Compression Techniques to Simplify the Analysis of Large Execution Traces[*]

Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge
University of Ottawa
SITE, 150 Louis Pasteur
Ottawa, Ontario, Canada, K1N 6N5
{ahamou, tcl}@site.uottawa.ca

Abstract

Dynamic analysis consists of analyzing the behavior of a software system to extract its proprieties. There have been many studies that use dynamic information to extract high-level views of a software system or simply help software engineers to perform their daily maintenance activities more effectively. One of the biggest challenges that such tools face is to deal with very large execution traces. By analyzing the execution traces of the software systems we are working on, we noticed that they contain many redundancies that can be removed. This led us to create a comprehension-driven compression framework that compresses the traces to make them more understandable. In this paper, we present and explain its components. The compression framework is reversible that is the original trace can be reconstructed from its compressed version. In addition to that, we conducted an experiment with the execution traces of two software systems to measure the gain attained by such compression.

1. Introduction

Dynamic analysis is used in a large variety of applications in software engineering such as software testing, software performance analysis and, lately, reverse engineering and program comprehension. Generally speaking, it consists of analyzing the behavior of a software system to extract its proprieties.

Dynamic information is typically expressed in the form of execution traces. It is extracted by instrumenting the code or modifying the execution environment. One of the main advantages of run-time information is its precision [1]. In this way, a software engineer can instrument exactly the source code that needs to be analyzed and ignore the other parts. In addition to that, dynamic analysis becomes crucial when it comes to understand the behavior of object-oriented systems. Indeed, dynamic binding makes it almost impossible to fully understand such systems by merely performing static analysis of the source code [7, 8].

In reverse engineering in general and program comprehension in particular, there have been many studies that use dynamic information to extract high-level views of a software system or simply help software engineers to perform their daily maintenance activities more effectively [4, 6, 7, 10, 11, 12]. However, because of the considerable size of the data gathered by performing dynamic analysis, there is a need to create efficient tools to assist software engineers. One of the challenges that such tools face is to deal with very large execution traces.

By analyzing the execution traces of the software systems we are working on, we noticed that they contain many redundancies that can be removed. This led us to create a *comprehension-driven* compression framework. In this paper, we present and explain its components. The compression framework is reversible – that is the original trace can be reconstructed from its compressed version. In addition to that, we experimented with two software systems to estimate the gain attained by such compression.

In information theory, there are many known data compression techniques and algorithms. They are all based on the same principle, which is compressing data by removing redundancy [2]. Based on this, the key concept of data compression is to "assign short codes to common events (symbols or phrases) and long codes to rare events" [2]. Even though these techniques produce very good results, the information, once compressed, is no longer readable by humans. So such algorithms certainly will not help in program comprehension. The framework for compression presented in this paper allows software engineers to better comprehend an execution trace by significantly reducing its size and at the same time

[*] This research is sponsored by the Consortium for Software Engineering Research (CSER) and supported by Mitel Networks and NSERC

keeping its content readable, hence the name comprehension-driven compression framework.

There are two types of compression techniques [2]: lossless and lossy. Lossless compression techniques achieve compression in such a way that the exact original data set can be reconstructed from its compressed version. Lossy compression techniques produce better compression by dropping some information – but the original data set cannot be exactly reproduced. This paper is generally about lossless compression. However, in program comprehension, lossy compression of execution traces can be useful as well. We leave this point as future work.

This paper focuses on traces of procedure calls, since many of today's legacy systems were developed using the procedural paradigm. This is the case, for example, with the large telecommunication system we are working on, provided to us by the company that sponsors our research. However, our framework can also be applied to object-oriented systems by considering method invocations. First we show that any procedure-call trace can be represented by a rooted ordered labeled tree. Then, we reduce the problem of detecting redundancies to the common subexpression problem described in [3, 5, 9]; we introduce this and explain it in depth in section 2. The common subexpression problem consists of transforming a rooted tree to an acyclic graph in such a way all the isomorphic subtrees are represented only once.

The rest of this paper is organized as follows. In section 2, we explain the common subexpression problem and a linear algorithm that solves it. In section 3, we present the compression framework and its components. In section 4, we present the results of applying the framework to the execution traces of two software systems. In section 5, we present our conclusions and future directions

In this paper, we use the word trace to mean procedure-call trace and the words redundancy and repetition are interchangeable.

2. The common subexpression problem

Any tree can be represented in a compact form by representing the occurrences of the same subtrees only once [5]. The result is a directed acyclic graph (see Figure 1). This process solves what is called the "common subexpression problem", and is also called "sharing" or "subtree factoring". The main advantage of such a transformation is to save memory, but it can also facilitate the construction of tools to explore trees efficiently.

