A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension

Heidar Pirzadeh and Abdelwahab Hamou-Lhadj
Software Behaviour Analysis Lab
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve Blvd. West
Montreal, QC, Canada H3G 1M8
{s_pirzad, abdelw}@ece.concordia.ca

Abstract—**The analysis of execution traces can reveal important information about the behavioral aspects of complex software systems, hence reducing the time and effort it takes to understand and maintain them. Traces, however, tend to be considerably large which hinders their effective analysis. Existing traces analysis tools rely on some sort of visualization techniques to help software engineers make sense of trace content. Many of these techniques have been studied and found to be limited in many ways. In this paper, we present a novel trace analysis technique that automatically divides the content of a large trace into meaningful segments that correspond to the program's main execution phases such as initializing variables, performing a specific computation, etc. These phases can simplify significantly the exploration of large traces by allowing software engineers to first understand the content of a trace at a high-level before they decide to dig into the details. Our phase detection method is inspired by Gestalt laws that characterize the proximity, similarity, and continuity of the elements of a data space. We model these concepts in the context of execution traces and show how they can be used as gravitational forces that yield the formation of dense groups of trace elements, which indicate candidate phases. We applied our approach to two software systems. The results are very promising.**

Keywords: Trace Analysis, Program Comprehension, Software Maintenance, Software Engineering

I. INTRODUCTION

A common and difficult problem experienced by software engineers when maintaining large complex system is to understand how the system is built and why it is built that way [4]. This is especially important when the system suffers from poor documentation (if it exists at all) and that the original designers have moved to new projects or company.

In this paper, we focus on techniques that assist the understanding of the behavioral aspects of a software system. These techniques often rely on tracing and run-time monitoring mechanisms. Traces, however, are difficult to work with since they tend to be considerably large. To address this issue, many trace abstraction and simplification techniques have been proposed with a common objective being to extract high-level views from raw traces (e.g. [1, 2, 3]). Although these techniques have been shown to be useful in the context of software maintenance, they suffer from several limitations such as their extensive reliance on user intervention, their dependence on particular visualization schemes that hinder their reuse, and so on [25]. The general consensus in the trace analysis community is that more work towards effective trace abstraction techniques is much needed.

The objective of this paper is to present a novel trace analysis technique that automatically divides the content of a trace into smaller and meaningful trace segments that correspond to the program's main execution phases such as initializing variables, performing a specific computation, etc. These phases can significantly simplify the exploration of large traces by allowing software engineers to browse the trace by focusing on its major parts (i.e. its execution phases) instead of a flow of mere low-level events.

Our phase detection approach is inspired by Gestalt laws of *similarity* and *good continuation* [9, 24, 27], a concept used in psychology to describe the operational principle of the human brain, particularly, the ability for humans to visually recognize objects and shapes 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. We apply these laws to traces of method (routine) calls (common traces used in program comprehension) to form dense groups of trace elements that indicate the presence of potential execution phases.

Organization of the paper: In the next section, we define what we mean by an execution phase. In Section III, we describe our phase detection approach. Two case studies are presented in Section IV, followed by related work. Finally, we conclude the paper and point out to future directions in Section V.

II. What is an Execution Phase?

Execution phases can appear at various levels of a program execution [5, 8, 12]. At the highest level of abstraction, a program execution can be considered as an algorithm or a general procedure for solving a specific problem. The phases in this case are the key steps of the algorithm. At a lower level of abstraction the execution phases of a program written in a specific programming language are portions of the program execution that collaborate with each other to implement a specific task ([5, 6]). The lowest level of abstraction of a program is presented as machine code. The execution phases in this level can show how the program accesses and uses system resources and other low-level tasks. For example, the phases can show distinctive patterns of hardware usage (e.g., CPU usage, memory access, communication ports access) or stable states of machine resources during the execution as noted by Sherwood et al. [8].

In this paper, we focus on identifying the key execution phases that compose a program at the source code level. Similar to other works (e.g., [5, 6, 12]) in the domain of trace analysis for program comprehension we use the following definition of execution phases: "a program [execution] phase is a portion of the program that exhibits common behavior at a level the programmer would recognize" [5]. In such context, we want to be able to take an execution trace (generated from exercising one or more particular features) and identify a set of high-level tasks (i.e. execution phases) where each task performs a portion of the overall work. Each phase denotes an *essential step* of the general procedure and phase transitions can denote the *logic* of the procedure or the flow of data from one step of the procedure to another. This way, the understanding of how a feature is implemented will no longer require, at least in the beginning, that its corresponding trace be explored as a flow of mere low-level program elements. Instead, a trace can be seen as a sequence of execution phases, in which a phase denotes an essential step of the general execution and the transitions among phases depict the logic (or the flow) that connects the phases. For instance, a program's execution trace could be composed of three major phases (**Error! Reference source not found.**): The initialization phase, the computation phase, and the finalization phase. Each phase can be further decomposed into smaller sub-phases that implement specific sub-tasks of the program.

