Effective Segmentation of Large Execution Traces Using Probabilistic and Gaussian Mixture Models

Mohammad Reza Rejali

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science at Concordia University

Montreal, Quebec, Canada

April 2015

Mohammad Reza Rejali

Concordia University

This is to certify that the thesis prepared

By: Mohammad Reza Rejali

Entitled: **Effective Segmentation of Large Execution Traces Using Probabilistic and Gaussian Models**

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

Master of Applied Science (Electrical & Computer Engineering)

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

Signed by the final examining committee:

_____ Dr.M. Zahangir Kabir _____ Chair

_____ Dr.Jamal Bentahar _____ Examiner

_____ Dr.Otmane Ait Mohamed_____ Examiner

_____ Dr.Wahab Hamou-Lhadj_____ Supervisor

Approved by:

_____

Chair of Department or Graduate Program Director

ABSTRACT

Effective Segmentation of Large Execution Traces Using Probabilistic and Gaussian Mixture Models

Mohammad Reza Rejali

Software maintenance is known to be a costly and time consuming activity. Software engineers need to spend a considerable amount of time in understanding the system before maintaining it. This is due to many reasons including the lack of good documentation and the shift of the original developers of the system to other projects or companies.

Dynamic analysis techniques, more particularly trace analysis, are used to alleviate the program comprehension problem by offering software engineers a set of techniques that can help them understand the behavioural aspects of software systems.

Execution traces however can be extremely large, which makes them cumbersome for effective analysis. There is a need to develop techniques to help software engineers understand the content of large traces despite their massive size. In this thesis, we present, SumTrace, a novel trace analysis technique. SumTrace takes a trace as input and automatically segments it into smaller and more manageable groups that reflect the execution phases of the traced scenario. The execution phases are summarized to help software engineers understand quickly different parts of the trace without having to analyze its entire content. SumTrace relies on a combination of probabilistic and Gaussian mixture models.

We applied SumTrace to the segmentation of large traces, generated from two software systems. The results are very promising. SumTrace is also fast since it only requires only one pass through a trace.

Acknowledgment

First and for most, I would like to thank my supervisor, Dr. Abdelwahab Hamou-Lhadj, for the support and advices he gave me throughout this whole research. He has guided me when I needed to and complimented my works when they were well done. I appreciate the fact that he keeps his door always open for helpful feedback and conversation. More than anyone else, his influence has contributed to my development as a researcher.

The thesis contains many statistics. I would like to thank Dr. Abbas Khalili from the Department of Mathematics and Statistics of McGill University for his help. I appreciate it his kindly supervision and for taking his time to contribute to this research. Also I would like to thank, Dr. Syed Shariyar Murtaza, a postdoc, for his great supervision and also his help to evaluate the results of the case studies we conducted.

Additionally, I want to thank you my dear friend Omid Askari, a Software Developer at Max Planck Institute, for his contribution to some of the ideas presented in this thesis.

I would also like to thank the Faculty of Engineering and Computer Science, Concordia University as well as NSERC (Natural Science and Engineering Research Council Canada) for their financial support.

Most importantly, I would like to thank my parents, for their awesome support and encouragements during my whole life. I cannot thank them enough for all the sacrifices they made throughout this whole process. I could not have done this without them.

Finally, I would like to thank everyone at the Software Behaviour Analysis (SBA) Research Lab at Concordia University for their friendship and great encouragement.

Table of Contents

List of Figures

List of Tables

List of Equations

Chapter 1 - Introduction

1.1. Problem and Motivation

The first step for maintaining a software system is to understand how it is built and why it is built in a certain way. This understanding allows maintainers to perform software engineering activities such as debugging, adding new features to an existing system, and improving system performance. Many approaches have been proposed to understand the behaviour of software systems. There are two categories of software analysis techniques. The first one, static analysis, relies on examining the source code. Analyzing the source code to understand the dynamics of a system is a difficult task because maintainers may need to go through different parts of the system even though only parts of the system are affected. The second category, dynamic analysis, which is the focus of this thesis, operates on analyzing run-time information, such as execution traces. Unlike static analysis, dynamic analysis allows software maintainers to only focus on parts of the system that need to be examined. Dynamic analysis is also suitable when one needs to see how the system behaves given a certain input. This way, one can connect program output to program input.

Run-time information is represented typically in the form of execution traces. There are several types of execution traces such as routine (method) calls, statement traces, and inter-process communication traces. Routine call traces contain sequences of the invoked functions. Statement traces contain a list of statements in the source code. They tend to be extremely large, which explains why they are not used often in program comprehension. Traces of inter-process

communication depict communication among processes. In this thesis, we focus on traces of routine calls since routines are the main building blocks of programs.

Despite their usefulness, traces have been historically difficult to analyze, mainly due to their large size. To address this issue, trace abstraction techniques have been proposed (see [5] for a survey). The common objective is to reduce the size of traces and simplify their understanding for the human viewer, by extracting high-level views from raw traces. Although these techniques have shown to be useful, they are not designed to recover execution phases invoked in a trace. An execution phase can be defined as a set of cohesive trace events that implement a given computation. To make this clear, consider for example a trace generated from applying a classification algorithm in machine learning. This trace is bound to contain the typical computations of a classification algorithm including preprocessing data, building a training model (such as a decision tree), evaluating the model, visualizing the results, etc. Such a trace may contain hundreds of thousands of calls. Knowing where each of these phases occurs in the trace can help software engineers to focus on only that particular part of the trace that interests them instead of browsing the whole trace content.

Segmenting a trace into execution phases is usually a challenging task because there is no support at the programming language level of how to explicitly indicate the beginning and ending of each phase. There are not too many studies in the literature that address this problem either (see related work chapter). The few studies that exist either rely heavily on human intervention for setting various thresholds [18][19][20], or are tied to specific visualization methods [3][22]. In fact, trace segmentation is an emerging area of trace analysis research and there is clearly a need for more advanced (and automated) solutions.

In this thesis, we propose a novel trace segmentation technique, called SumTrace, which does not only generate meaningful phases from a trace, but also summarizes each phase. The summarized trace contains only the distinct functions invoked in the original trace. In other words, our approach turns a trace of hundreds of thousands of function calls into a few phases of hundreds of function calls that, as we will show in the case study, provide an accurate (and representative) high-level view of the implementation of the traced scenario. The long-term vision is to design a powerful technique that would allow a software engineer to read a trace just like reading a document where each phase summarizes a given section.

SumTrace is based on a combination of probabilistic [1][14] and Gaussian mixture models [25]. In SumTrace, the occurrence of a function is treated as a random variable, being 1 if it occurs and 0 otherwise. It is clear that after each function any other function may or may not appear. So, we consider the probability of appearance of each function after another one as the basis for our probabilistic model. In this model, if the probability of occurrence of one function after another function is high then the two functions can be considered related. We use an innovative mechanism (as we will show in chapter 3) to group related functions into dense clusters which suggest the presence of execution phases. To automatically determine the phase boundaries (beginning and ending of each phase), we propose to use Gaussian mixture models.

1.2. The Concept of Execution Traces

An execution trace is a sequence of events (e.g., method calls, classes, system calls, etc.), resulting from running a software under particular scenario [20].

A trace event can have different attributes (e.g., nesting level, time stamp, code line number, the thread in which the event occurs, etc.) [20]. In this thesis, we focus on traces of routine calls. By

routine, we mean functions, procedures, and methods as well. An example of a trace of routine calls is given in Figure 1. In this example, the function ‘run’ of the Alma [31] system calls functions ‘step1’ and ‘step2’. Function ‘step1’ calls ‘readInFile’, etc.

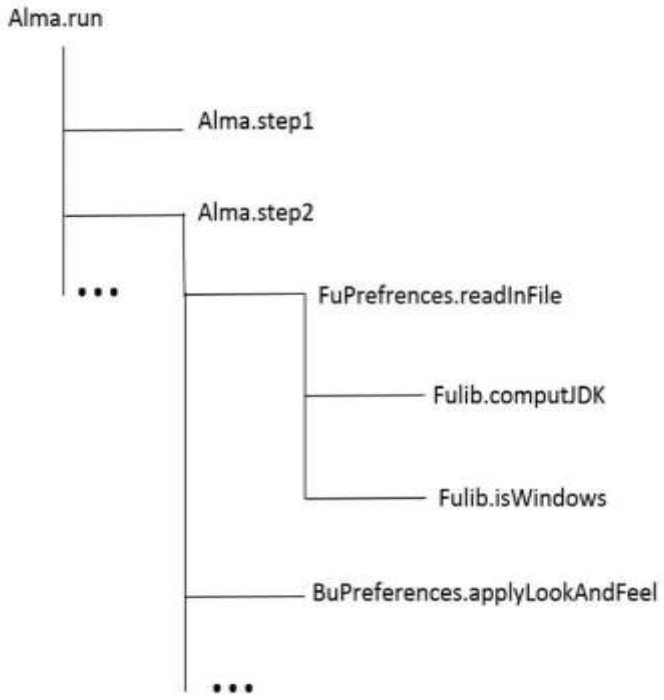

Figure 1. Example of a function call trace

There are different existing methods for generating execution traces. The common approach is injecting a piece of code (called probes) that will be invoked during system execution. A probe is a printout statement that can print information of interest. Instrumentation can be done in different environments. There are three main types of approaches for instrumentation. The first one is to instrument the source code while the other kinds instrument the bytecode (or a compiled version of the code) in the system. The execution environment can be instrumented too. For example probes can be inserted in the point of interest. In particularly, in object oriented systems, probes can be inserted in the body of method. Instrumentation can be done before execution or

during execution (this is known as dynamic probing). In this thesis, we use the Eclipse Test and Performance Tools Platform [28] to instrument the code during the execution of the application.

1.3. Research Contributions

The main research contributions of this thesis are as follows:

- A novel statistical approach based on a probabilistic model, which automatically segments a large trace into meaningful clusters that represent the execution phases of the traced scenario. The method also summarizes the content of each phase.
- The phase detection algorithm based on Gaussian mixture models which minimize the human interventions in comparison with previous methods.
- A complete validation of the approach on large traces generated from two object-oriented systems.

1.4. Thesis Outline

The rest of the thesis is structured as follows:

Chapter 2 - Background

This chapter begins by identifying the needed terminology to understand the concepts presented in this thesis. The chapter continues with a detailed literature review, followed by a general discussion.

