Exploiting Text Mining Techniques in the Analysis of Execution Traces

Heidar Pirzadeh[1], Abdelwahab Hamou-Lhadj[1], and Mohak Shah[2]
1Software Behaviour Analysis Lab
Department of Electrical and Computer Engineering
Concordia University
{s_pirzad, abdelw}@ece.concordia.ca
2Accenture Technology Labs
161 N Clark St, Chicago, IL, 60601, USA
mohak.shah@accenture.com

Abstract—**The analysis of execution traces can be useful in many software engineering activities including debugging, feature enhancement, performance analysis, and any other task that requires some degree of understanding of the way a system behaves. Traces, however, tend to be considerably large, which often hinders effective analysis of their content. There is a need to investigate ways to help software engineers find and understand important information conveyed in a trace despite the trace being massive. Motivated by the work done in the area of text mining, we propose, in this paper, a trace exploration approach based on examining the trace execution phases. The approach consists of automatically identifying relevant information about the phases as well as the ability to provide an efficient representation of the flow of phases by detecting redundant phases using a cosine similarity metric. We applied our approach to large traces generated from two different systems and were able to quickly understand their content and extract higher level views that characterize the essence of the information conveyed in these traces.**

Program Comprehension, Dynamic Analysis, Text mining, Software Maintenance.

I. INTRODUCTION

The analysis of execution traces is an important enabler for many software maintenance activities that require some understanding of the system behavior, such as locating system defects [14] and feature enhancement [13]. Trace exploration is however a challenging task partly due to the sheer size of typical traces, often millions of events. To alleviate this task, several trace abstraction techniques have been proposed (e.g., [1, 2, 3]). Although these techniques have shown to be useful in software maintenance, they suffer from several limitations, such as extensive reliance on user intervention, and dependence on particular visualization schemes [4]. The general consensus in the area of trace analysis is that more research is needed.

In our previous work [18], we presented an approach, called trace segmentation, for automatically dividing the content of a trace into smaller and meaningful trace segments that correspond to the program's main execution phases. Our definition of an execution phase is similar to the one presented by Reiss [6]: A segment of program's execution that performs a specific task, where the composing elements exhibit a common behavior at a level the programmer would recognize, such as initializing variables, performing specific computations, etc. Each phase consisted of a group of trace elements clustered together based on their relevance to the task being represented. We also showed that these phases can significantly simplify the exploration of large traces by allowing software engineers to navigate through a trace as a flow of execution phases instead of a series of mere low-level events.

In this paper, we continue this work by proposing a method for automatically identifying the trace elements that are most relevant to the implementation of each execution phase. This is particularly important since it can significantly simplify the exploration of large phases by allowing software engineers to quickly understand the phases of an execution and select the intended ones before deciding to dive into the details. Another contribution of the paper is a technique for identifying similar phases within a trace. It is possible that a single major computation happens several times in different periods during the execution of a program. Existing phase detection algorithms (e.g., [6, 12]) usually detect each occurrence as a different phase (although they are very similar). A better representation of a program execution is the one where a phase is indentified only once and referred to it in other places.

To achieve our objectives, we adopt techniques from the area of text mining, more particularly the Term Frequency, Inverse Document Frequency (TF-IDF) technique and the cosine similarity measure. We validate our approach using traces generated from two software systems.

The contributions of this paper can be further refined to help with tasks such as:

- The ability for software engineers to understand the most relevant elements that implement the traced scenario (or software feature). As such, the contribution of this paper falls under the category of feature location research.
- The recovery of high-level behavioral design models from large execution traces. The most important elements of a trace uncovered by our approach can be represented, for example, in a UML sequence diagram. These models can in turn be used for redocumentation,

or for assessing if the system does what it is supposed to do by comparing the resulting diagrams with existing design diagrams.

- Any area where trace summaries are needed. This will be particularly useful if the proposed techniques are integrated in a tool, in which the ability to switch between a high-level view of a trace and a detailed view is provided.

The rest of the paper is organized as follows: Section II draws analogy between trace analysis and text mining. In Section III, we present the approach. Section IV is concerned with the case study. We conclude the paper in Section VI after we present the related work.

II. TRACE ANALYSIS AND TEXT MINING

Dealing with large data spaces, whether the data takes the form of traces, text, or any other artifact, is in principle subject to similar challenges, among which perhaps the most important ones consist of coping with the large volume of data, overcoming the limited capacity of the human working memory [5], and the constant need to reduce the presence of noise in the data so as to focus on what is important.

Research in the area of text mining has long been active in addressing these challenges, which motivated us to explore how the application of existing techniques can be applied to the analysis of traces. However, before drawing any parallel between the two domains, we first need to determine the mapping between the concepts in the two domains as stated by Gentner in his theory of structural mapping [10]. Figure 1 shows three types of objects in the domain of text mining: Corpus, Document, and Term, where a document is a sequence of terms and a corpus is a set of documents. We propose to view an execution trace as a corpus and the execution phases that compose it as the corpus documents. In our previous work [18, 22], we

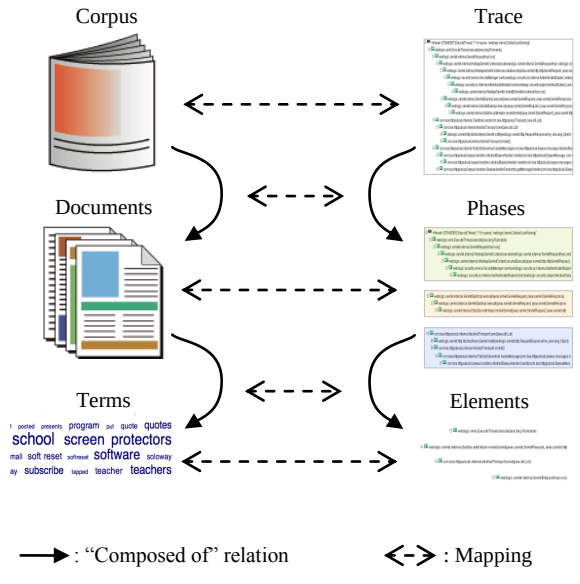

Figure 1. Mapping between the text mining and trace analysis domains

proposed a way to automatically detect execution phases from a trace. The essence of this decomposition is revisited in Section III.*A*. Each trace element can be viewed as a term within a phase document. Note that the mapping takes place not only between objects, but also between the containment relations between the objects.

