TRIADE: A Three-Factor Trace Segmentation Method to Support Program Comprehension

Raphaël Khoury[1], Abdelwahab Hamou-Lhadj[2]
[1]Département d'informatique et de mathématique
Université du Québec à Chicoutimi, Canada
Saguenay, Canada
raphael.khoury@uqac.ca, shale@acm.org,
fabio@petrillo.com

Mohamed Ilyes Rahim[1], Sylvain Hallé[1], Fabio Petrillo[1]
[2]Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada
wahab.hamou-lhadj@concordia.ca

***Abstract*—Trace analysis allows software engineers to gain insights into the behavior of the systems they maintain, and thus serves as an essential tool to aid in multiple tasks that require an understanding of complex systems, including security analysis, debugging and maintenance. However, the considerable size of execution traces can hinder the effectiveness of trace analysis. There exist techniques that extract higher level abstractions from a lengthy trace by automatically segmenting a trace into a number of cohesive segments, allowing software engineers to focus only on the segments of interest. In this paper, we improve on related work on segmenting traces of method calls by considering three factors: method names, method calling relationship, and method parameters. We show experimentally that this approach is more effective for the purpose of dividing a trace in a manner concordant with the underlying behavior of the program than existing algorithms. We also examine the issue of key element extraction from a trace, and again demonstrate experimentally that traces segmented using our method can more readily be subjected to this analysis.**

Keywords: Trace Analysis, Trace Segmentation, Gestalt Psychology, Program Comprehension.

I. Introduction

Execution traces are logs that record a program's execution. These traces form an essential basis for several types of program analysis and software engineering tasks including debugging, feature enhancement, and performance analysis. However, the considerable size of execution traces quickly becomes an obstacle to their use. For example, Hamou-Lhadj et al. [6] and Cornelissen et al. [1] found that even running simple but representative executions of basic programs generates extremely large traces, up to millions of lines of events.

To reduce the size of traces, while keeping their main content, several trace abstraction techniques have been proposed (e.g., [2], [6]). These techniques operate in different ways. Some focus on detecting and eliminating utilities and other low-level events from a trace, and hence exposing its main content. Others leverage visualization approaches to guide developers through the trace exploration process.

Recently, a new family of trace abstraction approaches has emerged. These techniques, known as trace segmentation (see for example [8], [18], [20]), concentrate on the problem of splitting the trace into smaller segments; each represents an execution phase of a system. For example, the execution of a program naturally begins with a start-up phase during which the program components are loaded and initialized. Each subsequent action demanded by the user, such as opening a file or activating a functionality of the program, consists of another distinct phase. Each segment may consist of several thousand trace events, but for program comprehension purposes, it is useful to treat it as a single atomic phase. A tool that supports trace segmentation can highlight these segments to allow analyst explore their content without necessarily worrying about the other parts of the trace.

Automatic identification of trace segments is not an easy task. The problem is that there is just so much variation in the trace that makes the detection of cohesive blocks a complex problem [8], [18], [20]. To address this, Pirzadeh and Hamou-Lhadj [18], [19] proposed a novel trace segmentation framework based on Gestalt laws of perception, which describe how people group similar items visually based on their perception. Gestalt psychology models the processes that occur in the brain when we see a scene, and how our perceptual systems follow certain grouping principles (e.g., good continuation, proximity, and similarity properties of the elements) [11] to integrate the scene elements (i.e., objects and regions) as a whole and not just as points and lines. In subsequent work [18], [20], the authors modeled these processes in the context of trace segmentation, more particularly they proposed two gravitational schemes based on Gestalt laws of good continuation and similarity to detect cohesive methods in a trace of method calls. Once the segments are identified, they proposed to use information retrieval techniques to extract key elements from trace segments, constructing high-level summaries from large and complex traces that can be used for trace exploration, recovery of system documentation, etc.

Their proposed trace segmentation approach groups methods into cohesive groups by looking at two factors: method names, and the calling relationship. In this paper, we improve their approach by introducing a new factor: method parameters. In other words, two methods are thought to be in the same program segment if they manipulate the same values or objects, or if the return value of one method is then used as the parameter of another method. We call our approach *TRIADE*, a three-factor trace segmentation method.

In addition, for the problem of key trace element extraction, we compare two alternative algorithms, the one that uses information retrieval metrics, more particularly TF-IDF (Term Frequency and Inverse Document Frequency) [7], based on

the work of Pirzadeh et al. [18], [20], and the use of the Helmholtz principle [3], a principle in psychology that explains how humans can identify patterns that diverge from randomness.

In short, the contributions of this paper are as follows:

- An improved method call trace segmentation process that leverages method parameters and return values in addition to method names and method calling relationship.

- A comparison between the use of the Helmholtz principle and TF–IDF for the problem of key trace element extraction.

