TRACE ABSTRACTION FRAMEWORK AND TECHNIQUES

HEIDAR PIRZADEH

A Thesis
In the Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

APRIL 2012

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: **Heidar Pirzadeh**

Entitled: **Trace Abstraction Framework and Techniques**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_______________________________________Chair
Dr. Ion Stiharu
_______________________________________ External Examiner
Dr. Giulio Antoniol
_______________________________________ External to Program
Dr. Juergen Rilling
_______________________________________ Examiner
Dr. Ferhat Khendek
_______________________________________ Examiner
Dr. Jamal Bentahar
_______________________________________Thesis Supervisor
Dr. Abdelwahab Hamou-Lhadj

Approved by ____
Dr. M. Kahirizi, Graduate Program Director

____
Dr. Robin A.L. Drew, Dean
Faculty of Engineering & Computer Science

Abstract

Trace Abstraction Framework and Techniques

Heidar Pirzadeh, Ph.D.

Concordia University, 2012

Understanding the behavioural aspects of software systems can help in a variety of software engineering tasks such as debugging, feature enhancement, performance analysis, and security.

Software behaviour is typically represented in the form of execution traces. Traces, however, have historically been difficult to analyze due to the overwhelming size of typical traces. Trace analysis, more particularly trace abstraction and simplification, techniques have emerged to overcome the challenges of working with large traces. Existing traces analysis tools rely on some sort of visualization techniques to help software engineers make sense of trace content. Many of these techniques have been studied and found to be limited in many ways.

In this thesis, we present a novel approach for trace analysis inspired by the way the human brain and perception systems operate. The idea is to mimic the psychological processes that have been developed over the years to explain how our perception system deals with huge volume of visual data. We show how similar mechanisms can be applied to the abstraction and simplification of large traces.

As part of this framework, we present a novel trace analysis technique that automatically divides the content of a large trace, generated from execution of a target system, into meaningful segments that correspond to the system's main execution phases such as initializing variables, performing a specific computation, etc.

We also propose a trace sampling technique that not only reduces the size of a trace but also results in a sampled trace that is representative of the original trace by ensuring that the desired characteristics of an execution are distributed similarly in both the sampled and the original trace. Our approach is based on stratified sampling and uses the concept of execution phases as strata.

Finally, we propose an approach to automatically identify the most relevant trace components of each execution phases. This approach also enables an efficient representation of the flow of phases by detecting redundant phases using a cosine similarity metric.

The techniques presented in this thesis have been validated by applying to a variety of target systems. The obtained results demonstrate the effectiveness and usefulness of our methods.

To Sara, the love of my life

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Abdelwahab Hamou-Lhadj, for the support and latitude he gave me to follow my interests and create my own research. He pushed me when I needed to be pushed and complimented me on jobs well done. I thank him for his trust, for keeping his door always open for helpful feedback and conversation. More than anyone else, his influence has contributed to my development as a scientist.

Much of this dissertation has been published in different venues; I like to thank the anonymous reviewers for their comments. I am also grateful to Dr. Giulio Antoniol, Dr. Juergen Rilling, Dr. Ferhat Khendek, and Dr. Jamal Bentahar for taking the time to read this dissertation and to serve on my thesis committee.

Financial support for my research was provided by: FQRNT (Fond Québécois de la Recherche sur la Nature et les Technologies) through a postgraduate scholarship, the Graduate Student Support Program of Concordia University, and an NSERC collaborative grant held by Dr. A. Hamou-Lhadj.

Additionally, I would like to thank the engineers and researchers from Ericsson Canada and l'École Polytechnique de Montréal for their willingness to interrupt their schedules to meet with me. I hope that my findings may prove helpful as they seek to build effective tracing and monitoring tools and techniques.

I would like to thank everyone at the Software Behavioural Analysis (SBA) Research Lab especially Arya, Luay, Afroza, and Ali for their friendship, encouragement and stimulating discussions.

I want to thank all of my friends who made life in Montreal enjoyable. Mohak Shah, thank you for helping me in many different endeavors and for being a great friend. Sasan, Akanksha, and Maher thanks for all the great times.

I like to thank my parents, Parvin and Youssef, and my sister, Setareh, for their continuous support along the way.

Finally, an enormous thank you to my lovely wife, Sara. I cannot thank you enough for all the sacrifices you made throughout this whole process – the least of which were moving to Montreal, your many trips to Quebec City, putting up with my crazy work schedule, and providing me with your support, encouragement, and advice when I needed it the most. Amazingly, you managed to do these while simultaneously pursuing your own Ph.D. I could not have done this without you. You are my wife, my friend, my love, my life, and I love you with all my heart.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

The goal of this thesis is to present new techniques that facilitate the analysis of large execution traces for maintenance tasks. More precisely, the techniques we propose in this thesis are 1) to enable software engineers to have an abstract view of an execution trace by viewing it as a series of segments composed of trace events, we call these segments *execution phases*; 2) to provide software engineers with an ability to generate a representative sample of a trace for quick analysis 3) to provide capabilities to find the most representative events of each execution phase, thus extract the most important events of a trace.

The organization of this chapter is as follows: We present the motivations of our work in Section 1.1. We briefly review the concept of execution traces in Section 1.2. We present the contributions of the thesis in Section 1.3. Finally, in Section 1.4, we describe the organization of the rest of the thesis.

1.1. Problem and Motivations

A common and difficult problem experienced by software engineers when maintaining large complex systems is to understand how the system is built and why it is built in a certain way [DRW00]. Software engineers often spend considerable amount of time to understand the code, documentation, and tap into other available sources of information to build an understanding of the software system. This is often caused by lack of good documentation (if it exists at all) and the fact that the original designers have moved to new projects or companies. Tools and techniques that help in understanding the system can therefore reduce the time spent on maintenance tasks.

The existing tools and techniques can be categorized based on the type of data they use. Some techniques collect and analyze the data that they use without executing the system - these are referred to as static analysis techniques. Another type of techniques where the data is gathered from a running system is referred to as dynamic analysis.

As one can expect, these techniques have pros and cons. While the most predominant techniques are the ones that rely on static analysis, they tend to be limited to analyzing the static relationship among the system entities. Dynamic analysis techniques, as noted by [Bal99], can be very useful for applications that require the understanding of the system's behaviour by relating the system inputs to its outputs. Unlike static analysis, dynamic analysis techniques focus on only parts of the system that need to be understood. In other words, software engineers do not need to understand the entire system if only part of it needs to be modified. Dynamic analysis techniques, however, can only provide a partial picture of the system.

In dynamic analysis, the system is executed according to an execution scenario and the runtime data is typically stored in a file that is referred to as an execution trace. Traces, however, tend to be considerably large. Tracing even a small system can generate large amounts of data that poses a real obstacle to any viable analysis.

There exist several approaches (e.g., [GD05, HBAL05, ZD04]) that aim to reduce the size of large traces while keeping their main content. The common practice is to develop heuristics that can guide the trace abstraction and simplification process. Although significant improvement has been made in the area, existing techniques suffer from several limitations such as their reliance on particular visualization methods, which hinders their reuse. Visualization techniques also require extensive user intervention. In other words, it is up to the user to choose among the variety of available features to gain insights into a trace.

There is clearly a need for effective trace abstraction and simplification techniques that can reduce the time and effort spent on understanding the content of large traces.

The objective of this research is to propose an effective framework for trace abstraction along with a set of techniques that can effectively simplify understanding of large execution traces.

1.2. The Concept of Execution Traces

Many different aspects of a running system can be monitored and traced. In fact, one can trace just about any aspect of the system that is deemed helpful to accomplish the task at hand. These different aspects include method and procedure calls and returns, variable values, loops and branches, inputs and outputs, inter-process communication, executed statements, system state transitions, external interrupts, system calls, as well as data structures such as process control blocks. Recording the events related to one or more of the mentioned aspects in a file will result in an execution trace.

An execution trace, therefore, is a sequence of events (e.g., method calls, classes, system calls, etc.) resulting from the execution of the software system under study (also referred to as the target system). The execution of a system can be based on feeding the system with a set of predefined test cases or exercising one or more software features of the system. A feature is defined as an observable functionality triggered by a user [EKS03].

A trace event can have a number of attributes (e.g., nesting level, timestamp, code line number, the thread in which the event occurs, etc). The focus of this thesis is on traces of method calls. The term method here means also function and procedure. Although we focus on method call traces, the techniques presented in this thesis are readily adaptable to other types of traces such as system call and inter-process communication traces.

A trace of method calls can be represented as a tree structure as shown in Figure 1.1. In this figure, we can see an example of interactions between two objects of the classes `Test` and `SimpMath` that implement multiplication of numbers using repeated addition. In this figure, we only show the methods being involved. It is also possible to show the actual objects and the method parameters. However, many studies in program comprehension choose to ignore such information. The rationale is that building a high-level of understanding of a trace using method calls alone might turn to be as effective as if other information is used.

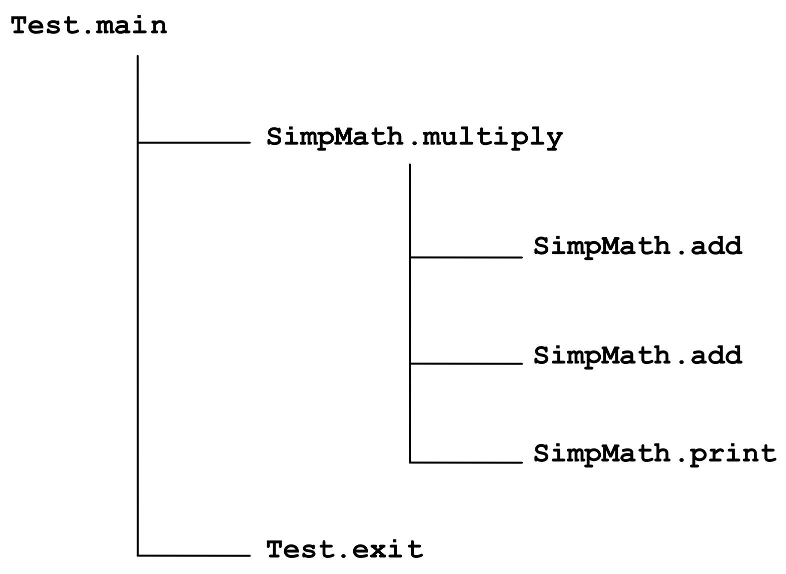


Figure 1.1. Example of trace of method calls

There exist different methods and tracing tools for generating execution traces. These methods usually *instrument* the system by injecting a piece of code (called probe) that will be invoked at the specified points. A probe can be, for example, a printout statement that outputs the desired data at the specified point. Instrumentation methods are different from one another in:

- Instrumentation site: different environments can be instrumented for generating a trace. Some tracing tools instrument the source code of the system while other tools instrument the bytecode (or the object file or a compiled version of the code) of the system. The execution environment in which the system runs can also be instrumented. Execution environment of the target system (be it a virtual machine or an operating system) can send out event notifications at certain points of the execution. Once received by the tool, the event can be added to a trace file.

- Probe injection points: it is also important to know where a probe is being injected. For example, in object-oriented systems, probes can be inserted into the body of a target method at entry/exit point to create events in the trace file that indicate the target method call's entry/exit. Another point of injection can be the body of any method that calls the target method (i.e., probe is injected into the caller method, and not the called method). The latter type of injections is useful when it is difficult or impossible to instrument the files containing the methods that need to be targeted.

- Instrumentation time: instrumentation can happen during or before the execution of the system. The latter case is referred to as static instrumentation. In this type of instrumentation, the instrumentation site is first modified on disk by injecting the probes. Then, the target system can be executed normally to have the probes collect the data. Once tracing is finished, the instrumentation site needs a clean up to its original state. In dynamic instrumentation, however, when the tracing tool (usually a profiler) is notified that a system component (e.g., a class, method, etc.) of the target system is being loaded, it modifies the in-memory representation of that component by inserting the probe [TPTP]. For this type of instrumentation, the tracing tool needs to run along with the target system. Unlike dynamic instrumentation, in static instrumentation, if the instrumentation environment is changed (e.g., a new feature is added or a bug is fixed), we need to do a clean-up (remove the probes) and perform another round of instrumentation.

1.3. Research Contributions

The major research contributions of this thesis are:

- A trace abstraction framework inspired by the human perception system.
- A trace segmentation approach that divides a trace into meaningful trace segments that characterize the main computations of the traced scenarios.
- An approach based on text mining for automatic extraction of the most important information conveyed in a trace.
- A technique for reducing the size of traces based on stratified sampling of the trace content.

1.3.1. Trace Abstraction Framework

We present an innovative approach for trace analysis inspired by the way the human brain and perception systems operate. There are psychological processes that have been developed over the years to explain how our perception system deals with huge volume of visual data. We draw parallels between trace analysis and the human perception system and propose a framework [PH11a, PH11b, PSHM11, PHS11] for trace analysis to facilitate the understanding of large traces.

1.3.2. Trace Segmentation

We present a novel trace analysis technique that automatically divides the content of a trace into smaller and meaningful trace segments that correspond to the program's main execution phases [PH11b, PAH10]. These phases construct a higher-level view of the execution trace that aim to simplify the exploration of large traces by allowing software engineers to browse the trace by focusing on its execution phases instead of a flow of mere low-level events. Our phase detection method is inspired by Gestalt laws that characterize the proximity, similarity, and continuity of the elements of a data space. We model these concepts in the context of execution traces to find execution phases. The effectiveness of the approach is shown through case studies.

1.3.3. Stratified Sampling of Execution Traces

To reduce the size of execution traces, sampling techniques, especially the ones based on random sampling, have been extensively used. Random sampling, however, may result in samples that are not representative of the original trace. In this thesis, we propose a trace sampling technique that not only reduces the size of a trace but also results in a sample that is representative of the original trace by ensuring that the desired characteristics of an execution are distributed similarly in both the sampled and the original trace [PSHM11]. Hence, the insights gained from analyzing the sample trace could be extrapolated to the original execution trace. Our approach is based on stratified sampling instead of random sampling and uses the concept of execution phases as strata. We show the effectiveness of our sampling technique through two case studies.

1.3.4. Identification of Relevant Events of a Trace

Motivated by the work done in the area of text mining, we propose a trace exploration approach based on examining the trace execution phases. The approach [PHS11] consists of automatically identifying relevant information about the phases as well as the ability to provide an efficient representation of the flow of phases by detecting redundant phases using a cosine similarity metric. We applied our approach to traces generated from two different systems and were able to quickly understand their content and extract higher-level views that characterize the essence of the information conveyed in these traces.

1.4. Organization of the Thesis

The rest of this thesis is organized as follows.

Chapter 2 - Background

This chapter presents the concepts and research areas that are related to our research. The chapter starts by briefly presenting software maintenance and program comprehension. The chapter continues with introducing different models of program comprehension. Next, reverse engineering is presented as a way of supporting program comprehension through static and dynamic analysis. The chapter proceeds with a detailed review of trace abstraction and simplification techniques. Trace visualization is then discussed along with introducing representation methods, interaction features, and navigation strategies.

Chapter 3 – Trace Abstraction Framework

This chapter starts by a discussion on the relationship between the human perception system and trace analysis. The chapter continues with a discussion on the psychological processes that govern the way the human perception system functions when looking at a scene. Inspired by these psychological processes, three novel trace analysis processes are proposed in this chapter. The chapter continues with introducing a framework that integrates the proposed trace analysis processes.

Chapter 4 – Trace Segmentation

This chapter starts by introducing the concept of execution phases and its potential applications. An approach for segmenting a trace into execution phases is then presented. The chapter proceeds by discussing different steps involved in our proposed trace segmentation technique. Evaluation of the trace segmentation technique is then presented

Chapter 5 – Content Prioritization

This chapter starts by explaining how automatic identification of the trace events that are most relevant to the implementation of each execution phase can be helpful in simplifying the exploration of large traces. The chapter proceeds by determining a mapping between the set of concepts in two domains of trace analysis and text mining. A content prioritization approach is proposed to extract the trace events that are most relevant to each execution phase. Different steps involved in this approach are discussed in details. The chapter proceeds by proposing a technique to find similar phases in a flow of phases. Evaluation of these techniques is then presented.

Chapter 6 – Stratified Trace Sampling

This chapter starts with discussing drawbacks of existing trace sampling approaches. The chapter continues with a theoretical description of the problematic cases of random sampling. Then, the concept of stratified sampling of execution traces is introduced as a way to solve the previously investigated problems. The chapter proceeds by explaining the process through which strata are specified. Then, the sampling process is described. The chapter continues by a running example. Two case studies are then presented and discussed as evaluations of the proposed stratified sampling approach.

Chapter 7 – Conclusion and Future Work

In the beginning of this chapter, we revisit the main contributions of the thesis. The chapter continues by describing opportunities for future research.

Chapter 2. Background

In this chapter, we introduce the background concepts that pertain to this thesis. In Section 2.1, we present the software maintenance as a related topic. In Section 2.2, we present program comprehension and its models. In Section2.3, we present reverse engineering and discuss static and dynamic analysis techniques. In Section 2.4, we present in detail existing approaches for trace abstraction and simplification for maintenance tasks. Finally, in Section 2.5, we present trace visualization of execution traces and its related concepts.

2.1. Software Maintenance

An integral part of the software lifecycle that accounts for a large portion of the total cost of the development of a software system is the maintenance of the system after its delivery. IEEE has categorized the maintenance activities into four major categories in its Standard for Software Maintenance [IEE98]:

- Adaptive activities: These activities are concerned with adjusting the system to adapt to changing external environments.

- Corrective activities: These activities deal with fixing discovered problems and bugs.

- Perfective activities: This type of activities is concerned with changing the system to enhance or add new features to the system.

- Preventive activities: the goal of this type of activities is to increase software maintainability and prevent problems in the future

The analysis of execution traces can help with many of the above task. For example, Jerding et al [JR97] showed the usefulness of analyzing execution traces in facilitating the corrective and adaptive maintenance tasks. Silva et al. [SPAM 11] conducted experiments that show how the analysis of execution traces can help in perfective maintenance task. Cornelissen et al [CZD11] also conducted a controlled experiment with a number tasks representative of real maintenance contexts to see how their trace analysis approach can improve the performance in terms of time spent and correctness.

2.2. Program Comprehension

Program comprehension is the study of how software engineers understand a system (usually in the absence of documentation and other reliable sources of information) before they can modify it. Basili showed that 50-60% of software engineering effort is spent on understanding the code [Bas97]. Similarly, Von Mayrhauser et al. [VV95] argued that for almost every type maintenance tasks software engineers need to understand the system (even if it is a partial understanding) before they can proceed. .

To understand a system, software engineers build a mental model that describes their mental representation of the system. Many models and strategies have been proposed [Pen87, VV95, Sto06] to explain the process through which software engineers build their mental models and obtain understanding about the system.

Top-down Model:

In this model of program comprehension, a software engineer is somewhat familiar with the system that he/she wants to understand. This familiarity can be due to his previous experience with performing a maintenance task on the system, knowing the technological aspect of the system, and so on. Because of this familiarity, the software engineer can make hypotheses about the code that he is investigating. To form a hypothesis, the software engineer uses his pre-existing knowledge to quickly find the code components (e.g., classes, methods, lines of code, etc) that he recognizes to act as cues to the presence of certain functionalities or structures. These cues are commonly referred to as *beacons*. A beacon can be a method name that shows the implementation of a specific feature. Detecting common code fragments, called *clichés,* that implement typical programming scenarios (e.g., sorting) is also used to form hypotheses.

Once a hypothesis is formed, the software engineer tries to evaluate the hypothesis based on the code. The process of evaluation is an iterative one where the hypothesis is whether refined, verified, or rejected. That is, in the evaluation process of a hypothesis, secondary hypotheses are made in a hierarchical manner until they can be matched to specific code in the system.

Bottom-up Model:

This model is commonly used when software engineers are unfamiliar with the system that they are going to understand. Unlike top-down model, bottom-up understanding starts at the code level and goes up to the level that a software engineer obtains an understanding about the whole system.

In the bottom-up model, the process of building a mental model proceeds by reading the code of the system, grouping the code statements, and forming mental abstractions about that group. The process of grouping is referred to as *chunking*. As mentioned earlier chunking is followed by relating the created groups of statements (chunks) to a higher level of abstraction. Building this relation is referred to as *cross-referencing*. The process of chunking and cross-referencing continues until software engineers obtain a high-level understanding of the system.

To explain how software engineers use chunking and cross-referencing to build their mental model in the bottom-up understanding, Pennington [Pen87] suggested that software engineers commence by building a program model. The program model is concerned with the control-flow of the code. More precisely, program model refers to an abstract representation of the control flow of the code that is obtained through a bottom-up investigation of the system. Once a program model is built, software engineers start developing a second model called situation model. The situation model represents the data-flow abstractions and functional abstractions of the system.

The Integrated Model:

In the integrated model of program comprehension, software engineers use both top-down and bottom-up strategies to understand the system at hand. Top-down strategy is used for parts of the system that are familiar to software engineers and bottom-up is used for the parts that are completely new to them. Software engineers switch between different strategies.

In the process of building a mental model, in addition to the program model and the situation model, software engineers use a knowledge base that represents the software engineer's knowledge gained so far through the bottom-up and top-down investigations of the system. Another source for updating the knowledge base is inferred knowledge.

Each of the mentioned models can be used to obtain a complete understanding of the system. However, as argued by [ES98], many software engineers take an as-needed approach, in which they focus only on the code relating to a particular task at hand which, in turn, results in a partial understanding of the system.

Tracing the control-flow and data-flow can be clearly used to build abstractions to gain an understanding of the program. Furthermore, strategies similar to bottom-up, top-down and integrated can be used to understand the execution traces. In this thesis, we only focus on understanding the flow of execution of a system by analyzing the method calls generated from executing the software features.

2.3. Reverse Engineering

One way to support the process of program comprehension is to use reverse engineering techniques. Reverse engineering can be defined as the process of obtaining a high-level and more abstract representation of the system from existing system artefacts such as the code. Software maintainers can use these views to speed up their comprehension process.

Reverse engineering techniques vary with respect to their sources of information to static analysis or dynamic analysis. Static analysis uses the source code as its main artefact to uncover the system's main components and their relationships. Performing static analysis has the benefit that all the system's execution paths could be potentially covered. However, it can only reveal the static aspects of the system. It is very limited to providing insights into the behavioural characteristic of a system's design.

Dynamic analysis, which is the focus of this thesis, is the study of how the system behaves by analyzing its execution traces. Unlike static analysis, dynamic analysis has the advantage of allowing the software engineer to focus only on parts of the system that need to be analyzed by studying the interactions among the involved components [Bal99]. There exist two types of dynamic analysis: Online (ante-mortem) analysis and Offline (post-mortem) analysis. Online analysis is the analysis of the behaviour of a system while it is running. This type of dynamic analysis comes handy when the system is not going to terminate its task any time soon (e.g., servers). In offline dynamic analysis, on the other hand, the analysis is performed when the execution is finished.

2.4. Trace Abstraction Techniques

As mentioned earlier, traces tend to be considerably large [RR01], which hinders any possible analysis [ZL01]. To address this issue, many trace abstraction and simplification techniques have been proposed with a common objective being to extract high-level views from raw traces. We categorize these techniques into four groups based on the criteria applied to reduce the size of a trace: 1) Event properties, 2) Patterns, 3) Techniques that are language-dependant, and 4) Sampling.

2.4.1. Event Properties

Many techniques focus on filtering trace events based on individual event properties. For example, one can filter all the events that occurred in a particular thread. Another example that has been frequently used is based on the nesting level of events. Rountev et al. [RC05] used this criterion to filter call graph in their technique. Cornelissen et al. [CDMZ07, CMZ08] used it for abstracting scenario diagrams by removing events that take place at nesting levels higher than a threshold. Kuhn et al. [KG06] also used a minimal nesting level threshold as one of several filtering criteria to reduce trace size. Other event properties such as execution time have also been used for filtering events from traces [MWM06]. Another possibility is to use metrics based on a variety of event properties. For example, in their trace summarization approach, Hamou-Lhadj et al. [HL06] proposed a metric to measure the extent to which an event can be considered a utility is or not. Utilities are then removed from the trace and a summary of the trace is constructed.

2.4.2. Pattern-based Techniques

Pattern-based techniques operate by grouping events into patterns for which software engineers can assign descriptions. The grouping is usually based on some sort of similarity among events. Systä et al. [SKM01] used Boyer-Moore string matching algorithm to detect patterns of events where identical events are repeated in sequences, referred to as behavioural patterns.

Hamou-Lhadj et al. [HL02] proposed an approach where the repeated instances of patterns of events are removed from a trace and represented only once with the objective of keeping only useful information. In their work, they started by a pre-processing step where the contiguous repetitions are removed. The next step was to find non-contiguous repetitions of events. For this, they used an algorithm for transforming a rooted call tree (the focus of their research) into an ordered directed acyclic graph. The result was a compressed trace in which repetitions were factored out.

Kuhn et al. [KG06] proposed an approach where the volume of data can be reduced by grouping sequences of events based the amount of changes in their nesting levels of routine call trees and replacing each group with the first event of that group along with the nesting level information of that group. This is followed by the application of several filters such as using the minimal nesting level threshold.

2.4.3. Programming Language-Based Filtering

Many programming languages add components that are dependent on the programming paradigm that is supported such as accessing functions and constructors in Object-Oriented

(OO) systems. Hamou-Lhadj et al. [HL06] proposed the removal of events from a trace as a step in their trace summarization process. In their algorithm, they filtered constructors and destructors because they did not implement core system operations. Accessing methods, nested classes, and methods related to programming languages libraries were among other kind of methods that were removed in their approach. Cornelissen et al. [CDMZ07] also removed accessing methods and their control flow, private and protected method calls, and constructors and their control flow to reduce the size of an execution trace.

2.4.4. Trace Sampling

Sampling techniques have also been used to reduce the size of traces (e.g., [CHMY03, LAZJ03, RR03, RZ05, Dug07]) just like in traditional information theory. Sampling consists of selecting parts of a trace for analysis instead of analyzing the entire trace. Existing approaches, however, have two major drawbacks: First, finding the right sampling parameters can be a difficult task and even if some parameters work well for one trace, they might not work for another trace (even if generated from the same system) [CHMY03]. The second and the most important drawback is that there is no guarantee that the resulting sample is representative of the original trace. This appears to be due to the fact that existing sampling techniques are blind to the information contained in the trace - they treat a trace as a stream of data for which the pieces are considered equal.

2.5. Trace Visualization

Visualization techniques have been used to facilitate trace exploration and analysis. There exist several tools that implement features that can be used by software engineers to abstract the content of traces (e.g. [SEAT, JR97, SKM01, DJM+02, CZD11]).

A trace visualization tool has a set of views along with a number of functional features. A view shows one or more execution traces or a number of their attributes via a visualization approach. A visualization approach combines representation methods, interaction features, and a number of navigation strategies (see Figure 2.1).

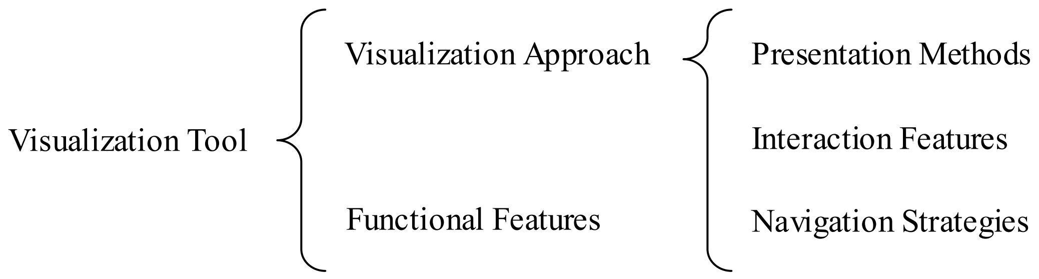


Figure 2.1. Decomposition of a visualization tool

2.5.1. Representation methods

The main factor in a visualization approach is its representation methods used to render the content of a trace. The representation methods include tables, charts, graphs, treemaps, and so on. Each of these representation methods is useful for understanding different types of information that can be derived or extracted from execution trace. For example, it is important to know where it is more preferable to use a table instead of a chart and what are the benefits and the limitations each representation method comes with.

2.5.2. Interaction Features

The interaction features allow users to interact with one or more elements of the representation method of a visualization tool. These features are used to establish and facilitate the dialog between the user and the presentation. This dialog is a sequence of interaction features (where one interaction is linked to another one) used to enrich user exploration and discovery. Examples of interaction features include highlighting (a method by which users can mark a representation element of interest to keep track of it) [BG98], rearranging of trace elements [Hya10], showing more/less details [SEAT], grouping/ungrouping of presentation elements EXTRAVIS [CHZ+07, CZH+08], and more.

2.5.3. Navigation Strategies

Navigation strategies can be regarded to as the use of a virtual camera across the presentation scene to enhance visualization. This camera can potentially move, change lens, and show overviews to help software engineers uncover where he is within the representation (context) and makes it possible for the user to go to other locations of interest and show them in more details (focus). Camera movement-based navigation can be seen when users perform panning, zooming, and scrolling. In Panning, the user grabs the scene and moves it in different directions using a mouse. Zooming gives the feeling to the user as if the camera is moving towards or away from the scene. Scrolling is similar to the panning except that the camera movement is controlled via scrollbars. The Histogram, Execution, Execution Pattern, and Reference Pattern views in Jinsight [DJM+02, DHKV93] and the liner view in ALMOST [RR99] are examples of views in trace analysis tools that support zooming and scrolling.

2.5.4. Functional Features

Functional features in a trace visualization tool are higher-level features that are supported by the tool to perform more advanced handling of a trace content including the application of trace abstraction algorithms, search, undoing/redoing, saving the current state and so on. The 'search' feature in trace analysis tools is used to retrieve trace events of interest just like in searching text. Depending on the size and the complexity of the system under analysis, the lack of a search feature in trace analysis tool can easily slow down or even make the whole investigation impractical. For example, although using EXTRAVIS brings about much convenience to maintainers, it suffers from the lack of search capabilities.

Shneiderman [Shn96] suggests that keeping history of the actions performed by a user is among the tasks that an effective tool should support. Therefore, it is necessary for trace visualization tools to keep a history of the performed actions. This history can be as simple as recording one step back and one step ahead or as complete as recording all the interaction performed by the user during the visualization.

Another important interaction feature consists of the ability to navigate through multiple views. For example, in SEAT, at any time, the software engineer can map the trace components to the source code (if available) to retrieve more information about the trace components. In ALMOST, double clicking on a method call in the linear view displays the corresponding implementation of the method in the source code view. Similarly, in VET [MWM06], the location within the execution trace and the information about the selected element are synchronized in the two available views.

Other advanced interaction features include animation in AVID [WMF+98], for example, software engineers can control the sequence of events they want to visualize by breaking the execution trace into a sequence of views called *cels*. Animation techniques are used to play the execution from one cel to another. There exist many other tools that support animated views such as VET [MWM06] and ExtraVis [CHZ+07, CZH+08].

Chapter 3. A Trace Abstraction Framework

3.1. Introduction

In this chapter, we present our framework for simplifying the analysis of large traces that is inspired by the way the human brain operates when dealing with information obtained through the visual sense. In the next section, we present the motivations for drawing such a parallel. We also list a number of processes that explain how the human perception system deals with visual data received from a scene.

In Section 3.3, we discuss how the human implicit perception of objects has motivated us to have a process that can find homogeneous segments of a trace. In Section 3.4, we discuss the global percept of a scene and suggest a process for trace analysis that can give us a trace that is in concept similar to a global percept by giving a representative sample of the trace. In Section 3.5, inspired by pop-out effect in human perception, we suggest a trace analysis process that finds important events of different parts of a trace. In Section 3.6, we explain how the mentioned trace analysis processes relate to one another and propose a trace analysis framework that incorporates those processes.

Parts of the material in this chapter is adapted and expanded from a paper published in the 33rd International Conference on Software Engineering (ICSE 2011), New Ideas and Emerging Results Track, 2011 [PH11a].

3.2. Human Perception vs. Trace Analysis

As discussed earlier, excessive size of execution traces, finding what is being performed in different parts of a trace, and building high-level views to help in program comprehension are among the difficulties of trace analysis. As suggested by Zayour et al. [ZL01] for an analysis to be practical the results need to be presented in an acceptable size so that the user is not overloaded with the information. Trace analysis, however, is not the only field where the users need to deal with large amount of data. Human perception is another area with similar challenges. More particularly, we noted three main commonalities between the two areas:

1. The amount of visual data received through our sensory is in general too high to be completely processed in detail so is the amount of information generated from a system run [FRC10].

2. The inability for the human brain to keep track of all relevant data in a specific domain of interest in both cases. There is a limited amount of information that can be handled by the human memory at any given time [Cow05]. Miller showed that short-term memory, or working memory, has a limited capacity (only 7±2 pieces of information, such as words or numbers, can be held at any one time) and cannot keep track of all the information from the visited knowledge domain [Mil56].

3. The need to acquire the necessary information in a relatively short time. In the case of trace analysis, time-to-market and other constraints such as the criticality of the analysis makes it necessary to obtain the needed information as quickest as possible. The same holds for human perception. Our perception system works in a way we grasp the essence of a scene within a small fraction of a second [FRC10]. This remarkable speed in gaining the information contributes to low response time and quick reactions.

There are a number of processes that are proposed in psychology to explain how the human brain and the perception system automatically (not voluntarily) deal with massive volumes of visual data considering limited short-term memory and necessity of a short response time. Some of the processes that attracted our attention are described below. We are however aware that the human perception system works in a complex way and that the following list of processes is far from being exhaustive [UKM06, FRC10, Boo02, GRT93, TG80, SF99]:

- Implicit Perception: It appears that our perceptual system segments local elements against their context and integrates them as objects and regions.

