An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls[*]

Abdelwahab Hamou-Lhadj
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
ahamou@site.uottawa.ca

Timothy C. Lethbridge
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
tcl@site.uottawa.ca

Abstract

Examining the behavior of a large legacy software system helps understand its functionality. Dynamic analysis techniques are well suited for this purpose. Run-time information is typically represented in the form of execution traces; however, the amount of information contained in a trace, of even a small program, can be very large and usually overwhelming. It becomes important to filter these traces and present only the information that adds value to the comprehension process. Many researchers agree that analyzing recurrent patterns in a trace can be useful to bridge the gap between low-level system components and high-level domain concepts. This paper introduces an efficient algorithm that extracts patterns of procedure calls of large execution traces. We also present a set of matching criteria that can be used in procedural as well as object oriented software systems to decide when two patterns can be considered equivalent.

Keywords:

Reverse engineering, program comprehension, dynamic analysis, execution traces, trace patterns

1. Introduction

Understanding a poorly documented software system is not an easy task. Program comprehension techniques aim at overcoming this difficulty. Tools based on these techniques can indeed help software maintainers to complete their daily tasks in a more efficient way [9]. In general, reverse engineering tools can be categorized according to whether they perform a static analysis of the code or a dynamic analysis of the executing system. In [10], Stroulia and Systä presented a large set of reverse engineering activities where dynamic analysis can be used, such as, extracting system modularization, understanding the role of software artifacts and so on. Many other researchers use run-time information to solve the popular problem of feature localization – locating low-level system components that implement a particular software feature [4, 5, 13]. Moreover, Zayour and Lethbridge [14] experimented with a large real world telecommunication system and found that traces of procedure calls, once made usable, can be very useful to help maintainers perform cognitively taxing activities. Their tool, called DynaSee, uses techniques such as redundancy removal, pattern detection and routine ranking to overcome the size explosion problem of run-time information. Among the features of DynaSee is the possibility for software engineers to replace a pattern of procedure calls (called trace pattern) with a textual description mapping low-level system components to high-level application domain concepts. However, they did not present an algorithm that detects these patterns.

In this paper, we present an efficient algorithm that extracts trace patterns. We also present a list of pattern matching criteria that can be used in procedural software systems to group similar but not necessarily identical patterns together. Our algorithm is based on a technique used to solve a problem known as the common subexpression problem [3, 6], which consists of transforming a rooted tree into its most compact form in such a way that all isomorphic subtrees are represented only once. Figure 1. illustrates this concept.

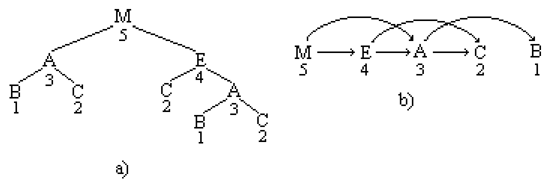

Figure 1. The graph b) represents the compact form of the tree a)

Jerding et al. [8] presented an algorithm that is similar, in principle, to the one provided in this paper. However, their algorithm has some limitations, as we will see in the related work section. The rest of this paper is organized as follows; the next section presents related

[*] This research is supported by NSERC

work. We define what we mean by trace patterns in Section 3. The algorithm that detects them is explained in Section 4. Section 5 describes a set of matching criteria that can be used to decide when two patterns are equivalent. Finally, we conclude in Section 6.

2. Related work

Jerding et al.[8] emphasized the importance of trace patterns for understanding the behavior of object oriented systems. They also presented an algorithm that identifies them. However, their algorithm considers all kinds of repetitions as patterns. This is probably due to the requirements of their visualization tool. For example, they considered contiguous repetitions as trace patterns (that is, candidate high-level concepts) at the same level as non-contiguous repetitions. We think that contiguous redundancies encumber the trace and do not add value to its content. They should be removed and replaced by the number of their occurrences, if necessary. The same choice was made by Zayour and Lethbridge [14] and De Pauw et al. [2]. In addition to that, their algorithm considers identical matches only.

De Pauw et al. [2] considered patterns that are similar but not necessarily identical and presented an interesting list of matching criteria. However, they briefly discussed the algorithm that detects them. In addition to that, most of their matching criteria apply to object oriented systems only.

3. Definition of a trace pattern

Ideally, a trace pattern captures a high-level domain concept. In procedural software systems, these concepts are usually implemented in the form of interactions between the system procedures. Zayour and Lethbridge define a trace pattern as "a sequence of calls that occurs repetitively but non-contiguously in several places in the trace" [14]. This definition excludes patterns that are not identical but that exhibit some similarities. We add to this definition the fact that instances of this sequence of calls do not need to be identical but satisfy some pattern matching criteria. Enabling fuzzy similarity can be very beneficial to trace compression and visualization. The pattern matching criteria can vary depending on the system at hand. They can be either specified by the users or extracted automatically using heuristics.

