFormatReader.purgeEmptyEntries(Collection<BibtexEntry>)
50	MsBibImporter.isRecognizedFormat(InputStream)

Figure 7.1 Bug#460 in JabRef: wrong author import from Inspec ISI file.

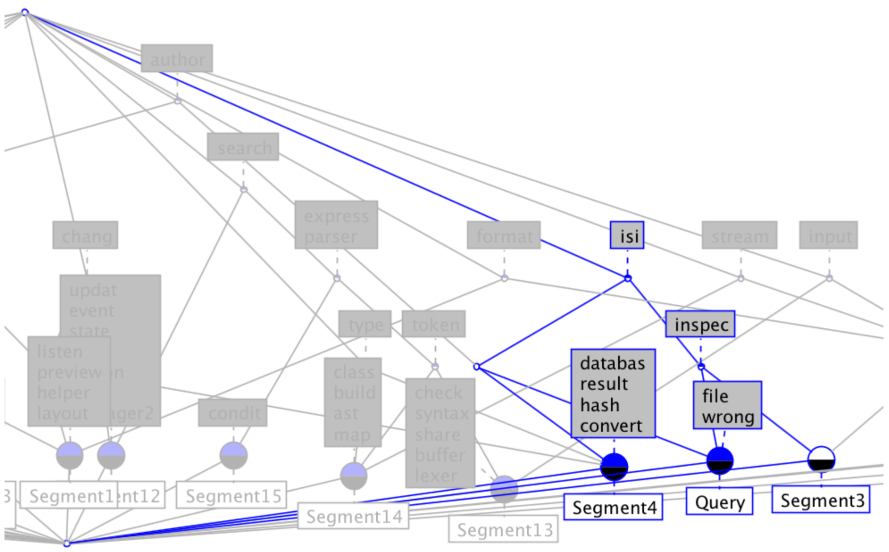


Figure 7.2 Bug#460 in JabRef: resulting FCA lattice.

The objects of our evaluation are execution traces collected from two Java programs belonging to different domains. JabRef[1] is an open source bibliography reference manager. It uses the BibTeX file format and provides a user-interface to manage BibTeX files. muCommander[2] is a lightweight, cross-platform file manager with a dual-pane interface. It allows users to perform file operations on a variety of local and networked file systems, including FTP, Windows shares, and so on.

Table 7.1 summarises characteristics of the programs, i.e., the interval considered (i.e., from release x to release y) the numbers of bugs occurred in such a time interval, the number of traces that include two or more gold set methods, and the total number of gold set methods. The execution traces were generated for the latter release—i.e., 2.6 for JabRef and 0.8.5 for muCommander.

The choice of JabRef and muCommander for this study is related to the availability of execution traces for the issue reports considered in the study as well as the associated gold set methods. Given the available dataset from Dit *et al.* (2013a), to evaluate the usefulness of SCAN we analysed a total of 65 execution traces from the two programs—17 from JabRef and 48 from muCommander.

Table 7.2 reports descriptive statistics about the numbers of method calls in the execution traces as well as in the segments that SCAN identifies.

1. http://jabref.sourceforge.net/
2. http://www.mucommander.com/

Table 7.1 Programs characteristics.

Program	Release Range	Issues	Traces with two or more gold set methods	Gold set methods
JabRef	2.0–2.6	39	17	280
muCommander	0.8.0–0.8.5	92	48	717
Overall		131	65	997

Table 7.2 Traces and segments characteristics.

	Number of method calls					
	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Traces	3K	57K	95K	95K	126K	264K
Segments	2	2	2	104	3	5K

7.2 Experimental Design and Analysis

The study aims at answering the following research questions:

– **RQ1:** *Does SCAN has a potential to support concept location?* During maintenance, developers generally are interested to understand some segments of the trace that implement concepts of interest rather than to analyse in-depth the entire execution trace. We formulate this research question to verify that the concepts of interest are grouped in few segments and thus ensure that our trace segmentation approach facilitates maintenance tasks. This research question aims at evaluating SCAN's ability to group gold set methods into a low number of segments thus reducing the number of methods to be analysed by developers when limiting the analysis to the set of segments containing the gold set methods rather than the entire execution trace. The conjecture—as explained in Section 7.1.1—is that such segments would contain most of the methods related to the concept, hence the developer could easily determine the set of methods impacted when performing the change—e.g., the bug fixing—by looking at the segments containing the seeds only.

– **RQ2:** *To what extent does SCAN support concept location tasks if used as a standalone technique?* We formulate this research question to verify the efficiency of our approach to guide developers towards segments that implement the concepts to maintain and reduce the number of methods to investigate. This research question investigates whether it is possible to automatically retrieve segments containing relevant methods and evaluates the number of methods to be analysed compared to the number of methods in the entire execution trace. We calculate the recall with respect to the segments containing the gold set methods as well as the recall with respect to the gold set methods.

To address **RQ1**, we first investigate whether methods from the gold set are grouped within few segments. To this end, we provide the number of segments containing the gold set methods and the total number of segments of the traces. The lower the number of segments containing the gold set methods the more grouped they are and thus the more potentially useful SCAN is. For example, the execution trace for the example bug report in Figure 7.1 is segmented by SCAN into 26 segments, and the gold set methods are concentrated in 2 of those segments—Segment 3 and Segment 4. Clearly, the sizes of the segments also impact the extent to which SCAN would help to reduce the effort: analysing many small segments would not be effort-prone as analysing few bigger segments (representing the execution of a whole concept), although the absence of cohesive segments would provide no guidance to the developer for knowing when to stop analysing methods. To mitigate this problem, we consider the size of the segments in terms of total number of method calls and in terms of unique methods.

