Pattern-Based Trace Correlation Techniques for Software Evolution

Maher Idris

A Thesis

In

The Department

Of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science

(Electrical and Computer Engineering)

at

Concordia University

Montreal, Quebec, Canada

March 2011

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Maher Idris

Entitled: "Pattern-Based Trace Correlation Techniques for Software Evolution"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________________ Chair
Dr. D. Qiu

______________________________________ Examiner, External
Dr. O. Ormandjieva, CSE To the Program

______________________________________ Examiner
Dr. A. Agarwal

______________________________________ Supervisor
Dr. A. Hamou-Lhadj

Approved by: ______________________________________
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

____________20_____ ______________________________
Dr. Robin A. L. Drew
Dean, Faculty of Engineering and
Computer Science

ABSTRACT

Pattern-Based Trace Correlation Techniques for Software Evolution

Maher Idris

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 thesis, 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. Traces, however, tend to be considerably large. We propose in this thesis to compare them based on their main behavioural patterns. 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.

Acknowledgment

During my study, I had the privilege to work with some experts and experienced professionals, which was truly an enriching experience. I would like to thank everyone who has helped me and contributed to this research project during my entire graduate study.

Dr. Abdelwahab Hamou-Lhadj has been the ideal thesis supervisor. I am heartily thankful for his sage advice, insightful criticisms, guidance and patient encouragement that have aided the completion of this thesis from the initial to the final level of working on my research.

Also, I would like to thank all my friends, lab mates, and colleagues including Ali, Akanksha, Amir, Khalid and Luay who made themselves available whenever I needed them to support and help me pursue this degree by sparing their precious time and sharing the literature and their invaluable assistance.

Dedication

This thesis is dedicated to my parents, who are like the candles that always lighten my life to help me touch my goals and dreams. Their unconditional support and having the feeling of their spirits around me make me reach the desired ultimate success full of achievements throughout every single step of my life in innumerable ways.

Special thanks to my sisters and brothers. Without your love, encouragement and being there, I would not have finished my Master's degree.

No words to express my feeling and love towards my family.

Thank you…

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Problem and Motivation

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 [Dunsmore 00], which is recognized to account for almost 80% of the cost of the software life cycle [Martin 83, Pigoski 97].

One challenge that engineers constantly face while 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 [Dunsmore 00]. Documentation is normally the main source of information where answers to these questions should be found, but it has been shown in practice that documentation is rarely up to date if at all exists. The problem is further complicated by the fact that software engineers, the initial designers of the system often move to new companies taking with them valuable information about the system.

In this thesis, 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. We use execution traces (dynamic analysis) by exercising the target features of the system under study to identify the differences between the implementation of the features under study. Once these differences are identified, we refer to the source code (static analysis) to understand the causes of these differences.

Execution traces have been used in various studies to observe and investigate the behavioural aspects of a software system. Traces, however, have been found to be difficult to work with. This is due to the large size of typical traces. Although many trace analysis tools and techniques have been proposed (e.g., [Cornelissen 08, Hamou-Lhadj 05a, De Pauw 98]), none tackles the problem of comparing traces.

In this thesis, we propose a novel trace correlation algorithm by comparing traces based on their main behavioural patterns instead of a mere event-to-event mapping. We have also developed two metrics to calculate the similarity of the generated traces.

We focus in this thesis on traces of routine calls. We use the term routine to mean procedure, function, and method. Our approach is independent from the programming language used to develop the application as long as the language supports the concept of routines.

1.2 Research Contributions

The main contributions of this thesis are as follows:

- A novel trace correlation approach based on comparing traces based on their behavioural patterns. The patterns are extracted by varying different matching criteria.

- We introduce two trace correlation metrics to measure the similarity between two different system versions based on the extracted patterns from execution traces that represent the underlying features.

- We applied correlation metrics to execution traces generated from an object-oriented target software system to show the applicability of our approach.

1.3 Thesis Outline

The rest of the thesis is structured as follows:

- **Chapter 2 - Background:** This chapter consists of the literature review. We present background information, including a brief overview of related topics to our research, namely, program comprehension, software maintenance and evolution, reverse engineering, software effort estimation, feature location, trace abstraction, and static and dynamic analysis. A survey of existing techniques for comparing different versions of software system is proposed along with their advantages and limitations.

- **Chapter 3 – Trace Correlation Approach:** The trace correlation approach and metrics are presented in this chapter. The chapter starts by presenting the definition of an execution trace and behavioural patterns. This is followed by explaining the matching criteria used to generalize patterns. The chapter continues with an overview of the feature trace generation process. Next, we present the overall trace correlation approach by explaining the main two phases; the first phase consists of pre-processing the trace whereas the second phase consists of applying the trace correlation technique. In the end, we complete this chapter with a discussion on the applicability of the presented metrics.