Chapter 3 - Approach

This chapter discusses the SumTrace approach. The chapter starts with the definition of the execution phases and then continues with presenting the trace summarization process which is based on probabilistic and Gaussian mixture models. We present a sample example to show the steps of the algorithm. The chapter concludes with a discussion.

Chapter 4 – Evaluation

We show the effectiveness of our approach on two different software applications. The chapter discusses the results and threats to the validation of the method.

Chapter 5 - Conclusion and Future Work

In the beginning of this chapter, we revisit the main contributions of this thesis to conclude the thesis. The chapter continues by presenting some opportunities for future research**.**

Chapter 2 – Background and Related Work

In this chapter, we present the background of this thesis by introducing the necessary concepts needed to understand the content of this thesis, followed by related work.

2.1. Software Maintenance and Program Comprehension

Software maintenance can be defined as the process for changing a system after it is released. Changes may be due to adding new features, fixing bugs, or improving the quality of the code. Chapin et al. [2] divide maintenance activities into four categories:

- Adaptive maintenance: This type of maintenance deals with adapting the system to environmental changes such as porting the system to new hardware or OS (operating system) platforms, without affecting their functionalities.
- Corrective maintenance: This type of maintenance deals with fixing bugs and other types of defects.
- Perfective maintenance: It deals with adding new functionality and features to meet new user functional and non-functional requirements
- Preventive maintenance: This type of maintenance consists of improving the quality of the system (through refactoring) to prevent future issues.

During software maintenance and evolution, software engineers spend around 60-90% of their time on understanding the programs [24]. There are different models for comprehending software systems (see [15]). In the first model, the top-down model, a software engineer has some idea about the system through previous experiences. He or she comes up with some

specific hypotheses about what the system does. The hypothesis will be evaluated as he or she explores the code. In the second model, known as the bottom-up model, a software engineer explores the code looking for clues that can be used to build higher level of understanding of the code. The software engineer starts analyzing the code by grouping code statements together into chunks, and looking for relations between different statements. This task is called cross-referencing. This process is repeated several times until the software engineer obtains a high level of understanding the system. The third and most frequent model is a hybrid model where the software engineer uses both top-down and bottom-up strategies for understanding the system.

According to previous studies (see [7]), tracing the control-flow or data-flow during maintenance can help software engineers understand the behavioural aspects of the system. The focus of this thesis is to understand the flow of execution of software by analyzing execution traces which are generated during run-time.

2.4. Dynamic Analysis

Dynamic analysis of software systems consists of analyzing run-time information of the system with the purpose to help software engineers perform maintenance tasks [7]. As discussed in the introductory chapter, the information generated from a system's execution takes usually the form of execution traces. There are other types of execution information such as system profiling (e.g., CPU and memory usage, number of executed statements, etc.), which tend to be more useful in performance analysis than in maintenance.

Traces contain the list of events which occur during program executions [7]. Execution traces can be generated in various ways. The most common approach is source code instrumentation, which requires modification of target software. Instrumentation is done automatically and consist

of adding probes in places of interests (e.g., beginning and ending of each function). In the absence of the source code, one can also instrument the execution environment. In this way, there is no need to modify the source code. Figure 5 shows a typical way of generating a trace from a software application. First, the maintainer considers a particular execution scenario. Then, the software is instrumented by inserting probes in places of interest. The system is recompiled with the new probes in it. The trace is generated as the system runs.

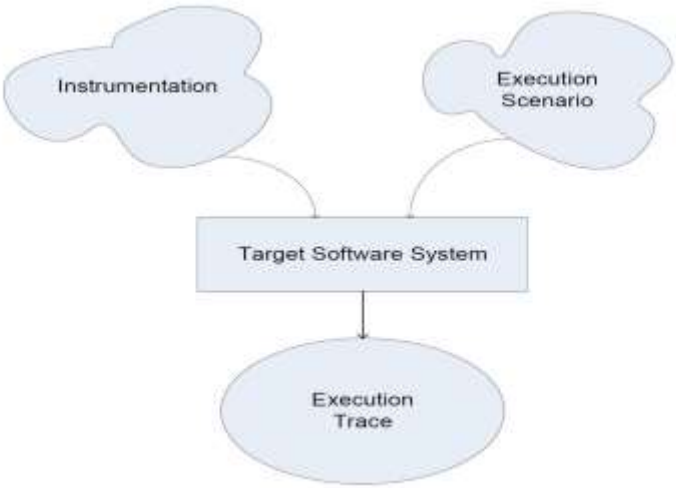

Figure 2. An example of generating an execution trace

2.5. Trace Summarization and Phase Detection Approaches

There exist various studies in the area of analyzing execution traces. In this chapter we group them into two categories: 1) Trace abstraction, or 2) Trace segmentation.

2.5.1. Trace Abstraction

Trace abstraction techniques aim at reducing the size of traces by extracting abstractions from raw events. This is usually done through filtering of trace events by using various criteria. Rountev et al. [23] proposed filtering events related to specific threads using the nesting level of events, and Kuhn et al. [10] used a minimal nesting level threshold to reduce the size of traces. According to the authors, events that appear after a certain nesting level (i.e., depth of the routine call tree) can be considered as utilities. They are not needed for understanding the traced functionality.

Other approaches are based on defining metrics for deciding on what to remove from a trace. For instance, Hamou-Lhadj et al. [5] presented a metric for removing functions that frequently appear in every part of the trace; these are called utilities. Other approaches for summarizing traces are focused on finding patterns in traces. Systa et al. [26] used Boyer-Moore string matching algorithm to find repeated sequence events, that they call them behavioural patterns. Hamou-Lhadj et al. [5][8] proposed an approach to remove repeated instances of events. First, they removed contiguous repetitions then they proposed an algorithm for transforming a rooted call tree to an ordered directed acyclic graph. This way, similar call subtrees were represented only once.

Reiss [22] introduced the concept of visualization of software phases. He developed a tool, called JIVE, for visualizing high level views of what is happening inside the target software. After a certain period of time, JIVE summarizes the information found in the execution traces. This information contains objects which are allocated and destroyed during a system's execution.

Cornelissen et al. [4] developed a technique for visualizing run-time data. The authors proposed a visualization scheme called the circular and massive sequence view. In the circular view, all structural elements are shown in the nesting level by using a circular representation. In another view, which is called the messages sequence view, the entities of software are located in an orderly fashion. The problem of this approach and most visualization approaches is scalability. The challenge starts when the target trace is considerably large; it becomes difficult to visualize in an appropriate scale.

2.5.2. Trace Segmentation

Watanabe et al. [29] proposed a technique for detecting phases in execution traces of large objected oriented codes. The authors used an approach, called the Least Recently Used objects (LRU) for observing objects that appear in the beginning of the program and disappear at the end of it. According to the authors, the sequence of consecutive events which collaborate to build a feature of the system form an execution phase. To visualize the phases, they developed Amida, a tool that detect phases automatically and show them in the form of sequence diagrams. The main challenge of this approach is also scalability.

Kuhn et al. [10] examined the relationship between the analysis of trace information and signals. They proposed a method for segmenting a trace by grouping sequences of events in the trace that exhibit a strong calling relationship. They pruned the trace in multiple places to obtain a reduced trace. Their technique removes a considerable amount of information, which may turn to be important for the users.

Pirzadeh et al. [18] proposed a trace segmentation approach based on Gestalt psychology [11]. They created two measures, similarity and continuation, to bring functions in a trace closer

together to form dense groups, which have later been identified as phases. Their work, however, is limited to repositioning calls to the same function with the hope that calls to different distinct functions end up together. In this thesis, we use a more formal process based on probabilistic model. Besides, our technique leverages trace segmentation to construct trace summaries.

Pirzadeh et al. [20] proposed another phase detection technique, in which the detection process operates on the trace, while it is being generated. This online algorithm keeps track of the methods encountered and raises a flag when a significant number of methods start disappearing and new ones start emerging. This approach requires extensive human intervention for setting thresholds. Our approach on the other hand aims to decrease this kind of interventions.

Medini et al. proposed a concept location technique that relies on trace segments [12] [13]. The trace segmentation approach presented by the authors is based on static analysis of the code. They measured method cohesiveness by comparing the body of methods using the cosine measure. The user needs to define various thresholds to decide on how to measure similarity between functions. Besides, Medini's approach does not summarize the phases as in SumTrace. They applied several algorithms on one trace to detect phases and discover the related phases while in our approach we instantly detect phases and summarize them. The simplicity and speed of our algorithm outperforms their approach.

Chapter 3 – The SumTrace Approach

SumTrace follows four steps as shown in Figure 3. In the first step, we collect a set of traces (that we call a *trace corpus*) from the system. This corpus only needs to be created once. The trace corpus is used in the next step to estimate a probabilistic model of occurrence of each pair of consecutive calls in the system. The intuitive idea is that often function calls in traces exhibit conditional dependencies over a period of time. For example, if function *b* appears most of the times after function *a,* then we can deduce that these two functions are contributing to the implementation of the same execution phase. In the third step, we take a trace that a maintainer wants to analyze and reposition (while summarizing) its events (calls) by bringing closer related functions together using the probabilistic model. What we mean by repositioning trace events is explained in the rest of this chapter. The last step consists of automatically identifying the beginning and the ending of each execution phase. To do so, we use a Gaussian mixture model. The result is a trace summary based on the extracted phases. The steps of our approach are further detailed in the next subsections.