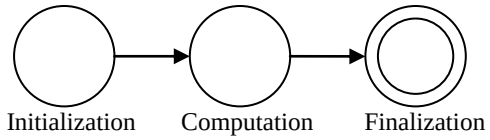


Figure 1. High-level phases of a program

The benefits of extracting the execution phases of a program in the area of program comprehension are numerous:

- Phases can provide important information on how a particular feature is implemented, which in turn can help software maintainers enhance these features.
- Phases can be further refined to recover a high-level behavioral view from raw traces, enabling the understanding of the traced scenario.
- Phases might be helpful in fault localization as they can show in what phase of the program execution the error has occurred.
- A program shown as phases can be an excellent means of communication between maintainers, developers, programmers to obtain a quick and clear idea and description of the program.

III. Phase Detection Approach

We have developed a phase detection technique that is inspired by the law of gravity (namely similarity and continuity), also known as Gestalt laws of perception, which describe how people group similar items visually based on their perception [21, 22, 27]. Gestalt psychology is an application of physics to essential parts of brain physiology by telling the physiologist what kind of process occurs 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) [27].

For example, Figure 2 shows a very simple trace composed of two routines `a` and `b` invoked several times. The figure also shows a ruler that is used to indicate the position of the calls in the trace (e.g. the first call to `a` appears in position 1, the second call in position 2, etc.). If asked to identify the major phases that appear in this trace, a programmer would most likely perceive two major phases: The first one is composed of the calls to `a`, while the second phase could consist of the calls to `b`. Gestalt psychology explains that the more similar two elements are to each other the more likely they are to be perceived as belonging to the same group [9].

Figure 2. A sample trace

As the trace grows in size and complexity, the programmer's visual perception of similarity becomes more difficult. The complexity here can be defined as the number of new elements that are invoked in a trace. Figure 3 shows an overview of our approach for automatic detection of execution phases from traces. We start first by applying gravitational schemes, defined based on Gestalt laws, on the input trace. Two schemes (more precisely the similarity and continuity schemes) are used as gravitational forces that yield the formation of dense groups of trace elements, which indicate the candidate phases. When the dense groups are formed, we automatically identify the beginning and end of each phase using K-means clustering with BIC (Bayesian Information

Criterion) support [13] (discussed in more details later in the paper).

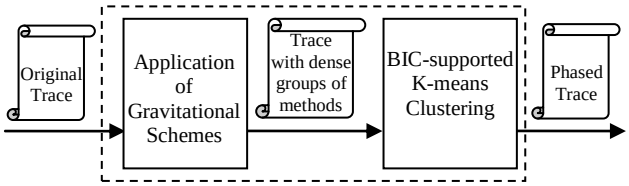


Figure 3. Detailed view of the execution phase detection unit

The two gravitational schemes that we have developed in this paper are based on the fact that a phase change in an execution trace corresponds to a significant change in the pattern of some attributes of the method calls in the trace over time. Therefore, our strategy is to reduce the distances between the method calls for which the characteristics can form a pattern specifying a phase. The effect of applying each of the two schemes (i.e. similarity and continuity schemes) is as follows:

Similarity scheme: By applying this scheme the method calls in the trace are repositioned in a way that the distance between the same method calls is reduced.

Continuity scheme: The application of this scheme results in the repositioning of the method calls in a way that the methods that are continuously called (and not returned as much) are made closer one to another to emphasize a trend in the execution of the program.

To help with the description of these techniques, we introduce the following definitions:

We define a trace *T* of method calls of size *n* (the number of calls invoked in the trace) as a sequence of one or more method calls, where each method call is denoted as $c_{i,d}$ where *i* represents the invocation order of the method call *c* and *d* 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 be able to apply our gravitational schemes we need to define distance between the method calls in a trace. For this, 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 $|j-i|$. This maps the ordinal scale of method calls to an interval scale. Furthermore, we define the function $Pos(c_{i,d})$ to return the position of the method call *c* in the interval scale (i.e., on a ruler). The position of a method call is also the order in which it was invoked right after the trace is generated. However, as the method calls are rearranged as a result of applying the two schemes, the new position of a method call might differ from its original order of invocation (rearrangement of the trace does not preserve the order of calls).

A. The Similarity Gravity Scheme

The objective of the similarity gravity scheme is to reposition the elements of a trace in such a way that similar elements gravitate to each other forming a group of dense elements, which could indicate the presence of a phase. In other words, the elements of a trace are re-positioned in a way that the distance between two same elements is less than the distance between two different elements given that the difference in terms of the invocation order is the same for the elements of both pairs.

A simple repositioning scheme based on the similarity gravity technique, 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:

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

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

As stated above, in the similarity gravity technique we visit each method call $c_{i,d}$ in the original trace; if there is a previous method call $c_{j,d'}$ to the same method, we reposition $c_{i,d}$ to half way from $c_{j,d'}$ (i.e., by reducing the distance to half). Otherwise we do not change its position ($c_{i,d}$ remains in *i*-th position). We chose to reduce the distance between calls to the same method by half, although one could use a different measure. The focus here is on the fact that the same methods are placed close enough to each other to form a dense group.

Figure 4 shows the result of applying the similarity gravity method to the sample trace of Figure 2. As we can see, the new positioning of the elements leads to two blocks that could indicate the presence of two phases. The first phase begins at the first method invocation (and contains calls to `a`) and the second phase starts at the fifth method invocation (calls to `b`). That is, after using the similarity gravity method, even if the similarity of the items in each of each group becomes imperceptible, the group still can be recognized by their structure and the distance between them. This is shown in Figure 6 where we replaced all methods with "●".