- **Chapter 4 – Evaluation:** This chapter introduces the case study which is used to validate our trace correlation algorithm. Initially, we describe the target system on which we apply our approach. Then, the chapter covers the usage scenario, followed by applying the trace correlation algorithm to the generated traces. The results of quantitative and qualitative analyses are then presented. The chapter ends with a summary discussion concerning the evaluation process of applying the trace correlation metrics to the target system.

- **Chapter 5 – Conclusion:** We conclude the thesis in this chapter. We revisit the main contributions and introduce future work directions. The chapter ends with our closing remarks.

Chapter 2 Background

In this section, we present the related topics to our work. These topics include program comprehension, software maintenance and evolution, reverse engineering, software effort estimation, feature location, trace abstraction and summarization, and finally static and dynamic analysis.

2.1 Software Maintenance and Evolution

Software maintenance and evolution is an important area in software engineering and also the most costly. Brooks states that over 90% of the usual system cost is spent on maintenance, and that any part of the system that is successfully implemented will inevitably need to be maintained [Brooks 95].

Parnas defines software maintenance as *"Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. ... (We must) lose our preoccupation with the first release and focus on the long term health of our products."* [Parnas 94]. According to IEEE, software maintenance is defined as all the modifications made to a program that are performed after the delivery to correct discovered problems, improve performance or to keep it adaptable and usable in the changing or changed environment [ANSI/IEEE Std].

There exist four types of maintenance activities [Lientz 80, ISO/IEC, Pfleeger 98]:

- **Corrective Maintenance:** This activity comprises all modifications of a software system that are performed after the delivery to fix and correct faults that caused the system to fail.
- **Adaptive maintenance:** All changes to the existing system after delivery to keep it usable in the new environment so as to meet new requirements.
- **Preventive Maintenance:** This activity consists of all the corrections and detections that might take place in the system to prevent failures before even they occur.
- **Perfective Maintenance:** This maintenance type comprises improvements and enhancements made to an existing program to improve its performance and maintainability.

Yau et al. have proposed in their research that the software maintenance process is comprised of four phases, and when a specific maintenance objective is set up, it can go through these four phases to be accomplished [Yau 80]. Figure 2.1 shows the maintenance process along with its four phases.

In conclusion, all the systems need to be maintained as they evolve with time, which often leads to many versions of the system to be released. This leads to the need to understand the differences between subsequent versions of a system in order to estimate the effort required to maintain the new versions. In this thesis, we tackle the challenging issue of understanding how subsequent versions of the same system vary.

