Stratified Sampling of Execution Traces: Execution Phases Serving as Strata

Heidar Pirzadeh[1], Sara Shanian[2], Abdelwahab Hamou-Lhadj[1], Luay Alawneh[1], Arya Shafiee[1]
[1]Software Behaviour Analysis Research Lab
Department of Electrical and Computer Engineering; Concordia University, Montreal, QC, Canada
{s_pirzad , abdelw, l_alawne, ar_s}@ece.concordia.ca
[2]Department of Computer Science. Laval University; Quebec City, QC, Canada
Sara.Shanian@ift.ulaval.ca

Abstract

The understanding of the behavioural aspects of a software system is an important enabler for many reverse engineering activities. The behaviour of software is typically represented in the form of execution traces. Traces, however, can be overwhelmingly large. To reduce their size, sampling techniques, especially the ones based on random sampling, have been extensively used. Random sampling, however, may result in samples that are not representative of the original trace. In this paper, we propose a trace sampling technique that not only reduces the size of a trace but also results in a sample that is representative of the original trace by ensuring that the desired characteristics of an execution are distributed similarly in both the sampled and the original trace. Hence, the insights gained from analyzing the sample trace could be extrapolated to the original execution trace. Our approach is based on stratified sampling instead of random sampling and uses the concept of execution phases as strata. We define an execution phase as a part of a trace that represents a specific task of the traced system. We also present an approach for the automatic detection of execution phases from a trace. Finally, we show the effectiveness of our sampling technique through two case studies.

Keywords: Trace analysis, program comprehension, sampling techniques, stratified sampling, execution phases

1. Introduction

Analyzing the content of execution traces can be a challenging task due to the large size of typical traces but, if done properly, the benefits are numerous. Traces can reveal important information about the way a system behaves and help answering questions on why it behaves in a certain way. This is useful for many maintenance tasks such as understanding how a particular feature is implemented or uncovering places where a bug occurs [CHMY03, LAZJ03, RR03, RZ05, Dug07]. Trace analysis techniques can also be used to correlate traces generated from subsequent versions of a system to understand important variations that can in turn help maintainers estimate the effort required to maintain evolving systems. In fact, the understanding of the behavioural aspects of software systems goes beyond maintenance to include recent research areas, like security and autonomic computing [WGZ04, GMSS00], where traces are used to characterize the normal behaviour of a system and detect any deviations from normalcy due to attacks, design faults, or changes in the environment.

Investing in trace analysis techniques (or what we prefer to call software behaviour analysis) is therefore an important research thread that is expected to have a significant impact. There exist today several studies that focus on ways to make traces smaller while (ideally) keeping as much of their essence as possible (e.g., [HBDL05, HL06, Rei05]). Several trace simplification and abstraction techniques have emerged over the years to help software engineers explore the content of large traces faster. These techniques vary in their design and range from the use of visualization techniques (e.g., [Rei07, Rei05]) to advanced filtering techniques based on the removal of utilities [HL06]. These techniques should not be confused with data compression methods as defined in Information Theory. The purpose of compression algorithms is to make data as small as possible; a decompression process is required to reconstitute the data to use it for any purpose. In contrast, the purpose of trace abstraction is to make the data somewhat smaller by eliminating unneeded data, keeping the result intelligible and useful without the need for 'decompressing'. Despite the significant progress that trace analysis has seen in recent years, the general consensus is that more research in this area is needed.

In this paper, we present an approach for reducing the size of traces that is based on the sampling of the trace content. The main drawback with existing trace sampling approaches is that there is no guarantee that the resulting sample is representative of the original trace. That is, using common sampling methods, we might not be able to make correct inference about a trace based on a sample from that trace. To overcome this limitation, we propose the stratified sampling of execution traces. We first divide the trace into trace segments that we refer to as execution phases. We define an execution phase as a part of a trace that performs a specific computation [PH11a]. In other words, a phase denotes a group of similar events[1] that together perform an essential step of the general execution. A trace can then be seen as a sequence of exhaustive and non-overlapping execution phases rather than a mere flow of events. By using execution phases as strata, we ensure that a certain number of events will be selected from each execution phase to yield a sample that is representative of the original trace. Our approach provides users with the freedom to choose their desired sample size and set a threshold for detecting phases.

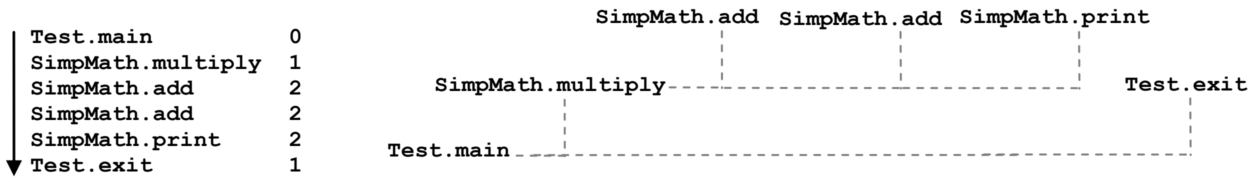
The remaining part of this paper is organized as follows. In Section 2, we review some background concepts and related work. In Sections 3, 4, 5, and 6 we present our approach and its components. In Section 7, we discuss the tool support. In Section 8, we validate the effectiveness of our approach in two case studies, followed by a list of threats to validity in Section 9. We conclude the paper in Section 10.

[1] In this paper, we focus on traces of method calls.

2. Background and Related Work

2.1. Background Concepts

Execution Trace: An execution trace is a sequence of events (e.g., method calls, invoked objects, system calls, etc) resulted from exercising one or more features[1] of a software system. The content of an execution trace (i.e., the events) is a representation of the functionalities triggered by the user. Each event can have a number of attributes (e.g., entry time, code line number, etc). Our focus, in this paper, is on traces where the events are method calls. Such traces are commonly used in the field of software maintenance [JSB97, Sys00]. A trace of method calls can be represented as a tree structure. An example of interactions among two objects of the classes Test and SimpMath, which implements multiplication of numbers using repeated addition, is shown as a trace of method calls in Figure 1 (a). The integer value at the right-hand-side of each method call indicates the value of the nesting level attribute of that method call. Figure 1 (b) shows the corresponding tree representation of the trace in Figure 1 (a).



```
Test.main          0
SimpMath.multiply  1
SimpMath.add       2
SimpMath.add       2
SimpMath.print     2
Test.exit          1
```

(a)

(b)

Figure 1. (a) A trace of method calls and (b) its corresponding tree representation.

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

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

[1] A feature is an observable functionality triggerable by a user [EKS03].

2.2. Related Work

Understanding a large software system can be a complex and challenging task for systems that have undergone several years of ad-hoc maintenance activities and/or for which initial documentation (if it exists) has become outdated and obsolete. It is also hard to rely on the knowledge of the initial designers of the system because they often move to new projects taking with them this knowledge. Reverse engineering research, using both static and dynamic analyses, has been very active in recent years to overcome these challenges. The objective of reverse engineering techniques is to recover high-level views of a system from its low-level artefacts such as the code. These views can later be used by maintainers to speed up their comprehension process.

An important activity in reverse engineering is the ability to detect the code components (e.g., classes, methods, lines of code, etc.) that implement a particular feature, a problem known as feature location. It is has been shown that software maintainers do not need to understand the entire system to perform a maintenance task. Instead, they can proceed with an as-needed approach by focusing on only the features that must be maintained.

Static feature location techniques [CR00, KDMS07] often use system dependency graphs to help the user identify a feature. A graph that models global variables, method invocations, and data flow in the source code is searched and navigated by the maintainer to find events related to a specific feature. Although some guidance can be provided to the maintainer based on the analysis of the structural dependencies of the system, the technique works well only if the maintainer has enough knowledge of the system to know where to start exploring the dependency graph. As a result, such an approach may be inefficient when trying to locate features in large and complex systems. Another group of static feature location techniques uses lexical analysis to find code components related to a feature (e.g., [BMW94]). First, the code is parsed and then using clustering and text pattern matching techniques, a set of events (routines and identifiers) related to a feature is returned based on naming similarities. The lexical approach has been improved using information retrieval techniques that discovers associations between identifiers in the code and terms used in a requirement specification [ACC+02, MM03].

When combined, these two static approaches yield better results. The lexical approach can be used to provide ranked results to the programmer's queries. The programmer can have a better idea of where to start navigation in the dependency graph. Static analysis can be used to derive mappings between events that are internal to the system. Considering that the features are defined based on external behaviours and functionalities of the system, static analysis, may not be successful and precise enough in deriving mapping between features and source code events.