We evaluated our work on traces of two open source Java systems. The results show that the use of method parameters, in addition to method names and method calling relationship, improves the accuracy of the segmentation process. We also show that the Helmholtz principle yields better results than TF-IDF.

The remainder of this paper is organized as follows: in Section II, we describe our 3-factor segmentation method and we outline the two key element extraction algorithms that we wish to compare. Section III provides an experimental comparison of this segmentation method with an existing one. Section IV presents an overview of related works. Concluding remarks are given in Section V.

II. TRIADE: THE 3-FACTOR TRACE SEGMENTATION METHOD

Taken together, trace segmentation and key element extraction can work in tandem to provide an effective method to infer a program's behavior from a trace. This is done through the following four-step process:

1) First, the trace is divided into its constituting segments using a trace segmentation algorithm, with each segment corresponding to a higher-level execution phase.
2) Second, we apply a key element extraction algorithm to each segment. This results in the creation of a summary of each trace segment, containing its most distinctive method calls.
3) In the third phase, the summaries are pruned of utility methods using an algorithmic process. Utility methods are short generic methods whose presence in a summary is not informative from the perspective of system maintainers who seek to understand the underlying behavior of the trace.
4) Finally, the size of the summary is further minimized by taking only the top method names (using a threshold that can be defined by a software maintainer), as ranked by the key element extraction algorithm.

At the end of this process, an intractably large trace is reduced to a series of a handful of short summaries, each of which consists of a small number methods. It is then a simple matter for a system maintainer to consult the documentation of the underlying program and understand the behavior of the program during each execution phase.

A. Trace Segmentation

The trace segmentation process proposed by Pirzadeh et al. [18], [20] operates in two steps namely phase detection and phase boundary identification. In the first step, trace elements are grouped into candidate phases using gravitational schemes that rely on the principles of similarity and good continuation. In the second step, the boundary of each phase is identified using a k-means clustering algorithm [16].

B. Step 1: Phase detection

To detect candidate phases using a gravitational scheme, each element of the trace is first assigned a position on an axis. Initially, positions are assigned sequentially, with each element equidistant from its predecessor and its successor, such that each element's position corresponds to its index in the trace (Figure 1). Elements that are deemed "similar" based on some criteria are then slid along this axis, as if they were pulled by gravitational forces resulting in the formation of dense clusters of similar elements.

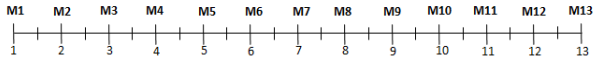

Figure 1: Schematic view of a trace before the beginning of the segmentation process.

The elements of the trace were then repositioned based on a similarity scheme, where syntactically identical element, (i.e. multiples calls to the same method) are brought closer together (Figure 2). Second, a repositioning based on a continuity scheme is applied to the vector resulting from this initial repositioning (Figure 3). The continuity scheme reduces the distance between two method calls according to their nesting levels: calls with a higher nesting level are repositioned closer to the methods that have called them.

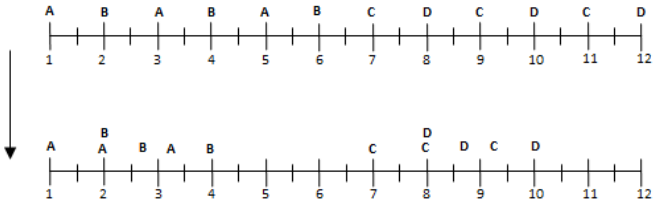

Figure 2: An example of applying the similarity scheme to a trace.

In this paper, we propose to add method parameters and return values as another criterion for bringing related methods closer. In other words, we also segment traces according to a parameter scheme (schematized in Figure 4). Two method calls are considered similar, and consequently repositioned in closer positions, if they share the same parameters or return values, or if one method's return value is used as parameter for the other. This is based on the intuition that if two methods manipulate the same objects or if a method takes as input an object or value returned by another method, then these two methods are likely to be part of the same execution phase. We found that the use of parameters provides substantial improvement to the effectiveness of the segmentation algorithm.

The fact that the segmentation process is now based on three factors also opens up the possibility of assigning a weight to each of three segmentation schemes (similarity, continuity, and parameters). This skews the analysis to assign more importance to the factor captured by this scheme. The choice of this value can be determined through a combination of code analysis and experimentation. In practice, a user may make an educated guess about which of the three factors under consideration is likely to be more determinant. For example, a compiler may apply multiple successive transformations to the same objects. As a result, parameters values would be a poor indicator of execution phase since the same object handler would appear as a parameter to methods occurring in different execution phases. Conversely, for a recursive program, similarity and continuity will carry more meaningful information.