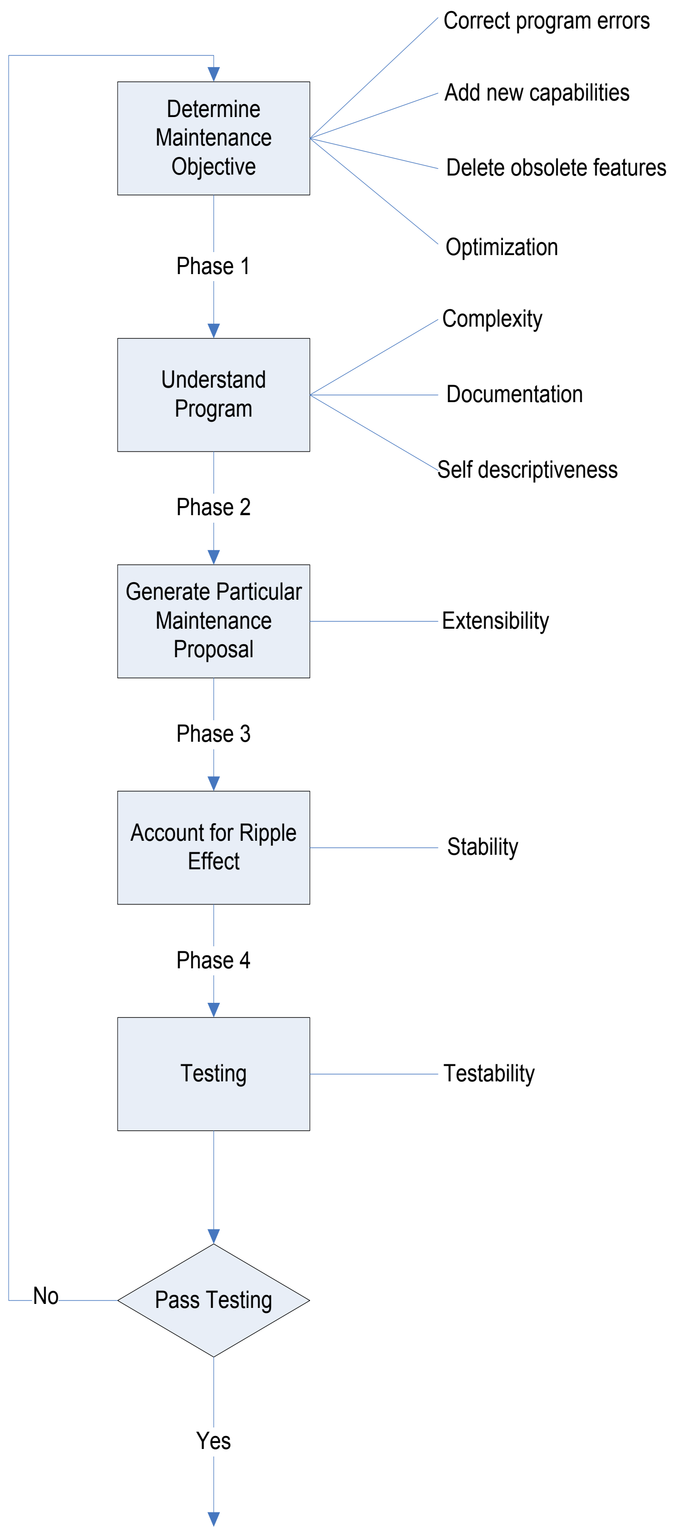


Figure 2.1 Software Maintenance Process (taken from [Yau 80])

2.2 Program Comprehension

"A person understands a program when he or she is able to explain the program, its structure, its behaviour, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program"

Biggerstaff et al. [Biggerstaff 93]

Program comprehension is defined by Rugaber as the process of obtaining knowledge about a software system under study to reach a certain level of insight to facilitate tasks such as fixing or correcting the system's bugs and errors, system enhancements, reusing and recovering the documentation [Rugaber 95]. This comprehension can be acquired through analyzing the system's static and dynamic aspects, features, and documentation [Ng 04]. Fjeldstad and Hamlen have reported that 50% of the time and effort is dedicated to program understanding during a maintenance task [Fjeldstad 83]. This is also supported by Standish, who states that *"if maintenance costs 70-90 percent of the life cycle, and understanding occupies 50-90 percent of maintenance cost* [Lientz 78]*, program understanding time may be the dominant time in the entire software life cycle and thus the dominant cost."* [Standish 84].

According to Mayrhauser et al., during the program comprehension process, software engineers often use existing knowledge about the software system in order to come up with new knowledge that would be considered as part of the system knowledge to ultimately meet the targets of a code cognition task [Mayrhauser 95]. They identified two types of knowledge that developers might have:

- **General Knowledge:** This type represents the knowledge gained from the past experience in the domain of software engineering. It is completely independent from the program that engineers try to comprehend.
- **Software-Specific Knowledge:** This is the second type of knowledge that developers can possess which stands for their deep insight and level of understanding the software application under study.

In addition, a study has been conducted by Littman et al. in which they have specified strategies for program comprehension [Robson 91]. In the experiment, they have showed the relationship between these strategies and summarized their experiment with main conclusions that there are two fundamental approaches to program comprehension:

- **Systematic Approach:** where the maintainer examines the entire program and elicits the interactions between the different modules. This is achieved before any attempt to perform modifications or changes to the program.
- **As-needed Strategy:** where the maintainer tries to reduce the amount of study prior to perform any modifications that take place in the system. Consequently, he or she tries to locate the section of the program which needs to go under the maintenance process to commence the modifications.

In fact, the systematic approach is applicable to small programs only, where larger program may require an as-needed approach to be adopted instead. Erdös et al. have accomplished a study which states that partial comprehension of complex programs is enough to perform maintenance process [Erdös 98]. Many legacy systems are so large and complex to be entirely comprehended regardless of the forms used for representation and yet they need to be maintained. In other words, it is not necessary for programmer to fully understand the program to perform maintenance. It is only necessary to comprehend parts of the program affected by the maintenance request. Our work supports this idea as we only focus on particular features of a system that need to be understood.

2.3 Reverse Engineering

The major tools used by engineers to assist and ease the process of program comprehension are reverse engineering tools. Nelson stated that reverse engineering is related to explore tools and techniques which help software engineers comprehend legacy systems [Nelson 96]. The objective is to improve the productivity of the maintainers as they solve maintenance tasks. Chikofsky et al. define reverse engineering as: *"the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction"* [Chikofsky 90]. Reverse engineering has many benefits such as handling the complexity of the system, recovering high-level models of the system and retrieving the missing information [Chikofsky 90, Biggerstaff 89].

Before we present the reverse engineering processes, we have to mention about the forward engineering besides the reverse engineering as both of them are fundamental concepts in the software system lifecycle. Forward engineering term is completely the opposite of reverse engineering term, and it is suggested to differentiate the traditional software engineering process from reverse engineering process. According to the study conducted by Chikofsky et al., we propose the relationship between the forward and reverse engineering in Figure 2.2.