Figure 4. The result of applying the similarity gravity to a trace of Figure 3

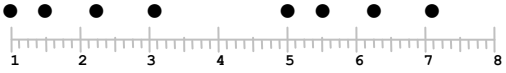


Figure 5. The result of applying the similarity gravity technique to a trace of Figure 3 where the routines are shown as dots

In Figure 6, we show the effect of applying the similarity gravity method to a sample trace. By applying similarity gravity scheme in Figure 6 (step 2) we find that the trace contains two phases. The first one starts at the first invocation and is composed of calls to routines `a` and `b`, while the other one which starts at the seventh invocation contains calls to `c` and `d`. Figure 6 (step 3) shows the same result when discarding the effect of similarity in perception (replacing all methods with "●".). Although the size and complexity have increased compared to the sample example of Figure 2, one can still quickly recognize two phases based on the formed groups.

Structurally recognizable groups can also be explained by Gestalt laws. Proximity, the most fundamental law of Gestalt laws, states that "being all other factors equal, the closer [in terms of distance] two elements are to each other the more likely they are to be perceived as belonging to the same group" [24]. This way, one may conclude that the similarity gravity technically converts similarity to proximity.

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

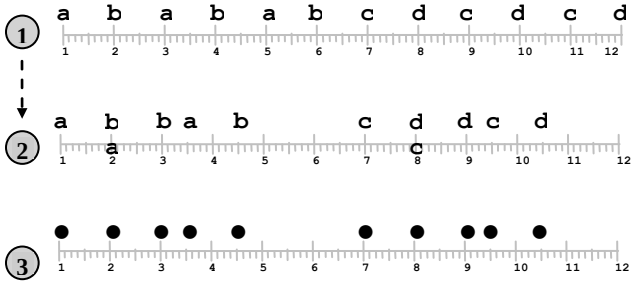


Figure 6. The result of applying the similarity gravity technique to a sample trace

B. The Continuity Gravity Scheme

The previous method should work well for traces where phases can be perceived by an analyst by looking at the linear representation of a trace, although some automatic assistance is needed. But what happens if there is no perceivable similarity between the linear representation of the elements of an execution trace? For example, Figure 7 (step 1) shows an execution trace with no visual similarity between its method calls (no unique method is invoked more than once).

To overcome this issue, we turn again to another one of Gestalt laws, the Law of Good Continuation [10], which states the tendency of things to group if they are visually not co-linear or nearly co-linear. In execution traces, the increments in nesting levels of the method calls are continuous. For example, in Figure 7 (step 1), one can notice that there is a good continuation between the calls from `a` to `○`, 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 [17, 12]. Watanabe et al. [12] 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) becomes local-minimum at the beginning of a phase indicating a phase transition. They also showed that the elements that have a high nesting level (i.e. which are deep in the tree hierarchy) were unlikely to initiate new phases.

We define a scheme called the continuity gravity that groups trace elements based on the nesting level of the method calls. The continuity gravity scheme groups trace elements by keeping the method calls with higher nesting level closer to the previous method calls. The higher is the nesting level of a method call, the stronger it is attracted by the previous method call.

A continuity gravity scheme that repositions the elements of a trace based on their nesting level, and that we call here Pos_{cont}, is as follows:

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

When applied to a trace, this scheme reduces the distance between method calls based on the nesting level (d) of the callee by changing the distance of two consequent 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 it could indicate a phase shift. Again, we chose here not to reposition subsequent subtrees where the nesting levels vary by more than half. A different criterion could be used as long as a drastic change among subtrees can be identified.

Figure 7 (step 2) shows the result of applying the continuity gravity scheme to the sample trace. As we can see, the new positioning of the elements leads to two distinguishable phases (the phases are more distinguishable when we omit to visualize the nesting levels). The first phase begins at the first method invocation and the second phase starts at the tenth method invocation. This way, we used the effect of continuity in perceptual grouping to build groups that are structurally recognizable. That is, after using the continuity gravity, even if the continuity of the items in each group becomes imperceptible due to the size and complexity of a trace, the trace clusters still can be recognized through their structure and the distance between them. If we omit to visualize the nesting level (see Figure 7 (step 3)) and replace the methods with a "●" (Figure 8), we can clearly see that two phase have been formed. We may say that the continuity gravity technically converts continuation to proximity.