4. The algorithm

A trace of procedure calls can be represented by a rooted, ordered, labeled tree. Each node corresponds to a procedure call. The node label can be the name of the procedure. The tree levels correspond to the nesting levels of the calls. A trace pattern is then represented as a repeated subtree. Our algorithm starts with a preprocessing stage that aims at removing contiguous repetitions due to loops and recursion. In [7], we presented a simple but efficient algorithm that does this. The hierarchical nature of the trace is maintained by adding a virtual call whose label starts with *Seq* followed by the number of occurrences of the repeated sequence. Please, note that this virtual call can be omitted in case of repetitions of single procedure calls as illustrated in Figure 2.

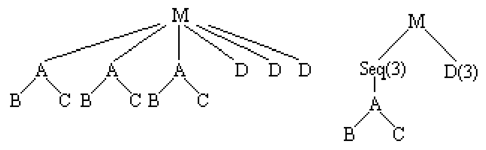

Figure 2. Removing contiguous repetitions

Now that the trace is preprocessed, we apply the pattern detection algorithm to extract trace patterns. As mentioned earlier, the idea behind this algorithm is based on transforming a rooted, ordered, labeled tree to its most compressed form by representing repeated subtrees only once. The result of this compression is a directed acyclic graph as shown in Figure 1. Flajolet et al. described a top-down recursive procedure that solves this problem in an expected linear time assuming that the degree of the tree is bounded by a constant [6]. Valiente presented an iterative version of Flajolet et al.'s algorithm with a slight improvement of its readability [12]. In our previous work, we used an adaptation of Valiente's algorithm to compress a trace of procedure calls [7]. In what follows, we extend it to consider similar but not necessarily identical patterns as well as enabling the frequency analysis of the patterns.

Before getting into the details of the algorithm, first, consider a function called *Match(n1, n2)* that takes two nodes *n1* and *n2* and returns true if the trees rooted at these nodes are considered similar according to predefined matching criteria. The function returns false otherwise. We discuss the specifics of this function in Section 5.

The algorithm proceeds by traversing the tree in a bottom-up fashion (from the leaves to the root). Each node is assigned a certificate (a positive integer between 1 and n, where n represents the size of the tree). The certificates are assigned in such a way that two nodes *n1* and *n2* have the same certificate if and only if *Match(n1, n2)* returns true, that is, the trees rooted at them exhibit some similarities but are not necessarily isomorphic as is the case in Valiente's algorithm.

To compute the certificate, the algorithm uses a signature scheme that identifies each node. The signature of a node *n* consists of its label and the certificates of its direct children, if there are any. A global hash table is used to store the certificates and signatures and ensure that similar subtrees will always hash to the same element. We added a new field to the table in order to select only patterns that satisfy a certain frequency threshold. Table 1. shows the resulting table that corresponds to applying the algorithm to the tree of Figure 1. The frequency field enables the frequency analysis of the trace. T. Ball showed that frequency analysis of dynamic information can help programmers cluster components according to their behavior and identify related computations [1].

Table 1. Result of the algorithm when applied to the tree of Figure 1.

Certificate	Signature	Frequency
1	B	1
2	C	1
3	A 1 2	2
4	E 2 3	1
5	M 3 4	1

The complexity of the algorithm consists of the time it takes to traverse the tree, the time it takes to compare two subtrees, i.e. compute the function *Match,* and the time it takes to compute the signatures. If exact match is selected and the degree of the tree is bounded by a constant, the algorithm performs in expected linear time.

One can easily see that the resulting table contains a compressed form of the tree. The last step of the algorithm is to walk through the table and extract the patterns that satisfy a given frequency threshold. The table is, first, sorted in order of descending certificates, i.e. the first element of the table is the one that has the highest certificate (this corresponds to the certificate of the root). We use a recursive procedure to display the components of each pattern. The frequency threshold can be specified by the user. Future work should focus on determining it automatically.

5. Pattern matching criteria

De Pauw et al. [2] studied situations where two sequences of calls can be considered as instances of the same pattern in object oriented systems. As a result they presented a list of matching criteria. We found that some of these criteria, namely, identity, repetition, depth-limiting and commutativity can be applied to procedural software systems as well. In this section, we explain these criteria and introduce three new ones: utility, distance and flattening. The design of the function *Match* depends on the selected matching criteria. Some of these criteria can be combined together. Future work should determine how.

5.1 Identity

The identity criterion is probably the simplest one to compute. Two sequences of calls are similar if they have the same topology, which mean, they have the same call structure, order of calls and so on. This criterion might be useful for novices who wish to construct an initial understanding of the trace.

5.2 Repetition

The number of repetitions of contiguous sequences of calls does not really add too much value to the trace. These repetitions can be ignored. For example, the two subtrees of Figure 3 can be considered as instances of the same pattern.