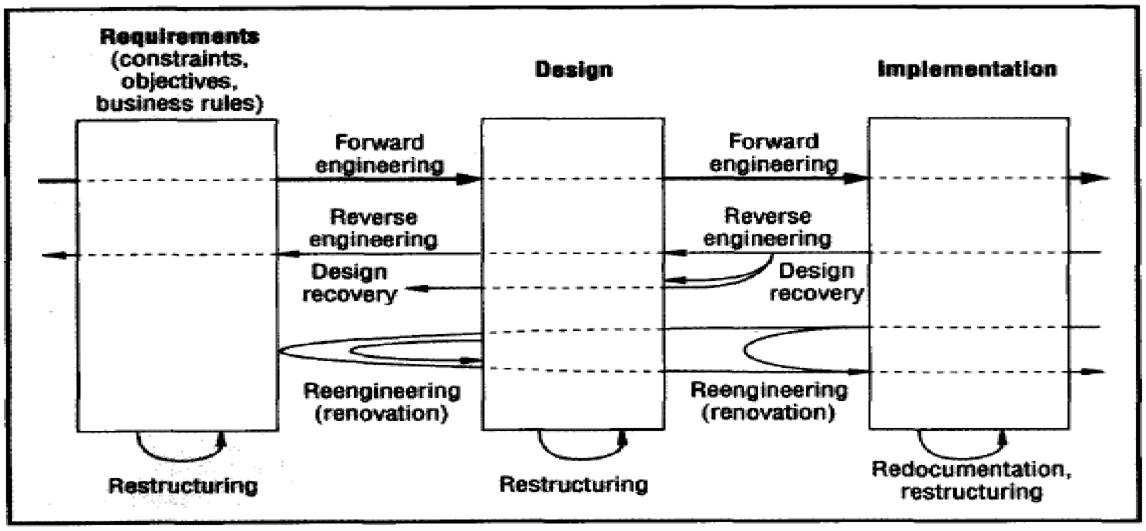


Figure 2.2 Relationship between terms; reverse engineering and forward engineering (taken from [Chikofsky 90])

As illustrated in Figure 2.2, the terms forward engineering and reverse engineering are defined as follows:

- **Forward Engineering:** is the traditional process for the typical software system where we move through the main phases starting from high-level abstractions and models to design phase and ending up the process in the last phase; the low-level implementation of the system. As shown in Figure 2.2, this process is comprised of steps to map the higher-level such as requirements to design and then to the low-level implementation [Chikofsky 90].
- **Reverse Engineering:** it is the opposite process of forward engineering. Reverse engineering is the analysis process of the subject system to categorize the components of the system and interrelationships between them and to show the system in a high-level abstraction. Figure 2.2 demonstrates the reverse engineering as a sequence of recovery steps to in the opposite direction, starting

from implementation through design and finishing at the high-level abstractions and models of the program [Chikofsky 90].

2.4 Software Development Effort Estimation

In software effort estimation, effort may refer to different things including time, cost and physical efforts. Software development effort estimation is the process of measuring in advance according to some specific techniques, the realistic effort need required to develop and maintain a software program to either complete it by adding new requirements or enhance it by deleting or modifying unnecessary or old features that might have errors, noise and uncertain functionality. The effort estimates can be used in many areas of software engineering, for example, project plans, iteration plans, budgets, investment analysis and for prices and bids purposes. Researches who work on the effort estimation are divided into groups depending on how they define the effort: time or cost. For example, a project might be concerned with estimating the required time more than the costs.

According to a research conducted by Boehm, cost and delivery time must be taken under consideration as coherent elements in the production of quality software that raises the customer level of satisfaction [Boehm 96].

The work we present in this thesis aims to help engineers estimate the actual effort with time and cost needed to maintain and evolve software systems by examining how newer versions of a system differ from older versions.

2.5 Static and Dynamic Analyses

Two major approaches of system analysis have been established: *Static analysis* and *dynamic analysis.*

1. **Static Analysis:** This type of analysis can be done without the need to execute the program. It is based on understanding the source code to provide a complete feedback on the system. Static analysis process examines the program code to derive all properties for all possible executions [Ball 99]. The static information gained by performing the static analysis of software systems expresses the structure of the software system under study with all its behavioural aspects and features that can be invoked in any system execution.
2. **Dynamic Analysis:** Dynamic analysis, the focus of this thesis, is the process of analyzing all the data gathered by executing a program according to a certain scenario to understand the run-time behaviour of the software system. Ball has defined it as "Dynamic analysis is the analysis of the properties of a running program" [Ball 99]. In contrast to static analysis, dynamic analysis examines the running program (through program instrumentation) to derive the properties that hold for one or more program executions (executed scenarios).

Dynamic analysis is control-based environment which depends on the program inputs such as using certain features in a particular scenario as well as the program outputs that reflect the program behaviours. Dynamic analysis involves instrumenting a program under investigation to record its runtime events. The run-time information or run-time events typically take the form of execution traces.

Static and dynamic analyses have been considered to be complementary techniques in many dimensions including completeness, scope and precision [Ball 99]. In addition, using both of them helps in the comprehension process of the program and speed up the maintenance activities. For this reason, we make use of both types of analysis techniques in this thesis. However, dynamic analysis occupies the bigger proportion of this thesis compared to static analysis. Nevertheless, static analysis is still important and useful so as it is needed to validate some of the information used to compare traces generated from different versions of a system.

