Pattern-Based Trace Correlation Technique to Compare Software Versions

[1]Maher Idris, [1]Ali Mehrabian, [1]Abdelwahab Hamou-Lhadj, [2]Richard Khoury
[1]*Software Behaviour Analysis Lab*
Electrical and Computer Engineering, Concordia University, Montréal, Canada
{m_idris, al_meh, abdelw}@ece.concordia.ca
[2]*Department of Software Engineering, Lakehead University, Thunder Bay, Canada*
rkhoury@lakeheadu.ca

Abstract. Understanding the behavioural aspects and functional attributes of an existing software system is an important enabler for many software engineering activities including software maintenance and evolution. In this paper, we focus on understanding the differences between subsequent versions of the same system. This allows software engineers to compare the implementation of software features in different versions of the same system so as to estimate the effort required to maintain and test new versions. Our approach consists of exercising the features under study, generate the corresponding execution traces, and compare them to uncover similarities and differences. We propose in this paper to compare feature traces based on their main behavioural patterns instead of a mere event-to-event mapping. Two trace correlation metrics are also proposed and which vary whether the frequency of the patterns is taken into account or not. We show the effectiveness of our approach by applying it to traces generated from an open source object-oriented system.

Keywords: Dynamic analysis, Trace correlation, Software evolution, Software maintenance

1 Introduction

One of the main challenges that engineers face when maintaining an existing system is to answer questions like what the system does, how it is built, and why it is built in a certain way [1]. Understanding an existing system has been shown to account for almost 80% of the cost of the software life cycle [2, 3]. Documentation is normally the main source of information where answers to these questions should be found, but in practice documentation is rarely up to date when it exists at all. The problem is further complicated by the fact that the initial designers of the system are often no longer available.

Execution traces have been used in various studies to observe and investigate the behavioural aspects of a software system. In most cases, traces have been found to be difficult to work with due to the large size of typical traces. Although many trace analysis tools and techniques have been proposed (e.g., [4, 5, 6, 7, 8]), most of them do not tackle the problem of correlating trace content. One of the few research studies that focuses on comparing traces is the work of Wilde [9], in which the author introduced the concept of Software Reconnaissance. The author compared traces based on their distinct components. The objective was to identify the components that implemented a specific feature (also known as solving the problem of feature loca-

tion). However, the author's approach did not take into account the interaction between the components in the trace, which is needed to understand differences in the execution trace.

In this paper, we focus on the problem of understanding the differences between subsequent versions of the same system, an activity that can help in many software engineering tasks including estimating the time and effort required to maintain new versions of the system, uncovering places in the code where faults have been introduced, understanding the rationale behind some design decisions, and so on. We propose a novel approach that allows software engineers to compare the implementation of software features in different versions of the same software system. Our approach is based on information gathered from two sources. First, we generate execution traces (dynamic analysis) by exercising the target features of the system to identify the differences between implementation. Several studies (e.g. [10, 6]) have showed that trace patterns often characterize the main content of a trace. Consequently, we propose two new metrics to measure the correlation between two traces based on their patterns, and we measure the extent of the differences between them. Once these differences are identified, we refer to the second source of information, the source code (static analysis), to understand the underlying changes. In other words, the result of the dynamic analysis not only shows the variation in the two implementations but is also used to guide software engineers in understanding where these variations appear.

The rest of this paper is structured as follows: In the next section, we briefly define the concept of trace patterns and we present our novel pattern-based approach for correlating traces. Our two proposed correlation measures are also presented in that section. A case study is presented in Section 3. We conclude the paper in Section 4.

2 Trace Correlation Approach

The aim of our approach is to compare traces generated from different versions of the same system. Both versions of the system are first instrumented and run using the same usage scenario. The generated traces then go through two main phases. The first phase consists of pre-processing the traces by removing continuous repetitions and noise caused by the presence of low-level utility components. The second phase consists of extracting similar patterns common to both traces resulting from the first phase and using them to compare the traces.