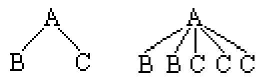

Figure 3. Repeated sequences can be ignored when looking for patterns

5.3 Ordering

This matching criterion is based on the commutative criterion presented in [2] without the restriction of considering objects of the same classes only, since, we do not deal with objects here. If the order of calls does not matter to software engineers, then it can be ignored. To generalize the algorithm to unordered trees, we need to sort the certificates that appear in the signatures before comparing them. If this criterion is used, it will certainly be beneficial to users who already have a certain understanding of the system. Future work should focus on determining the importance of the order of calls according to the tree levels where they occur. For example, the order may not be important at the leaf level where utility procedures are used. This is not necessary the case at higher levels.

5.4 Depth-Limiting

Depth-limiting allows comparing two subtrees up to a certain depth. The calls that are beyond this depth are ignored. In a layered system, components of one layer communicate with the components of the layer below. Patterns of the same layer can be grouped together. This is useful to users familiar with the system architecture. We intend to experiment with different execution traces to determine at which level of the trace tree this criterion could be applied.

5.5 Utility

Utility procedures are domain independent routines that implement specific tasks (e.g. sorting an array). Users may decide to ignore them when comparing patterns. There are different heuristics that are used to detect such procedures (e.g. compute fan-in and fan-out). Consider the two sequences of calls in Figure 4., where u1, u2, u3 and u4 are utility procedures. These two sequences can be considered similar if we decide to ignore the utility procedures.

One way of implementing this concept is to group the utility procedures in one subsystem and then go through the trace and replace their occurrences by the name of this subsystem. This results in a trace with a higher level of abstraction.

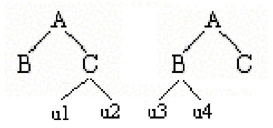

Figure 4. These two sequences can be considered similar if the utility procedures are ignored

5.6 Distance

Two patterns may have almost the same procedure calls but slightly different structures. For example, a control statement can lead to different execution paths depending on the program inputs. That is, the same program behavior might result in slightly different sequences of procedure calls. We would like to be able to group these sequences together as being one common pattern. For this purpose, we need to evaluate the difference between their structures. The tree edit distance can be used [11]. This criterion might be useful to expert users who are already familiar with the source code.

5.7 Flattening

This criterion does not consider the hierarchical structure of the patterns at all. Instead, it flattens them into a linear structure and compares them. If the same calls exist more than once then they are reduced to one occurrence. This subsumes most of the criteria presented in this paper and will certainly result in a very good compression rate. However, we need to analyze situations where it could be applied usefully.

6. Conclusion and future work

Dynamic analysis is important to understand the behavior of any software system whether it is based on OO concepts or not. Dynamic analysis tools should be as important as static analysis tools. In fact, the combination of both provides, without any doubt, the best solution to address program comprehension issues.

Patterns of procedure calls can be used to bridge the gap between low-level system components and high-level domain concepts. In this paper, we showed an algorithm that extracts them in an efficient manner. We also presented a set of matching criteria that can be used, in conjunction with the ones presented in [2], to group similar patterns. Future work should focus on validating these criteria and classify their usage according the user's knowledge of the systems. The long term goal is to determine heuristics that automatically select patterns that most likely correspond to high-level concepts.

References

[1] T. Ball, "The concept of dynamic analysis", *ACM SIGSOFT Software Engineering Notes*, v.24 n.6, , Nov. 1999, pp.216-234

[2] W. De Pauw, D. Lorenz, J. Vlissides and M. Wegman, "Execution Patterns in Object-Oriented Visualization", In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98), USENIX, 1998, pp. 219-234

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

[4] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis", *ICSM*, 2001

[5] T. Eisenbarth, R. Koschke, D. Simon, "Feature-Driven Program Understanding Using Concept Analysis of Execution Traces", *IWPC*, 2001

[6] P. Flajolet, P. Sipala, J.–M. Steyaert, "Analytic variations on the common subexpression problem", *In Automata, Languages, and Programming*, Springer-Verlag, 1990

[7] A. Hamou-Lhadj, T. C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces", *IWPC*, 2002

[8] D.F. Jerding, J.T. Stasko, T. Ball, "Visualizing Interactions in Program Execution", *ICSE*, 1997

[9] M. –A.D. Storey, K. Wong, H.A. Muller, "How Do Program Understanding Tools Affect How Programmers Understand Programs?", *WCRE*, 1997

[10] E. Stroulia, and T, Systä, "Dynamic analysis for reverse engineering and program understanding", *ACM SIGAPP Applied Computing Review*, 2002

[11] K. C. Tai, "The tree-to-tree correction problem", *ACM*, 26(3):422-433, 1979

[12] G. Valiente, "Simple and Efficient Tree Pattern Matching", *Research report*, *Technical University of Catalonia,* E-08034, Barcelona, 2000

[13] N. Wilde and M. Scully, "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance: Research and Practice*, 1995, Vol. 7, pp. 49-62

[14] I. Zayour and T.C. Lethbridge, "A Cognitive and User Centric Based Approach For Reverse Engineering Tool Design", *CASCON*, 2000