We estimate the number of methods in the segments containing the gold set methods and divide it by the number of methods in the entire execution trace. This ratio represents the number of methods to be analysed if one is able to retrieve the segments containing the gold set methods. The lower such ratio, the better. In the above example, the total number of method calls is 52—2 in Segment 3 and 50 in Segment 4. The ratio is thus 0.0038 $(= 52/13,616)$.
We also estimate the ratio of the unique number of methods in the segments containing the gold set methods over the total number of methods in the execution trace. The lower the ratio the greater the gain in terms of number of unique methods that developers need to understand. This ratio indicates the upper bound limit for the reduced number of methods that an automatic technique retrieving the segments must return to developers. It is an upper bound as reducing more, i.e., not analysing some of those segments, would result in incomplete change implementation. Any technique that identifies additional segments would be decreasing the reduction of methods to be analysed. Reaching the upper bound assumes that we have a perfect way to identify those and only those segments. For the above example, the unique number of methods called in Segments 3 and 4 is 34 and the unique number methods in the entire execution trace is 479 resulting in a ratio of 0.07 $(= 34/479)$. This ratio is a proxy for the effort that developers would need to spent if they concentrate on the segments containing the gold set methods rather than the entire trace.

To address **RQ2**, we use the labels of segments, the relations among the segments produced by SCAN, and the title of the bug report as search query to retrieve the segments having one or more terms in common with this query. For each trace, SCAN builds the FCA lattice using the segments of the trace while adding the title of the bug report (query) to the set of objects of the lattice. SCAN considers as objects segments and the query; attributes are the terms in the segments/query. By analysing the resulting FCA lattice, SCAN identifies the segments in relation with the query: First, starting from the node representing the query and following the paths towards the top node SCAN collects all encountered nodes. Next, for all the collected nodes SCAN identifies the segments they are connected to. The set of collected segments contains all segments with which the query has same concept, sub-concept, or super-concept relation.

We evaluate the ability of SCAN to retrieve relevant segments using the gold set methods and calculating two types of recall. We calculate the recall with respect to the segments containing the gold set methods as well as the recall with respect to the gold set methods only. To calculate the recall for segments, we divide the number retrieved segments containing gold set methods by the total number of segments containing the gold set methods (see Equation 7.1). For the example shown in Figure 7.2, the recall for segments is 1 (i.e., 100%) as both Segments 3 and 4 are retrieved.
To calculate the recall for methods, we divide the number retrieved gold set methods by the total number of gold set methods (see Equation 7.2). The recall for methods in the above example is also 1 (= 7/7) as SCAN retrieves all the gold set methods.

$$Recall_{\text{Segments}} = \frac{\text{retrieved segments containing gold set methods}}{\text{total number of segments containing the gold set methods}} \quad (7.1)$$

$$Recall_{\text{Methods}} = \frac{\text{retrieved gold set methods}}{\text{total number of gold set methods}} \quad (7.2)$$

Here we also provide a proxy for the effort that developers would need to spent if they concentrate on the segments retrieved by SCAN rather than the entire trace. This estimate of effort is expressed as the ratio of the number of methods to be analysed if analysing the methods contained in the retrieved by SCAN segments and the entire trace. For the above example, SCAN retrieves 5 segments with a total of 133 unique methods being called. The trace consists of 479 unique methods being called thus resulting in a ratio of 0.27 (= 133/479), i.e., 27%.

Finally, we also analyse how the recall varies when the number of terms in labels varies from 10 to 100. Previously, we limited the number of words in a label to ten as we were seeking to provide a concise summary of a segment—this constraint for conciseness was imposed by the purpose of the label, i.e., help developers to quickly grasp the concepts of a segment. For the purpose of automatic concept location, we increase the number of words as the labels will be used to automatically retrieve the relevant segments and thus a higher amount of terms is not an issue.

7.3 Experiment Results and Discussions

This section reports the results of our experimental study to address the research questions formulated in Section 7.2.

RQ1: Does SCAN has a potential to support concept location?

To answer this research question, we investigate whether multiple methods from the gold set are grouped within the same segments. Thus, from the 131 traces of JabRef and mu-Commander, we are interested in those containing at least two gold set methods, i.e., 65 of those traces, see Tables 7.1. Table 7.3 shows statistics of the numbers of gold set methods for those traces. We observe that the gold sets consist of around six methods on average, as shown by the column Mean of Table 7.3.

Table 7.3 Number of gold set methods.

Program	Min	1st Qu.	Median	Mean	3rd Qu.	Max
JabRef	2	3	4	5.5	7	16
muCommander	2	2	3	6	7	35
Overall	2	2	3	5.9	7	35

Table 7.4 Distribution of the gold set methods across the segments.

	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Number of segments containing the gold set methods	1	1	2	2.2	3	5
Overall number of segments in a trace	14	30	38	35.9	41	68
Percentage of the size of the segments containing gold set methods (over the size of the trace)	0%	1.56%	2.16%	2.48%	3.37%	6.47%
Unique number of methods appearing in segments required to understand (compared to the unique number of methods required to understand the entire trace)	0%	29.63%	44.77%	47.09%	61.92%	81.83%

Table 7.4 provides statistics on the distribution of the gold set methods across the different segments of the traces. We observe that the gold set methods are usually concentrated in only two segments, as shown by the column Mean of Table 7.4 (first row) while on average the total number of segments for a trace is close to 36 (second row). Therefore, we conclude

that, indeed, methods of the gold set are grouped into segments and thus SCAN has the potential to guide developers to other methods useful to their concept location task.