2.1 First Phase: Trace Pre-Processing

During the pre-processing phase, we begin by eliminating contiguous repetitions due to the existence of loops and recursion in the code. We then filter out utility routines such as accessing methods (sets and gets) from the raw traces. These routines encumber the traces without adding much to their content. We rely on naming conventions to identify such utilities. For example, any routine that starts with 'set' or 'get' is automatically removed. We can also refer to the system folder structure to identify utilities packages. This is aligned with Hamou-Lhadj and Lethbridge study and in which the authors showed that an effective analysis of a trace should include a utility removal stage that cleans up the trace from noise [5].

2.2 Second Phase: Trace Correlation

The trace correlation phase is comprised of two main steps: the pattern detection step and the trace correlation measure. The goal is to take the two traces, extract their behavioural patterns, and compare the extracted patterns using different similarity measures. Two traces exhibit the same behaviour if the pattern sets are deemed similar. A threshold needs to be identified beyond which one can consider two traces similar. We anticipate, however, that this threshold is application-dependent and that a tool that supports our technique should provide enough flexibility to modify this threshold.

To reduce the size of the pattern space, several matching criteria have been proposed in the literature to measure the extent to which two sequences of events could be deemed similar without being necessarily identical (see [6, 10] for some examples). Some of these criteria include ignoring contiguous repetitions, ignoring the order of calls, or treating subtrees as a set. For example, the sequence A_{BBBCCC}[1] and A_{BC} can be considered similar if the contiguous repetitions are ignored. Similarly, the sequence A_{BBBCC} and A_{CBBB} could be considered instances of the same pattern if the contiguous repetitions and the order of calls are ignored during the correlation process. Although it is still unclear how these matching criteria can be used for different maintenance tasks, the common consensus is that some sort of generalization is needed to reduce the size of the pattern space.

In [10], Hamou-Lhadj et al. presented an algorithm to detect and extract the patterns from a trace using predefined matching criteria. The algorithm uses one criterion at a time. Our method adopts the idea of the algorithm while providing the ability for using and applying more than one matching criteria to extract the similar patterns as desired. We also improved the performance of the algorithm when applied to large traces.

In the rest of this section, we use the sample traces presented in Figure 1 to illustrate the concept of correlating traces based on their behavioural patterns. Tables 1 and 2 show the patterns extracted from these traces. In this example, two matching criteria are used, which are ignoring the order of calls and removing utilities. We assume, in this example, that the utilities are the routines that start with 'u'.

2.3 Trace Correlation Metrics

We developed two new metrics to calculate the similarity between the traces of two versions of the same system based on their behavioural patterns. The two trace correlation metrics are the non-weighted trace correlation metric and the weighted trace correlation metric. Both correlation metrics range between 0 and 1, where 0 is complete dissimilarity and 1 is absolutely identical.

The non-weighted trace correlation metric, NW_TCM, is used to compare two execution traces based on the proportion of similar extracted patterns they share in common. More formally, NW_TCM is defined as follows:

$$NW_TCM(T1,T2) = \frac{\left(\frac{CPtrnN}{T1TotalPtrnN}\right) + \left(\frac{CPtrnN}{T2TotalPtrnN}\right)}{2} \quad (1)$$

[1] We use the notation A_B to mean A calls B

where CPtrnN is the total number of patterns common to both traces, and T1TotalPtrnN and T2TotalPtrnN are the total number of patterns of Trace 1 and Trace 2 respectively.