2.6 Trace Abstraction and Summarization

Traces are known to be hard to work with due to the large size of typical traces, often millions on lines long. Therefore, there is a need to find ways to reduce their size while keeping as much of their essence as possible. Trace abstraction and summarization techniques aim to achieve this objective. We surveyed the most cited techniques in what follows:

2.6.1 Pattern Detection Techniques

Trace patterns are defined as non-contiguous repetitions of the same sequence of events. [De Pauw 98, Hamou-Lhadj 03a]. De Pauw and Hamou-Lhadj proposed techniques to reduce the trace size generated from the target system by extracting the patterns which can be used to reflect the main events invoked in a trace. Maintainers can focus on understanding these patterns instead of reading the whole trace.

2.6.2 Sampling

In the context of execution traces, a sample means a representation of part of the trace that consists of a specific number of events based on the sampling parameters [Chan 03, Whaley 00, Dugerdil 07]. Sampling is a good technique for reducing the initial trace size but suffers from the challenging task of defining proper sampling parameters that can be generalized to other applications.

2.6.3 Grouping

Grouping is considered as summarization technique since it summarizes and compresses the monotone subsequence of execution traces into one event, which results in considerable saving of space and allow up to two dozen feature traces to be presented simultaneously. Kuhn et al. introduced a grouping technique, namely, monotone subsequence summarization that represents entire traces as signals in time [Kuhn 06]. They describe their approach as a trace signal which is composed of monotone subsequences separated by pointwise discontinuities. The technique cuts the signal at the pointwise discontinuities to create the monotone subsequences and compress each one into a summarized one method-call chain. Therefore, the summarized chain is much shorter that the original trace signal.

Utilities removal is another trace summarization technique presented in [Hamou-Lhadj 06], Hamou-Lhadj et al. built a trace summarization algorithm that consists of removing utility components from large traces. The authors argued that utilities clutter trace content without adding much information. They proposed a way to detect automatically utility components by analyzing the source code. Once the utilities are identified, they devised a technique that filters out these utilities from traces to reduce their size without loss of important information.

2.6.4 Visualization

Many studies have been conducted for visualizing information about the behaviour of the target system by focusing on features of interest. Visualization is an efficient technique for engineers who want to achieve some activities concerning particular parts of the source code or the system's attributes including large traces [Hamou-Lhadj 04, De Pauw 93]. Limitations of using these techniques are summarized in the need of user intervention as well as the inability to reuse various visualization techniques in other tools.

Chapter 3 Trace Correlation Approach

3.1 Traces of Routine Calls

An execution trace is a record of what goes on when executing the target system, usually by exercising its features according to specific user-scenarios. There are different types of traces including traces of inter-process communication, statement-level traces, routine call traces, and so on. In fact, one can trace about anything in the system that can help the task at hand. Traces have been used in many software engineering areas where there is a need to gain insight of the behavioural aspects of a software system.

In this thesis, we focus on traces of routine calls. Traces of routine calls have been shown to be useful for program comprehension tasks [Hamou-Lhadj 04]. A trace of routine calls is a tree structure consisting of subsequent calls of routines that are represented as tree nodes. Each node is considered as a root for its subtree. The routine that invokes other methods is called *parent* while the invoked method in this case becomes a *child*. In addition, if the node is the last routine call, namely, the end of the tree or subtree, then it is named *leaf*. An example of a trace of routine calls is depicted in Figure 3.1.

Figure 3.1 also shows a relationship between the depth limit and nesting level; the higher the depth, the higher the nesting level. Each node has a specific label to identify the name of the routine. This name can be a full name of the routine, which consists of the package, class name (for object-oriented systems), and the routine name.

