

Understanding the Complexity Embedded in Large Routine Call Traces with a Focus on Program Comprehension Tasks

Abdelwahab Hamou-Lhadj

*Department of Electrical and Computer Engineering, Concordia University
1455 de Maisonneuve Blvd. West
Montréal, Québec H3G 1M8 Canada
abdelw@ece.concordia.ca*

Timothy C. Lethbridge

*School of Information Technology and Engineering (SITE), University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
tcl@site.uottawa.ca*

Abstract

The analysis of execution traces has been shown to be useful in many software maintenance activities that require a certain understanding of the system's behaviour. Traces, however, are extremely large, hence are difficult for humans to analyze without effective tools. These tools usually support some sort of trace abstraction techniques that can help users understand the essence of a trace despite the trace being massive. Designing such tools requires a good understanding of the amount of complexity embedded in traces. Trace complexity has traditionally been measured using the file size or the number of lines in the trace. In this paper, we argue that such metrics provide limited indication of the complexity of a trace. We address this issue by presenting a catalog of metrics for assessing the various facets of traces of routine calls, with the ultimate objective being to facilitate the development of tools for the exploration of lengthy traces. We show the effectiveness of our metrics by applying them to 35 traces generated from four software systems.

Keyword: Trace analysis, program comprehension, software maintenance, software metrics

1. Introduction

Understanding the behaviour of software systems is an important aspect of program comprehension, which in turn is an essential process in software maintenance. The behaviour of particular runs of a software system is typically represented using execution traces. However, raw traces of most interesting runs of a program being examined tend to be overwhelmingly large. This has led to the development of trace analysis tools of various kinds. A survey of many of these tools and their supported techniques is presented in [1]. Using these tools, software engineers can reduce the size of the trace by applying several trace abstraction techniques ranging from mere removal of repetitive components to identifying patterns and automatic filtering of unwanted components.

However, the behaviour embedded in large traces can be so complex that, without good understanding of the complexity of a trace, it is very hard to know which techniques to

apply. There is, therefore, a need to embed *quantitative guidance* in tools, in order to semi-automatically or automatically help the user extract the information most likely to be relevant. By quantitative guidance, we mean *measuring* some aspects of a trace, or part of it, in order to determine whether that the trace is simple enough to understand or useful enough to display.

At first glance, one might imagine that measuring a trace could be rather straightforward: A naïve approach might just be to report the file size or the number of lines in the trace. However, a myriad of subtleties arise, for example:

- The file size and number of lines depend on the schema used to represent the trace and the syntax used to convey the data specified by such a schema.
- Not all elements of a trace are equally important, nor do all elements contribute equally to complexity. For example, a long series of identical method calls would be rather simpler to understand than a highly varied and non-repetitive sequence of the same length.
- The notion of complexity is itself rather vague, suggesting that we need to be able to measure a wide variety of aspects that may contribute to complexity so we can later experiment with various approaches to complexity reduction.

The objective of this paper is to present a set of metrics that we have developed which measure relevant properties of execution traces, particularly properties that can reveal the effort required to analyze a large trace and explore its various components. We achieve this by measuring aspects of an execution trace ranging from the volume of information contained to more structural relationships among the components invoked in the trace. We also show the application of these metrics to over thirty execution traces generated from four software systems. The size of these traces varies from hundred of thousands to million of events.

This paper is organized as follows: In the next section, we present background information and related work. We define the metrics in Section 3. The application of the metrics to 35 traces generated from four software systems is presented in Section 4. We conclude the paper in Section 5.

2. Background and Related Work

In this section, we motivate the focus on analyzing traces of routine calls by showing their importance in program comprehension. We introduce the concept of comprehension units that will help us define a subset of the proposed metrics. The section ends with a discussion of related work.

2.1. Traces of Routine Calls

Software engineers can generate several types of traces, including traces of routine calls, of inter-process communication, and of executed statements. In fact, one can trace just about any aspect of the system that is deemed helpful to accomplish the task at hand. In this paper, we focus on tools that process traces of routine (method) calls. Such traces are at an intermediate level of abstraction between highly detailed statement-level traces and traces of interactions, such as messages, among high-level system components. The former are rarely used since they tend to produce vastly more data than needed, whereas the latter

can only reveal the architectural characteristics of the system. Traces of routine calls have also been shown to be useful at solving practical maintenance problems such as uncovering faulty behaviours [2], adding new features to an existing system [3, 4], or detecting system inefficiencies [5].

There are various techniques for generating traces. The first is based on instrumenting the source code; this consists of inserting probes (e.g., print statements) at appropriate locations in the source code. Instrumentation is usually done automatically. The second technique consists of instrumenting the execution environment in which the system runs. For example, the Java Virtual Machine can be instrumented to generate events of interest. The advantage of this technique is that it does not require modification of the source code. Finally, it is also possible to run the system under the control of a debugger. In this case, breakpoints are set at locations of interest (e.g., entry and exit of a method). This technique has the advantage of modifying neither the source code nor the environment; however, it can considerably slow down the execution of the system.

A trace is usually saved in a text file that contains a sequence of lines in which each line represents an event. An example of this representation is given by Richner and Ducasse in [6]. Each line records the class of the sender, the identity of the sender, the class of the receiver, the identity of the receiver and the method invoked. The order of calls of the resulting call tree can be maintained in two ways, either each entry and exit of the method is recorded, which results in a very large trace file, or an integer is added to represent the nesting level of calls. In this case, we do not need to record the exit event of a method.

2.2. The Concept of Comprehension Unit

We define a *comprehension unit* as a distinct subtree of a call tree. More precisely, it is possible to partition the nodes of a tree into equivalence classes such that two nodes n_1 and n_2 of a tree t are equivalent if and only if the subtrees of t rooted at these nodes are isomorphic. In other words, they have the same structure and the nodes at corresponding places have identical labels. The set of all distinct subtrees of a given tree consists of a subtree from each equivalence class.

Another way of defining comprehension units is based on the fact that a rooted tree can always be transformed into its maximum compact form by representing isomorphic subtrees only once [7, 8]. This transformation results in a directed acyclic graph (DAG). Each subgraph of the DAG represents a distinct subtree of the tree. Figure 1 shows a tree structure and its corresponding ordered DAG. The graph shows that the trace contains six distinct subtrees (i.e., comprehension units for call trees); the subtree rooted at 'B' is repeated twice in the tree but only once in the corresponding graph.

A subset of the metrics presented in this paper is based on this concept of comprehension units. We hypothesize that in order to fully understand the trace, without any prior knowledge of the system, the analyst would need to investigate all the comprehension units that constitute it. It is important to notice that in practice, full comprehension would rarely be needed because the analyst will achieve his or her comprehension goals prior to achieving a full comprehension. Also, he or she will likely not need to try to understand the differences among the many comprehension units that only have slight differences. The concept of comprehension units therefore helps set an

upper bound on complexity of a trace in terms of the amount of work needed to understand it.

We deliberately chose to use the term ‘comprehension unit’. Both words in this choice of terminology have been criticised, but we intend to maintain our choices. The word ‘comprehension’ is used so as to emphasise the fact that we are indeed trying to consider the distinct parts of the trace that need to be *comprehended*. Also the term ‘unit’ indicates that these are items that can be counted; it is clear that the amount of comprehension required to understand the internals of two comprehension units will likely differ widely, but the same would be true of other ‘units’ in software engineering such as lines of code, methods, etc. In other words, it does not matter that the amount of understanding is different, and it does not matter that there will be other things (other than simply the subtrees) to understand. The key idea is that two equal comprehension units will not need understanding more than once.

It should be emphasized that the DAG corresponding to a call tree must be *ordered* in order to be able to restore the initial order of calls. In addition, it is important to note that the graph representation of a trace does not necessarily result in a loss of information associated with individual nodes of the tree such as timing information. What is needed is to augment the nodes of the graph with ordered collections that holds the information associated with each node.

2.3. Existing Dynamic Metrics

Research in software metrics has long been concentrated on the development of metrics that measure the complexity of static aspects of a program. These metrics include the number of lines of code, the cyclomatic complexity metric, coupling and cohesion, etc. Very little attention, however, has been placed on metrics that assess the complexity embedded in run-time information.