As explained in Section 7.2, we also provide a rough estimate of the effort developers would save if they focus their understanding activity on segments containing the gold set methods rather than understanding the entire trace. The effort to understand a segment (respectively, a complete trace) is estimated by the number of unique method calls in that segment (respectively, trace). Table 7.4 presents statistics regarding the percentage of the sizes of the segments containing the gold set methods with respect to the overall sizes of the traces. The percentage is also provided in terms of unique methods. We conclude that the size of the segments containing the gold set methods is smaller than 3% of the size of the entire traces. If focusing the understanding on relevant segments only (i.e., those containing methods from the gold set) rather than on the entire trace, we can reduce the number of methods to analyse by about 47%.

Consider again the example shown in Figure 7.1. Rather than analysing the entire execution trace, the developers may focus on Segment 4, which provides the context in which method `isiAuthorsConvert(String)` is called. By looking at the method calls in Segment 4, they can understand that the author conversion is performed in the context of importing a bibliographical entry. They can also realise that, in general, importing an ISI entry also requires parsing the month and pages, which were not performed adequately. Hence, Segment 4 also contains other gold set methods, appearing at positions 1, 3, 4, 6, 7, 17, 18, 20, and 21, which were modified to fix problems in the import of an ISI entry, the parsing of month and pages in particular, while fixing the author conversion.

We thus answer **RQ1**: Does SCAN has a potential to support concept location? SCAN has the potential to be useful during concept location because it groups gold set methods in only two segments in general. Assuming that the segments containing the gold set methods can be retrieved, understanding those relevant segments saves about 53% of the methods that developers would need to understand compared to the entire execution traces.

RQ2: To what extent does SCAN support concept location tasks if used as a standalone technique?

In **RQ1**, we assumed that the results from a concept location technique are available and that developers are guided by those results to select the segments to understand—segments relevant to a given concept. However, such results may be unavailable and we are interested to know whether we can retrieve the relevant segments even without such results.

Table 7.5 shows the results of $Recall_{\text{Segments}}$, i.e., the recall with respect to the segments containing the gold set methods, for different sizes of the labels (we vary the size from 10 to 100, step by 10). We observe that, for example, when the maximum number of terms in a label is 100, we can retrieve 75% of the segments containing the gold set methods; for 68% of the traces, we retrieve all segments—i.e., 100% recall. The minimum retrieved segments is 0% because gold set methods are sometimes filtered in the preprocessing of the segmentation.

Table 7.5 $Recall_{\text{Segments}}$: retrieving segments containing gold set methods.

Label Size	Min	1st Qu.	Median	Mean	3rd Qu.	Max
10	0%	0%	0%	35.48%	100%	100%
20	0%	0%	33.33%	45.22%	100%	100%
30	0%	0%	50%	50.86%	100%	100%
40	0%	0%	50%	56.55%	100%	100%
50	0%	28.57%	100%	64.59%	100%	100%
60	0%	33.33%	100%	67.41%	100%	100%
70	0%	50%	100%	73.18%	100%	100%
80	0%	50%	100%	74.31%	100%	100%
90	0%	50%	100%	74.31%	100%	100%
100	0%	50%	100%	75.08%	100%	100%

Table 7.6 $Recall_{\text{Methods}}$: retrieving gold set methods.

Label Size	Min	1st Qu.	Median	Mean	3rd Qu.	Max
10	0%	0%	0%	36.05%	100%	100%
20	0%	0%	33.33%	43.26%	100%	100%
30	0%	0%	33.33%	46.72%	100%	100%
40	0%	0%	50%	53.27%	100%	100%
50	0%	20%	66.67%	58.88%	100%	100%
60	0%	33.33%	80%	62.36%	100%	100%
70	0%	33.33%	100%	68.18%	100%	100%
80	0%	33.33%	100%	69.24%	100%	100%
90	0%	33.33%	100%	69.24%	100%	100%
100	0%	33.33%	100%	70.01%	100%	100%

Table 7.6 shows $Recall_{\text{Methods}}$, i.e., the recall with respect to the gold set methods (rather than the segments that contain them). Thus, considering again the case where the number of terms in a label is limited to 100, we observe that, on average, we retrieve 70% of the gold set methods. Table 7.7 shows that analysing the retrieved segments represents on average 57% of the methods that one would have to understand to analyse the entire trace.

Finally, we observe that increasing the size of the labels leads to more gold set methods and more segments that contain them to be retried. However, it also increases the number of methods to be analysed.

Table 7.7 Number of methods needed to understand the retrieved segments compared to the number of methods needed to understand the entire trace.

Label Size	Min	1st Qu.	Median	Mean	3rd Qu.	Max
10	0%	0%	3.88%	27.05%	50.24%	81.83%
20	0%	0.32%	30.11%	33.39%	67.07%	81.83%
30	0%	1.65%	38.08%	37.2%	67.07%	81.83%
40	0%	22.57%	43.73%	42.66%	68.61%	81.83%
50	0%	25.5%	54.93%	47.78%	71.82%	81.83%
60	0%	31.99%	57.32%	50.6%	77.3%	82.85%
70	0%	35.74%	63.16%	53.75%	77.3%	82.85%
80	0%	40.79%	63.16%	54.4%	77.3%	82.85%
90	0%	44.78%	65.08%	55.76%	77.3%	82.85%
100	0%	49.31%	65.08%	56.56%	77.08%	82.85%

From Tables 7.5, 7.6 and 7.7, we conclude that a value ranging between 70 and 100 terms in the labels seems to be optimal as it retrieves close to 74% of the segments containing the gold set methods, which corresponds close to 69% of the gold set methods, while saving near 45% of the methods to analyse.