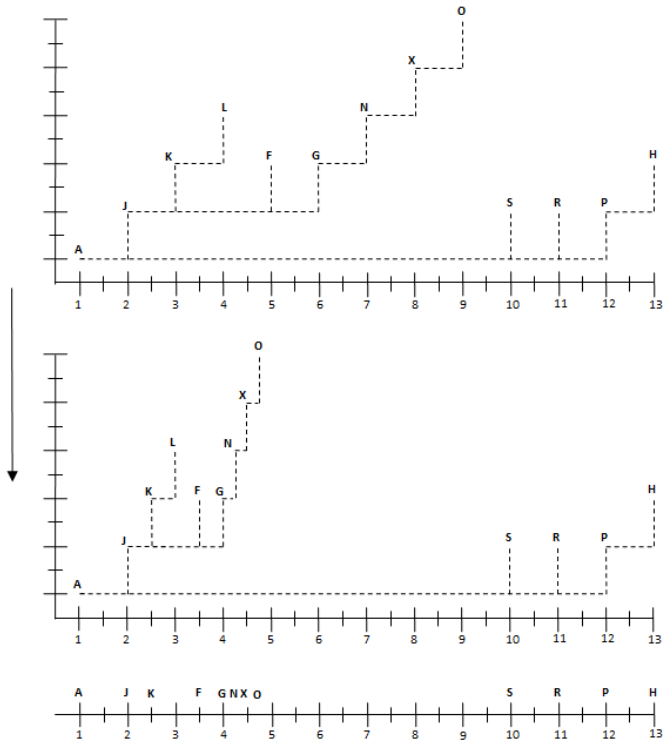

Figure 3: An example of applying continuity scheme to a trace.

C. Step 2: Phase Boundary Identification

After the repositioning schemes (similarity, continuity, and parameters) has been applied, each method's position has been altered to increase its proximity to other methods sharing the same name, the same parameters, as well as to those that are part of the same nesting hierarchy. We then apply the k-means clustering algorithm to identify the most apt locations for the beginning and end of phases. K-means [13] is a widely used clustering algorithm that allows n observations to be grouped into k clusters, in such a way as to minimize the variance inside each cluster. In our case, the observations are the methods calls, positioned in a vector space and the clusters will correspond to segments.

While k-means has been shown effective at this task in previous research, it does have a drawback: the expected number of clusters has to be specified manually by the user. Several algorithmic methods have been proposed to select the correct number of clusters in an automated manner. Pirzadeh et al. [20] surveyed these alternatives and suggested their own solution, namely to generate multiple alternative partitions, with different values of k, and compare them using a Bayesian Information Criterion [16] to determine which value is more likely to be accurate. In Section 6, we will suggest a method by which trace segmentation and key element extraction can work in tandem to identify the correct number of phases.

D. Key Trace Elements Extraction

As discussed above, the purpose of key element extraction is to generate a 'summary' that highlights the most salient aspects of the trace. This is often necessary since even after the trace has been segmented into its constituent phases, each segment is still too long to be analyzed and understood by a programmer. In this section, we present two techniques that can be used to accomplish this task. Both techniques have their origin in natural language processing and Gestalt psychology, and have since been adapted to the problem of trace analysis.

1) TF-IDF: TF-IDF [7] is a technique that has its origin in the processing of natural language texts. It provides a measure of the importance of a word in a given document, when that document is placed in the context of a more general corpus of texts, containing several documents. While several variants have been proposed, the main idea is to assign a score to each word in a document. This score *increases* as the number of occurrences of that word rises in the document, but *decreases* if the word occurs frequently in the other texts of the corpus. As a result, frequently occurring words, such as 'the' or 'and' exhibit a low score, since they are common throughout the corpus. On the other hand, words that occur frequently in a given text, but are rare in the remainder of the corpus, would exhibit a high TF-IDF score. These words are likely to be the most meaningful ones in the document under consideration.

Pirzadeh et al. [20] showed that this technique can be used effectively to extract key elements from segmented execution trace. The method treats each method call as a word, and treats each segments as documents. Thus, TF-IDF will filter out method names that occur frequently throughout the execution trace, and assign high weights to method calls that are frequent in a given segment, but are otherwise rare in the remainder of the trace.

2) Helmholtz: Dadachev et al. [3] proposed an alternative key element extraction algorithm, and Lei et al. [8] provide a detailed algorithmic procedure to apply this method to the problem of extracting key elements from execution traces. Their method draws upon the Helmholtz principle [12], a principle of psychology that states that humans more readily detect patterns when they diverge from randomness. In regard to the analysis of function calls in a trace, we designate as a meaningful event any function call that occurs in a given phase at a rate that exhibits a large deviation from randomness, as compared with its rate of occurrence in the entire trace.