After a thorough review of existing literature, we have not encountered any work that focuses on measuring the various facets of a trace of routine calls. Most existing trace analysis techniques simply use the number of events invoked in a trace or the size of the trace file as an indicator of complexity. Hamou-Lhadj and Lethbridge survey these techniques in [1].

The studies most relevant to this paper are those conducted by Reiss et al. [9]. The authors propose converting a dynamic call tree into an ordered DAG (as discussed earlier) with the objective being to save storage space. They measure the compression gain of applying such a transformation to several traces. A subset of our metrics – those that involve what we call comprehension units – has the objective of measuring various properties of the ordered DAG resulting from transforming the call tree. The work we present in this paper with respect to this particular subset of metrics progresses beyond the above work in two ways. First, our primary goal is to use these metrics to facilitate the exploration of complex traces rather than simply saving space. Second, we define the metrics in a formal way. Finally, as we will describe in Section 3.3, we design the metrics to support various ways of assessing the similarity between subtrees of a call tree before turning them into a DAG; existing studies consider exact matches only.

Ducasse et al. proposed a metric-guided visualization technique, called run-time polymetric view, to help software maintainers understand various aspects of the execution

of object-oriented systems [10, 11]. Their approach allows a software maintainer to view statistical information about the system's execution such as the number of classes that are the most instantiated, the number of invoked methods for each class, and other characteristics of the system's execution. For this purpose, their technique relies on collecting aggregate data about the system's execution without having to save the traces. As such, one cannot use their technique to understand the flow of execution of the system. The metrics presented in this paper go beyond collecting statistics about the system's execution to form a complete framework for assessing the complexity of traces in order to build techniques for reducing this complexity and hence enabling software maintainer understand the flow of execution of the system.

Greevy et al. proposed a technique that can help software engineers understand how a software system evolves by studying the impact of change on the software features of the system [12]. To achieve this, they developed many metrics to characterize the role that a software component (more precisely, a class) plays in implementing the features of the system. Examples of these metrics include measuring the number of classes that do not participate in any of the features under analysis, the number of classes that participate in only one feature, the number of classes that participate in less than half of the features, etc. These metrics have not been designed to assess the complexity of a trace but rather to understand how a trace generated from exercising a particular feature changes during software evolution.

Dufour et al. presented an extensive list of metrics to measure various aspects of runtime information generated from Java programs [13]. Their metrics range from measuring the size of traces to more advanced features such as the degree to which polymorphism is used in a Java program. Although these metrics are interesting, they do not directly quantify the amount of information found in a trace of routine calls. However, they can help design techniques for reducing the complexity of traces. For example, one can hide the details of polymorphic methods if the need is to display the content of the trace at a high-level. Knowing the number of polymorphic calls invoked in a trace can help predict the gain resulting from applying this complexity reduction technique.

3. The Catalog of Metrics

In this subsection, we develop a set of metrics that can be used to measure useful properties of execution traces. We motivate the design of these metrics using the Goals/Questions/Metrics (GQM) model [14] as a framework.

The top level goal can be stated as follows:

To enable software engineers to more quickly understand the behaviour of a running system through the analysis of its execution traces.

The expression "behaviour of a running system" simply means the execution path taken when the system is provided with certain inputs. Understanding the path taken through a running system (i.e., dynamic analysis) can be essential to diagnose why something is being executed incorrectly, or is not being executed.

In this paper, we use the concept of software features as a means to exercise the system and generate execution traces. Perhaps the most common definition of a software feature is

by Eisenbarth et al, who describe it is a behavioural aspect of the system that represents a particular functionality, triggered by an external user [15]. These authors also discuss the relationship between a software feature, a scenario, and a computational unit. According to them a scenario is an instance of a software feature, where the user needs to specify a series of inputs to trigger the feature. A computational unit refers to the source code components that are executed by exercising the feature on the system. A feature is implemented by many computational units, and at the same time a given computational unit can be used in the implementation of multiple features. A trace can then be seen as a sequence of computational units invoked by exercising a specific feature, or more precisely, a particular scenario of this feature.

To achieve the above mentioned top-level goal, and in accordance with the GQM approach, we have designed questions that aim to characterize the effort required for understanding an execution trace. We present these questions along with the metrics that address them.

The metrics we present are designed to explore the possible space of size and complexity metrics. Not all of them are necessarily of equal value. Later in the paper we present case studies where we use some of them to actually measure some traces.

3.1. Call Volume Metrics

This category of metrics aims to answer the following related questions:

Q1a: How many nodes of any type does a trace contain in total?

Q1b: What is the ratio of the size of the processed trace to the size of the original trace?

The first sub question (Q1a) asks in simplest terms how much we would have to ‘look at’ if we had to examine the entire trace, or how much we would have to ‘process’ if we were to perform a computation on the entire trace. Answering this question establishes a baseline for other measurements we might want to take.

We can answer Q1a for any trace, including the original trace obtained from a trace-gathering tool, or a trace that has been processed in some way to shorten it, e.g. by treating a sequence of identical nodes as one node.

The second subquestion (Q1b) allows us to determine how much smaller a processed trace is, as compared to the original trace. This allows us to measure the effectiveness of the processing at reducing the volume of nodes that have to be examined.

The following metrics are designed to address these questions:

Full size [S]: The full size is the raw number of calls recorded in the trace; it is the number of nodes in the call tree without any of the manipulations described below, such as removing repetitions. Most existing trace analysis techniques rely on the size of traces as the main metric for complexity, e.g., [2, 3, 5, 6]. Full size forms a baseline for subsequent computations and will always be numerically larger than any of the other size metrics described below. Note that we say that it is the number of calls *recorded*. It is possible that some or many calls are not recorded. For example, a software engineer will often choose to not record invocations of private methods. He or she may also *sample* execution during a

certain time period only, or record only invocations in a certain subsystem. S is therefore always relative to the instrumentation used to gather the trace, but nevertheless represents the size of everything the software engineer has to work with in a given trace.

Size after removing contiguous repetitions [Scr]: This is the number of lines in the trace after removing contiguous repetitions due to loops and recursion. In other words, many identical calls are mapped into a single line by processing the trace. We refer to this process as the repetition-removal stage. Note that by identical, we are referring to identity between subtrees. At the leaf level of the call tree, the sequences AAABBB and AABBBB would result in $Scr=2$ (i.e., the nodes AB), whereas AABBBAA would result in $Scr=3$ (nodes ABA). And one level higher, $A_{CCDD}A_{CD}B_D^1$ and $A_{CDDD}B_{DDD}$ would both result in $Scr=5$ (nodes $A_{CD}B_D$), whereas $A_{CCD}B_{DD}A_{CD}$ would result in $Scr=8$ (nodes $A_{CD}B_D A_{CD}$).

According to our experience working with many execution traces, it seems that the number of lines after repetition removal is a much better indicator of the amount of work that will be required to fully understand the trace. This is due to the fact that the full size of a trace, S , is highly sensitive to the volume of input data used during the execution of the system. To understand the behaviour of an algorithm by analyzing its traces, it is often just as effective to study the trace after most of the repetitions are collapsed. And, in that case, studying a trace of execution over a very large data set would end up being similar to studying a trace of execution over a moderately sized data set.

Size, treating all called routines as a set [Sset]: This is the number of lines that remains after all repetitions and ordering are ignored. So for example $A_{CCDD}A_{CD}B_D$, $A_{CDDD}B_{DDD}$ and $A_{CCD}B_{DD}A_{CD}$ would all result in $Sset=5$ (nodes $A_{CD}B_D$).

Collapse ratio after removing contiguous repetitions [Rcr]: This is the ratio of the number of nodes after removing the contiguous repetitions from the full trace. $Rcr = Scr/S$.

Knowing that a program does a very high proportion of repetitive work (that Rcr is low) might lead program understanders to study possible optimizations, or to reduce input size. Knowing that Rcr is high would suggest that understanding the trace fully will be time consuming.

The Collapse Ratio is analogous to the notion of ‘compression ratio’ used in the context of data compression. However, we have carefully avoided using the term ‘compression’ since it causes confusion: The purpose of compression algorithms is to make data as small as possible; a decompression process is required to reconstitute the data in order to use it for any purpose. On the other hand, the purpose of collapsing is to make the data somewhat smaller by eliminating unneeded data, with the intent being that the result will be intelligible and useful without the need for ‘uncollapsing’.

