Trace Abstraction Based on Automatic Detection of Execution Phases

Akanksha Agarwal

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

December 2010

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Akanksha Agarwal

Entitled: “Trace Abstraction Based on Automatic Detection of Execution Phases”

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

Master of Applied Science

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

Signed by the final examining committee:

______________________________________ Chair
Dr. D. Qiu

______________________________________ Examiner, External To the Program
Dr. A. Youssef (CIISE)

______________________________________ Examiner
Dr. N. Kharma

______________________________________ Supervisor
Dr. A. Hamou-Lhadj

Approved by: ______________________________________
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

____________20_____

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

Abstract

Trace Abstraction Based on Automatic Detection of Execution Phases

Akanksha Agarwal

Understanding the behavioural aspects of a software system is an important activity in many software engineering activities including program comprehension and reverse engineering.

The behaviour of software is typically represented in the form of execution traces. Traces, however, tend to be considerably large which makes analyzing their content a complex task. There is a need for trace simplification techniques that can help software engineers make sense of the content of a trace despite the trace being massive.

In this thesis, we present a novel approach that aims to simplify the analysis of a large trace by detecting the execution phases that compose it. An example of a phase could be an initialization phase, a specific computation, etc. Our algorithm processes a trace generated from running the program under study and divides it into phases that can be later used by software engineers to understand where and why a particular computation appears. We also show the effectiveness of our approach through a case study.

Acknowledgment

I would like to thank all the people who have helped and inspired me during my entire graduate study.

I am heartily thankful to my supervisor, Dr. Abdelwahab Hamou-Lhadj, for his supervision, advice and guidance from the initial to the final level of understanding of the subject during my research at Concordia University.

I owe my gratitude to Heidar (Amir) Pirzadeh for his crucial contribution which made him a backbone of this research. I am grateful for his willingness to share his bright ideas with me which helped me a lot in shaping up my research.

I am indebted to my friends and colleagues including Agam, Ali, Maher and Walid who made themselves available to help me throughout by sparing their precious times for me, offering advice and suggestions whenever I needed them and have always been a constant source of encouragement in these two years.

Most importantly, none of this would have been possible without the love and patience of my family. I would like to thank my family for their gentle love and support throughout my life and for showing persistent confidence in me.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Problem and Motivation

Understanding the behavioural aspects of a software system can help in many software engineering activities such as debugging, adding new features to an existing system, or simply understanding what the system does and why it does it this way. This is particularly important for those systems with poor documentation and for which the initial designers have left the company taking with them valuable knowledge about the system.

The behaviour of software system is typically represented in the form of execution traces. There exist several types of traces including traces of routine (method) calls, traces of inter-process communications, statement traces, etc. In fact, one can trace any aspect of the system depending on the task at hand. Traces, however, have historically been difficult to work with. The challenge is that they tend to be extremely large, often hundreds of thousands lines. There is a need for techniques to simplify the content of large traces to facilitate their analysis. Recently, there has been a noticeable increase in the number of studies in the area of trace abstraction and simplification [Moonen 08, Hamou-Lhadj 05, De Pauw 98, Jerding 97, Renieris 99, Malony 91, Jerding 98]. These techniques, however, suffer from several limitations including the fact that they rely extensively on user input and that most of these techniques rely on some sort of visualization scheme, which limit their reuse [Hamou-Lhadj 04].

The objective of the study presented in this thesis is to develop techniques to facilitate the analysis of large execution traces in order to help software engineers understand the main behaviour of the traced program, which in turn can enable software engineering tasks that require some understanding of the system behavioural aspects. For example, a software engineer who wishes to improve an existing feature of a poorly documented system will most likely need to understand how the feature is implemented before making any changes that preserves the system's reliability. He or she can then generate a trace by exercising this feature and proceed to understanding and analyzing its content to build an initial understanding of how the feature is implemented. This understanding aims to compensate for a lack of proper documentation and access to system experts.

In this thesis, we propose a novel approach to simplify the analysis of large traces by automatically extracting the main execution phases they contain. We define an execution phase as any part of a trace that performs a specification task including initialization of variables, specifications computations, etc. By doing this, we transform the trace from a mere raw of events to a more meaningful sequence of phases that can be readily explored by a software engineer to understand different parts that comprise a trace at a higher-level of abstraction.

Our algorithm for the automatic detection of execution phases is based on the fact that a phase shift within a trace appears when a certain set of events responsible for implementing a particular task and which are prevalent in one phase, start to "fade" as the program enters a new phase, where new events start to appear. In addition to this, our phase detection technique operates on the trace while it is generated (i.e., online). This is contrasted with the post-mortem analysis of a trace and which requires the trace be first generated in its entirety before any processing is applied. This offline approach has the obvious shortcoming of having to store the entire trace although it may only be necessary to explore part of it. Our phase detection approach is also automatic to a great extent relieving users from heavy intervention that is not desirable when working with traces.

The traces on which we focus on in this thesis are traces of routine calls. By routine, we mean a procedure, function, or method. Our approach applies to procedural and object-oriented systems and it is language-independent as long as the programming language used to develop the system support the concepts of routines.

1.2 Research Contributions

The main research contributions of this thesis are as follows:

- A novel trace abstraction technique based on the idea of dividing a large trace into meaningful segments, called execution phases, which reflect the main tasks of the traced program. Our approach is automatic and operates on the trace as it is generated.

- A novel algorithm for extracting execution phases from a trace. The algorithm is based on the idea that trace elements fade as new phases emerge.
- The phase detection algorithm has been applied to the execution traces generated from an object-oriented target software system in order to show the applicability of our approach.

1.3 Thesis Outline

The rest of the thesis is structured as follows:

Chapter 2 - Background

The thesis begins with the background literature review, including a brief overview of the topics that are related to our research, namely, reverse engineering, program comprehension, static and dynamic analysis. The remainder of this chapter contains a detailed survey of the existing execution phase detection techniques, including their advantages and disadvantages, which are followed by a general discussion in the end.

Chapter 3 – Phase Detection Approach

The phase detection algorithm is presented in this chapter. The chapter starts with the definition of execution phases followed by the overall approach which includes the approach diagram featuring the steps of our phase detection algorithm. The chapter continues with the detailed description of the feature-trace generation process and the two steps that constitute our approach, i.e. phase change detection and phase shift location. Next, we present a working example which shows how the algorithm is applied to detect the execution phases in a sample trace. The last section of this chapter concludes with a discussion on the applicability of our approach on real data.

Chapter 4 - Evaluation

This chapter introduces a case study which is used to evaluate the execution phase detection approach presented in the previous chapter. In the beginning of this chapter, the target system that is chosen for the case study is described which is followed by a discussion on the usage scenario based on which the trace has been generated. The quantitative and the qualitative results of applying our phase detection algorithm and the evaluation process are discussed in details. The chapter ends with a brief discussion of the approach.

Chapter 5 - Conclusions

We conclude the thesis in this chapter by revisiting the main research contributions. We also present opportunities for future research. The closing remarks are presented at the end of the chapter.

Chapter 2 Background

2.1 Related Topics

The topics related to our thesis include reverse engineering, program comprehension, static analysis and dynamic analysis.

2.1.1 Reverse Engineering

Reverse engineering is concerned with investigating techniques and tools to help software engineers understand the complex legacy software systems [Nelson 96]. Unlike forward engineering, which involves the advancement from one step to another in the software development life cycle, the process of reverse engineering is to go in a reverse direction, starting from the implementation phase to gathering the requirements and hence trying to get the structural and behavioral aspects of existing software systems by building several static and dynamic abstract models [Nelson 96]. Reverse engineering can be achieved by gathering all the software components, identifying their inter-relationships, and presenting these entities at higher levels of abstraction. There are particularly four types of reverse engineering processes [Nelson 96]. They include:

1. ***Re-documentation:*** Re-documentation is the simplest and the oldest form of reverse engineering. Several legacy systems are very poorly documented and understanding their artefacts such as source code and the other information is a difficult task. Hence re-documentation came into existence. It is the process of transforming the old code, documents related to the code and the programmer's knowledge into a new or updated form of documentation which can be textual or graphical [Nelson 96]. This form of reverse engineering is responsible for correction of system documentation at the same level of abstraction. Re-documentation is an important activity as the software engineers need to refer to the program documentation to understand what the code is doing and why it is doing it this way.

2. ***Design Rediscovery:*** The main purpose of this form of reverse engineering is to re-design a model of the system at a higher level of abstraction using the same domain knowledge and documentation, along with the source code.

3. ***Restructuring:*** It involves the transformation of a system to another representation at the same level of abstraction, rather than abstracting it to a higher-level, while preserving its functionality and behaviour. Restructuring improves the quality attributes of the software products by re-organizing the logical structure of existing software systems [Arnold 89]. For example, the GOTO statements which were heavily used in the older software of Cobol or Fortran are now being replaced with their modern equivalents such as loops and conditional statements. The other

examples include editing documentation, rearranging the code by renaming variables, abstracting functions etc. These changes greatly improve the readability of software programs.

4. ***Reengineering:*** Reengineering of a software system was described by Chikofsky and Cross as “the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction” [Chikofsky 90]. While reverse engineering advances from the low-level program code to a higher-level of abstraction, reengineering makes use of the increased understanding to re-implement the code in a new form [Rugaber 95]. Hence, reengineering can be defined as a process of modifying the software system by adding a new functionality to it or by rectifying the existing errors after the system has been reverse engineered.

We believe that the approach presented in this thesis can help with many of the above reverse engineering and reengineering tasks. For example, the extracted phases from a large trace can be further refine to recover the behavioural design diagram of the traced scenarios, which in turn can serve many purposes including documenting the design, helping in restructuration efforts and so on.

2.1.2 Program Comprehension

According to Rugaber, program comprehension is the process of acquiring knowledge about a computer program in order to perform certain activities on it such as error correction, reuse, system enhancement and documentation [Rugaber 95]. Biggerstaff et al. define program comprehension as "A person understands a program when he or she is able to explain the program, its structure, its behaviour, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program" [Biggerstaff 93].

It is a research area which led to the development of several revere engineering tools and techniques to help software engineers understand legacy software systems. It is very much required to understand the software sufficiently before it can be modified because maintaining the systems totally depends on understanding the structure of the program. The main problems that the programmers face today are the difficulties in understanding existing code, due to its unfamiliarity and the lack of proper documentation. A large part of the software maintenance process is devoted to comprehend the system that has to be maintained. Fjeldstad and Hamlen reported that 62% of the time and effort spent on understanding, enhancement and correction tasks are devoted to comprehension activities. These activities involve reading the documentation, scanning the source code, and understanding the modifications to be made [Fjeldstad 83]. It is therefore very crucial for programmers to have a deep insight of the software they have to modify in order to maintain it. In short, the better programmers understand the software system at hand, the better will be the maintenance process, and hence, software development will be improved.