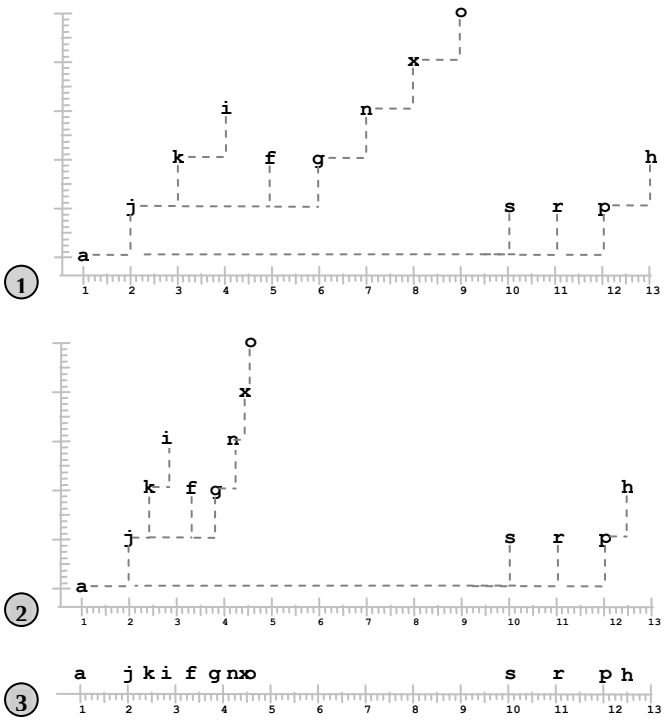


Figure 7. The result of applying the continuity gravity technique to a sample trace.

Figure 8. The resulting trace with the routines replaced with dots

C. Integration of Gravities

An efficient phase detection technique should integrate both the continuity and similarity gravity. First, the distance between the current method call and the previous method call is updated based on the changes in the nesting level (application of continuity scheme). Next, the position is updated according to the similarity of the current method call to previous method calls (application of similarity scheme). We call this the integrated gravity scheme. When applied to a trace, the integrated scheme first reduces the distance between method calls based on their nesting level, then, reduces the distance between method calls to a same method.

The integrated gravity repositioning scheme can be iteratively applied to a trace to detect major phases, their sub-phases, etc, until we reach the individual elements of the trace. For this, we need to have a means to harness the gravity so that phases could be detected with different levels of granularity. A threshold *t* is defined 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. This way, 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 and that a tool that supports our approach should allow enough flexibility to vary the threshold.

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

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

D. Identifying the Beginning and End of Phases

Once the method calls of a trace are repositioned according to the integrated gravity scheme, we need a way to automatically identify the beginning and end of each phase since it would be impractical to expect from programmers to distinguish the various phases visually for considerably large traces. For this, we propose to use a clustering algorithm that can group the method calls of the repositioned trace into clusters of method calls that are close to each other. We chose K-means clustering as our clustering algorithm [22]. K-means is an unsupervised clustering technique that partitions the data points (instances) into a predetermined number (*K*) of non-hierarchical clusters.

The number of clusters is pre-specified by randomly drawing *K* points as initial centroids ($\mu_1 \cdots \mu_K$) (showing the center of each cluster). The rest of the algorithm is iteratively performed according to the below two steps, trying to minimize the overall sum of distances of the points from their cluster centroids:

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

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

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

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

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

where $R_i = |D_i|$.

The iteration continues until we have the same cluster assignment in two successive iterations.

In K-means clustering the number of clusters (i.e., the number of phases) should be given as an input to the

algorithm. In other words, the user must know the number of phases before running the K-means algorithm (perhaps by counting the number of distinct phases that he can visually perceive on the plot). This, however, can be error prone. Therefore, it would be advantageous if the number of clusters could be selected automatically according to the complexity of the data. Pelleg and Moore [11] proposed an approach to find the best partitioning of the data where the average variance of the clusters is minimum. It is obvious that as the number of clusters increases the average variance of the clusters decreases (as *k* approaches the number of 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. This is done via the Bayesian Information Criterion (BIC) which is a model selection criterion [23]. In order to avoid the problem of overfitting the data, BIC is penalized based on the number of parameters in the model.

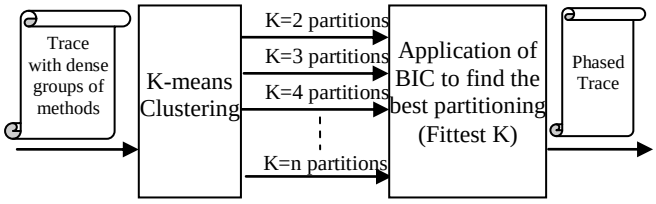


Figure 9. Detailed view of BIC-supported K-means clustering

As shown in Figure 9, 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 continuity schemes) for a set of different values of *K* which results in a set of alternative partitionings ($P_1 \cdots P_x$). To evaluate these partitionings, we compute the BIC score of each partitioning, the highest BIC means best available partitioning of the execution trace and consequently the best estimation of the number of clusters *K*, and which also corresponds to the number of identified phases Since the dimension of the data in our case is 1, we use a special formulation of the BIC (for a more general case of BIC formulation see [11]):

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

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

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

where $D_i \subseteq D$ is the set of points that have μ_i as their nearest centroid, and $R_i = |D_i|$

The BIC score provides us with the best partitioning of the execution trace elements according to its complexity. This way, we can specify the number of phases in our repositioned execution trace and locate the phases automatically.

IV. Case Study

In order to evaluate the effectiveness of our phase detection approach, we conducted two case studies where we applied our technique to two execution traces generated from two different systems. The first execution trace is generated from JHotDraw 5.2 [16], which is a framework implemented in Java for technical and structured graphics. It consists of 11 packages, 171 classes, and 1414 methods. JHotDraw 5.2 has 9419 lines of code. The second case study was conducted on an execution trace generated from ArgoUML 0.27, an open source UML modeling tool implemented in Java. It consists of 1853 classes, 10214 methods, and 130995 lines of code.