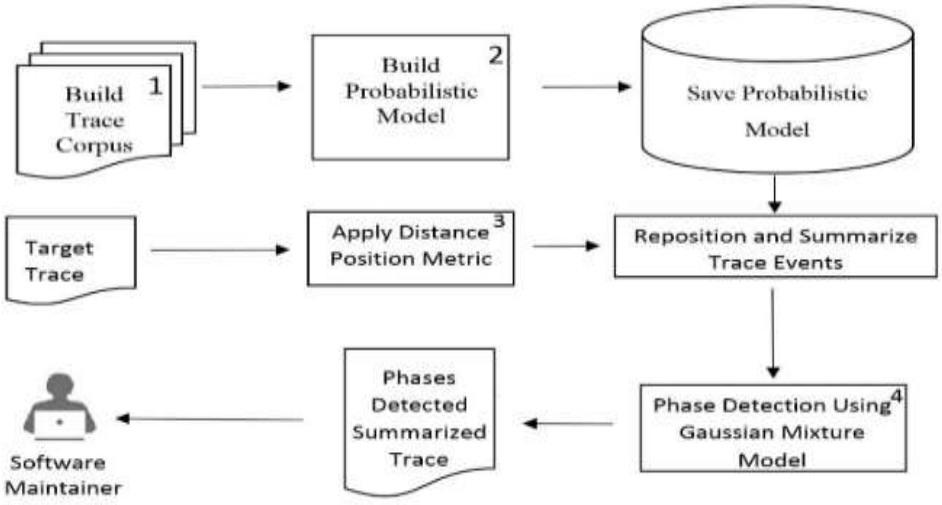

Figure 3. The SumTrace process for extracting execution phases from traces

3.1. Building a trace corpus

To estimate the probability that two or more functions appear frequently together, we need to collect enough data from the system that will be used as a corpus. One possible approach is to use static analysis, more particularly, by building a static call graph. The advantage of this approach is that it provides full coverage of the system. However, it has two main limitations. First, it can only estimate the calling probability, i.e., the probability of a function *a* calling another function *b*. If *a followed by b* appears frequently in a trace without having *a* calling *b,* a static call graph can hardly be used to measure this probability of occurrence. The second limitation is that static call graphs may miss calls due to polymorphism and dynamic bindings.

In this thesis, we propose to rely purely on dynamic analysis. We collect as many traces as possible from the system and use the resulting trace corpus to build the probabilistic model. Function call traces of a software system can be collected using a tracer (e.g., TPTP, The Eclipse Test and Performance Tools Platform [28]). To generate a function call trace, we need to

insert probes at each entry and exit of a function. An example of three traces is presented in Figure 4. We will use these fictive traces as a running example. Start and End events are added to mark the beginning and end of a trace. The interval scale is used in the second step of the approach.

More formally:

Definition 1: A trace *T* of size *S* (i.e., the number function calls invoked in the trace) can be seen as a sequence of events, where each event is a function call, denoted by f_i (*i* represents the invocation order of the function call *f*).

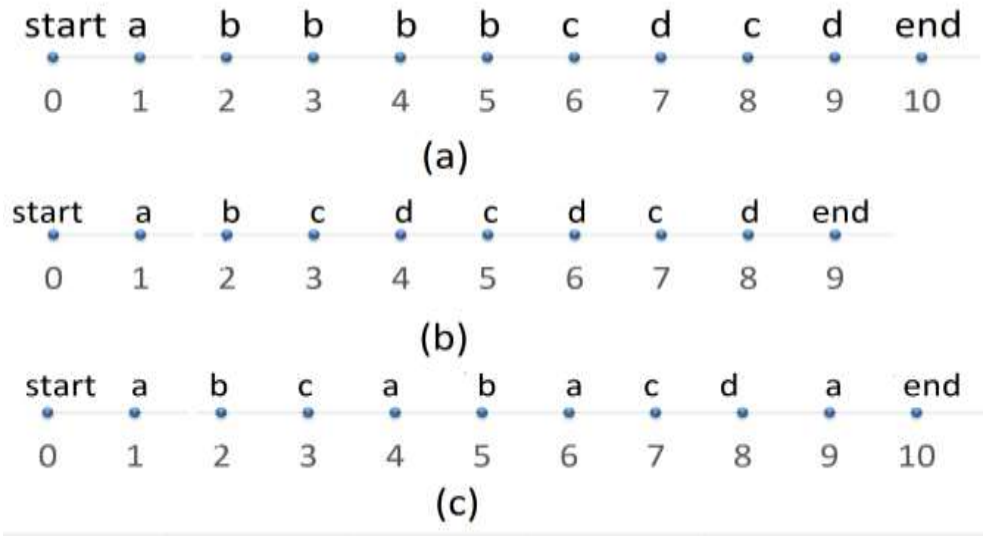

Figure 4. An example of three traces mapped into an interval scale

Unlike static analysis, a pure dynamic analysis approach suffers from the completeness problem—the resulting model may not cover all the paths of the system. Therefore, we need to have a way for determining the number of traces needed to build a representative corpus from which we construct our probabilistic model.

To achieve this, we need to exercise as many different features as possible of the system to obtain adequate coverage of the system. Another alternative solution is to exercise test cases (if available) and

use coverage criteria to have a better set of traces needed to build a representative corpus. It should also be noted that the corpus needs to be updated as the system changes (due to patches and new releases). We anticipate that rebuilding the trace corpus from scratch may not needed, and that incremental updates can be considered. We need to conduct more studies to understand the overhead of maintaining such a corpus on the overall approach.

3.2. Constructing the probabilistic model

After collecting the traces, we measure the conditional (transition) probabilities for any two consecutive functions *i* and *i+1* occurring in the set of traces. The conditional probability is measured using Equation 1. For example, the conditional probability of *"c|a"*, i.e., the function *c* occurs given that the function *a* occurred right before *c* in the three traces of Figure 2 is 1/6 (0.17). This is because *a* occurs 6 times in all three traces and *ac* occurs only once (in the trace of Figure 2c). The conditional probability matrix for the functions in traces of Figure 4 is shown in Table 1.

$$P(f_j|f_i) = \frac{Freq(f_i and f_j)}{Freq(f_i)}$$

Equation 1.Conditional Probability

where $Freq(f_i\ and\ f_j)$ measures the frequency of f_j appearing right after f_i in the trace corpus, and $Freq(f_i)$ measures the number of times f_i appears in all traces.

Using the model, we can determine which functions appear frequently together. For example, we can see that *d* appears in 83% of the cases after *c*, which suggests that these two functions should be part of the same execution phase, because they are contributing to the implementation of the same task.

Table 1 . Probabilistic model table for consecutive functions in traces of Figure 4

f_i	f_j					
	Start	**a**	**B**	**c**	**d**	**End**
Start	0.00	1.00	0.00	0.00	0.00	0.00
a	0.00	0.00	0.67	0.17	0.00	0.17
b	0.00	0.14	0.43	0.43	0.00	0.00
c	0.00	0.17	0.00	0.00	0.83	0.00
d	0.0	0.17	0.00	0.50	0.00	0.33
End	--	--	--	--	--	--

3.3. Applying the probabilistic model for summarizing a trace

Once we construct the probabilistic model, we apply it to the trace that we want to summarize. We call this trace, the target trace. By summarizing, we mean two things: First, we divide the trace into meaningful segments which reflect the execution phases of the traced scenarios. Second, we identify the best phase for each distinct function invoked in the trace. It should be noted that the number of distinct functions in a trace is considerably small (usually in the order of hundreds) as shown by Hamou-Lhadj et al. [6] in their empirical evaluation of the complexity of traces. Therefore, a technique that can place each distinct function in one phase has the apparent advantage of reducing significantly the size of traces. Besides, having functions that

crosscut many phases will make it hard to distinguish among the phases, which may defeat the purpose of the summarization process. We are aware that there exist utility functions that appear almost everywhere in the trace and that it may not make sense to have them assigned to only one phase. We will discuss this issue in the next chapter, when we present the case studies.

To facilitate the understanding of the rest of this sub-section, we introduce the following definitions:

Definition 2: We define an initial mapping from the invocation order of the trace events into an interval scale in such a way that the distance between two consecutive calls is 1. For example, the result of mapping the trace *abbbbcdcd* to an interval scale is shown in Figure 4a. The interval unit is not important as long as the distance between the events is consistent.

Definition 3: We define $Pos(f_i)$ to determine the position of the function call f_i, using the mapping of the trace into the interval scale as per Definition 2. Right after the generation of the trace, the position of any function call of the trace equals its order of invocation (i.e., i). We will see that after repositioning the trace events that this position will change.

Definition 4: We introduce the function $DistinctPos(f_i)$ to return the order of invocation of the function f_i in a given trace by taking into account only the occurrence of distinct functions. To make this clear, take for example the trace in Figure 2a. The position of the function *d*, $Pos(d_7) = 7$, whereas its distinct position is $DistinctPos(d_7) = 4$, because it appeared after *a*, *b*, and *c* were invoked. The reason behind DistinctPos is to avoid being dependent to target traces. This way the repositioning formula is more based on trace corpus and it provides more general results (see the calculation example.)

The process of summarizing the content of the target trace starts by repositioning the trace events using the interval scale (Definition 2) in such a way that cohesive functions are brought closer together by reducing the distance between two consecutive calls based on their probability of occurrence in the model.

The repositioning of the trace events is performed as follows: For each two consecutive calls f_i and f_j (i.e., f_j appearing right after f_i), the new position of f_j is as follows:

$$Pos(f_j) = \begin{cases} DistinctPos(f_j) + P(f_j|f_i)\dfrac{Pos(f_i) - DistinctPos(f_j)}{2} & if\ C1 \\ Pos(f_j) + P(f_j|f_i)\dfrac{Pos(f_i) - Pos(f_j)}{2} & Otherwise \end{cases}$$

Equation 2.Distance-position metric to rearrange functions

where C1 is a condition that is satisfied if f_j is visited for the first time. Note that if f_j is the first function in the target trace then we consider Pos(f_i) = Pos(start) = 0. It should also be noted that there might be situations for which f_j in the target trace does not appear in the probabilistic model, i.e., it was not invoked when building the trace corpus. In this case, we simply consider $P(f_j|f_i)$ to be zero. Future work should focus on ways to improve the probabilistic model when new functions are discovered in the target traces or when the system changes due to patches, etc.

The idea behind Equation 2 is to reduce the distance between $f_j$and f_i based on the probabilistic model constructed in the previous step. If the probability of f_j appearing after f_i converges to 1, then the distance between f_j and $f_i$is reduced to half. A probability closer to 0 would mean that

the position of f_j remains almost as the previous one. Note that the distance could have been reduced by more than half. The focus here is on the fact that the same functions are placed close enough to each other to form a dense group. We do not think that the amount by which we reduce the gap between cohesive functions matters much as long as it is used consistently.

Also, recall from Definition 3 that the initial positions (i.e., right after the trace is generated) of all function calls invoked in the trace equals their order of invocation. In addition, we choose to use the distinct position when the function is processed for the first time to have a distance measure that is less sensitive to repetitions and other variations in the trace. Consider, for example, the case of the trace in Figure 2a. The first call to *d* appears at position 7, despite the fact that it appears only after 3 distinct functions (*a*, *b*, and *c*) were called. The repetitive calls to *b* created what we consider to be bias in the data. Removing contiguous repetitions from the original trace is not an option because there might be situations where *d* appears after multiple calls to the same functions, but in no particular order. For example, in the trace of Figure 2c, *d* appears after many calls to *a*, *b*, and *c*. The distinct position is only needed the first time we visit a new function. The position of the subsequent calls to this function are updated using their position, measured with Pos( ).