One of the main objectives of this thesis is to assist software engineers in the program comprehension process with a focus on the understanding of how the system behaves, instead of what the system looks like. We achieve this by allowing them to map the execution phases detected by our program to the specific code that implement the features corresponding to those execution phases.

2.1.3 Static and Dynamic Analysis

Static analysis of software systems is performed without actually executing the program. The static information obtained by the static analysis of software systems describes the structure of the software and reveals the properties that hold for all possible system executions. This information is extracted by analyzing the source code and can be viewed using several reverse engineering tools like Rigi [Müller 88]. This static information is composed of the artefacts contained in the program and the relationship between them. For example, in the case of Java, these artefacts could be packages, classes, methods, variables etc. Based on the dependencies between these artefacts, static dependency graphs are constructed which can further be employed for various studies.

Dynamic analysis of the software programs (the focus of this thesis) is performed by actually executing the program to understand the run-time behaviour of the software system. Another definition of dynamic analysis is presented by Ball: “dynamic analysis is the analysis of the properties of a running software system” [Ball 99]. The major difference between static and dynamic analysis is that in static analysis the system properties hold true for all the executions whereas in dynamic analysis the properties for each execution hold only for the executed scenarios.

As we mentioned in the introductory section, the information that is generated from executing a software system takes generally the form of execution traces. Other run-time information such as system profiles are another form but they are most used to analyze the system performance and are, therefore, outside the scope of this thesis. Traces contain a record of the events that take place while the program is executed. For example, the routine (method) call traces constitute a trail of methods where each method is called by another in a sequential fashion. Depending upon how the probe is written, a variety of information can be obtained in a trace, apart from just acquiring the names of the methods, such as the nesting level of methods, the nature of each method whether it is public, private or protected, etc. In this thesis, we focus on traces of method (routine) calls, leaving other types of traces for future research.

The run-time information can be generated in different ways including source code instrumentation (done automatically), which requires modification of the target system. Instrumenting of the execution environment is another possible alternative, which neither requires the modification of source code nor the presence of code itself. Figure 2.1 shows a typical way on how traces are generated.

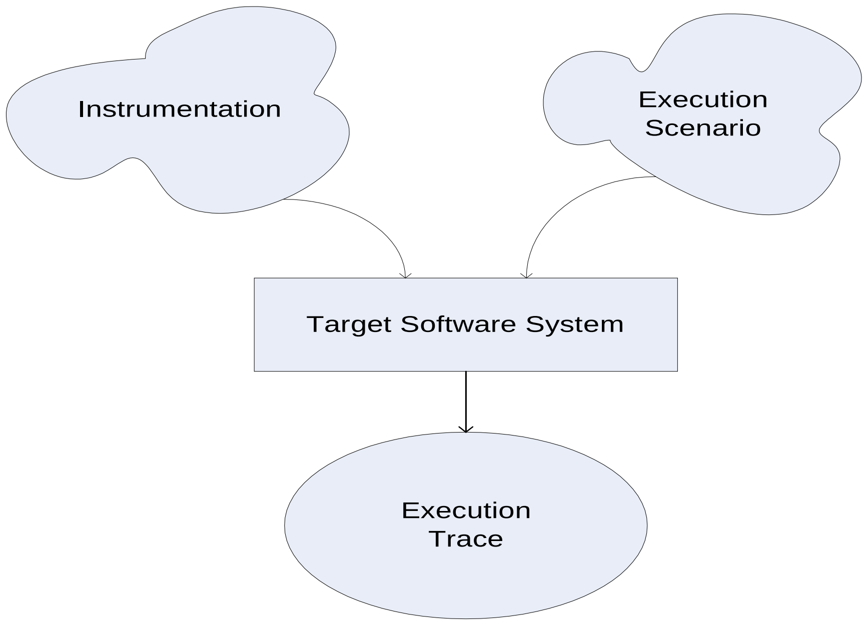

Figure 2.1 Execution trace generation

2.2 A Survey of Existing Phase Location Techniques

In this section, we present a survey of the most cited execution phase detection tools and techniques. Although we did not attempt to examine and include all the studies that exist in the previous literature, however, we believe that the ones presented in this section reflect the current state of art in phase detection approach.

Steven P. Reiss [Reiss 05] introduced the concept of dynamic detection and visualization of software phases. He created a software visualization tool called as JIVE which helps software engineers understand the behaviour of a software system by providing them a high level view of what actually is happening inside the target Java system. The main task of JIVE is to summarize the information of execution after a certain period of time, for example, in the interval of 10 milliseconds. This execution information is comprised of the numbers of calls made by methods of one class or a group of classes, the information about objects being allocated and destroyed and the information about the behaviour of the threads occurring in different parts of execution. There exists a similarity between their technique and our approach which is that both approaches work dynamically, i.e. the phases are determined while the program is being executed. However, one of the drawbacks of determining the phases through their technique is that the generation of the phases is greatly defined by the programmer, whereas our method is almost fully automatic.

In [Watanabe 08], Watanabe et al. proposed a novel technique to detect phases in the execution traces of large object-oriented software programs by using a Least Recently Used (LRU) Cache for observing the objects which are prepared at the beginning of the phase and are destroyed with the end of the phase. They define a phase as a consecutive sequence of run-time events where some phase can correspond to a feature and the other phase may represent a minor phase. Their approach is somewhat similar to our phase detection technique in a way that they frequently update the LRU cache when the objects responsible for a new phase are assembled. To achieve the visualization, they integrated their approach to a sequence diagram visualization tool called Amida which automatically detects the phases and visualizes them in the form of sequence diagrams. Dealing with object creation and deletion, however, poses serious challenges to the scalability of the approach. In this thesis, we focus on method call traces. We also present a different algorithm than the one presented by Wanabe et al.

In [Cornelissen 08], the authors were concerned with developing techniques that allow the visualization of data which is gathered at run-time from a software system in a summarized way, while still maintaining the integrity and readability of data. In order to achieve this, they presented two views of a software system: circular bundle view and the massive sequence view. In the former view, all the structural elements which comprise a software system are projected on the circumference (outline) of a circle in a nested fashion and are then viewed while their inter-relationships are drawn in the middle of the circular bundle. These relationships are then bundled together to avoid visual clutter and hence improve scalability. If the edges in a certain portion of the circumference are thicker, it indicates that most of the activities are centered around these calls. Another view that is described by the authors is the message sequence view, also named as message sequence charts in which the entities of a software system are arranged in an orderly fashion. This view greatly supports the readability by displaying all the information in a vertical manner. But on the other hand, if there is an extremely large amount of information, then this type of arrangement creates a problem in navigation. The massive sequence view indicates that there are three major "phases" in the execution scenario. They are the input phase, calculation phase and an output phase. The authors have used several zoom-in and zoom-out techniques to visualize the circular bundles. The difference between their technique and ours is that they focus on the visualization tools and techniques to give a representation of execution phases, whereas, our technique automatically detects and locates the phases in an execution trace no matter which visualization technique is used.

Chapter 3 Execution Phase Detection Approach

In this chapter, we present our approach for detecting and locating the various execution phases that constitute an execution trace. The rest of the chapter is organised as follows: In Section 3.1, we define in more detail what we mean by execution phases. The overall approach of detecting phases in large traces including the process of generating traces, detecting phase changes, and locating where the phase shift occur is presented in Section 3.2 which is followed by a brief summary of the chapter in the last Section 3.3.

3.1 What is an Execution Phase?

We define an execution phase as a segment of a trace that performs a particular computation such as initializing variables, executing a specific algorithm, and so on. Wantanabe et al. describe a phase as a feature that represents the functionality of a program at higher levels of abstraction [Watanabe 08]. They also state that it is suitable to divide a large execution trace into smaller execution phases before performing any further processing of the trace content. This can assist software developers in understanding the content of a trace by focusing on smaller segments (i.e., its execution phases) instead of going through the entire trace.

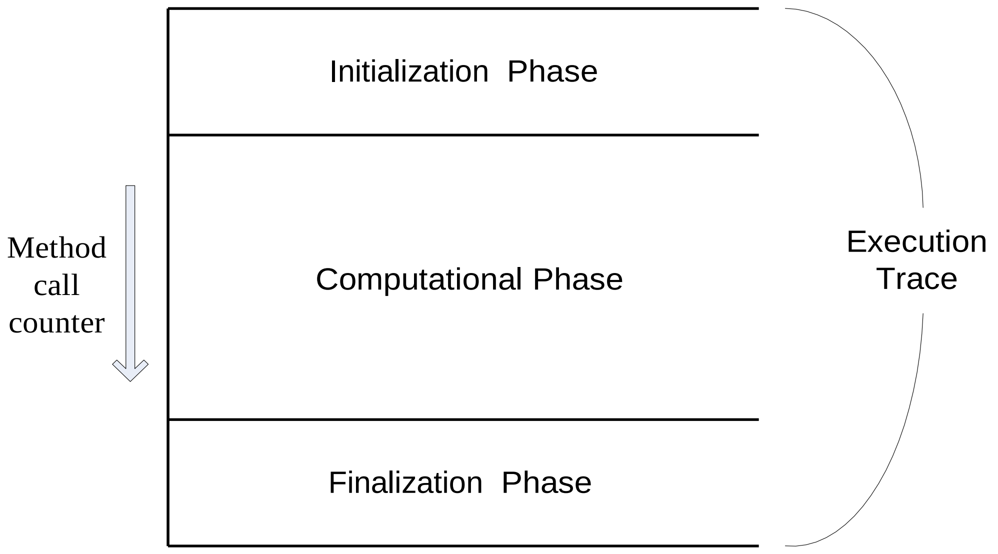

Figure 3.1 Execution phases in a trace

At a high-level, a program run may involve three main phases (Figure 3.1): Initialization phase, main computation, and a finalization phase. These phases can be further divided into sub phases revealing more details about what the program is doing. Our objective, in this thesis, is to propose an algorithm that takes a method call trace as input and automatically extracts its main execution phases at various levels of abstraction. The idea behind our phase detection approach, which is described in more detail later, is to detect when and where during the execution of a program, execution phases appear. Since a phase implements a particular computation, it is therefore reasonable to assume that it has some components that distinguish it from the other phases. In other words, while browsing a trace, the methods that implement a particular phase start to "fade" as new methods begin to emerge, indicating the beginning of another phase.

In addition, our proposed phase detection algorithm operates on a trace while it is generated (i.e., on the fly). This is contrasted with an offline approach, where the entire trace is first collected before applying the algorithm. Online processing of traces is usually more desirable than an offline approach since the users can see the results early and may need to make decisions based on this early feedback without having to wait until the entire trace is generated.

3.2 Overall Approach

Figure 3.2 shows a general overview of our approach for detecting and locating the execution phases in a trace. First, the system is instrumented and a trace is generated by exercising the scenario under analysis. The two fundamental steps of our phase detection algorithm are: Phase Change Detection and Phase Shift Location. The objective of the phase change detection is to estimate whether the methods which are prevalent in one phase have begun to disappear as new ones have started to appear.