Collapse ratio treating calls as a set [Rset]: Analogously to the above, this is $Sset/S$.

¹ We use the notation A_C to represent ‘A calls C’

3.2. Component Volume Metrics

This category of metrics is concerned with measuring the number of distinct components of various types (e.g., classes, packages, etc.) that are invoked during the execution of a particular scenario. The motivation behind the design of these metrics is that traces that crosscut many different components of the system are likely to be harder to understand than traces involving fewer components. In addition to this, it is very common that software engineers exploring traces map the trace components to their implementation counterpart in order to better understand a particular functionality. Knowing that a trace involves a large number of the system components would suggest that tools ought to support techniques that would easily allow this mapping. This family of metrics is inspired by the code coverage metrics used in software testing [16], which are used to describe the degree to which the source code of a program has been tested. In our case, we want to know the amount of code invoked in typical execution traces. More specifically, these metrics aim to investigate the following two related questions:

Q2a: How many system components are invoked in a given trace?

Q2b: What is the ratio of the number of components invoked in a trace to the total number of components of the system?

The term ‘component’ is very general. Separate metrics can be used to measure invocations of different types of components. In this paper, since the target systems that are analyzed are all programmed in Java then it may be useful to measure the following:

Number of packages [Np]: This is the number of distinct packages invoked in the trace. By ‘invoking’ a package we mean that some method in the package is called.

Number of classes [Nc]: This is the number of distinct classes invoked in the trace.

Number of methods [Nm]: This is the number of distinct methods that appear in the trace (i.e., irrespective of how many times each method is called).

It may be useful to create similar metrics based on other types of components, e.g. the number of threads involved.

The following ratios enable one to determine the proportion of a system that is covered by the trace. The more of a system covered, the more time potentially required to understand the trace, but the more complete an understanding of the entire system may be gained. Thus we may measure the following:

Ratio of number of trace packages to the number of system packages [Rpsp]: This is the ratio of the number of packages invoked in a trace to the number of packages of the instrumented system. More formally, if we let:

- NSp = The number of packages of the instrumented system
- Np = The number of packages as defined above

Then: $R_{psp} = N_p / N_{Sp}$

Ratio of number of trace classes to the number of system classes [Rcsc]: This is the ratio of the number of classes invoked in a trace to the number of classes of the instrumented system. This is computed analogously to Rpsp.

Ratio of number of trace methods to the number of system methods [Rmsm]: This is the ratio of the number of methods invoked in a trace to the number of methods of the instrumented system.

3.3. Comprehension Unit Volume Metrics

The comprehension unit volume category of metrics is computed from the graph representation of the trace. A similar metric was presented by Reiss et al. [9], except that Reiss et al. focused on the size of the file that contains the graph compared to the size of the trace file. We argued earlier that the file size can be misleading since it depends on the schema used to carry the trace data. In this study, we focus on the number of comprehension units of the trace which are represented as the nodes of the graph. In addition, Reiss et al. used only exact matches when comparing sequences of calls, whereas in this study, we believe that exact matching is not reflective of the trace size because of the number of repetitions in typical traces. As we did with the raw size metrics, we therefore propose using two matching criteria when building the graph, namely, ignoring the number of repetitions (cr) and treating the called routines as a set (set). In this category, we are interested in investigating the following questions:

Q3a: How many comprehension units exist in a trace using different matching criteria when comparing sequences of calls?

Q3b: What is the ratio of the number of comprehension units to the size of trace?

We suggest that metrics based on comprehension units will give a more realistic indication than call volume of the complexity of a trace.

Number of comprehension units [Scu_{sim}]: This family of metrics represents the number of comprehension units (i.e., distinct subtrees) of a trace. The number may vary depending on the way similarity among the subtrees is computed. The subscript ‘sim’ is used to refer to the similarity function used, which can be ‘exact’, ‘cr’, ‘set’ or some other function.

Considering exact matches only will result in a maximum number of comprehension units, Scu_{exact} . For example, consider the tree of Figure 2; the two subtrees rooted at ‘B’ differ because the number of contiguous repetitions of their subtrees differs.

If the number of contiguous repetitions is ignored when comparing two subtrees we use the subscript cr, and the resulting metric is Scu_{cr} , which will always be equal to or less than Scu_{exact} . We define Scu (unsubscripted) to mean Scu_{cr} , since in our experience, Scu_{cr} is a more useful basic measure of comprehension units than Scu_{exact} .

We can also compute Scu_{set} which compares the set of subtrees, ignoring all repetition (contiguous or not) and also ignoring order. Intuitively, it appears that $Scu_{set} \leq Scu_{cr} \leq Scu_{exact}$, since, when computing Scu_{set} , we not only ignore the number of repetitions when comparing sequences of calls but also ignore the order of calls. We did not attempt to prove

that $Scu_{set} \leq Scu_{cr} \leq Scu_{exact}$ since we do not think that this is important in this study. What is important is that further metrics (with other subscripts) can be computed by further varying the matching criteria. For example, we can compare two sequences of calls up to a certain depth, or by ignoring specific routines, etc. A discussion on how various matching criteria can be used to reduce the size of traces (by generating more patterns) can be found in [17, 18]. Most of these criteria require some sort of knowledge of the system under study. For example, the depth-limiting criterion assumes that the analyst has some understanding of the structure of the studied system. In this paper, we did not assume any knowledge of the system and hence did not attempt to compute these criteria.

Graph to tree ratio [Rgt_{sim}]: This family of metrics measures the ratio of the number of nodes of the ordered directed acyclic graph to the number of nodes of the trace. We expect to find a very low ratio since the acyclic graph factors out repetitions. More formally, let us consider:

- Scr = The size of a trace T after removing contiguous repetitions.
- Scu_{cr} = The size of the resulting graph after transforming T to a graph.

Then: $Rgt_{cr} = Scu_{cr}/Scr$.

Using another similarity function, we can also have $Rgt_{set} = Scu_{set}/Sset$.

If Rgt_{sim} is very low then this suggests that even a huge original trace might be relatively easy to understand.

3.4. Pattern-Related Metrics

This category of metrics is concerned with measuring the number of patterns that exist in a trace. We define a trace pattern as a comprehension unit that is repeated in a trace non-contiguously. Trace patterns are used by several trace analysis tools as means to uncover key domain concepts from the trace. The assumption is that the same sequence of calls that appear in various places of a trace might encapsulate some knowledge about the system (such as a particular aspect of an algorithm, etc.) that is worth investigating further. More about trace patterns and how they are used is presented by Jerding et al. in [3], and Systä et al. [19]. In this category of metrics, we, therefore, investigate the following questions:

Q4a: How many patterns exist in a trace using different matching criteria to compute the patterns?

Q4b: What is the ratio of the number of patterns to the number of comprehension units?

We present the following metrics:

Number of trace patterns [Nptt_{sim}]: This metric simply computes the number of trace patterns that are contained in a trace, given ‘sim’ as the similarity function (as before, if it is omitted, we assume cr). If Nptt is small this suggests that the complexity of the trace will be high.

Ratio of the number of patterns to the number of comprehension units [Rpcu]: This metric computes the ratio of the number of patterns to the number of comprehension units. In other words, we want to assess the percentage of comprehension units that are also

patterns. We will compute R_{pcu} for both matching criteria (i.e., cr and set). We therefore have two metrics:

- The ratio of the number of patterns to the number of comprehension units using the “ignore repetitions” matching criterion: $R_{pcu_{cr}} = N_{ptt_{cr}}/S_{cu_{cr}}$
- The ratio of the number of patterns to the number of comprehension units using the “ignore repetitions and ignore order” matching criterion: $R_{pcu_{set}} = N_{ptt_{set}}/S_{cu_{set}}$

3.5. Discussion

This section discusses the *necessity* and *sufficiency* of the four families of metrics presented in this paper. In general, the metric families are necessary since they each tell a different story about a trace: Leaving one or other family out would leave gaps. On the other hand, sufficiency of a metric is normally evaluated on the basis of whether the metric captures an underlying concept (in cases where the metric is a surrogate for an underlying concept). The metrics described here are generally very direct measures, as opposed to surrogates, so sufficiency is less of an issue.