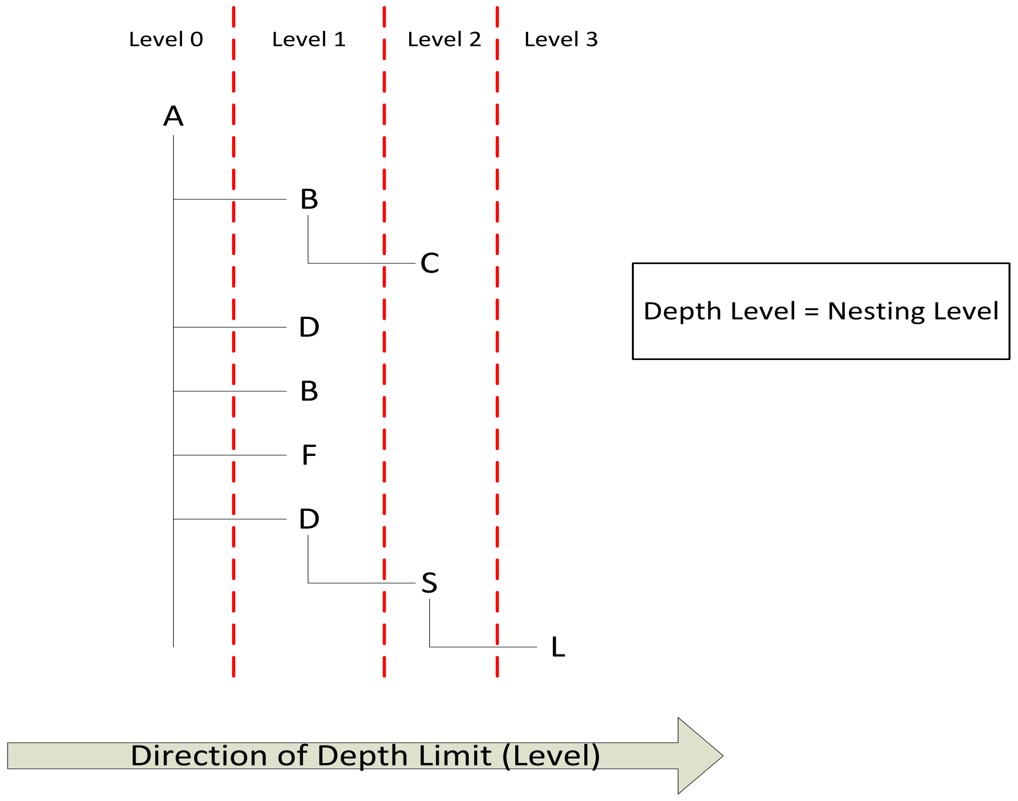


Figure 3.1 An example of routine (method) call trace

There exist many mechanisms for generating execution traces, among which code instrumentation is the most popular one. This technique is automatic and supported by many tools; many of them are available as open source. Code instrumentation consists of inserting probes into the code. A probe can be seen as a print out statement. When the instrumented code is executed, a log file is generated. To generate traces of routine calls, we need to have at least two probes for each routine: One probe at the entry of the routine and another one at the exit. Figure 3.2 illustrates the typical process of generating routine call traces through code instrumentation.

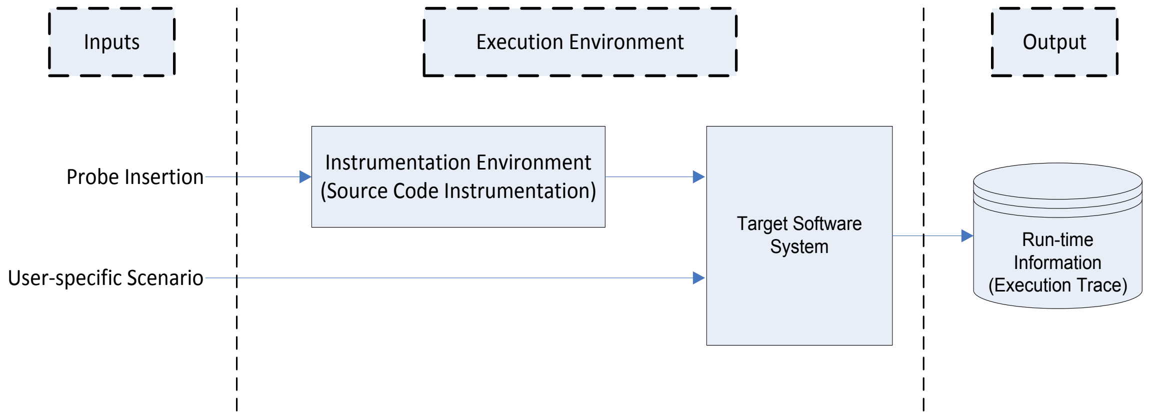

Inputs
Execution Environment
Output
Probe Insertion
User-specific Scenario
Instrumentation Environment (Source Code Instrumentation)
Target Software System
Run-time Information (Execution Trace)

Figure 3.2 Execution Trace Generation Process

3.2 Trace Correlation Approach

Figure 3.3 shows a general overview of our approach for comparing traces generated from subsequent versions of the same system. Both versions of the system are first instrumented and run using the same usage scenario. The generated traces go then through two main phases (see Figure 3.3). The first phase consists of pre-processing the traces by removing continuous repetitions and noise in the trace caused by the presence of low-level utility components. The second phase consists of comparing the traces resulting from Phase 1.