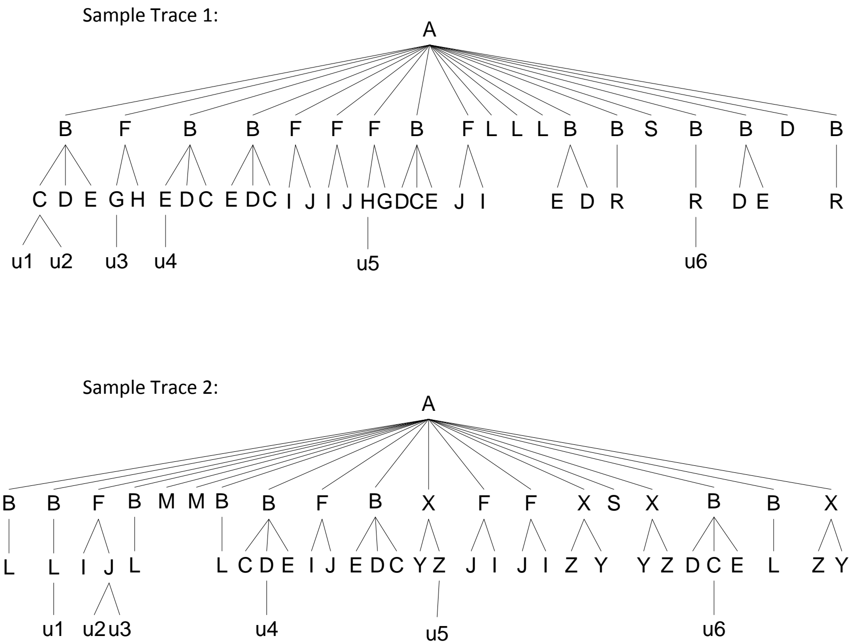

Fig. 1. Two sample routine (method) call traces

Table 1. Similar Patterns extracted from Sample Trace 1 of Figure 1

Number	Pattern	Frequency
1	B_{CDE}	3
2	F_{IJ}	3
3	F_{GH}	2
4	B_{DE}	2
5	B_{R}	3
Total		13

Table 2. Similar Patterns extracted from Sample Trace 2 of Figure 1

Number	Pattern	Frequency
1	B_{DCE}	3
2	F_{IJ}	4
3	B_{L}	4
4	X_{YZ}	4
Total		15

The weighted trace correlation metric, W_TCM, modifies the previous metric by taking into account the frequency of the patterns, i.e., the number of times the patterns occur in the traces. More formally, W_TCM can be calculated as follows:

$$W_{TCM(T1,T2)} = \frac{\left(\frac{T1CPtrnFreqN}{T1TotalFreqN} \times \frac{CPtrnN}{T1TotalPtrnN}\right) + \left(\frac{T2CPtrnFreqN}{T2TotalFreqN} \times \frac{CPtrnN}{T2TotalPtrnN}\right)}{2} \quad (2)$$

where T1CPtrnFreqN and T2PtrnFreqN are the number of occurrence of the common similar patterns of both traces in Trace 1 and Trace 2 respectively, T1TotalFreqN and T2TotalFreqN are the total frequency of all patterns found in Trace 1 and Trace 2 respectively, and CPtrnN, T1TotalPtrnN and T2TotalPtrnN have the same meaning as before.

After we performed the pattern detection algorithm on the example traces of Figure 1, we obtained the five frequent patterns of Trace 1 and four frequent patterns of Trace 2 presented in Tables 1 and 2 respectively. The two common patterns between both traces, B_{CDE} and F_{IJ}, can be discovered from these frequent patterns. Referring back to Tables 1 and 2, we can see that these patterns account for 6 of the 13 frequent pattern occurrences in Trace 1, and 7 of the 15 occurrences in Trace 2. The results of applying the non-weighted and weighted correlation metrics are as follows:

$$NW_TCM(T1,T2) = \frac{\left(\frac{2}{5}\right) + \left(\frac{2}{4}\right)}{2} = 0.45 = 45\%$$

$$W_TCM(T1,T2) = \frac{\left(\frac{6}{13} \times \frac{2}{5}\right) + \left(\frac{7}{15} \times \frac{2}{4}\right)}{2} = 0.21 = 21\%$$

Applying the NW_TCM metric results in a 45% similarity between Trace 1 and Trace 2, reflecting the fact that nearly half the frequent patterns of each trace are common to both traces. The W_TCM metric, on the other hand, gives a similarity of 21% due to the fact that it also takes pattern frequencies into account. We anticipate that the decision of which metric best reflects the similarity or dissimilarity between the traces will depend on the task at hand. For example, if one needs to understand the impact of a particular input on the resulting traces then the weighted metric could be considered since it takes into account the frequency of the patterns (which are often subject to the input data used to trigger the system). Further studies should provide more insight on situations where each of these metrics can be most informative.