There exists a variety of text mining techniques, in this study, we want to focus on the ones that make use of three types of information:

- Local information: It refers to the information inside individual documents (e.g., term frequency).
- Global information: It is the information from the collection of documents in the corpus (e.g., document frequency).
- Domain information that refers to the information from the domain of a term. For example, the term "can" has different significance in the domain of literature and the domain of packaging [29].

Similarly, we consider the information inside each phase (e.g., the frequency of an element within a phase) as local information in trace analysis. Global information in this domain can be considered as the information from the collection of phases in the trace (e.g., number of phases in which a particular element is invoked). Finally, domain information in trace analysis is the information about the trace elements where they are defined. This information could be extracted from the source code or the documentation (e.g., static call graph information of each element).

III. APPROACH

Figure 2 shows our approach for extracting relevant information from a trace based on the analysis of its execution phases. The approach encompasses two main phases. The first phase (Trace Segmentation) consists of automatically dividing the content of a trace into execution phases. To segment a trace, we use the phase detection technique that we presented in [18]. Other phase detection techniques such as the ones presented in [6, 12, 7] could also be used.

The second phase consists of the application of a newly designed technique called *content prioritization* with which the trace elements of each phase are weighted and the ones that have the highest weight are deemed to be the most representative elements of a phase. Once the elements are weighted, we extract the most representative ones. We also use the weighted elements to detect similar phases.

A. Trace Segmentation

Our process for segmenting a trace into execution phases is composed of two steps [18]:

- Step 1: Identifying candidate phases using the application of gravitational schemes (see next subsection) to group the trace elements into phases.
- Step 2: Identifying the beginning and end of each phase using K-means clustering.

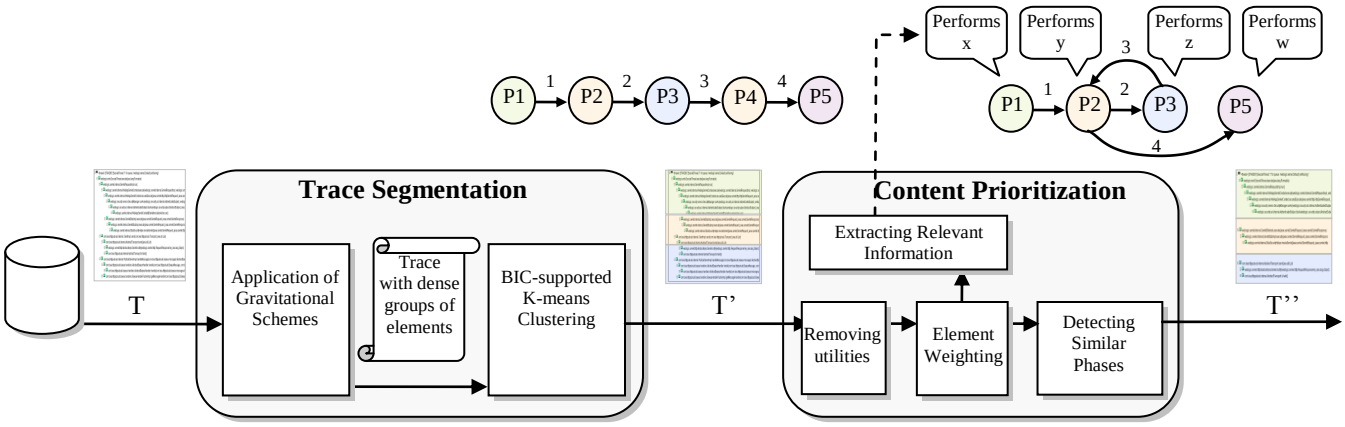

Figure 2. Overview of our proposed approach

Step 1: Phase detection

In our previous work [18], we have developed a phase detection technique that is inspired by Gestalt laws of perception (namely similarity and good continuation), which describe how people group similar items visually based on their perception [21]. Gestalt psychology is an application of physics to essential parts of brain physiology by telling the physiologist what kind of processes occur 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) [9, 21] to integrate the scene elements (i.e., objects and regions) as a whole and not just as points and lines.

We have developed two gravitational schemes, more precisely the similarity and continuity schemes, based on Gestalt laws, which we use as gravitational forces that yield the formation of dense groups of trace elements, the candidate execution phases. The similarity and continuity schemes operate as follows (please refer to [18] for a formal description of these schemes).

The objective of the similarity gravity scheme is to reposition the elements of a trace in such a way that similar elements gravitate towards 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 to make the distance between two same elements less than the distance between two different elements.

In Figure 3, we show the effect of applying the similarity gravity scheme to a sample trace composed of routines[1] `a`, `b`, `c`, and `d` (Figure 3(1)). The trace elements are mapped to a ruler indicating the position of the events as they occur. Assume that the similarity scheme works in a way that the distance between two calls to the same routine is reduced by half, the resulting trace after the application of the similarity scheme would therefore lead to two dense groups (see Figure 3(2)). The first one starts at the first invocation and is composed of calls to methods `a` and `b`, while the other one, which starts at the seventh invocation, contains calls to `c` and `d`. These groups might indicate the presence of two phases in the trace, although in practice one needs to carefully define the criteria by which two routines should be attracted to each other (see [18] for more discussion on this).

The second scheme that we have developed, the continuity gravity scheme, groups trace elements based on the nesting level of the routine calls by keeping the calls with higher nesting level closer to the previous routine calls. The higher is the nesting level of a routine call, the stronger it is attracted by the previous routine call. Using the nesting level of calls to detect execution phases has also been the topic of a number of studies [8, 12]. Watanabe et al. [12] used the nesting levels of a call tree to detect phases and locate phase shifts. They suggested that the depth of the call stack (i.e., the nesting level) is in 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.