The call volume metrics can be used to indicate the complexity of a trace by simply looking at its length (i.e., the number of calls). These metrics also take into account the length of the trace after removing contiguous repetition (Scr) as well as ignoring the order of calls ($Sset$). The advantage of these metrics is that they do not require extensive processing. Their limitation is that they do not consider the non-contiguous repetitions that occur in a trace. In other words, two identical subtrees that occur in a non-contiguous way will be counted twice. To address this issue, we presented the comprehension unit metrics (Scu , Rgt), which measure the content of a trace by factoring out all kinds of repetitions whether they are contiguous or not. We believe that these metrics are a better indicator of the work required to explore and understand the content of a trace since software engineers do not need to understand the same comprehension unit twice.

We presented the component volume metrics to assess the number of the system’s components invoked in a trace. These metrics are necessary given that software engineers will most likely need to map the content of a trace to the source code in order to get more information. Knowing that a trace, or part of trace, crosscuts several system components is an indicator that this mapping might be complex to perform. These metrics can be used independently from the other metrics presented in this paper.

Finally, trace pattern metrics are necessary in order to evaluate the number of patterns in a trace. As shown by Jerding et al. [4], software engineers often rely on exploring trace patterns so as to uncover the core behaviour embedded in a trace. These metrics can be used in combination with the comprehension unit metrics so as to compute the ratio of the number of patterns to the number of comprehension units. This ratio can be used to assess the extent to which a trace performs repetitive work.

4. Case Studies

4.1. Target Systems

We analyzed the execution traces of four Java software systems: Checkstyle² (ver. 3.3), Toad³ (ver. 2.0), Weka⁴ (ver. 3.0), and JHotDraw⁵ (ver. 5.1). Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. This is very useful to projects that want to enforce a coding standard. The tool allows programmers to create XML-based files to represent almost any coding standard. Toad is an IBM tool that includes a large variety of static analysis tools for monitoring, analyzing and understanding Java programs. Although these tools can be run as standalone tools, they can provide a much greater understanding of a Java application if they are used together. Weka is a collection of machine learning algorithms for data mining tasks. Weka contains tools for data pre-processing, classification, regression, clustering, and generating association rules. JHotDraw is a Java framework for creating drawing editors. It provides support for the creation of various types of geometric shapes ranging from simple shapes such as points, circles and lines to complex graphics. JHotDraw also supports advanced features for editing and manipulating shapes.

Table 2. Characteristics of the target systems

	Packages	Concrete Classes	Non-private Methods	KLOC
Checkstyle	43	671	5827	258
Toad	68	885	5082	203
Weka	10	147	1642	95
JHotDraw	11	161	1918	189

Table 2 summarizes certain static characteristics of the target systems relevant to computing the proposed trace metrics. For simplicity, we deliberately ignore the number of private methods (including private constructors) both in the table and in the traces: they are used to implement behaviour that will be localized within a class, so tracing them would provide minimal value in terms of comprehending the system, when compared to the added cost of a much larger trace. Abstract methods are also excluded since they have no presence at run time. Finally, the number of classes does not include abstract classes for the same reason.

4.2. Generating the Traces

We used our own instrumentation tool based on BIT (Bytecode Instrumentation Toolkit) [20] to insert probes at the entry and exit points of each system's non-private methods. Constructors are treated in the same way as regular methods. Traces are generated as the system runs, and are saved in text files. Except for JHotDraw, all other target systems can be invoked using both a graphical user interface and the command line. We

² <http://checkstyle.sourceforge.net>

³ <http://www.cs.waikato.ac.nz/ml/weka>

⁴ <http://alphaworks.ibm.com/tech/toad>

⁵ <http://www.jhotdraw.org>

favoured the command line approach over the GUI to avoid encumbering the traces with GUI components. We generated traces from the target systems by exercising the features described in Tables 3, 4, 5, and 6. Except for JHotDraw, where the selected features involve drawing and editing shapes, for all the other systems, we used the examples that come with the tools as input data.

We chose to exercise the systems using software features since it is well known that programmers often adopt an as-needed strategy when comprehending a large system and in which case they tend only to comprehend these portions of the program that are relevant to the affected feature [21]. In other words, we wanted to examine traces that are similar to the ones programmers will most likely generate for their needs, i.e., based on software features. We deliberately chose features, by carefully examining the documentation of each system, that cover different aspects of the system and that are not slight variations of each other. For the Weka system, for example, we exercised the system using various types of machine learning algorithms including classification, association, and clustering algorithms. This would allow us to better interpret the results by generating traces that cover different components of the system. A trace file contains the following information:

- Thread name,
- Full class name (e.g., weka.core.Instance),
- Method name, and
- A nesting level that maintains the order of calls.

We noticed that all the tools use only one thread, so we ignored the thread information.

Table 3. Checkstyle Features

Trace	Description
C-T1	Checks that each java file has a package.html.
C-T2	Checks that there are no import statements that use the .* notation.
C-T3	Restricts the number of executable statements to a specified limit.
C-T4	Checks for the use of whitespace.
C-T5	Checks that the order of modifiers conforms to the Java Language specification.
C-T6	Checks for empty blocks.
C-T7	Checks whether array initialization contains a trailing comma.
C-T8	Checks visibility of class members.
C-T9	Checks if the code contains duplicate portions of code.
C-T10	Restrict the number of &&, and ^ in an expression.

Table 4. Toad Features

Trace	Description
T-T1	Generates several statistics about the analyzed components.
T-T2	Detects and provides reports on uses of Java features, like native code interaction and dynamic reflection invocations, etc.
T-T3	Generates statistics in html format about unreachable classes and methods, etc.
T-T4	Specifies bytecode transformations that can be used to generate compressed version of the bytecode files.
T-T5	Generates the inheritance hierarchy graph of the analyzed components.
T-T6	Generates the call graph using rapid type analysis of the analyzed component.
T-T7	Generates an html file that contains dependency relationships among class files.

Table 5. Weka Features

Trace	Description
W-T1	Cobweb Clustering feature.
W-T2	EM Clustering feature.
W-T3	IBk Classification feature.
W-T4	OneR Classification feature.
W-T5	Decision Table Classification feature.
W-T6	J48 (C4.5) Classification feature.
W-T7	SMO Classification feature.
W-T8	Naïve Bayes Classification feature.
W-T9	ZeroR Classification feature.
W-T10	Decision Stump Classification feature.
W-T11	Linear Regression Classification feature.
W-T12	M5Prime Classification feature.
W-T13	Apriori Association feature.

Tabel 6. JHotDraw Features

Trace	Description
J-T1	Drawing circles, lines, and rectangles.
J-T2	Drawing polygons, rounded rectangles, and ellipses.
J-T3	Drawing shapes and adding connectors.
J-T4	Drawing shapes, adding text, simple editing operations such as changing colors.
J-T5	Drawing and editing of shapes: changing the shape size and moving and rotating shapes.

4.3. Analysing the Results of Applying the Metrics

The metrics presented in this paper have been implemented as a standalone application that will be integrated with our trace analysis tool, called SEAT (Software Exploration and Analysis Tool) [22]. The new version of the tool will be made available for download after the integration is completed. The collection of metrics resulted in a large set of data that we present in Appendix A. To help interpret and analyze the results, we show in Table 7 the average value of each metric.

The Call Volume Metrics:

As shown in Table 7, the number of calls invoked in traces of all systems is large as expected, in a range of hundreds of thousands of calls. Since we need at least two events to collect each call (the entry and exit of the traced routine), the size of some of the traces used in this case study goes beyond millions of events. This is particularly the case of JHotDraw where the average size of its traces approximates 1.35 million events (675974 calls). The removal of contiguous repetitions was expected to reduce significantly the size of raw traces, which was also confirmed by this case study. The collapse ratio of removing contiguous repetitions (Rcr) varies from one system to another, 5% (Toad), 13% (Weka), 7% (JHotDraw), and 46% (Checkstyle). However, the resulting traces of all four systems continue to contain thousands of calls, which is still high for the users of trace analysis tools. Therefore, simply removing repetition is a necessary trace compaction technique but far from sufficient.