An early work in the common subexpression problem was presented by Downy, Sethi and Tarjan [3]. They suggested solving it using a linear-time algorithm based on radix sorting. However, their algorithm seems to be of theoretical interest only due to its complexity and the use of a large number of hidden constants.

A more practical algorithm is presented by Flajolet et al [5]. Their top-down recursive algorithm computes the compact form of a rooted binary tree in expected linear time. The algorithm combines a dynamically maintained table and hashing techniques to assign a unique identifier (UID) to each distinct subtree. Two subtrees are then isomorphic if their corresponding roots have the same identifier. The unique identifier of a node *n* consists of a triple *<Label(n), UID(Left(n)), UID(Right(L))>* where *Right(n)* and *Left(n)* represent the right and left children of *n* and *Label(n)* represents the label of the node *n.* The unique identifier is computed for each node. However, before assigning it to the node, the algorithm checks if the identifier has not already been assigned to another node. For this purpose, the algorithm uses a hash table to store the identifiers. If yes, the identifier is then returned from the table. Otherwise, the algorithm allocates the new identifier to the node and updates the table. The time the algorithm takes to compute the unique identifiers is constant because the input tree is a binary tree. Since a hash table is accessible in an expected constant time, the running time of the algorithm is reduced to the time it takes to traverse the tree, which is linear.

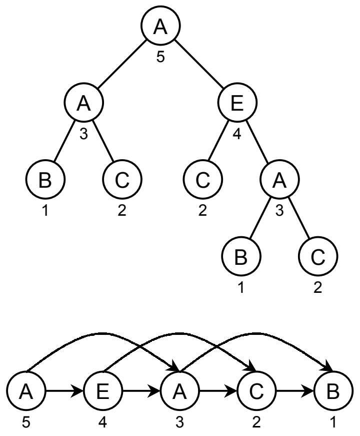

Figure 1. Compacted directed acyclic graph of a tree

Valiente presents an iterative version of Flajolet et al's algorithm [9]. His algorithm is based on a bottom-up traversal of the tree using a queuing mechanism. Veliente's approach extends the concept of unique identifiers to the notion of certificates and signatures. A certificate is a positive integer between 1 and n (n represents the size of the tree). The certificates are assigned to the nodes in such a way that two nodes have the same certificate if and only if the trees rooted at them are isomorphic as illustrated in Figure 1. To compute the certificate, the algorithm uses a signature scheme that identifies each node. The signature is similar to the unique identifier of Flajolet et al's algorithm. Similarly to the previous algorithm, a hash table is used to maintain the certificates and signatures, which limits the complexity of the algorithm to the time it takes to traverse the tree and the time it takes to compute the signature. If the degree of the tree (the number of children of any given node) is bounded by a constant, then the algorithm takes linear time.

3. Trace compression framework

In this section, we present a framework for trace compression without loss of information. The framework takes as input an execution trace file based on procedure calls and returns as output its compressed form. The trace file can be seen as a sequence of calls $C_0C_1 \ldots C_{n-1}$ where n represents the size of the trace (number of calls). Generally speaking, C_i is composed of a triple $<F_i, P_i, L_i>$ where F_i is the file name of the file that declares the called procedure, P_i stands for the procedure call and L_i is a positive number representing the nesting level. Figure 2a shows an example of a trace containing 6 calls. The same trace can be represented by a tree as shown in figure 2b.

In the case of concurrent systems, the process' names are also indicated, usually before the file name, which extends the definition of a call to a quadruple. Some other information may be traced as well such as the running time of each procedure and so on.

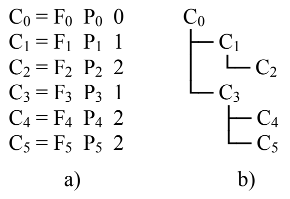

Figure 2. Example of a procedure-call trace

An easy way of compressing the trace is to take its tree representation and apply the common subexpression algorithm. However, due to the size of the trace, the degrees of the tree may vary considerably and therefore have a significant impact on the performance of the algorithm. To improve performance, and also help software engineers browse the trace more easily, we therefore propose *preprocessing* the trace tree before using the subexpression algorithm. Preprocessing the trace consists of detecting and removing non-overlapping contiguous redundancies generated by loops and recursion. Following preprocessing, we apply the common subexpression algorithm to remove any remaining redundancies.

3.1 Trace Preprocessing

Redundant calls caused by simple loops and recursion tend to encumber the trace and should be removed. We store the number of occurrences of each redundancy so it is possible to reconstruct the original trace. We distinguish between contiguous redundancies of a single procedure call and contiguous redundancies of a sequence of calls.