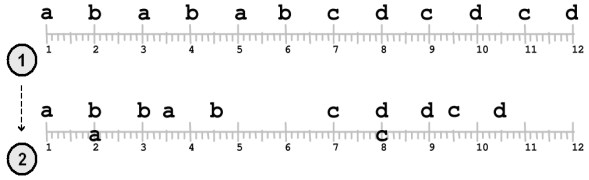

Figure 3. An example of applying the similarity scheme

Figure 4 shows the result of applying the continuity gravity scheme to a sample trace, assuming that the distance between two consecutive nesting calls is reduced by half. The new positioning of the elements seems to lead to two distinguishable phases (the phases are more distinguishable when we omit to visualize the nesting levels – Figure 4(3)). The first phase begins at the first method invocation and the second phase starts at the tenth method invocation.

[1] The focus of this paper is on traces of routine (method) calls.

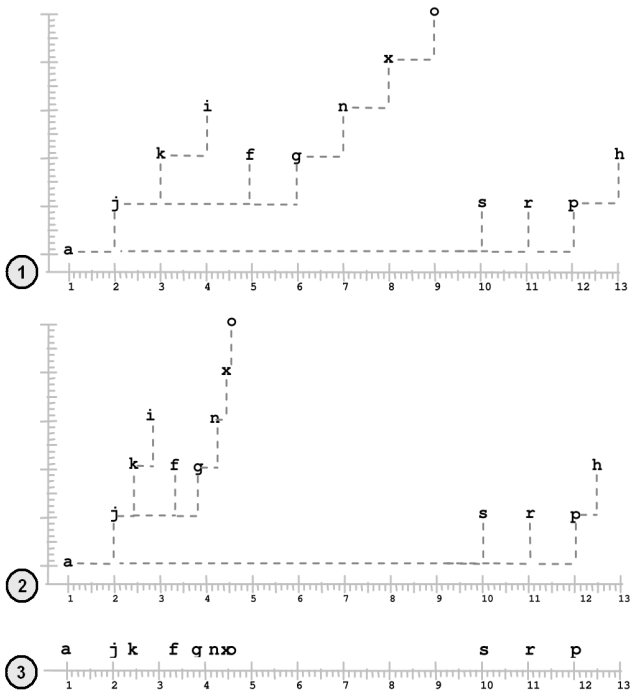

Figure 4. An example of applying continuity scheme

Step 2: Identification of Phase Boundaries

Once the trace elements are repositioned using the similarity and continuity gravity schemes, we need a way to automatically identify the beginning and end of each phase because it would be impractical to expect from programmers to distinguish the various phases visually for considerably large traces. For this, we chose K-means clustering as our clustering algorithm. In K-means clustering the number of clusters (i.e., the number of phases) should be given as an input to the 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 data points the variance becomes zero; this is known as overfitting). Therefore, the problem is reduced to finding a tradeoff between the number of clusters and the average variance of the clusters that keeps the number of clusters and the variance both minimized. This tradeoff is reached 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.

As shown in Figure 5, 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. To evaluate these partitionings, we compute the BIC score of each partitioning, the highest BIC means the 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.

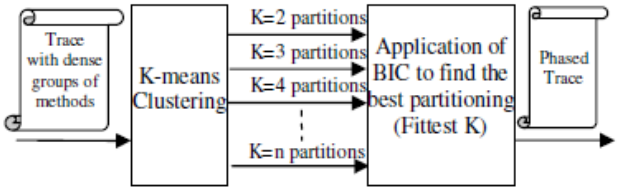

Figure 5. BIC supported K-means clustering

In [18], we applied our phase detection algorithm to large traces generated from two object-oriented systems. We validated the results by studying documentation of both systems and showed that our approach was successful in dividing these traces into meaningful phases.

B. *Content Prioritization*

The second step of our approach (also the main contribution of this paper) is to extract the trace elements that are most relevant to each execution phase. For this purpose, we use text mining techniques as previously mentioned. More precisely, the content prioritization phase is composed of the following steps, which are listed here and discussed in more detail in the subsequent sections:

1. We first remove utility routines from the execution phases to reduce the noise in the data. The process of removing utilities is similar to removing stop words from text.
2. We apply a weighting function to weigh elements of a phase according to their relevance. The higher the weight, the more representative the element.
3. We propose a way to select the most representative elements in a phase from the list of ranked elements obtained in 2.
4. We measure the similarity between phases based on their weighted elements.

1) *Utility Removal:*

Text mining techniques usually start with a preprocessing step that removes stop words -The words that add little value to the process of finding relevant information. Stop word identification, which is the process of identifying these words, makes use of domain and global information. For example, in the domain of English literature, stop words are among auxiliary verbs (e.g., have, be), pronouns (he, it), or prepositions (to, for). Similarly, we proceed with removing utilities from the trace before weighting its elements. According to Hamou-Lhadj et al. [25], a utility is a component that implements a low-level concept such as accessing methods or language libraries. In this paper, we limit ourselves to this type of utilities that can be detected without advanced processing techniques.

2) Element Weighting:

In text mining, the process of term weighting is used for finding representative terms in each documents of a corpus. One of the best known weighting schemes is called TF-IDF (Term Frequency, Inverse Document Frequency) [19]. The goal of TF-IDF term weighting is to obtain high weights for terms that are representative of a document's content and lower weights for terms that are less representative. The weight of a term depends both on how often it appears in the given document (term frequency, or *tf*) and on how often it appears in all the documents of the collection (document frequency, or *df*). In general, a high frequency of a term (high *tf*) in one document shows the importance of that term while if a term is scattered between different documents (high *df*), then it is considered less important. Therefore, if a term has high *tf* and low *df* (or high *idf* -inverse document frequency) then it will have a higher weight.

A similar idea can be adapted to trace analysis to weigh the elements of a trace. We suggest a weighting function that considers the frequency of trace elements across the execution phases. Our hypothesis is that a trace element that appears often in a particular phase, but appears relatively infrequently in other phases potentially indicates that it is doing something important in that particular phase. We use the trace *T* shown in Figure 6, where the execution phases are already identified, to illustrate the proposed weighting function.