- Global Percept: The segmented scene is then quickly scanned so as the brain obtains an overall impression of it.

- Preattentive Process: The scene is analyzed in more detail by visiting the regions in a certain order. The pop-out effect is an important factor in this process.

In the next subsections, we elaborate more on the ideas behind these processes and suggest how trace analysis techniques can be developed based on these processes.

3.3. Data Segmentation

When we look at a scene, information flows from the physical sources and elements of the scene into our sensory device (i.e., eyes). But how do we start the analysis of the data received from the scene? As observed by Bowers et al. [BRBP90], first, an implicit perception of the scene occurs in which coherent regions and areas of the scene are identified. The implicit perception "precedes and guides a person to a conscious perception of [detected regions]" [BRBP90]. Thus, the first step in an implicit perception of a scene is finding and locating coherent segments of the scene.

Given that different points of the scene carry no significance and are not segmented as groups when they reach the sensory device, a grouping mechanism [Kof99] can explain the detection of regions. The same mechanism explains why we see a scene as objects and surfaces that represent our best guesses of the meaning of the particular scene. Similar mechanism might be useful in trace analysis. By analogy, a trace can be seen as the scene in which the points and lines represent trace events. What we need is to develop a mechanism that groups these events into something meaningful that can help software engineers quickly understand what is happening in a trace. In the next chapter, we propose a mechanism that automatically identifies the main computations of the traced scenario using the ideas discussed here. However, before we propose our mechanism, we need to further dig into the human perception systems to understand how the grouping of scene elements is done. This is explained through a set of laws known as Gestalt laws of perception that we describe in the following subsection.

3.3.1. Gestalt Laws

To explain how human perception performs, Musatti [Mus31] suggested that we perceive the grouping of scene elements that comprises the most homogeneous or uniform organization. Koffka [Kof99] and Wertheimer [Wer58] also suggested that our perception tends to group element that results in the simplest and most homogeneous (in certain characteristics) and regular organization of each group. Gestalt laws of perception describe how people group items visually based on their perception [Kof99, SF99, QB09]. Gestalt psychology [QB09] is an application of physics to essential parts of brain physiology describing the processes occurring in the brain when we see visual objects and how our perceptual systems follow certain grouping principles to integrate the scene elements (i.e., objects and regions) as a whole and not just as points and lines. These laws explain how our perceptual system segments local elements against their context and integrates them as objects.

Law of Similarity

In a scene, elements that have similar characteristics are often distinguished from other elements as they tend to be seen as a group. This type of grouping is referred to as the Gestalt law of similarity. For example, when asked to describe the shapes illustrated in Figure 3.1, the majority of respondents will most likely point out to successive columns of squares and circles despite the fact that the figure can also be described as rows of combined squares and circles. This is because the human perception system follows unconsciously a certain similarity-based grouping principle when it comes to interpreting a scene.

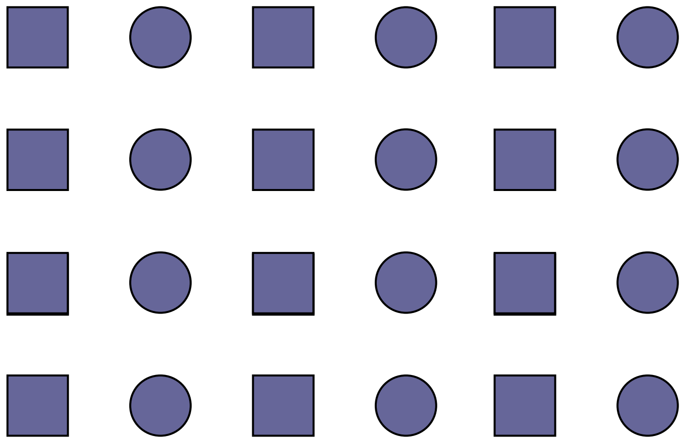


Figure 3.1. Gestalt law of similarity

The law of similarity does not force a certain scene dimension. That is, the elements of the scene can be distributed in two dimensions as in Figure 3.1 or in a single dimension (a linear representation) as shown in Figure 3.2. In this figure, also, similar elements tend to be grouped together as a group of `a`s and a group of `b`s.

a a a a b b b b

Figure 3.2. Similarity in a sequence of elements

Law of Good Continuation

The law of Good Continuation [GPSG01] refers to the tendency of things to be perceived as a group if they are visually co-linear or nearly co-linear. The lines shown in Figure 3.3 (a) tend to be interpreted as the Figure 3.3 (b) and not as Figure 3.3 (c). The law of good continuation explains why we perceive the elements of a scene as a smooth flow rather than yielding abrupt changes.

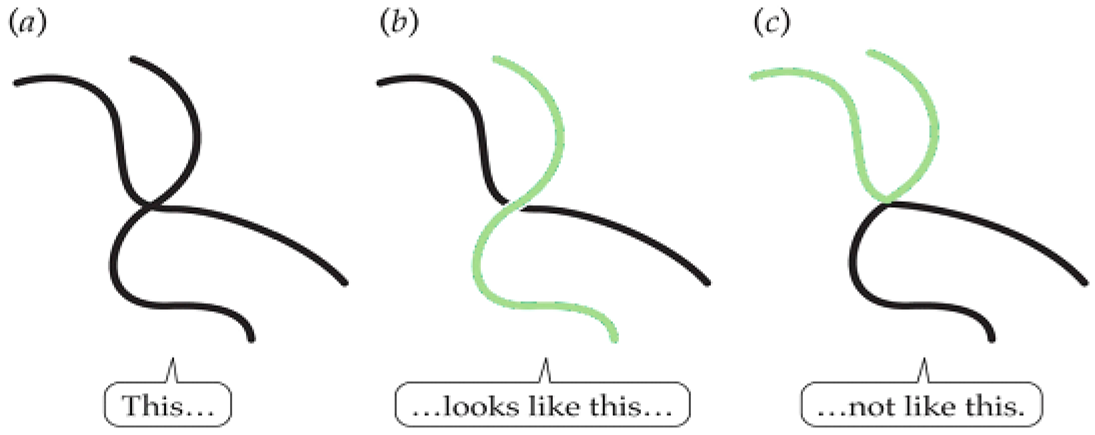


Figure 3.3. Gestalt law of good continuation (from [WKL+08])

Law of Proximity

The Law of Proximity, the most fundamental law of Gestalt laws, states that "being all other factors equal, the closer [in terms of distance] two elements are to each other the more likely they are to be perceived as belonging to the same group" [SF99].

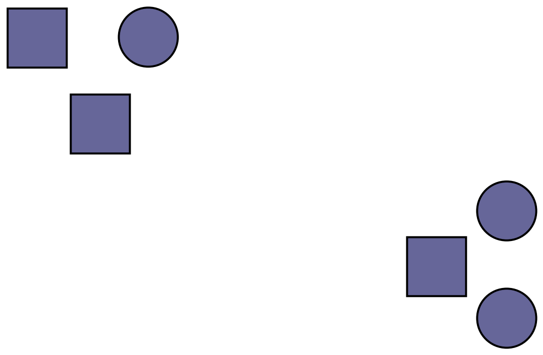


Figure 3.4. Gestalt law of proximity

As shown in Figure 3.4, although the shapes on the top left-hand-side of the figure are different from one another, they form a group because of their close distance. The same holds for the shapes on the bottom right-hand-side of the figure that form a group.

3.3.2. Segmenting a Trace

Inspired by how human perception segments a scene we suggest trace segmentation as an initial step in analyzing a trace. A trace as a flow of segments can provide the user with a high-level view that can help in trace exploration. Furthermore, each segment of the trace is composed of trace events that have similar behaviours and characteristics that can isolate them from the events in other groups. Thus, such segmentation can yield a behavioural model of the trace, where the execution flows from one phase to another.

Grouping mechanisms, similar to Gestalt laws of similarity and good continuation, can be used to find the places where the behaviour of the execution changes and to segment the trace at those locations.

3.4. The Gist

In the first steps of analyzing a scene, we have the impression to see the entire scene, and rarely focus on the details. This representative image, called the *gist* of the scene, is provided by performing a single short sampling that lasts in the order of 0.1 sec [Oli05]. Boothe explains that the sampling mechanism cannot be a dumb process, as "it would seriously limit our ability to maintain a high-fidelity perceptual database" [Boo02]. A more intelligent sampling needs to be performed. This smart sampling is suggested to be evolutionarily advantageous because it can speed up information processing. Finding the gist of a scene is a remarkable aspect of our perception that provides us with the ability to understand the meaning of a complex novel scene very quickly. A similar idea can be applied to trace analysis to quickly find a sample that shows the general context of a trace and that is representative of its contents.

3.4.1. Building a Gist

Fast scene perception suggests that we do not need to perceive the details of the scene to identify its gist. As suggested by Olivia et al. [OT06], the general context of a scene can be inferred from the spatial properties of the elements of the scene. Many studies (e.g., [SO94, Bar04]) have suggested that elements and regions of low spatial frequencies in a scene represent global information about the scene. For example, in Figure 3.5, the right panel shows an image containing only the low spatial frequency whereas the left panel shows the original image containing the entire spectrum. Our perception system is likely to perceive the image in the left panel at a glance by the low spatial frequencies in the image as shown on the right panel.

The low spatial frequency regions in a scene are the regions that are most homogeneous. As shown Figure 3.6, low spatial regions in a figure are the ones where the characteristics (e.g., color) of pixels on a reference plot (that covers an area 16 pixels) does not change abruptly and significantly.

Uchida et al. [UKM06] suggest that processing limited low-level information from short chunks could facilitate rapid construction of global percept of a scene. An effective process could be one that performs sampling on each homogeneous segment of the scene rather than on mere unstructured details of the scene.

In execution traces, similarly, the initial global information about the trace might be extracted by applying sampling that samples the homogeneous regions of a trace instead of blindly sampling the entire trace.

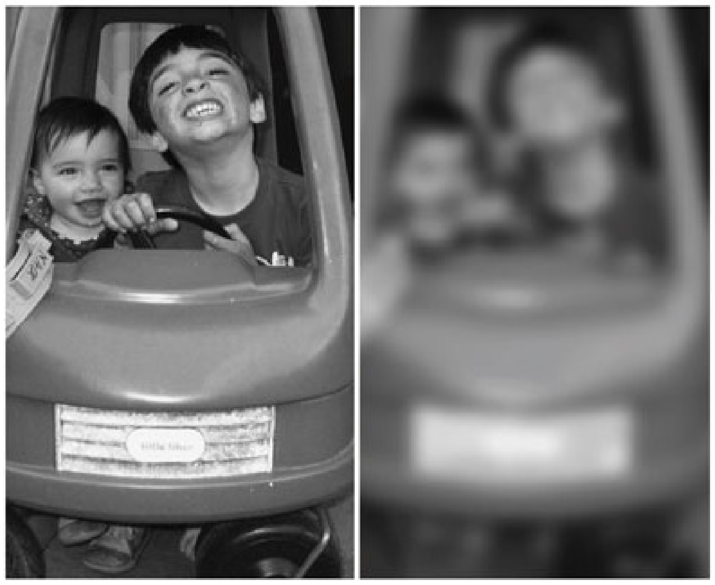


Figure 3.5. Spatial frequencies convey different information about the scene (form [Bar04]).

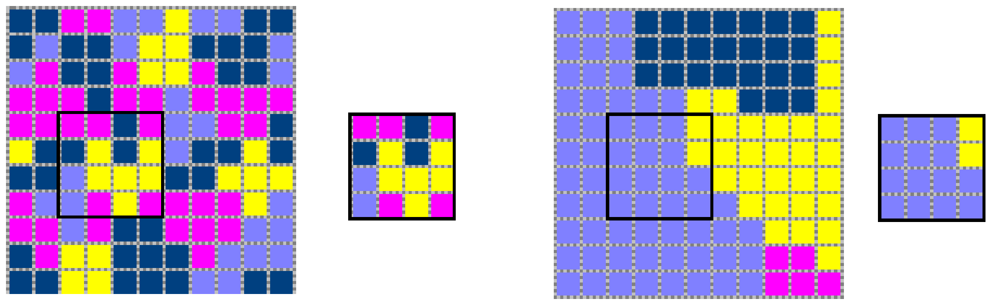


Figure 3.6. Examples of regions with high and low spatial frequencies

3.4.2. Trace Sampling

In the domain of trace analysis, regular and random sampling techniques have often been used to reduce the size of execution traces (e.g., [Dug07, CHMY03]). In general, sampling techniques are concerned with selecting a sample of a trace for analysis instead of analyzing the entire trace. However, since trace sampling is often not based on information about the trace (e.g., distribution of the trace events, its homogeneous subsequence, outliers, etc.) it may result in a sample that is not representative of the original trace.

Inspired by the process of obtaining the gist in the human perception system, we have proposed an effective sampling process [PSHM11] that makes use of proportional stratified sampling techniques studied in Information Theory. In stratified sampling [Coc77], first, the trace needs to be segmented into non-overlapping exhaustive subsets that are homogeneous (called stratum) and then sample instances are drawn from each stratum. By doing this, we guarantee that the final sample contains elements that are representative of every part of the trace.

3.5. Pop-out Effect

When we look at a scene, parts of the scene might quickly draw our attention. This phenomenon is referred to as the "pop-out effect". The pop-out effect is also explained in evolutionary terms: it may often be necessary for survival to notice scene elements that have different properties than ones of their surroundings, even if we are not explicitly looking for them, because they may be predators, preys, etc. [Ros99]. In general, odd elements pop out from a larger group of homogeneous elements. For example, as shown in Figure 3.7, a red circle among green circles pops out and draws our attention.

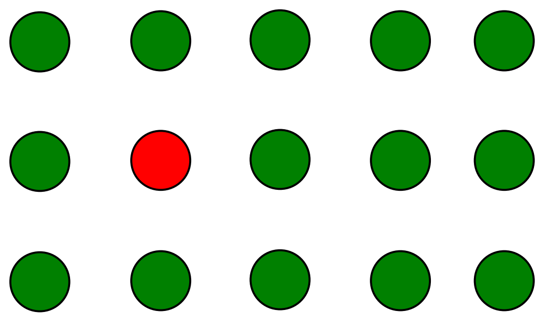


Figure 3.7. Pop-out effect

The pop-out effect leads to rapid detection of elements that differ greatly from surrounding elements usually in single dimension such as color or orientation [TG80, FRC10, WC99]. Although many models that explain the pop-out effect focus on local differences between each scene element and its neighbours, one could imagine scenes in which the variation in more distant elements (and not the local ones) increases or decreases the degree in which an element could pop-out [Alv11]. In general, a high frequency of an element in one region shows the importance of that element whereas an element that is scattered across different regions is considered less important. In other words, the elements that appear more times in one region and not in many other regions of a scene are the ones that pop-out.

Inspired by the pop-out effect in human perception where some scene elements are given more priority during the analysis, for execution traces, we need to develop a mechanism that can help us extract important events from a trace by a process that prioritizes the events based on how they appear in different parts of the trace.

3.5.1. Important Events of a Trace

The idea of weighting trace events based on their frequency has been proposed in many studies (e.g., [Dug07, Bal99]) to detect the events that do not follow the same frequency distribution (they are invoked considerably more than the other events). However, simply relying on mere frequency, we do not think that it is sufficient to detect important events of a trace. In fact, Durgerdil et al. [Dug07] showed that the events that appear frequently all over the trace, called temporal omnipresent events, are the least important events.

Inspired by the pop-out effect in human perception, we suggest a new technique that takes into account both frequency of events and their appearance in homogeneous segments of a trace. This will be, in nature, similar to term frequency – inverse document frequency (TF-IDF) in information retrieval [Joa98]. In our proposed technique, a weight assigned to each trace event (term) in each segment (document). The events that are more representative of a segment obtain higher weights in that segment and lower weights are assigned to less representative events. We propose a weighting function in which higher weight is assigned to a trace event that appears often in a particular segment, but appears not in many other segments of the trace.

3.6. Proposed Framework

We introduce, in this section, a framework for trace analysis that is inspired by three processes that were discussed in the previous sections. Our proposed framework is composed of three components (see Figure 3.8). We discuss trace segmentation, smart sampling, and content prioritization components in more details in Chapter 4, 5, and 6.

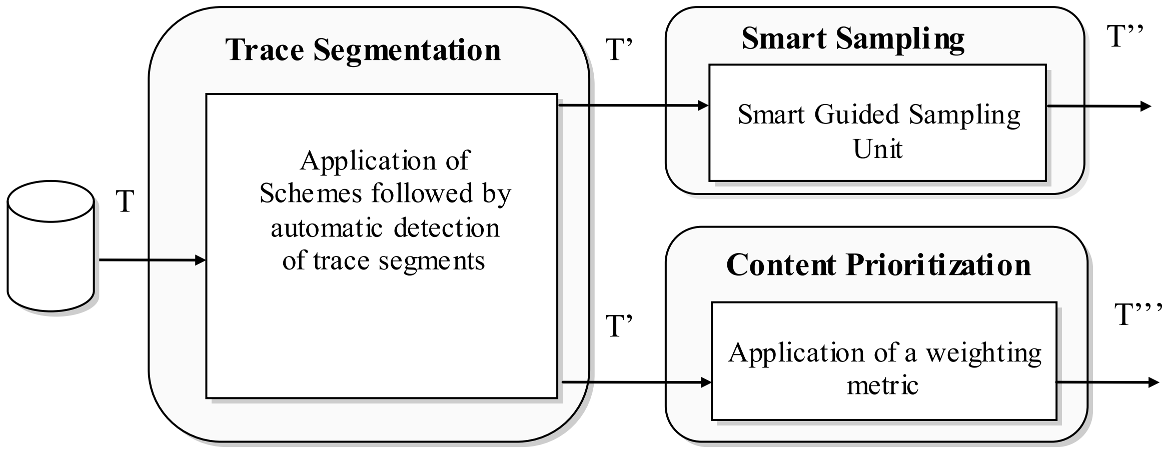

Trace Segmentation
Application of Schemes followed by automatic detection of trace segments
T
T’
T’
Smart Sampling
Smart Guided Sampling Unit
T’’
Content Prioritization
Application of a weighting metric
T’’’

Figure 3.8. Our proposed framework and its components for trace analysis

The first component of our framework, the trace segmentation component, aims to divide a large trace into meaningful and homogeneous segments that that we call execution phases. Examples of execution phases could be initializing variables, applying a specific algorithm, etc. To perform trace segmentation, we use a number of schemes inspired by Gestalt laws. Given a trace T as input, the trace segmentation component applies the schemes on the trace content and will result in a trace T’ which is an annotated trace where the execution phases are indicated.

As shown in Figure 3.8, in smart sampling of an execution trace, we use the trace homogeneous segments (execution phases) to serve as strata in the sampling process. In other words, once the phases are detected (i.e. the trace T’ is obtained –Figure 3.8), we start the stratified sampling process, which is implemented in our framework, as part of the smart sampling component. This component receives a phased execution trace T’ as its input and outputs a sample of the execution trace using stratified sampling. Since the events within each stratum are homogeneous, we perform the selection of trace events from each stratum using random sampling. The size of a sample of each phase is relative to the size of the phase. The result of this phase in a sampled trace T'' which is a smaller trace than T' and yet representative of the content of T'.

As shown in Figure 3.8, the content prioritization component, receives a phased trace T' as input and outputs a trace T''' containing only the representative events of each phase. For this, the content prioritization component ranks the trace events of each execution phase. The ranking is based on the frequency of occurrence of each event in a phase and the number of other phases in which the same event has occurred.

3.7. Summary

In this chapter, we investigated possible analogies between the two fields of human perception system and trace analysis. We focused on three processes that happen in our perception system when we look at a scene. These processes are preattentive and therefore, unlike attentional processes, they can be different from one person to another.

Inspired by the psychological processes that govern the human perception system, we introduced processes for trace analysis. The first process aims to identify different segments of an execution trace that groups trace events that collaborate to a common task. Schemes similar to Gestalt laws can be applied to a trace to find its homogeneous segments.

We proposed a second process for generating a representative sample of an execution trace. This process was inspired by the process of building the gist of a scene in our perception system. For this process, we proposed a stratified trace sampling process where strata are the segments detected in the first process.

Given that not all events in a trace have the same importance, the last process that we proposed in this chapter is intended to prioritize events according to their relevance to different parts of a trace. This process is inspired by the pop-out effect in human perception that helps in quick identification of elements of a scene that are different from others.

Finally, we proposed a framework that integrates the proposed processes so that the results obtained by one can be used, if needed, by other processes. Three components in our framework are intended to implement the three proposed processes for trace analysis.

Chapter 4. Trace Segmentation

4.1. Introduction

In this chapter, we focus on the problem of creating abstractions from large traces. Trace abstraction is useful in itself to reduce the size of traces and simplify their understanding for the human viewer. We present a number of algorithms by which a trace can be divided into smaller and more manageable trace segments that characterize the main execution phases of the traced scenario. We call this trace segmentation. For example, a trace that is generated from a compiler will contain events that represent the various compiler's phases including parsing, preprocessing, lexical analysis, semantic analysis, and so on. Knowing where each of these phases occurs in the trace is usually a challenging task since there is no support at the programming language level of how to explicitly indicate the beginning and end of each phase.

Trace segmentation should not be confused with the techniques that extract very high level views of a system execution used for redocumentation and general understanding of the system behaviour (often shown as UML sequence diagrams) such as the ones we developed in [HBAL05, HL06]. These techniques are not designed to uncover the specific computational phases of the traced scenario. On the other hand, the phase detection methods can be further refined to build higher-level views of the system.

We believe that a technique that could automatically identify these phases (and their sub-phases) has numerous benefits:

- Software engineers can easily navigate through the trace content by viewing it as a flow of execution phases instead of mere low-level events

- Phases can provide important information on how a particular feature is implemented, which in turn can help software maintainers enhance these features.

- Phases can be further refined to recover a high-level behavioural view from raw traces, enabling the understanding of the traced scenario.

- Phases might be helpful in fault localization as they can show in what phase of the system's execution the error has occurred. An execution shown as phases can be an excellent means of communication between maintainers, developers, programmers to obtain a quick and clear idea and description of the system.

To the best of our knowledge, this is the first time that automatic extraction of execution phases from a trace is attempted.

Some visualization approaches such as the ones presented in [CHZ+07] offer an overview of the execution trace that helps the user to detect segments of an execution trace that are visually distinguishable from one another. This overview shows the call relations between the methods from different classes and packages that can be used to visually detect the different segments of the system's execution. Being aware of the execution scenario, the user can also hypothetically relate the repetitive (or ordered) patterns of events to the repetitive (or ordered) features in the execution scenario. Users then need to verify their hypothesis by going deeper in the trace content. Unlike our approach that automatically finds the trace segments, their approach does not provide an actual segmentation for the trace data, and the burden of detecting, hypothesizing, and verifying the segments is on the user's shoulders.

In their tool called Jive, Reiss et al. [Rei05, Rei07] visualize the behaviour of a Java program as it is running at certain intervals of time. The statistical information about the system's behaviour during each interval is captured at the class and thread levels (e.g., numbers of invocations from one class, the information about the behaviour of the threads during execution) in the form of a vector. While the system is executing, the current vector (which belongs to the last interval) is compared with a general vector (a vector that represent the statistics of the system since last three intervals). The two vectors not being "close enough" can indicate a new phase. The vector for the new phase is then compared with the saved vectors of all previously detected phases. If the vector of any previous phase is close to the current vector, that segment is being repeated, otherwise, this is a completely new segment. The duration of an interval, the amount of closeness when comparing the vectors, and the number of intervals that are considered in computing the general interval are the parameters that can significantly impact the quality of phase detection in their approach. Furthermore, the user is only provided with some statistics of each phase, which may not be much helpful in understanding the tasks delivered by the phase. Visualization-based approaches commonly require human intervention and are open to human interpretation.

Another online phase detection technique proposed by Watanabe [WII08] is based on the investigation of Least Recently Used (LRU) cache for observing objects that are created or destroyed during a system's execution. The assumption is that at the beginning of each phase many new objects are created to implement the tasks in that phase and once the tasks are delivered the corresponding objects are destroyed, therefore, a significant change in the cache (list of current objects) can show the emergence of a new phase. The proposed approach has a number of shortcomings. Several parameters (cache size, window size, threshold, and phase search distance) control their phase detection algorithm. However, no solution is suggested on how to tune these parameters. Changing one or more of these parameters can result in different phases. As the authors also mentioned, their algorithm falls short when one method is contributing to one or more features. Furthermore, once the phases are detected, the user needs to manually investigate each phase contents to understand what is happening inside the phases (no phase description is provided). Handling repeated phases is another issue that is not addressed in their approach.

Kuhn et al. [KG06] suggest a possible analogy between analysis of trace information and signal processing. For this, they first transform method call traces into time series by plotting the nesting level of the calls against points in time through the execution. Then, they apply trace size reduction technique based on events interaction where sequences of events are grouped based on the amount of changes in their nesting levels and each group is replaced with the first event of that group. This volume reduction makes it possible to visualize a large number of calls in multiple traces on a single screen. This way, users can manually identify similar phases within a trace and between the traces. This technique does not take into consideration the method names when preparing the plot to match patterns between trace signals. In addition, it removes a lot of information that it considers inessential data from a trace by applying multiple filtering logics (independent of method names), having as target mainly the representation size. This could potentially result in loss of important trace information during the abstraction process.

This chapter is organized as follows: in Section 4.2, we define the concept of execution phases. In Section 4.3, we introduce our proposed phase detection approach along with its different steps and components. In Section 4.4, we explain how the phases are located on the execution trace. In Section 4.5, we propose a method for automatic tuning of the parameters as an optional step. In Section 4.6, we discuss the tool support for our approach. In Section 4.7, we present the case studies to evaluate our proposed sampling approach. Finally, in Section 4.9, we conclude this chapter with a summary.

Parts of the material in this chapter is adapted and expanded from a paper published in the 16th IEEE International Conference on Engineering of Complex Computer Systems, 2011 [PH11b].

4.2. Reasoning about Execution Phases

We define an execution phase as a segment of a program's execution that performs a specific task. At a very high-level, one could argue that any program is composed of three major phases (Figure 4.1): The initialization phase, the computation phase, and the finalization phase. Each phase can be further decomposed into smaller sub-phases that implement specific sub-tasks of the program.

Execution phases can appear at various levels of a system's execution [Rei07, SPHC02, WII08]. At the highest level of abstraction, a system's execution can be considered as an algorithm or a general procedure for solving a specific problem. The phases in this case are the key steps of the algorithm. At a lower level of abstraction the execution phases of a system implemented in a specific programming language are segments of the system's execution that collaborate with each other to implement a specific task ([Rei05, Rei07]). The lowest level of abstraction of a system's execution is presented as machine code. The execution phases in this level can show how the system accesses and uses resources. For example, the phases can show distinctive patterns of hardware usage (e.g., CPU usage, memory access, communication ports access) or stable states of machine resources during the execution as noted by Sherwood et al. [SPHC02].

This thesis focuses on identifying the key execution phases that compose a system's execution at the source code level. In such context, we want to be able to take an execution trace (generated from exercising one or more particular features) and identify execution phases where each phase performs a portion of the overall program part that is being traced. Each phase denotes a step of the flow of execution and phase transitions denote the logic of the execution flow from one step to another. This way, the understanding of how a feature is implemented will presumably no longer require, at least in the beginning, that its corresponding trace be explored as a flow of mere low-level events. Thus, a trace can be seen as a sequence of execution phases, in which a phase denotes a step of the general execution and the transitions among phases depict the logic that connects the phases. Moreover, each phase can be further decomposed into a number of smaller sub-phases, hence, varying the granularity of execution phases.

For example, Figure 4.1 shows a system's execution composed of three major phases: The initialization phase, the computation phase, and the finalization phase. The logic of the flow (represented by the numbers over the transition) shows that from initialization, we go to computation and from there we go to finalization.

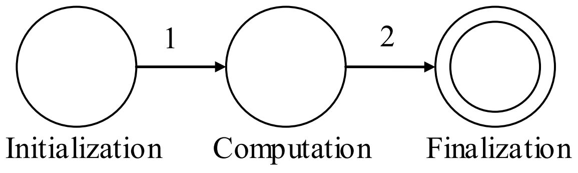


Figure 4.1. High-level phases of a system's execution

4.3. Phase Detection Approach

Recall that an execution phase is a segment of a system's execution that exhibits common behaviour at a level the programmer would recognize. For example, Figure 4.2 shows a very simple trace composed of several calls to two methods `a` and `b`. The figure also shows a ruler that is used to indicate the position of the calls in the trace (e.g. the first call to `a` appears in position 1, the second call in position 2, etc.). If asked to identify the major phases that appear in this trace, a programmer would most likely perceive two major phases: The first one is composed of the calls to `a`, while the second phase could consist of the calls to `b`.

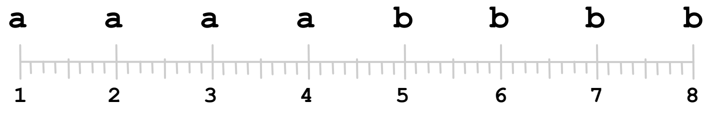


Figure 4.2. A sample trace

As the trace grows in size and complexity, manual detection of phases becomes harder. The complexity here can be defined as the number of new methods that are invoked in a trace.

Figure 4.3 shows an overview of our approach for automatic detection of execution phases from traces. The phase detection component receives a trace (hereby referred to as the original trace) as its input. This component analyzes the original trace in one pass, automatically dividing its content into the system's main execution phases. The output of the phase detection component is an annotated version of the original trace where the execution phases are marked (hereby referred to as the phased trace). The phase detection component is composed of two main units: "Application of Gravitational Schemes" and "Clustering and Mapping". In the application of gravitational schemes, two schemes are applied on the original trace to group its similar and continuous events into dense groups. The output of this unit is an intermediate representation of the original trace where the events are rearranged by the gravitational forces. We refer to this intermediate representation as the rearranged trace. The Clustering and Mapping unit then uses a clustering algorithm to locate the dense groups on the rearranged trace and maps them back to the original trace as execution phases. This unit outputs a phased trace.

The effect of applying each of the similarity and continuation schemes is as follows:

- ***Similarity scheme***: By applying this scheme, the events in the trace are rearranged in a way that the distance between similar trace events is reduced. We consider two method calls similar if they call the same methods.

- ***Continuation scheme***: The application of this scheme results in the repositioning of the trace events in a way that the consecutive events are made closer one to another if there a continuous change (no sudden jump or drop) in the values of a certain attribute of the events. For example, in traces of method calls, the consecutive method calls that are in the calling nesting level are made closer to one another to emphasize a trend in the execution of the system.

The two gravitational schemes that we have developed are also aligned with the fact that a phase change in an execution trace corresponds to a significant change in the pattern of attributes of the events in the trace over time [WII08, Rei05]. Therefore, our strategy can be seen as reducing the distances between the events for which the characteristics can form a pattern specifying a phase. Again, this is similar in principal to the way a human brain automatically groups points and lines into shapes and regions as explained in the previous chapter.

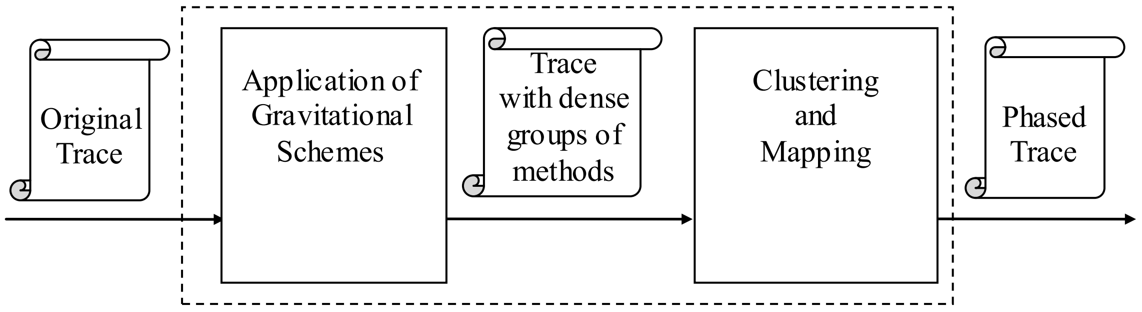

Original Trace
Application of Gravitational Schemes
Trace with dense groups of methods
Clustering and Mapping
Phased Trace

Figure 4.3. Detailed view of the execution phase detection unit

To help with the description of these techniques, we introduce the following definitions: a trace T of size n (the number events, here, method calls invoked in the trace) is a tree, where each node is a method call denoted as $c_{i,d}$ where i represents the invocation order of the method call c and d (depth of the node) shows the nesting level of the call. Each method call $c_{i,d}$ can result in calls of zero or more methods, with $c_{i+1,d+1}$ as its first callee, if any.

$$T = \{c_{1,0}, c_{2,d}, \ldots, c_{i,d'}, c_{i+1,d''}, \ldots, c_{n,d'''}\}$$

To apply our gravitational schemes, we define the distance between the method calls in the trace. The difference in invocation orders between the method calls in the trace is considered as distance between the method calls and it is assumed that there is equal distance of one between consecutive invocations in the original trace. For instance, the distance between $c_{i,d}$ and $c_{j,d'}$ would be equal to $|j-i|$. This way, we map the ordinal scale of method calls to an interval scale. Furthermore, we define the function $Pos(c_{i,d})$ to return the position of the method call *c* in the interval scale (i.e., on a ruler). The position of a method call is also the order in which it was invoked right after the trace is generated. However, as the method calls are rearranged as a result of applying the two schemes, the new position of a method call might differ from its original order of invocation. We use this rearranged trace to find the phases of the original trace.