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

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

A hybrid approach that combines dynamic and static analysis techniques to feature location has also been proposed [EKS03]. This approach uses dynamic analysis to gather traces that correspond to software features of the system and adds static code dependency information to the content of traces to build a concept lattice that maps features to code components. One of the shortcomings of this approach is that overlapping components (i.e., the ones that implement several features) can appear in the concept lattice. To overcome this issue, users are required to navigate through the concept lattice and identify manually the components specific to each feature. This process requires a considerable effort from the users and a good understanding of the source code as well as the domain of the system.

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

Dynamic analysis techniques often rely on the tracing and runtime monitoring mechanisms. Traces, however, are difficult to work with because they tend to be considerably large. To address this issue, many trace abstraction and simplification techniques have been proposed with the common objective of extracting high-level views from raw traces. Trace summarization [HL06] is a type of trace abstraction technique in which an execution trace is taken as input and the summary of its main content is returned as output. UML sequence diagrams are obtained from this summary that gives the high-level representation of software system. The summary of main content of a trace is obtained by removing low-level implementation details, such as utilities from the trace. A utility is defined as "Any event of a system designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the system" [HL06]. The extent to which an event can be considered a utility is measured by utilityhood metric based on static fan-in and fan-out as proposed in [HL06].

Visualization approaches, such as the ones presented in [Rei05, Rei07, CHZ+07], usually offer an overview of the execution trace through a mural view that helps the user to detect segments of an execution trace that are visually distinguishable from one another and referred to as execution phases. In their tool called ExtraVis [EXT, CHZ+07], Cornelissen et al. proposed an approach for detecting execution phases in a trace. ExtraVis offers a mural view where the call relations are visualized based on the system's static structure. This view helps the user to visually detect the different phases of the system's execution. Being aware of the execution scenario, the user can also hypothetically relate the repetitive (or ordered) patterns of events to the repetitive (or ordered) features in the execution scenario. Then, zooming on a pattern, the user can verify the hypothesis. In ExtraVis, the user is required to analyze large amounts of data visually.

Jive by Reiss et al. [Rei05, Rei07] is a tool that uses murals to visualize execution phases of a system while it is running. This visualization technique can help the programmer to understand the system on the fly. As a result, unlike our approach, the phase detection process in Jive is done in an online fashion. Although visualization techniques can provide a good trace abstraction mechanism, we believe that these techniques alone are limited to the presentation of large amount of data [HL04]. What is needed is to have built-in algorithms that can quickly analyze and reduce the size of traces while keeping as much of their essence as possible. The resulting views can then be rendered using existing visualization techniques.

Watanabe et al. [WII08] proposed an online phase detection technique that is based on the investigation of memory cache for observing objects that are working for the current phase; a significant change in the cache shows the emergence of a new phase. Several parameters (cache size, window size, threshold, and phase search distance) control their phase detection algorithm. However, no solution is suggested on how to tune these parameters and changing one or more may result in different phases.

Kuhn and Greevy [KG06] proposed a trace analysis techniques inspired by signal processing. The authors, first, transform execution traces into time series by plotting the nesting level of the methods against points in time through the execution. Then, the volume of data can be reduced to up to 90% by the application of several filters, such as a minimal nesting level threshold, making it possible to visualize a large number of events in multiple traces on a single screen. Users can visually identify similar phases within a trace and between the traces. This technique cannot guarantee for similarities between methods in a phase because it does not take into consideration the method names when preparing the plot to match patterns between trace signals. Also, it removes a lot of information that it considers inessential data by applying multiple filtering logics (independent of method names), having as target mainly the representation size. This filtering potentially results in loss of important trace information during the abstraction process.

Pirzadeh et al. [PAH10] proposed a phase detection technique based on the fact that a phase shift within a trace appears when a certain set of methods, responsible for implementing a particular task which are predominant in one phase, start disappearing as the system enters another phase. The authors proposed an algorithm that operates on the trace while it is being generated. The online algorithm keeps track of the methods encountered and raises a flag when a significant number of these methods start disappearing and new ones start emerging. This approach is significantly different from the one proposed in this paper because it is not based on measured similarity and continuity among trace events. Furthermore, the approach presented in this paper involves less parameter tuning.

Zaidman proposed in his Ph.D. thesis a technique for extracting the most important classes that are most relevant to the implementation of the traced scenario using Web page ranking techniques [Zai06]. The objective of his approach is to address the problem of locating features in code. These techniques often require a lot of processing and the use of different sources of information (including the source code). The resulting components do not necessarily form a sample of a trace. That is, there might be some analyses of the trace content that may require more than the components identified using a pure feature location technique. We recognize that further research should be carried out to see the relationship between trace sampling and feature location techniques. Our quick intuition is that the need for one or the other should be guided by the type of analyses ones wishes to perform. Also, it should be noted that sampling should be done quickly and at minimum cost. Feature location techniques rarely meet these constraints.

Sampling techniques have also been used to reduce the size of traces (e.g., [CHMY03, LAZJ03, RR03, RZ05, Dug07]). Sampling consists of selecting parts of a trace for instead of analyzing the entire trace. Existing approaches, however, are suffering from a major drawback: there is no guarantee that the resulting sample is representative of the original trace. This lack of representativeness appears to be due to the fact that existing sampling techniques are blind to the information contained in the trace; they treat a trace as a data stream for which the pieces are considered equal. Furthermore, many sampling approaches need manual tuning of several parameters. Finding the right sampling parameters can be a difficult task and even if some parameters work well for one trace, they might not work for another trace (even if generated from the same system) [CHMY03].

In conclusion, there exist many techniques to help with the understanding the behavioural aspects of software systems. These techniques focus on the analysis of execution traces. The common objective is to find effective ways to reduce the size of traces so as to allow software engineers to quickly understand and analyze their content. As one can expect, these techniques vary significantly in their design and cover a wide range of approaches including visualization and sampling. In this paper, we propose a new approach based on stratified sampling that aims to reduce the size of traces. Our sampling approach improves over existing techniques by extracting sampled traces that are representative of the original trace.

3. Reasoning about Sampling

A simple and a naive way to sample the content of a trace is to consider every *n*-th generated event. The size of the trace can be controlled automatically based on the sampling parameter *n*.

The sampling parameter (also called distance) for a given trace *T* in systematic sampling is usually represented as follows:

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

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

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

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

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

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

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

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

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

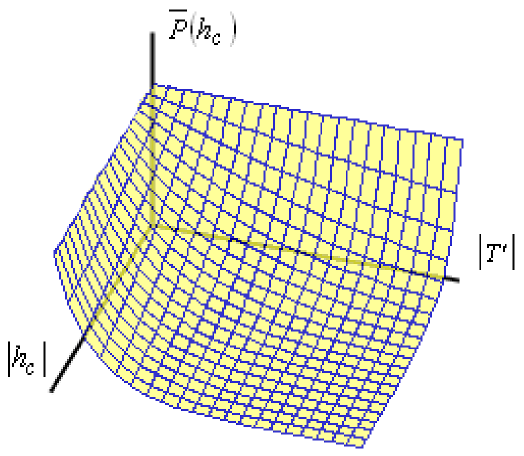


Figure 2. Behavior of $\overline{P}(h_c)$ with respect to $|T'|$ and $|h_c|$

Figure 2 shows the behaviour of $\overline{P}(h_c)$ according to the changes of $|T'|$ and $|h_c|$. As shown in Figure 2, in random sampling, the smaller the size of the sampled trace, the higher is the probability of having an unrepresentative sample. Furthermore, small sizes of homogeneous subsequences can also result in unrepresentative samples. Therefore, we need a different sampling approach.

4. Stratified Sampling of Execution Traces

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

In the stratified sampling of a population, first, the population is separated into a desired number of partitions[1] (called strata) and then sample elements are drawn from within each stratum. The size of the sample from each stratum is kept proportional to the size of the stratum (this is called proportional stratified sampling). We thus guarantee that the final sample contains elements representative of every part of the population. The process of stratified sampling is shown as a flow chart in Figure 3.