To illustrate the way the repositioning mechanism works, consider, for example, the trace of Figure 2c, as the target trace. The new position of each function is calculated as shown below.

1. Pos(a)= DistinctPos(a) + P(a|start) * ((Pos(start)-DistinctPos(a)) / 2=1+1*(0-1)/2=0.5
2. Pos(b) = 2+0.67*(0.5-2)/2=1.49
3. Pos(c)=3+0.43*(1.49-3)/2=2.67
4. Pos(a)= 0.5+0.14*(2.67-0.5)/2=0.65

5. Pos(b)= 1.49+0.14*(0.65-1.49)/2=1.43

6. Pos(a)=0.65+0.67*(1.43-.0.65)/2=0.91

7. Pos(c)=2.67+0.17*(0.91-2.67)/2=2.52

8. Pos(d)=4+0.83*(2.52-4)/2=3.388

9. Pos(a)=0.91+0.17*(3.38-0.91)/2=1.11

Note that the new position of a given function supersedes the previous one. The resulting summary consists of the trace distinct functions mapped into an interval scale that varies from 0 to the number of distinct functions of the target trace, where each distinct function is best placed based on the probabilistic model.

The summary resulting from processing the trace in Figure 4c is shown in Figure 5. From Figure 4, we can infer that *a* and *b* form a group that may suggest the presence of an execution phase. Functions *c* and *d* form another phase. In practice, this clear demarcation may be hard to obtain, especially for large traces. Therefore, we should find a way to automatically distinguish between the formed groups. This is the subject of the next subsection.

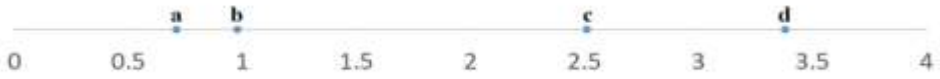

Figure 5. A summarized trace extracted from the trace of Figure 4c

3.4. Detection of phase boundaries

To decide on the phase boundaries, we use a probabilistic approach based on Gaussian mixture models[25]. These models are often used as model-based techniques for clustering problems. Here, the phase boundary identification is treated as a clustering problem where each phase can

be considered as a cluster. Unlike other clustering techniques, the Gaussian mixture model assigns probability to each data point based on estimated parameters (variance and mean) to determine the best partitioning of the data. This criterion makes our algorithm less sensitive to the number of clusters, which is a challenging task in other clustering algorithms such as k-means [18]. This said, the Gaussian mixture model requires less human intervention for deciding on the number of clusters. Suppose a summarized target trace, obtained in the previous step, contains N distinct functions. Let $d_1, d_2,...,d_{N-1}$ be the pairwise distances between the positions of two consecutive functions in the summarized trace. In this approach d_i is considered as a random sample of observations between the positions of any two randomly selected consecutive functions. We assume that a known transformation of d, say, $T(d) = d^*$, follows a Gaussian mixture model

$$T(d) = d^* \sim \sum_{k=1}^{K} \pi_k * N(d^*; \mu_k, \sigma_k^2),$$

Equation 3.Guassian mixture model for a transformation of distances

where π_k is interpreted as the proportion of each cluster out of all clusters, K represents the number of clusters, and $d^* = \log(d)$ is the log transformation of our distances obtained by using the distance position metric (which will be termed as log-distance in the remaining text). Also, $0 \leq \pi_k < 1$, $\sum_{k=1}^{K} \pi_k = 1$, and $N(d^*;\mu_k,\sigma_k^2)$ is the probability density function of a Gaussian distribution with mean μ_k and variance σ_k^2, for each $k(cluster) = 1,2,...,K$.

Gaussian mixture models are popular model-based techniques for clustering problems in statistics and machine learning. These models are computationally easy to fit, and in practice they often provide a very good approximation to the true probability distribution of real data. In

our case studies, the data is d* which is treated as a continuous random variable and its distribution is approximated with a Gaussian mixture reasonably well [25].

To put it simply, consider the trace in Figure 5 that contains four distinct functions. First, we calculate the distances between two consecutive functions in the repositioned trace and then we transform distances to log-distances which are shown in Table 2. Afterwards, we create clusters of the log-distances using the Gaussian mixture models. Note that the example we have used is only for illustration purpose, i.e., to show the calculations. However, in the real world, we have millions of function calls in actual traces and hundreds of distinct functions. Gaussian mixture models actually determine multiple normal distributions in data and form their clusters. Multiple normal distributions are found in a large number of data points (functions) but not in few functions as shown so far with the example of four functions. Therefore, in this section, we modify our example to assume that the number of functions in the summarized trace is approximately 180 (i.e., the trace contains 180 distinct functions; this example is inspired from one of the case studies presented in the next section). We assume that the summarized target trace containing 180 functions is the one shown in Figure 5.

Table 2. Distances and transformation to log-distances

position	d=distance	Function	d^*=log-distance
0.977	0.233	A	-1.4567168
1.21	1.3	B	0.2623643
2.51	0.87	C	-0.1392621
3.38	NA	D	NA

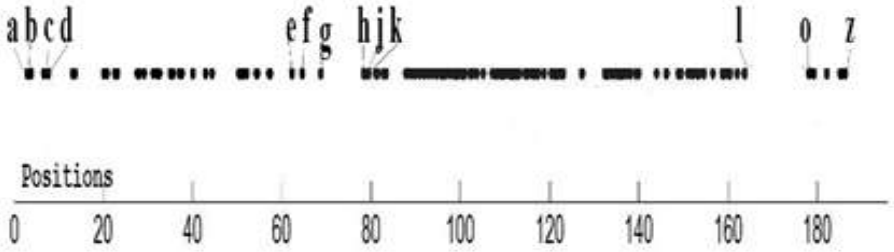

Figure 6. The summarized trace containing 180 functions

Nonetheless, the number of clusters is still unknown after transforming distances to log-distances. Unknown parameters of the model are $K, \pi_k, \mu_k \sigma_k^2$, which are estimated as follows: For each value of $K = 1,2,...,K^*$, and some pre-specified upper bound K^*, the parameter estimates $\{(\hat{\pi}_k, \hat{\mu}_k, \hat{\sigma}_k^2) : k = 1,2,...,K\}$ are obtained using the data $d_1, d_2, ..., d_{N-1}$ and the well-known expectation maximization (EM) algorithm[16]. The best model is then selected using the Bayesian information criterion (BIC)[25] of the final selected model is given by:

$$f(d^*) = \sum_{k=1}^{\hat{K}} \hat{\pi}_k * N(d^*; \hat{\mu}_k, \hat{\sigma}_k^2)$$

Equation 4. Best fitted Gaussian Mixture Models

For each distance d_i^* between two functions the probability of belonging to cluster k is given by:

$$P_{ik} = P(d_i^* \in \text{cluster } k | d^*) = \frac{\hat{\pi}_k N(d_i^*; \hat{\mu}_k, \hat{\sigma}_k^2)}{f(d_i^*)},$$

for all $i = 1,2,...,N-1$ and $k = 1,2,...,\hat{K}$.

Equation 5. Probability of belonging to a cluster

We ran EM algorithm on our data several times to obtain the best maximum likelihood. Once we obtained optimum likelihood we stop the algorithm and collect the results. Note that we used

BIC to estimate the initial number of clusters, while, as we mentioned before, for Gaussian Mixture Models, increasing the number of clusters keeps a stable partitioning, which is not the case for K-means [18]. In K-means, adding a new cluster may result in a completely different partitioning. In our analysis, the main cluster is the one with the largest mean value. We use this cluster to determine the phase boundary. In Gaussian mixture models, each cluster represents a different normal distribution. Therefore, the phase boundary can be determined by finding out the outlier of the normal distribution of the cluster with largest mean. Thus, we determine the phase boundary by using Equation 6

$$di^* \geq \hat{\mu}_k + 2 * \hat{\sigma}_k$$

Equation 6. Determining the phase boundaries

In the case of Figure 6, after using Equation 6, the functions have been segmented into three phases as shown in Figure 5. This actually means that the log-distance between functions "g and h" and "l and o", in Figure 7, is higher than the phase-boundary value obtained from Equation 6. This allows us to automatically find out the phases for our repositioned trace.

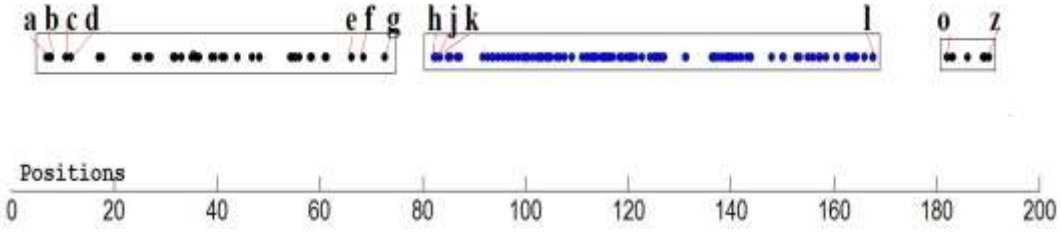

Figure 7. The summarized target trace with phases

We implemented our technique in C#. The complexity of the repositioning technique is linear, based on the number of calls in the trace. We used R [21] and a library called "mix tools" that implements the Gaussian models.

Chapter 4 - Evaluation

We evaluated the effectiveness of SumTrace through a number of intrinsic case studies. We based our evaluation on the documentation provided by the original developers and maintainers of the selected subject systems. The choice of intrinsic studies constrained us to select subject systems that satisfy two conditions: 1) the systems have to be publicly available to allow the replication of this study, and 2) the systems need to be well-documented to allow us to verify the results. These conditions led us to choose well-known open source systems: JHotDraw [9] and Weka [30].

4.1. JHotDraw

We performed the first case study on JHotDraw (version 5.2), which is a framework implemented in Java for technical and structured graphics [9]. It consists of 11 packages, 171 classes, 1414 methods and 9419 lines of codes.

We imported JHotDraw's Java source code into Eclipse, used TPTP to instrument the source code, and collected traces [28]. We ran JHotDraw several times covering a variety of scenarios (functionalities) such as drawing different shapes, changing colours, and changing fonts, etc. We collected 34 traces to build the corpus by exercising various features of JHotDraw.