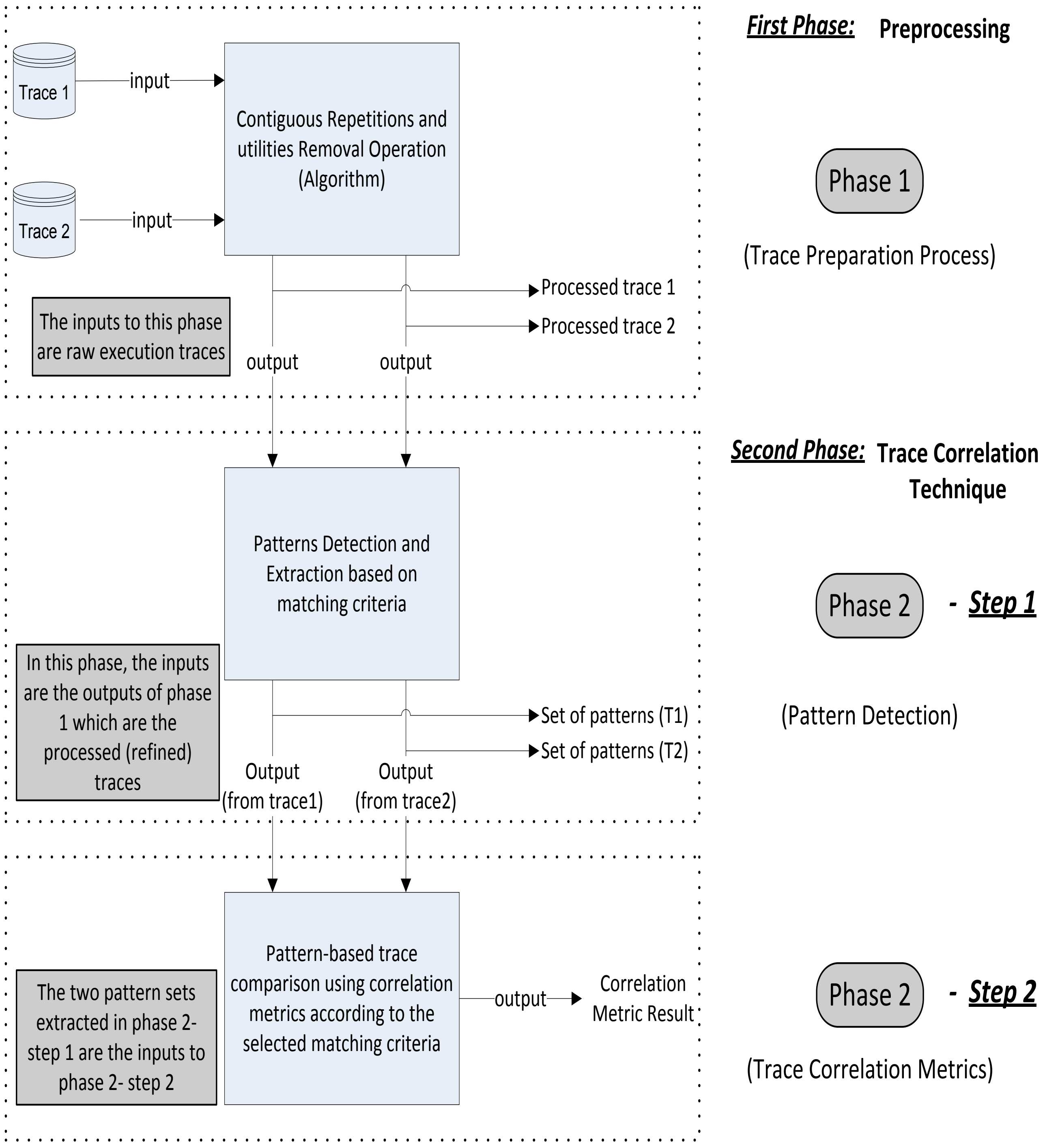

First Phase: Preprocessing
Trace 1
input
Trace 2
input
Contiguous Repetitions and utilities Removal Operation (Algorithm)
Processed trace 1
Processed trace 2
The inputs to this phase are raw execution traces
output
output
Phase 1
(Trace Preparation Process)
Second Phase: Trace Correlation Technique
Patterns Detection and Extraction based on matching criteria
Phase 2 - Step 1
(Pattern Detection)
In this phase, the inputs are the outputs of phase 1 which are the processed (refined) traces
Set of patterns (T1)
Set of patterns (T2)
Output (from trace1)
Output (from trace2)
Pattern-based trace comparison using correlation metrics according to the selected matching criteria
output
Correlation Metric Result
The two pattern sets extracted in phase 2- step 1 are the inputs to phase 2- step 2
Phase 2 - Step 2
(Trace Correlation Metrics)

Figure 3.3 Overall Approach Diagram

There are different ways for comparing traces. Perhaps the most naive one is to compare the traces line by line. This is often ineffective due to the fact that many events could have been invoked in different orders due to the existence of threads. In this thesis, we propose a novel approach for comparing traces based on their main behaviour. These behaviours are represented in the form of trace patterns as we will describe in Section 3.2.2.

3.2.1 First Phase: Trace Pre-processing

As we mentioned before, a trace is first pre-processed to reduce its complexity. During this step, the raw traces go through the trace preparation process where we first filter out utility routines such as accessing methods (sets and gets). We rely on naming conventions to identify utilities. For example, any routine that starts with ‘set’ or ‘get is automatically removed. Also, in some cases, we refer to the system folder structure to identify packages that serve as utilities. Any routines that belong to these packages are also removed from the trace. These utilities clutter the trace without adding much information to its content. Hamou-Lhadj et al. showed that effective analysis of a trace should include a utility removal stage that cleans up the trace content from noise [Hamou-Lhadj 06]. The second pre-processing step consists of removing contiguous repetitions due to the presence of loops and recursion. An example of applying the pre-processing phase is shown in Figure 3.4 where utility routines are first removed then the contiguous repetitions as well.