4.3.1. The Similarity Scheme

The objective of the similarity scheme is to reposition the events of a trace in such a way that similar events gravitate to each other forming a group of dense events, which could indicate the presence of a phase. In other words, the events of a trace are repositioned in a way that the distance between two same events is less than the distance between two different events given that the difference in terms of the invocation order is the same for the events of both pairs. A simple repositioning scheme based on the similarity scheme, which we refer to as Pos_{sim}, and which divides by half the distance of similar methods changes the position of method calls as follows:

$$Pos_{sim}(c_{i,d}) = \begin{cases} Pos_{sim}(c_{j,d'}) + \dfrac{i - Pos_{sim}(c_{j,d'})}{2} & \text{if C1} \\ i & \text{Otherwise} \end{cases}$$

$$C1 : \text{there is a previous call } c_{j,d'} \text{ to the same method}$$

We visit each method call $c_{i,d}$ in the original trace. If there is a previous method call $c_{j,d'}$ to the same method, we reposition $c_{i,d}$ to half way from $c_{j,d'}$ (i.e., by reducing the distance to half). Otherwise, we do not change its position ($c_{i,d}$ remains in i-th position).

A pseudo code that implements the similarity scheme is shown in Figure 4.4. As shown in the figure, the function `CalculateSimilarity` calculates a new position for method call `currentEvent` and returns `similarityPosition` as output. `CalculateSimilarity` uses a data structure `positionTable` that, for each method, keeps the position of the last call to that method in the trace. Line 1 initializes `similarityPosition` to zero and line 2 initializes the `similarityDenominator` to 2. `similarityDenominator` sets the fraction by which two similar method calls are made closer to each other. Line 3 sets `currentName` to the name of the method invoked in `currentEvent` and line 4 finds the position of `currentEvent` in the original trace (recall that the order of a method call is the position of that method call in the original trace, thus, `currentOrder` is set to the order of `currentEvent`). Line 5 sets the `currentPosition` of the `currentEvent` equal to its `currentOrder`. Line 6 checks if there has been any other method call to a method with `currentName` earlier in the trace. If so, the algorithm finds the distance `offset` between the `currentEvent` and the previous call to the method invoked in `currentEvent` (line 7 and 8). Line 9 sets the `similarityPosition` to half of offset from the position of the previous call to the method invoked in `currentEvent` and line 10 updates the position of the method in the `positionTable`. If the no similar method has been previously invoked (i.e., we do not have a method with `currentName` in the `positionTable`), line 12 adds the current method and its position to the `positionTable`. Finally, as the result of the algorithm, Line 13 returns a new position `similarityPosition` for `currentEvent`. In this pseudo-code, by setting `similarityDenominator` to 2, we chose to reduce the distance between calls to the same method by half, although one could use a different measure. The focus here is on the fact that the calls to a same method are placed closer to each other to form a dense group (see Section 4.5 for more on fine tuning `similarityDenominator` parameter).

```
Algorithm CalculateSimilarity ( currentEvent )
Input: the method call currentEvent that we want to update its position.
Output: a new position similarityPosition for currentEvent.
Begin
1:      similarityPosition ← 0
2:      similarityDenominator ← 2
3:      currentName ← name of currentEvent
4:      currentOrder ← order of currentEvent
5:      currentPosition ← currentOrder
6:      if similarityTable contains currentName
7:          previousSimilarEventPosition ← position of currentName in positionTable
8:          offset ← currentPosition - previousSimilarEventPosition
9:          similarityPosition ← previousSimilarEventPosition + offset / similarityDenominator
10:         update the position of currentName in positionTable with similarityPosition
11:     else
```

```
12:         similarityPosition ← currentPosition
13:     return similarityPosition
End
```

Figure 4.4. Calculating the position of a method call according to the similarity scheme

Figure 4.5 shows the result of applying the similarity scheme to the sample trace of Figure 4.2. The formation of two dense groups of method calls could indicate the presence of two phases. The first phase begins at the first method invocation (and contains calls to a) and the second phase starts at the fifth method invocation (calls to b). After using the similarity scheme, even if the similarity of the items in each of each group becomes imperceptible, the groups still can be recognized by their structure and the distance between them. This is shown in Figure 4.6 where we replaced all method calls with "●".

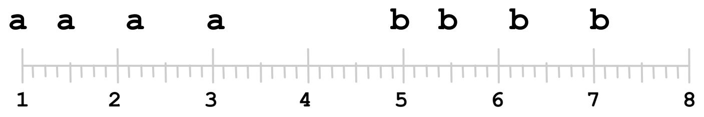


Figure 4.5. The result of applying the similarity scheme to the trace of Figure 4.2.

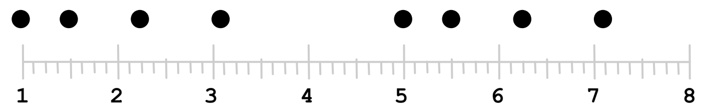


Figure 4.6. Events are shown as dots

In Figure 4.7, we show the effect of applying the similarity scheme to another sample trace (this trace does not reflect real world traces and is used here for illustration purposes only). The resulting trace appears to contain two dense groups that could indicate the presence of two distinct phases. The first one starts at the first invocation and is composed of calls to methods a and b, while the other one which starts at the seventh invocation contains calls to c and d. Figure 4.7 (step 3) shows the same result when discarding the effect of similarity in perception (replacing all methods with "●"). Although the size and complexity have increased compared to the sample example of Figure 4.2, one can still quickly recognize two phases based on the formed groups. As discussed in Section 3.3.1, structurally recognizable groups can are explained by Gestalt law of proximity. Thus, one may conclude that the similarity scheme technically converts similarity to proximity.

Although in this work, we only consider the similarity between the names of method calls, one can define other similarities (e.g., cohesiveness either from a structural or from a conceptual point of view) and apply the scheme introduced here.

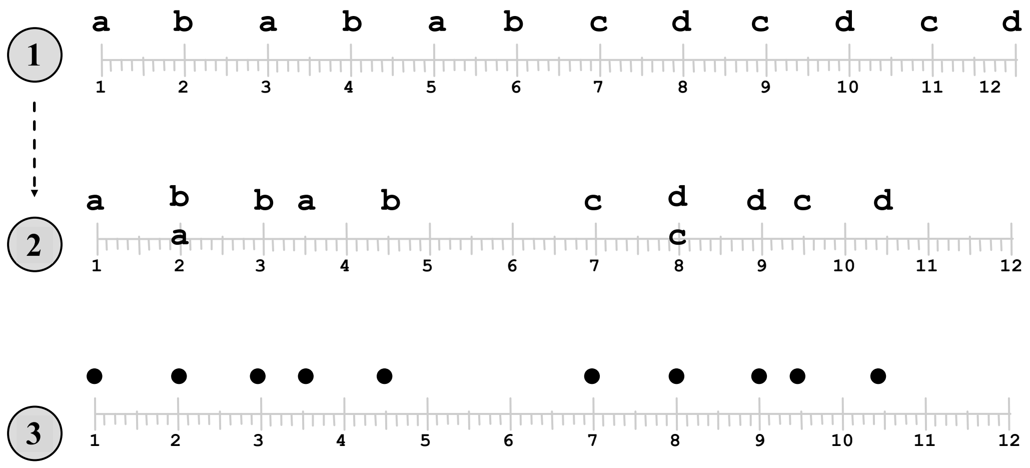


Figure 4.7. The result of applying the similarity scheme to a sample trace

4.3.2. The Continuation Scheme

The similarity scheme works well for a trace that contains similar events. But what happens if there is no noticeable similarity between the events of an execution trace? For example, Figure 4.9 (step 1) shows an execution trace where it is hard to distinguish the similar nodes -no distinct method is invoked more than once. However, one can perceive two different segments in this trace. As discussed in Section 3.3.1, this perception can be explained by the Gestalt law of Good Continuation. We use the continuation scheme to group trace events using the nesting level of the method calls.

For example, one can notice that there is a good continuation between the calls from a to o, which can intuitively suggests the existence of a phase. Using the nesting level of calls to detect execution phases has also been the topic of other studies [KG06, WII08]. Watanabe et al. [WII08] used the nesting levels of a call tree to detect phases and locate phase shifts. The authors suggested that the depth of the call stack (i.e., the nesting level) is a local-minimum at the beginning of a phase indicating a phase transition. They also showed that the events that have a high nesting level (i.e., which are deep in the tree hierarchy) were unlikely to initiate new phases.

The continuation scheme groups trace events by keeping the method calls with higher nesting levels closer to the previous method calls. The higher the nesting level of a method call, the stronger it is attracted by the previous method call. A continuation scheme that repositions the events of a trace based on their nesting level, and that we call here Pos_{cont}, is as follows:

$$Pos_{cont}(c_{i,d}) = \begin{cases} Pos_{cont}(c_{i-1,d'}) + \dfrac{1}{d} & \text{if } d > d'/2 \\ i & \text{Otherwise} \end{cases}$$

When applied to a trace, this scheme reduces the distance between method calls based on the nesting level (d) of the callee by changing the distance of two consecutive method calls from 1 to 1/d. The condition $d > d'/2$ disables gravity for the cases in which the nesting level of the current method call is drastically lower than the nesting level of the previous method call (i.e., the case of local minimums). For example, a call with a nesting level 6 that immediately occurs after a call with a nesting level 12 will not be repositioned because it indicates a drastic change in nesting levels ($6 \leq 12/2$) and thus a possible phase shift.

The pseudo code for the continuation scheme is shown in Figure 4.8. As shown in the figure, the function `CalculateContinuity` calculates a new position for method call `currentEvent`, and returns `continuityPosition` as the calculation result. Line 1 initializes `continuityPosition` to zero. Line 2 sets `currentNestingLevel` to the nesting level of `currentEvent` and line 3 sets `currentOrder` to the order of `currentEvent`. Line 4 sets the `currentPosition` of the input method call to the order of the method call `currentOrder`. Line 5 sets `continuityDenominator` to 2. `continuityDenominator` sets the fraction by which the variation in nesting levels is considered significant. Line 6 checks if the method call `currentEvent` is not the first method call of the trace (i.e., there has been a previous method call). If so, `prvNestingLevel` is set to the nesting level of the previous method call and `prvcontinuityPosition` is set to the position of the previous method call. Line 9 checks whether the change from `prvNestingLevel` to `currentNestingLevel` is significant or not. If not significant, the algorithm sets the position of current event `continuityPosition` closer to `prvcontinuityPosition` according to `currentNestingLevel`. Otherwise, `continuityPosition` will not change (it will be equal to `currentPosition`). Finally, as the result of the algorithm, Line 16 returns a new position `continuityPosition` for `currentEvent`. In this pseudo-code, by setting `continuityDenominator` to 2, we chose not to reposition subsequent subtrees where the nesting levels vary by more than half. A different value for `continuityDenominator` could be used as long as a significant change among subtrees can be identified (see Section 4.5 for more on fine-tuning this parameter).

```
Algorithm CalculateContinuity ( currentEvent )
Input: the method call currentEvent that we want to update its position.
Output: a new position continuityPosition for currentEvent.
Begin
1:   continuityPosition ← 0
2:   currentNestingLevel ← nesting level of currentEvent
3:   currentOrder ← order of currentEvent
4:   currentPosition ← currentOrder
5:   continuityDenominator ← 2
6:   if we have a prvEvent
7:        prvNestingLevel ← nesting level of prvEvent
8:        prvContinuityPosition ← position of prvEvent
9:        if currentNestingLevel > prvNestingLevel / continuityDenominator
10:            continuityPosition ← prvContinuityPosition + (1 / currentNestingLevel)
11:       else
12:            continuityPosition ← currentPosition
13:  return continuityPosition
End
```

Figure 4.8. Calculating the position of a method call according to the continuation scheme

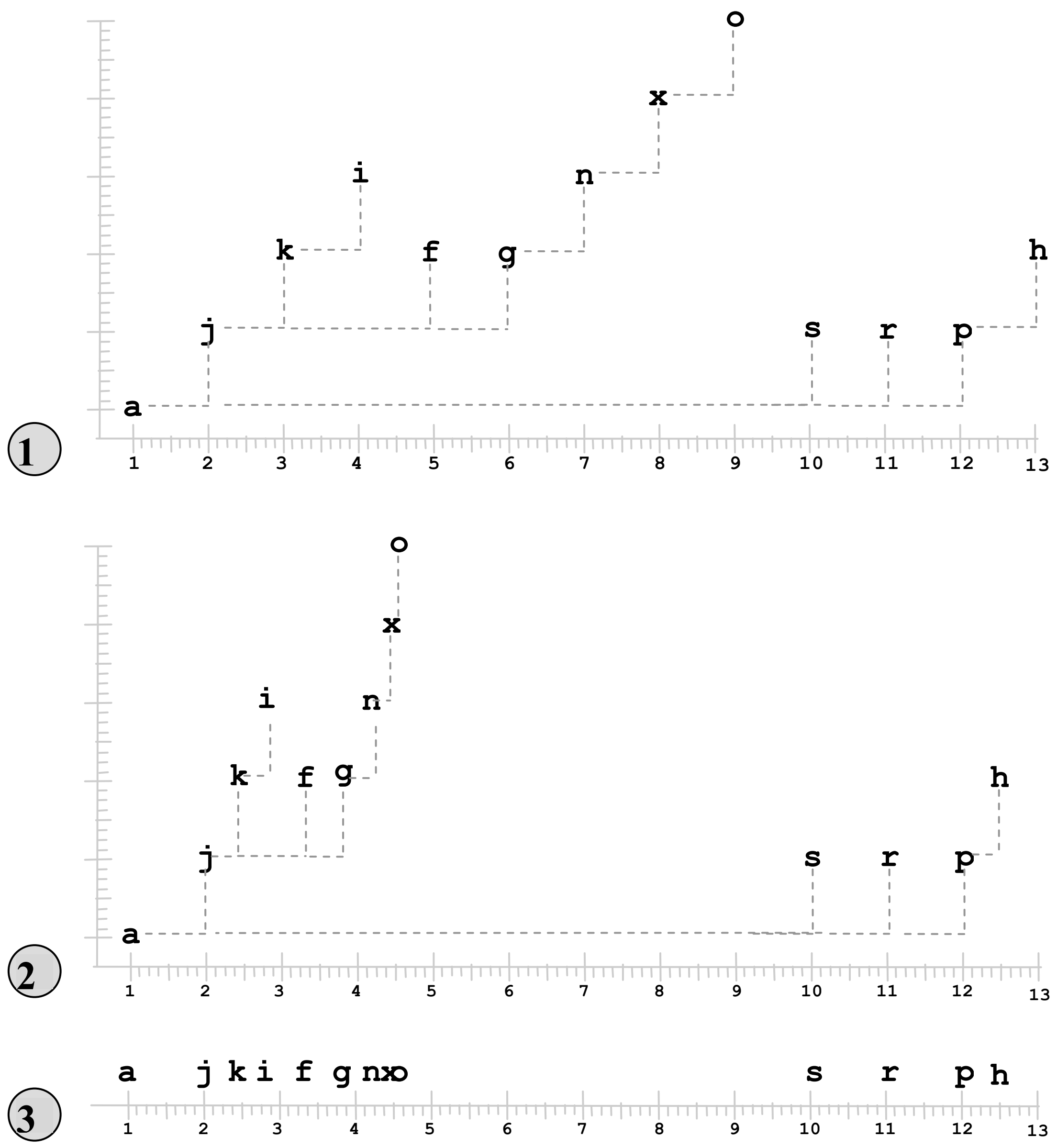


Figure 4.9. The result of applying the continuation scheme to a sample trace.

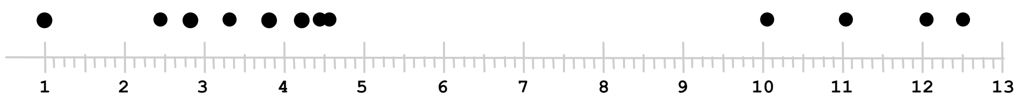


Figure 4.10. The resulting trace with the routines replaced with dots

Figure 4.9 (step 2) shows the result of applying the continuation scheme to the sample trace Figure 4.9 (step 1). As we can see, the new positioning of the trace events leads to two distinguishable groups of method calls. The first phase begins at the first method invocation and the second phase starts at the tenth method invocation. This way, we used the effect of good continuation in perceptual grouping to build groups that are structurally recognizable. Even if the manual detection group of method calls bearing good continuation becomes harder due to the size and complexity of a trace, the application of continuation scheme automatically forms dense groups of method calls that are recognizable through their structure and the distance between them. If we omit to visualize the nesting level (see Figure 4.9 (step 3)) and replace the methods with a "●" (Figure 4.10), we can clearly see that two phase have been formed. We may say that the continuity scheme technically converts continuation to proximity.

4.3.3. Integration of Schemes

We combine the similarity and the continuation schemes into an integrated scheme to facilitate their application. When applied to a trace, the integrated scheme first reduces the distance between method calls based on their nesting level (as we have in the continuation scheme), followed by the application of the similarity scheme. This results in reduction of the distance between calls to the same methods.

The integrated repositioning scheme can be iteratively applied to a trace to detect major phases, their sub-phases, etc, until we reach the individual events of the trace. To harness gravity so that phases could be detected with different levels of granularity, a threshold *t* is introduced to prevent two considerably distant methods from attracting each other and hence forming a large block. More precisely, the threshold *t* works in such a way that a call to a method *m* is attracted to a previous call to the same method only and if only the distance between these two calls is less than the threshold. Major phases can be detected by setting a threshold *t* that is close to the size of the trace. An integrated scheme with threshold t changes the position of method calls as follows:

$$Pos_{sim}(c_{i,d}) = \begin{cases} Pos_{sim}(c_{j,d'}) + \dfrac{Pos_{cont}(c_{i,d}) - Pos_{sim}(c_{j,d'})}{2} & \text{if C1 \& C2} \\ Pos_{cont}(c_{i,d}) & \text{Otherwise} \end{cases}$$

where

C1 : there is a previous call $c_{j,d'}$ to the same method

C2 : $Pos_{cont}(c_{i,d}) - Pos_{sim}(c_{j,d'}) < t$

and

$$Pos_{cont}(c_{i,d}) = \begin{cases} Pos_{cont}(c_{i-1,d'}) + \dfrac{1}{d} & \text{if } d > d'/2 \\ i & \text{Otherwise} \end{cases}$$

Figure 4.11 shows a pseudo code that implements the integrated scheme. The pseudo code is a modified version of the algorithm that implements the similarity scheme (Figure 4.4). In the modified version, along with the `currentEvent` its `continuityPosition` (calculated via `CalculateContinuity`) is also passed to the algorithm to find the final position of the `currentEvent`. Line 1 initializes `IntegratedPosition` to `continuityPosition`. If there was a previous call to the method invoked in `currentEvent` (i.e., we have this method in the `positionTable`) line 4 fetches the position of that method call in the original trace (recall that the order of a method call is the position of that method call in the original trace). Lines 5 to 10 apply the similarity scheme on the `currentEvent` if the distance between the currentEvent and the previous call to the method invoked in `currentEvent` in the original trace is less than `t`. Finally, as the result of the algorithm, Line 16 returns a new position `integratedPosition` for `currentEvent`. In this pseudo-code, `t` is passed as an input to the algorithm. The smaller the threshold `t`, the more fine-grained phases we can detect. We anticipate that the threshold is application-specific and that our tool that supports this approach allows enough flexibility to vary the threshold (see Section4.5 for more on fine-tuning of this parameter).

```
Algorithm IntegratedScheme ( continuityPosition, currentEvent, threshold )
Input: the method call currentEvent that we want to find its position. The position
continuityPosition of currentEvent calculated by method CalulateContinuity. Threshold
threshold.
Output: a new position integratedPosition for currentEvent.
Begin
1:    integratedPosition ← continuityPosition
2:    similarityDenominator ← 2
3:    currentName ← name of currentEvent
4:    if similarityTable contains currentName
5:        previousSimilarEventOrder ← order of currentName in positionTable
6:        if continuityPosition – previousSimilarEventOrder <= t
7:            previousPosition ← position of currentName in positionTable
8:            offset ← continuityPosition - previousPosition
9:            integratedPosition ← previousPosition + offset / similarityDenominator
10:           update position of currentName in positionTable with integratedPosition
10:       endif
12:   return integratedPosition
End
```

Figure 4.11. Calculating the position of a method call according to the integrated scheme

4.4. Identifying the Beginning and End of Phases

Once the method calls of a trace are repositioned according to the integrated scheme and dense groups of method calls are formed on the rearranged trace, we map the groups back to the original trace as phases.

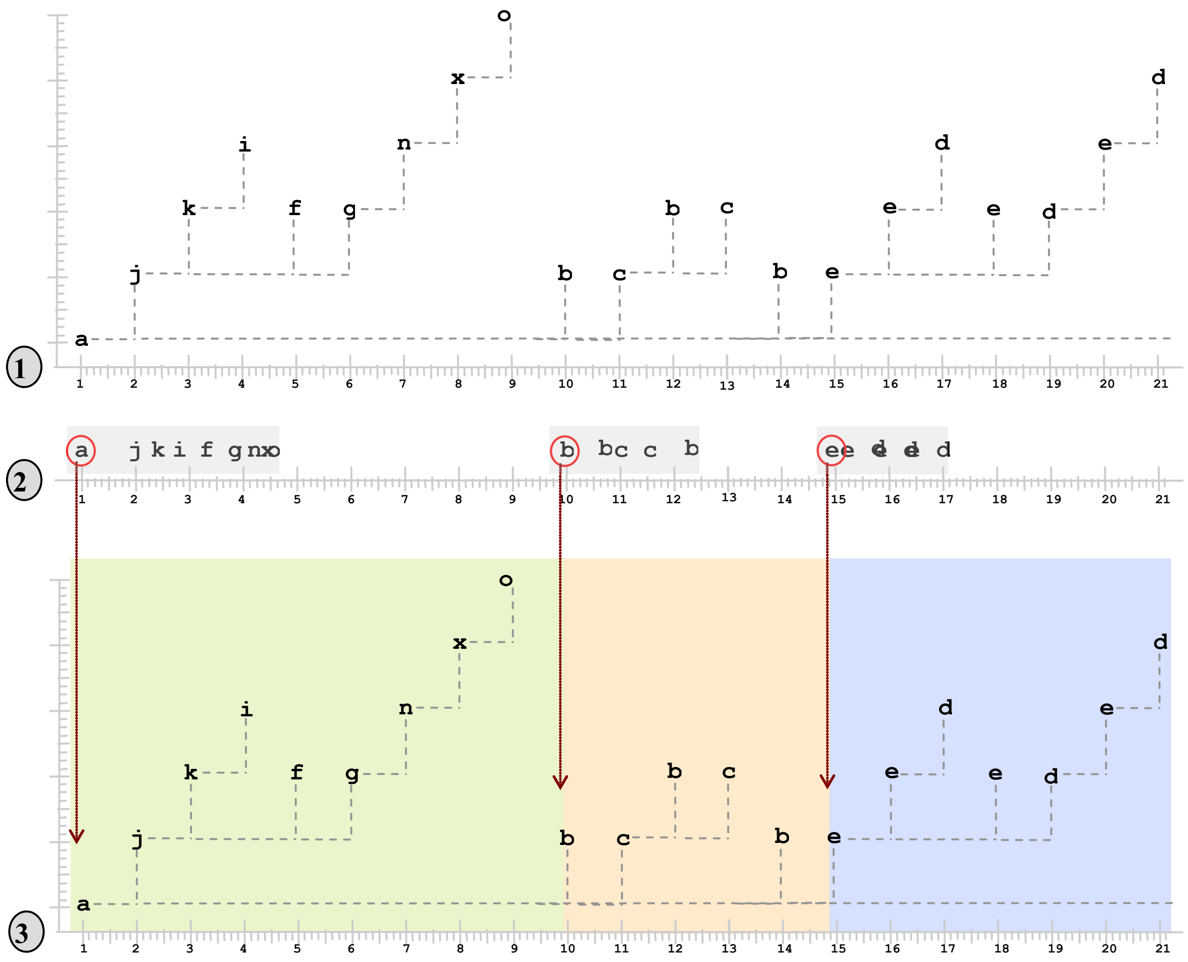


Figure 4.12. Mapping clustered groups to phases on the original trace

Figure 4.12 shows the steps that are taken to perform this mapping. First, we need to automatically locate the groups on the rearranged trace because it would be impractical to expect from programmers to distinguish the various groups visually for considerably large traces. By locating, we mean to identify the beginning and end of each group on the rearranged trace. We use a clustering algorithm to automatically find the beginning and the ending method call of each dense group based on the positions of method calls. The beginning method call of each group is then marked as the beginning of a corresponding phase on the original trace. Therefore, the beginning of each phase in the original trace is marked by the beginning of a corresponding group in the rearranged trace and the ending of that phase is marked by the beginning of next phase.

4.4.1. Clustering Algorithm

In [TH98], a number of hierarchical and partitional clustering algorithms and their applications to software engineering are presented. In this thesis, we chose a partitional clustering algorithm, the K-means clustering [Mac67], as our clustering algorithm. We chose to use K-means clustering because of its simplicity and observed speed [Vas07]. The study of the impact of various clustering algorithms on our approach is left as future work. K-means is an unsupervised clustering technique that partitions the data points into a predetermined number (*K*) of non-hierarchical clusters. The algorithm starts by choosing *K* data points (in our case data point are the method calls of the rearranged trace) as initial centroids ($\mu_1 \cdots \mu_K$). Each of these points is an initial center of a cluster. The rest of the algorithm is iteratively performed according to the below two steps, trying to minimize the overall sum of distances of the points from their cluster centroids:

1. Each instance *x* is assigned to the cluster with the closest centroid (the distances in our case are Euclidian):

$$x \in D_i \text{ if} \|x - \mu_i\| < \|x - \mu_j\| \forall j \neq i$$

where D_i is the set of points that have μ_i as their nearest centroid.

2. : Update the centroid of each cluster by moving it to the center of assigned points to that cluster:

$$\mu_i = \frac{1}{R_i} \sum_{x \in D_i} x$$

where $R_i = |D_i|$. The iteration continues until we have the same cluster assignment in two successive iterations.

K-means clustering yields a partitioning where data points close to a centroid are grouped together as a cluster. It should be mentioned that performing clustering with a different value for *K* results in different partitioning of data. Therefore, it is important to input the input a *K* value that is as close as possible to the actual number of dense groups (see Section4.5 for a discussion on how to automatically find the number of dense groups)

4.5. Parameter Tuning

Recall that given an execution trace, our approach builds a rearranged trace by applying the integrated scheme on the original trace. The quality of our phase detection approach depends on how it automatically locates groups of events on the rearranged trace to map them to phases on the original trace. The automatic localization of groups, itself, depends on the following factors:

1- How the methods are repositioned to form dense groups: as discussed in Section 4.3.3, the integrated scheme repositions the method calls in a way that methods that bear similarities are made closer to one another to form dense groups. Our integrated scheme, by default, halves the distance between calls to the same method. As shown in Figure 4.11, this reduction in distance is controlled by the parameter `similarityDenominator`. The algorithm also reduces the distance between the callees and callers as long as the nesting level of the callee is greater than half of the nesting level of the caller. That is, the drop in nesting levels by half is considered a significant change in nesting levels. As shown in Figure 4.11, the detection of a significant change is controlled by the parameter `continuityDenominator`. The user can change both of these repositioning parameters as long as they results in reducing the distance between calls that bear similarities.

2- How the number of dense groups is detected: as mentioned in Section 4.4, to correctly locate each dense group, our clustering needs the number of dense groups (*K*) as its input. This parameter can be set, for example, based on visual inspection of the rearranged trace and manual counting of the number of dense groups.

Thus, our phase detection involves fine-tuning of a number of parameter (e.g., *K* in K-means clustering, `similarityDenominator` in the integrated scheme) to automatically (and adequately) locate the dense groups. The settings of these parameters can be different from one trace to another based on manual observations (e.g. based on visual inspection). As an optional step after the phase detection, in this section, we offer a general guideline for automatic setting and tuning of these parameters.

Intuitively, dense groups that are well separated from one another show that the phase detection has resulted in well distinct phases. The same criteria are used as quality factors in clustering: low intra-cluster distances (high intra-cluster similarity) and high inter-cluster distances (low inter-cluster similarity) [HKM11]. Many clustering algorithms follow a procedure through which they optimize a global criterion function that measures one or both of the mentioned criteria (e.g., the first step of K-means clustering discussed in Section 4.4 is one of such functions). Similar global criterion functions can be used to find the best partitioning[1] of data among a set of available partitionings obtained choosing different clustering parameters. As shown in Section 4.4.1, given a set of alternative partitionings ($P_1 \cdots P_x$), the criterion function assigns a score to each partitioning and the partitioning with the highest score (or the lowest score depending on the criterion function used) shows the best partitioning and therefore, the better clustering parameters. In this thesis, we discuss two criterion functions: Bayesian Information Criterion and Penalized Sum of Squared root Errors.

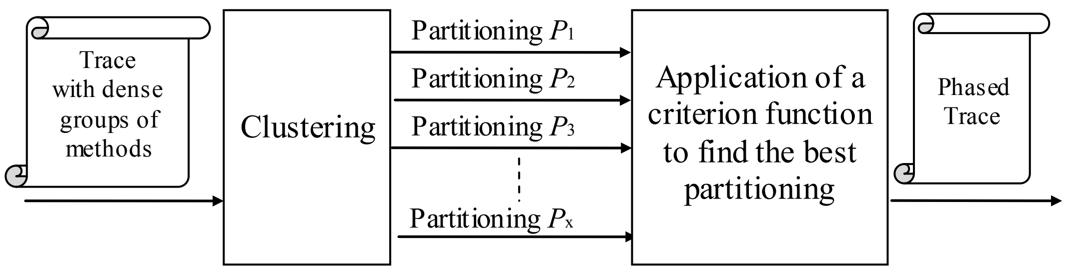

Trace with dense groups of methods
Clustering
Partitioning P1
Partitioning P2
Partitioning P3
Partitioning Px
Application of a criterion function to find the best partitioning
Phased Trace

Figure 4.13. Application of criterion function to find the best partitioning

[1] Recall that a partitioning is the result of clustering

4.5.1. Bayesian Information Criterion

Pelleg et al. [PM00] proposed an approach to find the best partitioning of data where the average variance of the clusters is minimum. It is clear that as the number of clusters increases, the average variance of the clusters decreases (as K approaches the number of data points the variance becomes zero - this is known as overfitting). Therefore, the problem of finding the best partitioning is reduced to finding a trade-off between the number of clusters and the average variance of the clusters that can keep the number of clusters and the variance both minimized. The Bayesian Information Criterion (BIC) [Sch78] can be used to find such a trade-off.

As shown in Section 4.4.1, one can perform the K-means clustering on the rearranged trace and change one or more parameters to obtain a set of alternative partitionings ($P_1 \cdots P_x$). To evaluate these partitionings, we compute the BIC score of each partitioning; the highest BIC means best available partitioning. Since the dimension of the data in our case is 1, we use a special formulation of the BIC (for a more general case of BIC formulation see [PM00]):

$$BIC(P_j) = \hat{l}_j(D) - K_j.\log(R)$$

where D is the set of data points in the input space, $R = |D|$, K_j is the number of clusters in the j-th partitioning, $K_j.\log(R)$ is the penalty, and $\hat{l}_j(D)$ is the log-likelihood of the data according to the j-th partitioning which can be computed as follows (see [11] for more details on using BIC formulation in K-means clustering):

$$\hat{l}_j(D) = \sum_{i=1}^{K_j} -\frac{R_i}{2}\log(2\pi\sigma^2) - \frac{R_i - K_j}{2} + R_i \log(\frac{R_i}{R})$$

where $D_i \subseteq D$ is the set of points that have μ_i as their nearest centroid, $R_i = |D_i|$, and σ^2 is the average variance of the distance from each point to its corresponding centroid. The BIC score provides us with the best partitioning of the repositioned execution trace according to its complexity.

4.5.2. Penalized Sum of Squared Errors

The second criterion function that we use in this thesis is similar to the criterion function in K-means clustering which uses the Euclidean distance to determine which data points (events) should be clustered together. Penalized Sum of Squared Errors (PSSE) determines the overall quality of the partitioning by using the sum-of-squared-errors function (SSE). SSE is the sum of the squares of the distances from the data points in each cluster to the center of that cluster. Thus, SSE is concerned with intra-cluster distances. Therefore, a lower SSE shows a better partitioning. However, a larger number of clusters will always reduce the amount of SSE in the resulting partitioning to the extreme case of zero error if each data point is considered its own cluster. To avoid this problem, in PSSE, SSE is penalized over the number of clusters. In particular, having a set of alternative partitionings ($P_1 \cdots P_x$) a PSSE criterion is defined as follows:

$$PSSE(P_j) = \left(\sum_{n=1}^{K_j} \sum_{x_i \in D_n} \|x_i - \mu_n\|^2 \right) K_j{}^2$$

where K_j is the number of clusters in the *j*-th partitioning, K_j^2 is the penalty, D_n is the set of data points (x_i) of the *n*th cluster that has μ_n as its centroid. The lowest PSSE score indicates the best available partitioning of the repositioned execution trace.