Our weighting function is composed of three factors: local, global, and normalization. The weight of an element *i* in a phase *k* has the following general form:

$$w_{i,k} = L_{i,k} G_i N_k$$

where $L_{i,k}$ is the local weight of the element *i* in phase *k* (local information) which is usually based on the number of occurrences of the element in the phase. G_i is the global weight of the element in the phases of the trace (global information). This factor tends to under-weigh these elements that are too common in the trace. N_k is the normalization factor for the element weights in phase *k*. We create an index vector called *element vector* for each phase. The magnitude of each element in this vector indicates how well that element represents the content of the phase based on its frequency within the phase and other phases. If an element occurs very frequently in some phases, but occurs rarely in the trace as a whole, it will be given high weight in the element vector.

The element frequency $ef_{i,j}$ of element *i* in phase *j* is defined as the number of times that *i* occurs in *j*. Similar to approaches in text mining, and since the importance of an element does not increase proportionally with the element's frequency, we use the following local weighting $L_{i,k}$ for element frequency.

$$L_{i,k} = \begin{cases} 1 + \log_{10} ef_{i,j} & \text{if } ef_{i,j} > 0 \\ 0 & \text{otherwise} \end{cases}$$

In Figure 6, $L_{i,k}$ is calculated for the elements in each of the 3 phases of trace T. For instance, in Phase 1, the element "d" is invoked twice, therefore, the local weight of L(d)=1+log(2)=1.3.

Trace T		$L_{i,k}$	G_i	$w_{i,k}$
Phase 1	a	L(a)=1	G(a)=0.17	w(a)=0.45
	d	L(d)=1.3	G(d)=0.17	w(d)=0.58
	c	L(c)=1.4	G(c)=0.17	w(c)=0.66
	i	L(i)=1	G(i)=0	w(i)=0
	d	L(e)=1	G(e)=0	w(e)=0
	c			
	e			
	c			
Phase 2	h	L(h)=1	G(h)=0.47	w(h)=0.50
	i	L(i)=1	G(i)=0	w(i)=0
	m	L(m)=1.3	G(m)=0.47	w(m)=0.65
	n	L(n)=1	G(n)=0.47	w(n)=0.50
	e	L(e)=1	G(e)=0	w(e)=0
	o	L(o)=1.3	G(o)=0.17	w(o)=0.24
	m			
	o			
Phase 3	a	L(a)=1	G(a)=0.17	w(a)=0.41
	o	L(o)=1	G(d)=0.17	w(d)=0.53
	d	L(d)=1.3	G(c)=0.17	w(c)=0.60
	c	L(c)=1.4	G(i)=0	w(i)=0
	i	L(i)=1	G(e)=0	w(e)=0
	d	L(e)=1	G(o)=0.17	w(o)=0.41
	c			
	e			
	c			

Figure 6. Running example to illustrate the weighting function

Following the general form, we want to assign the weight $w_{i,k}$ to element *i* in phase *k* in proportion to the frequency of occurrence of the element in phase *k*, and in inverse proportion to the number of phases in which the element is invoked. It should be noted that the phase lengths, and hence the number of non-zero element weights assigned to a phase, varies widely. To allow a meaningful final retrieval similarity, it is convenient to use a length normalization factor as part of the element weighting formula. A high-quality element weighting formula for $w_{i,k}$, the weight of element *i* in phase *k* is

$$w_{i,k} = \underbrace{\frac{\overbrace{(\log(ef_{i,k}) + 1)}^{L_{i,k}} * \overbrace{\log(N / n_i)}^{G_i}}{\sqrt{\sum_{j=1}^{e} [(\log(ef_{j,k}) + 1) * \log(N / n_j)]^2}}}_{\frac{1}{N_k}}$$

where $ef_{i,k}$ is the occurrence frequency of element *i* in phase *k*, *N* is the total number of phases, n_i is the number of phases with element *i* assigned and *e* is the total number of elements. The factor $\log(N/n_i)$ is an inverse phase frequency (similar to "*idf*") factor which decreases as the elements are used widely in a trace and the denominator in the equation is used for weight normalization. This factor is used to adjust the element vector of the phase to its norm, so all the phases have the same modulus and can be compared no matter the size of the phase.

This weighting system enables us to adjust the weighting for an element according to not only local but also global information available in the entire trace. Figure 6 shows $w_{i,k}$ for the elements in each phase. For instance, $w_{d,1}$ the weight of element "d" in Phase 1, given that *G*(d) in the trace is 0.17, is calculated as follows:

$$w_{"d",Phase1} = \frac{1.3 * 0.17}{\sqrt{(0.17)^2 + (0.22)^2 + (0.26)^2}} = 0.45$$

3) Extracting Relevant Information:

The output of the element weighting step is a list of phase elements ranked according to their relevance. We need to determine a threshold with which we can select the most representative elements among this list. For example, a software engineer can decide to only consider the top 20% of the elements that have the highest ranking to be the most representative elements of a phase.

Another possible method is to set a cap on the maximum number of elements we want to extract and compute the number of elements per phase proportionally to the size of that phase as follows:

$$R(P_i) = \left\lceil M * \frac{|P_i|}{|T|} \right\rceil$$

where $R(P_i)$ in the number of most relevant elements of a phase P_i, *M* is the maximum number of element considered given as input, |*T*| is the size of the trace after removing the utilities, and $|P_i|$ is the size of a phase. In the example of Figure 6, if the maximum number of elements that we want is set to 3, given that the size of the trace which is 25, we have:

- *R*(*Phase1*)=3*8/25=1
- *R*(*Phase2*)=3*8/25=1
- *R*(*Phase3*)=3*9/25=1

Then for each phase we chose the top 1 element from the element vector as the most representative of that phase. This way, "m" is the most representative element of Phase 2 and "a" is the most representative of both phases 1 and 3.

We can further improve the information contained in each phase, and hence facilitate the browsing the trace, by enriching the phase most representative elements with any descriptive information such as source code comments or any information extracted from valid documentation. We are aware that this informal source of data might not be reliable in practice though. In the worst case scenario, the element names will be the only information that can be used. Figure 7 shows the high-level flow of phases in Trace T of Figure 6 where the relevant information about each phase is added. The trace T can now be browsed as a sequence of phases with relevant information rather than a large trace of events.