The first type of redundancy is easy to detect in linear time. However, one needs to distinguish between repeated calls due to loops and those due to recursion. The former appear in the trace file in the form of contiguous calls with the same nesting level. That is C_{i+1} is a repetition of C_i generated by a loop if $F_{i+1} = F_i$, $P_{i+1} = P_i$ and $L_{i+1} = L_i$. Repetitions due to recursion appear exactly the same way except that the nesting levels are increased by 1 after each call, that is $L_{i+1} = L_i+1$. Once the redundant calls are detected, we need to represent them only once and store the number of their occurrences. For this purpose, we delete the repeated lines and add a counter to the call that represents them.

The rest of this section discusses the second type of redundancy – detection of contiguous redundant sequences.

In a trace based on procedure calls, the size of a sequence represents the number of procedure calls that exist in a loop or nested loops (we deal with nested sequences later). Let d be the largest size of a repeated sequence. It is a well-known fact that d is extremely small compared to the size of the trace.

Once the contiguous sequences are detected, we need to remove them and replace them by the number of their occurrences. However, it is important to maintain the hierarchical nature of the trace as shown in Figure 3 so it is possible to use it as input for the next step of the framework. For this reason, we add a virtual call that will be inserted at the beginning of the repeated sequence. The number of occurrences is added between brackets as illustrated in Figure 3. A consequence of such a representation is to avoid detecting overlapping sequences.

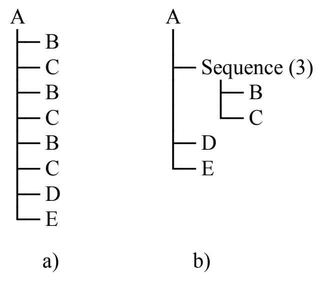

Figure 3. A contiguous sequence replaced with the number of its occurrences

The rest of this section is organized as follows. First, we present an algorithm that detects non-overlapping contiguous sequences and represent them once. Then, we discuss its running time.

Our algorithm is divided into two main steps. The first step is concerned with detecting non-nested contiguous redundancies. The second step uses the algorithm defined in the first step to detect nested sequences. In both steps, we are interested in non-overlapping sequences only. To begin with, we introduce the following definitions:

Definition 1:

Two calls C_i and C_j are similar (denoted as $C_i = C_j$) if they refer to the same file name, procedure name and nesting level. In addition to that, in case C_i and C_j are repeated (the first type of redundancy discussed above), they have to be repeated the same number of times. This last condition can be verified by comparing the counters of C_i and C_j.

Definition 2:

Two sequences of calls $S_1 = C_1C_2...C_n$ and $S_2 = C_1C_2...C_m$ are similar if $n = m$ and $C_i = C_j$ using Definition 1, for all i and j such that $0 < i \leq n$ and $0 < j \leq m$.

The algorithm of Figure 4 detects and removes non-overlapping non-nested contiguous redundancies. The algorithm proceeds through the calls of the trace. For a call C_i, the algorithm looks at most d lines back starting from $i-1$ until it finds a call C_j such that $C_j = C_i$ (according to Definition 1). If j exists, that means that $C_jC_{j+1}...C_{i-1}$ is a candidate redundant sequence, we will call S. In this case, the algorithm looks for all the possible contiguous occurrences of S (using Definition 2). For example, If S is repeated, then its first redundancy should be the sequence $C_iC_{i+1}...\ C_{i+(i-j)-1}$. If the algorithm finds redundancies of S then it removes them, inserts the virtual call *SEQUENCE(m)* such that m is the number of occurrences of S, shifts the nesting levels of the calls of S and finally makes sure not to have overlapping by ignoring part of the trace that has been already processed (using the variable *start*) – it is important not to forget that, in this step, we are looking for non-nested sequences only.

To estimate the running time of the algorithm, let us consider m_1 the total number of all removed occurrences of any redundant sequence. Therefore, the number of lines that have been removed from the trace according to the algorithm is at most dm_1 - obviously $dm_1 < n$. The time the algorithm takes to look for candidate redundant sequences (step 2.2) is bounded by $O(d(n-dm_1))$ since the number of lines that are not removed is equal to $n-dm_1$ and the largest size of a sequence is d. Step 2.5 makes at most dm_1 comparisons and step 2.7 takes at most $O(dm_1)$ to remove the repeated occurrences. Step 2.9 can be bounded by $O(dm_1)$ since the number of redundant sequences is certainly less or equal than the number of their occurrences. Step 2.8 and 2.10 can be bounded by $O(dm_1)$ as well for the same reason. Since, $dm_1 < n$ then the whole algorithm takes $O(dn)$ time. Since d is normally very small compared to n, therefore the algorithm runs in linear time.

```
1       start = 0; i = 0;
2       While (i < n)
2.1       For call  $C_i$
2.2       Find j such that: $start <= j < i$ and $(i – j) <= d$
          and $C_j = C_i$
2.3       If j exists then
2.4         $C_jC_{j+1}....C_{i-1}$ is a candidate sequence denoted $S$
2.5         Find all contiguous redundancies of $C_jC_{j+1}....C_{i-1}$
2.6         If there are any
2.7           Remove them
2.8           Insert a virtual call before $Cj$ labeled
              SEQUENCE($m$) such that $m$ is the number of
              occurrences of $S$
2.9           Shift the nesting levels of $C_j... C_{i-1}$
2.10          $start$ = the index of the call that comes after the
              last call of the last redundancy to avoid
              overlapping
2.11        End if
2.12      Else
2.13        i++
2.14      End if
3       End
```