4.5.3. Number of Dense Groups

Using either of the criterion functions discussed earlier we can automatically find the best value among a set of possible values for parameter *K*. Figure 4.14 shows how we can automatically find the number of groups in our repositioned execution trace and locate them. In this figure, first the rearranged trace is passed to K-means Clustering component. Applying K-means clustering with *K*=1 results in partitioning *P*1 that has a single cluster. Similarly, K-means is applied on the rearranged trace with *K*=2, *K*=3, up to *K*=*n* which respectively results in partitionings *P*1, *P*2, to *P*x. These available partitionings are then passed to the component that applies the criterion function. This component finds the best available partitioning and consequently the best estimation of the number of dense groups which, in turn, corresponds to the number of identified phases

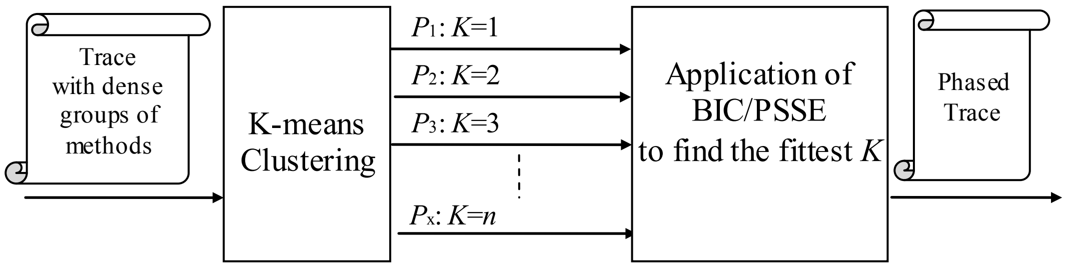

Trace with dense groups of methods
K-means Clustering
P_1: K=1
P_2: K=2
P_3: K=3
P_x: K=n
Application of BIC/PSSE to find the fittest K
Phased Trace

Figure 4.14. Application of criterion function to find *K*

4.5.4. Parameter Repositioning

The repositioning of trace elements as presented earlier simply divides the distance between calls by two. This can be further improved by having the result of the partitioning algorithm guide the tuning of this parameter. As shown in Figure 4.15, one can reposition the original trace having the repositioning parameter set to *M*1. The trace repositioned with *M*1 is then passed to the clustering component that outputs a corresponding partitioning *P*1. Similarly, the rearranged traces using *M*2, *M*3 to *M*x are clustered to corresponding partitionings *P*2, *P*3, to *P*x. These available partitionings are then passed to the component that applies the criterion function. This component finds the best available partitioning and consequently the best estimation for value of the repositioning parameter.

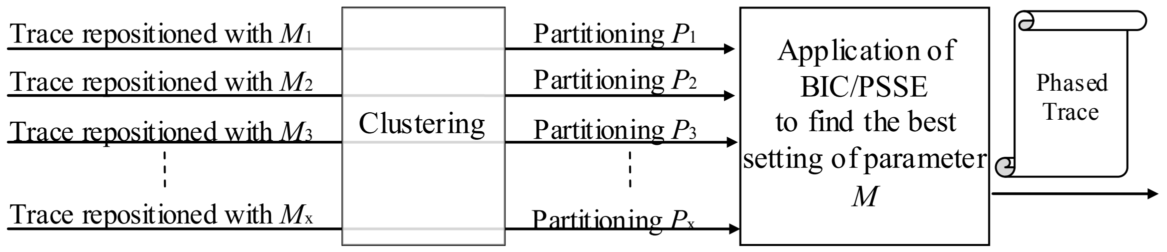

Trace repositioned with M_1
Trace repositioned with M_2
Trace repositioned with M_3
Trace repositioned with M_x
Clustering
Partitioning P_1
Partitioning P_2
Partitioning P_3
Partitioning P_x
Application of BIC/PSSE to find the best setting of parameter M
Phased Trace

Figure 4.15. Find the best value for parameter *M* that impacts repositioning of the trace

4.6. Tool Support

Different tools have been implemented and reused to support the proposed approach. These tools are implemented in Java as part of a tool-suite called Tratex, which is an Eclipse plug-in, developed by members of the Software Behaviour Analysis Research Lab at Concordia

University. Both the integrated scheme and the clustering algorithm are implemented as part of Tratex.

The integrated scheme is implemented as an algorithm that does one pass through the trace. The algorithm first creates a *position table* (a hash table). For each method, the position table keeps the position of the last call to that method (as used in the similarity scheme). The algorithms proceeds by visiting each method call in the original trace, it looks up the position table for that method, if not found, it adds the method and the position of the method call to the table, otherwise, it fetches the previous position of the method call. The algorithm also records the nesting level of the previous method call (as used in continuity scheme). The time complexity of the algorithm that implements the integrated scheme is $O(n)$, where n is the size of the trace.

The K-means and cluster evaluation implemented in Java-ML library [ADS09] are used as the bases of the clustering unit. Modifications are made to the code to support threading[1].

The time complexity of the standard K-means algorithm is $O(icnd)$, where i is the number of iterations, c is the number of clusters, n is the size of the dataset, and d is the dimensionality [DRS08]. In our case, the size of the dataset is equal to the size of the trace, number of iteration is fixed to 50, dimension is 1, and the number of clusters is set from 1 to 14 (in parallel).

[1] In the case where user wants the number of phases to be automatically selected, several clusterings are performed and the one with the best score is selected as the best clustering. Each clustering can be implemented as a thread because the clusterings are independent of one another.

4.7. Case Study

4.7.1. Target Systems and Traces

In this thesis, we evaluate our technique through a number of intrinsic case studies. For this, we perform our evaluations according to the documentations provided by the original developers and maintainers of our target systems. The choice of intrinsic studies constrained us to select our target systems that satisfy two conditions: 1) the systems have to be publicly available for replication purposes and 2) the systems need to be well documented to allow us to verify the results of our study. These conditions led us to choose well-known open source systems. To evaluate the effectiveness of our phase detection approach, we conducted two case studies where we applied our technique to execution traces generated from two different systems that satisfy the mentioned conditions.

The first execution trace is generated from JHotDraw 5.2 [JHO]. JHotDraw is a framework implemented in Java for technical and structured graphics. It consists of 11 packages, 171 classes, and 1414 methods. JHotDraw 5.2 has 9419 lines of code.

The second case study was conducted on an execution trace generated from ArgoUML 0.27 [ARG]. ArgoUML is an open source UML modeling tool implemented in Java. It consists of 1853 classes, 10214 methods, and 130995 lines of code.

To deal with multiple threads of execution, we simply treat each thread as a separate trace. During data collection, the system is instrumented in such a way that the nesting levels of method calls in each thread are independent of nesting levels of method calls in other threads. For example, in the flow of execution of a system we might see a method call with nesting level 4 followed by another method call with nesting level 7 from a different thread (while in a single thread the nesting level from one method call to the next cannot increase more than 1 level).

4.7.2. JHotDraw

Scenario Description

For our first case study, we used an execution trace generated from JHotDraw by exercising a scenario that involves the following actions:

Drawing three different figures (a rectangle, a round-rectangle, and an ellipse) followed by drawing the same three figures for the second time on the same sheet and closing the application.

Since JHotDraw registers all mouse movements, and mouse movements are required while drawing figures, the resulting trace was bound to contain a lot of noise. We have therefore filtered these mouse movements to obtain a trace that is cleaner. We are aware that the detection of noise in a trace might not always be straightforward and that noise detection techniques such as the ones presented by Hamou-Lhadj et al. in [17] might be needed. The resulting trace contained 4197 method invocations (8394 events considering entry and exit), which is considered a small trace. It is used here as a proof of concept. We show how our approach works on a larger trace in the second case study.

Results

We first applied our approach to detect the major phases in the trace. This is achieved by setting the threshold to the size of the trace. Figure 4.16 shows the result of applying the integrated gravity scheme on the trace. The result is shown in the form of a histogram, where the x-axis shows the distance between the positions of the calls and the y-axis represents the number of methods whose position falls into one interval of x-axis. As we can see in Figure 4.16, there are two dense groups of method calls (DG1 and DG2) that have been formed and which indicate the possibility of the existence of two major execution phases.

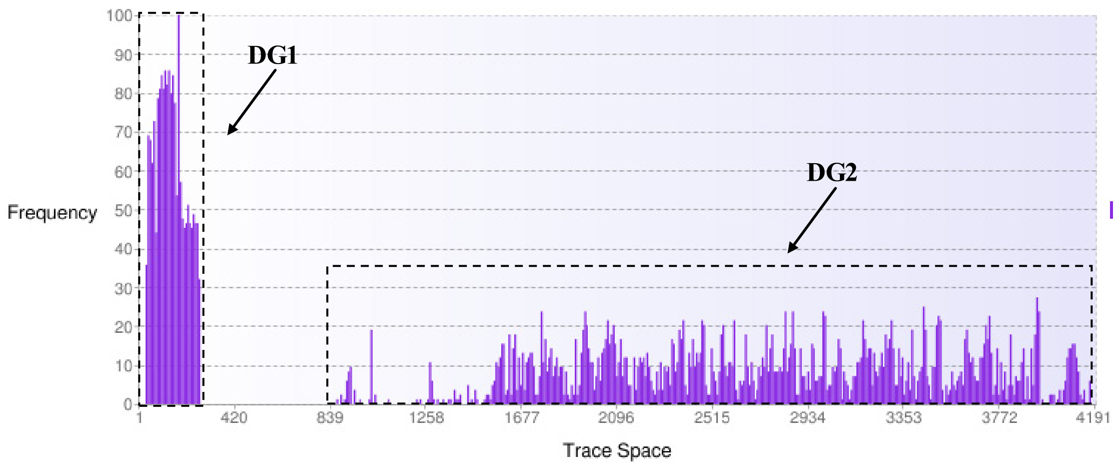


Figure 4.16. The result of applying the integrated scheme

These dense groups are then mapped back to the original trace as two execution phases. Table 4.1 shows the specification of the execution phases corresponding to dense groups. We explored the contents of the two phases and found that Phase1 represents the initialization of variables (about 1500 invocations), whereas Phase2 contains the methods invoked in the trace to perform the core computations (i.e. drawing the figures).

Table 4.1. Detected major phases

Phases	**Size**	**Location**
Phase 1	1548	1-1548
Phase 2	2649	1549-4197

A corresponding phase flow that shows the major phases of the traced scenario is shown in Figure 4.16.

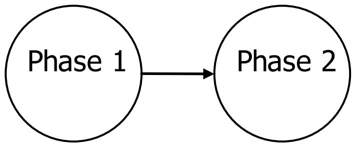


Figure 4.17. Major phases in the traced scenario

We applied our technique to Phase2 with a lower threshold so as to detect the sub-phases that it composes. As for the threshold, we set it to the longest sequence of method call with no repetition that is possible in the second phase. For this, we set *t* equal to the number of unique method calls in Phase2. Figure 4.18 shows the results of applying the integrated scheme, using *t*=125, to the Phase2 of the JHotDraw trace. A number of dense groups can be seen in this figure.

As an optional step in our technique, we can automatically determine the number of dense groups formed on the rearranged trace (this process is explained in Section 4.5.3). For this, the trace resulting from applying the integrated gravity is partitioned by K-means clustering for *K* from 1 to 10. The BIC score for different partitionings of DG2 are shown in Figure 4.19. The highest BIC score was for the partitioning with *K* = 7 dense groups as the best fit. Except for the last of these seven dense groups (DG7), the six phases (DG1 to DG6) are similar in terms of length and density. The application of our integrated scheme took 0.412 seconds and the clustering and mapping (including the optional step of parameter tuning)[1] took 7.06 seconds on an Intel Core i5 CPU 2.30GHz, 4.00 GB main memory, running Windows 7.

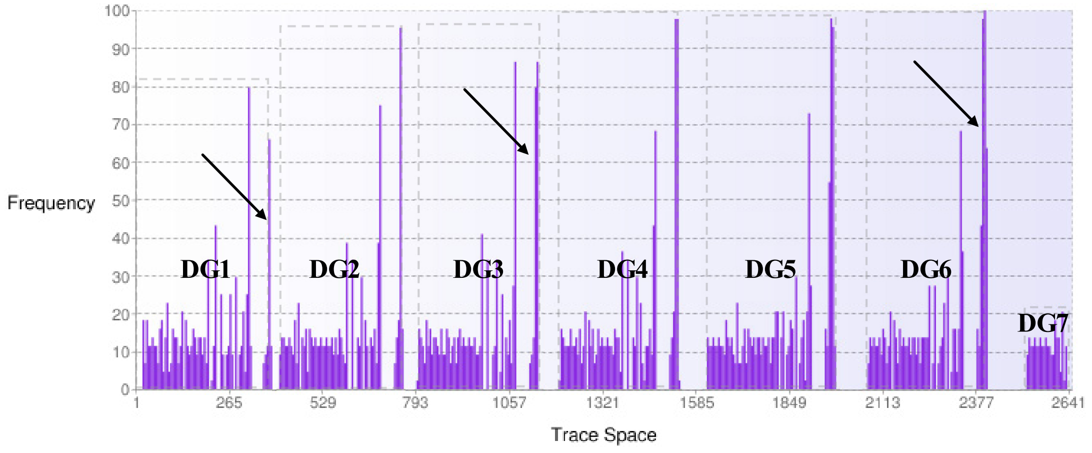


Figure 4.18. The result of applying the integrated scheme on Phase2

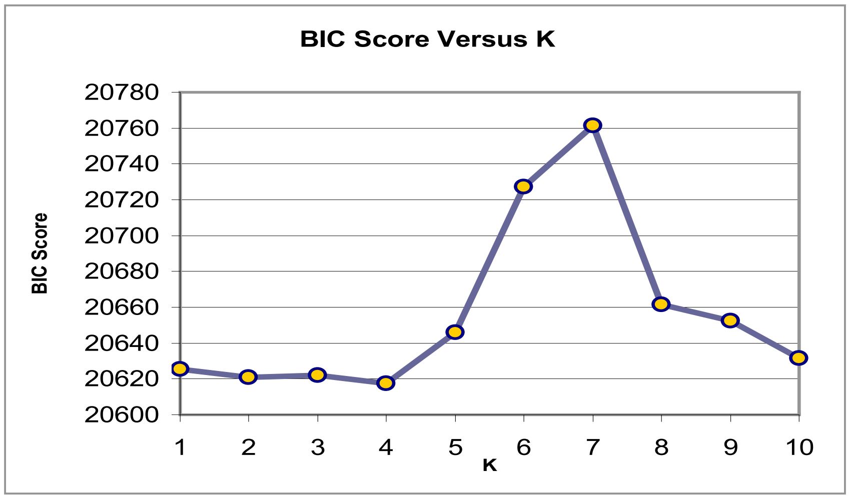


Figure 4.19. The BIC score for different partitionings of DG2, the partitioning with 7 clusters is the fittest

[1] Performing the clustering step without parameter tuning (setting *K* to the number of dense group that visually can be perceived) took 1.488 seconds.

Table 4.1 shows the specification of the execution phases corresponding to dense groups of Phase2 (i.e. drawing the figures). A corresponding phase flow that shows the sub-phases of Phase2 is shown in Figure 4.16. We validated the results by referring to JHotDraw documentation and by manually analysing the methods invoked in each phase.

Table 4.2. Detected sub-phases of Phase2

Sub-phases of Phase2	Size	Location
Phase2.1	374	1549-1922
Phase2.2	385	1923-2307
Phase2.3	403	2308-2710
Phase2.4	427	2711-3137
Phase2.5	461	3138-3598
Phase2.6	487	3599-4085
Phase2.7	112	4086-4197

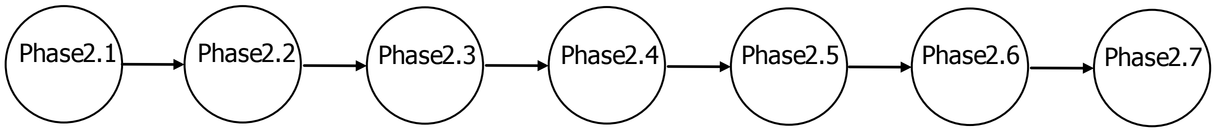


Figure 4.20. Major phases in the traced scenario

After exploring the content of the trace, we found that Phase 2.7 contains methods that end the application (finalization methods) including the following methods:

```
contrib.MDI_DrawApplication.internalFrameClosing
contrib.MDI_DrawApplication.internalFrameDeactivated
contrib.MDI_DrawApplication.internalFrameClosed
application.DrawApplication.actionPerformed
application.DrawApplication.exit
samples.javadraw.JavaDrawApp.destroy
application.DrawApplication.destroy
samples.javadraw.JavaDrawApp.endAnimation
```

As for the phases Phase 2.1 to Phase 2.6, we found that each of these phases corresponded to the drawing of a figure. For example, the phase Phase 2.1 contained methods involved in drawing a rectangle. Phase 2.2 contained the methods responsible of drawing a circle, etc.

To further determine the sub-phases that compose Phase 2.1, we re-applied the integrated gravity scheme on this phase with a lower threshold. This resulted in three sub-phases which contained methods for "selecting the rectangle button in the buttons menu", "preparation for creating and adding a rectangle to the sheet", and "drawing of a rectangle on the sheet". An example of methods involved in the third sub-phase of Phase 2.1 is:

```
standard.DecoratorFigure.draw
figures.AttributeFigure.draw
figures.AttributeFigure.getFillColor
figures.AttributeFigure.getAttribute
figures.AttributeFigure.getDefaultAttribute
figures.FigureAttributes.get
util.ColorMap.isTransparent
util.ColorMap.color
figures.RectangleFigure.drawBackground
figures.RectangleFigure.displayBox
```

We applied the same process to Phase2.2 to Phase2.6 and were able to confirm that each phase corresponded to the drawing to one of the figures and that all of them consisted of three other sub-phases. One interesting observation was that the length and the density of part of the phase which contains the methods that actually draw the figure on the sheet (indicated by blue arrows) grows from Phase 2.1 to Phase 2.6 while the first parts of all phases (Phase 2.1-Phase 2.6) exhibit similar distribution. This is due to the massive use of design patterns in JHotDraw where some features just differ for the invocation of a few methods. That is, drawing a circle is performed very similarly to drawing a rectangle. This explains the phase parts that are similar. It also justifies the fact that these phases formed a single major phase (Phase 2) at a higher level of granularity. The reason for the growing part (indicated by blue arrows) is that every time we draw a figure, JHotDraw redraws the existing figures on the sheet.

4.7.3. ArgoUML

Scenario Description

For the second case study, we applied our technique to a trace generated from ArgoUML by exercising the following scenario: Starting up ArgoUML, drawing a class on the class diagram, and quitting ArgoUML. The resulting trace contained 35753 method calls (to 2331 different methods). Note that a method invocation requires at least two events to be collected, the entry and exit of a method. The trace size in terms of events is therefore about 71506 events, which is considered a relatively large trace.

Results

Figure 4.21 shows the result of applying the integrated gravity scheme on the ArgoUML trace. A clear division of the execution trace into five major phases can be seen in this figure. Figure 4.22 shows PSSE scores of partitioning with different numbers of groups (see Section 4.5.2 for a discussion on how PSSE works). A lower PSSE suggests a better partitioning. Thus, $K = 5$ as the number of dense groups is selected as the best fit. The application of our integrated scheme took 1.168 seconds and the clustering and mapping (including the optional step of parameter tuning)[1] took 24.32 seconds on an Intel Core i5 CPU 2.30GHz, 4.00 GB main memory, running Windows 7.

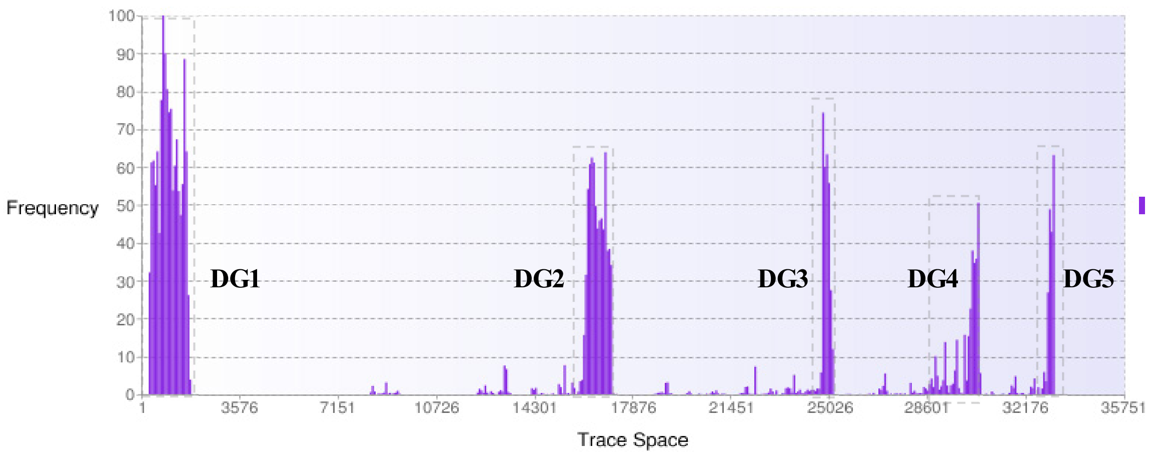


Figure 4.21. Dense groups formed on the rearranged trace of ArgoUML

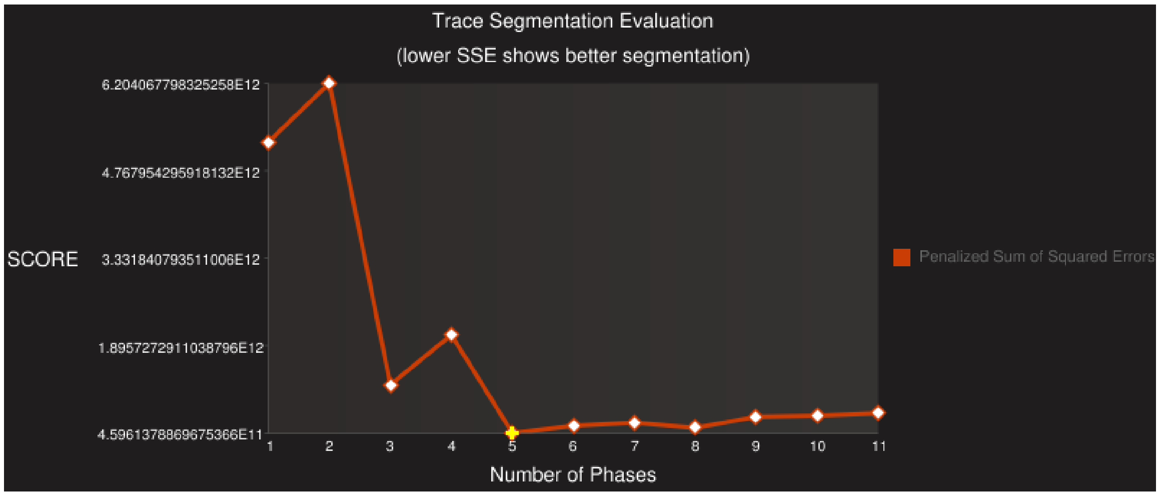


Figure 4.22. PSSE scores for different partitionings

[1] Performing the clustering step without parameter tuning (setting *K* to the number of dense group that visually can be perceived) took 3.523 seconds.

Five clusters are then mapped back to the original trace as five execution phases. Table 4.4 shows the specification of these execution phases. The first phase contains about 16000 method calls. When checked against the documentation, as expected, the methods of the first phase indicate the initialization of ArgoUML where the main application frame and project are set up. The project corresponds to a model that contains an empty class diagram, and an empty use case diagram. The second detected phase is concerned with loading auxiliary modules from the input stream.

Table 4.3. Specifications of major phases

Phases	**Size**	**Location**
Phase1	16036	1-16036
Phase2	8766	16037-24802
Phase3	4221	24803-29023
Phase4	4095	29024-33118
Phase5	2635	33119-35753

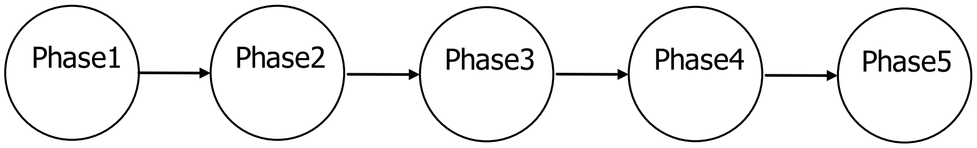


Figure 4.23. Major phases of the traced scenario

The third phase is the phase where the actual class element is drawn. This phase is followed with two other small phases. The first of these phases (i.e., Phase 4) refreshes and updates the models and the last phase (Phase 5) terminates the application. An example of the methods involved in the last phase is:

```
org.argouml.ui.cmd.ActionNotation.menuSelected
org.argouml.kernel.ProjectSettings.getNotationName

org.argouml.notation.NotationNameImpl.getIcon

org.argouml.notation.NotationNameImpl.sameNotatioAs

org.argouml.ui.cmd.ActionNotation.menuDeselected

org.argouml.ui.cmd.ActionExit.actionPerformed

org.argouml.ui.ProjectBrowser.tryExit

org.argouml.ui.ProjectBrowser.saveScreenConfiguration

org.argouml.configuration.Configuration.save
```

We further applied our technique, with a lower threshold to Phase 3 (drawing a class) to understand how this is accomplished.

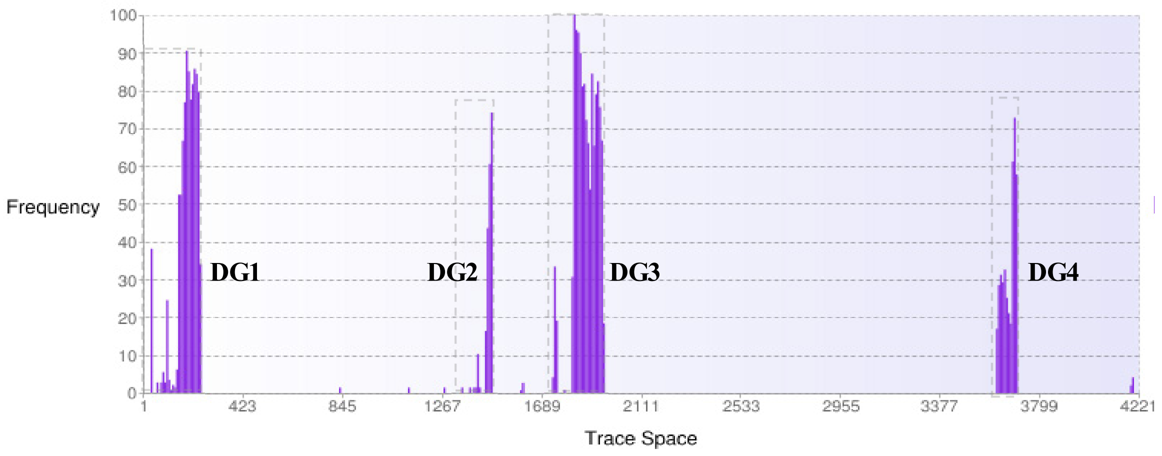


Figure 4.24. The result of applying integrated scheme on Phase3

Figure 4.24 shows the result of applying the integrated gravity, using *t*=20, to Phase 3. Four dense groups can be seen on the rearranged trace. PSSE scores of different partitionings also show that the partitioning with 4 clusters (*K* = 4) is the best fit. The PSSE scores are shown in Figure 4.25 (lowest score shows the best fit). Table 4.4 show the specifications of execution phases corresponding to the formed dense groups. These phases that compose Phase 3 of the traced scenario are shown in Figure 4.26.

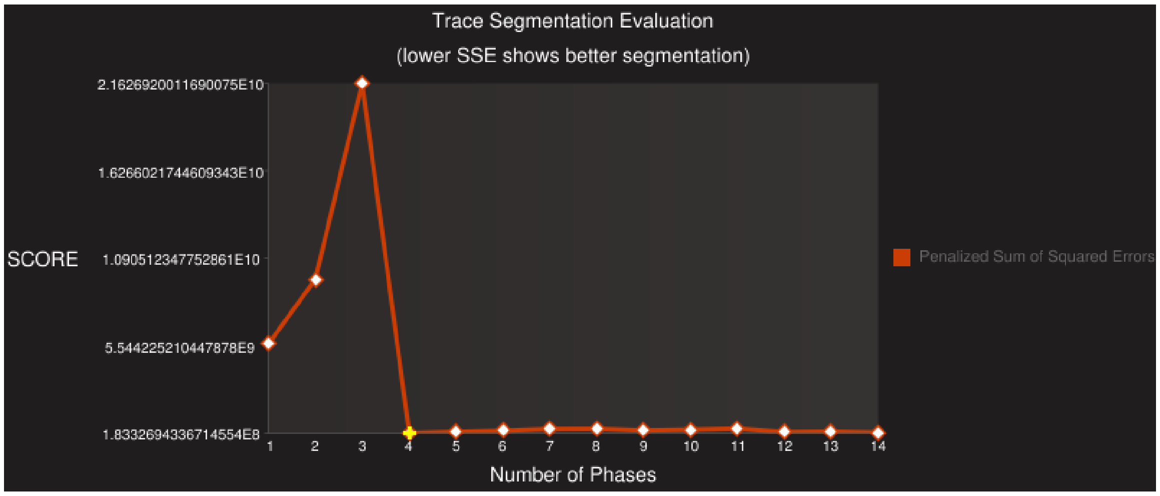


Figure 4.25. PSSE scores of different partitionings

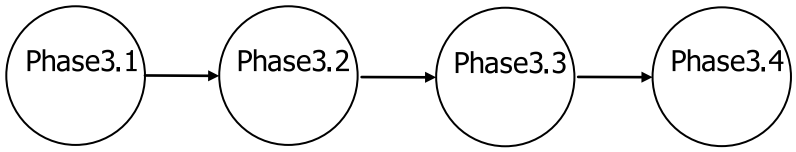


Figure 4.26. Sub-phases of phase3

Table 4.4. Specifications of the sub-phases of Phase 3

Phases	**Size**	**Location**
Phase1	1406	24803-26208
Phase2	320	26209-26528
Phase3	1905	26529-28433
Phase4	590	28434-29023

The important tasks that are performed when drawing a class organized based on the sub-phases that were identified are summarized as high-level descriptions in Table 4.5. To describe these tasks as shown in Table 4.5, we referred to ArgoUML source code. We validated these tasks using ArgoUML documentation and the Cookbook for Developers of ArgoUML [25].

Table 4.5. Summary of the task performed in Phase 3

- Phase 3: Adding a class:
  - o Sub-phase 3.1:
    - ⇒ Command to create nodes with the appropriate `modelelement`: Delegate creation of the node to the uml model subsystem and return an object which represents a UML class diagram.
    - ⇒ Define a renderer object for UML Class Diagrams: Return a Fig that can be used to represent the given node.
  - o Sub-phase 3.2:
    - ⇒ Prepare the box coordinates to display graphics for a UML Class in a diagram.
    - ⇒ Determine whether the `graphmodel` will allow adding the node (Define a bridge between the UML meta-model representation of the design and the `GraphModel` interface used by GEF).
    - ⇒ Determine if the given object is present as a node in the graph
    - ⇒ Final call at creation time of the Fig, i.e. here the node icon is put on a Diagram: where the displayed diagram icons for UML `ModelElements` looks like nodes and has editable names and can be resized.
    - ⇒ Add the given node to the graph, if of the correct type.
  - o Sub-phase 3.3:
    - ⇒ Give continuous feedback to aid in the making of good design decisions: Perform critiques about well-formedness of the model.
    - ⇒ Change the mode of multieditorpane (particularly the `TabDiagrams`) to deselect all tools in the toolbar (Unselect all the toolbar class button).
  - o Sub-phase 3.4:
    - ⇒ Hit the class (prepare selection of the class diagram). Necessary since GEF contains some errors regarding the hit subject.
    - ⇒ Compute handle selection, if any, from cursor location.
    - ⇒ Prepare selection of the current element (through an extension package for swing classes. This package provides ArgoUML independent swing extensions.)

4.8. Threats to Validity

Although our approach performed well when applied to the trace in our case study, there are several aspects that can impact its effectiveness.

First, the phase detection algorithm relies on method calls names to assess the similarity between the method calls and decide whether to bring them together or not when applying the similarity scheme. Method names, however, might not be sufficient. For example, some methods might be overloaded and we should not assume that they all perform computations related to the same phase. Also, one can use patterns of calls instead of mere method names. Future work should focus on investigating better similarity criteria and metrics between the trace events.

During the first case study, we had to remove some mouse movement events because they cluttered the trace. However, we did not attempt to remove all low-level utilities, an activity which might be needed when we generalize our approach and apply it to other systems. In general, we need to study the impact of removing utilities on the final sample before the phase detection algorithm is applied.

Varying clustering algorithms might also have an impact on the phase detection method, which forms the basis for sampling. It is therefore important to study how various clustering algorithms can be used.

4.9. Summary

In this chapter, we presented the trace segmentation process that consists of dividing a trace in execution phases. We defined an execution as something that characterizes program computations that are invoked in a trace.

We also proposed an implementation for the trace segmentation component. We created what we call gravitational schemes that when applied on an execution trace result in formation of dense groups of events. Each of these dense groups can indicate the presence an execution phase on the original trace.

The two schemes that we proposed are inspired by the Gestalt laws of similarity and good continuation. These schemes are also aligned with the fact that a phase change in an execution trace corresponds to a significant change in the pattern of attributes of the events in a trace over time. We also created an integrated scheme that combines the similarity and the continuity schemes.

We used K-mean clustering to automatically find the beginning and the ending of each dense group. Our approach has a number of parameters that can be set by the user if necessary. As an optional step to our phase detection technique, we provided an automatic way to tune these parameters by defining the criterion function. Finally, we performed a number of case studies on open source systems to evaluate the effectiveness of our phase detection approach. For the evaluation, we compared the results with available system documentation.