The table shows that collapsing the traces by treating sequences of calls as a set (i.e., ignoring the repetitions and the order of calls) results in a considerably smaller traces, especially for Weka, JHotDraw, and Toad systems, where the average size (S_{set}) of the resulting traces is 2476 for Weka (which represents 2% of the original size), 11005 for JHotDraw (2% of the original size), and 5929 for Toad (3% of the original trace). Although this technique leads to excellent results, we consider it extreme since it goes against the definition of a trace of routine calls itself where the sequence and the order of calls is an important feature. In addition, it is difficult, without further studies, to assess which maintenance task can truly benefit from such extreme compaction scheme. For example, if the maintenance task at hand is fixing defects then ignoring the order of calls may hinder proper understanding of the code. On the other hand, we hypothesize that this technique can be valuable for maintenance tasks that require abstracting out the content of a trace at a very high-level where an analyst only needs to understand the main computations reflected in the trace. We, therefore, recommend that the ability to select different matching criteria be supported by a trace analysis tool.

Table 7. The average result of applying the metrics to the traces of the target systems

Metrics	Checkstyle	Toad	Weka	JHotDraw
Full Size (S)	74615	220409	145985	675974
Size after removing contiguous repetitions (Scr)	33293	10763	16147	48367
Size after removing contiguous repetitions, treating calls as a set (S_{set})	27749	5929	2476	11005
Collapse ratio after removing contiguous repetitions (Rcr)	46%	5%	13%	7%
Collapse ratio after removing contiguous repetitions, treating calls as a set (Rset)	39%	3%	2%	2%
Number of packages (N_p)	13	20	3	7
Ratio of number of trace packages to the number of system packages (R_{psp})	31%	29%	28%	64%
Number of classes (N_c)	111	173	14	125
Ratio of number of trace classes to the number of system classes (R_{csc})	16%	20%	10%	78%
Number of methods (N_m)	550	610	122	572
Ratio of number of trace methods to the number of system methods (R_{msm})	9%	12%	7%	30%
Number of comprehension units, using the ignore repetition criterion (Scu_{cr})	1103	826	186	1208
Number of comprehension units, treating calls as a set (Scu_{set})	1099	816	147	1155
Graph to tree ratio, using ignore repetitions criterion (Rgt_{cr})	8%	8%	3%	3%
Graph to tree ratio, treating calls as a set (Rgt_{set})	10%	14%	17%	12%
Number of trace patterns ($N_{ptt_{cr}}$)	177	204	52	269
Number of trace patterns, treating calls as a set ($N_{ptt_{set}}$)	174	167	35	187
Ratio of the number of patterns to the number of comprehension units ($R_{pcu_{cr}}$)	15%	25%	26%	22%
Ratio of the number of patterns to the number of comprehension units, treating calls as a set ($R_{pcu_{set}}$)	15%	20%	22%	16%

The Component Volume Metrics:

Our analysis shows that the number of distinct packages invoked in all traces is consistent across Weka, Toad, and Checkstyle systems from which we generated relatively small to medium-sized traces, between 28% and 31% of the system packages. However, the number of packages invoked in traces of JHotDraw counts for 64% of the total number of packages. Most of these packages implement graphical user interface capabilities, which are invoked in all features of JHotDraw. We conducted further analysis to understand the package frequency distribution in traces of all target systems. We did not attempt to replicate this analysis using classes and methods for simplicity reasons.

The results of this analysis, reported in Appendix B, show that in each system there are one or two packages which are responsible of more than 60% and sometimes 80% of the set of calls invoked in a trace (we use the number of calls after removing contiguous repetitions, Scr). For example, the package ‘antlr’ constitutes almost 84% of all total invocations of Checkstyle traces. ‘antlr’ stands for ANother Tool for Language Recognition⁶ and is a utility library that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions of various programming languages such as Java. It is used by Checkstyle to build a representation of the Java systems that need to be processed. It is more a utility library than a package that performs actual checks for design compliance, which is the main functionality of Checkstyle. Similarly, the packages ‘com.ibm.toad.cfparse’ and ‘com.ibm.toad.utils’ contribute to more than 63% of the size of Toad traces (Scr). The cfparse package stands for Class File Parser and is used by all Toad features to parse Java class files. The package com.ibm.toad.utils as indicated by its name groups system-scope utilities used by the various components of Toad. These two packages are clearly low-level implementation details that encumber the content of traces. The package ‘weka.core’ of the Weka system generates almost 87% of total invocations. This package implements general purpose operations for manipulating datasets used in the implementation of the various data mining algorithms of Weka. Finally, the packages CH.ifa.draw.standard and CH.ifa.draw.util of the JHotDraw system are responsible of 80% of the total number of calls invoked in JHotDraw traces. These packages contain standard data structures and utility functions such as trees (implemented in the CH.ifa.draw.standard.QuadTree class), enumerations (class CH.ifa.draw.standard.FigureEnumerator), geometric operations (class CH.ifa.draw.util.Geom), etc.

These results indicate that abstraction techniques based on the removal of utility components can be an excellent way to reduce significantly the size of traces. It is expected that a tool that supports these techniques should allow enough flexibility to the users to determine a threshold above which a component is considered as a utility, since what constitutes a utility for a software engineer may be an important component for someone else depending on the user’s experience of the system and the software maintenance at hand.

The challenge with utility detection techniques is that, as the system undergoes several maintenance cycles, it becomes hard to distinguish the utility components from non-

⁶ <http://www.antlr.org>

utilities. There is normally no program language syntax to designate a utility, and they may or may not be given names that make it clear they are intended to be utilities. An effective tool should therefore support the automatic (or semi-automatic) detection of utilities. An example of a utility detection approach is the one we proposed in our previous work [23], and where we showed how fan-in analysis can be employed to detect utility routines. We argued that a utility component is a component that is used by many other components of the system and therefore it is expected to have a high fan-in. Based on that, we devised a utilityhood metric that ranks routines according to their fan-in value. Although the approach showed promising results, it needs to be fine tuned to detect utility routines defined at the package levels and not only at the system scope. The metric also suffers from the drawback that it requires an entire static call graph of the system under study.

Comprehension Units Metrics

Table 7 shows the average number of comprehension units (Scu_{cr}) and the ratio achieved by transforming traces into acyclic graphs (Rgt_{cr}), which varies from 3% to 8% (i.e., a reduction of 97% to 92% of the size of a trace). These results demonstrate the effectiveness of transforming the traces into ordered directed acyclic graphs, which confirms the fact that a trace analysis tool should never save traces as tree structures. It should be emphasized that if the exact match is used then this transformation is lossless, i.e., the reverse transformation will result in the initial tree.

As expected, Scu_{set} , the average number of comprehension units in all traces treating the called routines as a set (i.e., ignoring the repetitions and the order of calls) is less than the number of comprehension units resulting from simply ignoring the number of repetitions (Scu_{rc}). However, the difference is not large, except for in the case of Weka where the average number of comprehension units treating the called routines as a set is $Scu_{set} = 147$ compared to $Scu_{rc} = 186$ (a ratio of 31%). For all other systems, the ratio of Scu_{set} to Scu_{rc} is less than 10%.

We analyzed sample traces of the target systems to understand the reasons behind this discrepancy, we found that Weka uses a small set of methods belonging to the core utility package (e.g., `weka.core.Instances.numAttributes`, `weka.core.Instances.instance`) to do most of the work related to the handling of data used by the traced machine learning algorithm. As a result, these methods appear extensively in all levels of the trace and the order in which they are called varies considerably. This does not appear to be the case of Checkstyle, Toad, and JHotDraw which use a very large set of methods for the handling of the various aspects of the traced computations, reducing the chances of having always the same methods repeated everywhere.

Another observation regarding the results of the comprehension unit volume metrics is that the number of distinct methods (Nm) invoked in the traces of all four systems is significantly smaller than the number of comprehension units contained in these traces even after applying two matching criteria (i.e., `cr` and `set`). This means that many subtrees still exhibit a certain degree of similarity. If each routine has generated the same sequence of calls whenever it is called then the number of comprehension units will be equal to the number of distinct routines invoked in the trace, since a comprehension unit consists of a distinct subtree of a trace. To further reduce the number of comprehension units, we need to investigate additional matching criteria by varying the ‘sim’ function. Based on the results of the component volume metrics shown in the previous section, that clearly show

that utility routines are the reason behind having large traces, it may be useful to compare subtrees by ignoring the utility components they contain when computing the number comprehension units. In our previous study on exploring the concept of utilities, we have concluded that utilities are the components that are called from different places within a system, as such, they tend to appear just about anywhere in a trace [23], depending on whether the calling routine needs to perform utility work or not. As a result, many sequences of calls triggered by the same routine may differ simply due to the presence or absence of utility routines.