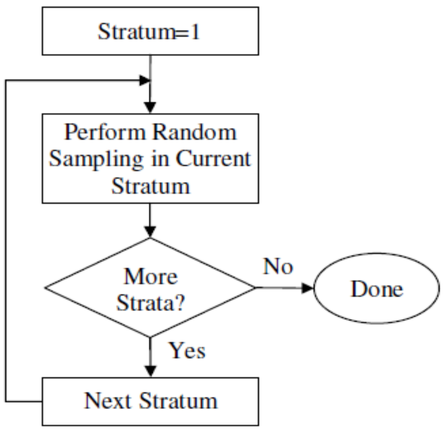


Figure 3 Stratified sampling process

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

Drawing a parallel between population sampling and trace sampling, we are interested in a trace sampling method that can create a more representative sample in comparison with existing trace sampling methods. We propose stratified sampling of execution trace. For strata specification we propose using execution phases as strata. Our definition of an execution phase is similar to the

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

one presented in [Rei07] in which a phase is defined as a segment of a system's execution that exhibits common behaviour at a level the programmer would recognize. In our previous work [PH11a], we proposed an approach to automatically detect execution phases in a trace. The essence of this approach is revisited in Section 5.

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

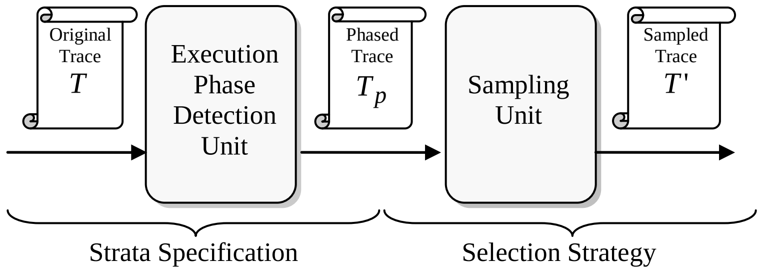


Figure 4. Overview of the proposed sampling framework

Our framework for stratified sampling of execution traces is shown in Figure 4. By splitting the process into different units, our proposed framework achieves a flexible and extensible architecture where each unit or its composing components can be supplemented or replaced by alternative approaches. As shown in Figure 4, we first need to identify the execution phases to serve as strata for our stratified sampling approach. We use the execution phase detection unit to detect the execution phases of the trace. The result of this unit is then given to the sampling unit that outputs a sampled trace. Both units are described in more details in the following sections.

5. Detection of Execution Phases

In our previous work [PH11a, PH11b], we have developed a phase detection technique that is inspired by two of the Gestalt laws of perception (namely similarity and continuation), which describe how people group similar items visually based on their perception [Kof35, SF99, QB09]. Gestalt psychology [QB09] is an application of physics to essential parts of brain physiology describing the processes occurring in the brain when we see visual objects and how our perceptual systems follow certain grouping principles (e.g., good continuation, proximity, and similarity properties of the elements) to integrate the scene elements (i.e., objects and regions) as a whole and not just as points and lines. These laws explain how our perceptual system segments local elements against their context and integrates them as objects. The tendency of perceiving elements having similar characteristics as belonging to an approximately homogeneous group suggests that the same mechanism can be used for extracting homogeneous segments (i.e., execution phases) in a trace that can serve as strata.

The phase detection unit (Figure 5) is implemented as a lightweight algorithm that analyzes the trace in one pass, automatically dividing its content into segments that correspond to the system's main execution phases, such as initializing variables, performing a specific computation, etc.

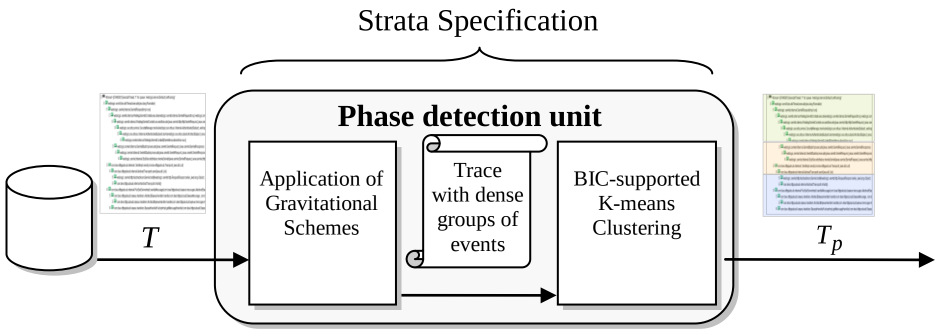

Strata Specification
Phase detection unit
Application of Gravitational Schemes
Trace with dense groups of events
BIC-supported K-means Clustering
T
T_p

Figure 5. Detailed view of the execution phase detection unit

Figure 5 shows the phase detection unit. In the application of gravitational schemes on the input trace, two schemes, the similarity and continuation schemes, define gravitational forces between the trace events. The application of these schemes yields the formation of dense groups of trace events, which indicate the candidate phases. We use K-means clustering with BIC (Bayesian Information Criterion) support [Sch78] (discussed in more details later in Section 5.4) to automatically identify phases. The effect of applying each of the similarity and continuation schemes is as follows:

- ***Similarity scheme***: By applying this scheme the events in the trace are rearranged in a way that the distance between similar trace events is reduced. We consider two method calls similar if they call the same methods.
- ***Continuation scheme***: The application of this scheme results in the repositioning of the trace events in a way that the consecutive events are made closer to one another if there a continuous change (no sudden jump of drop) in the values of a certain attribute of the events. For example, in traces of method calls, the consecutive method calls that are in the calling nesting level are made closer to one another to emphasize a trend in the execution of the system.

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

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

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

To apply our gravitational schemes, we define the distance between the method calls in the trace. The difference in invocation orders between the method calls in the trace is considered as distance between the method calls and it is assumed that there is equal distance of 1 between consecutive invocations in the original trace. For instance, the distance between $c_{i,d}$ and $c_{j,d'}$ would be equal to $|j-i|$ (i.e., the distance between two points on a ruler). This way, we map the ordinal scale of method calls to an interval scale. Furthermore, for each of the schemes, we define a function $Pos(c_{i,d})$ to return the position of the method call *c* in the interval scale. The position of a method call is also the order in which it was invoked right after the trace is generated. However, as the method calls are rearranged as a result of applying the two schemes, the new position of a method call might differ from its original order of invocation. We use this rearranged trace to find the phases from the original trace.

5.1. The Similarity Scheme

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

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

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

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

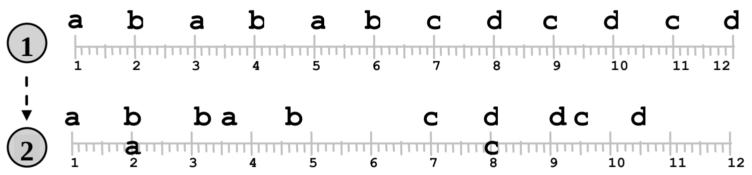


Figure 6. An example of applying the similarity scheme

In Figure 6, we show the effect of applying the similarity scheme to a simple trace (this trace does not reflect real world traces and is used here for illustration purposes only). The resulting trace appears to contain two dense groups that may indicate the presence of two distinct phases. The first one starts at the first invocation and is composed of calls to methods `a` and `b`, while the other one which starts at the seventh invocation contains calls to `c` and `d`.

5.2. The Continuation Scheme

The similarity scheme works well for a trace that contains similar events. But what happens if there is no noticeable similarity between the events of an execution trace? For example, Figure 7 (step 1) shows an execution trace where it is hard to distinguish the similar events -no distinct method is invoked more than once. However, one can perceive two different segments in this trace. This perception can be explained by another Gestalt law, the Law of Good Continuation [GPSG01]. The Law of Good Continuation refers to the tendency of things to group if they are visually co-linear or nearly co-linear. In this paper, we use the continuation scheme to group trace events using the nesting level of the method calls.

For example, in Figure 7 (step 1), one can notice that there is a good continuation between the calls from `a` to `o`, which can intuitively suggests the existence of a phase. Using the nesting level of calls to detect execution phases has also been the topic of other studies [KG06, WII08]. Watanabe et al. [WII08] used the nesting levels of a call tree to detect phases and locate phase shifts. The authors suggested that the depth of the call stack (i.e., the nesting level) is a local-minimum at the beginning of a phase indicating a phase transition. They also showed that the events that have a high nesting level (i.e., which are deep in the tree hierarchy) were unlikely to initiate new phases.