Chapter 5. Content Prioritization

5.1. Introduction

In this chapter, we propose a method for automatically identifying the trace events that are most relevant to the implementation of each execution phase. This is particularly important since it can significantly simplify the exploration of large traces by allowing software engineers to quickly understand the phases of an execution and select the intended ones before deciding to dive into the details.

We also propose a technique for identifying similar phases within a trace. It is possible that a single major computation happens several times in different periods during the execution of a system. Existing phase detection algorithms usually detect each occurrence as a different phase (although they are very similar). A better representation of a system's execution is the one where a phase is indentified only once and referred to it in other places.

To achieve our objectives, we adopt techniques from the area of text mining, more particularly the Term Frequency, Inverse Document Frequency (TF-IDF) technique and the cosine similarity measure. The contributions of this chapter can be further refined to help with tasks such as:

- The ability for software engineers to understand the most relevant events that implement the traced scenario (or software feature). As such, the contribution of this paper falls under the category of feature location research.

- The recovery of high-level behavioural design models from large execution traces. The most important events of a trace uncovered by our approach can be represented, for example, in a UML sequence diagram. These models can in turn be used for redocumentation, or for assessing if the system does what it is supposed to do by comparing the resulting diagrams with existing design diagrams.

- Any area where trace summaries are needed. This will be particularly useful if the proposed techniques are integrated in a tool, in which the ability to switch between a high-level view of a trace and a detailed view is provided.

This chapter is organized as follows: in Section 5.2, we discuss a number of works that bear some similarities with the approach proposed in this chapter. In Section 5.3, we draw a parallel between trace analysis and text mining. In Section 5.4, we introduce our proposed content prioritization process and its related steps. In Section 5.5, we present the case studies that evaluate our proposed approach. Finally, in Section 5.7, we conclude this chapter with a summary.

Parts of the material in this chapter is adapted and expanded from a paper published in the 27th IEEE International Conference on Software Maintenance (ICSM), 2011 [PHS11].

5.2. Related Work

The following studies bear similarities with our work in the sense that they also try to indentify events of a trace that are important to the users' tasks. However, no previous study has been done on extracting relevant trace events based on the execution phases.

Software Reconnaissance [WS95], introduced in 1995, is one of the best-known techniques that fully relies on dynamic analysis to locate source code components that implement a specific feature. Software Reconnaissance starts by generating multiple execution traces by exercising several features of the system in a way that one execution trace exercises the desired feature and the others do not. The generated traces are then compared for overlap removal. Roughly said, if the set of code components invoked in the execution traces not exercising the desired feature is subtracted from the set of such components in the feature specific trace, the result contains components of the system relevant to the feature of interest. Although the ultimate goal is to only identify the components of a single feature, Software Reconnaissance requires the exercising of several features of the system. Moreover, the number of features that must be considered for the approach to be effective is unclear. Software Reconnaissance has been enhanced by including three measurements used to identify the extent to which a particular component belongs to a feature [WGH00]. As another enhancement to this approach [AG05], the traces can be filtered for unwanted events (e.g., mouse motion) before the comparison phase.

The idea of ranking how likely each code component belongs to a given feature based on pure dynamic analysis was also proposed as an approach to feature location in [ED05]. This approach argues that a code component executed several times in the execution of a feature under different situations (i.e., normal and exceptional scenarios) should be regarded as an important component, whereas a component that occurs in traces of several features should be considered as a utility component and should be ranked lower in comparison with other components.

A hybrid approach that combines dynamic and static analysis techniques to feature location has also been proposed [EKS03]. This approach uses dynamic analysis to gather traces that correspond to software features of the system and adds static code dependency information to the content of traces to build a concept lattice that maps features to code components. One of the shortcomings of this approach is that overlapping components (i.e., the ones that implement several features) can appear in the concept lattice. To overcome this issue, users are required to navigate through the concept lattice and identify manually the components specific to each feature. This process requires a considerable effort from the users and a good understanding of the source code as well as the domain of the system. A similar approach that combines static and dynamic analysis was proposed by Antoniol et al. [AG05] to locate and compare different features in multi-threaded object-oriented C++ programs. Smit et al. [SSW08] proposed an approach for identifying usage scenarios from GUI event traces. Poshyvanyk et al. [PGM+07] introduced an approach based on information retrieval (IR) for feature location. Asadi et al. [ADAG10, AAG10] also proposed an interesting approach that uses IR to identify concepts in execution traces. In their work, for each method, they first extract terms from source code of that method. Then they perform a multiple-step pre-processing (tokenizing, stopword removal, stemming) on the terms of each method. This is followed by applying a weighting scheme that assigns a weight to each term of each method. The weighting scheme works based on the appearance of terms across methods: the terms that are shared between more methods are less representative of that method and vice versa. Then, they segment the trace several times according to an optimization function. The near optimal segmentation is output. As the authors mention their approach suffers from problems in scalability (in both time and space) as well as in the possibility to handle longer traces. Furthermore, their approach may produce a different concept assignment on each run.

In [RHR08], Rohatgi et al. proposed another feature location approach based on impact analysis: measuring the impact of a modification made to a code component on the rest of the system. The approach uses dynamic analysis to generate a trace that corresponds to the feature under study and applies static analysis to rank the components invoked in the generated trace according to their relevance with respect to the executed feature. The ranking mechanism guides software engineers in locating feature-specific components without the need for prior knowledge of the system. This approach operates on only one trace that corresponds to the feature under study and it facilitates the automatic identification of feature-specific components.

5.3. Trace Analysis and Text Mining

Dealing with large data spaces, whether the data takes the form of traces, text, or any other artefact, is in principle subject to similar challenges.

Research in the area of text mining has long been active in addressing the challenges related to document size. Much of the work has been devoted to extracting summaries and relevant information from large document corpuses, which motivated us to explore how the application of existing techniques can be applied to the analysis of traces.

However, before drawing any parallel between the two domains, we first need to determine the mapping between the concepts in the two domains as stated by Gentner in his theory of structural mapping [Gen83]. Figure 5.1 shows three types of objects in the domain of text mining: Corpus, Document, and Term, where a document is a sequence of terms and a corpus is a set of documents. We propose to view an execution trace as a corpus and the execution phases that compose it as the corpus documents. In Chapter 4, we proposed a way to automatically detect execution phases from a trace. Each trace event can be viewed as a term within a phase document. Note that the mapping takes place not only between objects, but also between the containment relations between the objects.

There exists a variety of text mining techniques; here we focus on the ones that make use of three types of information:

- Local information: It refers to the information inside individual documents (e.g., term frequency).
- Global information: It is the information from the collection of documents in the corpus (e.g., document frequency).
- Domain information that refers to the information from the domain of a term. For example, the term "can" has different significance in the domain of literature and the domain of packaging [Pri10].

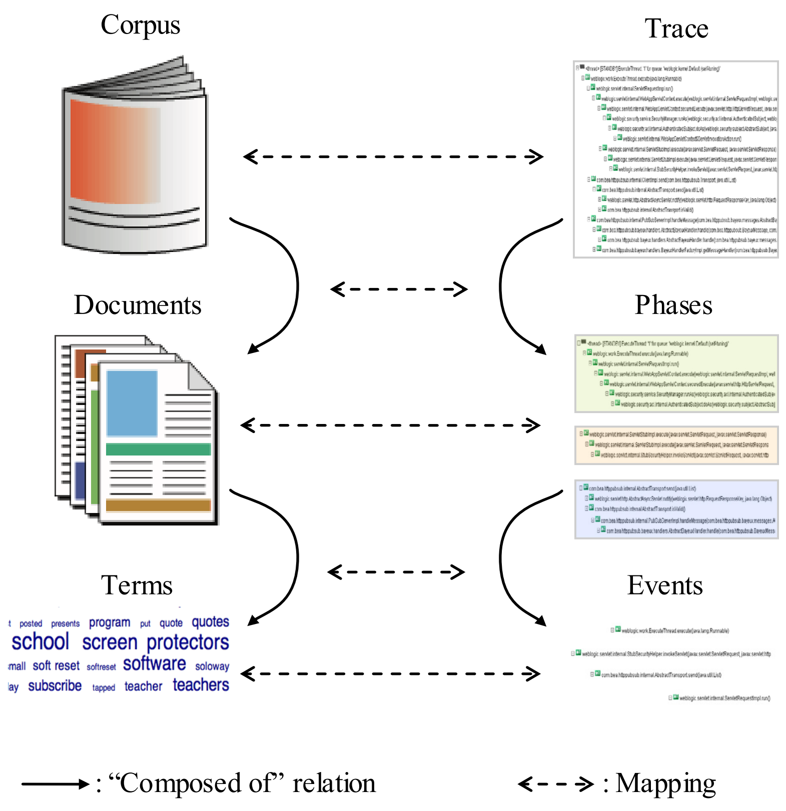

Corpus
Trace
Documents
Phases
Terms
Events
posted presents program put quote quotes
school screen protectors
soft reset softreset software soloway
subscribe tapped teacher teachers
: "Composed of" relation
: Mapping

Figure 5.1. Mapping between domains of text mining and trace analysis

Similarly, we consider the information inside each phase (e.g., the frequency of an event within a phase) as local information in trace analysis. Global information in this domain can be considered as the information from the collection of phases in the trace (e.g., number of phases in which a particular event is invoked). Finally, domain information in trace analysis is the information about the trace events where they are defined. This information could be extracted from the source code or the documentation (e.g., static call graph information of each element).

Figure 5.2 shows our approach for extracting relevant information from a trace based on the analysis of its execution phases. The approach encompasses two main phases. The first phase (Trace Segmentation) consists of automatically dividing the content of a trace into execution phases. To segment a trace, we use the phase detection technique that we as presented in Chapter 4. Other phase detection techniques such as the ones presented in [Reiss07, WII08, PAH10] could also be used.

The second phase consists of the application of a newly designed technique called content prioritization with which the trace events of each phase are weighted and the ones that have the highest weight are deemed to be the most representative events of a phase. Once the events are weighted, we extract the most representative ones. We also use the weighted events to detect similar phases.

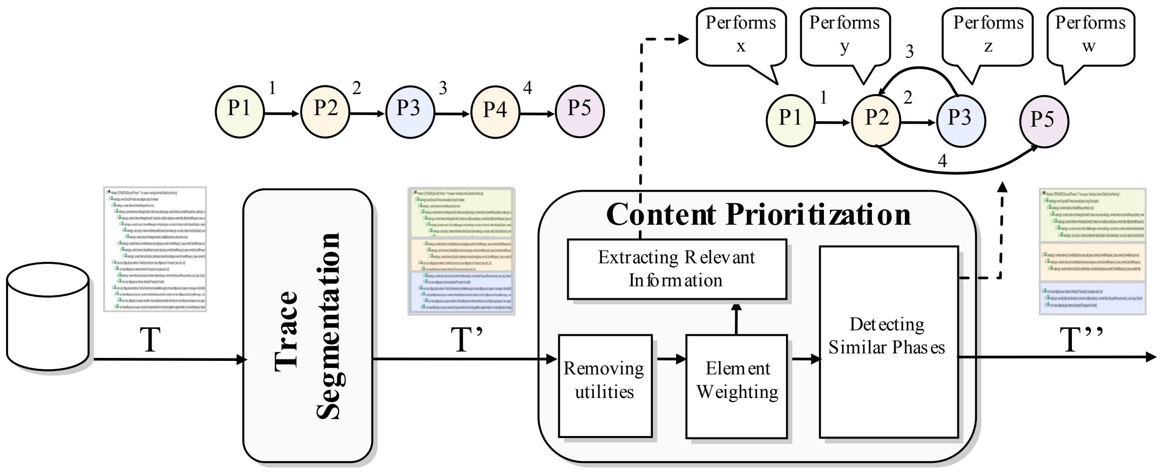


Figure 5.2. Overview of our proposed approach

5.4. Content Prioritization

Once the phases of a trace are found, the phased trace is passed to the content prioritization component to extract the trace events that are most relevant to each execution phase. For this purpose, we use text mining techniques as previously mentioned. More precisely, the content prioritization phase is composed of the following steps, which are listed here and discussed in more detail in the subsequent sections:

1. We first remove utility methods from the execution phases to reduce the noise in the data. The process of removing utilities is similar to removing stop words from text.

2. We apply a weighting function to weigh events of a phase according to their relevance. The higher the weight, the more representative the event.

3. We propose a way to select the most representative events in a phase from the list of ranked events obtained in 2.

4. We measure the similarity between phases based on their weighted events.

5.4.1. Utility Removal

Text mining techniques usually start with a pre-processing step that removes stop words - The words that add little value to the process of finding relevant information. Stop word identification, which is the process of identifying these words, makes use of domain and global information. For example, in the domain of English literature, stop words are among auxiliary verbs (e.g., have, be), pronouns (he, it), or prepositions (to, for). Similarly, we proceed with removing utilities from the trace before weighting its events. According to Hamou-Lhadj et al. [HL06], a utility is a component that implements a low-level concept such as accessing methods or language libraries. We limit ourselves to this type of utilities that can be detected without advanced processing techniques. This type of filtering falls in filtering a trace based on programming conventions as discussed in 2.4.3.

5.4.2. Event Weighting

In text mining, the process of term weighting is used for finding representative terms in each documents of a corpus. One of the best-known weighting schemes is called TF-IDF (Term Frequency, Inverse Document Frequency) [Joa98]. The goal of TF-IDF term weighting is to obtain high weights for terms that are representative of a document's content and lower weights for terms that are less representative. The weight of a term depends both on how often it appears in the given document (term frequency, or *tf*) and on how often it appears in all the documents of the collection (document frequency, or *df*). In general, a high frequency of a term (high *tf*) in one document shows the importance of that term while if a term is scattered between different documents (high *df*), then it is considered less important. Therefore, if a term has high *tf* and low *df* (or high *idf* -inverse document frequency) then it will have a higher weight.

A similar idea can be adapted to trace analysis to weigh the events of a trace. We suggest a weighting function that considers the frequency of trace events across the execution phases. Our hypothesis is that a trace event that appears often in a particular phase, but appears relatively infrequently in other phases potentially indicates that it is doing something important in that particular phase. We use the trace *T* shown in Figure 5.3, where the execution phases are already identified, to illustrate the proposed weighting function.

Our weighting function is composed of three factors: local, global, and normalization. The weight of an event *i* in a phase *k* has the following general form:

$$w_{i,k} = L_{i,k} G_i N_k$$

where $L_{i,k}$ is the local weight of the event *i* in phase *k* (local information) which is usually based on the number of occurrences of the event in the phase. G_i is the global weight of the event in the phases of the trace (global information). This factor tends to under-weigh the events that are too common in the trace. N_k is the normalization factor for the event weights in phase *k*.

We create an index vector called event vector for each phase. The magnitude of each event in this vector indicates how well that event represents the content of the phase based on its frequency within the phase and other phases. If an event occurs very frequently in some phases, but occurs rarely in the trace as a whole, it will be given high weight in the event vector.

The event frequency $ef_{i,k}$ of event *i* in phase *k* is defined as the number of times that *i* occurs in *k*. Similar to approaches in text mining, and since the importance of an event does not increase proportionally with the event's frequency, we use the following local weighting $L_{i,k}$ for event frequency.

$$L_{i,k} = \begin{cases} 1 + \log_{10} ef_{i,k} & \text{if } ef_{i,k} > 0 \\ 0 & \text{otherwise} \end{cases}$$

In Figure 5.3, $L_{i,k}$ is calculated for the events in each of the 3 phases of trace T. For instance, in Phase 1, event d is invoked twice, therefore, its local weight (noted as *L*(d) in Figure 5.3) is calculated as $L_{\mathrm{d},1} = 1 + \log(2) = 1.3$.

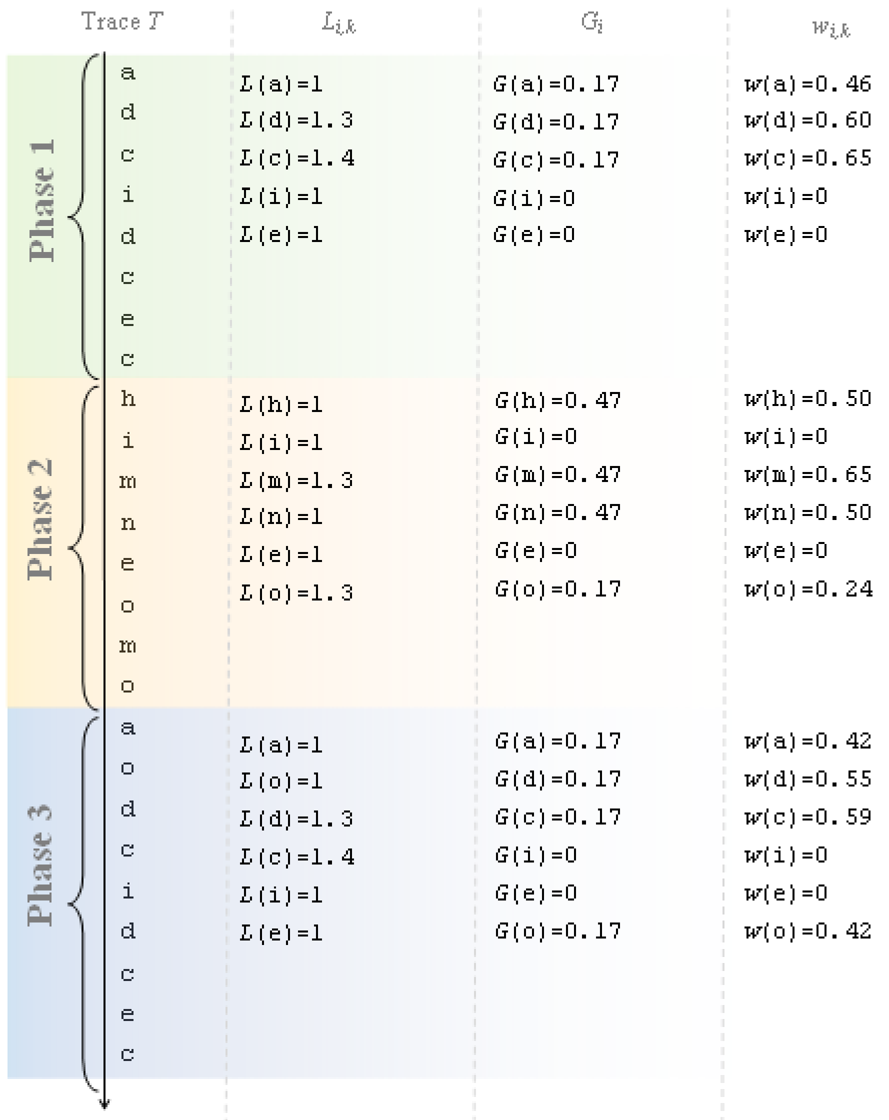
Following the general form, we want to assign the weight $w_{i,k}$ to event *i* in phase *k* in proportion to the frequency of occurrence of the event in phase *k*, and in inverse proportion to the number of phases in which the event is invoked. It should be noted that the phase lengths, and hence the number of non-zero event weights assigned to a phase, varies widely. To allow a meaningful final retrieval similarity, it is convenient to use a length normalization factor as part of the event weighting formula. A high-quality event weighting formula for $w_{i,k}$, the weight of event *i* in phase *k* is

$$w_{i,k} = \underbrace{\frac{\overbrace{(\log(ef_{i,k})+1)}^{L_{i,k}} * \overbrace{\log(N/n_i)}^{G_i}}{\sqrt{\sum_{j=1}^{e}\left[\left(\log\left(ef_{j,k}\right)+1\right)*\log\left(N/n_j\right)\right]^2}}}_{\frac{1}{N_k}}$$

where $ef_{i,k}$ is the occurrence frequency of event *i* in phase *k*, *N* is the total number of phases, n_i is the number of phases with event *i* assigned and *e* is the total number of events. The factor $\log(N/n_i)$ is an inverse phase frequency (similar to "*idf*") factor that decreases as the events are used widely in a trace and the denominator in the equation is used for weight normalization. This factor is used to adjust the event vector of the phase to its norm, so all the phases have the same modulus and can be compared no matter the size of the phase.

This weighting system enables us to adjust the weighting for an event according to not only local but also global information available in the entire trace. Figure 5.3 shows $w_{i,k}$ for the events in each phase. The weight of the events that do not appear in a phase is zero (events that are not shown in the vector of a phase are also of weight zero). For instance, $w_{\mathrm{d},1}$ the weight of event d in Phase 1, given that $G(\mathrm{d})$ in the trace is 0.17, is calculated as follows:

$$w_{\mathrm{d},1} = \frac{1.3 * 0.17}{\sqrt{(0.17)^2 + (0.22)^2 + (0.24)^2}} = 0.60$$



	Trace T	$L_{i,k}$	G_i	$w_{i,k}$
Phase 1	a	L(a)=1	G(a)=0.17	w(a)=0.46
	d	L(d)=1.3	G(d)=0.17	w(d)=0.60
	c	L(c)=1.4	G(c)=0.17	w(c)=0.65
	i	L(i)=1	G(i)=0	w(i)=0
	d	L(e)=1	G(e)=0	w(e)=0
	c			
	e			
	c			
Phase 2	h	L(h)=1	G(h)=0.47	w(h)=0.50
	i	L(i)=1	G(i)=0	w(i)=0
	m	L(m)=1.3	G(m)=0.47	w(m)=0.65
	n	L(n)=1	G(n)=0.47	w(n)=0.50
	e	L(e)=1	G(e)=0	w(e)=0
	o	L(o)=1.3	G(o)=0.17	w(o)=0.24
	m			
	o			
Phase 3	a	L(a)=1	G(a)=0.17	w(a)=0.42
	o	L(o)=1	G(d)=0.17	w(d)=0.55
	d	L(d)=1.3	G(c)=0.17	w(c)=0.59
	c	L(c)=1.4	G(i)=0	w(i)=0
	i	L(i)=1	G(e)=0	w(e)=0
	d	L(e)=1	G(o)=0.17	w(o)=0.42
	c			
	e			
	c			

Figure 5.3 Running example to illustrate the weighting function

5.4.3. Extracting Relevant Information

The output of the event-weighting step is a list of phase events ranked according to their relevance. We need to determine a threshold with which we can select the most representative events among this list. For example, a software engineer can decide to only consider the top 20% of the events that have the highest ranking to be the most representative events of a phase.

Another possible method is to set a cap on the maximum number of events we want to extract and compute the number of events per phase proportionally to the size of that phase as follows:

$$R(P_i) = \left\lceil M * \frac{|P_i|}{|T|} \right\rceil$$

where *R*(P_i) in the number of most relevant events of a phase P_i, *M* is the maximum number of events considered given as input, $|T|$ is the size of the trace after removing the utilities, and $|P_i|$ is the size of a phase. In the example of Figure 5.3, if the maximum number of events that we want is set to 3, given that the size of the trace which is 25, we have:

- *R*(*Phase1*)=3*8/25=1
- *R*(*Phase2*)=3*8/25=1
- *R*(*Phase3*)=3*9/25=1

Then for each phase, we chose the top 1 event from the event vector as the most representative of that phase. This way, `m` is the most representative event of Phase 2 and `c` is the most representative of both phases 1 and 3.

We can further improve the information contained in each phase, and hence facilitate the browsing the trace, by enriching the phase most representative events with any descriptive information such as source code comments or any information extracted from valid documentation. We are aware that this informal source of data might not be reliable in practice though. In the worst-case scenario, the event names will be the only information that can be used. Figure 5.4 shows the high-level flow of phases in Trace *T* of Figure 5.3 where the relevant information about each phase is added. The trace *T* can now be browsed as a sequence of phases with relevant information rather than a large trace of events.

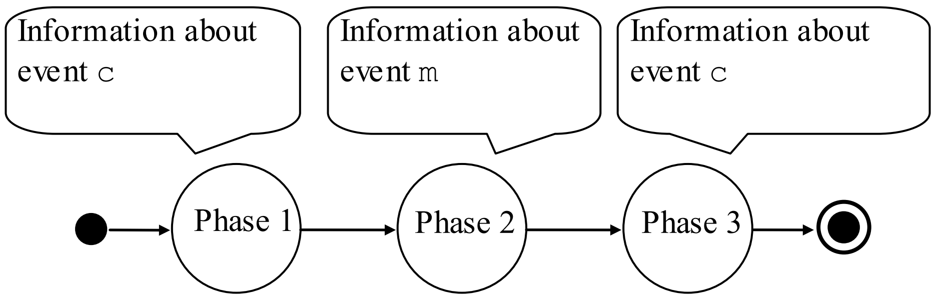

Information about event c
Information about event m
Information about event c
Phase 1
Phase 2
Phase 3

Figure 5.4. Flow of phases with relevant information added

5.4.4. Determining Similar Phases

The similarity between two objects is in general regarded as how much they share in common. In the domain of text mining, the most commonly used measure for evaluating the similarity between two documents is the cosine of the angle between term vectors representing the documents. In the same way, the similarity between two phases can be calculated based on the list of their events vector.

In general, the cosine similarity between two vectors V_x and V_y is calculated as follows:

$$S(V_x, V_y) = \frac{\sum_{i=1}^{n} w_{i,x} * w_{i,y}}{\sqrt{\sum_{i=1}^{n} (w_{i,x})^2} * \sqrt{\sum_{i=1}^{n} (w_{i,y})^2}}$$

where $w_{i,x}$ and $w_{i,y}$ are respectively the weight of event i in vectors V_x and V_y, and the denominator of the fraction is for normalization. The weights cannot be negative and, thus, the similarity between two vectors ranges from 0 to 1, where 0 indicates independence, 1 means exactly the same, and in-between values indicate intermediate similarity. Since the events vectors in our case are already normalized we measure the similarity between each pair of phases by calculating the cosine of the angle between the event vectors P_x, P_y representing the phases as follows:

$$S(P_x, P_y) = \sum_{i=1}^{n} w_{i,x} * w_{i,y}$$

To determine the similarity between the phases, we take the event vector of each phase and measure the similarity between each pair of vectors. If the similarity between two phases is more than a user-specified threshold, they are considered as the same. As a result, for phases that are repeated in a sequence of phases, the first occurrence is kept and the next occurrences are referred to the first occurrence.

For our sample trace *T* in Figure 5.3, the similarities between phases are shown in Figure 5.5. If we consider a threshold of 85% for two phases to be considered similar, then Phase 1 and Phase 3 are the same. This enables us to reduce the high-level flow of the phases to the one shown in Figure 5.5.

	Phase 2	**Phase 3**
Phase 1	0%	91%
Phase 2		9%

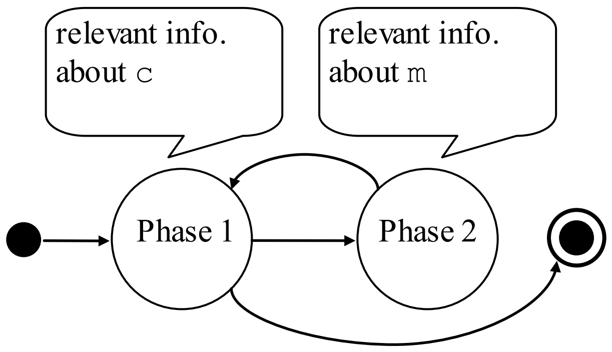


Figure 5.5. Calculating the similarity between phases

5.5. Case Study

We conducted experiments with traces generated from WEKA [WEKA] and ArgoUML [ARG] that we present separately in this section. Both systems where instrumented using TPTP (the Eclipse instrumentation tool).

5.5.1. WEKA

WEKA is an open source machine learning tool that implements several learning algorithms [WEKA]. We used WEKA 3.7.3 (latest version) which consists of 76 packages, 1133 classes, 14210 methods, and 226220 lines of code.

To generate a trace, we applied the WEKA machine learning toolkit to build a decision tree learning algorithm for classifying data instances. Each data instance is typically a vector of attribute values where each attribute denotes some measurement of interest. Training involves executing the decision tree learning algorithm on a set of training data instances. The algorithm identifies specific patterns in the data and outputs a decision tree model each node of which uses a predicate built on specific data attributes to fine tune the classification. The output decision tree model is subsequently evaluated on a separate set of test data instances. The model evaluates the predicate on each node of the decision tree against their corresponding values in each data instance and outputs a class prediction. Different performance statistics, e.g., prediction accuracy over all test instances, are then calculated for evaluation purposes [JS11]. As such, the core process of learning a model consists of three main stages: data input, learning a model from training data, evaluating the model on test data.

The detailed steps of the scenario we used to generate a trace are: (a) Run the WEKA Explorer tool, select a training set, go to the Classify tab, (b) Select the classifier J48 (see [28] for a description on this algorithm), select the "supplied test set" option, (c) Select a test set, start the classification, close the program. The generated trace contains 872,291 method calls. Since it is common to report the size of a trace considering two events (entry and exit events) for each method, the size of the trace in terms of events is 1,744,582. The number of distinct methods involved in the trace is 1309.

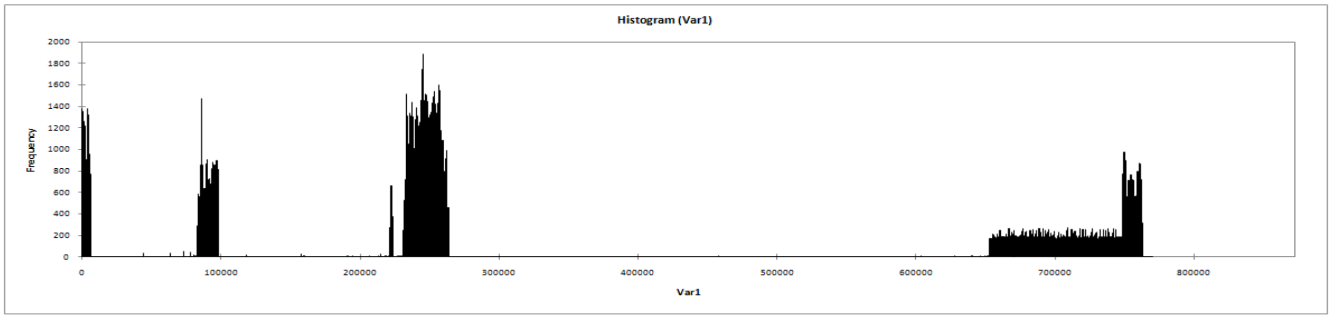


Figure 5.6. Detected phases for execution trace of WEKA

The application of the phase detection technique resulted in the identification of four main phases. This can be seen in Figure 5.6 that shows four dense groups of methods appearing in the phase detection. These dense groups are then mapped back as execution phases to the original trace. Table 5.1 shows the size of the detected phase (SP) in terms of the number of method calls.

In the next step, the original trace with its phases annotated is given to the utility removal component, where accessing methods are removed. The resulting trace is then processed to weigh the events of each phase. Table 5.1 shows the number of events with non-zero weights in the event vector for each phase (SV). The event vector is passed to the "preparation of relevant information" component where the top first 20% of the methods in each event vector is selected as most representative methods for each phase. This threshold is set by the user; a higher percentage provides the user with a higher number of methods. This customization is integrated in our tool to allow enough flexibility to vary this threshold. Table 5.3 shows the representative methods of each phase (the methods are sorted based on their original order of invocation to help with better understanding of the flow of events). Table 5.1 shows the number of representative methods for each phase (SR).

Table 5.1. Statistics about representative events

Phases	(SP)	(SV)	(SR)	Ratio SR/SP
Phase 1	82544	95	19	0.02%
Phase 2	124586	137	27	0.02%
Phase 3	445291	69	14	0.003%
Phase 4	219871	127	25	0.01%

We referred to the source code and the WEKA documentation [WEKA] to extract descriptions of the routines that were deemed most representative of each phase. We were able to interpret the phases that composed the original trace by analyzing the phases' most relevant information, which significantly simplified the understanding of the entire trace. We now briefly discuss the information contained in the trace.

The first phase involves initialization of the WEKA toolkit itself. Since WEKA has a Graphical User Interface (GUI), this initialization also involves calls to processes that establish communication channels through this GUI. Some prominent examples in the most frequently called routine in Phase 1 can be seen with regard to the `weka.core.tee` objects. These objects refer to the WEKA's I/O stream initialization that enables it to both communicate with GUI interface selections by the user and establish streams for data input and results output.

The next phase (Phase 2) involves reading and organizing the data in requisite data structures. This phase prepares the data as well as enables data capabilities based on data specifics. Data organization and preparation is represented by the calls to the `weka.core.Instances` and `weka.core.Capabilities` methods that involve organizing and handling instances in an ordered set, and set the classifier-specific data handling preferences respectively.

The following phase (Phase 3) involves executing the learning algorithm to build a model on the training data. This is represented by methods in the `weka.classifiers.trees.j48 class`.

Finally, the last phase consists of the evaluation of the decision tree model output by Phase 3, on a set of test instances. This is indicated by calls to the methods in `weka.classifiers.Evaluation` class.

Since a prediction is obtained on each individual test data instance, repeated calls to `weka.classifiers.Evaluation.evaluationForSingleInstance` method can be seen. This method performs an instance-wise evaluation of the decision tree model. This phase also estimates different performance statistics for the model.

The phase event vectors were also used to determine the similarity between each pair of phases. As shown in Table 5.2, the calculated similarities between every pair of phases were less than 1%.

Finally, the high-level view of the flow of phases with assigned description (extracted by reading the WEKA documentation) is shown in Figure 5.7. This high-level information flow is obtained by investigating a very small percentage of the original trace, which is quantified as SR/SP (Table 5.1). The content prioritization step, except for assigning description to the selected phase events which is done manually, took 74 sec on an Intel Core Duo CPU 2.00GHz, 2MB cache, 1GB main memory, running Windows XP.