Consider a trace T that is partitioned into P phases (T_1, T_2, …, T_P), using the method given in stage 1. Let M be a function call that is present in one or more of these P phases. Assume that the method call M appears a total of K times in

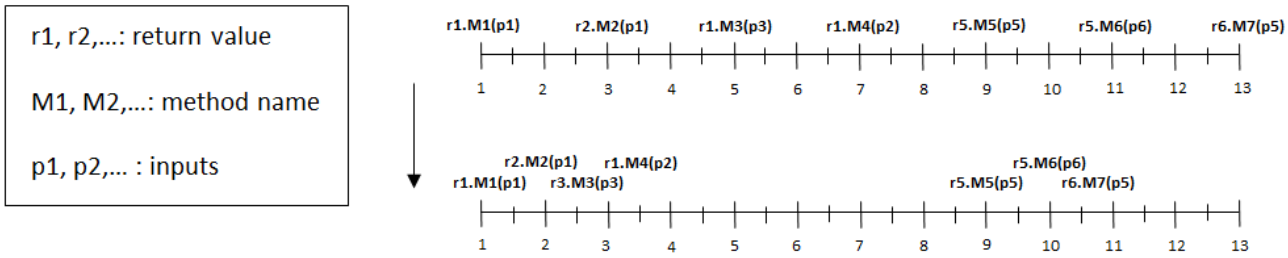

Figure 4: An example of parameter scheme.

all P phases. These occurrences of M are collected in a set $S_M = \{M_1, M_2, ..., M_K\}$.

Now consider the possibility that the function call M occurs n times in some phase T_P. A probabilistic analysis can assign a likelihood to whether the number of occurrences of M in TP is either an unexpected or expected event. The key elements of the segment are those for which this probability is greater than a threshold.

E. Utility Elimination and Summary Minimization

Previous research [17] shows that system analysis is often encumbered when the data set being analyzed contains too many *utilities*: short methods that provide a generic service to multiple clients. The same utilities could be called throughout the execution of a program, and the computations they perform tend to be abstract. As a consequence, no information about the program's behavior could be gleaned from their presence in a trace-segment summary.

Hamou-Lhadj and Lethbridge [17] studied extensively the problem of utilities in traces. They defined utilities as methods that are called from multiples distinct call sites, providing an algorithmic basis for detecting utilities, and removing them from the summaries, since their presence represents unwanted noise in a process aimed at understanding the underlying behavior of an execution trace.

We experimentally set at 20 the threshold of distinct call sites after which a method is designated as a utility and elided from the summary. In our subsequent experiments, only a very small number of methods (much less than 1%) met this threshold, and the final summaries still contain numerous utility method calls. However, even this small number was sufficient to improve the quality of our results to the point where the behavior of each segment can always be gleaned from the segment's summary.

As a final step, any summary comprising more that 10 methods is reduced in size by selecting only the top ten methods (using the ranking created by the TF-IDF and Helmholtz algorithms), and deleting all the others. This step ensures that the summary always has a size that is tractable to a manual human analysis. In those cases where the summary consists of less than 10 methods, this step is skipped entirely. It is a simple task to examine the documentation of 10 or fewer methods in order to have an idea of the segment's underlying behavior from them.

III. Experimental Results

We evaluated the effectiveness of our approach by generating traces from open source Java programs. While we performed multiples tests with a wide variety of programs, we selected two representative experiments to present here out of space considerations. The programs we selected are JHotDraw[1] and ArgoUML[2]. We used AspectJ [10] to generate execution traces from these programs, containing 3 or 4 distinct user-actions respectively. Each line of the trace consists in a single method call or method return, and indicates the method's name, the type and value of each of the method's parameters, its nesting level, and the type and value returned by the method. Values of literals, string and elementary types are provided explicitly, but those of objects are provided by references. The traces range in length from 17 000 lines to 143 298 lines.

We segmented each trace using both Pirzadeh's 2-factor approach (similarity and continuation) as well as the TRIADE approach described earlier (which uses parameters a third factor). We then generated summaries of each segment using both TF-IDF and Helmholtz. Thus, for each segment we generated three of four summaries (depending on the number of user actions used to exercise the system and generate the traces). We performed utility elimination in the manner described in the previous section, and whenever a summary comprised than 10 methods, we elided it by considering only the top 10 highest ranked methods. A different number can be used. We decided to go with 10 top methods after analyzing the content of the extracted summaries. In practice, we expect software maintainers to try different thresholds until reasonably well-defined summaries are obtained.

As can be seen in this section, we posit that our approach always generates a summary from which the behavior of the underlying segment can be determined accurately, by simply examining the methods present in the summary and consulting the program's documentation to disregard any method that consists of a utility or is insufficiently specific to the program's behavior. We refer to the method whose presence in a summary contributes to the understanding of the underlying behavior of the program as a *behavior-specific* method.

As mentioned above, each segment summary also contains multiple utility methods calls. In fact, the utility method calls often outnumber the behavior-specific method calls(i.e, method

[1] http://www.jhotdraw.org
[2] http://argouml.tigris.org/

calls that implement key functionality). However, this is not a hindrance to the trace comprehension process. The reason is that utility method calls can simply be ignored. As long as all behavior-specific method calls present in a summary cluster around a single higher-level process, the underlying behavior of the trace could be gleaned with ease. The analysis only fails in two cases: when *every* method in the summary is a utility, or if the summary erroneously contains behavior-specific methods unrelated to its actual behavior (i.e. methods related to the behavior present in another, usually adjacent trace segment). We refer to such cases as *misassigned* methods. As will be shown in this section, these cases only occurred in our experiment when using the 2-factor approach proposed by Pirzadeh et al.