Figure 4. Algorithm for detecting non-overlapping non-nested contiguous redundancies

However, this algorithm excludes contiguous sequences due to recursion and the sequences presented in Figure 5. In fact, one can generalize it to deal with these sequences as well. However, the goal of the preprocessing step is to reduce the size of the trace to increase of the performance of the common subexpression algorithm, so we want to keep it as simple as possible and more importantly run in linear time. The common subexpression algorithm will detect all remaining redundancies.

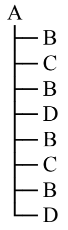

Figure 5. A sequence that is not detected by the algorithm of Figure 4

Nested sequences may occur usually due to the presence of nested loops in the source code. A nested loop can appear either in the same procedure or involve two or more procedures that call each other. Let k be the largest level of nesting. k will normally be very small since loops nested more than a few times are impractical for programmers. An intuitive algorithm that detects nested sequences consists of iterating the algorithm of Figure 4 *k* times as shown in Figure 6

```
1       t = 0
2       While (t < k)
2.1         Perform the algorithm of Figure 4
2.2         t++;
2.3         d++;
2.4     End while
3       End
```

Figure 6. An algorithm for detecting nested sequences

If we consider $k = 2$, the example of Figure 7 requires two passes of the trace. During the first pass, the algorithm detects and removes the sequences EF in both subtrees and replaces them by their corresponding virtual calls. The second pass detects the sequence BCDSequence(2)EF and represents it only once. We notice that *d* (the size of the largest contiguous redundant sequence) is incremented by 1 in order to consider the virtual calls that are eventually inserted after each iteration. The reader can notice that the running time of this algorithm is $O(kn(d+k))$. Since *d* and *k* are very small compared to *n*, the running time of the algorithm is then reduced to $O(n)$.

k and *d* are considered as thresholds and need to be determined. One way of determining them is to perform a static analysis of the files that are involved in the execution trace. Another way is to run the algorithm until no more redundant sequences are detected.

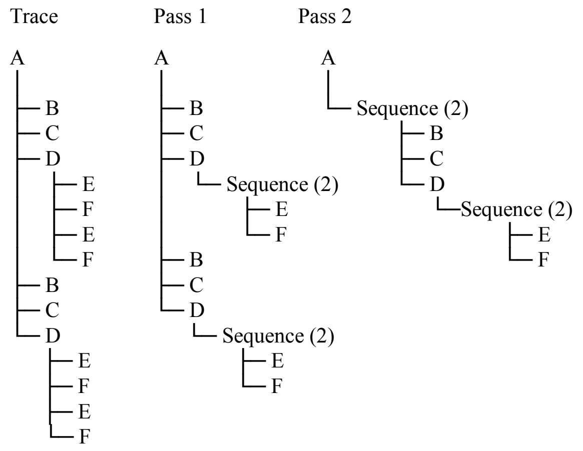

Figure 7. An example of detection and replacement of nested sequences

3.2 Application of the common subexpression algorithm

The next step consists of removing more complex redundancies, such as those shown in Figure 5 and the non-contiguous redundancies. First, the trace is represented in the form of a rooted tree and then the common subexpression algorithm is used.

3.2.1 Tree representation of a trace

Any trace based on procedure calls can be represented by a rooted ordered labeled tree as illustrated in the figures used in the previous section. The nodes represent the procedure calls. The tree levels are similar to the nesting levels. We will need to perform a preorder traversal of the tree in order to output the trace with respect to the sequence of calls. The depth of the tree should be equal to the largest nesting level.

It is important that the tree be ordered. This means that two subtrees are not isomorphic if they have the same nodes but their order is different. However, some application domains may omit this restriction, resulting in greater compression but an inability to reverse the compression. This could be useful in program comprehension, for example, where it may not always be important to know the exact sequence of calls when merely trying to understand high-level architecture. In this paper, we consider ordered trees only.

3.2.2 Application of the algorithm

The main step of the framework is to detect and remove all remaining redundancies that may exist in the trace, such as those shown in Figure 5 and non-contiguous redundancies. One should be very careful not to ignore the nesting levels when detecting these sequences. The reason is that they can be subject to further analysis in order to discover other characteristics of the system (e.g. dependencies between the trace components).