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

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

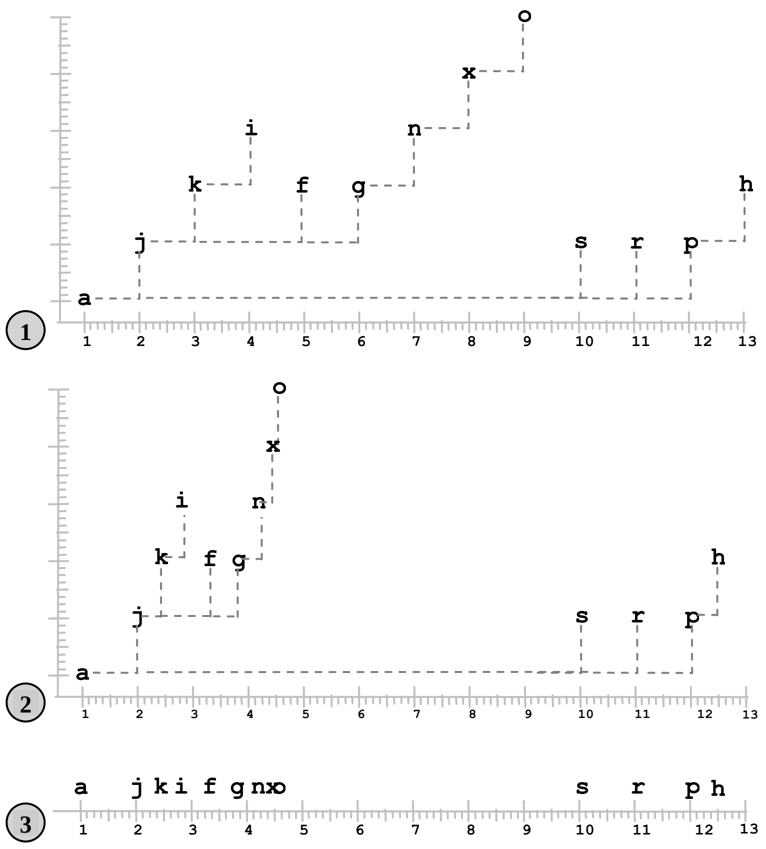


Figure 7. An example of applying the continuation scheme

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

Figure 7 (step 2) shows the result of applying the continuation scheme to a sample trace. As we can see, the new positioning of the trace events leads to two distinguishable groups of event. These groups, that indicate the presence of two phases, are more distinguishable when we omit visualization of the nesting levels in step 3. The first phase begins at the first method invocation and the second phase starts at the tenth method invocation.

5.3. Integration of the Schemes

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

The integrated repositioning scheme can be iteratively applied to a trace to detect major phases, their sub-phases, etc,, until we reach the individual events of the trace. To harness gravity so that phases could be detected with different levels of granularity, a threshold *t* is defined which can be used to prevent two far methods from attracting each other and hence forming a large block. More precisely, the threshold *t* works in such a way that a call to a method *m* is attracted to a previous call to the same method only and if only the distance between these two calls is less than the threshold. Major phases can be detected by setting a threshold *t* that is close to the size of the trace. The smaller the threshold, the more fine-grained phases we can detect. We anticipate that the threshold is application-specific. Our tool that supports this approach allows enough flexibility to vary the threshold (see Section 7). An integrated scheme with threshold *t* changes the position of method calls as follows:

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

where

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

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

and

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

5.4. Identifying the Beginning and End of Phases

Once the method calls of a trace are repositioned according to the integrated scheme and dense groups of method calls are formed in the rearranged trace, we need a way to automatically identify the beginning and end of each phase on the original trace because it would be impractical to expect from programmers to distinguish the various phases visually for considerably large traces. We use a clustering algorithm to automatically find the beginning and the ending method call of each dense group based on the positions of method calls. The beginning method call of each group is then marked as the beginning of a corresponding phase on the original trace. Therefore, the beginning of each phase in the original trace is marked by the beginning of a corresponding group in the rearranged trace and the ending of that phase is marked by the beginning of the next phase.

In [TH98], a number of hierarchical and partitional clustering algorithms and their applications to software engineering are presented. In this paper, we chose a partitional clustering algorithm, the K-means clustering [Mac67], as our clustering algorithm. The study of the impact of various clustering algorithms on our approach is left as future work. In K-means clustering, the number of clusters *K* (i.e., the number of phases) should be given as an input to the algorithm (perhaps by counting the number of dense groups that can be visually perceived on the rearranged trace). However, it would be advantageous if the number of dense groups could be automatically selected according to the complexity of the data. Pelleg et al. [PM00] proposed an approach to find the best partitioning of data where the average variance of the clusters is minimum. It is clear, that as the number of clusters increases, the average variance of the clusters decreases (as *K* approaches the number of data points the variance becomes zero; this is known as overfitting). Therefore, the problem is reduced to finding a tradeoff between the number of clusters and the average variance of the clusters that can keep the number of clusters and the variance both minimized. We find this trade-off via the Bayesian Information Criterion (BIC) [Sch78].

As shown in Figure 8, we assume that the user has run the K-means algorithm on the repositioned trace (i.e., a trace with dense groups of methods formed using the similarity and continuation schemes) for a set of different values of *K*, which results in a set of alternative partitionings. To evaluate these partitionings, we compute the BIC score of each partitioning, the highest BIC score means the best available partitioning of the execution trace and consequently the best estimation of the number of clusters *K* (which also corresponds to the number of phases).

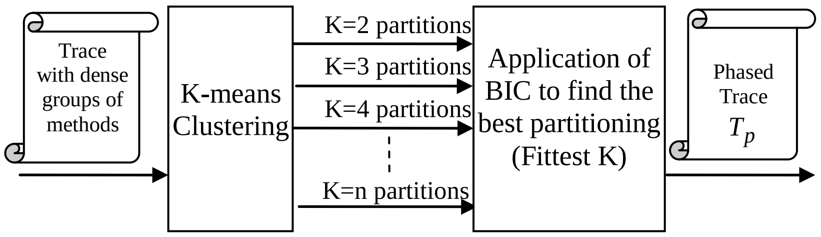

Trace with dense groups of methods
K-means Clustering
K=2 partitions
K=3 partitions
K=4 partitions
K=n partitions
Application of BIC to find the best partitioning (Fittest K)
Phased Trace T_p

Figure 8. Detailed view of BIC-supported K-means clustering

In [PH11a], we applied our phase detection algorithm to large traces generated from two object-oriented systems. We validated our results by studying documentation of both systems and showed that our approach was successful in dividing these traces into meaningful phases.

6. Sampling Unit

Once the phases are detected, we start the stratified sampling process, which is implemented in our framework, as part of the sampling unit (shown in Figure 9). The sampling unit receives a phased execution trace as its input and outputs a sample of the execution trace using stratified sampling.

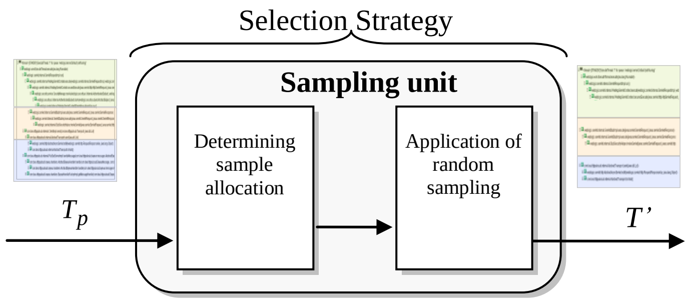


Figure 9. The Sampling unit in details

We use the sample trace shown in Figure 10 (part 1) to explain how the sampling is performed in a step by step fashion. Figure 10 (part 2) shows the trace divided into 4 major phases as a result of the application of our phase detection approach presented in Section 5. The trace in Figure 10 contains 25 events and we would like to obtain a sampled trace of size 6.

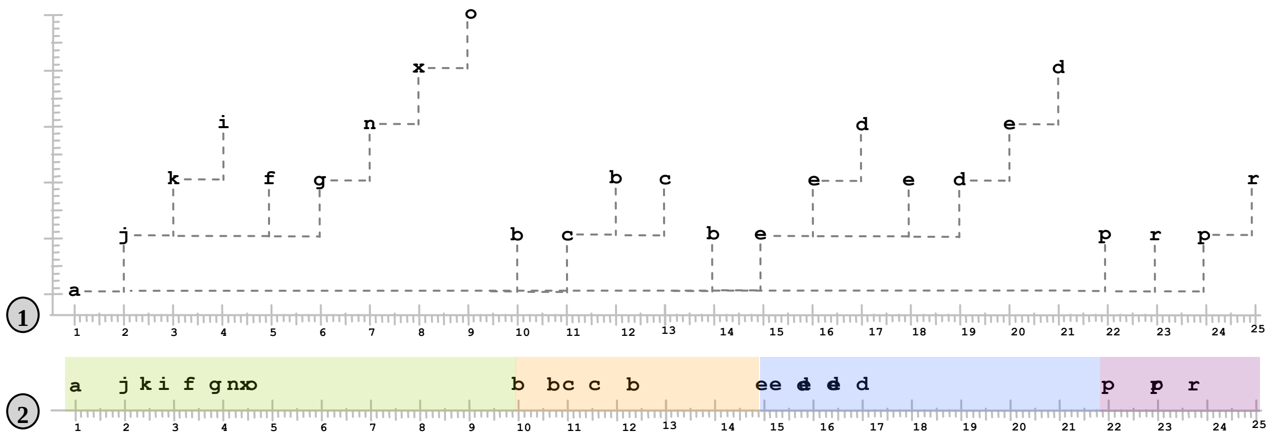


Figure 10. An example of applying the integrated scheme on a sample trace

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

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

where $Stratum_h$ is the number of events in each stratum and $|T_p|$(the size of the phased trace) is equal to the size of the original trace. Table 1 shows the number of events within each stratum of our sample trace in Figure 10.