Once a phase change is detected, the phase shift location step consists of detecting the exact location of the phase transition. It is desirable to know the exact location of a phase shift in the trace in order to distinguish the different phases from one another. This is obtained by detecting where exactly the methods belonging to one phase have started to effectively fade or completely disappear leaving their place to new methods belonging, presumably, to the next phase. The components of our approach are explained in more detail in the subsequent sections.

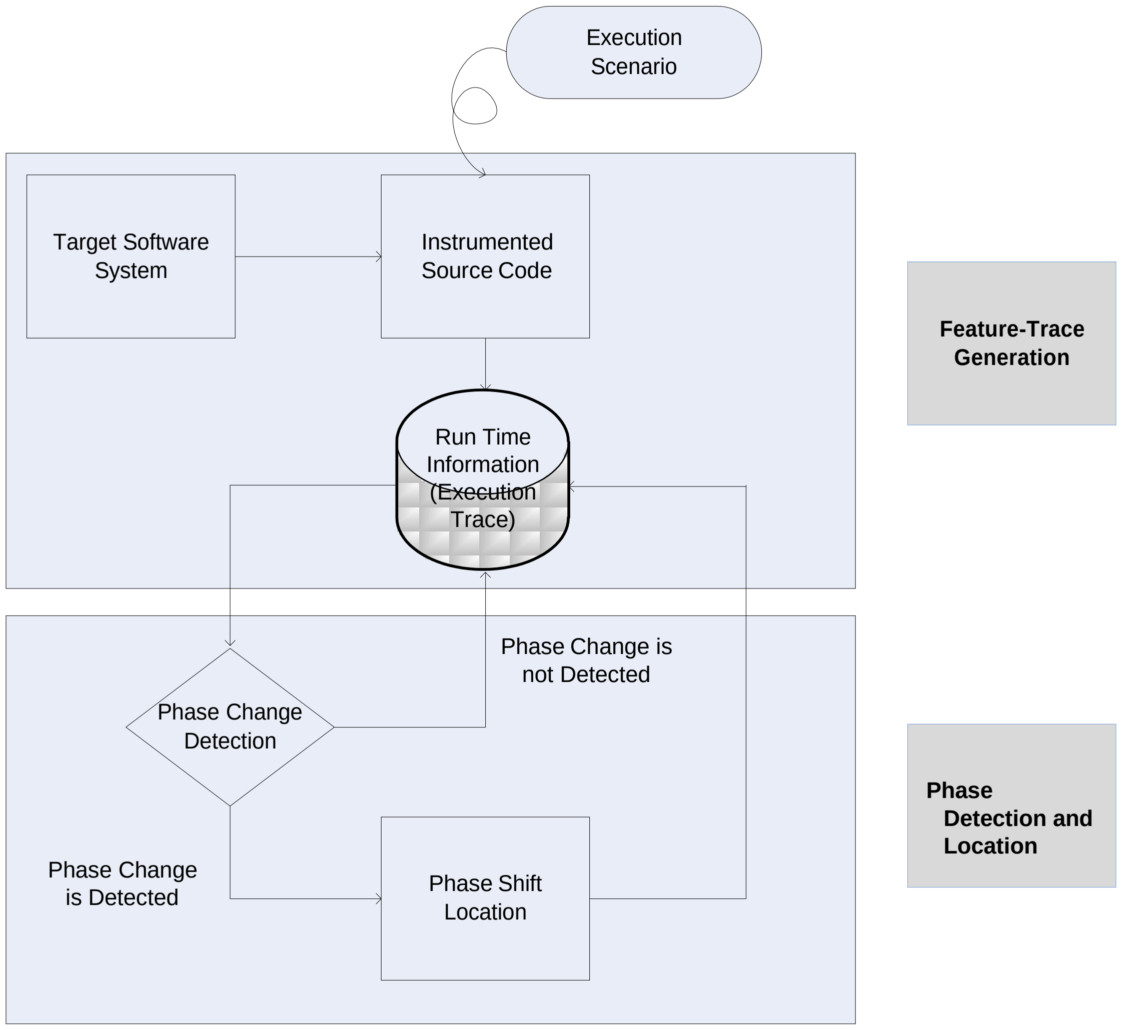

Figure 3.2 Overall approach diagram

3.2.1. Feature-Trace Generation

The first step of our approach is to generate a suitable feature-trace which is obtained by exercising an execution scenario that involves the execution of several essential features under study. For example, for a drawing tool such as the JHotDraw software system (the system used for the case study), the features may include laying the foundation of a drawing, drawing a figure thereafter which could be followed by inserting more figures and animations, modifying these figures, deleting them etc. We have used source code instrumentation to generate traces because of its simplicity and the availability of tool support.

3.2.2. Phase Change Detection

The phase change detection step aims to detect a shift from one set of frequently appearing methods to another set of newly introduced methods. As the program executes, a set of distinct methods are captured in a set which is called a *working set (WS).* The prevalence of the methods in a working set is computed based on the order in which they are invoked and then they are arranged in a descending sequence of their prevalence. That means, the methods with high prevalence value appear at the beginning of the working set whereas the less frequent methods come in the end. The way the prevalence is computed is presented later in this chapter.

As new methods appear in the execution of the program, the working set is constantly updated so that it can reflect the changes in the program's behaviour. However, updating the working set on each new method invocation can be relatively expensive in terms of computations. To alleviate this, we propose updating the working after a certain number of method calls occur. We call this the *chunk of method calls.* Therefore, the update rate of the working set depends on the chunk size which is provided as an input to the phase change detection program (depicted in Figure 3.3). This chunk size is a variable which can be set to different values by the user.

```
1   PhaseFinder($Chunk_i$: chunk of methods, T:threshold)
2   {
3      if (i == 1)   // If it is the first chunk of the trace
4          WS = new workingset()
5      for each method m in $Chunk_i$
6      {
7          if m is not in WS
8          {
9              WS.add(m)
10         }
11         WS.rank_methods()  // using the methods prevalence
12     }
13     $Snapshot_i$  = WS
14     if (i == 1) //$Snapshot_0$ is created when the first chunk is processed
15         $Snapshot_o$  = $Snapshot_i$
16     Distance = compare ($Snapshot_o$, $Snapshot_i$)
17     if (Distance < T)
18     {
19         for each candidate m // This part is used for phase shift location
20         {
21             for every chunk in {$Snapshot_o$ ... $Snapshot_i$}
22                     if m.rank(chunk) is close to mid-rank
23                             chunk.vote()
24                return (chunk with maximum votes)
25         }
26         $Snapshot_o$  = $Snapshot_i$
27     }
28   }
```

Figure 3.3 The pseudo code of phase finding algorithm

In order to detect a phase change, the methods of the current snapshot of a working set are compared with the methods contained in the original snapshot of working set (lines 16-18 of the algorithm). If less than a certain threshold, *T*, of the methods of the original working set appear in the current working set, then this suggests that a phase change has taken place since this means that new methods are now becoming more prevalent than the ones already in the working set. Determining the threshold *T* in advance is not possible since it might depend on the application. It is given in our algorithm as input. In practice, the tool that supports our approach should be able to allow enough flexibility to vary the threshold *T*. In Section 3.3 of this thesis, we propose a technique for determining proper threshold *T* and the chunk size that would lead to an adequate set of phases.

Another decision we made in our algorithm is concerned with the ways working sets are compared. Instead of comparing all the methods of the current working set, one possible optimization is to compare only a few of them which have the highest ranking (i.e. the ones that appear in the beginning of the working set). The number of methods that are to be compared can be equal to the chunk size since, in the worst case scenario, the number of new distinct methods that can be found in a new chunk will always be less than or equal to the chunk size.

Figure 3.4 shows an example of a routine call sample trace that will be used to illustrate the algorithm. Considering that the chunk size is set to 3, the original working set WS will contain the first 3 methods A, B and C of the trace. These methods are sorted in a descending order based on their ranking (prevalence).

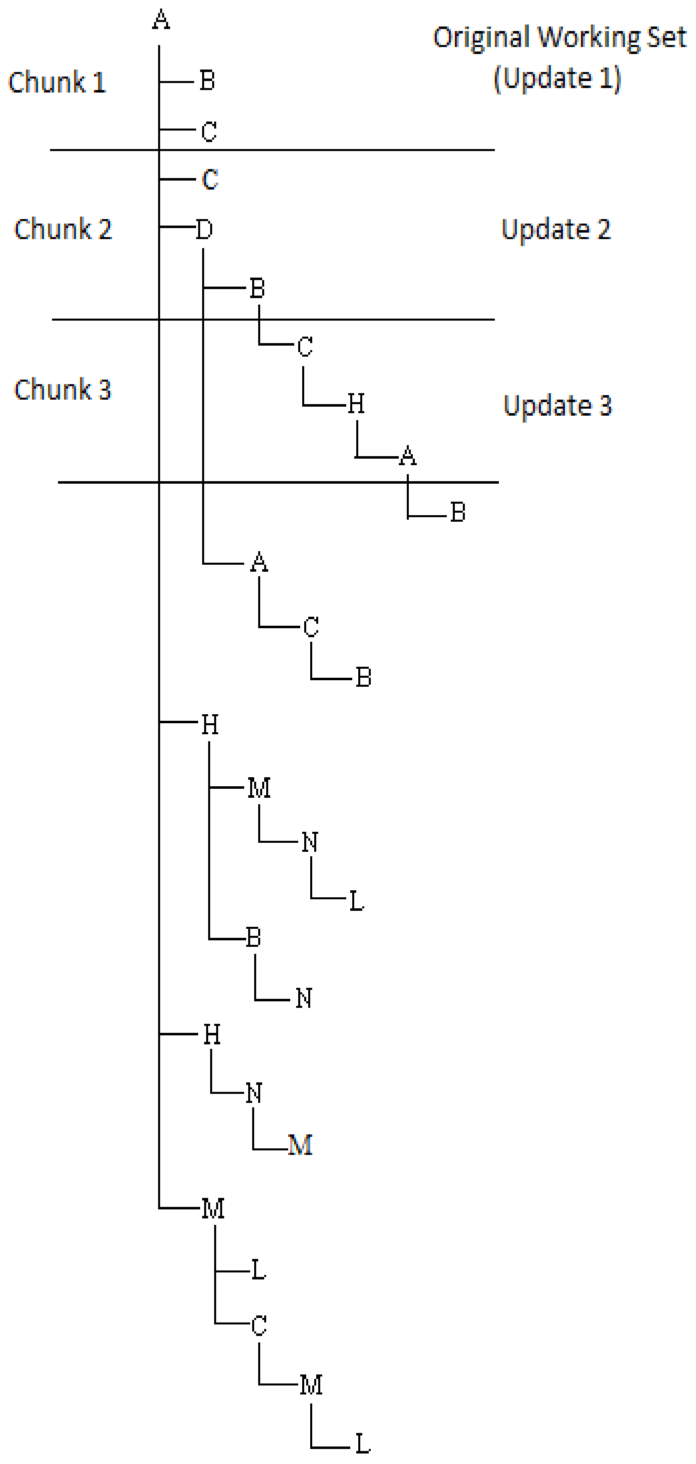

Figure 3.4 A sample routine (method) call trace

The prevalence function takes into account the frequency of a method in part of the trace that has been processed so far (frequency(m)), the first chunk number in which the method was first introduced (first_chunk(m)), the current chunk number under process (curr_chunk) and the chunk size (chunk_size). The complete equation is as follows:

$$P(m) = \frac{frequency(m)}{(current\_chunk - first\_chunk(m) + 1) * chunk\_size}$$

This equation keeps track of the prevalence of all the methods as the algorithm advances through the chunks of the trace. If a set of methods keep appearing relatively at the similar rate after each chunk is processed, then this is a good indicator that the program is still in the same phase. If some of the methods start fading based on a certain threshold percentage, then this is an indication of the beginning of a new phase.