The result of applying the common subexpression algorithm is an acyclic graph where all repetitions are represented only once. The size of the graph corresponds to the number of its nodes.

3.3 Measuring Compression

There are different ways of measuring the gain attained by a compression process. The easiest way is to compute the difference between the size of the original trace and the size of the compressed version. In this paper, we use a compression ratio that we define as the ratio of the compressed trace size to the original trace size. The smaller the ratio, the better the result. Generally speaking, we are interested in two compression ratios. The first one estimates the gain reached after preprocessing the trace – removing most of the contiguous redundancies. The second compression ratio represents the gain attained after applying the common subexpression algorithm.

More formally, Let *n* be the size of the original trace. *n* represents the number of calls that exist in the trace. Since any trace can be represented by a rooted tree, *n* also represents the number of nodes of the tree representation of the original trace. Let n_1 represent the size of the trace after preprocessing it, which is the number of nodes of the tree representation of the trace after removing the repetitions caused by loops and recursion. Let n_2 be the size of the acyclic graph (the number of its nodes). The compression ratios we are interested in are r_1 and r_2 such that:

1. $r_1 = n_1 / n$
2. $r_2 = n_2 / n$

4. Experiment

4.1 Description

We experimented with execution traces of two software systems. Our main objective is to estimate the expected gain attained by compressing them. However, we also want to understand how this gain is reached and if there is a way to further improve it.

The first system is a drawing editor under UNIX called XFig [13]. The reason we chose XFig is because its domain concepts are intuitive and easy to understand. This facilitates the design of the software features that will be traced. Another reason is that XFig is a small open source system and therefore easy to instrument.

The second system is a large legacy telecommunication system provided to us by the company that sponsors our research. An interesting thing about this system is that it is already instrumented with probes, usually used for testing or performance analysis. An internal mechanism allows generating all kinds of traces just by turning the probes on and executing the scenarios that correspond to the software features we wish to trace. We discuss the specifics of each system in the next section.

In order to attain our objective, we collect different kinds of data for a deeper analysis. There are three different types of data we think are important to our experiment. We categorize them according to the following criteria:

- Size criteria
- Design criteria
- Performance criteria

Size criteria: In this category, we want to estimate the gain in terms of size obtained by applying the trace compression framework shown in the previous section. Typical results would be:

1. The initial size of the trace n
2. The size of the trace after preprocessing it n_1
3. The compression ratio r_1 such that $r_1 = n_1 / n$
4. The size of the trace after using the common subexpression algorithm n_2.
5. The compression ratio r_2 such that $r_2 = n_2 / n$

Design criteria: The aim is to understand how the trace components can influence the compression result. We are interested in the following data:

1. The number of procedures involved in a specific trace. It is important to distinguish between a procedure and a procedure call. The same procedure can be called several times
2. Similarly, we are interested in the number of files

There are many other data that can help understand the behavior of the trace components and therefore suggest a better compression technique. For example, if a particular procedure call appears everywhere in the trace, chances are that the corresponding procedure is a utility procedure. Such a procedure may not be of great interest for program comprehension as shown in [12] and might be ignored. However, this paper focuses on compression without loss of information only.

Performance criteria: The running time of the common subexpression algorithm is linear if the degrees of the tree are small and bounded by a certain constant. However, the degrees of an execution trace can vary considerably even though the trace has been preprocessed. In order to assess the performance of the algorithm, we present and compare two graphs that show the variation of the degrees of the tree according to depth before and after the preprocessing step.

4.2 Experiment Design

The first step of our experiment is to design the scenarios that need to be traced. An easy way of doing this is to take different software features of each system and generate the corresponding execution traces. However, to increase the accuracy of our experiment, the software features should cover different parts of the system.

XFig:

XFig is a drawing system under UNIX [13]. XFig uses a menu to allow the users to manipulate objects interactively. We instrumented XFig semi-automatically by adding print statements at the entry and return points of each procedure. We considered software features that are related to the ability to draw and edit different shapes. We chose to compress the execution traces of the following software features:

1. Draw a circle
2. Draw a regular polygon
3. Draw a polyline
4. Draw text
5. Delete an object
6. Rotate an object

7. Move an object
8. Copy an object

The Telecommunication System:

The telecommunication system, we experimented with, is a large legacy system. Generating the system's traces needs a very good understanding of how the scenarios are executed. We asked the software engineers to provide us with the software features they are interested to investigate and the way to execute them. Among the execution traces we gathered, we present an analysis of five of them.

Due to the concurrent nature of the system, a trace file shows all the processes being executed at the same time. Therefore, there is a need to split the trace file into many other files that we call *process files*. For each trace file, we will have as many *process files* as processes in the trace. The compact form of the original trace is the accumulation of compacting all its process files.