Table 5.2. Similarities between phases for WEKA Trace

	P2	P3	P4
P1	0.18 %	0.04%	0.00%
P2		0.98%	0.68%
P3			0.48%

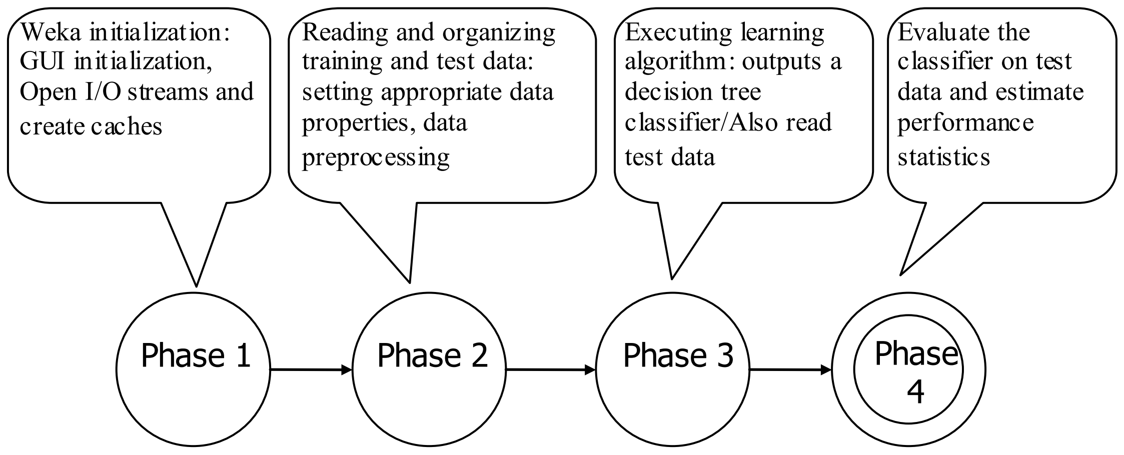

Weka initialization: GUI initialization, Open I/O streams and create caches
Reading and organizing training and test data: setting appropriate data properties, data preprocessing
Executing learning algorithm: outputs a decision tree classifier/Also read test data
Evaluate the classifier on test data and estimate performance statistics
Phase 1
Phase 2
Phase 3
Phase 4

Figure 5.7. Flow of phases with relevant information added

Table 5.3. Representative elements of the WEKA trace

Reps. of Phase 1	**Reps. of Phase 3**
weka.core.Tee.add	weka.core.WekaEnumeration.hasMoreElements
weka.core.WekaPackageManager.loadPackages	weka.core.WekaEnumeration.nextElement
weka.core.ClassDiscovery.initCache	weka.classifiers.trees.j48.Distribution.add
weka.core.ClassCache.initFromDir	weka.classifiers.trees.j48.Distribution.numClasses
weka.core.ClassCache.add	weka.classifiers.trees.j48.Distribution.total
weka.core.ClassCache.cleanUp	weka.core.Instances.quickSort
weka.core.ClassCache.extractPackage	weka.core.Instances.partition
weka.core.ClassCache.initFromJar	weka.classifiers.trees.j48.EntropyBasedSplitCrit.logFunc
weka.core.ClassDiscovery.find	weka.classifiers.trees.j48.Distribution.shiftRange
weka.core.ClassDiscovery.hasInterface	weka.classifiers.trees.j48.Distribution.perBag
weka.core.ClassCache.remove	weka.classifiers.trees.j48.InfoGainSplitCrit.splitCritValue
weka.core.ClassDiscovery.addCache	weka.classifiers.trees.j48.EntropyBasedSplitCrit.newEnt
weka.gui.GenericPropertiesCreator.isValidClassname	weka.classifiers.trees.j48.Distribution.numBags
weka.core.ClassDiscovery.isSubclass	weka.classifiers.trees.j48.Distribution.perClassPerBag
weka.core.Stopwords.add	
weka.core.Tee.size	
weka.core.converters.AbstractFileSaver.resetOptions	
weka.core.converters.AbstractSaver.resetOptions	
weka.gui.GenericObjectEditor.registerEditor	
Reps. of Phase 2	**Reps. of Phase 4**
weka.gui.explorer.Explorer.addCapabilitiesFilterListener	weka.gui.explorer.ClassifierErrorsPlotInstances.process
weka.core.Instances.numAttributes	weka.classifiers.Evaluation.evaluateModelOnceAndRecordPrediction
weka.core.Attribute.indexOfValue	weka.classifiers.Evaluation.evaluationForSingleInstance
weka.core.AbstractInstance.weight	weka.core.AbstractInstance.dataset
weka.core.Instances.numInstances	weka.core.DenseInstance.freshAttributeVector
weka.core.Instances.instance	weka.core.DenseInstance.toDoubleArray
weka.gui.explorer.PreprocessPanel.updateCapabilitiesFilter	weka.classifiers.trees.J48.distributionForInstance
weka.core.Capabilities.assign	weka.classifiers.trees.j48.ClassifierTree.distributionForInstance
weka.core.Capabilities.handles	weka.core.AbstractInstance.numClasses
weka.core.Capabilities.disable	weka.classifiers.trees.j48.ClassifierTree.localModel
weka.core.Capabilities.hasDependency	weka.classifiers.trees.j48.ClassifierTree.son
weka.core.Capabilities.disableDependency	weka.classifiers.trees.j48.ClassifierSplitModel.classProb
weka.core.Instances.classIndex	weka.classifiers.trees.j48.NoSplit.weights
weka.core.Capabilities.enable	weka.classifiers.trees.j48.Distribution.prob
weka.core.AbstractInstance.classIndex	weka.classifiers.Evaluation.updateStatsForClassifier
weka.core.AbstractInstance.isMissing	weka.classifiers.Evaluation.updateMargins
weka.core.DenseInstance.value	weka.classifiers.Evaluation.makeDistribution
weka.core.Instances.attributeStats	weka.classifiers.Evaluation.updateNumericScores
weka.core.AttributeStats.addDistinct	weka.classifiers.evaluation.NominalPrediction.updatePredicted
weka.experiment.Stats.add	weka.core.AbstractInstance.classAttribute
weka.experiment.Stats.calculateDerived	weka.classifiers.evaluation.NominalPrediction.distribution
weka.gui.explorer.ClassifierPanel.updateCapabilitiesFilter	weka.classifiers.evaluation.NominalPrediction.actual
weka.gui.explorer.ClustererPanel.updateCapabilitiesFilter	weka.classifiers.evaluation.NominalPrediction.weight
weka.gui.explorer.AttributeSelectionPanel.updateCapabilitiesFilter	weka.gui.visualize.Plot2D.convertToPanelX
weka.core.Instances.swap	weka.gui.visualize.Plot2D.convertToPanelY
weka.core.Attribute.isString	
weka.core.Capabilities.enableDependency	

5.5.2. ArgoUML

For the second case study, we applied our technique to a trace generated from ArgoUML [ARG] by exercising the following scenario: Starting up ArgoUML, drawing a class on the class diagram, and quitting ArgoUML). The resulting trace contained 38321 method calls (2330 distinct methods). Figure 5.8 shows the dense groups formed as the result of applying the phase detection technique to the ArgoUML trace. These groups are mapped back to the original trace as five phases. Table 4 shows the size of each detected phase as the number of method calls (SP). Similar to the previous case study, we applied the removed the accessing methods.

Then, we performed the weighting step on the resulting trace events. The top 20% of each vector was selected as most representative events of each phase (see Table 5.4 for the number of representatives for each phase (SR)). The information about the representative methods is gathered from the documentation and comments in the source code of the system [TCA].

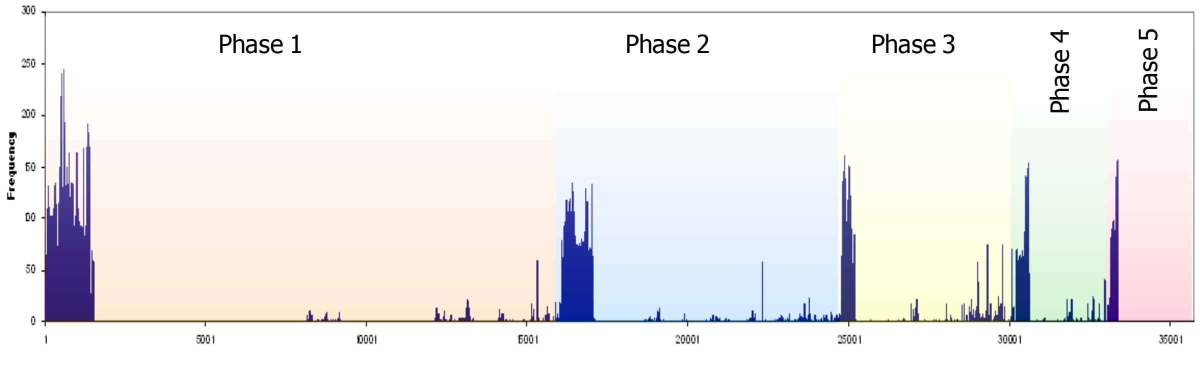


Figure 5.8. Detected phases for execution trace of ArgoUML

Table 5.4. Statistics about representative events

Phases	(SP)	(SV)	(SR)	Ratio SR/SP
Phase 1	16035	334	47	0.29%
Phase 2	9089	231	34	0.37%
Phase 3	4225	270	38	0.89%
Phase 4	3832	113	16	0.41%
Phase 5	5140	83	12	0.23%

Similar to the previous system, we were able to understand the original trace by examining the most relevant events of its phases, which we briefly review in what follows.

The first phase focuses on the initialization of ArgoUML where the main application frame (e.g., main panes: navigation pane, multieditor pane, to-do pane, and details pane), status bar, and project are set up. The second phase is concerned with loading auxiliary modules from the input stream and adding them to the Post Load Actions list, which contains actions that are run after ArgoUML has started. The third phase is the phase where the actual class element is drawn. This phase is followed with two other small phases. The first of these phases (Phase 4) refreshes and updates the models properties set in the previous phase, such as boundaries, NameText, font, etc. The representative methods of the last phase (e.g., save methods, menu selection method, and exit methods) clearly show the termination of the application. As an example, Table 5.6 shows the representative methods of Phase 3 and Phase 5.

Table 5.5. Similarities between phases for ArgoUML Trace

	P2	P3	P4	P5
P1	0.79 %	0.16%	0.01%	0.00%
P2		0.32%	0.33%	0.53%
P3			2.50%	0.13%
P4				3.75%

The event vectors are then used to measure the similarity between phases. As shown in Table 5.5, a very small similarity between the phases does not suggest any change to the sequence of phases in the high-level. Finally, the high-level view of the flow of phases with assigned description is shown in Figure 5.9. Table 4 shows the percentage of the event investigated in each phase to extract relevant information (SR/SP). The content prioritization stage except for the information gathering which is done manually took 14 sec on an Intel Core Duo CPU 2.00GHz, 2MB cache, 1GB main memory, running Windows XP.

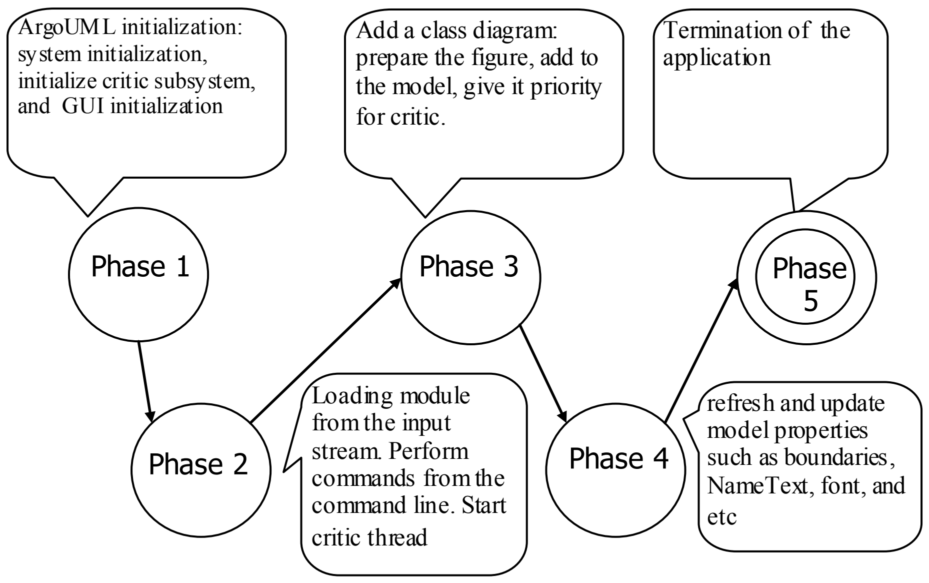

ArgoUML initialization: system initialization, initialize critic subsystem, and GUI initialization
Add a class diagram: prepare the figure, add to the model, give it priority for critic.
Termination of the application
Phase 1
Phase 3
Phase 5
Phase 2
Loading module from the input stream. Perform commands from the command line. Start critic thread
Phase 4
refresh and update model properties such as boundaries, NameText, font, and etc

Figure 5.9. Flow of phases with relevant information added

Table 5.6. Representative events of the ArgoUML trace

Reps. of Phase 3
org.argouml.ui.explorer.ExplorerTreeModel.traverseModified
org.argouml.ui.explorer.ExplorerTreeNode.nodeModified
org.argouml.uml.ui.UMLModelElementListModel2.getTarget
org.argouml.ui.StylePanel.getPanelTarget
org.argouml.uml.util.namespace.StringNamespaceElement.toString
org.argouml.uml.ui.UMLModelElementListModel2.setAllElements
org.argouml.uml.ui.UMLModelElementListModel2.addAll
org.argouml.uml.ui.UMLModelElementListModel2.rebuildModelList
org.argouml.uml.ui.UMLModelElementListModel2.setTarget
org.argouml.uml.ui.UMLModelElementListModel2.addOtherModelEventListeners
org.argouml.uml.ui.UMLModelElementListModel2.targetSet
org.argouml.uml.ui.UMLCheckBox2.getTarget
org.argouml.uml.ui.ScrollList.addNotify
org.argouml.uml.diagram.ui.FigCompartment.getBigPort
org.argouml.ui.explorer.ExplorerTreeModel.modelElementChanged
org.argouml.uml.ui.PropPanel.collectTargetListeners
org.argouml.uml.ui.UMLList2.getTargettableModel
org.argouml.uml.ui.UMLCheckBox2.setTarget
org.argouml.uml.util.namespace.StringNamespace.pushNamespaceElement
org.argouml.uml.util.namespace.StringNamespaceElement.-init-
org.argouml.ui.explorer.ExplorerTreeModel$ExplorerUpdater.schedule
org.argouml.uml.ui.UMLCheckBox2.targetSet
org.argouml.uml.diagram.static_structure.ui.FigClassifierBox.propertyChange
org.argouml.uml.diagram.ui.FigNodeModelElement.propertyChange
org.argouml.uml.diagram.ui.FigEditableCompartment$FigSeperator.getMinimumSize
org.argouml.uml.ui.ScrollList.-init-
org.argouml.uml.ui.UMLModelElementListModel2.removeOtherModelEventListeners

org.argouml.ui.explorer.ExplorerTreeNode.getPending
org.argouml.ui.explorer.ExplorerTreeModel.getNodeUpdater
org.argouml.uml.ui.ScrollList.removeNotify
org.argouml.uml.diagram.static_structure.ui.StylePanelFigClass.itemStateChanged
org.argouml.uml.util.namespace.StringNamespace.toString
org.argouml.uml.diagram.ui.FigNodeModelElement.getStereotypeFig
org.argouml.uml.diagram.ui.FigNodeModelElement.getNameFig
org.argouml.uml.diagram.ui.FigNodeModelElement.addElementListener
org.argouml.notation.NotationProvider.addElementListener
org.argouml.uml.diagram.static_structure.ui.FigClassifierBoxWithAttributes.updateListeners
org.argouml.uml.diagram.ui.FigNodeModelElement.addElementListeners
Reps. of Phase 5
org.argouml.notation.NotationNameImpl.getIcon
org.argouml.notation.NotationNameImpl.sameNotationAs
org.argouml.configuration.Configuration.save
org.argouml.configuration.ConfigurationHandler.saveDefault
org.argouml.ui.ProjectBrowser.saveScreenConfiguration
org.argouml.ui.cmd.ActionNotation.menuDeselected
org.argouml.kernel.ProjectSettings.getNotationName
org.argouml.ui.ProjectBrowser.tryExit
org.argouml.configuration.ConfigurationProperties.saveFile
org.argouml.ui.cmd.ActionExit.actionPerformed
org.argouml.ui.cmd.ActionNotation.menuSelected
org.argouml.profile.internal.ocl.EvaluateExpression.loadState

5.6. Threats to Validity

In our case studies, we used the content prioritization approach to find out what is happening in different execution phases of each system. First, we used our trace segmentation technique to detect the execution phases. One can use other phase detection techniques to extract phases. Changing the phase detection component might result in detection of different phases which in turn can potentially change the end result of our content prioritization approach. It is therefore important to investigate how various phase detection algorithms can be used.

Furthermore, in our case studies, we manually generated a human readable description for each phase based on the relevant events of each phase. This can pose a threat to internal validity. In fact, given the same set of relevant events, one might come up with a different description. Thus, automatic ways of generating descriptions should be investigated.

5.7. Summary

Large amount of data in any form (traces, text, etc.) is in principle subject to similar challenges, among which perhaps the most important ones consist of coping with the size, overcoming the limited capacity of the human working memory, and the constant need to reduce the presence of noise in the data so as to focus on what is important. Based on this observation, in this chapter, we proposed an approach for prioritizing the content of execution traces that aim to identify the most relevant events of different parts of a trace.

Our idea was that an event that is frequently invoked in a certain part of a trace is relevant to what is being done in that part, while an event that is shared between different parts of the trace is less representative of any part of the trace. We found this idea, in nature, similar to the idea of TF-IDF in text mining.

We explored the possibility of applying existing techniques in text mining to rank different events of a trace. For this, we first determined a mapping between the concepts in the two domains of text mining and trace analysis. We proposed to map an execution trace as a corpus and the trace segments (execution phases) that compose it as the corpus's documents. Each event is also mapped as a term within a phase document. We made sure that the mapping holds not only between objects, but also between the containment relations between the objects of the two domains.

We used the phases that we detect through our proposed phase detection technique and ranked their events according to TF-IDF. As a result, each phase was assigned an events vector where the events were arranged according to their relevance. We reported top events as the relevant events of each phase.

We also used the events vectors to find the similarity of each pair of phases in our phase flow. This way, we made it possible to detect if a phase is being repeated in the flow of phases. We kept the first occurrence of each phase and refer to it if there is a repetition.

Finally, we performed a number of case studies on open source system to evaluate our content prioritization approach. We identified the representative events of each phase in our traces. We also reported the phase flow in each case study and verified if the phase flow is in accordance to what was expected according to the system’s documentations. The results were promising.

Chapter 6. Stratified Sampling of Execution Traces

6.1. Introduction

One way to cope with a large execution trace is to sample the trace and use the sampled trace for further analysis. This technique has been used in several approaches (e.g., [CHMY03, RR03, RZ05, Dug07]) to reduce the size of traces. Sampling consists of selecting a subset of trace events for analysis instead of analyzing the entire trace.

A major drawback of existing sampling approaches is that there is no guarantee that the resulting sampled trace is representative of the original trace. This appears to be due to the fact that existing sampling techniques are blind to the information contained in the trace; they treat a trace as a stream of data for which the pieces are considered equal.

In a study performed by Cornelissen et al. [CMZ08], the authors compared four trace abstraction techniques. These techniques were consisted of sampling, subsequence summarization [KG06], language-based filtering, and filtering events based on their nesting levels. For sampling a trace, they kept every n-th event (i.e., a systematic sampling was performed). The result of their study suggested that while sampling performs well in terms of reducing trace size, it is the least useful technique (among the four techniques) in preserving high-level and medium-level information contained in their traces. This means that the sampled traces in their study were not representative of the original traces.

Using unrepresentative samples can seriously limit the trace analysis, as we might not be able to make inference about a trace based on the sample from that trace.

To overcome this limitation, in this chapter, we present an approach for reducing the size of traces that is based on the stratified sampling of execution traces. We first divide the trace into execution phases using the approach presented in Chapter 4. A trace can then be seen as a sequence of exhaustive and non-overlapping execution phases rather than a mere flow of events. By using execution phases as strata, we ensure that a certain number of events will be selected from each execution phase to yield a sample that is representative of the original trace.

This chapter is organized as follows: in Section 6.2, we theoretically discuss the cases that can be problematic in random sampling of execution traces. In Section 6.3, we introduce our proposed approach for stratified sampling of execution traces along with the steps needed in the process of sampling. In Section 6.4, we present the sampling unit of our proposed approach. In Section 6.5, we present the case studies to evaluate our proposed sampling approach. Finally, in Section 6.7, we conclude this chapter with a summary.

Parts of the material in this chapter is adapted and expanded from a paper published in the 16th IEEE International Conference on Program Comprehension, 2011 [PSHM11].

6.2. Reasoning about Sampling

To help with the description of these techniques, we present the following definitions. A *population* can be defined as a group of elements (people, plants, animals, cars, numbers, etc.) about which we want to make judgments. Studying an entire population may be slow and expensive. *Sampling* is a process through which we select parts of a population for analysis instead of analyzing the entire population. To be able to generalize the results of the analysis on a sample to the population, the sample has to be representative of the population. Sample representativeness means that the characteristics of the sample closely match those of the population. Thus, the goal in sampling is to find a representative sample of the population.

In trace sampling, the population is the trace under study. We refer to this trace as the *original trace*. A *sampled trace* is a trace generated through the sampling of an original trace. Similar to other fields, in trace sampling, the aim is to generate a sampled trace that is representative of the original trace. Given that an original trace represents the functionalities triggered by the user, a sampled trace is *representative* of its original trace if the sampled trace can represent similar functionalities triggered in the original trace.

A simple and a naive way to sample the content of a trace is to consider every n-th generated event. The size of the trace can be controlled automatically based on the sampling parameter n. The sampling parameter (also called distance) for a given trace T in systematic sampling is usually represented as follows:

$$n = \frac{|T|}{|T'|}$$

where $|T'|$ is the size of the sampled trace which must be specified by the user and $|T|$ is the size of the original trace (i.e., the number of events recorded during the generation of the trace).

Although efficient, this sampling technique might be biased when the original trace, for example, possesses iterations (or patterns) that coincide with the *n* value. As an example, suppose in a trace *T* where one specific event *e* is repeated after each six other events. If the sampling parameter happens to be seven ($n = 7$), then depending on where the sampling starts, one could obtain a sample either with all *e*s or with no *e*.

One way to overcome this problem is to use random sampling, which is a technique that is commonly used in trace analysis (e.g., [CHMY03, RR03, RZ05, Dug07]). Instead of selecting every *n*-th event, trace events are sampled in a way that each event has equal chance to be selected (if we have *x* event, each of them would have $1/x$ chance of being selected). If we do not exclude the events that have been drawn from further selection, the resulting sampling strategy is called random sampling with replacement. Otherwise, it is called random sampling without replacement. It should be noted that random sampling can result in a sampled trace where the invocation order of the events is changed. This could be potentially dangerous when sampling a trace where the temporal order of events must be kept. To avoid this, one can sort the events in the sampled trace according to their temporal order in the original trace, if this information is available. For example, if we have a trace *T*:{e1, e2, e3, e4, e5, e6, e7} and we want to generate a sampled trace *T'* of size 3 with random sampling by drawing events one by one without replacement. The first event drawn is e6, followed by e2, and e7, that is, {e6, e2, e7}. Once sorted, we have the sampled trace *T'*: {e2, e6, e7}. The problem with random sampling is that it makes no use of auxiliary information about the trace (e.g., distribution of the trace events, the homogeneous nature of its parts, outliers, etc.) that could assist in selecting a sample that is more representative of the original trace.

Statistically, when we are dealing with a population that is not homogeneous (i.e., it is made up of elements that are different from each other in sub-populations, and each sub-population represents a group of similar events), then random sampling might result in an unrepresentative sample [Bru60]. It is a common situation for execution traces not to be homogeneous. The reason is that a trace is composed of a sequence of events where each subsequence represents a specific task performed by the system. The events in one particular set of events can be completely different from the ones of another subsequence.

We can study the representativeness problem of random sampling of execution traces in the following formal framework. Given an original trace *T* of method calls, a sampled trace *T'* can be built by randomly drawing method calls from the original trace without replacing them. Let *T* be composed of homogeneous subsequences of method calls $\{h1, h2, \ldots, hn\}$. Then, $\overline{P}(h_c)$ the probability that no method call from a candidate homogeneous subsequence h_c appears in the sampled trace *T'* is calculated as follows:

$$\overline{P}(h_c) = \left(1 - \frac{|h_c|}{|T|}\right) \times \left(1 - \frac{|h_c|}{|T|-1}\right) \times \left(1 - \frac{|h_c|}{|T|-2}\right) \times \ldots \times \left(1 - \frac{|h_c|}{|T|-|T'|}\right)$$

$$= \frac{(|T| - |h_c|)! \times (|T| - |T'|)!}{(|T|)! \times (|T| - |h_c| - |T'|)!}$$

where $|h_c|$ is the size of h_c, $|T'|$ is the size of the sampled trace. Thus, $\overline{P}(h_c)$ is the multiplication of the probability that, on each draw from the execution trace, we do not select a method call from h_c. The formula shows the problematic situations in random sampling of trace *T,* which are the cases where no method from a homogeneous subsequence appears in the sample (high values of $\overline{P}(h_c)$), resulting in an unrepresentative sample (and sometimes a sampled trace that is not informative at all). Therefore, having a trace *T* with size $|T|$, we need to analyze the value of $\overline{P}(h_c)$ according to the size of the sampled trace $|T'|$ and the size of a candidate homogeneous subsequence $|h_c|$ looking for cases that result in high $\overline{P}(h_c)$.

Figure 6.1 shows the behaviour of $\overline{P}(h_c)$ according to the changes of $|T'|$ and $|h_c|$. As shown in Figure 6.1, in random sampling, the smaller the size of the sampled trace, the higher is the probability of having an unrepresentative sample. Furthermore, small sizes of homogeneous subsequences can also result in unrepresentative samples.

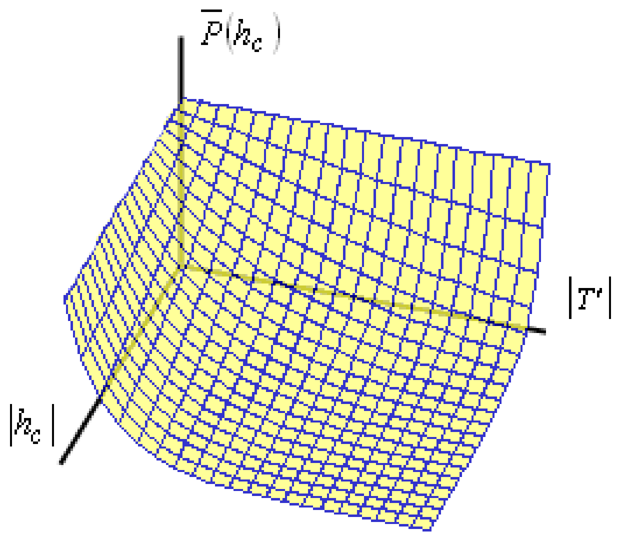


Figure 6.1. Behavior of $\overline{P}(h_c)$ with respect to $|T'|$ and $|h_c|$

6.3. Stratified Sampling of Execution Traces

Another approach to sampling, extensively studied in Information Theory is known as stratified sampling [Coc77]. Stratified sampling techniques are generally used when the population on which sampling is applied is heterogeneous as a whole but can be divided into homogeneous sub-populations, referred to as strata. We deal with similar situation in execution traces, as they are composed of a sequence of events where one can find subsequences that represent specific tasks performed by the system. The level of granularity of a task depends on the type of samples that we want to extract.

In stratified sampling of a population, first, the population is separated into a desired number of partitions[1] (called strata) and then sample elements are drawn from within each stratum. The size of the sample from each stratum is kept proportional to the size of the

[1] A partition is a non-overlapping and exhaustive sub-population.

stratum (this is called proportionate stratified sampling). We thus guarantee that the final sample contains elements representative of every part of the population. The process of stratified sampling is shown as a flow chart in Figure 6.2.

The quality of stratified sampling is determined by the way strata are specified (strata specification), and the strategy by which sample elements are drawn from within each stratum (selection strategy). In strata specification, strata are commonly created by dividing the population into partitions of relatively homogeneous elements.

Drawing a parallel between population sampling and trace sampling, we are interested in a trace sampling method that can create a more representative sample in comparison with existing trace sampling methods. We propose stratified sampling of execution trace. For strata specification, we propose using execution phases as strata. We use the approach proposed in Chapter 4 to detect execution phases.

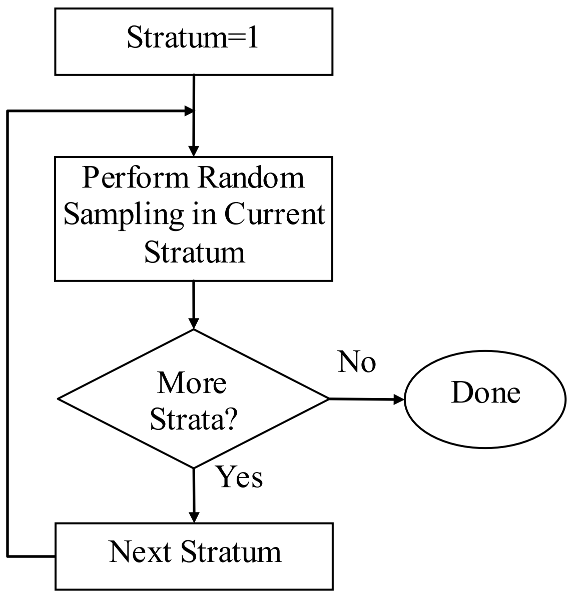


Figure 6.2 Stratified sampling process

Once the phases are detected, we select sample events within each stratum (phase) using random sampling. It might sound contradictory that we are using random sampling after criticizing it in the previous sections. In fact, the problem with random sampling is when it is used on non-homogeneous data spaces such as an entire trace. This is not the case now because it is applied to the contents of execution phases, as we detect in Chapter 4, are by definition homogeneous.

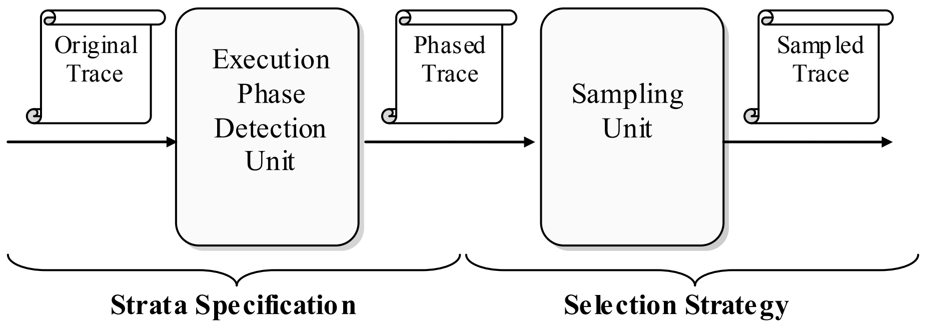


Figure 6.3. Overview of the proposed sampling framework

Our framework for stratified sampling of execution traces is shown in Figure 6.3. By splitting the process into different units, our proposed framework achieves a flexible and extensible architecture where each unit or its composing components can be supplemented or replaced by alternative approaches. As shown in Figure 6.3, we first need to identify the execution phases to serve as strata for our stratified sampling approach. We use the execution phase detection unit to detect the execution phases of the trace. The result of this unit is then given to the sampling unit that outputs a sampled trace. Both units are described in more details in the following sections.

6.4. Sampling Unit

Once the phases are detected, we start the stratified sampling process, which is implemented in our framework, as part of the sampling unit (shown in Figure 6.4). The sampling unit receives a phased execution trace as its input and outputs a sample of the execution trace using stratified sampling.

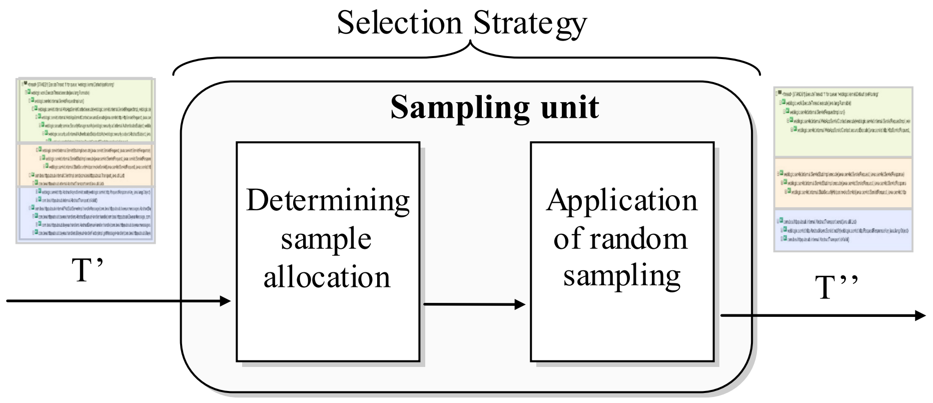


Figure 6.4. The Sampling unit in details

We use the sample trace shown in Figure 6.5 (part 1) to explain how the sampling is performed in a step-by-step fashion. Figure 6.5 (part 2) shows the trace divided into 4 major phases as a result of the application of our phase detection approach presented in Chapter 4. The trace in Figure 6.5 contains 25 events and we would like to obtain a sampled trace of size 6.