We collected a target trace by executing the following scenario in JHotDraw: Create a new view, draw a rectangle, line, and circle, run animation, stop the animation, and close the application. The target trace has around 233,000 function calls, and after applying our approach, the resulting

trace contains 189 function calls (by keeping only the distinct functions and repositioning them as discussed earlier).

The next step is to find phase boundaries (distance threshold). To determine phase boundaries, we first determine the number of clusters in the summarized target trace using the BIC score. The BIC score turned out to be two for JHotDraw (i.e., two clusters). We created the two clusters of log-distances of functions by applying the EM clustering algorithm (other algorithms can also be used). We then used the cluster with the largest log-distances to determine the phase boundary using Equation 6. The phase boundary for JHotDraw turned out to be a log-distance of 1.9. Therefore, we created phases in the summarized target trace whenever the log-distance between two functions increased beyond 1.9. We found three phases as shown in Figure 8.

To validate the phases, we used JHotDraw documentation. We found that the phases correspond respectively to initialization, computation, and finalization of the system. Table 3 shows the details of the three phases including the number of functions, and a selected set of functions for each phase. The full results are presented in Appendix A. After checking manually the functions in each phase against both source code comments and JHotDraw documentation, we found that the first phase contains functions which initialize JHotDraw. Examples of these functions include *createDrawing, newWindow, createDrawingView*. The second phase contains the core computation of the traced scenario which consists of drawing shapes. The functions in this phase include *drawLine, draw, color*, etc. The last phase contains functions that terminate the application such as *exit* and *destroy*.

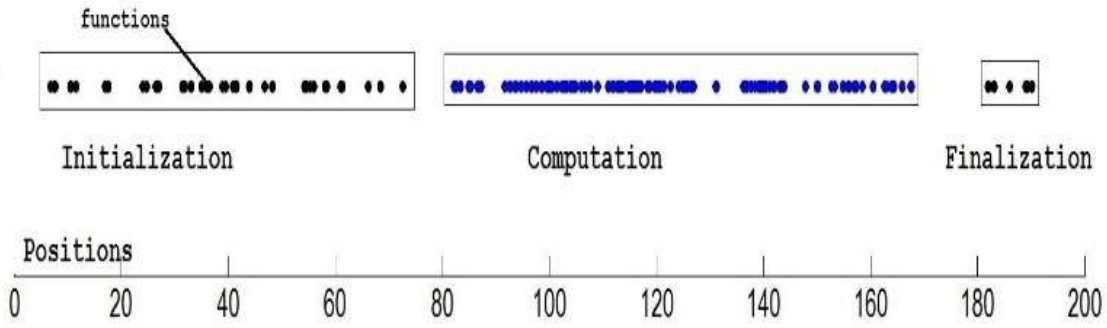

Figure 8. Main phases of the target trace in JHotDraw

Table 3. Sample functions in each phase

Phase	**Number of functions**	**List of sample functions**
Initialization	49	CommandMenu.actionPerformed javadraw/JavaDrawApp.createDrawing MDI_DrawApplication.newWindow DrawApplication.createDrawingView
Computation	133	StandardDrawingView.repairDamage RedoCommand.isExecutableWithView DecoratorFigure.containsPoint PolyLineFigure.drawLine PolyLineFigure.draw util/ColorMap.color DecoratorFigure.draw DrawApplication.view

		JavaDrawApp.startAnimation Animator.start
Finalization	7	JavaDrawApp.endAnimation Animator.end Application.exit JavaDrawApp.destroy DrawApplication.destroy

In Table 4, we provided a description of each phase based on our examination of the code and documentation. Automatic labeling of phases, though it is outside the scope of the thesis, is also possible. We can, for example, use information retrieval (IR) to extract keywords from function names, source code comments, and other artifacts to construct labels. IR-based techniques such as the ones used in feature location research (see **Error! Reference source not found.**) can also be adapted.

With our approach, a maintainer can further zoom into a phase to identify its sub-phases, especially if the number of functions in a phase is large. For example, Phase 2 has 133 functions. We can divide it into sub-phases by reapplying the clustering step to only this fragment of the summarized trace. According to Equation 6, the phase boundary is 1.4 log-distance. This resulted into five sub-phases for Phase 2. Figure 9 shows the repositioned functions in sub-phases of Phase 2 and Table 5 shows the functions that belong to each sub-phase.

Table 4. The three phases in the target trace of JHotDraw

Phase	**Description**
Initialization	Make a new view, maximize view, and unselect the pointer button and so on.
Computation	Draw the rectangle, fill color, unselect the rectangle, and draw a line, run animation, and so on.
Finalization	Ending the animation, deselect the view, destroy the view and close the application

Again, we turned to JHotDraw documentation and source code comments to manually label the phases based on the functions they contain. The first sub-phase contains functions that prepare the view for the drawing (example of functions includes *select*, *activate, view*, etc.). The second one contains functions for drawing rectangle, circles, and adding figures to the viewing area as suggested by the name of the functions belonging to this phase. The third one contains functions for modifying the shapes. The fourth sub-phase contains functions for drawing a line and changing both its color and size. Finally, the fifth sub-phase contains functions for running the animation which contains moving the figures in the view.

Table 5. Sample functions in sub-phases of phase 2

Phase	List of sample functions
Preparation	CreationTool.activate AbstractCommand.isExecutable AbstractCommand.view StandardDrawingView.fireSelectionChanged
Figure Drawing	FigureChangeEventMulticaster.add AbstractFigure.addToContainer DecoratorFigure.displayBox RectangleFigure.basicDisplayBox CompositeFigure.add
Figure Rendering	RectangleFigure.displayBox RectangleFigure.drawBackground RectangleFigure.drawFrame ColorMap.isTransparent ColorMap.color
Draw Lines	PolyLineFigure.drawLine PolyLineFigure.draw
Animation	JavaDrawApp.startAnimation Animator.start StandardDrawingView.selectionHandles

The total time of execution of our approach on a computer system containing Intel core i5 3.10 GHz CPU and 12 GB of RAM was less than 2 minutes (this did not include the collection of traces used to build the probabilistic model). The time to collect traces depends on the scenarios that are exercised and the context in which the system is used. In our case, it took approximately 15 minutes to collect 38 traces.

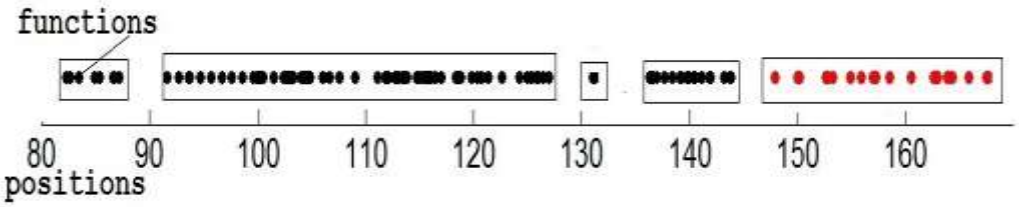

Figure 9. Five sub-phases of Phase 2

4.2. Weka

We performed a second case study on Weka (ver. 3.7.11) [30]. Weka is a software application that contains a collection of machine learning algorithms. The algorithms can either be applied directly to a dataset or called from your own Java code. Weka contains algorithms for data pre-processing, classification, regression, clustering, association rules, and visualization. It is also well-suited for developing new machine learning algorithms [30].

In order to create probabilistic model, we collected 68 traces by executing various scenarios covering the different classification algorithms in Weka to collect 68 traces. This includes changing different parameters of the classification algorithms, setting different datasets for training and testing, and evaluating various output settings for each algorithm including the plots generated by Weka. We built the probabilistic model from these 68 traces.

We generated the target trace by importing a sample dataset which comes with Weka, applying the decision stump classification algorithm on the dataset, using 10-fold cross validation, and closing Weka. During this process, Weka also generated plots of different attributes in the data to facilitate the visualization of relationships among the attributes in the dataset. Weka also performed computations to plot the results in the form of ROC (Receiver Operating Characteristic) curves. Since Weka is multi-threaded, we created separate traces for each thread and focused on the analysis of the core thread (the one that focuses on performing the classification, evaluation, and plotting of the results) as the target trace.

The size of the target trace was around 123,000 function calls and after the execution of our approach, the target trace was reduced to 179 function calls. For finding the phase boundary, we calculated the BIC score for the log-distances between two consecutive functions in the summarized target trace. In this case, the best BIC score was 3 (i.e., 3 clusters). We applied the EM algorithm on the log-distances to determine the clusters and selected the cluster with the large distances to determine the phase boundary by using Equation 6. The phase boundary turned out to be the log-distance of 2.3. We created phases in the summarized target trace whenever the log-distance between two consecutive functions increased beyond 2.3. This resulted into three phases as shown in Table 6 and Figure 10.

By reviewing Weka`s documentation and the code, we found that the first phase is dominated by functions that are used to prepare the classifier such as starting the task (example of a function is *taskStarted* in Table 6), checking attribute types, initializing logging facility (*log*), etc. The second phase is concerned with executing the classifier. The functions in the second phase include splitting and sorting the instances (*findSplitNumeric, sort*), determining entropy (*ContingencyTable.entropy*), building a classifier (*buildClassfier*), and evaluating the classifier (*meanAbsoluteError*). The last phase contains functions that output the results of the classification including determining the recall, precision and other measures (*numFalseNegatives, falsePositiveRate, recall,* etc.), displaying the results to the GUI (*addPlot*), plotting the ROC curve (*areaUnderROC*), and finishing the task (*taskFinished*). We summarized the description of each phase in Table 7. Again, this is done manually by examining the functions in each phase and referring to Weka documentation and source code.