An issue that needs to be addressed when a real-time legacy system is used is the fact that the trace may be incomplete, that is, it may not contain some of the procedure calls that actually occurred. This is reflected as an inconsistency in the trace with respect to the nesting levels. There are many reasons that may cause such inconsistencies, e.g. a bug in the trace generation application, real-time interference, and so on.

One solution to this problem is to complete the trace by filling up the gaps with virtual procedure calls. In this case, we say that the trace contains errors and it becomes necessary to estimate the error ratio, which is the number of missing calls to the size of the original trace. We represent the error ratio as:

$$e = g / (g+n)$$

Such that g is the number of gaps (missing procedure calls) and n the size of the trace.

4.3 Results

In this section, we summarize the results of applying the trace compression framework to the scenarios of XFig and the telecommunication system.

XFig:

Table 1 shows the size and design criteria results of compressing XFig's traces. The compression ratio reached after removing the contiguous redundancies is almost the same for all the traces except for trace 2 (drawing a regular polygon). A regular polygon is a polygon whose sides have all the same length, and whose angles are all the same. By default, XFig draws a regular polygon with 5 sides. We think that the presence of so many contiguous redundancies (r_1 = 7.13%) is due to the fact that XFig repeats the same pattern to manipulate the 5 sides and angles at the same time. Trace 4 (drawing text) has the smallest number of contiguous redundancies (r_1 = 41.92%). We think that this is due to the nature of this feature, which tends to be different from manipulating shapes.

The next step of the framework shows interesting results. The compression ratios after removing all redundancies lies between 2.46% (trace 2) and 9.92% (trace 1). The results seem homogeneous except for trace 2 that we think is due to the same reasons mentioned above.

Table 1. Size and design criteria of compressing XFig's traces

Trace	n	n_1	r_1 (%)	n_2	r_2(%)	# Proced.	# Files
1	2198	623	28.34	218	9.92	167	30
2	9076	647	7.13	223	2.46	174	31
3	5140	889	17.30	236	4.59	178	30
4	2710	1136	41.92	248	9.15	190	28
5	3077	700	22.75	236	7.67	190	33
6	6215	869	13.98	261	4.20	197	31
7	3381	839	24.82	253	7.48	187	33
8	4336	830	19.14	267	6.16	190	33

We think that one of the major factors that contributed in getting such results is the number of procedures involved in the traces. For example, the software feature represented by trace 1 uses 167 procedures to generate 2198 calls. This represents 7.59% only. Therefore, there must be many redundancies all along the trace. Another reason can be associated with the nature of the system itself. It is very common, in the case of a drawing system, to have repeated patterns to manage the layout, the coordinates, the zoom and so on. This could explain why the compression ratios are so low.

The performance criteria consist of analyzing how the degrees of the tree representation of the trace vary according to depth. The reason we are interested in this is to evaluate the performance of the common subexpression algorithm (see section 2). For this purpose, we present two graphs using 3 traces of XFig (we cannot show all the results here because of space limitations).

The first graph shows the variation before the preprocessing step see Figure 8a and the second graph shows the variation after the preprocessing step (see Figure 8b). In Figure 8b, we notice that the degrees of the tree decrease continuously, which makes the performance of the subexpression algorithm depends essentially on the size of the trace.

Another observation related to the graph illustrated in Figure8b derives from the fact that, after the preprocessing step, the 3 curves look similar even though the scenarios are different. In fact, one can do a very useful analysis of the structure of XFig and its complexity just by analyzing its traces. For example, according to the graph, it is evident that XFig uses similar patterns to draw a polyline or text, or to move an object. The analysis of XFig's structure is then reduced to discovering these patterns.

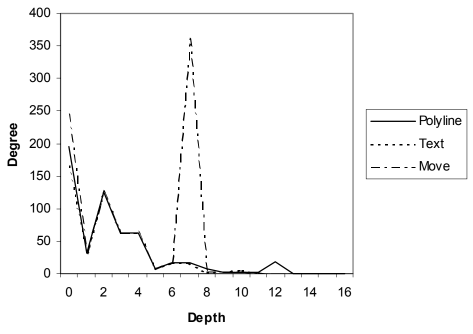

Figure 8a. Variation of the degrees of the tree according to depth before the preprocessing step

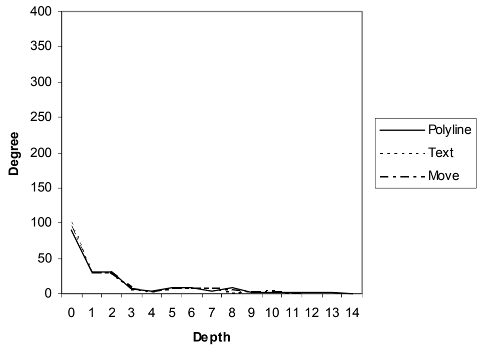

Figure 8b. Variation of the degrees of the tree according to depth after the preprocessing step

The Telecommunication System:

Table 2a shows the size criteria results of compressing the telecommunication system's traces. The table 2b shows the design criteria. Before presenting the compression ratios, we notice that table 2a contains two additional columns to represent the number of missing procedure calls *g* and the error ratio *e* presented in section 4.2. Even though, the error ratio is very small, one needs to do a deeper analysis to understand its impact on the resulting compression. We leave this point as future work.