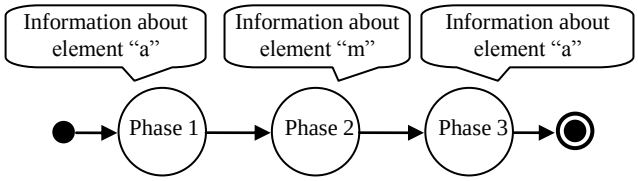

Figure 7. Flow of phases with relevant information added

4) Determining Similar Phases:

The similarity between two objects is in general regarded as how much they share in common. In the domain of text mining, the most commonly used measure for evaluating the similarity between two documents is the cosine of the angle between term vectors representing the documents. In the same way, the similarity between two phases can be calculated based on the list of their matching elements.

More precisely, we measure the similarity between each pair of phases by calculating the cosine of the angle between the element vectors P_x, P_y representing the phases:

$$S(P_x, P_y) = \frac{\sum_{i=1}^{n} w_{i,x} * w_{i,y}}{\sqrt{\sum_{i=1}^{n} (w_{i,x})^2} * \sqrt{\sum_{i=1}^{n} (w_{i,y})^2}}$$

The weights cannot be negative and, thus, the similarity between two phases ranges from 0 to 1, where 0 indicates independence, 1 means exactly the same, and in-between values indicate intermediate similarity.

To determine the similarity between the phases, we take the element vector of each phase and measure the similarity between each pair of vectors. If the similarity between two phases is more than a user-specified threshold they are considered as the same. As a result, for phases that are repeated in a sequence of phases, the first occurrence is kept and the next occurrences are referred to the first occurrence. It should be noted that, by definition, consecutive execution phases must be dissimilar enough to be detected as separate phases in the first place.

For our sample trace T in Figure 6, the similarities between phases are shown in Figure 8. If we consider a threshold of 90% for two phases to be considered similar, then Phase 1 and Phase 3 are the same. This enables us to reduce the high-level flow of the phases to the one shown in Figure 8.

	Phase 2	Phase 3
Phase 1	0%	91%
Phase 2		9%

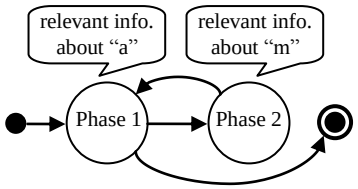

Figure 8. Calculating the similarity between phases

IV. CASE STUDIES

We conducted experiments with traces generated from two software systems WEKA [24] and ArgoUML [16] that we present separately in this section. Both systems where instrumented using TPTP (the Eclipse instrumentation tool).

A. *WEKA:*

WEKA is a machine learning tool that implements several learning algorithms [24]. We used WEKA 3.7.3 (latest version) which consists of 76 packages, 1133 classes, 14210 methods, and 226220 lines of code.

To generate a trace, we applied the WEKA machine learning toolkit to build a decision tree learning algorithm for classifying data instances. Each data instance is typically a vector of attribute values where each attribute denotes some measurement of interest. Training involves executing the decision tree learning algorithm on a set of training data instances. The algorithm identifies specific patterns in the data and outputs a decision tree model each node of which uses a predicate built on specific data attributes to fine tune the classification. The output decision tree model is subsequently evaluated on a separate set of test data instances. The model evaluates the predicate on each node of the decision tree against their corresponding values in each data instance and outputs a class prediction. Different performance statistics, e.g., prediction accuracy over all test instances, are then calculated for evaluation purposes [27]. As such, the core process of learning a model consists of three main stages: data input, learning a model from training data, evaluating the model on test data.

The detailed steps of the scenario we used to generate a trace are: (a) Run the WEKA Explorer tool, select a training set, go to the Classify tab, (b) Select the classifier J48 (see [28] for a description on this algorithm), select the "supplied test set" option, (c) Select a test set, start the classification, close the program.

The generated trace contains 87,2291method calls. Since each routine requires two events (entry and exit events) , the size of the trace in terms of events is 1,744,582. The number of distinct routines involved in the trace is 1309.

The application of the phase detection technique resulted in the identification of four main phases. This can be seen in Figure 9 that shows four dense groups of methods appearing in the phase detection. Table 1 shows the size of the detected phase (SP) in terms of the number of method calls.

In the next step, the original trace with its phases annotated is given to the utility removal component, where accessing methods are removed. The resulting trace is then processed to weigh the elements of each phase. Table 1 shows the number of elements with non-zero weights in the element vector for each phase (SV). The element vector is passed to the "preparation of relevant information" component where the top first 20% of the methods in each element vector is selected as most representative methods for each phase. This threshold is selected arbitrarily. Future studies should focus on investigating ways to set this threshold automatically, although we anticipate that this threshold will vary from one system to another. A tool that integrates our techniques should allow enough flexibility to vary this threshold. Table 2 shows the representative methods of each phase (the methods are sorted based on their original order of invocation to help with better understanding of the flow of events). Table 1 shows the number of representative methods for each phase (SR).

Table 1. Statistics about representative elements

Phases	(SP)	(SV)	(SR)	Ratio SR/SP
Phase 1	82544	95	19	0.02%
Phase 2	124586	137	27	0.02%
Phase 3	445291	69	14	0.003%
Phase 4	219871	127	25	0.01%

We referred to the source code and the WEKA documentation [24] to extract descriptions of the routines that were deemed most representative of each phase. We were able to interpret the phases that composed the original trace by analyzing the phases' most relevant information, which significantly simplified the understanding of the entire trace. In what, we briefly discuss the information contained in the trace.