Table 6. Selected functions of the Weka phases

Phase	Number of functions	Sample Functions
Classifier Preparation	38	LogPanel.taskStarted FileLogger.append Logger.log Capabilities.enable Capabilities.enableAllAttributeDependencies
Classifier Processing	74	DecisionStump.findSplitNumeric Instances.sort Instances.deleteWithMissingClass Attribute.copy Instances.relationName ContingencyTable.entropy DecisionStump.buildClassifier DenseInstance.toDoubleArray
Classifier Results	67	Evaluation.toSummaryString Evaluation.meanAbsoluteError Evaluation.numFalseNegatives Evaluation.falsePositiveRate Evaluation.recall

		Evaluation.areaUnderROC VisualizePanel.addPlot ClassPanel.addRepaintNotify ClassifierErrorsPlotInstances.createPlotData TaskMonitor.taskFinished LogPanel.taskFinished

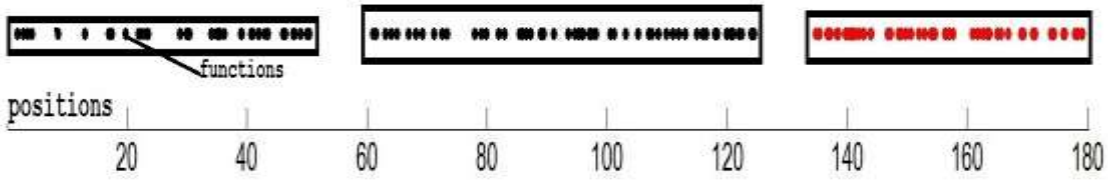

Figure 10. Three phases in the target trace of Weka

Table 7. Description of the Weka phases

Phase	Description
Classifier Preparation	Checking attribute types, parsing classifier options (parameters) from the user, initializing logging facility, and enabling the classifier capabilities against the dataset
Classifier Processing	Splitting the instances, sorting the instances, determine entropy, building the classifier, , and measuring accuracy for instances
Classifier Results	Evaluating the instances ,determining the recall, precision and other measures per attribute of a label, plotting ROC curves and other curves in GUI, Finishing task

Since the last two phases have the largest number of functions, we decided to further divide them into sub-phases. For saving space, we shall only discuss Phase 3 that we refer to as 'classifier results'. Table 8 shows the sub-phases of the third phase of Weka and Figure 11 shows the repositioned functions of these sub-phases. These sub-phases were obtained by repeating the clustering step of our approach on the functions of Phase 3. The value of phase boundary is 0.7 (log-distance) according to Equation 6. The names of the sub-phases and the functions in them are described in Table 8. It can be seen that the sub-phases clearly separate the functionalities of Weka. For example, the sub-phase, called attribute evaluation, contains functions that compute different measures on different attributes of the label (i.e., class values). Similarly, the ROC

evaluation sub-phase contains functions related to the ROC curve, the visualization sub-phase contains functions about the plotting of charts, and the task finalization sub-phase contains functions about the finalization of processing of the classifier.

Table 8. Selected functions of sub-phases of the third phase of Weka

Sub-phases	List of functions
Attribute Evaluation	Attribute.name Evaluation. Recall Evaluation. Precision
ROC Evaluation	Evaluation.areaUnderPRC Evaluation.weightedAreaUnderROC
Visualization	AbstractPlotInstances.canPlot VisualizeUtils.processColour ClassPanel.addRepaintNotify VisualizePanel.addPlot
Task Finalization	LogPanel.taskFinished TaskMonitor.taskFinished

The time to execute our approach took less than 2 minutes, after the collection of traces. The time to collect traces was approximately 65 minutes, because of the various settings required to execute classification of algorithms.

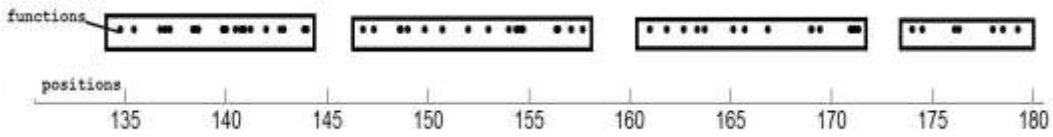

Figure 11. Sub-phases in the third phase of Weka's target trace

4.3. Discussion and Limitations

The results of applying SumTrace to traces of two software systems show that the approach is promising in segmenting and summarizing the traces into distinct execution phases. We believe that the key success of SumTrace is attributed to the use of a formal process for measuring the cohesion among functions, which is based on a probabilistic model. Building a probabilistic model, however, requires a data corpus. In our case, we used a collection of traces. We argued that these traces should cover various features of the system to provide good coverage. Determining the exact number of traces needed to build a representative depends on many factors including the complexity of the system.

In addition, the trace corpus needs to be updated whenever the system changes (new patches, etc.), which might be time consuming. We need to investigate ways to increment the corpus as parts of the system change.

Another important aspect of SumTrace is that it assigns each distinct function of the trace to a specific phase. As a result, we obtain a summary that is as large as the number of distinct functions in the target trace. At first sight, this may appear a little odd, because some functions (such as utilities) may be shared among phases. In fact, it all depends on the objective of the trace segmentation process. If the objective is to identify the detailed implementation of each phase by providing the list of its functions, then we need to allow the same function to appear in

multiple places. This can be achieved by modifying SumTrace to keep the new position of every single occurrence of a function when repositioning the trace events. Currently, when a new call to the same function occurs, the new position calculated with Equation 2 supersedes the previous one. If, on the other hand, the objective is to summarize the trace, which is the case in this study, the focus should be on placing the functions that are most relevant to the implementation of a phase in this phase and this phase only. If the phase has extra (and perhaps less relevant) functions, this should not impact the overall understanding of the phase content, especially because the size of phases is relatively small (again this is because we only keep distinct functions). We can also examine the automatic removal of utilities before applying SumTrace such as the ones proposed by Hamou-Lhadj et al. in [6][8].

Finally, in a normal run of a system, the same execution phase may appear multiple times in the trace. For example, drawing a rectangle could be performed multiple times at different points of the traced scenario. So how does SumTrace handle multiple instances of the same phase? This is easily achievable by taking each phase (result of SumTrace) and search in the original trace for segments that have similar functions. This leads to an interesting future study which relates to phase search and localization.

Chapter 5 - Conclusion

In this chapter we conclude our thesis by summarizing our research contributions in Section 5.1, which also includes a discussion about the results achieved by our approach and its effectiveness. In Section 5.2, we elaborate on opportunities for future research to further improve the effectiveness and accuracy of the present approach. Finally in section 5.3 we provide our closing remarks for this thesis.

5.1. Research Contributions

In this dissertation, we proposed a new statistical approach for summarizing function call traces into distinct execution traces. We have proposed a trace summarization approach, called SumTrace, which leverages the concept of trace segmentation. We also used probabilistic and Gaussian mixture models to generate summarized execution phases from large traces with minimum human intervention. The output of this approach provides maintainers a way to grasp the content of large traces by segmenting their trace content. It helps the maintainer to look at each phases and recognize the distinct functions of each phase rapidly.

We experimented with SumTrace on traces of two large systems and show that it holds real promise in segmenting effectively and efficiently the content of large traces.

5.2. Opportunities for Further Research

The immediate future work consists of conducting further experimentation on other feature traces. In particular, we intend to target larger systems.

Another future work is to investigate how we can build representative corpuses that can be used to guide the construction of the probabilistic model. One alternative is to use test cases and coverage criteria to decide on the number of traces that would form the corpus. The problem with this is that execution test cases may be an expensive task. Besides, not all systems have a full set of test cases.

Another limitation of our approach is that it does not account for changes in the system such as new patches, etc. We will need to update the corpus whenever the system changes. We believe that a complete reconstruction may be avoided if one can detect only the elements of the system that have been modified. Future work should address this question while having in mind the trade-off between accuracy and completeness.

In addition, we need to investigate the impact of utility functions on the whole process. Utility functions are the ones that appear in multiple places (called by many components). They can be seen as noise in the data. In the current version of SumTrace, we treat utility functions just like any other function. We may consider removing them and assess the impact of the accuracy of SumTrace to build representative phases.

Finally, a trace analysis approach such as SumTrace is only adopted if it is well embedded in a trace analysis tool suite. Future work should focus on providing adequate tool support to SumTrace. The tool can then be used by software engineers solving maintenance tasks. This will allow us to conduct user studies and assess the effectiveness of SumTrace in practice.

5.3. Closing Remarks

The automatic segmentation of large execution traces can simplify the analysis of dynamic information of a software system, which in turn can help in software comprehension tasks. SumTrace aims to provide such a trace segmentation process. SumTrace is simple and efficient. It only requires one pass through the trace to extract meaningful segments. We believe that, if supported by adequate tools, SumTrace can be effectively used by software engineers working on understanding the behavioural aspects of a software system. As such, we believe that SumTrace greatly contributes to the state of the art in trace analysis research.

Bibliography:

[1] E. Cinlar. *Introduction to Stochastic Processes*. Dover publications, Mineola, NY, USA, 1975.

[2] N. Chapin, E. Hale, K. Kham, F. Ramil, W. Tan, "Types of software evolution and software maintenance," *Wiley Journal of Software Maintenance and Evolution, 13(1)*, 2001, pp. 3–30.

[3] K. Chen and V. Rajlich "Case Study of Feature Location Using Dependence Graph", In *Proceeding of the 8th International Workshop on Program Comprehension*, 2000, pp. 241-249.

[4] B. Cornelissen, A. Zaidman, A .V. Deursen, L. Moonen and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Engineering (TSE)*, 35(5), 2009, pp. 684-702.

[5] A. Hamou-Lhadj A. and T. C. Lethbridge, "Techniques for Reducing the Complexity of Object-Oriented Execution Traces," In *Proceedings of the 1th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2003, pp. 35-40.

[5] A. Hamou-Lhadj, and T. C. Lethbridge, "A Survey of Trace Exploration Tools and Techniques," In *Proceedings of the Centre for Advanced Studies on Collaborative research Conference,* 2004, pp. 42-54.

[6] A. Hamou-Lhadj and Timothy C. Lethbridge, "Understanding the Complexity Embedded in Execution Traces with a Focus on Program Comprehension Tasks," *IET Software Journal, 4(2),* 2010, pp. 161 - 177.

[7] A. Hamou-Lhadj, "The Concept of Trace Summarization," In *Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis*, 2005, pp. 43-47.

[8] A. Hamou-Lhadj, and T. Lethbridge, "Reasoning about the Concept of Utilities," *ECOOP International Workshop on Practical Problems of Programming in the Large,*

Oslo, Norway, Lecture Notes in Computer Science (LNCS), Vol 3344, Springer-Verlag, pp. 10-22, 2004.

[9] JHotDraw, "Opensource GUI framework for technical and structured graphics", Online: www.jhotdraw.org

[10] A. Kuhn and O. Greevy, "Exploiting the analogy between traces and signal processing," In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 320-329.

[11] K. Koffka. *Principles of Gestalt psychology*. Hartcourt, NewYork, 1935.