The compression ratio attained after removing the contiguous redundancies is almost the same for all the traces and lies between 82.43% and 90.91% which is very high compared to XFig's traces. We think that this is due to the complex nature of this system and to the presence of missing calls that we had to replace with distinct virtual calls. The next step of the framework shows a compression ratio that lies between 14.04% to 31.86%, which is still a good result given that this compression is reversible.

Table 2a. Size criteria (*n* represents the sum of the initial size of the trace and *g*)

Trace	n	g	e (%)	n_1	r_1 (%)	n_2	r_2 (%)
1	17465	589	3.37	14396	82.43	2452	14.04
2	11095	313	2.82	9715	87.56	3308	29.82
3	10175	381	3.74	8654	85.05	2361	23.20
4	3621	121	3.34	3226	89.09	649	17.92
5	3609	109	3.02	3281	90.91	1150	31.86

Table 2b. Design criteria of compressing the telecommunication system's traces.

Trace	# Procedures	# Files
1	802	189
2	828	184
3	876	190
4	657	160
5	668	164

We notice that the number of procedures and files involved are considerably higher compared to XFig's traces. This could explain why the compression ratios are also higher. In addition to that, we think that the presence of errors can be a significant factor as well. In fact, one can come up with a very interesting complexity metric based on the concepts presented here. The telecommunication system is obviously more complex than XFig and this is clearly shown in analyzing its traces.

The graph presented in Figure 9a shows the variation of the degrees according to the depths of the rooted trees corresponding to three processes of trace 2 before preprocessing them. We cannot present all the results here but most of the graphs look similar to the graph presented in this figure. Figure 9b shows how the degrees decrease after the preprocessing step, which will result in an increase in the performance of the common subexpression algorithm.

Obviously, there is a gain in terms of execution time in having the preprocessing step. However, one needs to experiment with more software systems to assess this gain. From the comprehension perspective, we think that removing contiguous redundancies will help software engineers to better understand the traces.

4.4 Discussion

We showed that procedure-call traces could be considerably compressed in a way that preserves the ability for humans to understand them. The experiment's results have exceeded our expectations with the satisfactory compression ratios that we got for both systems. However, we need to do more experiments to see how all this could be useful to software engineers.

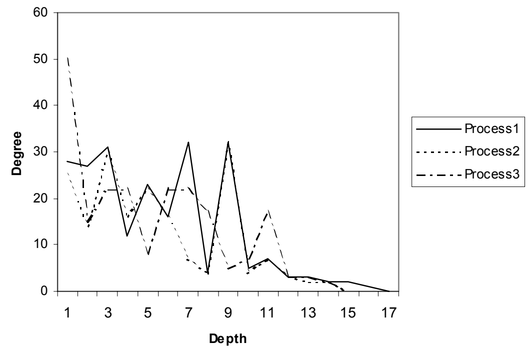

Figure 9a. Variation of the degrees of the tree according to depth before the preprocessing step

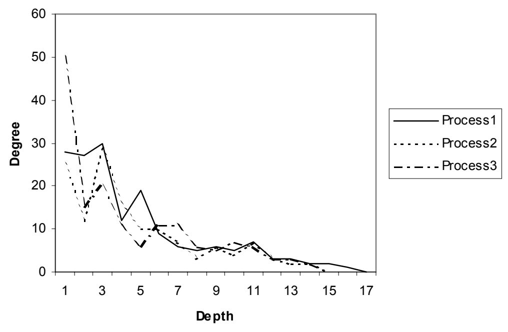

Figure 9b. Variation of the degrees of the tree according to depth after the preprocessing step

We saw that one of the major factors that make this compression possible is the number of procedures involved in the trace. We notice that the higher this number, the higher the compression ratio (hence the lower the reduction in trace size). This is easy to explain since the trace is composed of invocations of these procedures. Therefore, if the trace is very large and the number of procedures it invokes is small then there must be many redundancies.

One possible improvement to the techniques showed above is to look for procedures that are not of a great interest to software engineers and remove them before the compression process. Such procedures can be referred to as utility procedures. However, the resulting compressed trace will not be reversible.