Table 1. Execution phases that were detected

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

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

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

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

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

Therefore, we select $|S_h|$ events from each stratum. For our sampled trace of Figure 10, the number of samples to be drawn from each stratum is calculated the same way:

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

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

7. Tool Support

Different tools have been implemented and reused to support the proposed approach. These tools are implemented in Java as part of a tool-suite called Tratex[1] which is an Eclipse plug-in. The

[1] http://www.ece.concordia.ca/~s_pirzad/sampling-case-study/

phase detection unit of the Tratex implements the integrated scheme (Section 5.3) and the clustering component (Section 5.4).

The integrated scheme is implemented as a one pass algorithm. The algorithm creates a *position table* (a HashMap[1]). For each method, the position table keeps the position of the last call to that method (as used in similarity scheme). Visiting each method call in the original trace, the algorithm looks up the position table for that method, if not found, it adds the method and the position of the method call to the table, otherwise, it fetches the previous position of the method call. The algorithm also records the nesting level of the previous method call (as used in continuity scheme). The time complexity of the algorithm that implements the integrated scheme is $O(n)$, where n is the size of the trace.

K-means clustering is chosen as the clustering algorithm in the clustering unit because of its main advantages: simplicity and observed speed, which allow it to run on large datasets [Vas07]. The K-means and cluster evaluation implemented in Java-ML library [ADS09] are used as the basis of the clustering unit. Modifications are made to the code to support threading[2]. The time complexity of the standard k-means algorithm is $O(icnd)$, where i is the number of iterations, c is the number of clusters, n is the size of the dataset, and d is the dimensionality [DRS08]. In our case, the size of the dataset is equal to the size of the trace, number of iteration is fixed to 50 (as common case for K-means clustering), dimension is 1 (because each event has a position on a single axis), and the number of clusters is set from 1 to 14 (in parallel).

The Sampling unit (Section 6) of the Tratex implements random sampling that is used in stratified sampling. The implementation of random sampling uses the Random class in Java to generate a random number in constant time. Similar implementation is used to generate sampled traces through random sampling in our case studies.

8. Case Studies

We apply our proposed phase-based stratified sampling approach in two case studies. The goal of the first case study is to analyze the use of phase-based stratified sampling of execution traces, with the purpose of comparing its usefulness with random sampling in program comprehension. The second case study is a real-world running example of the problematic case that is theoretically discussed in Section 3 and shows how our phase-based stratified sampling compares to random sampling of execution traces. The case studies are on traces generated from two different systems: WEKA 3.0 [WEKA] and JHotDraw 5.2 [JHO]. To generate the traces we instrumented both systems using TPTP (the Eclipse Test and Performance Tools Platform) [TPTP].

[1] Provides constant-time performance for the lookup operation [Java].

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

8.1. First Case Study

The experimental unit in this study is WEKA [WEKA], a Java-based machine learning tool that implements several learning algorithms. WEKA (v. 3.0) has 10 packages, 147 classes, 1642 public methods, and 95 KLOC. We selected to analyze the C4.5 classification algorithm that builds a decision tree for classifying data instances. For this we use the C4.5 trace (the original trace) generated by Hamou-Lhadj et al. [HL06].

The main factor in this study is the sampling method that is applied on the original trace to extract a sample to be used for program comprehension. The dependent variable in this study is the comprehension level. The comprehension level is evaluated on authoritative bases. That is, for each sampling method, the extracted sampled trace is compared with established reference data provided by Hamou-Lhadj et al. [HL06]. The reference data is a summarized version of the original trace (hereby referred to as the oracle trace) that has been shown to have captured the most important interactions of the trace and to be effective in program comprehension. The comprehension score of a sampled trace is the percentage of events that exist in both the sampled trace and the oracle trace. This way, a higher comprehension score of a sampled trace represents a higher comprehension level and a lower comprehension score represents a lower comprehension level that can be achieved using the sampled trace. It is important to note that the comprehension score calculated by means of comparison to the oracle trace represents, by consequence, a subjective feedback, and that more objective measurements such as those used to assess the comprehension level during the maintenance tasks may result in a more precise conclusions.

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

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

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

Experiment Design and Procedure

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

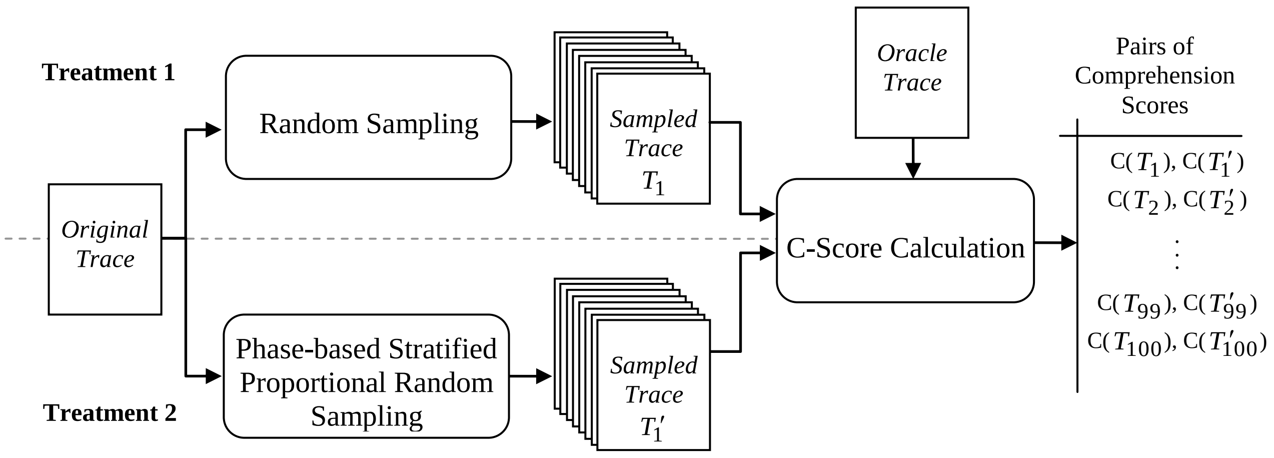


Figure 11. Overview of the experiment

For this, as shown in Figure 11, we apply random sampling on the original trace to obtain a sampled trace T_1 of a given size. Similarly, we apply the phase-based stratified random sampling on the original trace to obtain another sampled trace T'_1. Both T_1 and T'_1 are of a given size s (equal to the oracle trace size) and the sampling is performed without replacement. The C-Score Calculation module, receives the sampled trace T_1 and compares it with the oracle trace and assigns it a comprehension score C(T_1), the percentage of the elements that are common between T_1 and the oracle trace. Similarly, a C(T'_1) is assigned to T'_1. The pair of (C(T_1), C(T'_1)) is added to the list of observations for statistical analysis.

Statistical Analysis:

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

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

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

Results

This section reports and analyzes results obtained from our experiments[1]. The original trace contained 97,413 method calls. We applied our phase detection technique on the trace to detect its major phases (i.e., threshold *t* = trace size). The application of our integrated scheme results in the formation of 11 dense groups in the rearranged trace. These groups can be seen in Figure 12. These 11 groups indicate 11 phases when mapped back the original trace. Table 2 shows the detected phases and their information. On an Intel Core i5 CPU 2.30GHz, 4.00 GB main memory, running Windows 7 it took 12.838 seconds for our phase detection algorithm to detect phases. Out of this time, 1.375 seconds were spent on the application of the integrated scheme and 11.463 seconds were spent on clustering.