We thus answer **RQ2**: To what extent does SCAN support concept location tasks if used as a standalone technique? When no technique is used to guide developers, SCAN can retrieve relevant segments. For the analysed traces, the recall with respect to the gold set methods is close to 69% while saving near 45% of the methods to analyse compared to the entire execution traces.

7.3.1 Threats to Validity

This section discusses the threats to the validity of our evaluation.

In **RQ1** and **RQ2**, we estimate a proxy of the effort a developer has to spend when performing concept location in terms of the number of methods to be analysed. We are aware that this is a roughly estimate, because the actual effort could involve many factors, such as the length and complexity of those methods, the overall code complexity, quality of the lexicon, experience of the developer, etc. However, in a context in which the impact set will be determined by performing a basic understanding of each candidate method—e.g., by looking at its signature and comments if any—such an approximation may result reasonable.

7.4 Summary

In this chapter, we conducted a study aimed at investigating the usefulness of SCAN to support concept location tasks. We investigated how SCAN could help in a concept location task, when used in combination with a state-of-the-art concept location approach (**RQ1**), or when used as a standalone approach (**RQ2**). Our conjecture was that methods relevant for a concept would be grouped by SCAN in one or two segments and therefore limiting the analysis to the methods in these segments would reduce the effort compared to analysing the methods in the entire execution traces (**RQ1**). We also investigated whether SCAN is able to automatically retrieve these relevant segments and whether analysing the retrieved segments reduces the number of methods to be analysed compared to the entire execution trace (**RQ2**). Results show that in general relevant methods are grouped in two segments and analysing only those segments reduces about 53% of the methods that developers would need to understand compared to the entire execution traces (**RQ1**). Results also show that, for the analysed traces, SCAN can retrieve close to 69% of the relevant methods while reducing the number of methods to analyse by 43% compared to analysing the entire traces (**RQ2**). We showed that SCAN provides useful information to developers performing concept location tasks: it provides relevant segments, labels for these segments, and relations among segments.

During maintenance, developers generally are interested to understand some segments of the trace that implement concepts of interest. We proposed an approach that groups the concepts to maintain in few segments (i.e., one or two). We showed also the usefulness of our approach to facilitate maintenance tasks by guiding developers towards segments that implement the concepts to maintain and reduce the number of methods to investigate.

CHAPTER 8

Conclusion

Program comprehension activities are crucial and preliminary to any maintenance or evolution tasks. Execution traces help developers to understand programs and relate methods (and methods calls) with concepts. The problem is that execution traces are often overly large and they cannot be used directly by developers for program comprehension activities, in general, and concept location, in particular.

Concept location approaches typically use static and–or dynamic information extracted from the source code of a program or from some execution traces to relate method calls to concepts. Both static and dynamic techniques have some limitations. Recent works focused on hybrid approaches integrating static and dynamic information to improve the performance in time as well as precision and recall of the concept location process (Antoniol et Guéhéneuc, 2005; Poshyvanyk *et al.*, 2007; Rohatgi *et al.*, 2008; Asadi *et al.*, 2010b). Several approaches use different techniques to locate concepts in source code and–or execution traces, e.g., Antoniol *et al.* (2006) proposed an epidemiological metaphor to analyse source code, Poshyvanyk *et al.* (2007) used latent-semantic indexing (LSI) to locate concept in source code and execution traces, Rohatgi *et al.* (2008) used graph dependency ranking on static and dynamic data, Pirzadeh and Hamou-Lhadj (2011) studied psychology laws describing how the human brain groups similar methods in execution traces. None of the proposed approaches provides a label (i.e., a set of terms) describing concepts implemented by segments or an identification of the relations between the segments.

Developers generally are interested to understand some segments of the trace that implement concepts of interest rather than to analyse in depth the entire execution trace. Extracting the set of concepts from execution traces facilitates maintenance tasks by guiding developers towards segments that implement the concepts to maintain. Our conjecture was that a high-level description of the set of concepts implemented in an execution trace allows developers to understand the execution trace and identify the segments implementing the concepts to maintain.

To reduce the complexity of analysing execution traces, we automatically split them into meaningful segments, each representing a concept. Then, we proposed SCAN, an approach to assign labels to the identified segments. The labels allow developers to have an idea of the concepts implemented by the segments and guide them towards segments implementing the concepts to maintain. We identified the relations among trace segments to provide a high-level description of the concepts implemented in an execution trace. We showed that SCAN is efficient for selecting the most important methods of a segment, labeling segments, and identifying relations among segments. We showed also that the information provided by SCAN is useful to developers performing concept location tasks.

8.1 Research Contributions

The main contributions of this dissertation are as follows:

- An execution traces segmentation approach that splits execution traces into segments using dynamic programming (DP) algorithm. The proposed trace segmentation approach helps developers by reducing the number of methods to investigate using execution traces during maintenance tasks.
  The results of this contribution were published in the proceedings of the 3rd International Symposium on Search-based Software Engineering (SSBSE'11) (Medini *et al.*, 2011).
- Labeling execution traces segments using Vector Space Model (VSM). The assigned labels allow developers to have an idea of the concepts implemented by the segments and guide them towards segments implementing the concepts to maintain. We presented an evaluation of the performances of SCAN. We investigated SCAN capability to select the most important methods of a segment and label segments. The results confirmed the ability of SCAN to select the most representative methods of a segment. We also showed that SCAN can automatically label segments when compared to the manual labels produced by participants.
- Identifiying relations among segments using Formal Concept Analysis (FCA). We provided a high-level description of the concepts implemented in an execution trace by identifying the relations among segments. We investigated SCAN capability to identify relations among segments. Results showed that participants agreed with the identified relations among segments identified by SCAN.
  The results of the second and third contributions were published and received the best paper award in the proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12) (Medini *et al.*, 2012).
- A study of the usefulness of the automatic trace segmentation and labeling in the context of concept location. We assessed whether SCAN supports concept location tasks if used as a standalone technique in practice. The results showed that SCAN can be useful during concept location because it groups gold set methods in only two segments in general. Results showed also that SCAN supports concept location tasks

if used as a standalone technique.