Another interesting finding is the idea that compressing a trace allows the software engineer to focus on useful information that it conveys. We believe that this is very helpful to understand the trace's structure and the system in general.

Finally, we notice that the preprocessing stage was very useful to reduce the trace size and therefore increase of the performance of the common subexpression algorithm.

5. Conclusions and future directions

The results shown in this paper can help build better tools based on execution traces in general and procedure-call traces in particular. We showed that an execution trace can be reduced to a very small size if the repeated patterns are removed and represented only once. The same principles can apply to other kinds of traces as well. In fact, trace compression is not concerned with reducing the size of the trace only but removing what is not necessary in order to focus on the useful information that is conveyed. We also showed that the result of the compression can be subject to a deeper analysis to understand the complexity of the software feature that is represented. We intend to conduct more experiments with this framework to see how helpful it is to software engineers.

Future directions should focus on lossy compression, that is, the original trace is not totally reconstructed from the compressed version. This can help to further reduce the trace by getting rid of what is not useful to software engineers. Types of information eliminated can include the number of repetitions, the order of calls, and some lower-level utility procedures. There are many criteria that help determine which procedures are useful to retain and which ones can be considered utilities. For example, the number of occurrences of the procedure in the trace, its execution time and so on.

In this paper, we considered ordered trees only, that is, the sequence ABC is different from the sequence ACB. It is very common that this gives the same result and it is just an omission from the developer to keep track of the order. It would be interesting to explore when this order is important and when it is not and suggest lossy compression based on that.

Finally, the non-contiguous redundancies that are detected using the common subexpression algorithm can be subject to further analysis to understand other proprieties of the system such as the relationships between its components. The common subexpression algorithm can be used to extract them efficiently.

Thanks

We would like to thank the software engineers at Mitel Networks who helped us to generate the execution traces that correspond to the software features of the telecommunication system used in this paper.

About the Authors

Abdelwahab Hamou-Lhadj is a Ph.D. student at the School of Information Technology and Engineering at the University of Ottawa. His research interests include reverse engineering, program comprehension and object-oriented technology. His URL is www.site.uottawa.ca/~ahamou

Dr. Timothy C. Lethbridge is an associate professor at the school of Information Technology and Engineering at the University of Ottawa. His research interests include user interfaces, software engineering tools, knowledge representation and software engineering. Dr. Timothy Lethbridge is also the co-author of McGraw Hill textbook - Object Oriented Software Engineering: Practical Software Development using UML and Java. His URL is www.site.uottawa.ca/~tcl

6. References

[1] T. Balls, “The concepts of dynamic analysis”, *In Nierstrasz and Lemoine [14]*, pages 216-234

[2] D. Salamon, “Data Compression, the complete reference ”, 2nd Edition, *Springer-Verlag*, NY 2002

[3] J.P. Downey, R. Sethi and R.E. Tarjan, “Variations on the common subexpression problem”, *J. ACM. 27*, pages 758-771, 1980

[4] T. Eisenbarth, R. Koschke, D. Simon, “Feature-Driven Program Understanding Using Concept Analysis of Execution Traces”, *In the Proceedings of the International Workshop on Program Comprehension (IWPC),* Toronto, Canada, May 2001

[5] P. Flajolet, P. Sipala, J.–M. Steyaert, “Analytic variations on the common subexpression problem”, *In Automata, Languages, and Programming*, volume 443 of Lecture Notes in computer science, pages 220-234, Springer-Verlag, 1990

[6] D.F. Jerding, J.T. Stasko, T. Ball, “Visualizing Interactions in Program Execution”, *ICSE*, 1997

[7] T. Richner, S. Ducasse, “Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information”, *IEEE*, 1999

[8] T. Systä, “On the relationship between Static and Dynamic Models in Reverse Engineering Java Software”, *In the Proceedings of the 6th WCRE*, pages 304--313, 1999.

[9] G. Valiente, “Simple and Efficient Tree Pattern Matching”, *Research report, Technical University of Catalonia, Department of Software*, E-08034 Barcelona, December 2000

[10] R.J. Walker, G. C. Murphy, J. Steinbok, M. P. Robillard, “Efficient Mapping of Software System Traces to Architectural Views”, *In Proceedings of CASCON*, pages 31-40, Toronto, Canada, November 2000.

[11] R.J. Walker, G. C. Murphy, B. Freeman-Benson, D. Swanson, J. Isaak, “Visualizing dynamic software system information through high-level models”, *In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, British Columbia, Canada, 18-22 October 1998

[12] I. Zayour and T.C. Lethbridge, “A Cognitive and User Centric Based Approach For Reverse Engineering Tool Design”, *In Proceedings of CASCON*, pages 31-40, Toronto, Canada, November 2000.

[13] Xfig System, http://www.xfig.rg