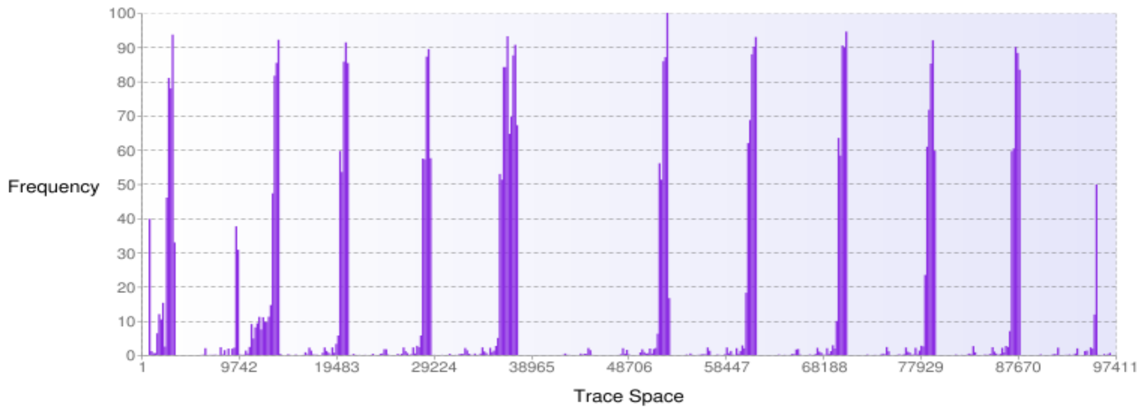


Figure 12. Eleven dense groups are resulted from applying the integrated scheme on the WEKA trace.

These phases were then used as strata to perform stratified sampling. To implement the random sampling within each stratum, our program randomly draws a method call from that stratum and excludes the drawn method call from further selection. The oracle trace contained 31 method calls. We executed the experiment 100 times (obtain 100 pairs of observation) with *s* = 31. Table 4 shows the list of observations.

A pair-wise application of the Wilcoxon test shows that comprehension scores of phase-based random sampling are significantly higher than random sampling scores, ($z = -2.628$, $n = 100$, $p = 0.009$, two-tailed). This information is shown in details in Table 3. This table tells us that the statistic is based on the negative ranks, that the *z*-score is –2.628 and that this value is significant at $p = 0.009$. Therefore, because this value is based on the negative ranks, we should conclude that there was a significant increase in comprehension score from random sampling to phase-based stratified random sampling.

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

Table 2. Phases information

Phase	Phase Location	Size
P1	1 – 8,836	8,836
P2	8,837 – 19,299	10,463
P3	19,300 – 27,673	8,374
P4	27,674 – 35,500	7,827
P5	35,501 – 51,651	16,151
P6	51,652 – 60,537	8,886
P7	60,538 – 69,728	9,191
P8	69,729 – 78,640	8,912
P9	78,641 – 87,275	8,635
P10	87,276 – 95,811	8,536
P11	95,812 – 97,413	1,602

Table 3. Statistical analysis

Descriptive Statistics

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

Ranks

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

a. Phase-based Stratified Sampling < Random Sampling
b. Phase-based Stratified Sampling > Random Sampling
c. Phase-based Stratified Sampling = Random Sampling

Test Statistics[a]

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

a. Wilcoxon Signed Ranks Test
b. Based on negative ranks.

Table 4. Comprehension scores of trace resulted from random sampling and stratified sampling

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

8.2. Second Case Study

In this section, we show how our approach compares to random sampling with regards to the problematic cases discussed in Section 3. To generate an execution trace, we used an execution scenario that involves several major features. The execution trace was generated based on a scenario where the features F1 to F12 shown in Table 5 were exercised one after another.

Table 5. The features included in the traced scenario

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

Because JHotDraw registers all mouse movements, and mouse movements are required while drawing figures, the trace that resulted from our scenario was bound to contain a lot of noise. We have therefore filtered these mouse movements to obtain a trace that is cleaner. We are aware that the detection of noise in a trace might not always be straightforward and that noise detection techniques such as the ones presented by Hamou-Lhadj et al. in [HL06] might need to be used. The resulting trace contained 36,571 method invocations and the trace file was of size 1.8 MB. Note that a method invocation requires at least two events to be collected, the entry and exit of a method. The trace size in terms of events is therefore about 73,142 events, which is considered a relatively medium-sized trace.

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

Figure 13 shows the results of applying the integrated gravity, using *t*, to the JHotDraw trace. The results are shown in the form of a histogram, where the *x*-axis shows the distance between the positions of the calls and the *y*-axis shows the frequency (the number of methods that their position falls into one interval of *x*-axis). As part of our technique, in order to automatically determine the number of phases and their location, the trace resulted from applying the integrated gravity scheme is partitioned by K-means clustering for *K* from 1 to 10. The highest BIC score was for the partitioning with $K = 8$ as the best fit. Figure 13 shows the location of the eight clusters (P1 to P8), highlighted by dashed rectangles. These phases are explained in Table 6. Each row contains the location of the phase in the execution trace, the task performed in the phase, and the corresponding stratum. This information was used for stratification. As shown in this table, we were able to use our phase detection technique to successfully recover parts of the trace that implement each of the features that were traced. We validated the results by browsing the source code and reviewing JHotDraw documentation. We want also to note that we have a very good understanding of JHotDraw design in our research team. We have conducted other studies using the same system.

Table 6. Execution phases that were detected

Phase	Phase Location	Description	Strata	Size
P1	1 – 1134	Initialization	Stratum 1	1,134
P2	1135 – 10948	New sheet, F1, F2	Stratum 2	9,814
P3	10949 – 14816	F3, F4	Stratum 3	3,868
P4	14817 – 22391	F5, F6	Stratum 4	7,575
P5	22392 – 26298	F7, F8	Stratum 5	3,907
P6	26299 – 32812	F9, F10	Stratum 6	6,514
P7	32813 – 36461	F11, F12	Stratum 7	3,649
P8	36462 – 36571	Finalization	Stratum 8	110

We chose to generate three sampled traces from the original trace that vary in size and assess their effectiveness in being representatives of the original traces. The three sampled traces are respectively of sizes 3,646, 1,829, and 365, shown as Strat1, Strat2, and Strat3 in Table 7. In this table, the sampling parameter $|Stratum_h| / |T|$ for each stratum is calculated according to the size of the stratum $Stratum_h$ and the size of original trace ($|T|$ =36,571). Then, the sampling parameter and $|T'|$ the sample size are used for calculation of the number of calls to be sampled from each stratum (i.e., $|s_h|$). For example, we can see that to obtain a sample trace of size 365 we need to randomly sample 12 calls from "Stratum 1" of the original trace, 96 calls from "Stratum 2" and so on.

The next step is to assess the representativeness of the three samples with respect to the original trace. We compared the sample traces generated by our approach to the ones generated using mere random sampling. For this purpose, we first created three other sampled traces (Rand1,

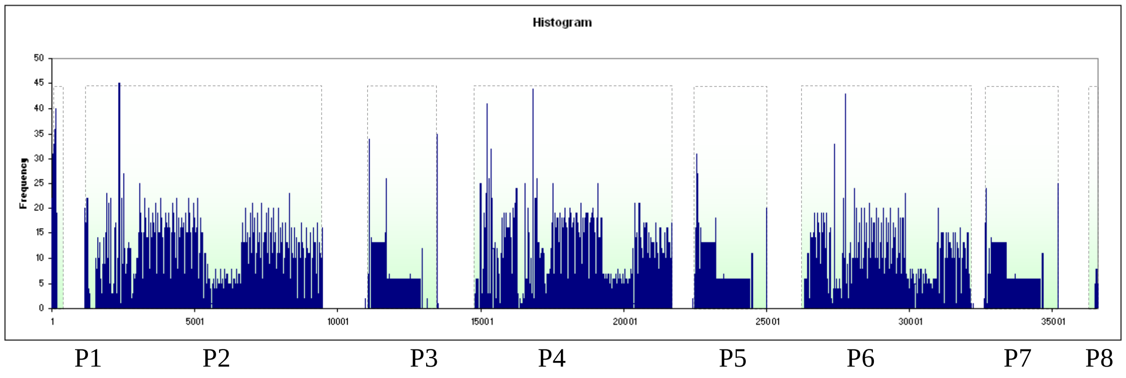


Figure 13. Detected phases in JHotDraw trace

Rand2, and Rand3) of sizes 3,646, 1,829, and 365 using random sampling from our original trace.