When applying the prevalence function to the methods of Chunk 1 in the trace of Figure 3.4 we obtain *P(A) = 1/{(1-1)+1}*3 = 1/3*. Likewise, the prevalence of methods B and C is also 1/3. This is so because all the three methods A, B and C appear only once in the first chunk. Since the prevalence of each method in Chunk 1 is the same, therefore, they all are assigned the same rank 1. The working set that is obtained after processing the first chunk of the trace is *{A, B, C}*. The content of the first working set is acknowledged as the original working set and it is updated with the processing of the following chunks of methods while the trace is being generated, therefore giving rise to different snapshots of the original working set.

As the algorithm processes the upcoming chunks, it updates the working set by adding the newly encountered methods, computes their prevalence and assigns them a ranking based on the corresponding prevalence. For instance, the Chunk 2 of the trace in Figure 3.4 contains methods C, D and B. On recomputing the prevalence of all the methods and updating the working set, we obtain:

*P (A) = 1 / {(2-1) + 1}*3 = 1/6.*

*P (B) = 2 / {(2-1) + 1}*3 = 2/6 = 1/3.*

*P (C) = 2 / {(2-1) + 1}*3 = 2/6 = 1/3.*

*P (D) = 1 / {(2-2) + 1}*3 = 1/3.*

The first update of the working set is *{B, C, D, A}*. It shows that the method A, which appeared at the first position in the previous (original) working set, now occupies the last position which indicates that it has gradually started to fade, whereas, methods B and C still have some strength. Each time we update the working set, we compare it with the original snapshot of working set and if there is a significant change between the original working set and the current one, then this indicates the beginning of a new phase. Assuming that the threshold *T* is set to 20% in this example, the methods B and C in original working set *{A, B, C}* do appear at the beginning of the current snapshot of working set *{B, C, D, A}*. That means, a phase change has not been detected so far. Hence, the process of updating the snapshots of working sets continued until less than 20% of the methods in original working set appear in the current snapshot of the working set.

Table 3.1 Phase Change Detection Example

Chunk no.	**Working set name**	**Methods introduced in chunk**	**Snapshots**	**Phase Shift Detected**
1	Original working set (Snapshot 1)	A, B, C	{A, B, C}	×
2	Snapshot 2	D	{B, C, D, A}	×
3	Snapshot 3	H	{C, H, A, B, D}	×
4	Snapshot 4	No new methods	{C, A, B, H, D}	×
5	Snapshot 5	M	{M, B, C, H, A, D}	×
6	Snapshot 6	N, L	{N, L, B, C, M, H, A, D}	×
7	Snapshot 7	No new methods	{N, B, H, C, L, A, M, D}	×
8	Current working set (Snapshot 8)	P	{N, P, L, B, H, C, M, A, D}	✓

Table 3.1 shows the snapshots of the working sets that correspond to each chunk. With a threshold set to 20%, a new phase will be detected, in this example, after the Chunk 8 is processed. This is so because none of the methods of the original working set appear in

the first 3 methods of the current working set (snapshot 8). Hence, Chunk 8 is a point of phase detection.

3.2.3 Phase Shift Location

Once we detect that there is a phase in the trace, it is now required to determine the exact location of phase transition. In order to achieve this, the distinct methods appearing in all the working sets, starting from the original working set to the one in which the phase is detected, are grouped together in what we call the O*bservation Set*. The observation set resulting from the previous example is: *{A, B, C, D, H, M, N, L, P}* since these are the methods that appear in the working set where the phase has been detected.

The next step is to find the exact chunk in the trace where most of these methods start to fade. If we consider the fading of a method *m* as it is going from its best rank (somewhere in one phase) to its worst rank (somewhere in another phase), then we presume that the starting point where the ranking of the method *m* starts to decline should be somewhere in the middle. We call this point the mid-rank point which we compute as follows:

$$midrank(m) = \frac{lowestrank(m) + highestrank(m)}{2}$$

The lowest and highest ranks represent the worst and best ranks of a method *m* in any working set where the method appears. Table 3.2 shows a list of methods in the observation set and their mid-rank points. For example, a method *A* has the lowest rank in Chunk 8 and the highest rank in Chunk 1. Therefore, its mid-rank point is 4.5 (i.e. (8+1)/2). Now we have to find the chunks in which the rank of method *A* is closest to its mid-rank. It can be observed that the ranking of *A* is close to 4.5 in Chunk 2, Chunk 5 and Chunk 6 (see Figure 3.5). For each method in the observation set, we list the chunks in which the method reaches its mid-rank point (as shown in Table 3.2).

Table 3.2 Mid-rank value of the methods of Figure 3.4

Method call	**Mid-rank**	**Chunks where the rank of the method is close to the mid-rank value**
A	4.5	Chunk2, Chunk5, Chunk6
B	2.5	Chunk3, Chunk4, Chunk5, Chunk6, Chunk7
C	3	Chunk5, Chunk6, Chunk7
D	6	Chunk3, Chunk4
H	4	Chunk7
M	5	Chunk6, Chunk8
N	1	Chunk6, Chunk7, Chunk8
L	3	Chunk8

P	1	Chunk8

We refer to the chunks in which a method *m* reaches its mid-rank point as the *voting chunk set* of this method. This set indicates the possible places where the method might start fading. For example, the voting chunk set of method C is {Chunk 5, Chunk 6, Chunk 7}. That means, C could have started to disappear in any of these chunks (see Figure 3.7). Similarly, the voting chunk set of methods A and B are {Chunk 2, Chunk 5, Chunk 6} and {Chunk 3, Chunk 4, Chunk 5, Chunk 6, Chunk 7} and are shown in Figure 3.5 and Figure 3.6.

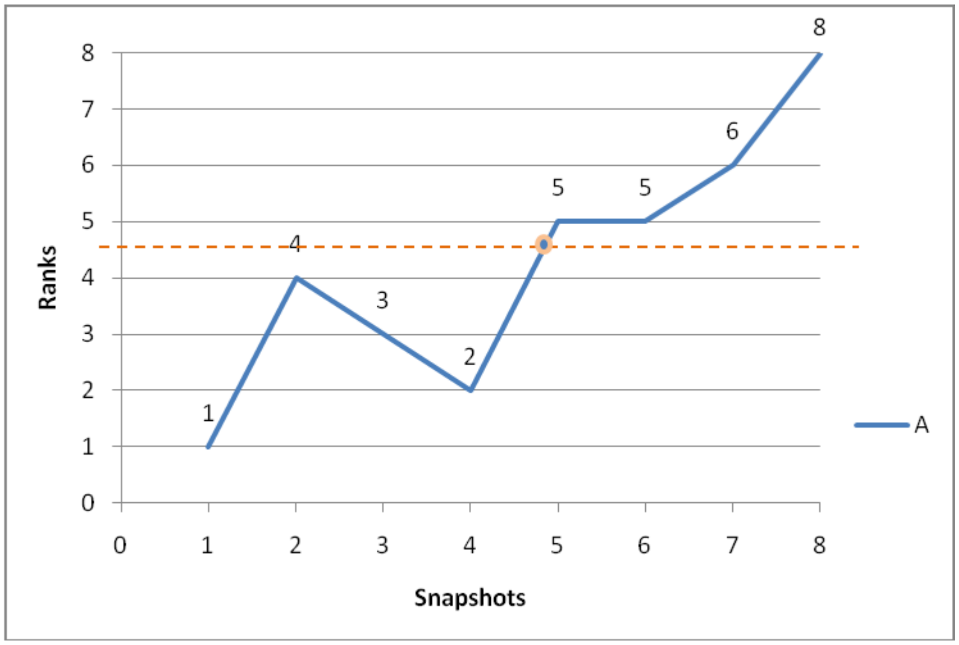

Figure 3.5 The rank of method A in each snapshot

Ranks of A = (chunk1: 1, chunk2: 4, chunk3: 3, chunk4: 2, chunk5: 5, chunk6: 5, chunk7: 6, chunk8: 8) mid-rank(A) = 4.5.

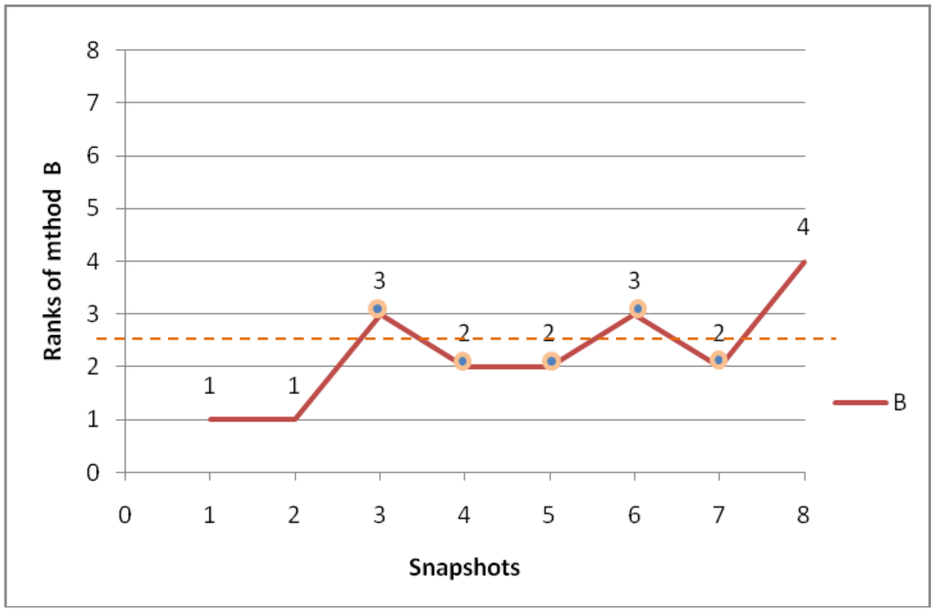

Figure 3.6 The rank of method B in each snapshot

Ranks of B = (chunk1:1, chunk2: 1, chunk3: 3, chunk4: 2, chunk5: 2, chunk6: 3, chunk7: 2, chunk8: 4) mid-rank(B)= 2.5

In order to find the phase transition, we simply need to compute the voting chunk set of all the methods in the observation set and locate the chunk that receives the highest vote (Lines 17 to 26 of the algorithm described in Figure 3.3). This is the chunk in which most methods of a phase have started to fade and therefore, this chunk will be considered as the location of phase transition.