A. Case Study I

1) Scenario Description

For our first case study, we used an execution trace generated from JHotDraw by exercising a scenario that involves several major activities: Drawing three different figures (a rectangle, a round-rectangle, and an ellipse) followed by drawing the same three figures for the second time on the same sheet and closing the application.

Since JHotDraw registers all mouse movements, and mouse movements are required while drawing figures, the resulting trace was bound to contain a lot of noise. We have therefore filtered these mouse movements to obtain a trace that is cleaner. We are aware that the detection of noise in a trace might not always be straightforward and that noise detection techniques such as the ones presented by Hamou-Lhadj et al. in [17] might be needed. The resulting trace contained 4400 method invocations (8800 events), which is considered a small trace. It is used here as a proof of concept. We show how our approach works on a larger trace in the second case study.

2) Results of Applying the Phase Detection Approach

We first applied our approach to detect the major phases in the trace. This is achieved by setting the threshold to the size of the trace (as explained in Section III.C). Figure 11 (a) shows the result of applying the integrated gravity scheme on the trace. The result is shown in the form of a histogram, where the x-axis shows the distance between the positions of the calls and the y-axis represents the number of methods whose position falls into one interval of x axis. As we can see in, there are two dense groups of methods (DG1 and DG2) that have been formed and which indicate the possibility of the existence of two major execution phases. We explored the contents of the two phases and found that DG1 represents the initialization of variables (about 1500 invocations),

whereas DG2 contains the methods invoked in the trace to perform the core computations (i.e. drawing the figures).

We applied our technique to DG2 with a lower threshold so as to detect the sub-phases that it composes. Figure 11 (b) shows the results of applying the integrated gravity, using t=20, to the DG2 segment of the JHotDraw trace. As part of our technique, in order to automatically determine the number of phases and their location, the trace resulting from applying the integrated gravity scheme is partitioned by K-means clustering for K from 1 to 10. The BIC score for different partitionings of DG2 are shown in Figure 10. The highest BIC score was for the partitioning with K = 7 as the best fit. Figure 11 shows the location of the seven clusters (P1 to P7) that have been formed with K = 7.

We validated the results by referring to JHotDraw documentation and by manually analysing the methods invoked in each phase. Except for the last of these seven phases (P7), the six phases (P1 to P6) are similar in terms of length and density. After exploring the content of the trace, we found that P7 contains methods that end the application (finalization methods) including the following methods:

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

As for the phases P1 to P6, we found that each of these phases corresponded to the drawing of one figure. For example, the phase P1 contained methods involved in drawing a rectangle. P2 contained the methods responsible of drawing a circle, etc.

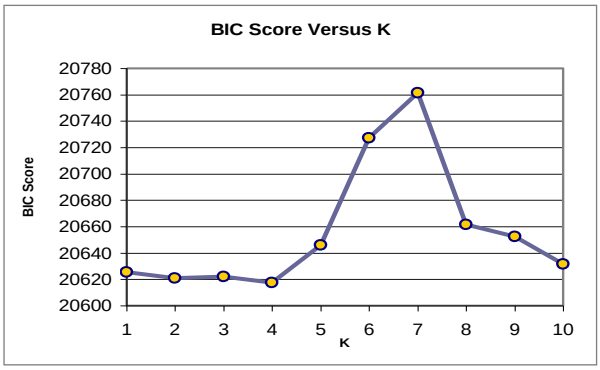
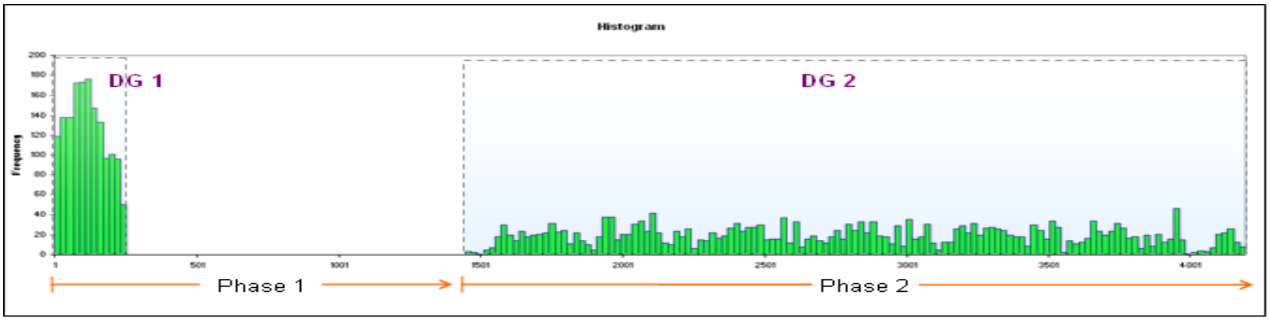