Table 7. Contribution of features of JHotDraw trace to the size of the sample traces using both stratified and random sampling

	10% of Orig. Size (**3,646**)		5% of Orig. Size (**1,829**)		1% of Orig. Size (**365**)	
	Strat 1	Rand 1	Strat2	Rand2	Strat3	Rand 3
Initialization	113	115	57	51	12	5
NewSheet	47	47	22	14	4	6
F1	31	31	14	13	3	2
F2	915	914	460	464	90	98
F3	420	418	211	225	46	47
F4	30	35	15	12	3	4
F5	13	11	7	8	1	2
F6	674	672	339	354	63	55
F7	14	11	6	9	1	1
F8	381	368	191	197	39	36
F9	15	8	6	6	1	1
F10	598	653	308	308	63	74
F11	21	14	8	4	1	15
F12	362	342	182	157	37	14
Finalization	11	7	6	7	1	0

Table 8. The samples obtained from applying our approach to JHotDraw trace

Strata	N_h	$\lvert Stratum_h \rvert / \lvert T \rvert$	$\lvert S_h \rvert$ ($\lvert T' \rvert$ = **365**)	$\lvert S_h \rvert$ ($\lvert T' \rvert$ = **1,829**)	$\lvert S_h \rvert$ ($\lvert T' \rvert$ = **3,646**)
1	1134	0.031	12	57	113
2	9815	0.265	96	492	983
3	3868	0.105	39	193	384
4	7575	0.207	76	379	756
5	3907	0.106	39	194	387
6	6514	0.178	65	326	650
7	3649	0.099	37	182	362
8	109	0.003	1	6	11

As mentioned earlier, a sampled trace is representative if it closely resembles the execution trace from which it is drawn. This resemblance could be quantified by the extent of similarity of statistical characteristics between the original trace and each set of generated samples. We consider the distribution of features and their contribution to the overall size of the original execution trace as our comparison reference. The closer the distribution of features in sample traces is to the actual distribution of features in the original trace, the more representative the sampled trace is. The distribution of features in each sample trace in terms of their contribution to the size of the sample trace is shown in Table 8. For example, the number of calls belonging to feature F12 in the sample trace “Strat 3” is 37; its contribution to the size of “Strat 3” (i.e., 365) is 10.13%. The same feature contributes only 14 calls (3.83%) to the sampled trace “Rand 3”, generated using random sampling. When we contrast this with the contribution of the features to the size of the entire trace (shown in Table 9), we can see that feature F12 represents 10.01% of the size of the original. Therefore, F12 is better represented in “Strat 3” than in “Rand 3”.

When we applied the same reasoning to all features, we found that our approach provided better results in more than 80% of the cases and in all these cases, it led to a more representative sampled trace. Furthermore, as the size of the sample decreases, our approach maintains its representativeness while random sampling, as theoretically discussed in Section 2, leads to cases that are significantly unrepresentative of the original trace. Some of these cases are shown in Figure 14. For example, as it can be seen in Figure 14, the trace “Rand 3” contains a sudden high number of calls from feature F11 (shown as the rightmost yellow bar) while the percentage of the

Table 9. Contribution of features to the size of the entire trace

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

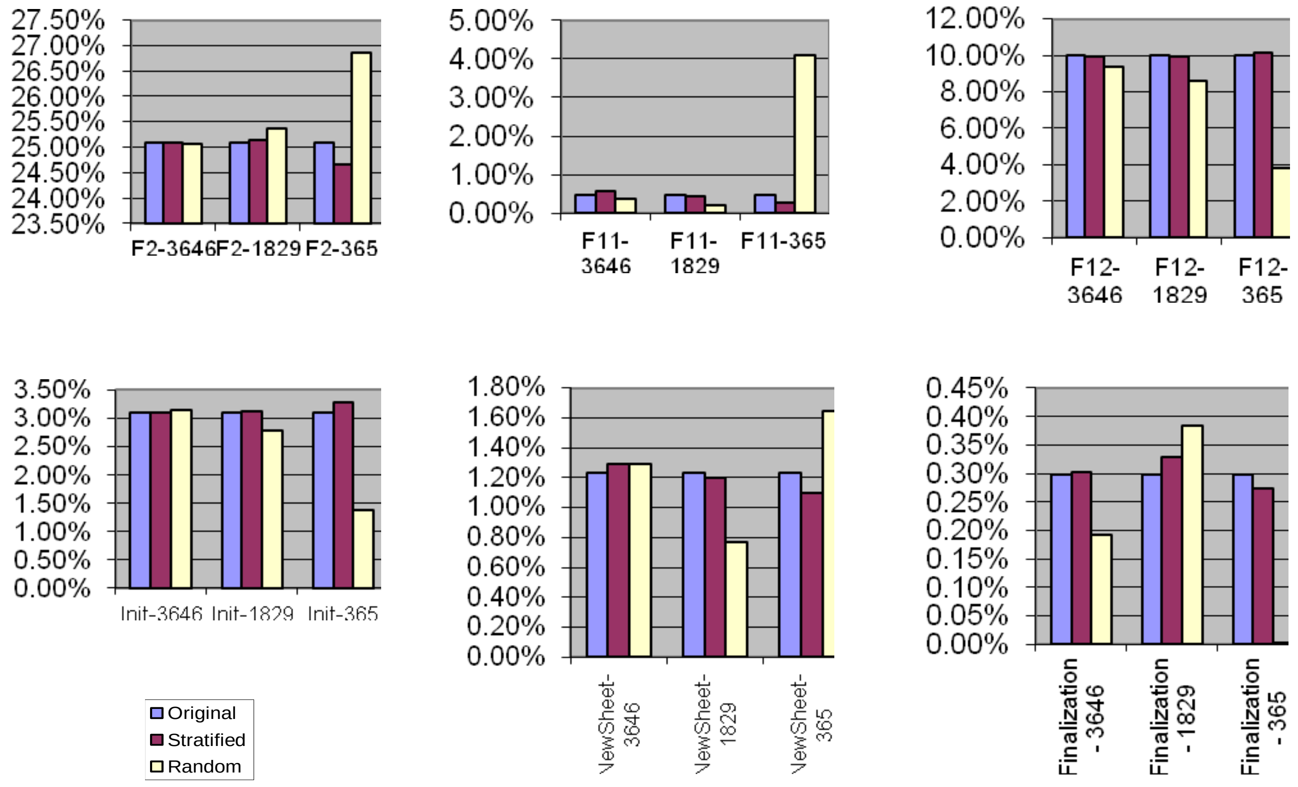


Figure 14. Comparison between the distribution of a number of features in the original trace, stratified sample, and random sample. The *x* axis shows the contribution of the feature to the size of the trace in percentage. The *y* axis shows the feature name and the sample size.

presence of the same feature in all the traces generated in our approach (the three plum bars) is maintained close the its percentage in the original trace (the blue bars). In all cases reported in Figure 14 show that our approach maintains the same distribution of features with respect to the original trace, which is not the case for random sampling. This demonstrates (at least for this case study), the superiority of our approach compared to random sampling.

9. Threats to Validity

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

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

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

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

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

10. Conclusion and Future Work

In this paper, we presented a novel sampling technique of large execution traces that does not only reduce the size of traces by also generates samples that are representative of the original traces. Our approach relied on stratified sampling, using execution phases as strata. We defined an execution phase as part of a trace that represents a specific task (or feature). We presented a phase detection algorithm based on Gestalt laws of gravity.

When applied to a trace generated from two target systems, our approach showed promising results. We recognize that further studies and experiments are needed. We also want to draw the attention to the need to investigate proper thresholds for the proposed phase detection algorithm. Several other future directions are needed to address the points raised in the previous section. We also intend to look into signal processing techniques and other related research areas in which sampling is used extensively to improve our sampling technique.

REFERENCES

[ACC+02] Antoniol, G.; Canfora, G.; Casazza, G.; De Lucia, A.; Merlo, E.;, "Recovering traceability links between code and documentation," Software Engineering, IEEE Transactions on , vol.28, no.10, pp. 970- 983, Oct 2002.

[AG05] Antoniol, G.; Guéhéneuc, Y.-G.;, "Feature identification: a novel approach and a case study," Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on , vol., no., pp. 357- 366, pp. 26-29, Sept. 2005.

[ADS09] Abeel, T.; de Peer, Y. V.; Saeys, Y.;, Java-ML: A Machine Learning Library, Journal of Machine Learning Research, 10, pp. 931-934, 2009.

[BMW94] Biggerstaff, T.J.; Mitbander, B.G.; Webster, D.E.; , "Program understanding and the concept assignment problem". Communications of the ACM, vol. 37, no. 5, pp. 72-82, May 1994.

[Bru60] Brunk, H.D.; , "An introduction to mathematical statistics", Ginn and Company, Boston, vol., no., 1960.

[CHMY03] Chan, A.; Holmes, R.; Murphy, G.C.; Ying, A.T.T.; , "Scaling an object-oriented system execution visualizer through sampling," Program Comprehension, 2003. 11th IEEE International Workshop on , vol., no., pp. 237- 244, 2003.