3. Case Study

3.1 Target System

We have applied our proposed trace correlation technique to traces generated from two versions of the Java-based software system called Weka [11], an open-source software which was developed in the University of Waikato, New Zealand. It is a machine learning tool that supports several algorithms such as classification algorithms, regression techniques, clustering, and association rules. We selected this system because its framework and components are well documented [12], and detailed

description of its architecture are available online. The versions of Weka that have been selected for this case study are versions 3.4 and 3.7. Weka version 3.4 is comprised of 55 packages, 732 classes, 8,980 methods and 147,335 lines of code (approximately 147 KLOC) while Weka version 3.7 contains 76 packages, 1129 classes, 14111 methods and 224,556 lines of code (approximately 224 KLOC).

3.2 Usage Scenario

We have applied our trace correlation technique to a specific software feature supported in both versions of Weka, namely the J48 classification algorithm used to construct efficient decision trees. In order to generate the execution traces for the selected feature, we instrumented Weka using the open source Eclipse Test and Performance Tool Platform Project (TPTP) [13]. Probes were inserted at each entry and exit method of the intended system, including the constructor and all invoked routines, in order to instrument it.

We used a sample input data provided in the documentation and the source code package of the Weka system to exercise the J48 feature. Executing the two instrumented versions of Weka with that data generated the two execution traces for that feature that we used in this study.

3.3 Applying the Trace Correlation Algorithm

The first phase of the algorithm is to pre-process the traces by filtering out utilities, contiguous repetitions, and the methods responsible for generating the GUI and initializing the environment. This allows us to focus only on those parts of the traces concerned with the implementation of the J48 algorithm.

In Table 3, we present statistical information regarding the size of the traces before and after the pre-processing stage. We can see that the removal of contiguous repetitions and utilities reduces the size of the raw traces considerably, but the resulting traces are still in the order of thousands of method calls. The size of the initialization routine is also very small compared to the total size of the entire traces. We can also see in Table 3 that the J48 trace in Weka 3.7 is considerably longer than the equivalent trace generated from the older version Weka 3.4. This indicates that significant changes have been made to this algorithm in the newer version. The objective of our research is to evaluate the extent of these changes and to uncover the exact nature and location of these changes in the source code.

Table 3. The size of execution traces of Weka system for versions 3.4 and 3.7.

Properties of Execution Traces	Weka 3.4	Weka 3.7
Original (raw) trace size	35,974	103,009
Original trace after removing contiguous repetitions	6,850	26,978
Initialization trace size	5,919	17,534
Initialization trace after removing contiguous repetitions	682	1,288
Original trace after removing initialization trace	5,510	24,700

Quantitative Analysis. The second phase of our approach begins by applying the pattern detection algorithm. We used the “ignore order” matching criterion during the extraction process. Future work should focus on experimenting with other matching criteria to study their impact on the final result. We discover 162 frequent patterns in

Weka 3.4 and 299 frequent patterns in Weka 3.7, almost twice as many. This result shows immediately that the implementation of the J48 algorithm in Weka has undergone several changes.

Next, we apply the correlation metrics to measure the differences between the two pattern sets extracted from the Weka traces. This finds only 64 similar patterns common to both traces. This means that less than half of the total patterns relevant to each version of Weka are shared in both versions. Applying our trace correlation metrics gives a non-weighted correlation (NW_TCM) of 30.45% and a weighted correlation (W_TCM) of only 5%. The results show again that these traces are considerably different from each other. To be able to explain these differences, we perform next a qualitative examination of the patterns that are not common between the two traces by exploring the source code of the two Weka versions.