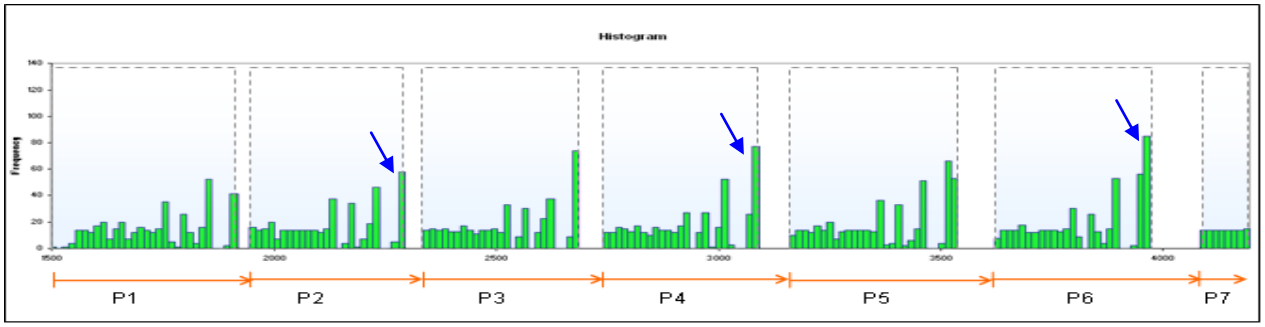
Figure 10. The BIC score for different partitionings of DG2, the partitioning with 7 clusters is the fittest

To further determine the sub-phases that compose phase P1, we re-applied the integrated gravity method on this phase with a lower threshold. This resulted in three sub-phases which contained methods for "selecting the rectangle button in the buttons menu", "preparation for creating and adding a rectangle to the sheet", and "drawing of a rectangle on the sheet". An example of methods involved in the third sub-phase of P1 is:

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



(a)



(b)

Figure 11. The result of applying the integrated gravity technique to detect major phases

We applied the same process to P2 to P6 and were able to confirm that each phase corresponded to the drawing to one of the figures and that all of them consisted of three other sub-phases. One interesting observation was that the length and the density of part of the phase which contains the methods that actually draw the figure on the sheet (indicated by blue arrows) grows from P1 to P6 while the first parts of all phases (P1-P6) exhibit similar distribution. This is due to the massive use of design patterns in JHotDraw where some features just differ for the invocation of a few methods. That is, drawing a circle is performed very similarly to drawing a rectangle. This explains the phase parts that are similar. It also justifies the fact that these phases formed a single major phase (phase 2) at a higher level of granularity. The reason for the growing part (indicated by blue arrows) is that every time we draw a figure, JHotDraw redraws the existing figures on the sheet.

B. Case Study II

1) Scenario Description

For the second case study we apply our technique to a trace generated from ArgoUML by exercising the following scenario: Starting up ArgoUML, drawing a class on the class diagram, and quitting ArgoUML). The resulting trace contained 35754 method calls (to 2331 different methods). Note that a method invocation requires at least two events to be collected, the entry and exit of a method. The trace size in terms of events is therefore about 71508 events, which is considered a relatively large trace.

2) Results of Applying the Phase Detection

Figure 12 (upper diagram) shows the result of applying the integrated gravity scheme on the ArgoUML trace. A clear division of the execution trace into five major phases was also supported by the BIC score with $K = 5$ as the best fit. When check against the documentation, as expected, the methods of the first phase indicate the initialization of ArgoUML where the main application frame (menus, toolbar, status bar, and main panes: navigation pane, multieditor pane, to do pane, and details pane) and project are set up. The project corresponds to a model that contains an empty class diagram, and an empty use case diagram.

The second detected phase is concerned with loading auxiliary modules from the input stream and adding them to the Post Load Actions list, which contains actions that are run after ArgoUML has started.

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

```
org.argouml.ui.cmd.ActionNotation.menuSelected
org.argouml.kernel.ProjectSettings.getNotationNam
e
org.argouml.notation.NotationNameImpl.getIcon
org.argouml.notation.NotationNameImpl.sameNotatio
As
org.argouml.ui.cmd.ActionNotation.menuDeselected
org.argouml.ui.cmd.ActionExit.actionPerformed
org.argouml.ui.ProjectBrowser.tryExit
org.argouml.ui.ProjectBrowser.saveScreenConfigura
tion
org.argouml.configuration.Configuration.save
```

We further applied our technique, with a lower threshold to Phase 3 (drawing a class) to understand how this is accomplished. Figure 12 (bottom diagram) shows the result of applying the integrated gravity, using t=20, to