[CHZ+07] Cornelissen, B.; Holten, D.; Zaidman, A.; Moonen, L.; van Wijk, J.J.; van Deursen, A.; , "Understanding Execution Traces Using Massive Sequence and Circular Bundle Views," Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on , vol., no., pp.49-58, pp. 26-29, 2007.

[Coc77] Cochran, W. G.; , "Sampling Techniques". John Wiley & Sons, Inc., New York, NY, , vol., no., 1977.

[Con80] Conover WJ. Practical nonparametric statistics. New York (NY): John Wiley & Sons; 1980.

[CR00] Chen, K.; Rajlich, V.;, "Case Study of Feature Location Using Dependence Graph", Program Comprehension, IWPC'00. Proceedings of 8th International Workshop on, pp. 241-249, 2000.

[Dug07] Dugerdil P.; , "Using trace sampling techniques to identify dynamic clusters of classes", CASCON'07, Proceedings of the IBM Software and Systems Engineering Symposium, vol., no., pp. 306-314, 2007.

[ED05] Eisenberg, A.D.; De Volder, K.; , "Dynamic feature traces: finding features in unfamiliar code," Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on , vol., no., pp. 337- 346, pp. 26-29, 2005.

[EKS03] Eisenbarth, T.; Koschke, R.; Simon, D.; , "Locating features in source code," Software Engineering, IEEE Transactions on , vol.29, no.3, pp. 210- 224, 2003.

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

[GPSG01] Geisler, W. S.; Perry, J. S.; Super, B. J.; Gallogly, D. P., "Edge Co-Occurrence in Natural Images Predicts Contour Grouping Performance", Vision Research Journal, vol. 41, no. 6, pp. 711-724, 2001.

[GMSS00] Ghosh, G. K. ; Michael, C.; Schatz, M.; and Schatz, M.,"A Real-Time Intrusion Detection System Based on Learning Program Behavior," in Proc. of the third Intl. Workshop on Recent Advances in Intrusion Detection, Toulouse, France, pp. 93-109, 2000.

[HL06] Hamou-Lhadj, A.; Lethbridge, T.;, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System," Program Comprehension, ICPC 2006. 14th IEEE International Conference on , vol., no., pp.181-190, 2006.

[HL04] Hamou-Lhadj, A.; Lethbridge, T.;, "A Survey of Trace Exploration Tools and Techniques", Collaborative research, CASCON '04, Proceedings of the 2004 Conference of the Centre for Advance Studies on , vol., no., pp. 42-54, 2004.

[HLBL05] Hamou-Lhadj, A.; Braun, E.; Amyot, D.; Lethbridge, T.: , "Recovering Behavioral Design Models from Execution Traces", Software Maintenance and Reengineering, CSMR'05, Proceedings of the 9th European Conference on, Manchester, UK, pp. 112-121, 2005.

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

[JSB97] Jerding, D. ; Stasko, J.; Ball, T.;, "Visualizing Interactions in Program Executions", In Proc. of the International Conference on Software Engineering, ACM Press, pp. 360-370, 1997.

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

[KDMS07] Kothari, J.; Denton, T.; Mancoridis, S.; Shokoufandeh, A.;, "Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence", Program Comprehension, 2007. ICPC 2007. The 15th IEEE International Conference on, Banff, Canada, 2007.

[KG06] Kuhn, A.; Greevy, O.; , "Exploiting the Analogy Between Traces and Signal Processing," Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on , vol., no., pp.320-329, pp. 24-27, 2006.

[Kof35] Koffka, K.; , "Principles of Gestalt Psychology". Hartcourt, New York, 1935.

[LAZJ03] Liblit, B.; Aiken, A.; Zheng, A.X.; Jordan, M.I.; , "Bug isolation via remote program sampling", programming language design and implementation, PLDI'03, Proceedings of the ACM SIGPLAN 2003 conference on , vol., no., pp. 141–154, 2003.

[Mac67] MacQueen, j.; "Some Methods for Classification and Analysis of Multivariate Observations", Math Statistics and Probability, Proceedings of the 5th Berkeley Symposium on, vol., no., pp. 281-296, 1967.

[MM03] Marcus, A.; Maletic, J.I.; , "Recovering documentation-to-source-code traceability links using latent semantic indexing," Software Engineering, ICSE'03. Proceedings of 25th International Conference on , vol., no., pp. 125- 135, 2003.

[MRS08] Manning, C. D.; Raghavan, P.; Schtze, H.;, "Introduction to Information Retrieval." Cambridge University Press, New York, NY, USA, 2008.

[PAH10] Pirzadeh, H.; Agarwal, A.; Hamou-Lhadj, A.; , "An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension," In proceedings of Software Engineering Research, Management and Applications (SERA), 2010 Eighth ACIS International Conference on , vol., no., pp.207-214, 2010.

[PH11a] Pirzadeh, H.; Hamou-Lhadj, A.; , "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, pp. 221–230, 2011.

[PH11b] Pirzadeh, H.; Hamou-Lhadj, A.; "A Software Behaviour Analysis Framework Based on the Human Perception Systems", In proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), New Ideas and Emerging Results Track, pp. 948–951, 2011.

[PM00] Pelleg, D.; Moore, A.; , "X-means: Extending K-means with efficient estimation of the number of clusters", Machine Learning, ICML'00, Proceedings of 17th International Conference on , vol., no., pp. 727–734, 2000.

[PSHM11] Pirzadeh, H.; Shanian, Hamou-Lhadj, A.; Mehrabian, A.; , "The Concept of Stratified Sampling of Execution Traces", In proceedings of the 19th International Conference on Program Comprehension (ICPC 2011), Kingston, Ontario, Canada, June 2011.

[QB09] Quinn, P. C.; Bhatt, R. S.; , "Perceptual organization in infancy: Bottom-up and top-down influences", Optometry and Vision Science (Special Issue on Infant and Child Vision Research: Present Status and Future Directions), vol. 86, no. 6, pp. 589–594, 2009.

[Rei05] Reiss, S. P.; , "Dynamic detection and visualization of software phases", Dynamic Analysis, WODA'05, Proceedings of the 3rd International Workshop on , pp. 1-6, 2005.

[Rei07] Reiss, S. P.; , "Visual representations of executing programs", Journal of Visual Languages and Computing, vol. 18, no. 2, 2007.

[RHR08] Rohatgi, A.; Hamou-Lhadj, A.; Rilling, J.; , "An Approach for Mapping Features to Code Based on Static and Dynamic Analysis," Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on , vol., no., pp.236-241, 2008.

[RR03] Reiss, S. P.; Renieris, M.; , "The BLOOM Software Visualization System", In Software Visualization – From Theory to Practice, MIT Press, vol., no., 2003.

[RZ05] Roberts, J.; Zilles, C.; "TraceVis: an execution trace visualization tool", Modeling, Benchmarking and Simulation, MoBS'05, Proceedings of Workshop on , vol., no., pp. 5-15, 2005.

[Sch78] Schwarz, G.; , "Estimating the dimension of a model", The Annals of Statistics, vol. 6, no. 2, pp. 461–464, 1978.

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

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

[Sys00] Systä, T.;, "Understanding the Behaviour of Java Programs", In Proceedings of the 7th Working Conference on Reverse Engineering, IEEE Computer Society, pp. 214-223, 2000.

[TH98] Tzerpos, V.; Holt. R. C.;, "Software botryology: Automatic clustering of software systems. In DEXA Workshop, pp 811–818, 1998.

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

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

[WGZ04] Wang, W.; Guan, X. H.; and Zhang, X. L. "Modeling program behaviors by hidden Markov models for intrusion detection," in Proc. of Intl. Conf. on Machine Learning and Cybernetics, Shanghai, China, pp. 2830-2835, 2004

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

[WGH00] Wong, W.E.; Gokhale, S.S.; Horgan, J.R.; , “Quantifying the closeness between program components and features”, Journal of Systems and Software - Special issue on software maintenance, vol. 54, no. 2, pp. 87-98, 2000.

[WII08] Watanabe, Y.; Ishio, T.; Inoue, K.; , “Feature-level phase detection for execution trace using object cache”, Dynamic Analysis, WODA’08, Proceedings of the International Workshop on , vol., no., pp. 8–14, 2008.

[WS95] Wilde, N.; Scully, M.C.; , “Software Reconnaissance: Mapping program features to code”, Journal of Software Maintenance: Research and Practice, vol. 7, no. 1, pp. 49-62, 1995.

[Zai06] Andy Zaidman, "Scalability Solutions for Program Comprehension through Dynamic Analysis", PhD. Dissertation, Universiteit Antwerpen, 2006.

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