We also found that the 3-factor approach consistently outperforms the 2-factor approach presented by Pirzadeh et al. [18], [20] in the following two ways: more of the 10 methods present in the summaries are behavior-specific rather than utilities, and furthermore, in our test sets, the 3-factor approach never misassigned a behavior-specific method in the wrong segment, which happened occasionally when using the 2-factor approach.

Finally, our results indicate that by the same metrics, the Helmholtz method of summarization outperforms TF-IDF, regardless of the method used to perform trace segmentation. The remainder of this section details our results.

A. JHotDraw

JHotDraw is an open source Java program for creating graphics. We generated a trace of JHotDraw using a 4 phase scenario (Launch JHotDraw, draw a line, draw a Rectangle and close JHotDraw).

a) Phase 1: The first phase consists in the initialization of the JHotDraw GUI (menus, toolbar, status bar, and main panes). Figure I presents a detailed listing of the methods present in each summary. It lists every method that appears in at least one summary, and indicates which summaries contain it on the right. Behavior-specific methods are in **bold** and red. We omitted the detailed summaries of the other phases out of space consideration. Several methods are from the `org.jhotdraw.draw.action`package and deal with the creation of the GUI and their presence in the summary makes underlying behavior of the trace immediately obvious. These include the following:

```
org.jhotdraw.draw.action.JPopupButton.add
org.jhotdraw.draw.action.
                  PaletteMenuItemUI.installDefaults
org.jhotdraw.draw.action.JPopupButton.add
org.jhotdraw.draw.action.JPopupButton.updateFont
```

Using the 3-factor method, the first 3 of these methods appear in both summaries of the first segment, and the latter 2 appear in the Helmholtz summary only. Every other method present in the summary is a utility and can be discarded as such by an analyst. Hence the underlying behavior of the trace-segment is immediately obvious after only a cursory reading of the documentation.

Several of these methods seem to share common parameter values with each other, as well as with other method calls related to the initialization of the GUI. It follows that when segmenting using the 3-factor method, these methods form a tight cluster and correctly appear in the TF-IDF and Helmholtz summaries of phase 1. However, it is interesting to note that the `*.installDefaults` method mentioned above does not share a parameter or return value with other phase 1 methods. The benefits of including parameters in the segmentation process thus extend to other methods, since the gravitational pull between two methods will have the effect of compressing together all intervening methods.

However, when segmentation is performed using the 2-factor approach, not a single one of the ten methods present in the Helmholtz summary is related to the underlying behavior of the trace (start-up and initialization of the GUI) while the TF-IDF summary contains only 2 such methods. The 2-factor Helmholtz summary also contains 2 misassigned methods, namely methods used by JHotDraw to draw a line on the screen. These methods should actually be in the summary of the subsequent phase.

b) phase 2: The second phase deals with the drawing of a single line. Most of the relevant methods are present in the classes: `org.jhotdraw.geom.BezierPath` and `org.jhotdraw.draw.BezierFigure`. In this case, the summaries generated using Helmholtz was particularly informative, and contained the following behavior-specific methods (and several others from the same class):

```
org.jhotdraw.draw.BezierFigure.basicSetEndPoint
org.jhotdraw.draw.BezierFigure.basicSetStartPoint
org.jhotdraw.draw.BezierFigure.basicSetBounds
org.jhotdraw.draw.BezierFigure.basicSetPoint
```

Several of these methods were absent from the TF-IDF summaries. This illustrates the effectiveness of the Helmholtz algorithm against the TF-IDF algorithm. In total, 8 of the 10 methods present in the 3-factor Helmholtz summary are behavior-specific, while the comparable figure is only 4 for the 2-factor Helmholtz summary. Uniquely in all the experiments performed, the 2-factor TF-IDF summary outperformed the 3-factor TF-IDF summary.

c) phase 3: The results for phase 3 mirror those of phase 2. The third phase is concerned with the drawing of a rectangle. Once again, the summary generated using Helmholtz to summarize the 3-factor trace yields a summary from which the higher-level behavior of the program is immediately evident. In this case, the summary contains the following methods, which are highly specific to the task at hand:

```
org.jhotdraw.draw.RectangleFigure.basicSetBounds
org.jhotdraw.draw.RectangleFigure.getFigureDrawBounds
org.jhotdraw.draw.RectangleFigure.drawFill
org.jhotdraw.draw.RectangleFigure.drawStroke
org.jhotdraw.draw.AbstractHandle.drawRectangle
```