The first phase involves initialization of the WEKA toolkit itself. Since WEKA has a Graphical User Interface (GUI), this initialization also involves calls to processes that

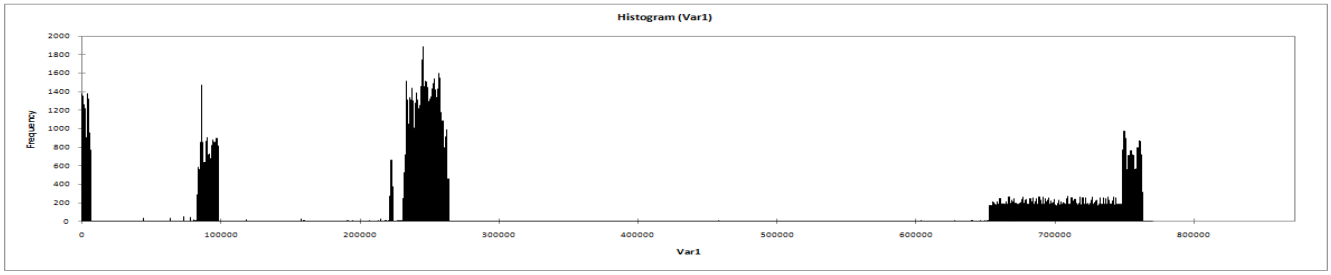

Figure 9: detected phases for execution trace of WEKA

Table 2. Representative elements of the WEKA trace

Reps. of Phase 1	Reps. of Phase 3
weka.core.Tee.add weka.core.WekaPackageManager.loadPackages weka.core.ClassDiscovery.initCache weka.core.ClassCache.initFromDir weka.core.ClassCache.add weka.core.ClassCache.cleanUp weka.core.ClassCache.extractPackage weka.core.ClassCache.initFromJar weka.core.ClassDiscovery.find weka.core.ClassDiscovery.hasInterface weka.core.ClassCache.remove weka.core.ClassDiscovery.addCache weka.gui.GenericPropertiesCreator.isValidClassname weka.core.ClassDiscovery.isSubclass weka.core.Stopwords.add weka.core.Tee.size weka.core.converters.AbstractFileSaver.resetOptions weka.core.converters.AbstractSaver.resetOptions weka.gui.GenericObjectEditor.registerEditor	weka.core.WekaEnumeration.hasMoreElements weka.core.WekaEnumeration.nextElement weka.classifiers.trees.j48.Distribution.add weka.classifiers.trees.j48.Distribution.numClasses weka.classifiers.trees.j48.Distribution.total weka.core.Instances.quickSort weka.core.Instances.partition weka.classifiers.trees.j48.EntropyBasedSplitCrit.logFunc weka.classifiers.trees.j48.Distribution.shiftRange weka.classifiers.trees.j48.Distribution.perBag weka.classifiers.trees.j48.InfoGainSplitCrit.splitCritValue weka.classifiers.trees.j48.EntropyBasedSplitCrit.newEnt weka.classifiers.trees.j48.Distribution.numBags weka.classifiers.trees.j48.Distribution.perClassPerBag
Reps. of Phase 2	**Reps. of Phase 4**
weka.gui.explorer.Explorer.addCapabilitiesFilterListener weka.core.Instances.numAttributes weka.core.Attribute.indexOfValue weka.core.AbstractInstance.weight weka.core.Instances.numInstances weka.core.Instances.instance weka.gui.explorer.PreprocessPanel.updateCapabilitiesFilter weka.core.Capabilities.assign weka.core.Capabilities.handles weka.core.Capabilities.disable weka.core.Capabilities.hasDependency weka.core.Capabilities.disableDependency weka.core.Instances.classIndex weka.core.Capabilities.enable weka.core.AbstractInstance.classIndex weka.core.AbstractInstance.isMissing weka.core.DenseInstance.value weka.core.Instances.attributeStats weka.core.AttributeStats.addDistinct weka.experiment.Stats.add weka.experiment.Stats.calculateDerived weka.gui.explorer.ClassifierPanel.updateCapabilitiesFilter weka.gui.explorer.ClustererPanel.updateCapabilitiesFilter weka.gui.explorer.AttributeSelectionPanel.updateCapabilitiesFilter weka.core.Instances.swap weka.core.Attribute.isString weka.core.Capabilities.enableDependency	weka.gui.explorer.ClassifierErrorsPlotInstances.process weka.classifiers.Evaluation.evaluateModelOnceAndRecordPrediction weka.classifiers.Evaluation.evaluationForSingleInstance weka.core.AbstractInstance.dataset weka.core.DenseInstance.freshAttributeVector weka.core.DenseInstance.toDoubleArray weka.classifiers.trees.J48.distributionForInstance weka.classifiers.trees.j48.ClassifierTree.distributionForInstance weka.core.AbstractInstance.numClasses weka.classifiers.trees.j48.ClassifierTree.localModel weka.classifiers.trees.j48.ClassifierTree.son weka.classifiers.trees.j48.ClassifierSplitModel.classProb weka.classifiers.trees.j48.NoSplit.weights weka.classifiers.trees.j48.Distribution.prob weka.classifiers.Evaluation.updateStatsForClassifier weka.classifiers.Evaluation.updateMargins weka.classifiers.Evaluation.makeDistribution weka.classifiers.Evaluation.updateNumericScores weka.classifiers.evaluation.NominalPrediction.updatePredicted weka.core.AbstractInstance.classAttribute weka.classifiers.evaluation.NominalPrediction.distribution weka.classifiers.evaluation.NominalPrediction.actual weka.classifiers.evaluation.NominalPrediction.weight weka.gui.visualize.Plot2D.convertToPanelX weka.gui.visualize.Plot2D.convertToPanelY

establish communication channels through this GUI. Some prominent examples in the most frequently called routine in Phase 1 can be seen with regard to the `weka.core.tee` objects. These objects refer to the WEKA's I/O stream initialization that enables it to both communicate with GUI interface selections by the user and establish streams for data input and results output.

The next phase (Phase 2) involves reading and organizing the data in requisite data structures. This phase prepares the data as well as enables data capabilities based on data specifics. Data organization and preparation is represented by the calls to the `weka.core.Instances` and `weka.core.Capabilities` methods that involve organizing and handling instances in an ordered set, and set the classifier-specific data handling preferences respectively.

The following phase (Phase 3) involves executing the learning algorithm to build a model on the training data. This is represented by methods in the `weka.classifiers.trees.j48 class`.