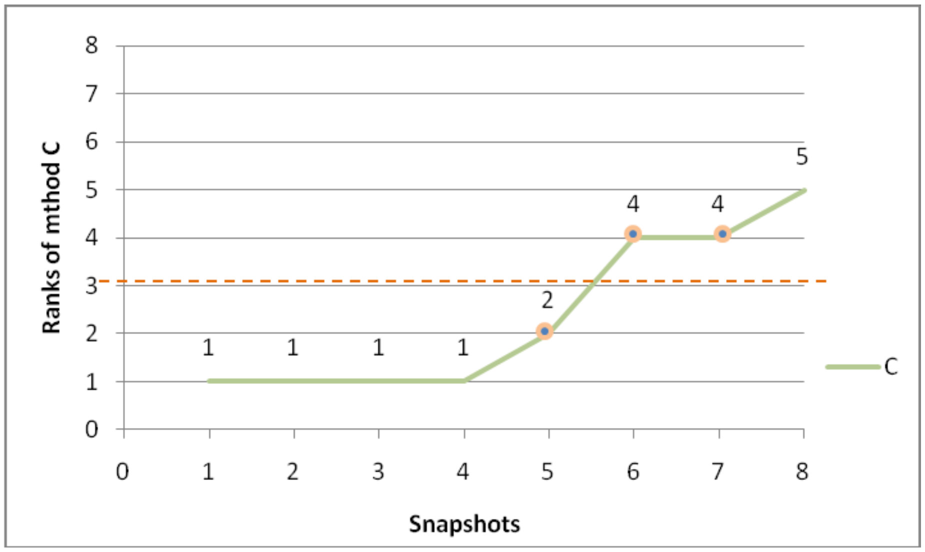

Figure 3.7 The rank of method C in each snapshot

Ranks of C = (chunk1:1, chunk2: 1, chunk3: 1, chunk4: 1, chunk5: 2, chunk6: 4, chunk7: 4, chunk8: 5) mid-rank(C)= 3

The results of the chunk voting for the sample trace in Figure 3.4 are shown in Table 3.3. The chunk that obtained the maximum votes is Chunk 6, which indicates that the phase transition has taken place in this chunk. If we look at the trace of Figure 3.4, we can see that starting from Chunk 6, most of the methods like A, B and C have started to appear less frequently whereas the new methods like H, M and N have started to emerge, therefore invoking a new phase. Hence, the overall approach detects the phase at Chunk 8 and locates the phase transition at Chunk 6.

Table 3.3 Phase shift location based on majority voting

Chunk no.	**Votes**	**Phase Shift**
Chunk1	0	✗
Chunk2	1	✗
Chunk3	2	✗
Chunk4	2	✗
Chunk5	3	✗
Chunk6	**5**	✓
Chunk7	4	✗
Chunk8	4	✗

3.2.4 Determining the chunk size and the threshold

As aforementioned, our approach depends greatly on the chuck size and the threshold *T* used to compare the content of the working sets. By varying these two variables, one may end up with different phases. The question is therefore: What would be the most suitable chunk size and threshold T for the application at hand?

To answer this question, we propose to vary the chuck size and the threshold T as follows:

- We vary the chunk size from 1 to the number of distinct methods invoked in the trace. This is the maximum number of methods that can form a chunk size when removing all repetitions.
- We vary the threshold *T* from 0% to 100%. This will cover all possible thresholds.

For each value of the chunk size and the threshold *T*, we extract the phases that have been identified. We refer to $phase\_set_{i,t}$ as a set of phases that have been uncovered with a chunk size equals to *i* and a threshold *T* equals *t*. Once all possible phase sets are identified (this is done automatically by simply applying the phase detection algorithm presented earlier), we measure the similarity between the phases contained in each set. A good phase set should be the one where the phases are most distinct from each other. This is based on the definition of an execution phase where an execution phase should represent a particular computation of the traced scenario. Therefore, the extracted phases should be as dissimilar as possible. We acknowledge that they will always contain some common components such as utilities, but there should also be components that are only proper to each phase. A better approach might require automatic removal of utilities using techniques such as the ones presented by Hamou-Lhadj et al. [Hamou-Lhadj 06].

To measure similarity between phases, we propose the concept of general similarity estimation *(GSim)* which is obtained by computing the average of the similarities between all the individual phases. We have developed a similarity metric which measures the commonality between two phases based on the distinct methods they have. Our general similarity estimation is computed as follows:

1. We first measure the similarity between every pair of phases in $phase\_set_{i,t}$
2. The general similarity is then the average of the similarities measure in (1)

To measure the similarity between every pair of phases, we simply divide the number of distinct methods that are common between the two phases to the total number of distinct methods contained in both the phases. For instance, consider that the number of phases that is detected in a trace is three P1, P2, and P3. We also refer to number of distinct methods contained in a phase as DM (Phase), then the general similarity of these phases is computed as follows:

$$Sim_{12} = \frac{|DM(P_1) \cap DM(P_2)|}{|DM(P_1) \cup DM(P_2)|}$$

$$Sim_{13} = \frac{|DM(P_1) \cap DM(P_3)|}{|DM(P_1) \cup DM(P_3)|}$$

$$Sim_{23} = \frac{|DM(P_2) \cap DM(P_3)|}{|DM(P_2) \cup DM(P_3)|}$$

$$GSim = \frac{Sim_{12} + Sim_{13} + Sim_{23}}{3}$$

In the above equations, *DM(P_1)* and *DM(P_2)* are the number of distinct methods contained in Phases 1 and phase 2 respectively. Therefore, Sim_{12} is the fraction of number of distinct methods present in Phases 1 and 2 over the total number of distinct methods that are contained in both the phases. Likewise, the similarities between Phase 1 and Phase 3, Phase 2 and Phase 3 are also computed. Once we have the individual similarities between all the three phases, the next step is to calculate the general similarity which is basically the average of all the previously computed individual similarities.

The lower percentage of general similarity indicates that the individual similarities between the phases are quite low as they have a few methods in common and hence they are more distinct to each other.

A more generalised formula of the general similarity is shown as follows:

$$GSim = \frac{(Sim_{12} + Sim_{13} + \ldots + Sim_{1n}) + (Sim_{23} + Sim_{24} + \ldots + Sim_{2n}) + (Sim_{(n-1)(n)})}{Total\ number\ of\ phases}$$

Where,

$$Sim_{12} = \frac{|DM(P_1) \cap DM(P_2)|}{|DM(P_1) \cup DM(P_2)|}$$

$$Sim_{23} = \frac{|DM(P_2) \cap DM(P_3)|}{|DM(P_2) \cup DM(P_3)|}$$

$$Sim_{(n-1)(n)} = \frac{|DM(P_{n-1} \cap DM(P_n)|}{|DM(P_{n-1}) \cup DM(P_n)|}$$

Once the general similarity is computed for all possible phase sets (by varying the chunk size and the threshold *T*), we need to select the phase set with the minimum general similarity (meaning that most phases are distinct from each other). However, the issue is that there may be many phase sets that have low similarity. To select the best possible phase set, we apply another similarity metric to measure the changes that appear as one goes from one phase to another. In other words, we need to look into how continuous phases vary. We call this the measure of *consecutive similarity*.

The difference between general similarity and consecutive similarity is illustrated in Figure 3.8.

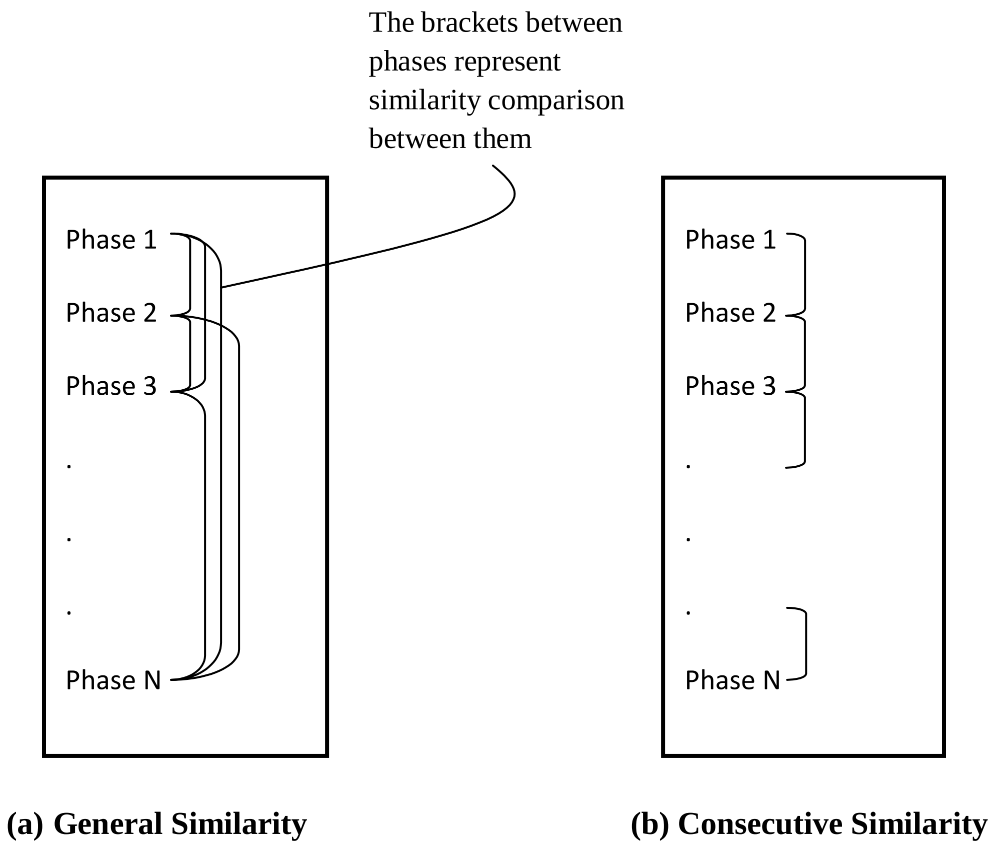
The brackets between phases represent similarity comparison between them
Phase 1
Phase 2
Phase 3
.
.
.
Phase N
(a) General Similarity
Phase 1
Phase 2
Phase 3
.
.
.
Phase N
(b) Consecutive Similarity

Figure 3.8 The difference between general and consecutive similarity

To measure the consecutive similarity in the set phase_set$_{i,t}$, we simply compute the summation of similarities (Sum_{cs}) between the consecutive phases and compute the average (Avg_{cs}). The phase_set$_{i,t}$ with the lowest average value is the one that indicates that the detected phases are highly disassociated from each other.