[12] S. Medini, G. Antoniol, Y. Gueheneuc, M. Di Penta and P. Tonella, "SCAN: An Approach to Label and Relate Execution Trace Segments," In *Proceedings of the 29th Working Conference on Reverse Engineering*, 2012, pp. 135-144.

[13] S. Medini, V. Arnaoudova, M. Di Penta, G. Antoniol, Y. G. Guéhéneuc, P. Tonella, "SCAN: an approach to label and relate execution trace segments," *Journal of Software: Evolution and Process*. 2014.

[14] W. Mendenhall, *Beginning Statistics: A to Z*, Duxbury Press, Pacific Grove, CA, 1993.

[15] M. A. Storey, K. Wong, and H. A. Muller, "How do Program Understanding Tools Affect how Programm ers Understand Programs?" In *Proceedings of 4th Working Conference on Reverse Engineering, IEEE Computer Society,* 1997, pp. 183-207.

[16] G. J. McLachlan and D. Peel, *Finite Mixture Models*. New York, Wiley, 2008

[17] D. Poshyvanyk, M. Gethers and A. Marcus, "Concept Location Using Formal Concept Analysis and Information Retrieval," In *Proceedings of ACM Transactions on Software Engineering and Methodology*, *21(4),* 2013, pp. 1-23.

[18] H. Pirzadeh and A. Hamou-Lhadj, "A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension," In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*, 2011, pp. 221-230.

[19] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, L. Alawneh and A. Sharifee, "Stratified Sampling of Execution Traces: Execution Phases Serving as Strata," *The Elsevier*

Journal on Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Maintenance, *78(8)*, 2013, pp. 1099–1118

[20] H. Pirzadeh, A. Agarwal and A. Hamou-Lhadj, "An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension," In *Proceedings of the 8th International Conference on Software Engineering Research, Management, and Applications*, 2010, pp.207-214.

[21] "R: A Language and Environment for Statistical Computing," R Foundation for Statistical Computing, 2011.

[22] S. P. Reiss, "Dynamic detection and visualization of software phases", In *Proceedings of the 3rd International Workshop on Dynamic Analysis (WODA)*, ACM, 2005, pp. 1-6.

[23] A. Rountev and B. H. Connell, "Object Naming Analysis for Reverse-Engineered Sequence Diagrams." In *Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 254-263.

[24] I. Sommerville. *Software Engineering, A Practitioner's Approach* (9th Edition). Addison-Wesley, 2010.

[25] G. Schwarz, *Estimating the Dimension of a Model*, Annals of Statistics, 1978.

[26] T. Systa, "Understanding the behavior of Java programs," In *Proceedings of Seventh Working Conference on Reverse Engineering*, 2000, pp. 214-223

[27] S. K. Thompson, *Sampling*, 3th edition, John Wiley, New York, NY, USA 1992.

[28] TPTP, "The Eclipse Test and Performance Tools Platform (TPTP) Project," Available online: www.eclipse.org/tptp/

[29] Y. Watanabe. T. Ishio, K. Inoue "Feature-level phase detection for execution trace using object cache" In *Proceedings of the International Workshop on Dynamic Analysis, co-located with the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM Press, New York, New York, USA, 2008, pp. 8–14.:

[30] I. H. Witten, E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques,* Morgan Kaufmann, San Francisco, CA, USA, 2005.

[31] http://www.desnoix.com/guillaume/alma/

Appendix A: Full Results of the Experiments

Table A1. Functions in each phase JhotDraw

Phase	Number of functions	Function names
Initialization	49	MDI_DrawApplication.promptNew MDI_DrawApplication.hasInternalFrames CommandMenu.actionPerformed javadraw/JavaDrawApp.createDrawing QuadTree._makeNorthwest MDI_DrawApplication.newWindow DrawApplication.createDrawingView StandardDrawingView.checkMinimumSize DrawApplication.fireViewCreatedEvent AbstractCommand.viewCreated StandardDrawing.addDrawingChangeListener AbstractTool.viewCreated DragNDropTool.viewCreated StandardDrawingView.clearSelection StandardDrawing.removeDrawingChangeListener MDI_DrawApplication.createContents MDI_DrawApplication.createInternalFrame DrawApplication.createContents StandardDrawingView.selectionZOrdered StandardDrawingView.selectionElements MDI_DrawApplication.internalFrameOpened MDI_DrawApplication.internalFrameActivated MDI_DrawApplication.activateFrame StandardDrawingView.isFocusTraversable StandardDrawingView.unfreezeView StandardDrawing.unlock DrawApplication.fireViewSelectionChangedEvent AbstractTool.checkUsable NullDrawingView.isInteractive UndoableCommand.commandExecutable CommandMenu.commandExecutable AbstractCommand.viewSelectionChanged NullDrawingView.removeFigureSelectionListener StandardDrawingView.addFigureSelectionListener UndoableTool.toolUsable AbstractTool.viewSelectionChanged ToolButton.toolUsable ToolButton.paintSelected PaletteIcon.selected

		DrawApplication.showStatus DrawApplication.toolDone ToolButton.name ToolButton.toolDeactivated StandardDrawingView.selectionCount DrawApplication.paletteUserOver ToolButton.tool AbstractTool.isEnabled AbstractTool.isUsable
Computation	133	DrawApplication.paletteUserSelected AbstractTool.activate PaletteButton.select UndoableTool.activate CreationTool.activate AbstractCommand.isViewRequired AbstractCommand.isExecutable UndoableCommand.isExecutable AbstractCommand.view StandardDrawingView.isInteractive ChangeAttributeCommand.isExecutableWithView DrawApplication.figureSelectionChanged StandardDrawingView.fireSelectionChanged AlignCommand.isExecutableWithView SelectAllCommand.isExecutableWithView CommandMenu.checkEnabled CutCommand.isExecutableWithView CopyCommand.isExecutableWithView PasteCommand.isExecutableWithView DuplicateCommand.isExecutableWithView DeleteCommand.isExecutableWithView GroupCommand.isExecutableWithView UngroupCommand.isExecutableWithView PolyLineFigure.points PolyLineFigure.displayBox StandardDrawingView.checkDamage PolyLineFigure.containsPoint SendToBackCommand.isExecutableWithView BringToFrontCommand.isExecutableWithView UndoCommand.isExecutableWithView StandardDrawing.drawingChangeListeners StandardDrawingView.repairDamage ToolButton.toolActivated AbstractFigure.containsPoint RedoCommand.isExecutableWithView DecoratorFigure.containsPoint ReverseVectorEnumerator.nextElement AbstractCommand.isExecutableWithView

		ReverseFigureEnumerator.hasMoreElements ReverseFigureEnumerator.nextFigure ReverseVectorEnumerator.hasMoreElements QuadTree.add Bounds.asRectangle2D EllipseFigure.basicDisplayBox CompositeFigure.figuresReverse AbstractCommand.figureSelectionChanged CompositeFigure.findFigure UndoableTool.toolActivated RectangleFigure.basicDisplayBox CreationTool.createFigure AbstractFigure.clone LineFigure.basicDisplayBox AttributeFigure.writeObject AbstractTool.isActive DecoratorFigure.basicDisplayBox QuadTree.remove AnimationDecorator.basicDisplayBox CompositeFigure._removeFromQuadTree CompositeFigure.figureChanged StandardDrawingView.drawing StandardDrawingView.add AbstractFigure.willChange AbstractFigure.invalidate AbstractFigure.changed StandardDrawingView.tool StandardDrawingView.isFigureSelected BouncingDrawing.add DecoratorFigure.decorate ToolButton.paint StandardDrawingView.editor DecoratorFigure.figureChanged StandardDrawingView.findHandle StandardDrawingView.drawingInvalidated DrawApplication.tool StandardDrawing.figureInvalidated DecoratorFigure.figureInvalidated AbstractFigure.listener AbstractFigure.displayBox FigureChangeEventMulticaster.addInternal FigureChangeEventMulticaster.add AbstractFigure.addFigureChangeListener AbstractFigure.addToContainer AbstractTool.deactivate AbstractTool.view DecoratorFigure.displayBox

		AnimationDecorator.displayBox CompositeFigure.add AbstractTool.editor CompositeFigure._addToQuadTree RectangleFigure.displayBox RectangleFigure.drawBackground RectangleFigure.drawFrame EllipseFigure.displayBox EllipseFigure.drawFrame EllipseFigure.drawBackground CompositeFigure.figures CompositeFigure.draw StandardDrawingView.drawDrawing StandardDrawingView.drawBackground Geom.range StandardDrawingView.constrainPoint StandardDrawingView.drawAll PolyLineFigure.drawLine PolyLineFigure.draw SimpleUpdateStrategy.draw StandardDrawingView.paintComponent ColorMap.isTransparent ColorMap.color AttributeFigure.draw DecoratorFigure.draw FigureEnumerator.nextFigure DrawApplication.view StandardDrawingView.selectionHandles StandardDrawingView.drawHandles Geom.lineContainsPoint AbstractFigure.isEmpty FigureEnumerator.hasMoreElements CreationTool.createUndoActivity UndoableAdapter.rememberFigures SingleFigureEnumerator.hasMoreElements SingleFigureEnumerator.nextElement UndoableTool.isActive UndoableTool.deactivate UndoableAdapter.isUndoable UndoManager.pushUndo PolyLineFigure.decorate UndoManager.clearRedos UndoManager.clearStack UndoableTool.editor util/UndoableTool.view AbstractFigure.size JavaDrawApp.startAnimation

		Animator.start
Finalization	6	PolyLineFigure.isEmpty JavaDrawApp.endAnimation Animator.end DrawApplication.exit JavaDrawApp.destroy DrawApplication.destroy

Table A2. Sample functions in each sub-phases of Phase 2 of JHotDraw

Phase	List of sample functions
Preparation	DrawApplication.paletteUserSelected AbstractTool.activate PaletteButton.select CreationTool.activate AbstractCommand.isViewRequired AbstractCommand.isExecutable UndoableCommand.isExecutable AbstractCommand.view StandardDrawingView.isInteractive ChangeAttributeCommand.isExecutableWithView DrawApplication.figureSelectionChanged StandardDrawingView.fireSelectionChanged AlignCommand.isExecutableWithView SelectAllCommand.isExecutableWithView CommandMenu.checkEnabled CutCommand.isExecutableWithView CopyCommand.isExecutableWithView PasteCommand.isExecutableWithView DuplicateCommand.isExecutableWithView DeleteCommand.isExecutableWithView
Draw Figures	FigureChangeEventMulticaster.add AbstractFigure.addToContainer DecoratorFigure.displayBox RectangleFigure.basicDisplayBox CompositeFigure.add Bounds.asRectangle2D CreationTool.createFigure RectangleFigure.drawFrame
Modify Figures	ReverseVectorEnumerator.hasMoreElements QuadTree.add CompositeFigure.figuresReverse AbstractCommand.figureSelectionChanged CompositeFigure.findFigure