A part of the results of the three last contributions were published in the Journal of Software: Evolution and Process 2014.

In the following, we explain these contributions in more detail.

Trace Segmentation

Developers generally are interested to understand some parts of the trace that implement the concept of interest rather than to analyse in-depth the entire execution trace. We proposed a new technique based on dynamic programming (DP) to identify segments in execution traces by finding cohesive and decoupled fragments of the execution trace. The proposed trace segmentation simplified the comprehension of large execution traces and allows developers to focus on segments to maintain and thus facilitate their maintenance tasks. Differently to the previous approaches based on genetic algorithms (GA), our dynamic programming (DP) approach can compute the exact solution to the trace segmentation problem. We empirically compared the DP and GA approaches, using execution traces from ArgoUML and JHotDraw, which were previously used to validate the GA approach (Asadi *et al.*, 2010b). Results indicated that the DP approach can achieve results similar to the GA approach in terms of precision and recall when its segmentation is compared with a manually-built oracle. Results also showed that the DP approach has significantly better results in terms of the optimum segmentation score vs. fitness function. More importantly, results showed that DP reduces dramatically the time required to segment traces: where the GA approach would take several minutes, even hours; the DP approach just takes a few seconds.

Labeling Trace Segments

We proposed SCAN (Segment Concept AssigNer) to assign labels to segments. The assigned labels provide relevant information on the concepts implemented by segments to help developers understand each segment.

We performed a manual validation on several traces of both JHotDraw and ArgoUML to evaluate the accuracy and effectiveness of assigning meaningful sets of words representative of the concepts implemented in segments. Results showed that SCAN was successful in assigning labels very similar to manually-defined labels and that these labels actually correspond to the concepts encountered in the segment based on documentation and method execution.

A segment manually labeled by one person may bias our evaluation of the labels generated by SCAN. To cope with this limitation, we performed an experiment to verify SCAN capability to select the most important methods of a segment and to label segments by 31 participants for six Java programs (ArgoUML, JHotDraw, Mars, Maze, Neuroph, and Pooka).

To evaluate the accuracy of SCAN to label segments, we analysed the labels of the trace segments generated by SCAN with respect to the labels produced by the participants. Results of the empirical evaluation confirmed the ability of SCAN to select the most representative methods of a segment, thus reducing on average up to 92% the information that participants must process while guaranteeing that close to 50% of the knowledge is preserved. We also showed that SCAN can automatically label segments with 69% precision and 63% recall when compared to the manual labels produced by the participants. Results also showed that participants agreed at 63% with the identified relations among segments identified by SCAN.

Relating Trace Segments

We proposed to identify different relations between segments to provide a high-level description of the concepts implemented in the entire execution trace. Our approach allowed us to detect relations between segments to automatically identify repeated computational phases and to abstract them into macro-phases. The approach related (1) segments activating the same concepts, (2) segments activating parts of concepts activated by other segments, and (3) segments activating the same set of concepts. We performed an experiment to evaluate SCAN capability to identify relations among segments against 31 participants on six Java programs (ArgoUML, JHotDraw, Mars, Maze, Neuroph, and Pooka). Results showed that participants agreed at 63% with the identified relations among segments identified by SCAN.

SCAN Usefulness Evaluation

We evaluated the usefulness of SCAN in the context of performing a concept location task for two Java programs: JabRef and muCommander. Results showed that SCAN has the potential to be useful during concept location because it groups gold set methods in only two segments in general. To understand those relevant segments reduces 53% of the methods that developers would need to understand compared to the entire execution traces. Results showed also that SCAN support concept location tasks if used as a standalone technique. The FCA lattice and the labeled segments produced by SCAN allows to retrieve 69% of the relevant methods while still saving 43% of the effort needed to analyse the entire trace. We concluded that SCAN provides useful information to developers performing concept location tasks: it provides relevant segments, labels for these segments, and relations among segments.

8.2 Limitations

Despite the above promising contributions, our work has the following limitations that should be addressed in the future:

Limitation of Trace Segmentation

Parallel and distributed systems use hundreds of thousands of processors. The comprehension of the behavior of a massively parallel code is a challenge. However, our approach cannot split parallel execution traces.

SCAN accepts as input one or more execution traces obtained by exercising some scenarios in a program. In this thesis, an execution trace is represented as a sequence of methods called during the execution of a scenario. However, SCAN is not available for operating system execution traces, such as Linux traces, in which a trace is a sequence of system calls.

Although our trace segmentation approach outperformed the GA approach in terms of computation times and fitness function values, we further studied the scalability of the DP trace segmentation approach on large systems and observed that the DP trace segmentation approach took about one day on traces of about 38,000 method invocations. We concluded that the computation times increase with trace sizes.

We proposed an approach to split traces based on conceptual cohesion and coupling. The trace segmentation proposed works offline, i.e., the traces are generated and saved. Our trace segmentation require the entire execution trace to compute the cohesion and coupling and thus split the trace. Thus, our trace segmentations could not split traces online.