The equations for computing the consecutive similarity are shown in what follows:

$$Sum_{CS} = Sim_{12} + Sim_{23} + Sim_{34} + \ldots + Sim_{(n-1)(n)}$$

$$Avg_{CS} = \frac{Sum_{CS}}{Total\ number\ of\ Phases}$$

The lowest value of the consecutive similarity indicates the best phase set (including the chunk size and threshold T). The next step of our process is to validate the phases by looking at the traced scenario and how the phases reflect various computations. We show the effectiveness of the approach in Chapter 4.

3.3 Summary

In this chapter, we presented our approach of detecting and locating the execution phases that constitute a trace. The objective is to simplify the analysis of large execution traces. The approach is primarily composed of two consequent steps: Phase change detection and phase shift location. The main purpose of phase change detection is to determine if the methods implementing a particular phase which appear frequently have started to disappear with the invocation of newly introduced methods, indicating the beginning of another phase. Once a phase change is detected we locate where exactly in the trace the more frequent methods, that constitute one phase, start to fade, using the phase shift location.

After locating all the phases in an execution trace, we introduced a concept of general and consecutive similarities to estimate the best combination of chunk size and threshold that can produce adequate phases.

Chapter 4 Evaluation

In this chapter, we present a case study to evaluate the applicability and effectiveness of our phase detection approach by applying it to a trace generated from the execution of an object-oriented software system.

This chapter is organized as follows: the next section describes the target system that will be instrumented for the trace generation process. In Section 4.2, we present the usage scenario chosen to generate the execution trace. In Section 4.3, we show the evaluation process of applying our phase detection algorithm on the execution trace. The quantitative and the qualitative results of this case study are presented in the subsections of Section 4.3. A summary of our findings is presented in Section 4.4.

4.1 Target Systems

We have applied the proposed phase detection algorithm to a trace generated from a software system called JHotDraw (version 5.2). JHotDraw is open source software and a well known Graphical User Interface framework implemented in Java for technical and structural graphics [JHotDraw 5.2]. It consists of 9 packages, 148 classes and 1963 methods. JHotDraw 5.2 has 17,819 lines of code.

We selected the JHotDraw software system because it is well documented. JHotDraw packages and classes are documented and the detailed description of the architecture can be found on a website dedicated to the tool [JHotDraw 5.2]. The availability of this documentation helped validate the results obtained by using our approach.

4.2 Usage Scenario

In order to generate the execution traces, the target software system JHotDraw was instrumented using an Eclipse plug-in called TPTP (the Eclipse Test and Performance Tool Platform Project). TPTP is an open source platform which allows the software developers to build test and performance tools. The detailed description of this tool can be found on the website and the entire information of the plug-in and its download is provided on [Eclipse TPTP]. To instrument the system, the probes were inserted at each method entry and method exit of the source code. The scenario we selected to exercise JHotDraw consisted of a variety of drawing and animation features. The execution trace obtained as a result of executing the above scenario contained approximately 43962 routine calls (since we needed two events to generate a routine call, the trace size in terms of events was 87924 events). However, this trace contained a lot of noise such as mouse movements, get and set methods etc. For more precise results, we filtered these utilities to obtain a trace which was much cleaner. The resulting trace after removing noise consisted of 16261 routine calls.

4.3 Applying the Phase Detection Algorithm

In the JHotDraw execution trace which is chosen for our case study, the total number of distinct methods was 370. So we varied the chunk size from 1 to 370 with an interval of 10 and the threshold T was varied from 0% to 100% again with an interval of 10%. In order to detect a phase change, the methods in the current snapshot of working set were compared with the methods contained in the original snapshot of the working set. If less than a certain threshold of the methods of the original working set appeared in the current working set, then this suggests that a phase change had taken place as described in the previous chapter. Table 4.1 shows the results of applying the phase detection algorithm by varying the chunk size and the threshold T. The rows represent the chunk size and the columns represent the threshold. The cells contain the number of phased that have been detected and the result of the general similarity. For example, when the chunk size is 10 and the threshold T = 10%, we obtain 805 phases (meaning that phase_set$_{10,10}$ contains 805 phases) and a general similarity of 24%.

Table 4.1 Phase_sets$_{i,t}$ and general similarity for all chunk sizes and thresholds

T/ Chunk Size	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
10	805 (24%)	813 (23%)	814 (23%)	814 (23%)	814 (23%)	814 (23%)	814 (23%)	814 (23%)	814 (23%)	814 (23%)
20	359 (45%)	404 (41%)	405 (41%)	407 (41%)	407 (41%)	407 (41%)	407 (41%)	407 (41%)	407 (41%)	407 (41%)
30	265 (65%)	268 (64%)	270 (62%)	271 (62%)	271 (62%)	272 (62%)	272 (62%)	272 (62%)	272 (62%)	272 (62%)
40	16 (24%)	197 (86%)	202 (82%)	203 (81%)	203 (81%)	204 (81%)	204 (81%)	204 (81%)	204 (81%)	204 (81%)
50	9 (26%)	159 (85%)	161 (83%)	162 (82%)	162 (82%)	162 (82%)	163 (81%)	163 (81%)	163 (81%)	163 (81%)

60	3 (17%)	132 (86%)	134 (84%)	135 (83%)	135 (83%)	135 (83%)	136 (82%)	136 (82%)	136 (82%)	136 (82%)
70	2 (17%)	12 (28%)	114 (85%)	116 (82%)	116 (82%)	116 (82%)	116 (82%)	117 (81%)	117 (81%)	117 (81%)
80	2 (16%)	9 (25%)	100 (84%)	101 (83%)	102 (82%)	102 (82%)	102 (82%)	102 (81%)	102 (81%)	102 (81%)
90	2 (16%)	5 (22%)	90 (83%)	90 (83%)	90 (82%)	90 (82%)	90 (82%)	91 (81%)	91 (81%)	91 (81%)
100	2 (16%)	4 (25%)	9 (26%)	81 (82%)	81 (82%)	81 (82%)	81 (82%)	82 (80%)	82 (80%)	82 (80%)
110	2 (16%)	3 (21%)	8 (25%)	73 (83%	73 (84%)	74 (81%)	74 (81%)	74 (81%)	74 (81%)	74 (81%)
120	2 (15%)	2 (20%)	7 (23%)	67 (83%)	68 (81%)	68 (81%)	68 (81%)	68 (81%)	68 (81%)	68 (81%)
130	2 (15%)	2 (19%)	5 (17%)	8 (26%)	63 (81%)	63 (81%)	63 (81%)	63 (81%)	63 (81%)	63 (81%)
140	2 (14%)	2 (19%)	4 (11%)	7 (23 %)	58 (82%)	58 (82%)	58 (82%)	59 (79%)	59 (79%)	59 (79%)
150	2 (14%)	2 (18%)	3 (6 %)	6 (23%)	54 (82%)	54 (82%)	54 (82%)	55 (79%)	55 (79%)	55 (79%)
160	2 (14%)	2 (18%)	3 (9%)	5 (20%)	6 (13%)	51 (81%)	51 (81%)	51 (80 %)	51 (80%)	51 (80%)
170	2 (14%)	2 (18%)	3 (4%)	4 (11%)	6 (18%)	48 (80%)	48 (81 %)	48 (81%)	48 (81%)	48 (81%)
180	2 (14%)	2 18%)	2 (5%)	3 (4%)	5 (12%)	45 (82%)	45 (82%)	46 (79%)	46 (79%)	46 (79%)
190	2 (14%)	2 (18%)	2 (8%)	3 (4%)	5 (12%)	6 (15%)	43 (80%)	43 (80%)	43 (80%)	43 (80%)
200	2 (13%)	2 (19%)	2 (7 %)	3 (4 %)	4 (11%)	5 (13%)	41 (80%)	41 (79%)	41 (79 %)	41 (79%)
210	2 (13%)	2 (19%)	2 (8%)	3 (3%)	4 (12%)	5 (12%)	39 (79%)	39 (79%)	39 (79%)	39 (79%)
220	2 (13%)	2 (20%)	2 (7%)	3 (4%)	3 (5%)	5 (13%)	37 (81%)	37 (80%)	37 (80%)	37 (80%)
230	2 (13%)	2 (18%)	2 (7 %)	2 (4%)	3 (5%)	3 (4 %)	5 (14 %)	35 (82%)	36 (78%)	36 (78%)
240	2 (13%)	2 (18%)	2 (7%)	2 (4%)	3 (5%)	3 (4%)	5 (14%)	34 (80%)	34 (80%)	34 (80%)
250	2 (14%)	2 (17%)	2 (6%)	2 (5%)	3 (5%)	3 (4%)	5 (16%)	33 (78 %)	33 (78%)	33 (78%)
260	2 (13%)	2 (15%)	2 (6%)	2 (4%)	3 (6%)	3 (4%)	5 (12%)	5 (12%)	32 (78 %)	32 (78 %)
270	2 (15%)	2 (14%)	2 (5 %)	2 (4 %)	2 (5%)	3 (3%)	3 (4 %)	5 (13%)	31 (77%)	31 (77%)
280	2 (14%)	2 (13%)	2 (5%)	2 (4 %)	2 (5%)	2 (4%)	3 (4%)	5 (13%)	30 (76%)	30 (76%)
290	1	2 (13%)	2 (6%)	2 (4%)	2 (6%)	2 (5%)	3 (4%)	5 (13%)	5 (13%)	29 (76%)
300	1	2 (0%)	2 (5%)	2 (4%)	2 (5%)	2 (5%	3 (5%)	4 (11%)	5 (13%)	28 (75%)
310	1	2 (0%)	2 (4 %)	2 (3%)	2 (4%)	2 (0%)	3 (0%)	3 (5%)	5 (12%)	27 (75%)
320	1	2 (0%)	2 (2%)	2 (3%)	2 (3%)	2 (5%)	2 (4%)	3 (4 %)	4 (6%)	5 (10%)
330	1	1	2	2	2	2	2	3	4	5

			(0%)	(3%)	(3%)	(4%)	(3%)	(0%)	(8 %)	(13%)
340	1	1	2 (1%)	2 (0%)	2 (3%)	2 (4%)	2 (3%)	2 (4%)	3 (0%)	4 (12%)
350	1	1	2 (1 %)	2 (0%)	2 (3%)	2 (3%)	2 (2%)	2 (2%)	3 (0 %)	4 (14%)
360	1	1	1	2 (1%)	2 (1%)	2 (2%)	2 (2%)	2 (1%)	3 (0%)	4 (13%)
370	1	1	1	2 (1%)	2 (2 %)	2 (2%)	2 (3%)	2 (2%)	3 (0%)	3 (0%)

In the above table, we can notice several phase_sets$_{i,t}$ with lower general similarities containing only 2 to 3 phases. When we analyzed the content of the trace, we found that the resulting phases divide the trace into very high-level computations. For instance, when the chunk size is set to 360 and the threshold is set to 90%, the total number of phases detected in the trace is two. These are located at the chunk numbers 1 and 41, which means that these phase shift locations are dividing the trace into three major phases: the initialization phase containing the first 360 methods, which is followed by two major computational phases, one of them starting from approximately the 361st method to 14,760th method and the other computational phase starting from approximately the 14,761st method till the end of the trace. Although these phases can provide a high-level understanding of where the major phases occur, we do not think that they are sufficient to understand the content of a trace since a software engineer will most likely want to know more about what goes on in each phase. Therefore, there is a need to investigate other phase sets.