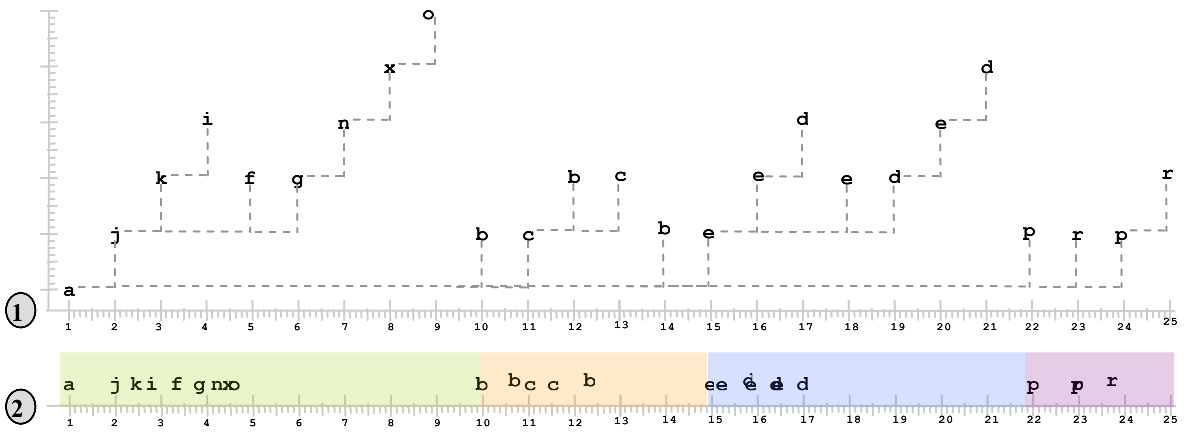


Figure 6.5. An example of applying the integrated scheme on a sample trace

As mentioned previously, the sampling unit treats the phases of the execution trace as strata. More precisely, given a phased trace *T*, the sampling unit defines *H* strata in the trace, where each stratum is a phase. Each event of the trace is assigned to one, and only one:

$$|T| = \left|Stratum_1\right| + \left|Stratum_2\right| + \ldots + \left|Stratum_{H\text{-}1}\right| + \left|Stratum_H\right|$$

where $Stratum_h$ is the number of events in each stratum. Table 6.1 shows the number of events within each stratum of our sample trace in Figure 6.5.

Table 6.1. Execution phases that were detected

Phase	**Phase Location**	**Strata**	**Size**
P1	1 – 9	$Stratum_1$	9
P2	10 – 14	$Stratum_2$	5
P3	15 – 21	$Stratum_3$	7
P4	22 – 25	$Stratum_4$	4

The next step is to select a sampled trace of size |*T'*|, which is an aggregation of the samples selected from each stratum. More precisely:

$$|T'| = |S_1| + |S_2| + |S_3| + \ldots + |S_{H-1}| + |S_H|$$

where $|S_h|$ is the number of events sampled from $Stratum_h$. For sample allocation, we must determine the size of the sample for each stratum. The number of events to be sampled from each stratum is kept proportional to the size of the stratum:

$$|S_h| \approx \frac{|Stratum_h|}{|T|} \times |T'| \qquad h \in \{1 \ldots H\}$$

Therefore, we select $|S_h|$ events from each stratum. For our sampled trace of Figure 6.5, the number of samples to be drawn from each stratum is calculated the same way:

$$|S_1| \approx \frac{9}{25} \times 6 = 2 \quad |S_2| \approx \frac{5}{25} \times 6 = 1$$
$$|S_3| \approx \frac{7}{25} \times 6 = 2 \quad |S_4| \approx \frac{4}{25} \times 6 = 1$$

Finally, since the events within each stratum are homogeneous, we perform the selection of trace events from each stratum using random sampling as discussed earlier.

6.5. Case Study

We apply our proposed phase-based stratified sampling approach in two case studies. The goal of the first case study is to analyze the use of phase-based stratified sampling of execution traces, with the purpose of comparing its usefulness with random sampling in program comprehension. The second case study is a real-world running example of the problematic case that is theoretically discussed in Section 6.2 and shows how our phase-based stratified sampling compares to random sampling of execution traces. The case studies are on traces generated from two different systems: WEKA 3.0 [WEKA] and JHotDraw 5.2 [JHO]. To generate the traces we instrumented both systems using TPTP (the Eclipse Test and Performance Tools Platform) [TPTP].

The Sampling unit (Section 6.4) of the Tratex implements random sampling that is used in stratified sampling. The implementation of random sampling uses the Random class in Java to generate a random number in constant time. Similar implementation is used to generate sampled traces through random sampling in our case studies.

6.5.1. WEKA

The experimental unit in this study is WEKA. We selected to analyze the C4.5 classification algorithm that builds a decision tree for classifying data instances. For this we use the C4.5 trace (the original trace) generated by Hamou-Lhadj et al. [HL06].

The main factor in this study is the sampling method that is applied on the original trace to extract a sample to be used for program comprehension. The dependent variable in this study is the comprehension level. The comprehension level is evaluated on authoritative bases. That is, for each sampling method, the extracted sampled trace is compared with established reference data provided by Hamou-Lhadj et al. [HL06]. The reference data is a summarized version of the original trace (hereby referred to as the oracle trace) that has been shown to have captured the most important interactions of the trace and to be effective in program comprehension.

The comprehension score of a sampled trace is the percentage of events that exist in both the sampled trace and the oracle trace. This way, a higher comprehension score of a sampled trace represents a higher comprehension level and a lower comprehension score represents a lower comprehension level that can be achieved using the sampled trace. It is important to note that the comprehension score calculated by means of comparison to the oracle trace represents, by consequence, a subjective feedback, and that more objective measurements such as those used to assess the comprehension level during the maintenance tasks may result in a more precise conclusions.

To investigate the effect of the main factor on the dependent variable we formulate the following hypotheses:

- H_0 (Null hypothesis): When performing a comprehension task, the use of phase-based stratified random sampling (versus random sampling) does not significantly improve (or decline) the comprehension level.
- H_a (Alternative hypothesis): When performing a comprehension task, the use of phase-based stratified random sampling (versus random sampling) significantly improves (or declines) the comprehension level.

We are interested in investigating how the effect of phase-based stratified sampling on software comprehension compares to the ones of random sampling. Therefore, our null hypothesis is two-tailed.

Experiment Design and Procedure

The experiment design in our study is a variant of *after-only with control group design* [Zik00] where we compare two groups of subjects: one treated with random sampling and the other treated with phase-based stratified sampling and then trying to infer a difference in the performance of the two treatments.

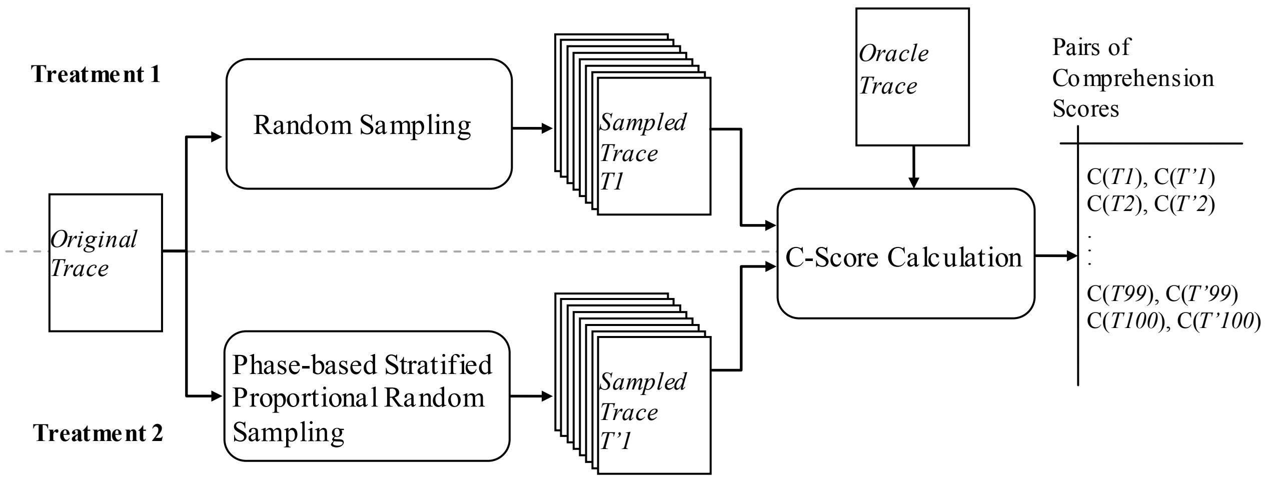


Figure 6.6. Overview of the experiment

For this, as shown in Figure 6.6, we apply random sampling on the original trace to obtain a sampled trace *T1* of a given size. Similarly, we apply the phase-based stratified random sampling on the original trace to obtain another sampled trace *T'1*. Both *T1* and *T'1* are of a given size *s* (equal to the oracle trace size) and the sampling is performed without replacement. The C-Score Calculation module, receives the sampled trace *T1* and compares it with the oracle trace and assigns it a comprehension score C(*T1*), the percentage of the events that are common between *T1* and the oracle trace. Similarly, a C(*T'1*) is assigned to *T'1*. The pair of (C(*T1*), C(*T'1*)) is added to the list of observations for statistical analysis.

Statistical Analysis:

Since the observed comprehension scores of sampled traces are not normally distributed, we need to use a type tests that has no assumption on normality of observations. Non parametric tests are a good choice as they have minimal assumptions on the observations. Furthermore, comprehension scores are ordinal data which is a natural fit for non-parametric tests. Moreover, non-parametric test are not sensitive outlier.

Since the same original trace (the subject) is used for both random sampling and phase-based stratified sampling, we use a paired test (similar in concept with pre-test/post-test data). Wilcoxon signed ranks test [Con80] is a non-parametric paired test. We decided to use Wilcoxon test to evaluate out two tailed null hypothesis.

We carried out our statistic analysis procedures through the statistical package for the social sciences software (SPSS v.20) [SPSS]. A statistic significance level of 0.05 was considered, that is, the null hypothesis could be rejected in all the situations where the probability associated with the statistics of the test (p-value) was inferior to this value.

Results

This section reports and analyzes results obtained from our experiments[1]. The original trace contained 97,413 method calls. We applied our phase detection technique on the trace to detect its major phases (i.e., t = trace size). The application of our integrated scheme results in the formation of 11 dense groups in the rearranged trace. These groups can be seen in Figure 6.7. These 11 groups indicate 11 phases when mapped back the original trace. Table

[1] For replication purposes, the experimental package and raw data from the experiment are available for downloading at: http://www.ece.concordia.ca/~s_pirzad/sampling-case-study/

6.2 shows the detected phases and their information. On an Intel Core i5 CPU 2.30GHz, 4.00 GB main memory, running Windows 7 it took 12.838 seconds for our phase detection algorithm to detect phases. Out of this time, 1.375 seconds were spent on the application of the integrated scheme and 11.463 seconds were spent on clustering.

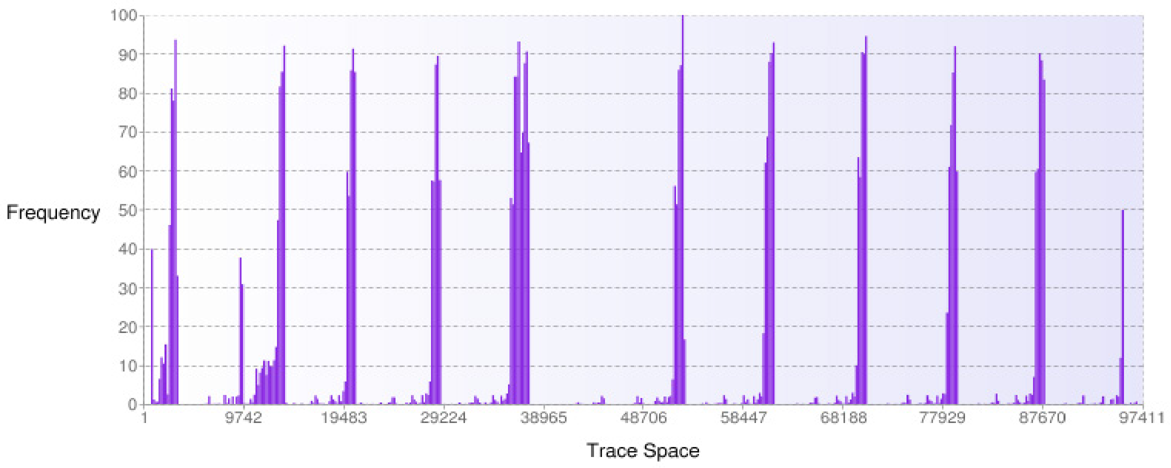


Figure 6.7. Dense groups resulted from applying the integrated scheme on the trace.

These phases were then used as strata to perform stratified sampling. To implement the random sampling within each stratum, our program randomly draws a method call from that stratum and excludes the drawn method call from further selection. The oracle trace contained 31 method calls. We executed the experiment 100 times (obtain 100 pairs of observation) with s = 31. Table 6.4 shows the list of observations.

A pair-wise application of the Wilcoxon test shows that comprehension scores of phase-based random sampling are significantly higher than random sampling scores, (z = –2.628, n = 100, p = 0.009, two-tailed). This information is shown in details in

Table 6.3. This table tells us that the statistic is based on the negative ranks, that the z-score is –2.628 and that this value is significant at p = 0.009. Therefore, because this value is based on the negative ranks, we should conclude that there was a significant increase in comprehension score from random sampling to phase-based stratified random sampling.

Table 6.2. Phases information

Phase	**Phase Location**	**Size**
P1	1 – 8836	8836
P2	8837 – 19299	10463
P3	19300 – 27673	8374
P4	27674 – 35500	7827
P5	35501 – 51651	16151
P6	51652 – 60537	8886
P7	60538 – 69728	9191
P8	69729 – 78640	8912
P9	78641 – 87275	8635
P10	87276 – 95811	8536
P11	95812 – 97413	1602

Table 6.3. Statistical analysis

Descriptive Statistics

	N	Mean	Std. Deviation	Minimum	Maximum
Random Sampling	100	.9355	1.73352	.00	6.45
Phase-based Stratified Sampling	100	1.7742	2.35468	.00	9.68

Ranks

		N	Mean Rank	Sum of Ranks
Phase-based Stratified Sampling - Random Sampling	Negative Ranks	17[a]	26.65	453.00
	Positive Ranks	37[b]	27.89	1032.00
	Ties	46[c]		
	Total	100		

a. Phase-based Stratified Sampling < Random Sampling

b. Phase-based Stratified Sampling > Random Sampling

c. Phase-based Stratified Sampling = Random Sampling

Test Statistics[a]

	Phase-based Stratified Sampling - Random Sampling
Z	-2.628[b]
Asymp. Sig. (2-tailed)	.009

a. Wilcoxon Signed Ranks Test

b. Based on negative ranks.

Table 6.4. Comprehension scores of trace resulted from random sampling and stratified sampling

	Random Sampling	Stratified Sampling		Random Sampling	Stratified Sampling		Random Sampling	Stratified Sampling
1	0.00	0.00	**35**	0.00	3.23	**69**	0.00	0.00
2	0.00	0.00	**36**	3.23	0.00	**70**	0.00	3.23
3	3.23	0.00	**37**	0.00	6.45	**71**	0.00	3.23
4	6.45	0.00	**38**	3.23	3.23	**72**	0.00	3.23
5	0.00	0.00	**39**	0.00	0.00	**73**	0.00	3.23
6	3.23	0.00	**40**	3.23	0.00	**74**	0.00	3.23
7	0.00	3.23	**41**	0.00	3.23	**75**	0.00	0.00
8	0.00	0.00	**42**	0.00	0.00	**76**	0.00	0.00
9	0.00	6.45	**43**	0.00	3.23	**77**	3.23	3.23
10	0.00	0.00	**44**	0.00	6.45	**78**	0.00	0.00
11	3.23	0.00	**45**	0.00	0.00	**79**	0.00	0.00
12	0.00	0.00	**46**	3.23	0.00	**80**	0.00	3.23
13	0.00	0.00	**47**	0.00	0.00	**81**	0.00	0.00
14	0.00	3.23	**48**	6.45	3.23	**82**	0.00	0.00
15	0.00	0.00	**49**	0.00	0.00	**83**	0.00	0.00
16	3.23	3.23	**50**	0.00	0.00	**84**	3.23	0.00
17	0.00	0.00	**51**	0.00	3.23	**85**	3.23	0.00
18	0.00	0.00	**52**	0.00	0.00	**86**	0.00	0.00
19	0.00	0.00	**53**	0.00	0.00	**87**	0.00	3.23
20	0.00	3.23	**54**	0.00	0.00	**88**	3.23	3.23
21	0.00	0.00	**55**	0.00	0.00	**89**	0.00	0.00
22	0.00	0.00	**56**	0.00	0.00	**90**	0.00	3.23
23	0.00	3.23	**57**	3.23	9.68	**91**	3.23	6.45
24	0.00	3.23	**58**	0.00	9.68	**92**	0.00	3.23
25	0.00	3.23	**59**	0.00	0.00	**93**	0.00	0.00
26	0.00	6.45	**60**	3.23	0.00	**94**	0.00	3.23
27	0.00	3.23	**61**	0.00	3.23	**95**	0.00	0.00
28	0.00	0.00	**62**	0.00	3.23	**96**	0.00	3.23
29	0.00	0.00	**63**	3.23	6.45	**97**	0.00	3.23
30	3.23	0.00	**64**	6.45	0.00	**98**	3.23	0.00
31	3.23	0.00	**65**	0.00	6.45	**99**	3.23	0.00
32	6.45	0.00	**66**	0.00	3.23	**100**	0.00	0.00
33	0.00	6.45	**67**	0.00	0.00			
34	0.00	3.23	**68**	3.23	3.23			

6.5.2. JHotDraw

In this section, we show how our approach compares to random sampling. To generate an execution trace, we used an execution scenario that involves several major features. The execution trace was generated based on a scenario where the features F1 to F12 shown in Table 6.5 were exercised one after another.

Table 6.5. The features included in the traced scenario

F1: Drawing a rectangle.	F5: Drawing a circle.	F9: Drawing a round rectangle.
F2: Moving the rectangle.	F6: Moving the circle.	F10: Moving the round rectangle.
F3: Saving work sheet.	F7: Deleting the circle.	F11: Deleting the round rectangle.
F4: Deleting the rectangle	F8: Saving work sheet.	F12: Saving work sheet.

Because JHotDraw registers all mouse movements, and mouse movements are required while drawing figures, the trace that resulted from our scenario was bound to contain a lot of noise. We have therefore filtered these mouse movements to obtain a trace that is cleaner. We are aware that the detection of noise in a trace might not always be straightforward and that noise detection techniques such as the ones presented by Hamou-Lhadj et al. in [HL06] might need to be used. The resulting trace contained 36571 method invocations and the trace file was of size 1.8 MB. Note that a method invocation requires at least two events to be collected, the entry and exit of a method. The trace size in terms of events is therefore about 73142 events, which is considered a relatively medium-sized trace.

We randomly set the phase detection threshold to $t = 200$ so as to detect phases that are not too large but not too fine-grained either. This threshold is a result of conducting several experiments with JHotDraw traces. We still do not have a solution on how such a threshold should be selected automatically to detect adequate phases. Although, we anticipate that it would be application-specific, further studies should be conducted to, at least, provide hints on acceptable ranges of thresholds and their impact on detecting phases. We have not done such studies yet.

Figure 6.8 shows the results of applying the integrated gravity, using t, to the JHotDraw trace.

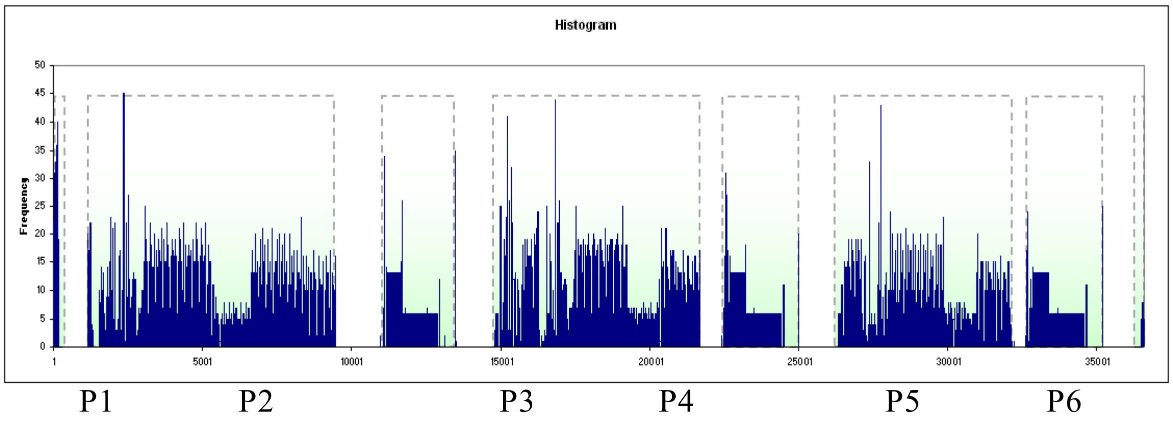


Figure 6.8. Detected phases in JHotDraw trace

The results are shown in the form of a histogram, where the x-axis shows the distance between the positions of the calls and the y-axis shows the frequency (the number of methods that their position falls into one interval of x-axis). As part of our technique, in order to automatically determine the number of phases and their location, the trace resulted from applying the integrated gravity scheme is partitioned by K-means clustering for *K* from 1 to 10. The highest BIC score was for the partitioning with *K* = 8 as the best fit. Figure 6.8 shows the location of the eight clusters (P1 to P8), highlighted by dashed rectangles.

These phases are explained in Table 6.6. Each row contains the location of the phase in the execution trace, the task performed in the phase, and the corresponding stratum. This information was used for stratification. As shown in this table, we were able to use our phase detection technique to successfully recover parts of the trace that implement each of the features that were traced.

Table 6.6. Execution phases that were detected

Phase	**Phase Location**	**Description**	**Strata**	**Size**
P1	1 – 1134	Initialization	Stratum 1	1134
P2	1135 – 10948	New sheet, F1, F2	Stratum 2	9814
P3	10949 – 14816	F3, F4	Stratum 3	3868
P4	14817 – 22391	F5, F6	Stratum 4	7575
P5	22392 – 26298	F7, F8	Stratum 5	3907
P6	26299 – 32812	F9, F10	Stratum 6	6514
P7	32813 – 36461	F11, F12	Stratum 7	3649
P8	36462 – 36571	Finalization	Stratum 8	110

We chose to generate three sampled traces from the original trace that vary in size and assess their effectiveness in being representatives of the original traces. The three sampled traces are respectively of sizes 3,646, 1,829, and 365, shown as Strat1, Strat2, and Strat3 in Table 6.7. In this table, the sampling parameter $|Stratum_h| / |T|$ for each stratum is calculated according to the size of the stratum $Stratum_h$ and the size of original trace ($|T|$ =36,571). Then, the sampling parameter and $|T'|$ the sample size are used for calculation of the number of calls to be sampled from each stratum (i.e., $|S_h|$). For instance, we can see that to obtain a sample trace of size 365 we need to randomly sample 12 calls from "Stratum 1" of the original trace, 96 calls from "Stratum 2" and so on.

The next step is to assess the representativeness of the three samples with respect to the original trace. We compared the sample traces generated by our approach to the ones generated using mere random sampling. For this purpose, we first created three other sampled traces (Rand1, Rand2, and Rand3) of sizes 3,646, 1,829, and 365 using random sampling from our original trace.

As mentioned earlier, a sampled trace is representative if it closely resembles the execution trace from which it is drawn. This resemblance could be quantified by the extent of similarity of statistical characteristics between the original trace and each set of generated samples. We consider the distribution of features and their contribution to the overall size of the original execution trace as our comparison reference. The closer the distribution of features in sample traces is to the actual distribution of features in the original trace, the more representative the sampled trace is.

Table 6.7. Contribution of features of JHotDraw trace to the size of the sample traces using both stratified and random sampling

	10% of Orig. Size (**3646**)		5% of Orig. Size (**1829**)		1% of Orig. Size (**365**)	
	Strat 1	Rand 1	Strat2	Rand2	Strat3	Rand 3
Initialization	113	115	57	51	12	5
NewSheet	47	47	22	14	4	6
F1	31	31	14	13	3	2
F2	915	914	460	464	90	98
F3	420	418	211	225	46	47
F4	30	35	15	12	3	4
F5	13	11	7	8	1	2
F6	674	672	339	354	63	55
F7	14	11	6	9	1	1
F8	381	368	191	197	39	36
F9	15	8	6	6	1	1
F10	598	653	308	308	63	74
F11	21	14	8	4	1	15
F12	362	342	182	157	37	14
Finalization	11	7	6	7	1	0

The distribution of features in each sample trace in terms of their contribution to the size of the sample trace is shown in Table 6.7. For instance, the number of calls belonging to feature F12 in the sample trace "Strat 3" is 37; its contribution to the size of "Strat 3" (i.e., 365) is 10.13%. The same feature contributes only 14 calls (3.83%) to the sampled trace "Rand 3", generated using random sampling. When we contrast this with the contribution of the features to the size of the entire trace (shown in Table 6.9), we can see that feature

F12 represents 10.01% of the size of the original. Therefore, F12 is better represented in "Strat 3" than in "Rand 3".

Table 6.8. The samples obtained from applying our approach to JHotDraw trace

Strata	N_h	$\lvert Stratum_h \rvert / \lvert T \rvert$	$\lvert S_h \rvert$ ($\lvert T' \rvert = \mathbf{365}$)	$\lvert S_h \rvert$ ($\lvert T' \rvert = \mathbf{1829}$)	$\lvert S_h \rvert$ ($\lvert T' \rvert = \mathbf{3646}$)
1	1134	0.031	12	57	113
2	9815	0.265	96	492	983
3	3868	0.105	39	193	384
4	7575	0.207	76	379	756
5	3907	0.106	39	194	387
6	6514	0.178	65	326	650
7	3649	0.099	37	182	362
8	109	0.003	1	6	11

Table 6.9. Contribution of features to the size of the entire trace

Initialization	3.101%
NewSheet	1.233%
F1	0.845%
F2	25.094%
F3	11.731%
F4	0.727%
F5	0.353%
F6	18.496%
F7	0.325%
F8	10.454%
F9	0.358%
F10	16.486%
F11	0.481%
F12	10.019%
Finalization	0.298%
TOTAL	**100%**

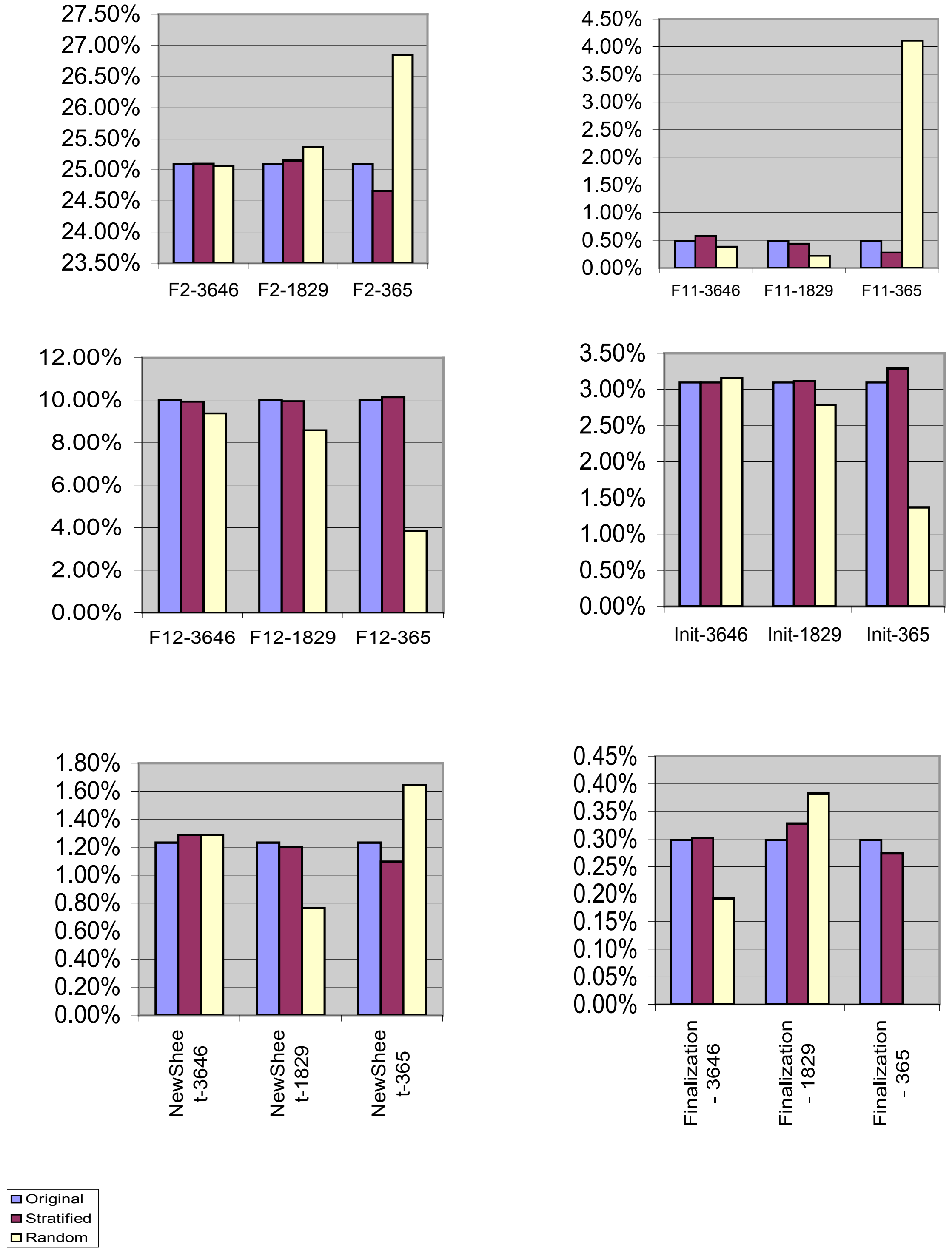


Figure 6.9. Comparison between the distribution of a number of features in the original trace, stratified sample, and random sample. The x axis shows the contribution of the feature to the size of the trace in percentage. The y axis shows the feature name and the sample size.

When we applied the same reasoning to all features, we found that our approach provided better results in more than 80% of the cases and in all these cases it led to a more representative sample trace. Furthermore, as the size of the sample decreases, our approach maintains its representativeness while random sampling, as theoretically discussed in Section 6.2, leads to cases that are significantly unrepresentative of the original trace. Some of these cases are shown in Figure 6.9. For instance, as it can be seen in Figure 6.9 the trace "Rand 3" contains a sudden high number of calls from feature F11 (shown as the rightmost yellow bar) while the percentage of the presence of the same feature in all the traces generated in our approach (the three plum bars) is maintained close the its percentage in the original trace (the blue bars). In all cases reported in Figure 6.9, our approach maintains the same distribution of features with respect to the original trace, which is not the case for random sampling. This demonstrates (at least for this case study) the superiority of our approach compared to random sampling.

6.6. Threats to Validity

In our case studies, we used our trace segmentation approach to find the execution phases of our traces. These phases are used by our smart sampling approach as strata to perform stratified sampling. Changing the phase detection technique can potentially result in the detection of a different number of phases (of different sizes). Therefore, we need to investigate how a different phase detection algorithm can impact the results of our sampling technique.

Another limitation of our approach is that the sampling is performed in a post-mortem manner, i.e., after the trace is generated and saved. This type of sampling contradicts the common application of sampling which consists of sampling a trace while it is being generated. Future work should focus on improving the algorithm to sample the traces on the fly.

During the second case study, we had to remove some mouse movement events because they cluttered the trace. However, we did not attempt to remove all low-level utilities, an activity which might be needed when we generalize our approach and apply it to other systems. In general, we need to study the impact of removing utilities on the final sample before the phase detection algorithm is applied.

6.7. Summary

A major shortcoming of existing trace sampling approaches is that they cannot guarantee that the resulting sampled trace will be representative of the original trace. This shortcoming can be attributed to these techniques being blind to the information contained in the trace.

In this chapter, we presented a novel sampling technique of large execution traces that not only reduces the size of traces but also generates sampled traces that are representative of the original traces. We proposed an approach that guaranties that a certain number of events from each homogeneous part of an execution trace will be included traces sampled from the original trace.

Our approach relied on stratified sampling, using execution phases as strata. For this, we used the phases resulted from our proposed trace segmentation algorithm. Then, we performed random sampling within each phase and drew a number of events as the sample events of that phase. The number of events drawn from each phase is proportional to the size of that phase.

We evaluated our proposed phase-based stratified sampling approach in two case studies. The first case study evaluated the usefulness of our proposed stratified sampling in comparison with random sampling in terms of program comprehension. The second case study showed a real-world running example of the problematic case and showed how our proposed approach compares to random sampling of execution traces.

Chapter 7. Conclusion

In order to perform any type of maintenance tasks software engineers first need to have some sort of understanding about the system that they are going to modify. Understanding software systems is a challenging task that consumes most of the time and efforts assigned to a maintenance task.

Development of trace analysis techniques and tools holds real potential to address the software maintenance problem. In this type of analysis, a system's runtime behaviour commonly presented in forms of execution traces is used for understanding the software system. Traces, however, tend to contain large amounts of data that pose a real obstacle to any viable analysis.

In this thesis, our contribution includes a set of novel approaches for trace analysis inspired by the way the human brain and perception system operate when dealing with information received through the visual sense. We review these contributions in the following section.