Most of these methods, however, are only present in the summary generated using Helmholtz, and only when the trace is generated using the 3-factor approach.More particularly, each of the two summaries generated from the 2-factor method only contains a single behavior-relevant method. As a consequence, the behavior of the program becomes difficult to understand. In total, 5 of the 10 methods present in the 3-factor Helmholtz summary, and 3 of the 10 methods present in the 3-factor TF-IDF summary are behavior-specific, and are related to the

Table I: Detailed results for phase 1 of JHotDraw

Method Name	3-factors Helmholtz	3-factors TF-IDF	2-factors Helmholt	2-factors TF-IDF
org.jhotdraw.draw.action.ToolBarButtonFactory$7.compare	x	x		x
org.jhotdraw.draw.action.AbstractSelectedAction.setEditor	x	x		x
org.jhotdraw.draw.action.JPopupButton.add	x	x		x
org.jhotdraw.draw.action.JPopupButton.getColumnCount	x	x		x
org.jhotdraw.draw.action.PaletteMenuItemUI.installDefaults	x	x		
org.jhotdraw.draw.action.Colors.shadow	x	x		
org.jhotdraw.util.ResourceBundleUtil.configureAction	x	x		
org.jhotdraw.draw.action.JPopupButton.updateFont	x			
org.jhotdraw.util.ResourceBundleUtil.getLAFBundle	x			x
org.jhotdraw.util.ResourceBundleUtil.configureToolBarButton	x			
org.jhotdraw.draw.action.AbstractSelectedAction$EventHandler.propertyChange		x	x	x
org.jhotdraw.draw.DefaultDrawingView.getSelectionCount		x		
org.jhotdraw.draw.action.AbstractSelectedAction.access$0		x		x
org.jhotdraw.app.action.AbstractApplicationAction.updateApplicationEnabled			x	
org.jhotdraw.app.action.RedoAction.updateProject			x	
org.jhotdraw.util.ResourceBundleUtil.getImageIcon			x	
org.jhotdraw.draw.action.ToolBarButtonFactory.addToolTo			x	
org.jhotdraw.app.action.UndoAction.updateProject			x	
org.jhotdraw.draw.ArrowTip.getDecoratorPath			x	
org.jhotdraw.geom.BezierPath$Node.getControlPoint			x	
org.jhotdraw.app.action.AbstractApplicationAction.createApplicationListener			x	
org.jhotdraw.beans.AbstractBean.addPropertyChangeListener				x
org.jhotdraw.draw.DefaultDrawingView.addFigureSelectionListener				x
org.jhotdraw.draw.action.JPopupButton.getPopupMenu				x

behavior of the underlying trace segment. By contrast, the 2-factor Helmholtz and TF-IDF summaries contain a single behavior-specific method each. They also contain methods that seem to belong to the previous segment.

From the program's documentation, it is easy to see the reason why the results of the 3-factor approach are so much better than those of the 2-factor approach. Indeed, several of the methods used to draw a figure manipulate the figure's coordinates. The presence of these values as parameters and return values yields a tight grouping between the methods that manipulate the same object.

d) phase 4: The final phase presents even starker results. In this phase, the program backups the project properties (the size and position of the drawn shapes) in the project properties file. It then saves the project if requested to do so by the user, and shuts down. The backup process involves methods that manipulate XML objects, since this is the format used to record the information. The presence of methods that write XML objects, such as the following, is thus a clear indicator of the behavior of the segment.

The Helmholtz summary of the 3-factor segment performs particularly well in this respect, and 7 of the 10 methods present in this summary are related to the underlying behavior, making it extremely easy for a system engineer to glean understanding from the trace. Only 2 phase relevant methods appear in the 3-factor summary generated with TF-IDF and neither one of the two summaries generated from the 2-factor approach contain any behavior-relevant methods. Worse than this, several methods related to the behavior of the preceding execution phase, such as methods related to drawing a rectangle, are present in the 2-factor summaries of this phase, which may lead a user to inaccurate conclusions about the program's behavior during this segment.

B. ArgoUML

As a second example, we generated a trace of a 4 phase scenario using ArgoUML, a diagramming application written in Java. The phases consist in launching ArgoUML drawing a use case diagram, drawing a class diagram and closing ArgoUML.

a) Phase 1: The first phase consists in launching the application, and in the initialization of the ArgoUML GUI (menus, toolbar, status bar, and main panes). The results obtained in this phase are distinct from our other results in that there was minimal overlap between the methods found to be most significative in the 2-factor and 3-factor approach summaries, with only 3 methods appearing in both a 2-factor and a 3-factor summaries. Furthermore, the two summaries generated using from the 2-factor approach were also completely distinct, which breeds a suspicion about the accuracy of the summaries generated.