To calculate the similarity of methods, we considered only the identifiers for splitting the execution traces. The quality of identifiers plays also a role and affects the quality of the obtained results. Many other factors could affects the similarity of methods and they are not considered such as method calls or methods belonging to the same class affect.

Table 4.4 in Chapter 4 shows a lower precision of concepts when we used execution traces with more than one concept. For example, the precision of "Draw Rectangle" concept of JHotDraw of Scenario (2) and Scenario (3) are lower than the precision of "Draw Rectangle" of Scenario (1) composed of one concept only. For ArgoUML also, the precision of "Create Note" concept of Scenario (2) is lower than the precision of the same concept of Scenario 1 composed of one concept only. The DP approach exhibits a lower precision for a given concept where an execution trace is composed of more than one concept than the precision for the same concept where an execution trace is composed of only one concept.

Limitation of Labeling and Relating Segments

Because the trace segmentation proposed in this thesis works offline, then labeling would also be performed offline. Yet, developers might need to perform labeling online.

Our approach defined a high-level description of the concepts implemented in the entire execution trace by automatically identifying repeated computational phases and abstracting them into macro-phases. We did not provide a visualisation of the extracted phases and macro-phases and of the identified relations between phases as shown in Figure 6.4 in Chapter 6. Developers need such visualisation to quickly understand the set of concepts extracted from execution traces.

SCAN uses method signature to extract relevant terms of a segment but sometimes we notice a mismatch between the functionnalies of methods and their names. The presence of short acronyms in method names could also generate labels containing short acronyms that could not be understandable to developers. SCAN depends on the quality of method names of programs. Low quality of method names will negatively impact the results by a low quality of labels.

8.3 Future works

We describe our plan to extend the work presented in this dissertation by following:

Execution Trace Segmentation

As mentioned earlier, in this thesis, we only considered the similarity of methods to calculate the exact splitting of the execution trace. The result might be improved if more informations are used. For example, right now, the fitness function has two factors (i.e.,conceptual cohesion and coupling). One direction of future work is to investigate how other factors can affect execution trace segmentation; for example, the nesting level of methods or methods belonging to a same class. These factors would be to our conceptual cohesion and coupling factors.

During the pruning of the execution trace, we removed the most frequent methods but we did not garantee that we removed all low-level utilities. In the future, we must study the impact of removing utility methods from execution traces before performing trace segmentation.

We believe that other problems, such as segmenting composed identifiers into component terms, could be modelled using dynamic programming and, thus, that we should be careful when analysing a problem: a different problem formulation may lead to surprisingly good performances.

We observed that our execution trace segmentation approach based on DP improves the previous GA results, despite the above mentioned limitations. As in previous work, we built a term-document matrix using a LSI subspace size equal to 50 to calculate the similarity between two methods. We plan to evaluate the accuracy of our execution trace segmentation with respect to different LSI subspace sizes. We plan to apply splitting algorithm on execution traces using different values of LSI subspace size and different programs.

Labeling and Relating Trace Segments

After defining automatically execution trace phases, we plan to present a tool to draw a flow diagram of phases with labels as shown in Figure 6.4. This diagram provides a high level presentation of the context of the execution trace using useful information about phases.

We plan also to perform user studies in which developers use SCAN generated segments, labels, phases and macro-phases of execution traces during their maintenance activities.

For example, we could group participants into two groups, G1 and G2. We would give the segments (i.e., method calls) and the labels describing the concepts to G1 while we would give only the segments to G2. Then, we could compare the performance of both groups to see if the generated labels help developers to be more productive when solving maintenance tasks.

Another direction of future work is to extend our approach (1) to identify execution segments from parallel execution traces.

Because the trace segmentation proposed in this thesis works offline, then labeling would be also performed offline. Developers might need to perform labeling online. Therefore, one direction for future work would be to adapt our approach to online labeling of traces while they are being generated.

Because the number of execution trace segments increases with the number of considered objects, the scalability of our approach is threaten. It would be better to investigate more the scalability of using formal concept analysis, such as to divide the relations identification problem into sub-problems and identify the result by combining the solutions of the sub-problems.

Manual labels contains the same keywords as SCAN generated labels but the manual ones are more informative. Table 5.5 in Chapter 5 presented SCAN generated and manual labels for the ArgoUML trace “New Class”. For example, in this table the SCAN generated label is “vertex state meta mdr type impl” and the manual label is “Display the vertex state” which is much more informative and helpful to developers. Using Natural Language Processing (NLP) techniques, we plan to generate natural language sentences using terms of the generated labels. Natural language sentences makes the generated labels more informative and the produced segments better suitable for program comprehension activities. Also, we will investigate the applicability of extracting labels as a re-documentation tool.

SCAN uses method signature to extract terms relevant to a segment but, sometimes, we observed a mismatch between the functions of methods and their names. Because SCAN depends on the quality of method names, low quality of method names will negatively impact the results, yielding low quality labels. We plan to add other source of information, such as emails, bug reports, and documentation of programs, to label segments.

REFERENCES

AGRAWAL, H., DEMILLO, R. A. et SPAFFORD, E. H. (1993). Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23, 589–616.

ANQUETIL, N. et LETHBRIDGE, T. (1998). Extracting concepts from file names: a new file clustering criterion. *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 84–93.

ANTONIOL, G. et GUÉHÉNEUC, Y. G. (2005). Feature identification: a novel approach and a case study. *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 357–366.

ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2006). Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32, 627–641.

ASADI, F., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2010a). Concept locations with genetic algorithms: A comparison of four distributed architectures. *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*. IEEE Computer Society Press, 153–162.