Finally, the last phase consists of the evaluation of the decision tree model output by Phase 3, on a set of test instances. This is indicated by calls to the methods in `weka.classifiers.Evaluation` class.

Since a prediction is obtained on each individual test data instance, repeated calls to `weka.classifiers,Evaluation.evaluationFo rSingleInstance` method can be seen. This method performs an instance-wise evaluation of the decision tree model. This phase also estimates different performance statistics for the model.

The phase element vectors were also used to determine the similarity between each pair of phases. As shown in Table 3, the calculated similarities between every pair of phases was less than 1%. This meant that the high-level flow of phases cannot be further reduced. The small value of similarities between consecutive phases also showed the good quality of the phase detection.

Table 3. Similarities between phases for WEKA Trace

	P2	P3	P4
P1	0.18 %	0.04%	0.00%
P2		0.98%	0.68%
P3			0.48%

Finally, the high-level view of the flow of phases with assigned description (extracted by reading the WEKA documentation) is shown in Figure 10. This high-level information flow is obtained by investigating a very small percentage of the original trace, which is quantified as SR/SP (Table 1). The content prioritization step, except for assigning description to the selected phase elements which is done manually, took 74 sec on an Intel Core Duo CPU 2.00GHz, 2MB cache, 1GB main memory, running Windows XP.

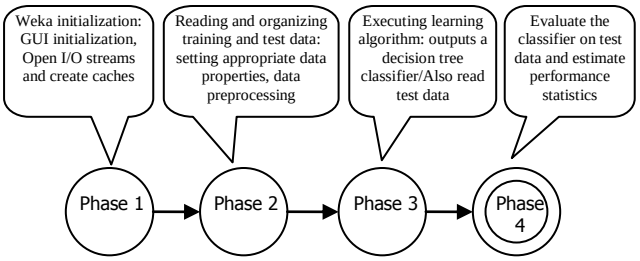

Figure 10. Flow of phases with relevant information added

B. *ArgoUML:*

For the second case study, we applied our technique to a trace generated from ArgoUML [16] by exercising the following scenario: Starting up ArgoUML, drawing a class on the class diagram, and quitting ArgoUML). The resulting trace contained 38321 method calls (2330 distinct methods). Figure 11 shows the result of applying the phase detection technique to the ArgoUML trace. Five phases have been identified. Table 4 shows the size of each detected phase as the number of method calls (SP). Similar to the previous case study, we applied the removal of utilities and the element weighting steps on the resulting trace elements. The top 20% of each vector is selected as most representative elements of each phase (see Table 4 for the number of representatives for each phase (SR)). The information about the representative

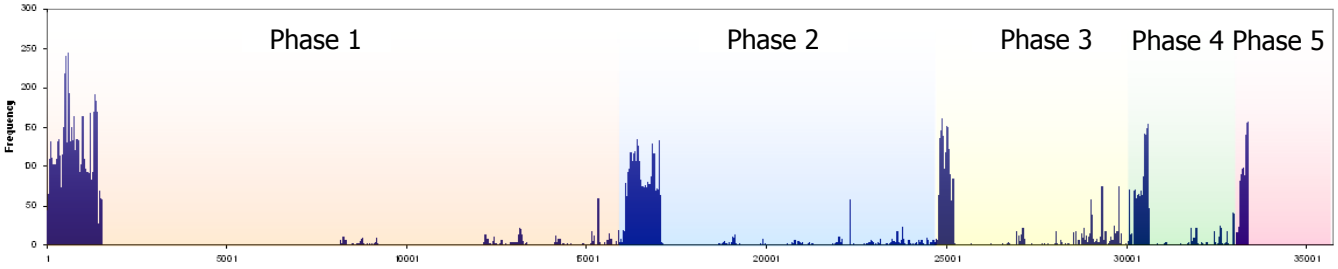

Figure 11. Detected phases for execution trace of ArgoUML

methods is gathered from the documentation and comments in the source code of the system [26].

Table 4. Statistics about representative elements

Phases	(SP)	(SV)	(SR)	Ratio SR/SP
Phase 1	16035	334	47	0.29%
Phase 2	9089	231	34	0.37%
Phase 3	4225	270	38	0.89%
Phase 4	3832	113	16	0.41%
Phase 5	5140	83	12	0.23%

Similar to the previous system, we were able to understand the original trace by examining the most relevant elements of its phases, which we briefly (due to space limitation) review in what follows.

The first phase focuses on the initialization of ArgoUML where the main application frame (e.g., main panes: navigation pane, multieditor pane, to-do pane, and details pane), status bar, and project are set up. The second 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 (Phase 4) refreshes and updates the models properties set in the previous phase, such as boundaries, NameText, font, and etc. The representative methods of the last phase (e.g., save methods, menu selection method, and exit methods) clearly show the termination of the application.

Table 5. Similarities between phases for ArgoUML Trace

	P2	P3	P4	P5
P1	0.79 %	0.16%	0.01%	0.00%
P2		0.32%	0.33%	0.53%
P3			2.50%	0.13%
P4				3.75%

The element vectors are then used to measure the similarity between phases. As shown in Table 5, a very small similarity between the phases does not suggest any change to the sequence of phases in the high-level as shown in Figure 12.

Finally, the high-level view of the flow of phases with assigned description is shown in Figure 12. Table 4 shows the percentage of the element investigated in each phase to extract relevant information (SR/SP). The content prioritization stage except for the information gathering which is done manually took 14 sec on an Intel Core Duo CPU 2.00GHz, 2MB cache, 1GB main memory, running Windows XP.

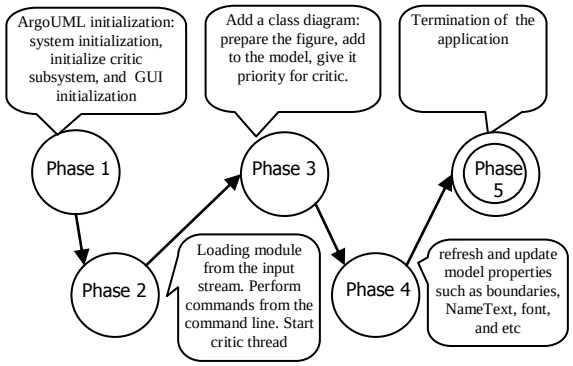

Figure 12. Flow of phases with relevant information added

V. Related Work

Wilde et al. [9] introduced the concept of Software Reconnaissance where traces generated by exercising several features are compared to identify components specific to the feature at hand. In our approach, we only generate one trace for generating a flow of phases. Eisenbarth et al. [20] proposed a hybrid feature location approach where formal concept analysis is applied on the execution traces to determine the relation between features.