The graph in Figure 4.1 shows that the phase sets that were identified can be grouped into two large clusters (with a few exceptions) based on the general similarity measure. The xaxis represents the general similarity (in percentage) and the y-axis represents the number of phases in a phase_set$_{i,t}$.

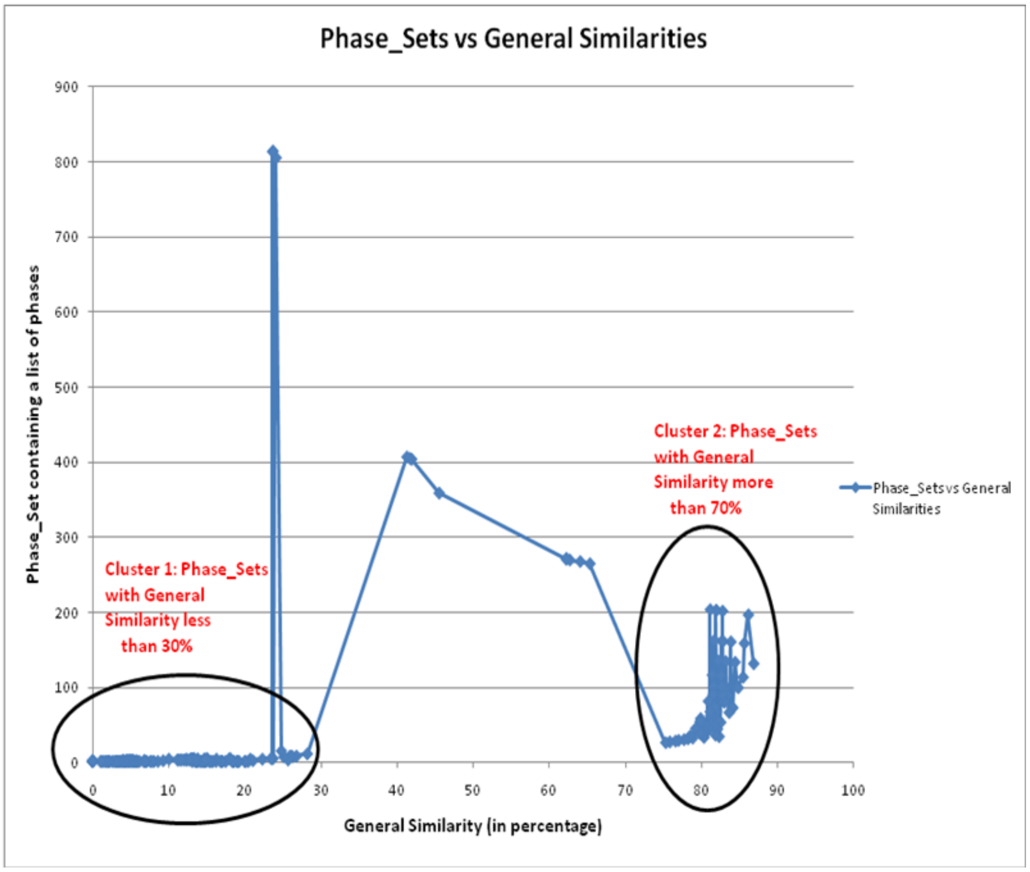

Figure 4.1 Phase_Sets and corresponding general similarities for the JHotDraw trace

The first cluster contains the phase_sets with a general similarity less that approximately 30% whereas the second cluster of phase_sets with general similarities more than approximately 70%. It should further be noticed in the graph that the cluster of phase_sets with more than 70% general similarities have the number of phases higher than one hundred which is less feasible for a trace generated simply by executing only a couple of features. Furthermore, we are interested in considering the cases with lower percentage of general similarities. Hence, the first cluster with the entries less than 30% general similarities is selected for further study. For more optimisation, we short list this cluster and select the phase_sets of general similarities less than 15%.

Once we have the phase_sets of lower values of general similarities, the objective now is to find the phase_set in which the consecutive similarity between its phases is the lowest. Therefore, our next step will be to find the consecutive similarities between all phases of each phase_set with less than 15% general similarity. Table 4.2 shows all the phase_sets and consecutive similarities between the phases contained in them.

Table 4.2 **Phase_sets and consecutive similarities for the JHotDraw trace**

Chunk Size (i)	**Threshold (t) (in percentage)**	**Phase_Set$_{i,t}$ (PS)**	**Summation of Consecutive Similarities (S)**	**Average of Consecutive Similarities (S/PS)**
130	30	5	68.660	13%
140	30	4	52.863	13%
170	40	4	53.204	13%
180	50	5	55.099	11%
190	50	5	58.761	11%
190	60	6	66.438	11%
200	50	4	53.966	13%
200	60	5	60.719	12%
210	50	4	56.624	14%
210	60	5	59.243	11%
220	60	5	58.692	11%
230	70	5	61.789	12%
250	70	5	73.585	14%
260	70,80	5	58.760	11%
270	80	5	59.882	11%
280	80	5	59.882	11%
290	80,90	5	59.882	11%
300	80	4	54.647	13%
300	90	5	60.719	12%
310	90	5	58.245	11%
320	**100**	**5**	**51.055**	**10%**
330	100	5	63.221	12%
340	100	4	58.358	14%
350	100	4	61.019	15%
360	100	4	55.168	13%

From the table, we can see that among all the phase sets, the phase_set$_{i,t}$ with lowest average consecutive similarity percentage is the one with chunk size i = 320 and threshold t = 100%. Therefore, it can be concluded that this phase_set$_{320,100}$ contains the phases that are highly distinct from each other. The five phases (four phase shift locations) contained in the phase_set$_{320,100}$ represent the chunk numbers where most of the previously occurring methods started to fade and new methods started to get invoked. The chunk numbers or the phase shift locations for the phase_set$_{320,100}$ with lowest consecutive similarity are 1, 3, 47 and 49. The following table shows the routine calls counter after which the phases are located.

Table 4.3 Phase shift locations and corresponding routine calls

Phase Shift Location/ Chunk Number (PSL)	**Chunk Size (CS)**	**Routine calls after which a phase is detected in the trace(RC = PSL * CS)**
1	320	320
3	320	960
47	320	15040
49	320	15680

We can conclude that the concept of consecutive similarity estimation highly supported our phase detection approach in identifying the chunk size and threshold that are best to detecting the phases in our trace. The following figure 4.2 represents the phase shift locations detected automatically by our algorithm in the JHotDraw trace obtained by exercising the scenario under consideration.

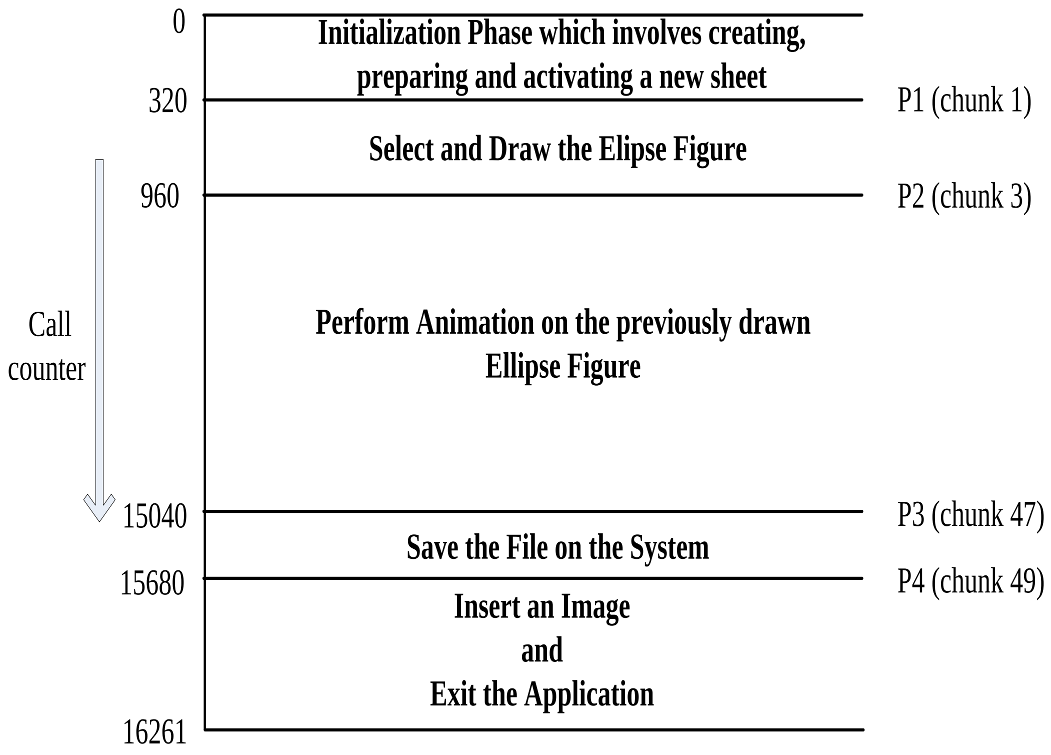
0
Initialization Phase which involves creating, preparing and activating a new sheet
320
P1 (chunk 1)
Select and Draw the Elipse Figure
960
P2 (chunk 3)
Call counter
Perform Animation on the previously drawn Ellipse Figure
15040
P3 (chunk 47)
Save the File on the System
15680
P4 (chunk 49)
Insert an Image and Exit the Application
16261

Figure 4.2 The execution phases in JHotDraw trace

The call counter represents the method calls that are responsible for implementing the consequent features. It should be noticed that the first phase, which is the initialization phase, is smaller as compared to the rest of the phases. This is because the trace has been pre-processed and the utilities, such as get and set methods and mouse movements, have been removed those of which constitute a major portion of the initialization phase. The solid lines in the figure represent the phase shift locations obtained by applying our algorithm on the corresponding execution trace. The phases after the initialization phase represent part of the trace which is responsible for drawing figures and performing other features. Each one of these phases contains a set of minor activities which include selecting the button of the figure, drawing the figure and unselecting the button. The last phase contains methods when the application terminates. In JHotDraw software system the finalization phase is extremely smaller and is difficult to locate, therefore, this finalization phase is merged with most commonly the last feature executed in the computational phase. In our case, the last feature of the scenario is inserting a figure, so the finalization phase which is approximately the last ten routine calls of the trace is merged with the inserting a figure feature.

To validate our results, we studied the content of the trace manually and compared the extracted phases with the ones that actually exist in the trace. We used JHotDraw documentation to understand the role of the invoked methods. We found that our phases match the manually detected phases, which shows the effectiveness of our algorithm. The methods that were responsible of the various features are listed in Table 4.4. Our phase detection algorithm was successful in putting these methods in each separate phase.