	AbstractFigure.clone LineFigure.basicDisplayBox AttributeFigure.writeObject AbstractTool.isActive DecoratorFigure.basicDisplayBox QuadTree.remove AnimationDecorator.basicDisplayBox CompositeFigure._removeFromQuadTree CompositeFigure.figureChanged StandardDrawingView.drawing StandardDrawingView.add AbstractFigure.willChange AbstractFigure.invalidate AbstractFigure.changed StandardDrawingView.tool StandardDrawingView.isFigureSelected ToolButton.paint StandardDrawingView.editor DecoratorFigure.figureChanged StandardDrawingView.findHandle StandardDrawingView.drawingInvalidated DrawApplication.tool StandardDrawing.figureInvalidated DecoratorFigure.figureInvalidated AbstractFigure.listener AbstractFigure.displayBox FigureChangeEventMulticaster.addInternal FigureChangeEventMulticaster.add AbstractFigure.addFigureChangeListener AbstractFigure.addToContainer AbstractTool.deactivate AbstractTool.view DecoratorFigure.displayBox AnimationDecorator.displayBox CompositeFigure.add AbstractTool.editor CompositeFigure._addToQuadTree RectangleFigure.displayBox RectangleFigure.drawBackground
Draw Lines	LineFigure.basicDisplayBox PolyLineFigure.drawLine PolyLineFigure.decorate PolyLineFigure.draw PolyLineFigure.decorate
Animation	JavaDrawApp.startAnimation Animator.start AnimationDecorator.basicDisplayBox

	UndoableTool.isActive UndoableTool.deactivate UndoableAdapter.isUndoable UndoManager.pushUndo UndoManager.clearRedos UndoManager.clearStack UndoableTool.editor UndoableTool.view AbstractFigure.size StandardDrawingView.selectionHandles

Table A3.Selected functions of the Weka phases

Phase	Num of functions	Sample Functions
Classifier Preparation	38	addToHistory GenericObjectEditor.makeCopy GenericObjectEditorHistory.add GenericObjectEditorHistory.copy Instances.copyInstances Environment.substitute Instances.checkForStringAttributes Instances.checkForAttributeType AbstractClassifier.makeCopy Utils.splitOptions Utils.forName Utils.checkForRemainingOptions Utils.joinOptions FileLogger.append LogPanel.taskStarted OutputLogger.doLog TaskMonitor.taskStarted ResultHistoryPanel.addResult Logger.log DecisionStump.buildClassifier LogPanel.statusMessage Capabilities.enableAll Capabilities.enableAllAttributes Capabilities.enableAllAttributeDependencies Capabilities.enable Capabilities.enableDependency Capabilities.enableAllClasses Capabilities.enableAllClassDependencies LogPanel.logMessage Capabilities.disableAll Capabilities.disableAllAttributes

		Capabilities.disableAllAttributeDependencies Capabilities.disable Capabilities.handles Capabilities.disableDependency Capabilities.disableAllClasses Capabilities.disableAllClassDependencies Capabilities.testWithFail Capabilities.test
Classifier Processing	74	Instances.deleteWithMissingClass Instances.deleteWithMissing AbstractInstance.classIsMissing DecisionStump.findSplitNumeric DecisionStump.findSplitNumericNominal Instances.sort Utils.sortWithNoMissingValues Instances.numClasses Utils.partition Utils.swap Utils.quickSort Utils.conditionalSwap Utils.sortLeftRightAndCenter Utils.eq ContingencyTables.entropyConditionedOnRows ContingencyTables.lnFunc Utils.normalize DecisionStump.toString DecisionStump.printClass Attribute.value DecisionStump.printDist Utils.log2 ClassifierErrorsPlotInstances.check AbstractPlotInstances.check ClassifierErrorsPlotInstances.determineFormat Attribute.copy ResultHistoryPanel.updateResult ClassifierErrorsPlotInstances.process DenseInstance.toDoubleArray DenseInstance.freshAttributeVector Instances.classIndex AbstractInstance.classIndex DecisionStump.distributionForInstance AbstractInstance.classValue Utils.sum AbstractInstance.attribute AbstractInstance.classAttribute Instances.relationName TaskMonitor.updateMonitor

		Attribute.numValues Utils.missingValue Utils.gr
Classifier Results	67	TaskMonitor.updateMonitor NominalPrediction.updatePredicted Evaluation.toSummaryString Evaluation.toSummaryString Evaluation.correct Evaluation.pctCorrect Evaluation.incorrect Evaluation.pctIncorrect Evaluation.kappa Evaluation.meanAbsoluteError Evaluation.relativeAbsoluteError Evaluation.rootMeanSquaredError Evaluation.meanPriorAbsoluteError Utils.missingValue Evaluation.rootRelativeSquaredError Evaluation.rootMeanPriorSquaredError Evaluation.coverageOfTestCasesByPredictedRegions Evaluation.sizeOfPredictedRegions Evaluation.unclassified Utils.gr Attribute.numValues Evaluation.toClassDetailsString Evaluation.toClassDetailsString Attribute.isNominal Attribute.name Evaluation.numFalseNegatives Evaluation.truePositiveRate Evaluation.numFalsePositives Evaluation.falsePositiveRate Evaluation.recall Evaluation.fMeasure NominalPrediction.actual NominalPrediction.weight Evaluation.numTrueNegatives Evaluation.precision Instances.classAttribute Instances.add DenseInstance.copy Evaluation.numTruePositives AbstractInstance.weight ThresholdCurve.makeHeader

		Evaluation.matthewsCorrelationCoefficient ThresholdCurve.makeInstance Utils.sort Utils.replaceMissingWithMAX_VALUE Instances.attribute Attribute.index Evaluation.areaUnderROC Utils.doubleToString Instances.attributeToDoubleArray Evaluation.weightedFalsePositiveRate Evaluation.weightedPrecision Evaluation.weightedTruePositiveRate NominalPrediction.distribution Evaluation.weightedRecall Evaluation.areaUnderPRC Evaluation.weightedFMeasure Evaluation.weightedMatthewsCorrelation Evaluation.weightedAreaUnderROC Attribute.type Evaluation.weightedAreaUnderPRC AbstractPlotInstances.canPlot Evaluation.toMatrixString Attribute.typeToStringShort Evaluation.toMatrixString Evaluation.num2ShortID ClassifierErrorsPlotInstances.finishUp AbstractPlotInstances.finishUp VisualizeUtils.processColour ClassPanel.addRepaintNotify LegendPanel.addRepaintNotify AttributePanel.addAttributePanelListener ClassifierErrorsPlotInstances.createPlotData PlotData2D.determineBounds VisualizePanel.addPlot Plot2D.addPlot Plot2D.determineBounds Plot2D.fillLookup Instances.numAttributes Plot2D.convertToPanelX Instances.instance DenseInstance.value Utils.isMissingValue AbstractInstance.isMissing Plot2D.convertToPanelY Instances.numInstances ClassifierErrorsPlotInstances.cleanUp AbstractPlotInstances.cleanUp

		Evaluation.predictions Evaluation.predictions ResultHistoryPanel.addObject LogPanel.taskFinished TaskMonitor.taskFinished

Table A4. Selected function of sub-phases of the third phase of Weka

Sub-phases	List of functions
Attribute Evaluation	Evaluation.toSummaryString Evaluation.toSummaryString Evaluation.correct Evaluation.pctCorrect Evaluation.incorrect Evaluation.pctIncorrect Evaluation.kappa Evaluation.meanAbsoluteError Evaluation.relativeAbsoluteError Evaluation.rootMeanSquaredError Evaluation.meanPriorAbsoluteError Evaluation.rootRelativeSquaredError Evaluation.rootMeanPriorSquaredError Evaluation.coverageOfTestCasesByPredictedRegions Evaluation.sizeOfPredictedRegions Evaluation.unclassified Evaluation.numFalseNegatives Evaluation.truePositiveRate Evaluation.numFalsePositives Evaluation.falsePositiveRate Evaluation.recall Evaluation.fMeasure NominalPrediction.actual NominalPrediction.weight Evaluation.numTrueNegatives Evaluation.precision Instances.classAttribute Instances.add DenseInstance.copy Evaluation.numTruePositives AbstractInstance.weight ThresholdCurve.makeHeader Evaluation.matthewsCorrelationCoefficient

	ThresholdCurve.makeInstance Utils.sort Utils.replaceMissingWithMAX_VALUE Instances.attribute Attribute.index Evaluation.areaUnderROC Utils.doubleToString Instances.attributeToDoubleArray
ROC Evaluation	Evaluation.weightedFalsePositiveRate Evaluation.weightedPrecision Evaluation.weightedTruePositiveRate NominalPrediction.distribution Evaluation.weightedRecall Evaluation.areaUnderPRC Evaluation.weightedFMeasure Evaluation.weightedMatthewsCorrelation Evaluation.weightedAreaUnderROC Attribute.type Evaluation.weightedAreaUnderPRC AbstractPlotInstances.canPlot Evaluation.toMatrixString Attribute.typeToStringShort Evaluation.toMatrixString Evaluation.num2ShortID
Visualization	ClassifierErrorsPlotInstances.finishUp AbstractPlotInstances.finishUp VisualizeUtils.processColour ClassPanel.addRepaintNotify LegendPanel.addRepaintNotify AttributePanel.addAttributePanelListener ClassifierErrorsPlotInstances.createPlotData PlotData2D.determineBounds VisualizePanel.addPlot Plot2D.addPlot Plot2D.determineBounds Plot2D.fillLookup Instances.numAttributes Plot2D.convertToPanelX Instances.instance DenseInstance.value Utils.isMissingValue AbstractInstance.isMissing Plot2D.convertToPanelY Instances.numInstances
Task Finalization	ClassifierErrorsPlotInstances.cleanUp AbstractPlotInstances.cleanUp Evaluation.predictions

	Evaluation.predictions ResultHistoryPanel.addObject LogPanel.taskFinished TaskMonitor.taskFinished