Qualitative Analysis. The dissimilarity between the two versions in terms of the number patterns (without taking into account the frequency) is almost 70%. After exploring the content of both traces, we found that the number of distinct methods of the J48 trace in Weka 3.4 was 656, whereas the number of distinct methods in the trace generated from Weka 3.7 was 1024. This led to the generation of many patterns that were in one trace and not in the other trace, patterns that were triggered by the new methods.

We focused on the patterns that exist in both traces and which derive from the same root nodes. This revealed that refactoring tasks have been used in Weka 3.7 to modify the way these methods were implemented. This includes adding new classes and methods, changing the names of existing methods, and moving classes and routines to other (existing or new) classes and components. For example, the size() method of the FastVector class of Weka 3.4 has been replaced by the utility routine size() implemented in the built-in Collection interface class of the Java package java.util. Consequently, the size() method was included in patterns in Weka 3.4 but did not appear in the extracted patterns of Weka 3.7 since it is an external utility method of that system.

Another difference we have observed when examining the patterns and the source code of both versions is that some invoked methods have been moved to new classes that were introduced in Weka 3.7. For example, the method hasMoreElements() was part of the FastVectorEnumeration class of the old version but moved to the new class WekaEnumeration in the new version. Refactoring the code in this manner leads to the discovery of patterns that begin from the same parent nodes in both version of the trace, but diverge in the lower levels.

The above discussion demonstrates the usefulness of using a pattern-directed approach not only to measure the similarity between two versions of the same system but also to guide the process of investigating the root causes of dissimilarities.

4. Conclusions and Future Directions

In this paper, we presented a new approach for comparing the implementation of different features in subsequent versions of the same system. Our method discovers similarities between two traces generated from the different versions of the software. In particular, we focused on calculating the trace correlation based on all the behavioural patterns extracted from the execution traces. Future work includes the need to conduct

more experiments with multiple versions of a given software system to further assess the efficiency of our approach. There is also a need to compare our results with other software comparison techniques; this could lead us to expand our method by adapting some of their strengths.

References

1. Dunsmore, A., Roper, M., Wood,, M.: The role of comprehension in software inspection. In: Journal of Systems and Software, 2(3), pp. 121-129 (2000)
2. Martin, J., Mcclure, C.: Software Maintenance: The Problem and its Solutions. Prentice-Hall: Englewood Cliffs NJ (1983)
3. Pigoski, T. M.: Practical Software Maintenance: Best Practices for Managing Your Software Investment, John Wiley and Sons, New York NY, pp. 384 (1997)
4. Cornelissen, B., Moonen, L.: On Large Execution Traces and Trace Abstraction Techniques, Delft: Software Engineering Research Group, ISSN 1872-5392 (2008)
5. Hamou-Lhadj, A., Lethbridge, T. C.: Summarizing the content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In: Proceedings of the 14th IEEE International Conference Program Comprehension, pp. 181-190 (2006)
6. De Pauw, W., Lorenz, D., Vlissides, J., Wegman, M.: Execution Patterns in Object-Oriented Visualization. In: Proceedings of the 69 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, NM, pp. 219-234 (1998)
7. Systä, T.: Understanding the Behaviour of Java Programs. In: Proceedings of the 7th Working Conference on Reverse Engineering, pp. 214-223, (2000)
8. Jerding, D., Rugaber, S.: Using Visualization for Architecture Localization and Extraction. In: Proc. of the 4th Working Conference on Reverse Engineering, pp. 219-234 (1997)
9. Wilde, N., Scully, M. C.: Software Reconnaissance: Mapping Program Features to Code. In: Software Maintenance: Research and Practice, pp. 49-62 (1995)
10. Hamou-Lhadj, A., Lethbridge, T.: An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls. In: Proceedings of the 1st ICSE International Workshop on Dynamic Analysis (WODA), Portland, Oregon, USA (2003)
11. Weka 3: Data Mining Software in Java. www.cs.waikato.ac.nz/ml/weka/
12. Witten, I. H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, Morgan Kaufmann (1999)
13. http://www.eclipse.org/tptp/

Acknowledgement

This work is partly supported by the Natural Sciences and Engineering Research Consortium (NSERC), Canada.