Table 4.4 Methods in JHotDraw source code and their responsibilities

Feature	Method Name	Responsibility
Ellipse	`CH.ifa.draw.figures.RectangleFigure.basicDisplayBox`	Sets the basic display box for figures to be drawn.
Animation on Ellipse	`CH.ifa.draw.samples.javadraw.JavaDrawApp.startAnimation`	Starts the animation of the figure.
Save As	`CH.ifa.draw.application.DrawApplication.promptSaveAs`	Shows a file dialog and saves drawing.
Insert Image	`CH.ifa.draw.insertImageCommand.execute, CH.ifa.draw.util.Iconkit.registerImage,CH.ifa.draw.util.Iconkit.loadImage`	Constructs an insert image command, registers the URL for the image source, loads an image file with the given name, caches it, and optionally waits for it to finish loading.
Exit Application	`CH.ifa.draw.application.DrawApplication.exit`	Exits the application.

4.4 Discussion

The overall results obtained by applying our phase detection algorithm on the real data reveal that our proposed approach is very effective in detecting execution phases in large traces. The target software system that we used to test our algorithm is JHotDraw 5.2

which is a well documented open source software system. We chose a scenario that generated a trace of tens of thousands of calls. We were able to divide the trace into meaningful phases, which reflect the high-level computations invoked in the traces.

Several phases have been identified by varying the chunk size and the threshold T. One of the challenges is to explore this large set. We had to focus on the phase sets cluster with low generality similarity since these are the sets that contain phases that are most distinct from each other. However, even with this, we were left with still a large set of phases. The consecutive similarity helped reduce this set to a more manageable set of phases from which we were able to identify the proper setting of the chunk size and threshold that best reveal adequate phases.

In practice and for more complex traces, we expect that the exploration of all possible phases might turn to be a challenging task. We recognize that more work needs to be done in this direction as we will describe in the last chapter of this thesis.

Chapter 5 Conclusion

5.1 Research Contributions

In this dissertation, we presented a novel approach for trace simplification which consists of dividing an execution trace into execution phases that represent the key computations contained in a trace. Our algorithm is based on the idea that a phase consists of methods that start to fade in the trace when a new phase starts to emerge. Using this algorithm, we believe that the software engineers can get a deep insight of what is happening inside the program without wasting time in understanding the content of overwhelmingly large and complex traces.

In particular, the algorithm contains two main steps: the phase detection and phase location steps. The objective of detecting a phase change is to identify when and where the methods which are responsible for implementing one phase of a program begin to fade, simultaneously causing the emergence of new methods which are responsible for implementing another phase. In order to detect a phase change, the collection of distinct methods of the program is captured into working sets, which were updated while the program executes. The objective was to detect when the most frequent methods become less frequent, which means that new ones are taking place.

For locating the execution phases once a phase is detected, we proceed by identifying the chunk from which many methods start to fade. We collect all the distinct methods from the chunks that comprise each phase. After performing a certain number of computations on these methods, the exact location of phase shift is determined.

Our algorithm relies on the chunk size and threshold used to compare working sets, another contribution of this thesis is a way to determine the best chunk size and threshold. In particular, we presented a concept of computing the similarity between execution phases which measure the degree by which the identified phases are from each other. The general similarity is a cumulative similarity of the individual similarities between all the execution phases for different sets of chunk sizes and thresholds. A high percentage of general similarity indicates that the phases are highly similar to each other and vice-versa. Another concept we introduced was the concept of finding the similarity between consecutive phases which is much more relevant in terms of phase distinction.

Finally, we applied our techniques to a trace generated from an object-oriented system. We validate the results using the system documentation. Our approach was capable of successfully detecting the phases in the generated trace

5.2 Opportunities for Further Research

Several future research directions are needed. First, we need to continue experimenting on different software systems to further assess the effectiveness of our approach. In addition, we need to improve the performance of the algorithm since it requires computing phases for each chunk size and threshold. What we need is to find ways to suggest adequate parameters without having to explore the range of all possible value. A heuristic-based approach is needed.

We also need to apply our techniques to other types of traces such statement-level traces, which are considerably larger than routine call traces. We also need to compare our result with existing trace abstraction techniques, and perhaps combine these different trace abstraction methods together. Finally, we need to work with software engineers to assess the value of our approach in practice. We anticipate that this can be done if the proposed algorithm is supported by a trace analysis tool.

5.3 Closing Remarks

Understanding the behaviour of a software system is a crucial task for many software engineering activities. To understand the behaviour of the system, however, one needs to process large traces; this is often a very tedious task. Several trace abstraction techniques have been proposed but the general consensus is that more research in the area is much needed to solve the trace analysis problem. We hope the work presented in this thesis can contribute to alleviate this problem.

Bibliography

Arnold 89	R. Arnold, “Software Restructuring”, *Proceedings IEEE*, pp. 607-617, April 1989.
Ball 99	T. Ball, “The concept of dynamic analysis”, *In Proc. 7th European Software Engineering Conference and ACM SIGSOFT Symp. on the Foundations of Software Engineering(ESEC/FSE),* pp. 216-234, Springer, 1999.
Biggerstaff 93	Ted J. Biggerstaff, Bharat G. Mitbander, Dallas Webster, “The concept assignment problem in program understanding”, *In Proceedings of the 15th International Conference on Software Engineering,* pp. 482-498, IEEE C.S., 1993.
Chikofsky 90	E. Chikofsky and J. Cross 1990, “Reverse Engineering and Design Recovery: A Taxonomy”, *IEEE Software, 7(1),* pp. 13–17, 1990.
Constantopoulos 95	Panos Constantopoulos, Mattias Jarke, John Mylopoulos, Yannis Vassiliou, “The software information base : a server for reuse”, *The VLDB Journal- The International Journal on Very Large Data Bases, Vol. 4,* pp. 1-43, 1995.

Cornelissen 08	Cornelissen, B. And Zaidman, A. And Holten, D. And Moonen, L. And van Deursen, A. and van Wijk, J.J. “Execution trace analysis through massive sequence and circular bundle views”, *the journal of Systems & Software, vol 81, 12,* pp 2252-2268, 2008, Elsevier.
Eclipse TPTP	http://www.eclipse.org/tptp/
Fjeldstad 83	R.K Fjeldstad and W.T. Hamlen, “Application Program Maintenance Study: Report to Our Respondents”, *In Proceedings GUIDE 48*, 1983.
Hamou-Lhadj 02	Abdelwahab Hamou-Lhadj, and Timothy C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces”, *In Proceedings of the 10th International Workshop on Program Comprehension,* pp. 159-168, 10th December 2002.
Hamou-Lhadj 03	Abdelwahab Hamou-Lhadj, and Timothy C. Lethbridge, "Techniques for Reducing the Complexity of Object-Oriented Execution Traces”, *In Proc. of the VISSOFT,* pp. 35-40, 2003.
Hamou-Lhadj 04	A. Hamou-Lhadj, and T. C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques”, *In Proc. of CASCON 2004,*

IBM Press, ACM Digital Library , Toronto, Canada, pp. 42-54, 2004.

Hamou-Lhadj 05 A. Hamou-Lhadj, and T. Lethbridge, and L. Fu, “SEAT: A Usable Trace Analysis Tool”, International Workshop on Program Comprehension (IWPC), IEEE CS, St. Louis, Missouri, USA, pp. 157-160, 2005.

Hamou-Lhadj 06 Abdelwahab Hamou-Lhadj, and Timothy C. Lethbridge, "Summarizing the content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *In Proc. of the 14th IEEE International Conference Program Comprehension*, pp. 181-190, 2006.

Jerding 97 D. Jerding, J. Stasko, T. Ball, “Visualizing Interactions in Program Executions”, *In Proceedings of the 19th ICSE,* Boston, Massachusetts, 1997, pp. 360-370.

Jerding 98 D.F Jerding, J.T Stasko, “The Information Mural: A Technique for Displaying and Navigating Large Information Spaces”, *In Proceedings of the IEEE Transactions on Visualization and Computer Graphics,* vol. 4, pp. 257-271, Jul-Sep 1998.

JHotDraw 5.2 http://www.jhotdraw.org/

Malony 91	A.D. Malony, D.H. Hammerslag, D.J. Jablonowski, “Traceview: A Trace Visualization Tool”, *In the Proceedings of IEEE Software”,* pp. 19-28, Illinois University, 1991.
Mayrhauser 95	Anneliese von Mayrhauser and A. Marie Vans 1995, “Program Comprehension during Software Maintenance and Evolution”, *Colorado State University, IEEE Computer,* 28 (8), pp. 44-55, August 1995.
Moonen 08	Moonen, Bas Cornelissen and Leon Moonen, “On Large Execution Traces and Trace Abstraction Techniques”, *Delft: Software Engineering Research Group,* 2008, ISSN 1872-5392.
Müller 88	Müller H. A., Klashinsky K., “Rigi – A System for Programming in-the-large”, In *Proceedings of the 10th International Conference on Software Engineering,* ACM Press, pages80-86, 1988.
Nelson 96	Michael L. Nelson, “A Survey of Reverse Engineering and Program Comprehension”, *April 1996, ODU CS-551 Software Engineering Survey.*
Pauw 98	W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, “Execution Patterns in Object-Oriented Visualization”, *In Proceedings of the*

	4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, NM, 1998, pp. 219-234.
Pirzadeh 10	H. Pirzadeh, A. Agarwal and A. Hamou-Lhadj, “An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension”, *In Proc. of the 8th International Conference on Software Engineering Research, Management & Applications (SERA 2010), Montreal, Canada, 2010.*
Renieris 99	Manos Renieris, Steven P. Reiss, “Almost: Exploring Program Traces”, *In Proceedings of the 1999 Workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM international conference on Information and knowledge management,* pp. 70-77, United States, 1999.
Rugaber 95	Spencer Rugaber, “Program Comprehension”, College of Computing, Georgia Institute of Technology, May 4, 1995.
Reiss 05	S.P Reiss, “Dynamic detection and visualization of software phases”, *in Proc. 3rd ICSE Int. Workshop on Dynamic Analysis (WODA), pages 1-6, 2005. ACM SIGSOFT Sw. Eng. Notes 30(4).*

Systa 00	T. Systa, “Understanding the Behaviour of Java Programs”, *In Proceedings of the 7th Working Conference on Reverse Engineering,* pp. 214-223, November 2000.
Systa 99	T. Systa, “On the relationships between static and dynamic models in reverse engineering Java software”, *In Proceedings of the 6th Working Conference on Reverse Engineering,* pp. 304-313, 1999.
Tverskey 77	Tverskey, Amos, “Features of Similarity”, *Psychological Review,* 84 (July), pp. 327-352.
Watanabe 08	Yui Watanabe, Takashi Ishio and Katsuro Inoue, “Feature-Level Phase Detection for Execution Trace Using Object Cache”, *In Proceedings of the 2008 International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008),* pp. 8-14, 2008.