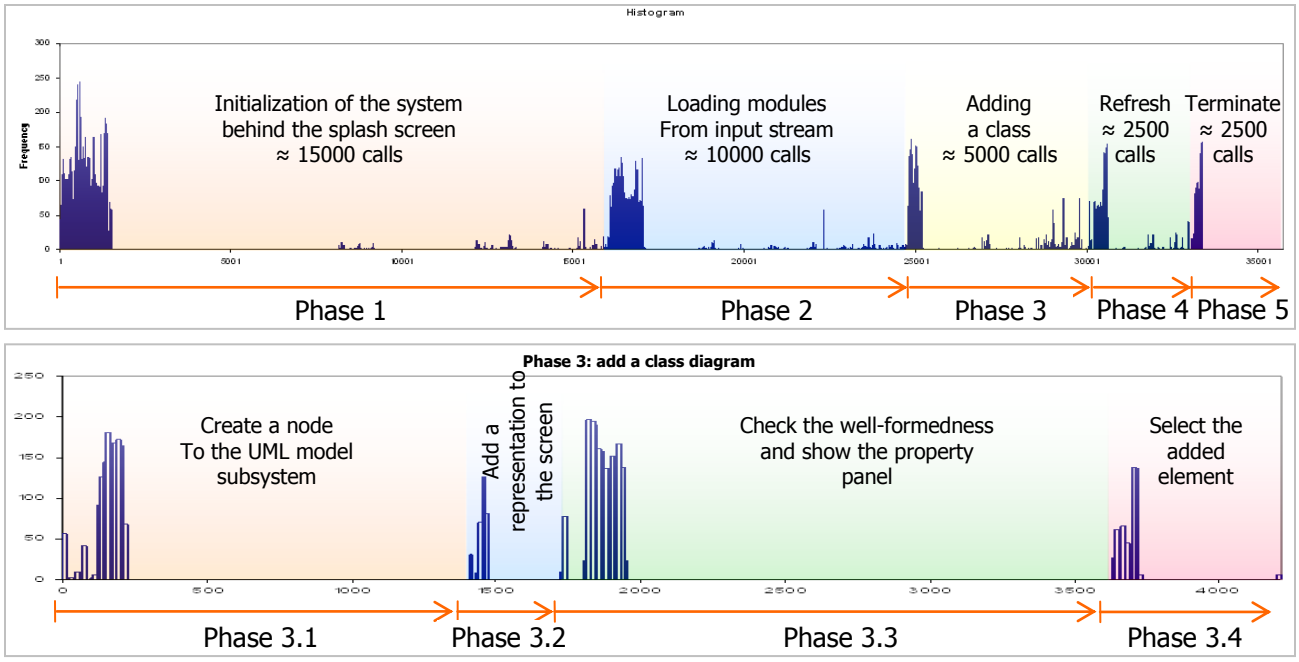


Figure 12. Up: major phases that comprise the execution trace, Bottom: the sub-phases that comprise Phase 3.

TABLE I. SUMMARY OF TASKS PERFORMED IN PHASE 3

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

Phase 3. The highest BIC score was for the partitioning with $K = 4$ as the best fit. Figure 12 shows the location of the four sub-phases and the high-level task that each of these sub-phases performs. The important tasks that are performed when drawing a class organized based on the sub-phases that were identified are summarized as high-level descriptions in Table 1. To describe these tasks as shown in Table 1, we referred to ArgoUML source code. We validated these tasks using ArgoUML documentation and the Cookbook for Developers of ArgoUML [26].

As we showed, in both cases studies, our approach was very successful in detecting phases in both traces and at different levels of granularity.

V. RELATED WORK

Research into phase detection has resulted in two groups of techniques over the years. While most related articles [8, 19, 18] are concerned with detecting execution phases at the hardware level for optimization purposes, there is only a very small group of techniques that are concerned with the phase detection for program comprehension. Trace abstraction techniques such as sampling [28], depth-limiting [29], and trace summarization [30] help maintainers in program comprehension. This work can also be regarded as a new approach for performing trace abstraction.

In their tool called ExtraVis, Cornilsson et al. [13] proposed an approach to phase detection. ExtraVis [15] offers an overview of the execution trace through its massive sequence (mural) view [14]. In this view the call relations are visualized based on the static system's structure of elements. This helps the user to visually detect the different phases of the system's execution. Similar to our approach, being aware of the execution scenario, the user can also hypothetically relate the repetitive (or ordered) patterns of elements to the repetitive (or ordered) features in the execution scenario. Then, zooming on a pattern, the user can verify his hypothesis.

The advantage of our approach over [13] is that it does not require efforts from the user to detect repeated call sequences and to determine the degree of similarity in certain execution phases since these are performed automatically as part of our technique while in ExtraVis the user is required to analyze large amounts of data visually. Our approach also differs in the fact that it takes the nesting level of methods into account when detecting phases while nesting levels are ignored in the mural view of ExtraVis.

Another approach that uses murals to visualize phases is proposed by Reiss et al. [5, 6]. In their tool called Jive they visualize the behaviour of a program as it is running. This 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. This makes it hard to visualize entire executions. Sampling the events for the execution trace is another shortcoming of Jive since it is hard to find proper sampling parameters.

Another online phase detection technique proposed by Watanabe [12] is based on the investigation of LRU cache for observing objects that are working for the current phase; a significant change in the cache shows the emergence of a new phase.

Kuhn and Greevy [17] suggest a possible analogy between analysis of trace information and signal processing. For this, they 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. This way, users can identify similar phases within a trace and between the traces. This technique cannot guaranty for similarities between methods in a phase. This is due to the fact that 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 is considered inessential data by applying multiple filtering logics (independent of method names), having as target mainly the representation size. This could potentially result in loss of important trace information during the abstraction process.

In our previous work [7], we proposed a phase detection techniques based on the fact that a phase shift within a trace appears when a certain set of methods responsible for implementing a particular task and which are predominant in one phase starts disappearing as the program enters another phase. We 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 that new ones start emerging. This approach is significantly different from the one proposed in this paper since it is not based on measured similarity and continuity among trace elements.