Finally, it should be noted that using a variety of matching criteria comes with a very challenging set of research questions. One of the issues is that it is difficult to predict how these criteria should be combined. Another problem is that it is not really clear how various combinations can help with specific maintenance tasks such as debugging, adding a new feature, etc.

Patterns Related Metrics

We noticed that many of the comprehension units of the analyzed traces are repeated non-contiguously, which qualify them as trace patterns. The average number of patterns using both matching criteria (i.e., cr and set) is very small compared to the number of comprehension units. For example, the use of the “ignore number of repetitions” criterion resulted in 177 patterns on average for Checkstyle, which represents 15% of the number of comprehension units. Toad traces contain on average 204 (25% of the number of comprehension units), Weka traces contain on average 52 patterns (26% of the number of comprehension units), and finally JHotDraw traces contain on average 269 patterns (22% of the number of comprehension units). This means that this technique alone can be ineffective for rapid understanding of the content of a trace, since a large number of comprehension units (which do not belong to any patterns) are left to be explored. For example, knowing that only 15% of Checkstyle’s comprehension units are patterns means that the analyst will still need to deal with the remaining 85% comprehension units. In this context, an effective trace analysis tool must rely on trace abstraction strategies other than pattern detection techniques. As discussed earlier, initially removing utilities may be a good strategy to enable further grouping of similar sequences of calls into patterns. We, therefore, suggest that a combination of utility removal techniques and pattern detection would be needed for effective abstraction of trace content.

5. Conclusion and Future Direction

In this paper, we developed a family of complexity metrics for the analysis of large execution traces of method calls. The metrics fall into four categories: 1) those that measure the number, or volume, of calls, taking into account various abstractions such as removing duplicates and sequence information; 2) those that measure the number of various types of components a program run interacted with; 3) those that measure comprehension units, which are distinct subtrees in the call graph, and 4) those related to trace patterns, which are comprehension units duplicates of which appear in non-contiguous parts of a trace.

Tools for program comprehension, as well as for other maintenance tasks, can use the metrics in an automatic or semi-automatic way to:

- Guide the generation of traces that have a desired level of complexity – if the metrics show a trace is too complex or too simple, the system’s input can be adjusted.
- Measure the complexity of various *parts* of a trace in order to guide the user to examine the most complex parts of a trace.
- Enable decisions about at what level of abstraction various parts of a trace should be displayed, as well as what should be hidden and what should be shown.
- Help estimate the complexity of a maintenance task, under the assumption that more complex traces will require longer to understand and possibly longer to fix correctly.

In addition, researchers and tool builders can use the metrics to investigate further techniques for reducing the complexity of traces and for understanding the nature of software in general.

To demonstrate the usefulness of the metrics, we presented case studies where we analysed traces of four different Java systems. By usefulness, we mean measuring various aspects of a trace in such a way that the resulting measurements can serve the above objectives. A further analysis of usefulness would be to observe and measure their use by software engineers in practice; we leave this to future work.

One of our metrics, R_{cr} , shows that when we remove contiguous repetitions, the size of the trace is reduced to between 5% and 46% of the original size. However, the resulting traces continue to have thousands of calls, which makes this basic type of collapsing necessary but not sufficient. A considerably better collapse ratio is reached (39% average collapse ratio for Checkstyle, 3% for Toad, 2% for Weka, and 2% for JHotDraw) when treating the called routines as a set (i.e., ignoring the repetitions and the order of calls).

In our case studies, the traces crosscut up to 30% of a system’s packages, 22% of a system’s classes and 14% of a system’s methods for Checkstyle, Toad, and Weka from which we generated small to medium sized traces. On the other hand, traces of JHotDraw, which are considerably larger than traces of the other systems crosscut 65% of the number of the system’s packages, 78% of the system’s classes, and 30% of the system’s methods. In all examined systems there are one or two packages that are the most responsible of the lengthy size of traces. Knowing this information will allow tools to adjust the amount of information displayed by either hiding the invocations made to these packages or using other visualization techniques.

In addition, the number of comprehension units, Scu_{cr} , in all studied traces is considerably smaller than the initial size of traces, S . This strengthens the assertion that traces should best be represented as ordered DAGs to allow tools to scale up to handle large traces. This is consistent with the results of the study conducted by Larus et al. [24]. When we applied the set matching criterion, we obtained two different results. For Checkstyle and Toad, the number of comprehension units Scu_{set} is only slightly smaller than Scu_{cr} , whereas for Weka, we observed that Scu_{set} is 31% less than Scu_{cr} . Taking a deeper look at the trace content of all systems, we found that unlike Checkstyle and Toad, Weka relies on a very small set of methods that belong to the core utility package to do most of the work related to data processing; this is an important aspect of Weka since it is a machine learning system. As a result, these methods appear almost everywhere in the traces, which explains why when treating the called routines as a set, we obtain reduction in the number of comprehension units.

Our case studies also showed that the number of comprehension units in our traces, using both matching criteria, Scu_{cr} and Scu_{set} , is higher than the number of distinct methods invoked, Nm . This means that there are still opportunities for further reduction of the size of the trace using other matching criteria. This degree of variability can be measured using Nm/Scu , which converges to zero if the variability is high. It equals one if there is no variability, i.e., $Nm = Scu$.

Another finding of this paper is that the number of patterns in all traces can be relatively high (some traces have over 250 patterns). In addition, the number of patterns is significantly smaller than the number of comprehension units using both matching criteria, i.e., cr and sim . This is a reason of concern for tools that rely exclusively on pattern detection algorithms since an analyst will need to understand the comprehension units that do not belong to any pattern. Clearly, pattern detection techniques alone are not sufficient for reducing the size of traces, unless advanced matching criteria are used. In this paper, we proposed using the removal of utilities as a matching criterion to further group similar calls into patterns. As future work, we will implement the above metrics in our trace analysis tool, called SEAT (Software Exploration and Analysis Tool) [22] and investigate how they can be used by software engineers to effectively explore the content of large traces. We will also investigate new trace abstraction techniques, especially the ones based on removing utility components as discussed in the paper.

6. References

- [1]. Hamou-Lhadj and T. Lethbridge, "A Survey of Trace Exploration Tools and Techniques", *In Proc. of the 14th IBM Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 42-55, 2004.
- [2]. T. Systä, "Dynamic Reverse Engineering of Java Software", *In Proc. of the ECOOP Workshop on Experiences in Object-Oriented Reengineering*, pp. 174-175, 1999.
- [3]. D. Jerding, and S. Rugaber, "Using Visualization for Architecture Localization and Extraction", *In Proc. of the 4th Working Conference on Reverse Engineering*, pp. 219-234, 1997.
- [4]. D. Jerding, J. Stasko, and T. Ball, "Visualizing Interactions in Program Executions", *In Proc. of the 19th International Conference on Software Engineering*, pp. 360-370, 1997.
- [5]. W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, "Visualizing the Execution of Java Programs", *In Proc. of the International Seminar on Software Visualization*, pp. 151-162, 2002.
- [6]. T. Richner, and S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles", *In Proc. of the 18th International Conference on Software Maintenance*, pp. 34-43, 2002.
- [7]. J. P. Downey, R. Sethi and R.E. Tarjan, "Variations on the Common Subexpression Problem", *Journal of the ACM*, 27(4), pp. 758-771, 1980.
- [8]. P. Flajolet, P. Sipala, and J-M. Steyaert, "Analytic Variations on the Common Subexpression Problem", *In Proc. of the 7th International Colloquium on Automata, Languages and Programming*, pp. 220-234, 1990.