ASADI, F., PENTA, M. D., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2010b). A heuristic-based approach to identify concepts in execution traces. *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 31–40.

BAEZA-YATES, R. et RIBEIRO-NETO, B. (1999). *Modern Information Retrieval*. Addison-Wesley.

BAKER, R. D. (1995). *Modern permutation test software*. Marcel Decker.

BASSETT, B. et KRAFT, N. A. (2013). Structural information based term weighting in text retrieval for feature location. *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press, 133–141.

BELLMAN, R. E. et DREYFUS, S. E. (1962). *Applied Dynamic Programming*, vol. 1. Princeton University Press.

BIERMANN, A. (1972). On the inference of turing machines from sample computations. *Artificial Intelligence*, 3, 181–198.

BIGGERSTAFF, T., MITBANDER, B. et WEBSTER, D. (1993). The concept assignment problem in program understanding. *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 482–498.

CHEN, K. et RAJLICH, V. (2000). Case study of feature location using dependence graph. *Proceedings of the International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society Press, 241–249.

CORMEN, T. H., LEISERSON, C. E. et RIVEST, R. L. (1990). *Introductions to Algorithms*. MIT Press.

CORNELISSEN, B., MOONEN, L. et ZAIDMAN, A. (2008). An assessment methodology for trace reduction techniques. *Proceedings of IEEE International Conference on Software Maintenance*. IEEECOMPSP, 107–116.

CORNELISSEN, B., ZAIDMAN, A., VAN DEURSEN, A., MOONEN, L. et KOSCHKE, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35, 684–702.

DAPENG, L., ANDRIAN, M., DENYS, P. et VACLAV, R. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 234–243.

DE LUCIA, A., DI PENTA, M., OLIVETO, R., PANICHELLA, A. et PANICHELLA, S. (2012). Using IR methods for labeling source code artifacts: Is it worthwhile? *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press, 193–202.

DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K. et HARSHMAN, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41, 391–407.

DEHAGHANI, S. M. H. et HAJRAHIMI, N. (2013). Which factors affect software projects maintenance cost more? *Acta Informatica Medica*, 21, 63–66.

DIT, B., HOLTZHAUER, A., POSHYVANYK, D. et KAGDI, H. (2013a). A dataset from change history to support evaluation of software maintenance tasks. *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 131–134.

DIT, B., REVELLE, M., GETHERS, M. et POSHYVANYK, D. (2013b). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25, 53–95.

EISENBARTH, T., KOSCHKE, R. et SIMON, D. (2001a). Feature-driven program understanding using concept analysis of execution traces. *Proceedings of the International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society Press, 300–309.

EISENBARTH, T., KOSCHKE, R. et SIMON, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering*, 29, 210–224.

EISENBARTH, T., R., K. et D., S. (2001b). Derivation of feature component maps by means of concept analysis. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society Press, 176–179.

EISENBERG, A. D. (2005). Dynamic feature traces: Finding features in unfamiliar code. *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 337–346.

FRAKES, W. B. et BAEZA-YATES, R. (1992). *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ.

GRISSOM, R. J. et KIM, J. J. (2005). *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, seconde édition.

HAIDUC, S., APONTE, J. et MARCUS, A. (2010a). Supporting program comprehension with source code summarization. *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 223–226.

HAIDUC, S., APONTE, J., MORENO, L. et MARCUS, A. (2010b). On the use of automated text summarization techniques for summarizing source code. *Proceedings of Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 35–44.

HAMOU-LHADJ, A., BRAUN, E., AMYOT, D. et LETHBRIDGE, T. (2005). Recovering behavioral design models from execution traces. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. ACM Press, 112–121.

HAMOU-LHADJ, A. et LETHBRIDGE, T. (2006). Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press, 181–190.

HAMOU-LHADJ, A. et LETHBRIDGE, T. C. (2002). Compression techniques to simplify the analysis of large execution traces. *Proceedings of the International Workshop on Program Comprehension (IWPC)*. 159–168.

HINTZE, J. L. et NELSON, R. D. (1998). Violin plots: A box plot-density trace synergism. *The American Statistician*, 52, 181–184.

JACCARD, P. (1901). Etude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 547–549.

JONES, K. S. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28, 11–21.

KOZACZYNSKI, V., NING, J. Q. et ENGBERTS, A. (1992). Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18, 1065–1075.

KUHN, A. et GREEVY, O. (2006). Exploiting the analogy between traces and signal processing. *Proceedings of IEEE International Conference on Software Maintenance.* IEEECOMPSP, 320–329.

LE, T.-D. B., WANG, S. et LO, D. (2013). Multi-abstraction concern localization. *Proceedings of the International Conference on Software Maintenance (ICSM).* IEEE Computer Society Press, 364–367.

LIU, D., MARCUS, A., POSHYVANYK, D. et RAJLICH, V. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. *Proceedings of the International Conference on Automated Software Engineering.* ACM, 234–243.

MARCUS, A., POSHYVANYK, D. et FERENC, R. (2008). Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34, 287–300.

MARCUS, A., SERGEYEV, A., RAJLICH, V. et MALETIC, J. I. (2004). An information retrieval approach to concept location in source code. *Proceedings of the Working Conference on Reverse Engineering (WCRE).* IEEE Computer Society Press, 214–223.

MEDINI, S., ANTONIOL, G., GUÉHÉNEUC, Y.-G., DI PENTA, M. et TONELLA, P. (2012). SCAN: an approach to label and relate execution trace segments. *Proceedings of Working Conference on Reverse Engineering (WCRE).* IEEE Computer Society Press, 135–144.