7.1. Research Contributions

Trace Abstraction Framework

We presented a novel framework for trace analysis and abstraction by drawing parallels between trace analysis and the human perception system. The proposed framework is composed of three components that are intended to perform trace segmentation, smart sampling, and content prioritization. Inspired by scene segmentation, the goal of the first component is to segment traces into meaningful and homogeneous segments to provide users with more structure data that should help in better exploration and easier analysis of execution trace. Inspired by the process of finding the gist of scenes, the second component aims to use the trace homogeneous segments to perform smart sampling. In this type of sampling, the different segments of trace are taken into account during the sampling process to generate a representative sample. Inspired by the pop-out effect in human perception, the goal of content prioritization component is to indentify events that are relevant to different parts of a trace. The ranking is based on the frequency of occurrence of each event in a phase and the number of other phases in which the same event has occurred.

Trace Segmentation

We presented a new trace analysis technique that automatically divides the content of a trace into smaller and meaningful trace segments. These segments correspond to the system's main execution phases. Presenting an execution trace as a flow of phases can facilitate trace exploration by software engineers and provide structure to the content of the trace which in turn can be used for more advanced approaches of trace analysis.

In our trace segmentation, we used the concepts of similarity and good continuation of Gestalt laws to measure the extent by which trace events can be grouped into dense clusters that indicate the presence of execution phases. We applied our approach to two software systems and the results are very satisfactory.

Stratified Sampling of Execution Traces

Using sampling techniques is an easy way for reducing the size of execution traces. Commonly used sampling methods such as random sampling may result in sampled traces that are not representative of the original trace. We proposed a stratified sampling method that not only reduces the size of a trace but also results in a sample that is representative of the original trace.

In our proposed approach, we use the homogeneous trace segments that are detected through our trace segmentation approach as strata. Then we perform random sampling to select events from within each segment based on the size of the segment. This way, we ensure that the desired characteristics of an execution are distributed similarly in both the sampled and the original trace. We showed the effectiveness of our sampling technique through two case studies.

Identification of Relevant Events Based on Text Mining

Different events of an execution trace are not equally important. Some events in a certain part of a trace can show what is being delivered in that part. Thus, identifying the most relevant events of different parts of a trace should help software engineers in finding the parts that they are interested in.

We proposed a trace analysis approach that automatically identifies relevant information about each execution phase of a trace. For this identification, we use TF-IDF as a known ranking technique used in text mining to find the events that are frequently invoked in a phase but not shared between many phases. These events are relevant event of that phase. Once the relevant events of each phase are identified, they are used to find similar phases. Finding the similarity of phases gives us the ability to provide an efficient representation of the flow of phases by detecting redundant phases. We applied our approach to traces generated from two different systems and we were able to quickly understand their content and extract higher-level views that characterize the essence of the information conveyed in these traces.

7.2. Opportunities for Further Research

7.2.1. Trace Analysis Framework

Various studies in the fields of psychology and neurophysiology have found that the human perception system analyzes the visual information in a scene through two types of processes: preattentive and attentional processes. Segmenting local elements against their context and integrating them as objects and regions is among the operations that occur during the preattentive process. The information returned by preattentive processes delivers the units for which attention can be allocated for further, more elaborated, processing handled by attentional processes.

While preattentive processes are built-in and standard, attentional processes, since they can occur on a voluntary level, may be equipped with learning strategies to lead to better results. Successful analysis of a scene is therefore dependent on successes of all three components of preattentive processes, attentional processes, and learning strategies.

In this thesis, we only investigated the preattentive process. One possible improvement is to augment our proposed framework in a way that all three groups of techniques (similar to the three discussed components in human perception) work together to result in better software behavioural analysis tools that can help understand various aspects of a running system.

For example, we can work investigate the second group of techniques (similar to attentional operations) along with guidelines from the field of educational psychology that explain how technologies --such as multimedia and interactive tools--can be used to foster understanding and learning of new subjects in human. The proposed techniques should adhere to the following learning strategies:

- Selecting: important information of an execution trace needs to be selected from less important (this is similar to selection strategy in educational psychology where less important information is excluded so as not to overload attention and working-memory of the user).

- Organizing: selected information has to be organized in a way that makes it easier to process for the user. This might be done by providing a textual outline of the selected information or by graphically presenting it through a suitable representation method such as a map (similar to organizing theory of active learning information needs to be presented such that user can mentally organize the presented material into a coherent structure).

- Associating: possible associations (e.g., dependency, co-occurrence) among organized information need to be presented. This could be done graphically (e.g., by highlighting, color-coding, or drawing arcs) to show the connection between the organized information (similar to integration strategy in education psychology).

7.2.2. Trace Segmentation

Our trace segmentation approach tries to find execution phases by investigating the changes that occur in the flow of events in an execution trace. These changes are currently of two types of measures: changes in the naming similarity of the events, changes in the continuity of the nesting level of events. One direction of future work is to investigate how changes of other measures can affect the phase detection. For example, significant changes in the elapsed time of method calls might suggest a new phase. This change can be mapped to our continuation scheme.

As mentioned earlier, in this thesis, we only considered the exact naming similarity of events in our similarity scheme for its simplicity and low overhead. The result might be improved if a more flexible type of similarity is used. For example, right now, the similarity of two method calls has a two-valued logic (i.e., it is whether 1 or 0). This could be improved by introducing a truth-value logic (that ranges in degree between 0 and 1). For this, one may use text cleaning and pre-processing like stemming. Other types of similarities, for example, based on analyzing the source code could also be used to find similarity of events as long as they result in a reasonable overhead.

In our segmentation approach, we have a clustering and mapping component that tries to find the beginning and end of each group of method calls and map the group to the original trace as an execution phase. In this component, we use K-mean clustering as our choice of clustering algorithm. This was due to simplicity, good speed, and availability of the K-means algorithm. However, other clustering algorithm might improve the results of clustering. For example, density based clustering is a potential candidate to be investigated. In general, any clustering algorithm that is automatic (does not have many parameters to be set by the user) and has a reasonable overhead could be a good choice for further exploration.

In our phase detection, we have a threshold *t* that provides the user with the opportunity to find the sub-phases of a major phase. We anticipate that this threshold is application-specific. However, providing users with some hints or suggestions might help the user to find an appropriate threshold more quickly. A useful future avenue, therefore, is developing heuristics to help in finding good thresholds.

Finally, the phases that are detected in our approach can help predict execution phases at runtime. Phase prediction can be used for optimizing the management of system resource while the system is executing.

7.2.3. Trace Sampling

At the theoretical level, our trace sampling method was inspired by the way our perception system achieves the gist of a scene. It has been suggested that scene elements with low spatial frequency are the ones that contribute most to building a gist. In this thesis, we used stratified sampling to find a sample of a trace. One possible future direction is to adapt image processing approaches that find elements of low spatial frequency in an image. One can adapt those approaches to the field of trace analysis for finding representative samples.

Our trace sampling method performs stratified sampling of execution traces by using the execution phases as strata. In this thesis, we used the phases resulted from our phase detection approach. Since the phase detection proposed in this thesis works in an offline manner, that the sampling would be also performed in a post-mortem manner, i.e., after the trace is generated and saved. Some approaches might need to perform trace sampling in an online manner. Therefore, one direction for future work would be to explore ways in which our approach can be adapted to online sampling where we can sample a trace while it is being generated.

7.2.4. Content Prioritization

An immediate direction to continue our content prioritization approach is to use the relevant events of each phase to generate a human readable description for that phase. For this, one can explore text mining approaches to generate summaries for each relevant event from available sources and eventually from a description for each phase.

7.2.5. General Directions

In this thesis, we evaluated our techniques through a number of case studies. Almost all of the case studies we performed in this thesis were of an intrinsic nature. That is, we performed our validations according to the documentations provided by the original developers and maintainers of open source systems.

Other investigations through extrinsic evaluations (e.g., controlled experiments) can be done to refine the validation of our techniques. We need to plan a controlled experiment where we can asses the impacts of our techniques on program comprehension.

We are integrating the techniques proposed in this thesis into a tool. We are also working on developing a new type of diagram called a phase flow diagram to representation the flow of phases of an execution trace. This diagram is equipped with useful information about phases such as relevant events and so on. We also like to investigate the usefulness of our proposed approaches in other software engineering and maintenance tasks such as redocumentation and testing.

References

[AAG10] Asadi, F., Antoniol, G., & Guéhéneuc, Y.-G. (2010). Concept Location with Genetic Algorithms: A Comparison of Four Distributed Architectures. 2nd International Symposium on Search Based Software Engineering (pp. 153-162). IEEE. doi:10.1109/SSBSE.2010.26.

[ACC+02] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering, 28(10), (pp. 970-983). doi:10.1109/TSE.2002.1041053

[ADAG10] Asadi, F., Di Penta, M., Antoniol, G., & Guéhéneuc, Y.-G. (2010). A Heuristic-Based Approach to Identify Concepts in Execution Traces. 2010 14th European Conference on Software Maintenance and Reengineering (pp. 31-40). IEEE. doi:10.1109/CSMR.2010.17

[ADS09] Abeel, T., Van de Peer, Y., & Saeys, Y. (2009). Java-ML: A Machine Learning Library. The Journal of Machine Learning Research, 10, (pp. 931-934). Retrieved from http://dl.acm.org/citation.cfm?id=1577069.1577103

[AG05] Antoniol, G., & Gueheneuc, Y.-G. (2005). Feature identification: a novel approach and a case study. 21st IEEE International Conference on Software Maintenance (ICSM'05) (pp. 357-366). IEEE. doi:10.1109/ICSM.2005.48

[Alv11] A Alvarez, G. (2011). Representing multiple objects as an ensemble enhances visual cognition. Trends in Cognitive Sciences, 15(3), (pp. 122-131). Elsevier Ltd.

Retrieved from http://www.mendeley.com/research/representing-multiple-objects-ensemble-enhances-visual-cognition/

[ARG] ArgoUML, URL: argouml.tigris.org

[Bal99] Ball, T. (1999). The concept of dynamic analysis. ACM SIGSOFT Software Engineering Notes, 24(6), (pp. 216-234). doi:10.1145/318774.318944

[Bar04] Bar, M. (2004). Visual objects in context. Nature reviews. Neuroscience, 5(8), (pp. 617-29). doi:10.1038/nrn1476

[Bas97] Basili, V. R. (1997). Evolving and packaging reading technologies. Journal of Systems and Software, 38(1), (pp. 3-12). doi:10.1016/S0164-1212(97)00065-4

[BCLW69] Bower, G., Clark, M., Lesgold, A., & Winzen, D. (1969). Hierarchical Retrieval Schemes in Recall of Categorized Word Lists. Journal of Verbal Learning and Verbal Behavior, 8, (pp. 323 – 343). Retrieved from http://www.citeulike.org/user/trisha/article/3070185

[BCS04] Bladh, T., Carr, D. A., & Scholl, J. (2004). Extending Tree-Maps to Three Dimensions: A Comparative Study. Proceedings of the 6th Asia-Pacific Conference on Computer-Human Interaction, 2, (pp. 50 – 59). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.1835

[Boo02] Boothe, R. G. (2002). Perception of the Visual Environment (p. 407), Springer-Verlag, New York, 13, ISBN: 0387987908.

[BRBP90] Bowers, K. S., Regehr, G., Balthazard, C., & Parker, K. (1990). Intuition in the context of discovery. Cognitive Psychology, 22(1), 72-110. doi:10.1016/0010-0285(90)90004-N

[Bru60] Brunk, H. (1960). An introduction to mathematical statistics. Boston: Ginn. Retrieved from http://0-www.worldcat.org.patris.apu.edu/title/introduction-to-mathematical-statistics/oclc/527286?referer=list_view

[CDMZ07] Cornelissen, B., van Deursen, A., Moonen, L., & Zaidman, A. (2007). Visualizing Testsuites to Aid in Software Understanding. 11th European Conference on Software Maintenance and Reengineering (CSMR'07) (pp. 213-222). IEEE. doi:10.1109/CSMR.2007.54

[CHMY03] Chan, A., Holmes, R., Murphy, G. C., & Ying, A. T. T. (2003). Scaling an object-oriented system execution visualizer through sampling. MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717) (pp. 237-244). IEEE Comput. Soc. doi:10.1109/WPC.2003.1199207

[CHZ+07] Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J. J., & van Deursen, A. (2007). Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. 15th IEEE International Conference on Program Comprehension (ICPC '07) (pp. 49-58). IEEE. doi:10.1109/ICPC.2007.39

[CMZ08] Cornelissen, B., Moonen, L., & Zaidman, A. (2008). An assessment methodology for trace reduction techniques. 2008 IEEE International Conference on Software Maintenance (pp. 107-116). IEEE. doi:10.1109/ICSM.2008.4658059

[Coc77] Cochran, W. G. (1977). Sampling Techniques (p. 448). Wiley; 3rd edition. Retrieved from http://www.amazon.ca/Sampling-Techniques-William-G-Cochran/dp/047116240X/ref=sr_1_1?s=books&ie=UTF8&qid=1329814822&sr=1-1

[Con80] Conover, W. (1980). Practical nonparametric statistics (p. 462), New York: Wiley; 2nd edition, Retrieved from http://www.worldcat.org/title/practical-nonparametric-statistics/oclc/005946525

[Cor89] Corbi, T. A. (1989). Program understanding: Challenge for the 1990s. IBM Systems Journal, 28(2), (pp. 294-306). doi:10.1147/sj.282.0294

[Cow05] Cowan, N. (2005). Working memory capacity (Google eBook) (p. 246). Hove, England: Psychology Press, Retrieved from http://books.google.com/books?hl=en&lr=&id=30D7fjmnDr8C&pgis=1

[CR00] Chen, K., & Rajlich, V. (2000). Case Study of Feature Location Using Dependence Graph, 2000 8th International Workshop on Program Comprehension, IWPC'00, (pp. 241-249). Retrieved from http://dl.acm.org/citation.cfm?id=518049.856972

[CZH+08] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk, (2008). Execution trace analysis through massive sequence and circular bundle views. Journal of Systems and Software, 81(12), (pp. 2252-2268). doi:10.1016/j.jss.2008.02.068

[CZD+09] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., & Koschke, R. (2009). A Systematic Survey of Program Comprehension through Dynamic Analysis. IEEE Transactions on Software Engineering, 35(5), (pp. 684-702). doi:10.1109/TSE.2009.28

[CZD11] Cornelissen, B., Zaidman, A., & van Deursen, A. (2011). A Controlled Experiment for Program Comprehension through Trace Visualization. IEEE

Transactions on Software Engineering, 37(3), (pp. 341-355). doi:10.1109/TSE.2010.47

[DHKV93] De Pauw, W., Helm, R., Kimelman, D., & Vlissides, J. (1993). Visualizing the behavior of object-oriented systems. Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications - OOPSLA '93, vol. 28, (pp. 326-337). New York, New York, USA: ACM Press. doi:10.1145/165854.165919

[DJM+02] Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., & Yang, J. (2001). Visualizing the Execution of Java Programs, Revised Lectures on Software Visualization, international Seminar, LNCS, vol. 2269. (pp. 151-162). Retrieved from http://dl.acm.org/citation.cfm?id=647382.724791

[DRW00] Dunsmore, A., Roper, M., & Wood, M. (2000). The role of comprehension in software inspection. Journal of Systems and Software, 52(2-3), (pp. 121-129). doi:10.1016/S0164-1212(99)00138-7

[Dug07] Dugerdil, P. (2007). Using trace sampling techniques to identify dynamic clusters of classes. Proceedings of the 2007 conference of the center for advanced studies on Collaborative research - CASCON '07 (pp. 306-316). New York, New York, USA: ACM Press. doi:10.1145/1321211.1321254

[ED05] Eisenberg, A. D., & De Volder, K. (2005). Dynamic feature traces: finding features in unfamiliar code. 21st IEEE International Conference on Software Maintenance (ICSM'05) (pp. 337-346). IEEE. doi:10.1109/ICSM.2005.42

[EKS03] Eisenbarth, T., Koschke, R., & Simon, D. (2003). Locating features in source code. IEEE Transactions on Software Engineering, 29(3), (pp. 210-224). doi:10.1109/TSE.2003.1183929

[ER84] Eylon, B.-S., & Reif, F. (2007). Effects of Knowledge Organization on Task Performance. Lawrence Erlbaum Associates. Cognition and Instruction, 1, (pp. 5-44), Retrieved from http://www.jstor.org/pss/3233519

[ES98] Erdos, K., & Sneed, H. M. (1998). Partial comprehension of complex programs (enough to perform maintenance). Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (pp. 98-105). IEEE Comput. Soc. doi:10.1109/WPC.1998.693322

[EXT] EXTRAVIS: http://www.swerl.tudelft.nl/extravis/

[FRC10] Frintrop, S., Rome, E., & Christensen, H. I. (2010). Computational visual attention systems and their cognitive foundations. ACM Transactions on Applied Perception, 7(1), (pp. 1-39). doi:10.1145/1658349.1658355

[GD05] Greevy, O., & Ducasse, S. (2005). Correlating Features and Code Using a Compact Two-Sided Trace Analysis Approach. Ninth European Conference on Software Maintenance and Reengineering (pp. 314-323). IEEE. doi:10.1109/CSMR.2005.21

[Gen83] Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. Cognitive Science, 7(2), (pp. 155-170). doi:10.1016/S0364-0213(83)80009-3

[GMSS00] Ghosh, A. K., Michael, C., & Schatz, M. (2000). A Real-Time Intrusion Detection System Based on Learning Program Behavior, 2000 the 3rd International

Workshop on Recent Advances in Intrusion Detection, Toulouse, France, (pp. 93-109). Retrieved from http://dl.acm.org/citation.cfm?id=645838.670724.

[GPSG01] Geisler, W. S., Perry, J. S., Super, B. J., & Gallogly, D. P. (2001). Edge co-occurrence in natural images predicts contour grouping performance. Vision research, 41(6), (pp. 711-24). Retrieved from http://www.ncbi.nlm.nih.gov/pubmed/11248261

[GRT93] Grabowecky, M., Robertson, L. C., & Treisman, A. (1993). Preattentive Processes Guide Visual Search: Evidence from Patients with Unilateral Visual Neglect. Journal of Cognitive Neuroscience, 5(3), (pp. 288-302). doi:10.1162/jocn.1993.5.3.288

[HBAL05] Hamou-Lhadj, A., Braun, E., Amyot, D., & Lethbridge, T. (2005). Recovering Behavioral Design Models from Execution Traces. Ninth European Conference on Software Maintenance and Reengineering (pp. 112-121). IEEE. doi:10.1109/CSMR.2005.46

[HHHL03] Heuzeroth, D., Holl, T., Hogstrom, G., & Lowe, W. (2003). Automatic design pattern detection. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (pp. 94-103). IEEE Comput. Soc. doi:10.1109/WPC.2003.1199193

[HKM11] Han, J., Kamber, M., & Pei, J., (2011). Data mining: concepts and techniques (p. 744), Morgan Kaufmann, 3rd edition. ISBN: 0123814790.

[HL02] Hamou-Lhadj, A., & Lethbridge, T. C. (2002). Compression techniques to simplify the analysis of large execution traces. Proceedings 10th International

Workshop on Program Comprehension (pp. 159-168). IEEE Comput. Soc. doi:10.1109/WPC.2002.1021337

[HL06] Hamou-Lhadj, A., & Lethbridge, T. (2006). Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. 14th IEEE International Conference on Program Comprehension (ICPC'06) (pp. 181-190). IEEE. doi:10.1109/ICPC.2006.45

[IEE98] IEEE Std. 1219-1998, Standard for Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 1998.

[Java] Oracle Corporation, Inc. Java Platform, Standard Edition 7.0 API Specification. Available at: http://docs.oracle.com/javase/7/docs/api/, 2011.

[JHO] JHOTDRAW, http://www.jhotdraw.org/

[Jin01] Jinsight Reference Manual, available at: http://www.research.ibm.com/jinsight/docs/refman/refman.htm

[Joa98] Joachims, T. (1998). Text Categorization with Support Vector Machines: Learning with Many Relevant Features. In C. Nédellec & C. Rouveirol (Eds.), Machine Learning ECML98, vol. 1398, (pp. 2-7). Berlin/Heidelberg: Springer-Verlag. doi:10.1007/BFb0026664

[JR97] Jerding, D., & Rugaber, S. (1997). Using visualization for architectural localization and extraction. Proceedings of the Fourth Working Conference on Reverse Engineering (pp. 56-65). IEEE Comput. Soc. doi:10.1109/WCRE.1997.624576

[JS11] Japkowicz, N., & Shah, M. (2011). Evaluating Learning Algorithms: A Classification Perspective (p. 424). Cambridge University Press. Retrieved from http://www.amazon.ca/Evaluating-Learning-Algorithms-Classification-Perspective/dp/0521196000

[JS91] Johnson, B., & Shneiderman, B. (1991). Tree-maps: a space-filling approach to the visualization of hierarchical information structures. Proceeding Visualization '91 (pp. 284-291). IEEE Comput. Soc. Press. doi:10.1109/VISUAL.1991.175815

[KG06] Kuhn, A., & Greevy, O. (2006). Exploiting the Analogy Between Traces and Signal Processing. 2006 22nd IEEE International Conference on Software Maintenance (pp. 320-329). IEEE. doi:10.1109/ICSM.2006.29

[Kof99] Koffka, K. , (1999), Principles of Gestalt Psychology (p. 732). Hartcourt, New York, Routledge reprint. ISBN: 0415209625.

[LAZJ03] Liblit, B., Aiken, A., Zheng, A. X., & Jordan, M. I. (2003). Bug Isolation via Remote Program Sampling. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (pp. 141 - 154). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.6113

[Mac67] MacQueen, j., (1976), Some Methods for Classification and Analysis of Multivariate Observations. Proceedings of the 5th Berkeley Symposium on Math Statistics and Probability, (pp. 281-296).

[Mil56] Miller, G. A. (1956) The magical number seven or minus two: Some limits on our capacity of processing information.The Psychological Review, vol. 63, (pp. 81–97).

[MRS08] Manning, C. D., Raghavan, P., Schtze, H., (2008), Introduction to Information Retrieval (p.482). Cambridge University Press, New York, NY, USA, ISBN: 0521865719.

[Mus31] Musatti, C. L, (1931), Forma e assimilazione. Archivio Italiano di Psicologia. 9: (pp. 61-156).

[MWM06] McGavin, M., Wright, T., & Marshall, S. (2006). Visualisations of execution traces (VET): an interactive plugin-based visualisation tool, the 7th Australasian User interface Conference (pp. 153-160). Retrieved from http://dl.acm.org/citation.cfm?id=1151758.1151780

[Oli05] Oliva, A., (2005) Gist of the scene. Neurobiology of Attention, L. Itti, G. Rees & J. K. Tsotsos (Eds.), (pp 251–256).

[OT06] Oliva, A., & Torralba, A., (2006) Building the gist of a scene: The role of global image features in recognition. Visual Perception, Progress in Brain Research, 155(1): (pp. 23–26).

[PAH10] Pirzadeh, H., Agarwal, A., & Hamou-Lhadj, A. (2010). An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension. 2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications (pp. 207-214). IEEE. doi:10.1109/SERA.2010.34

[Pen87] Pennington, N. (1987). Comprehension strategies in programming, G. M. Olson, S. Sheppard and E. Soloway (eds) Empirical Studies of Programmers, Second Workshop (Norwood, NJ: Ablex), (pp. 100-113). Retrieved from http://dl.acm.org/citation.cfm?id=54968.54975

[PGM+07] Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., & Rajlich, V. (2007). Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. IEEE Transactions on Software Engineering, 33(6), (pp. 420-432). doi:10.1109/TSE.2007.1016

[PH11a] Pirzadeh, H., & Hamou-Lhadj, A. (2011). A software behaviour analysis framework based on the human perception systems. Proceeding of the 33rd International Conference on Software Engineering (ICSE'11) New Ideas and Emerging Results Track, (pp. 948–951). ACM Press. doi:10.1145/1985793.1985955

[PH11b] Pirzadeh, H., & Hamou-Lhadj, A. (2011). A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension. 2011 16th IEEE International Conference on Engineering of Complex Computer Systems (pp. 221-230). IEEE. doi:10.1109/ICECCS.2011.29

[PHS11] Pirzadeh, H., Hamou-Lhadj, A., & Shah, M. (2011). Exploiting text mining techniques in the analysis of execution traces. 2011 27th IEEE International Conference on Software Maintenance (ICSM'11), (pp. 223 – 232). Retrieved from http://resolver.scholarsportal.info/resolve/10636773/v2011inone/223_etmtitaoet.xml

[PM00] Pelleg, D., & Moore, A. (2000). X-means: Extending K-means with Efficient Estimation of the Number of Clusters. Proceedings of the Seventeenth International Conference on Machine Learning, (pp. 727-734). Morgan Kaufmann. Retrieved from http://www.mendeley.com/research/xmeans-extending-kmeans-with-efficient-estimation-of-the-number-of-clusters/

[Pri10] Price, R. J., (2010). Automatic stop word identification and compensation, US Patent, 7, (pp. 720-792).

[PSHM11] Pirzadeh, H., Shanian, S., Hamou-Lhadj, A., & Mehrabian, A. (2011). The Concept of Stratified Sampling of Execution Traces. 2011 IEEE 19th International Conference on Program Comprehension (ICPC'11) (pp. 225-226). IEEE. doi:10.1109/ICPC.2011.17

[QB09] Quinn, P. C., & Bhatt, R. S. (2009). Perceptual organization in infancy: bottom-up and top-down influences. Optometry and vision science : official publication of the American Academy of Optometry, 86(6), (pp. 589-94). doi:10.1097/OPX.0b013e3181a5238a

[RC05] Rountev, A., & Connell, B. H. (2005). Object naming analysis for reverse-engineered sequence diagrams. Proceedings. 27th International Conference on Software Engineering, 2005. (ICSE'05). (pp. 254-263). IEEE. doi:10.1109/ICSE.2005.1553568

[Rei05] Reiss, S. P. (2005). Dynamic detection and visualization of software phases. Proceedings of the third international workshop on Dynamic analysis (WODA '05), (pp. 1-6). New York, New York, USA: ACM Press. doi:10.1145/1083246.1083254

[Rei07] Reiss, S. P. (2007). Visual representations of executing programs. Journal of Visual Languages & Computing, 18(2), (pp. 126-148). doi:10.1016/j.jvlc.2007.01.003

[RHR08] Rohatgi, A., Hamou-Lhadj, A., & Rilling, J. (2008). An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. 2008 The 16th IEEE

International Conference on Program Comprehension (pp. 236-241). IEEE. doi:10.1109/ICPC.2008.35

[Ros99] Rosenholtz, R. (1999). A simple saliency model predicts a number of motion popout phenomena. Vision research, 39(19), 31, (pp. 57-63). Retrieved from http://www.ncbi.nlm.nih.gov/pubmed/10615487

[RR01] Reiss, S. P., & Renieris, M. (2001). Encoding program executions. Proceedings of the 23rd International Conference on Software Engineering. (ICSE'01) (pp. 221-230). IEEE Comput. Soc. doi:10.1109/ICSE.2001.919096

[RR99] Renieris, M., & Reiss, S. P. (1999). Almost: Exploring Program Traces. 1999 Workshop on New Paradigms in Information Visualization and Manipulation, 31, (pp. 70 – 77). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.8813

[SB88] Salton, G., & Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. Information Processing and Management, 24, (pp. 513 – 523). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.9086

[Sch78] Schwarz, G. (1978). Estimating the Dimension of a Model. The Annals of Statistics, 6(2), (pp. 461-464). Retrieved from http://projecteuclid.org/euclid.aos/1176344136

[SF99] Smith-Gratto, K., Fisher, M., (1999), Gestalt theory: A foundation for instructional screen design, Journal of Instructional Technology Systems, vol. 27, no. 4, (pp. 361–371).

[Shn96] Shneiderman, B. (1996). The eyes have it: a task by data type taxonomy for information visualizations. Proceedings 1996 IEEE Symposium on Visual Languages (pp. 336-343). IEEE Comput. Soc. Press. doi:10.1109/VL.1996.545307

[SKM01] Systä, T., Koskimies, K., & Muller, H. (2001). Shimba: an environment for reverse engineering Java software systems. Software: Practice and Experience, 31(4), (pp. 371-394). doi:10.1002/spe.386

[SO94] Schyns, P., & Oliva, A. (1994). From blobs to boundary edges: Evidence for time and spatial scale dependent scene recognition. Psychological Science, 5, (pp. 195 – 200). Retrieved from http://www.citeulike.org/user/swachsmu/article/2354810

[SPAM11] Silva, L. L., Paixao, K. R., Amo, S. de, & Maia, M. de A. (2011). On the Use of Execution Trace Alignment for Driving Perfective Changes. 2011 15th European Conference on Software Maintenance and Reengineering (pp. 221-230). IEEE. doi:10.1109/CSMR.2011.28

[SPHC02] Sherwood, T., Perelman, E., Hamerly, G., & Calder, B. (2002). Automatically characterizing large scale program behavior. the 10th international Conference on Architectural Support For Programming Languages and Operating Systems, ACM SIGPLAN Notices, 37(10), (pp. 45-57). doi:10.1145/605432.605403.

[SPSS] SPSS, Inc., 2009, Chicago, IL, www.spss.com

[SSW08] Smit, M., Stroulia, E., & Wong, K. (2008). Use Case Redocumentation from GUI Event Traces. 2008 12th European Conference on Software Maintenance and Reengineering (pp. 263-268). IEEE. doi:10.1109/CSMR.2008.4493323

[Sto06] Storey, M.-A. (2006). Theories, tools and research methods in program comprehension: past, present and future. Software Quality Journal, 14(3), (pp. 187-208). doi:10.1007/s11219-006-9216-4

[Sys00] Systa, T. (2000). Understanding the behavior of Java programs. Proceedings Seventh Working Conference on Reverse Engineering (pp. 214-223). IEEE Comput. Soc. doi:10.1109/WCRE.2000.891472

[TCA] Tolke, E. L., Klink, M., & Tolke, L., Cookbook for Developers of ArgoUML: An introduction to Developing ArgoUML. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.119.792, URL: http://argouml.tigris.org/

[TG80] Treisman A.M., Gelade, G. (1980) A Feature-Integration Theory of Attention, Cognitive Psychology, 12, (pp. 97–136).

[TH98] Tzerpos, V., & Holt, R. C. (1998). Software botryology. Automatic clustering of software systems. Proceedings Ninth International Workshop on Database and Expert Systems Applications (pp. 811-818). IEEE Comput. Soc. doi:10.1109/DEXA.1998.707499

[TPTP] Eclipse TPTP, "Eclipse Test & Performance Tools Platform Project," http://www.eclipse.org/tptp/.

[UKM06] Uchida, N., Kepecs, A., & Mainen, Z. F. (2006). Seeing at a glance, smelling in a whiff: rapid forms of perceptual decision making. Nature reviews. Neuroscience, 7(6), (pp. 485-91). doi:10.1038/nrn1933

[Vas07] Vassilvitskii, S, (2007), K-Means: Algorithms, Analyses, Experiments, Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Rajeev Motwani.

[VV95] Von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. IEEE Computer, 28(8), (pp. 44-55). doi:10.1109/2.402076

[WC99] Wolfe, J.M., Cave, K.R. (1999). The psychophysical evidence for a binding problem in human vision. Neuron 24, (pp. 11–17).

[WEKA] WEKA, URL: www.cs.waikato.ac.nz/ml/weka/

[Wer58] Wertheimer, M., (1958). Principles of perceptual organization. D. C. Beardslee, & M. Wertheimer (Eds.), Readings in perception. Princeton, NJ: Van Nostrand.

[WGH00] Eric Wong, W., Gokhale, S. S., & Horgan, J. R. (2000). Quantifying the closeness between program components and features. Journal of Systems and Software, 54(2), (pp. 87-98). doi:10.1016/S0164-1212(00)00029-7

[WII08] Watanabe, Y., Ishio, T., & Inoue, K. (2008). Feature-level phase detection for execution trace using object cache. 2008 international workshop on dynamic analysis held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008) - WODA '08 (pp. 8–14). New York, New York, USA: ACM Press. doi:10.1145/1401827.1401830,

[WKL+08] Wolfe, Jeremy M.; Kluender, Keith R.; Levi, Dennis M.; Bartoshuk, Linda M.; Herz, Rachel S.; Klatzky, Roberta L.; Lederman, Susan J. (2008). "Gestalt

Grouping Principles". Sensation and Perception (2nd ed.). Sinauer Associates. (pp. 78-80).

[WMF+98] Walker, R. J., Murphy, G. C., Freeman-Benson, B., Wright, D., Swanson, D., & Isaak, J. (1998). Visualizing dynamic software system information through high-level models. Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM SIGPLAN Notices, 33(10), (pp. 271-283). doi:10.1145/286942.286966

[WS95] Wilde, N., & Scully, M. C. (1995). Software reconnaissance: Mapping program features to code. Journal of Software Maintenance: Research and Practice, 7(1), (pp. 49-62). doi:10.1002/smr.4360070105

[ZD04] Zaidman, A., & Demeyer, S. (2004). Managing trace data volume through a heuristical clustering process based on event execution frequency. Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. (pp. 329-338). IEEE. doi:10.1109/CSMR.2004.1281435

[Zik00] Zikmund, W., (2000). Exploring marketing research. Applied Social Research Methods Series, vol. 5., 7th ed. Sage Publications, California, USA.

[ZL01] Zayour, I., & Lethbridge, T. C. (2001). Adoption of reverse engineering tools: a cognitive perspective and methodology. Proceedings 9th International Workshop on Program Comprehension. IWPC 2001 (pp. 245-255). IEEE Comput. Soc. doi:10.1109/WPC.2001.921735