- [9]. S. P. Reiss, M. Renieris, "Encoding program executions", *In Proc. of the 23rd International Conference on Software Engineering*, pp. 221-230, 2001.
- [10]. S. Ducasse, M. Lanza, R. Bertuli, "High-level Polymetrics Views of Condensed Runtime Information", *In Proc. of the 8th European Conference on Software Maintenance and Reengineering*, pp. 309 – 318, 2004.
- [11]. M. Lanza and S. Ducasse, "Polymetric Views — A Lightweight Visual Approach to Reverse Engineering", *IEEE Transactions on Software Engineering*, 29(9), pp. 782–795, 2003.
- [12]. O. Greevy, S. Ducasse "Analyzing Software Evolution through Feature Views", *Journal of Software Maintenance and Evolution: Research and Practice*, 18(6), pp. 425-456, 2007.
- [13]. B. Dufour, K. Driesen, L. Hendren and C. Verbrugge, "Dynamic Metrics for Java", *In Proc. of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 149 – 168, 2003.
- [14]. V. Basili, G. Caldiera, and H.D. Rombach, "Goal Question Metric Approach", *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., pp. 528-532, 1994.
- [15]. T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, 29(3), pp. 210 – 224, 2003.
- [16]. M. L. Hutcheson, "Software Testing Fundamentals: Methods and Metrics", Wiley, 2003.
- [17]. W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, "Execution Patterns in Object-Oriented Visualization", *In Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, pp. 219-234, 1998.
- [18]. A. Hamou-Lhadj, and T. Lethbridge, "An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls", *In Proc. of the 1st ICSE International Workshop on Dynamic Analysis*, pp. 33-36, 2003.
- [19]. T. Systä, K. Koskimies, and H. Müller, "Shimba – An Environment for Reverse Engineering Java Software Systems", *Software Practice & Experience*, 31 (4), pp. 371-394, 2001.
- [20]. H. B. Lee, B. G. Zorn, "BIT: A tool for Instrumenting Java Bytecodes", *In Proc. of the USENIX Symposium on Internet Technologies and Systems*, pp. 73-82, 1997.
- [21]. N. Wilde , M. Buckellew, H. Page , V. Rajlich , L. Pounds, "A comparison of methods for locating features in legacy software", *Journal of Systems and Software*, 65(2), pp.105-114, 2003.
- [22]. A. Hamou-Lhadj, and T. Lethbridge, and L. Fu, "SEAT: A Usable Trace Analysis Tool", *In Proc. of the International Workshop on Program Comprehension*, pp. 157-160, 2005.
- [23]. A. Hamou-Lhadj, and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *In Proc. of the International Conference on Program Comprehension*, pp. 181-190, 2006.

- [24]. J. R. Larus, “Whole program paths”, *In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 259-269, 1999.

Appendix A. Detailed Results of Applying the Metrics to Traces of the Target Systems

Table A1. CheckStyle Statistics

Checkstyle	Call Volume Metrics					Component Volume Metrics						Comprehension Units and Patterns Metrics							
Traces	S	Scr	Sset	Rcr	Rset	Np	Rpsp	Nc	Rcsc	Nm	Rmsm	Scucr	Scuset	Rgtcr	Rgtset	Npttcr	Npttset	Rpcu	Rpcuset
C-T1	84040	37957	31906	45%	38%	14	33%	114	17%	590	10%	1261	1254	3%	4%	221	219	18%	17%
C-T2	81052	35969	30122	44%	37%	13	30%	109	16%	540	9%	1106	1103	3%	4%	176	174	16%	16%
C-T3	81639	36123	30460	44%	37%	13	30%	110	16%	561	10%	1153	1149	3%	4%	186	183	16%	16%
C-T4	84299	37062	31031	44%	37%	14	33%	117	17%	590	10%	1191	1182	3%	4%	194	188	16%	16%
C-T5	80393	35455	29813	44%	37%	13	30%	106	16%	547	9%	1098	1095	3%	4%	177	176	16%	16%
C-T6	81550	36087	30302	44%	37%	14	33%	113	17%	562	10%	1125	1121	3%	4%	178	178	16%	16%
C-T7	89085	41414	34393	46%	39%	14	33%	148	22%	700	12%	1455	1448	4%	4%	240	233	16%	16%
C-T8	83106	37163	31222	45%	38%	14	33%	114	17%	586	10%	1234	1229	3%	4%	206	203	17%	17%
C-T9	1013	618	488	61%	48%	9	21%	70	10%	276	5%	306	306	50%	63%	19	15	6%	5%
C-T10	79969	35083	29634	44%	37%	13	30%	105	16%	521	9%	1071	1068	3%	4%	168	168	16%	16%
Max	89085	41414	34393	61%	48%	14	33%	148	22%	700	12%	1455	1448	50%	63%	240	233	18%	17%
Min	1013	618	488	44%	37%	9	21%	70	10%	276	5%	306	306	3%	4%	19	15	6%	5%
Average	74615	33293	27749	46%	39%	13	31%	111	16%	550	9%	1103	1099	8%	10%	177	174	15%	15%

Table A2. Toad Statistics

Toad	Call Volume Metrics					Component Volume Metrics						Comprehension Units and Patterns Metrics							
Traces	S	Scr	Sset	Rcr	Rset	Np	Rpsp	Nc	Rcsc	Nm	Rmsm	Scucr	Scuset	Rgtcr	Rgtset	Npttcr	Npttset	Rpcu	Rpcuset
T-T1	219507	10409	5724	5%	3%	20	29%	172	19%	615	12%	827	820	8%	14%	199	162	24%	20%
T-T2	218867	10141	5597	5%	3%	20	29%	169	19%	592	12%	794	787	8%	14%	193	153	24%	19%
T-T3	226026	13132	7142	6%	3%	20	29%	191	22%	704	14%	971	956	7%	13%	246	199	25%	21%
T-T4	220438	10811	5972	5%	3%	20	29%	177	20%	626	12%	835	828	8%	14%	206	170	25%	21%
T-T5	218681	10002	5547	5%	3%	20	29%	164	19%	558	11%	754	746	8%	13%	184	155	24%	21%
T-T6	219171	10394	5724	5%	3%	20	29%	170	19%	605	12%	816	809	8%	14%	204	168	25%	21%
T-T7	220170	10450	5797	5%	3%	20	29%	165	19%	568	11%	782	768	7%	13%	197	159	25%	21%
Max	226026	13132	7142	6%	3%	20	29%	191	22%	704	14%	971	956	8%	14%	246	199	25%	21%
Min	218681	10002	5547	5%	3%	20	29%	164	19%	558	11%	754	746	7%	13%	184	153	24%	19%
Average	220409	10763	5929	5%	3%	20	29%	173	20%	610	12%	826	816	8%	14%	204	167	25%	20%

Table A3. Weka Statistics

Weka	Call Volume Metrics					Component Volume Metrics						Comprehension Units and Patterns Metrics							
Trace	S	Scr	Sset	Rcr	Rset	Np	Rpsp	Nc	Rcsc	Nm	Rmsm	Scucr	Scuset	Rgtcr	Rgtset	Npttcr	Npttset	Rpcu	Rpcuset
W-T1	193165	4081	1382	2%	1%	2	20%	10	7%	75	5%	89	89	2%	6%	23	21	26%	24%
W-T2	66645	6747	143	10%	0%	3	30%	10	7%	64	4%	66	66	1%	46%	10	7	15%	11%
W-T3	39049	7760	590	20%	2%	2	20%	12	8%	114	7%	177	121	2%	21%	28	23	16%	19%
W-T4	28139	4914	840	17%	3%	2	20%	10	7%	116	7%	159	125	4%	15%	41	30	26%	24%
W-T5	157382	26714	3116	17%	2%	3	30%	19	13%	188	11%	293	231	1%	7%	81	69	28%	30%
W-T6	97413	25722	12206	26%	13%	3	30%	23	16%	181	11%	340	272	1%	2%	97	75	29%	28%
W-T7	283980	21524	1348	8%	0%	3	30%	15	10%	131	8%	168	153	1%	11%	49	32	29%	21%
W-T8	37095	6700	525	18%	1%	3	30%	13	9%	114	7%	141	120	2%	23%	37	23	26%	19%
W-T9	12395	637	333	5%	3%	2	20%	10	7%	93	6%	96	95	15%	29%	21	17	22%	18%
W-T10	43681	6427	480	15%	1%	2	20%	10	7%	97	6%	118	103	2%	21%	31	23	26%	22%