Poshyvanyk et al. [17] introduced an approach based on information retrieval (IR) for feature location. Our work adapts a text mining technique on the trace elements for identifying their significance in each phase.

Asadi et al. [30, 31] also proposed an interesting approach which uses IR to identify concepts in execution traces. Our work is different from theirs in that we use all three types of trace global, local, and domain information for our flow of phases while they use English literature for stop-word removal, static structure of the code for local and global information.

Greevy et al. [1] exploited the relationship between features and classes to analyze the way features of a system evolve and to detect changes in the code from a feature perspective. Rather than detecting feature specific components, the main focus of the authors approach is on

studying how the classes may change their roles during software evolution.

Visualization approaches [6, 14, 15] have also been used to reduce the amount of trace information to look at. The advantage of our approach over these is that it does not require efforts hypothesis in detecting phases, their flow, and their relevant information. Watanabe [12] proposed a phase detection technique based on the investigation of LRU cache and [8] suggest a possible analogy between analysis of trace information and signal processing where users can identify similar phases within a trace and between the traces. Both techniques are not focused on providing the user with relevant information about phases.

VI. CONCLUSION

We proposed a technique for extracting important information about a trace by analyzing the most relevant information of the execution phases that compose it. We demonstrated through the case study that our approach can significantly simplify the analysis of large traces. Immediate future direction would be to continue experimenting with this approach. We also need to investigate thresholds that govern the phase detection technique as well as the way most representative phase elements are selected. We would also like to investigate ways in which our proposed technique 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 system's conceptual plans.

ACKNOWLEDGMENT

This work is partly supported by NSERC (Natural Sciences and Engineering Research Council of Canada).

REFERENCES

[1] O. Greevy, and S. Ducasse, “Correlating features and code using a compact two-sided trace analysis approach,” In Proc. of CSMR’05, 314-323, 2005.

[2] A. Hamou-Lhadj, E. Braun, D. Amyot, T. Lethbridge, “Recovering behavioral design models from execution traces,” In Proc. of CSMR’05, 112-121, 2005.

[3] A. Zaidman and S. Demeyer, “Managing trace data volume through a heuristical clustering process based on event execution frequency,” In Proc. of CSMR’04, 329–338, 2004.

[4] A. Hamou-Lhadj, and T. C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques,” In Proc. of CASCON’04, 42-54, 2004.

[5] G. A. Miller, “The magical number seven or minus two: Some limits on our capacity of processing information,” Psychological Review, 63(2), 281–97, 1956.

[6] S. P. Reiss, “Visual representations of executing programs,” Journal of Visual Languages & Computing, 18(2), 126-148, 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 SERA’10, 207-214, 2010.

[8] A. Kuhn and O. Greevy, “Exploiting the analogy between traces and signal processing,” In Proc. of ICSM'06, 320-329, 2006.

[9] N. Wilde, M. C. Scully, “Software Reconnaissance: Mapping program features to code,” Journal of Software Maintainence: Research and Practice, 7(1), 49-62, 1995.

[10] D. Gentner, “Structure-mapping: a theoretical framework for analogy,” Cognitive Science,7(2), 155–170, 1983.

[11] D. Pelleg and A. Moore, “X-means: Extending K-means with efficient estimation of the number of clusters,” In Proc. of ICML’00, 727–734, 2000.

[12] Y. Watanabe, T. Ishio, K. Inoue, “Feature-level phasedetection for execution trace using object cache,” In Proc. of WODA’08, 8-14, 2008.

[13] T. Systä, “Understanding the Behaviour of Java Programs”, In Proc. of WCRE’00, 214-223, 2000.

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

[15] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, A. van Deursen, “Understanding Execution Traces Using Massive Sequence and Circular Bundle Views,” In Proc. ICPC’07, 49-53, 2007.

[16] ArgoUML, URL: argouml.tigris.org

[17] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, 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, 2007.

[18] H. Pirzadeh, A. Hamou-Lhadj, "A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension", In Proc of ICECCS '11, 221–230, 2011.

[19] T. Joachims, “Text Categorization with Support Vector Machines: Learning with Many Relevant Features,” In Proc. of ECML98, 137-142, 1998.

[20] T. Eisenbarth, R. Koschke, D. Simon “Locating features in source code,” IEEE Transactions on Software Enginnering, 29(3), 210-224, 2003.

[21] K. Koffka. Principles of Gestalt Psychology. Hartcourt, NY, 1935.

[22] H. Pirzadeh, A. Hamou-Lhadj, “A Software Behaviour Analysis Framework Based on the Human Perception Systems”, In Proc. of ICSE’11, New Ideas and Emerging Results Track, 2011.

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

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

[25] 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. ICPC’06, 181–190, 2006.

[26] L. Tolke et al. Cookbook for Developers of ArgoUML: An Introduction to Developing the ArgoUML, URL: http://argouml.tigris.org/

[27] N. Japkowicz and M. Shah. Evaluating Learning Algorithms: A Classification Perspective. Cambridge University Press, 2011.

[28] J. R. Quinlan. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, 1993.

[29] R.J. Price “Automatic stop word identification and compensation”, US Patent, 7, 720-792, 2010.

[30] F. Asadi, M. Di Penta, G. Antoniol, Y.-G. Gueheneuc, “A heuristic-based approach to identify concepts in execution traces,” In Proc. of CSMR’10, 15-18, 2010.

[31] F. Asadi, G. Antoniol, Y.-G. Gueheneuc, “Concept Location with Genetic Algorithms: A Comparison of Four Distributed Architectures,” In Proc. of SSBSE '10, 153-162, 2010.