The two summaries generated using the 3-factor approach, however, are largely similar, sharing 7 methods. Both of these summaries contain two distinctive methods from which the underlying behavior (application start-up) of the trace can be gleaned almost immediately:

```
org.argouml.application.Main$1.i18nmessageFormat
org.argouml.application.helpers.ApplicationVersion.
   getStableVersion
```

The packages `org.argouml.application` and `org.argouml.application. helpers` contain the main class and starting point of the ArgoUML application and provides "helper" classes that make available basic functionality for the application respectively. The method `getStableVersion` receives the version of the application at initialisation time. Likewise, the method `i18nmessageFormat` provides proper translation to text strings that must be customized to the user's language—a task performed at startup. Every other method present in all four summary seems to be a utility, except for a misassigned behavior-specific method wrongly placed in the 2-factor Helmholtz summary.

b) Phase 2: The second phase consists in drawing a use case diagram. In this case, our experiment provided evidence of the superior effectiveness of Helmholtz over TF-IDF. As mentioned earlier, each of the summaries was narrowed to 10 methods, by considering only the most relevant entries. From those, the 3-factor Helmholtz summary contains 2 methods from the packages `org.argouml.uml.diagram. use_case.ui`. These two methods suffice to understand the underlying behavior of the trace since none of the other methods present in the summaries are behavior-specific, and can thus be ignored when trying to glean the behavior of the program from its execution trace. As their names indicate, these packages contain the classes that implement a UML Use Case diagram using the UCI Graph Editing Framework (GEF). Neither of the two TD-IDF summary contained any behavior-specific method. The 2-factor Helmholtz summary did contain a single method from `org.argouml.uml.diagram.use_case.ui`. However, it also contained a misassigned behavior-specific method, which actually refers to the behavior of the subsequent program phase. As a result, a determination of the behavior of the underlying trace would be next to impossible with any analysis other than the 3-factor approach proposed in this paper.

c) Phase 3: The third phase consists in drawing a class diagram. As can be shown, the results of the summaries's analysis in this phase emphasize the effectiveness of the 3-factor approach over the 2-factor approach.

Every single method present in the Helmholtz summary of the 3-factor approach is relevant to understanding the underlying behavior of the trace segment. Likewise, all but 1 of the 10 methods present in the TF-ITF summary are similarly meaningful. Some examples of methods present in one or both of these two summaries include:

```
org.argouml.uml.diagram.static_structure.
   ui.FigClassifierBox.updateListeners
org.argouml.uml.diagram.static_structure.
   ui.FigClassifierBox.updateCompartment
org.argouml.uml.diagram.static_structure.
   ui.FigClass.updateNameText
```

These methods belong to the package `org.argouml.uml.diagram. static_structure.ui` which in turn contains classes that implement a class diagram. Also present are several methods from the package `org.argouml.uml. diagram.ui.*` . This package provides various support for diagrams: actions, GEF Figs, Go rules, Property Panels for diagrams, GEF Selection support, etc.

The TF-IDF summary of the 2-factor approach also performs well, with 9 out 10 methods present in the summary being from one of the two packages mentioned above. However, only 1 of the 10 methods present in the Helmholtz 2-factor summary is behavior-specific.

d) Phase 4: In the fourth phase, the program saves the project setting (called persisting) and shuts down. Once again, multiple methods that make the behavior of the trace segment evident are present in the summaries. In particular, the three following methods,

```
org.argouml.persistence.UmlFilePersister.hasAnIcon
org.argouml.persistence.PgmlUtility.getEnclosingId
org.argouml.persistence.AbstractFilePersister.
   getExtension
```

The package `org.argouml.persistence` contains support for saving projects to media. Using our 3-factor method, these methods correctly appear in the summary generated using both Helmholtz and TF-IDF. Neither one of the two 2-factor summaries contains any method from this package, or any other method which could be used to understand the underlying behavior of the trace.

IV. Related Work

A survey of studies that aim to identify the most important elements of a trace is presented in Dit et al. [4]. However, most of the techniques reported lack a mechanism to segment the trace into execution phases, and to extract key elements from each phase.

Our research is closely related to that of Pirzadeh et al. [20]. These authors proposed an approach in which they divided a trace into a set of segments that correspond to the main actions done by the user, by relying upon method names as well as on the nesting depth of methods to form a set of dense groups of trace elements. One of the contributions of this paper is to extend the algorithm of Pirzadeh to a 3-factor rather than 2-factor algorithm, yielding a higher precision.

There exists other trace segmentation approaches. Watanabe et al. [22] proposed an approach based on the nature of object-oriented programming. They use LRU cache to detect the beginning and end of phases. A change in the frequency of the cache update that means that a phase transition is occurring. In [9], Kuhn et al. introduce an approach derived from signal processing, representing entire traces as signals in time. They visualized traces as time plots, and then presented a technique for summarization.