W-T11	403704	34447	1161	9%	0%	4	40%	16	11%	147	9%	220	171	1%	15%	55	32	25%	19%
W-T12	378344	54871	9678	15%	3%	5	50%	26	18%	194	12%	431	272	1%	3%	151	87	35%	32%
W-T13	156814	9368	386	6%	0%	2	20%	9	6%	72	4%	123	91	1%	24%	46	14	37%	15%
Max	403704	54871	12206	26%	13%	5	50%	26	18%	194	12%	431	272	15%	46%	151	87	37%	32%
Min	12395	637	143	2%	0%	2	20%	9	6%	64	4%	66	66	1%	2%	10	7	15%	11%
Average	145985	16147	2476	13%	2%	3	28%	14	10%	122	7%	186	147	3%	17%	52	35	26%	22%

Table A4. JHotDraw Statistics

JHotDraw	Call Volume Metrics					Component Volume Metrics						Comprehension Units and Patterns Metrics							
	S	Scr	Sset	Rcr	Rset	Np	Rpsp	Nc	Rcsc	Nm	Rmsm	Scucr	Scuset	Rgtr	Rgtset	Npttr	Nptset	Rpcu	Rpcuset
J-T1	693439	57794	7544	8%	1%	7	64%	118	73%	488	25%	1092	1035	2%	14%	235	141	22%	14%
J-T2	694912	38987	5517	6%	1%	7	64%	119	74%	506	26%	920	915	2%	17%	190	131	21%	14%
J-T3	692101	66210	16925	10%	2%	7	64%	127	79%	647	34%	1640	1548	2%	9%	376	263	23%	17%
J-T4	694419	53395	16634	8%	2%	7	64%	129	80%	659	34%	1416	1346	3%	8%	327	240	23%	18%
J-T5	605000	25451	8407	4%	1%	7	64%	134	83%	561	29%	973	933	4%	11%	217	159	22%	17%
Max	694912	66210	16925	10%	2%	7	64%	134	83%	659	34%	1640	1548	4%	17%	376	263	23%	18%
Min	605000	25451	5517	4%	1%	7	64%	118	73%	488	25%	920	915	2%	8%	190	131	21%	14%
Average	675974	48367	11005	7%	2%	7	64%	125	78%	572	30%	1208	1155	3%	12%	269	187	22%	16%

Appendix B. Analysis of the Package Frequency in all Traces of the Target Systems

Table B1. The contribution of Checkstyle packages to the size of traces

Packages	Number of invocations	Percentage
com.puppycrawl.tools.checkstyle.checks.duplicates	11	0.00%
com.puppycrawl.tools.checkstyle.checks.metrics	32	0.01%
com.puppycrawl.tools.checkstyle.checks.javadoc	49	0.01%
com.puppycrawl.tools.checkstyle.checks.blocks	38	0.01%
com.puppycrawl.tools.checkstyle.checks.design	89	0.03%
com.puppycrawl.tools.checkstyle.checks.imports	90	0.03%
com.puppycrawl.tools.checkstyle.checks.whitespace	99	0.03%
com.puppycrawl.tools.checkstyle.checks.sizes	122	0.04%
com.puppycrawl.tools.checkstyle.checks	175	0.05%
Org.apache.commons.logging	325	0.10%
com.puppycrawl.tools.checkstyle.checks.coding	549	0.16%
Org.apache.commons.cli	640	0.19%
Org.apache.commons.beanutils.converters	941	0.28%
Org.apache.commons.logging.impl	1236	0.37%
Org.apache.commons.collections	1779	0.53%
Org.apache.regexp	3014	0.91%
Org.apache.commons.beanutils	4420	1.33%
antlr.collections.impl	4536	1.36%
com.puppycrawl.tools.checkstyle	6408	1.92%
com.puppycrawl.tools.checkstyle.grammars	11430	3.43%
com.puppycrawl.tools.checkstyle.api	19087	5.73%
Antlr	277861	83.46%
Total	332931	100%

Table B2. The contribution of Toad packages to the size of traces

Packages	Number of invocations	Percentage
com.ibm.toad.jan.construction.builders.ehgbuilder	70	0.09%
com.ibm.toad.jan.lib	308	0.41%
com.ibm.toad.analyzer	536	0.71%
com.ibm.toad.jan.construction.builders.rgimpl	1061	1.41%
com.ibm.toad.cfparse.utils	1099	1.46%
com.ibm.toad.jan.lib.cgutils	1239	1.64%
com.ibm.toad.jan.lib.rgutils	1274	1.69%
com.ibm.toad.jan.jbc.utils	1309	1.74%
com.ibm.toad.jan.construction.builders.cgbuilder	1309	1.74%
com.ibm.toad.jan.construction.builders.cgbuilder.cgimpl	1333	1.77%
com.ibm.toad.jan.construction.builders.javainfoimpl	1386	1.84%
com.ibm.toad.jan.construction.builders.hgimpl	1383	1.84%
com.ibm.toad.jan.construction.builders	1589	2.11%
com.ibm.toad.jan.construction	1712	2.27%
com.ibm.toad.cfparse.attributes	2303	3.06%
com.ibm.toad.jan.lib.hgutils	2650	3.52%
com.ibm.toad.jan.coreapi	2915	3.87%
com.ibm.toad.utils	22816	30.28%
com.ibm.toad.cfparse	25501	33.85%
com.ibm.toad.jan.jbc	3546	4.71%
Total	75339	100%

Table B3. The contribution of Weka packages to the size of traces

Packages	Number of invocations	Percentage
weka.clusterers	463	0.22%
weka.estimators	1101	0.52%
weka.associations	1605	0.76%
weka.filters	3014	1.44%
weka.classifiers	6380	3.04%
weka.classifiers.j48	7256	3.46%
weka.classifiers.m5	8612	4.10%
weka.core	181481	86.46%
Total	209912	100%

Table B4. The contribution of JHotDraw packages to the size of traces

Packages	Number of invocations	Percentage
CH.ifa.draw.standard	167594	68%
CH.ifa.draw.util	29092	12%
CH.ifa.draw.figures	25001	10%
CH.ifa.draw.application	14198	6%
CH.ifa.draw.framework	6977	3%
CH.ifa.draw.contrib	2660	1%
CH.ifa.draw.samples.javadraw	2064	1%
Total	247586	100%

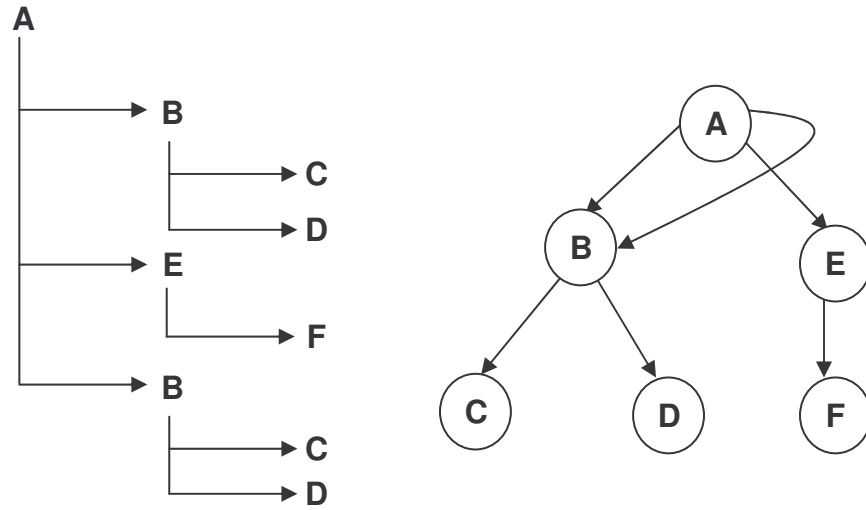


Figure 1. The graph representation of a trace is a better way to spot the number of distinct subtrees it contains

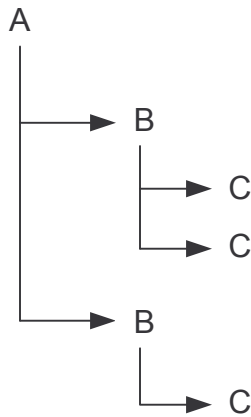


Figure 2. The two subtrees rooted at 'B' can be considered similar if the number of repetitions of 'C' is ignored