MEDINI, S., GALINIER, P., DI PENTA, M., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2011). A fast algorithm to locate concepts in execution traces. *Proceedings of the International Symposium on Search-based Software Engineering (SSBSE).* ACM, 252–266.

MILLER, G. A. (1995). WordNet: A lexical database for English. *Communications of the ACM*, 38, 39–41.

MITCHELL, B. S. et MANCORIDIS, S. (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32, 193–208.

NG, J. K.-Y., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2010). Identification of behavioral and creational design motifs through dynamic analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 22, 597–627.

PIRZADEH, H. et HAMOU-LHADJ, A. (2011). A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension. *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS).* IEEE Computer Society Press, 221–230.

PIRZADEH, H., HAMOU-LHADJ, A. et SHAH, M. (2011a). Exploiting text mining techniques in the analysis of execution traces. *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 223–232.

PIRZADEH, H., SHANIAN, S., HAMOU-LHADJ, A. et MEHRABIAN, A. (2011b). The concept of stratified sampling of execution traces. *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press, 225–226.

PORTER, M. F. (1980). *Readings in Information Retrieval*. Sparck Jones, Karen and Willett, Peter.

POSHYVANYK, D., GETHERS, M. et MARCUS, A. (2013). Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology*, 21, 1–23.

POSHYVANYK, D., GUÉHÉNEUC, Y.-G., MARCUS, A., ANTONIOL, G. et RAJLICH, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33, 420–432.

POSHYVANYK, D. et MARCUS, A. (2006). The conceptual coupling metrics for object-oriented systems. *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 469–478.

RAJLICH, V. et GOSAVI, P. (2004). Incremental change in object-oriented programming. *IEEE Software*, 21, 62–69.

REISS, S. P. et RENIERIS, M. (2001). Encoding program executions. *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 221–230.

ROHATGI, A., HAMOU-LHADJ, A. et RILLING, J. (2008). An approach for mapping features to code based on static and dynamic analysis. *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press, 236–241.

SAFYALLAH, H. et SARTIPI, K. (2006). Dynamic analysis of software systems using execution pattern mining. *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press, 84–88.

SARTIPI, K. et SAFYALLAH, H. (2010). Dynamic knowledge extraction from software systems using sequential pattern mining. *International Journal of Software Engineering and Knowledge Engineering*, 20, 761–782.

SHAFIEE, A. (2013). *Phase Flow Diagram: A New Execution Trace Visualization Technique*. Mémoire de maîtrise, Concordia University, Canada.

SIFF, M. et REPS, T. (1999). Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25, 749–768.

SOH, Z., SHARAFI, Z., VAN DEN PLAS, B., PORRAS, G. C., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2012). Professional status and expertise for uml class diagram comprehension: An empirical study. *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press, 163–172.

SRIDHARA, G., HILL, E., MUPPANENI, D., POLLOCK, L. et VIJAY-SHANKER, K. (2010). Towards automatically generating summary comments for java methods. *Proceedings of the International Conference on Automated Software Engineering*. ACM, 43–52.

SRIDHARA, G., POLLOCK, L. et VIJAY-SHANKER, K. (2011a). Automatically detecting and describing high level actions within methods. *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 101–110.

SRIDHARA, G., POLLOCK, L. et VIJAY-SHANKER, K. (2011b). Generating parameter comments and integrating with method summaries. *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE Computer Society Press, 71–80.

TOLKE, E., KLINK, M., TOLKE, L. et VAN DER WULP, M. (2004). Cookbook for developers of argouml: an introduction to developing argouml.

VILLE, B. G. R. (1999). *Formal Concept Analysis*. Mathematical Foundations, Springer.

WANG, X., POLLOCK, L. et VIJAY-SHANKER, K. (2014). Automatic segmentation of method code into meaningful blocks: Design and evaluation. *Journal of Software: Evolution and Process*, 26, 27–49.

WILDE, N., GOMEZ, J. A., GUST, T. et STRASBURG, D. (1992). Locating user functionality in old code. *Proceedings of Conference on Software Maintenance, 1992*. IEEECOMPSP, 200–205.

WILDE, N. et SCULLY, M. C. (1995). Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7, 49–62.

WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M. C., REGNELL, B. et WESSLÉN, A. (2000). *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers.

YEVTUSHENKO, S. A. (2000). System of data analysis "concept explorer". *Proceedings of the National Conference on Artificial Intelligence*. 127–134.

ZHANG, X., GUPTA, R. et ZHANG, Y. (2003). Precise dynamic slicing algorithms. *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 319–329.

APPENDIX A

PUBLICATIONS

The following is a list of our publications related to this dissertation.

Conference Articles

- **Soumaya Medini**, Giuliano Antoniol, Yann-Gaël Guéhéneuc, Massimiliano Di Penta and Paolo Tonella. SCAN: an approach to label and relate execution trace segments. Proceedings of Working Conference on Reverse Engineering (WCRE), October 2012, 135–144. This paper received the best paper award of WCRE'12.

- **Soumaya Medini**, Pilippe Galinier, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. A fast algorithm to locate concepts in execution traces. Proceedings of the International Symposium on Search-based Software Engineering (SSBSE), September 2011, 252–266.

Articles in Journal

- **Soumaya Medini**, Venera Arnaoudova , Massimiliano Di Penta , Antoniol Giuliano, Yann-Gaël Guéhéneuc, and PaoloTonella. SCAN: An Approach to Label and Relate Execution Trace Segments. Journal of Software: Evolution and Process 2014; volume 26, issue 11, 962–995.