Medini et al. [15] proposed a concept location technique that relies on trace segments. Their approach splits execution traces into segments using conceptual cohesion and coupling measured by comparing the body of methods using the cosine measure. The trace segmentation approach presented by the authors is reformulated as a dynamic programming problem. While they used static analysis, which makes their approach a heavy weight one, we use dynamic analysis to automatically generate distinct execution phases from execution traces. Also, a user needs to define various thresholds to decide on how to prune the execution trace before applying the dynamic programming algorithm. Medini et al. [14], [15] proposed SCAN (Segment Concept AssigNer) an approach to assign labels to sequences

Table II: Summary of results.

	3-factor						2-factor					
	Helmhotlz			TF-IDF			Helmhotlz			TF-IDF		
	S	A	M	S	A	M	S	A	M	S	A	M
JHotdraw phase 1	10	5	0	10	3	0	9	0	2	10	0	0
JHotdraw phase 2	10	8	0	10	1	0	10	9	0	10	5	0
JHotdraw phase 3	10	5	0	10	3	0	10	1	1	10	1	2
JHotdraw phase 4	10	7	0	10	2	0	10	0	1	10	0	2
ArgoUML phase 1	10	2	0	10	2	0	10	0	1	10	0	0
ArgoUML phase 2	10	2	0	10	0	0	10	1	1	10	0	1
ArgoUML phase 3	10	10	0	10	9	0	10	1	0	10	9	0
ArgoUML phase 4	10	3	0	10	2	0	3	0	0	10	0	0

of methods in execution traces and to identify relations among execution traces segments. SCAN can be easily adapted to TraceSegmenter to label and relate the extracted phases.

Poshyvanyk et al. [21] introduced a technique based on information-retrieval for feature location using PROMESIR. Greevy et al. [5] introduced a feature-driven approach to trace segmentation. They extracted execution traces to achieve a mapping between features and classes and then analyzed the changes of roles of the classes while the features of system evolve.

V. Conclusion

In this study, we proposed a techniques that use three factors for segmenting large execution trace, based on the trace segmentation process presented by Pirzadeh et Hamou-Lhadj [18], into its constituent execution phases, and extracted from each phase a summary of human-tractable size composed of its key events. We show that using method parameters improves the segmentation process, and the summaries produced contain more behavior-relevant methods and fewer misplaced methods, regardless of the method used to perform key element extraction. We further compare two key element extraction algorithms and show that Helmholtz typically outperforms TF-IDF.

References

[1] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 49–58, June 2007.

[2] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[3] B. Dadachev, A. Balinsky, H. Balinsky, and S. Simske. On the helmholtz principle for data mining. pages 99–102, Sept 2012.

[4] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. 25, 01 2013.

[5] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 314–323, March 2005.

[6] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *14th International Conference on Program Comprehension, Proceedings*, pages 181–190, 2006.

[7] T. Joachims. Text categorization with support vector machines: Learning with many relevant features, 1998.

[8] R. Khoury, L. Shi, and A. Hamou-Lhadj. Elements extraction and traces comprehension using gestalt theory and the helmholtz principle. pages 478–482, Oct 2016.

[9] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 320–329, Sept 2006.

[10] R. Laddad. *AspectJ in Action: Practical Aspect-oriented Programming*. In Action Series. Manning, 2003.

[11] S. Lehar. *The world in your head : a gestalt view of the mechanism of conscious experience*. In Action Series. Mahwah, N.J. : Lawrence Erlbaum Associates, 2003.

[12] D. Lowe. *Perceptual Organization and Visual Recognition*. The Springer International Series in Engineering and Computer Science. Springer US, 1985.

[13] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[14] S. Medini, V. Arnaoudova, M. D. Penta, G. Antoniol, Y. Guéhéneuc, and P. Tonella. SCAN: an approach to label and relate execution trace segments. *of Software: Evolution and Process*, 26(11):962–995, 2014.

[15] S. Medini, P. Galinier, M. D. P. Y.-G. Guéhéneuc, and G. Antoniol. A fast and scalable algorithm to locate concepts in execution traces. In *3d International Symposium on Search Based Software Engineering, Proceedings*, pages 252–266, 2011.

[16] D. Pelleg and A. Moore. X-means: Extending k-means with efficient estimation of the number of cluster. In *7th International Conference on Machine Learning*, pages 727–734, 2000.

[17] H. Pirzadeh, L. Alawneh, and A. Hamou-Lhadj. Quality of the source code for design and architecture recovery techniques: Utilities are the problem. In *2009 9th International Conference on Quality Software (QSIC 2009)(QSIC)*, volume 00, pages 465–469, 08 2009.

[18] H. Pirzadeh and A. Hamou-Lhadj. A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension. In *16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 221–230, 2011.

[19] H. Pirzadeh and A. Hamou-Lhadj. A software behaviour analysis framework based on the human perception system. In *33rd International Conference on Software Engineering (ICSE'12 NIER Track)*, pages 948 – 951, 2011.

[20] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah. Exploiting text mining techniques in the analysis of execution traces. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 223–232, 2011.

[21] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.

[22] Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. pages 8–14, 01 2008.