VI. Conclusions and Future Directions

In this paper, we presented a dynamic analysis approach for detecting execution phases of a program. We argued that execution phases can greatly simplify the analysis of large execution traces generated from exercising the system features, and hence enable the understanding of the behavior of software. Our approach is inspired by Gestalt laws for measuring the similarity and the continuity of the elements of a data space. We applied these laws to the content of execution traces to form dense groups that are candidate phases. The case studies showed promising results.

Although, we anticipate that the threshold *t* would be application-specific, further studies should be conducted to, at least, provide hints on acceptable ranges of thresholds. We also need to embed our approach in a trace analysis tool and work with software maintainers to evaluate its effectiveness in practice.

In the future, a first practical step will be to automatically identify relevant information about phases and provide an efficient representation of the flow of phases by detecting redundant phases. We would also like to investigate the ways in which our phase detection approach can help maintainers in redocumentation, extraction of crosscutting concerns, and fault localization. Finally, we are also interested in investigating how trace segmentation based on phase detection can play an important role in recovering a program's conceptual plans.

References

[1] O. Greevy, and S. Ducasse, "Correlating features and code using a compact two-sided trace analysis approach", *In Proc. of the 9th European Conference on Software Maintenance and Reengineering*, 2005, pp. 314–323.

[2] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, "Recovering behavioral design models from execution traces", *In Proc. of the 9th European Conference on Software Maintenance and Reengineering*, 2005, pp. 112 – 121.

[3] A. Zaidman and S. Demeyer, "Managing trace data volume through a heuristical clustering process based on event execution frequency", *In Proc. of the 8th European Conference on Software Maintenance and Reengineering*, 2004, pp. 329–338.

[4] A. Dunsmore, M. Roper, and M. Wood, "The role of comprehension in software inspection", *Journal of Systems and Software, Springer, 52(2-3)*, 2000, 121-129.

[5] S. P. Reiss, "Dynamic detection and visualization of software phases", *In Proc. of the 3rd International Workshop on Dynamic Analysis*, 2005.

[6] S. P. Reiss, "Visual representations of executing programs", *Journal of Visual Languages and Computing, 18(2)*, 2007.

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

[8] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior", *In Proc. of the 10th international Conference on Architectural Support For Programming Languages and Operating Systems*, 2002.

[9] M. Fisher, and K. Smith-Gratto "Gestalt theory: a foundation for instructional screen design", *Journal of Educational Technology Systems*, 27(4), 1998, pp. 361-371.

[10] W. S. Geisler, J. S. Perry, B. J. Super, and D. P. Gallogly "Edge Co-Occurrence in Natural Images Predicts Contour Grouping Performance", *Vision Research Journal*, 41(6), 2001, pp. 711-724.

[11] D. Pelleg and A. Moore, "X-means: Extending *K*-means with efficient estimation of the number of clusters", *In Proc. 17th Int. Conf. Machine Learning*, 2000, pp. 727–734.

[12] Y. Watanabe, T. Ishio, and K. Inoue, "Feature-level phasedetection for execution trace using object cache", *In Proc. of the International Workshop on Dynamic Analysis*, 2008, pp. 8–14.

[13] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen, "Understanding Execution Traces Using Massive Sequence and Circular Bundle Views", *In Proc. the 15th IEEE International Conference on Program Comprehension*, 2007, pp.49-58.

[14] D. F. Jerding and J. T. Stasko, "The Information Mural: A Technique for Displaying and Navigating Large Information Spaces", *IEEE Transactions on Visualization and Computer Graphics, 4(3)*, 1998, pp. 257-271.

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

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

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

[18] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques", *In Proc. of the 36th IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 217-227.

[19] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general purpose architectures", *In Proc. of the 33rd Ann. Intl. Sym. on Microarchitecture*, 2000, pp. 245-257.

[20] M. Wertheimer. *Laws of Organization in Perceptual Forms*. Harcourt Brace Jovanovich, London, 1938.

[21] K. Koffka. *Principles of Gestalt Psychology*. Hartcourt, New York, 1935.

[22] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations", *In Proc. Of the 5th Berkeley Symp. Math. Statistics and Probability*, 1967, pp. 281-296.

[23] G. Schwarz, "Estimating the dimension of a model", *The Annals of Statistics, 6(2)*, 1978, pp. 461–464.

[24] K. Smith-Gratto and M. Fisher, "Gestalt theory: A foundation for instructional screen design," *Journal of Instructional Technology Systems, 27(4)*, 1999, pp. 361–371.

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

[26] L. Tolke, M. Klink, M. Wulp. *Cookbook for Developers of ArgoUML: An Introduction to Developing the ArgoUML*. The Regents of the University of California. URL: http://argouml-downloads.tigris.org/nonav/argouml-0.18.1/cookbook-0.18.1.pdf

[27] P. C. Quinn and R. S. Bhatt, "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)*, 86(6), 2009, pp. 589–594.

[28] A. Chan, R. Holmes, GC. Murphy, ATT. Ying, "Scaling an object-oriented system execution visualizer through sampling", *In Proc. 11th Int. Workshop on Program Comprehension*, 2003, pp. 237–244.

[29] A. Rountev, B. H. Connell, "Object naming analysis for reverse-engineered sequence diagrams", *In Proc. of the 27th International Conference on Software Engineering*, 2005, pp. 254–263.

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