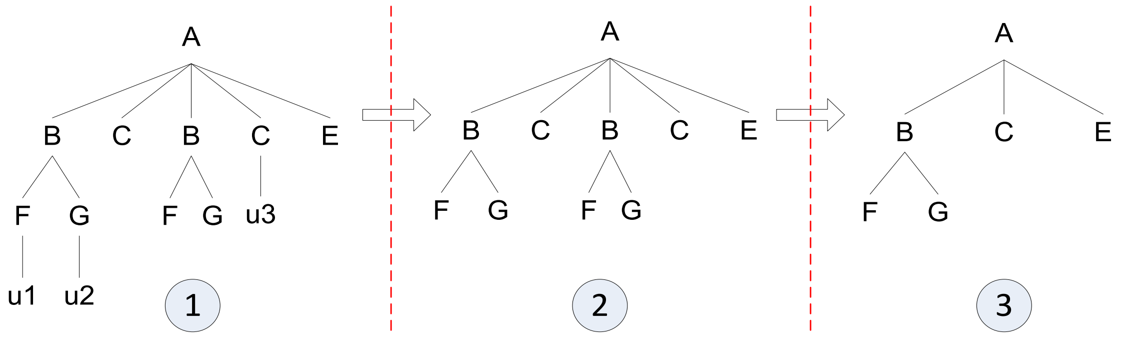


Figure 3.4 1) Raw Trace. 2) Trace after removing utilities; we assume utilities start with ‘u’. 3) Trace after removing contiguous repetitions.

3.2.2 Second Phase: Trace Correlation Technique

In this section, we describe the second phase of our approach which is the trace correlation technique. Comparing traces based on their events is rather ineffective since the events can occur in different orders due to threading and other factors such as the presence of noise (utilities). It is therefore important to investigate another unit of comparison. In this thesis, we propose comparing traces based on the main behaviours they embed. These behaviours are reflected in the trace in the form of trace patters [DePauw 04, Hamou-Lhadj 06]. A trace pattern is defined as a sequence of events that is repeated non-contiguously in the trace. Trace patterns have been used in other studies to help software engineers understand the key aspects of a trace (e.g. [Jerding 97b]). The trace correlation phase is comprised of two main steps: The pattern detection step and the trace correlation measure. The idea is to take the two traces in question, extract their behavioural patterns, and compare the extracted patterns using different similarity measures. Two traces exhibit the same behaviour if the pattern sets are similar.

3.2.2.1 Pattern Detection

As mentioned earlier, we define a trace pattern as a sequence of events that is repeated non-contiguously in the trace. This translates into non-contiguous repetitions of similar subtrees in a trace of routine calls. Hamou-Lhadj et al. have proposed a very efficient algorithm for automatically detecting such patterns in large traces [Hamou-Lhadj 03b].

To reduce the number of patterns, 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 [De Pauw 98]. In the following subsections, we present the most common matching criteria used to generalize patterns.

A. Identity

The identity matching criterion is the basic and simplest criterion. Two sequences of calls are considered similar if their corresponding subtrees are isomorphic. In other words, they have the same labels, structure, order of calls and topology [De Pauw 98, Hamou-Lhadj 03b]. Identical matching can result in a large number of patterns that might differ only slightly. Figure 3.5 shows an example of the identity matching criterion.

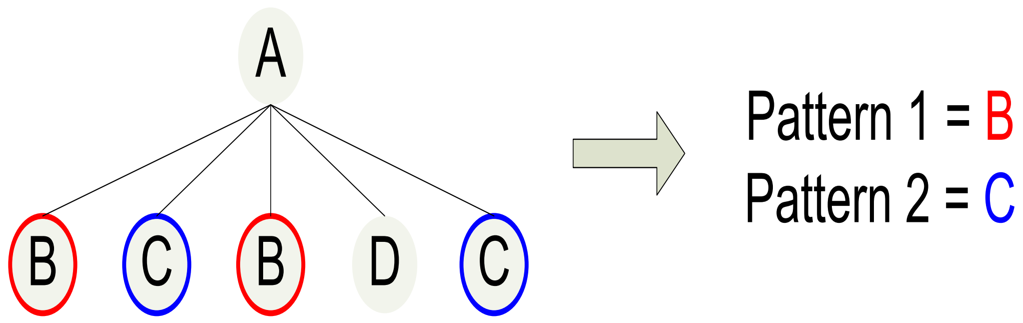


Figure 3.5 Detecting Patterns using Identity Matching Criterion

B. Ordering

The idea behind this criterion, which is also called the commutativity criterion [De Pauw 02], is to consider two sub-trees as similar if they have the same method calls and number of calls no matter the order in which the calls occur. Figure 3.6 illustrates an example of the ordering criterion. In this figure, the subtrees rooted at B are considered similar if the order in which the child routines occur is not taken into account.

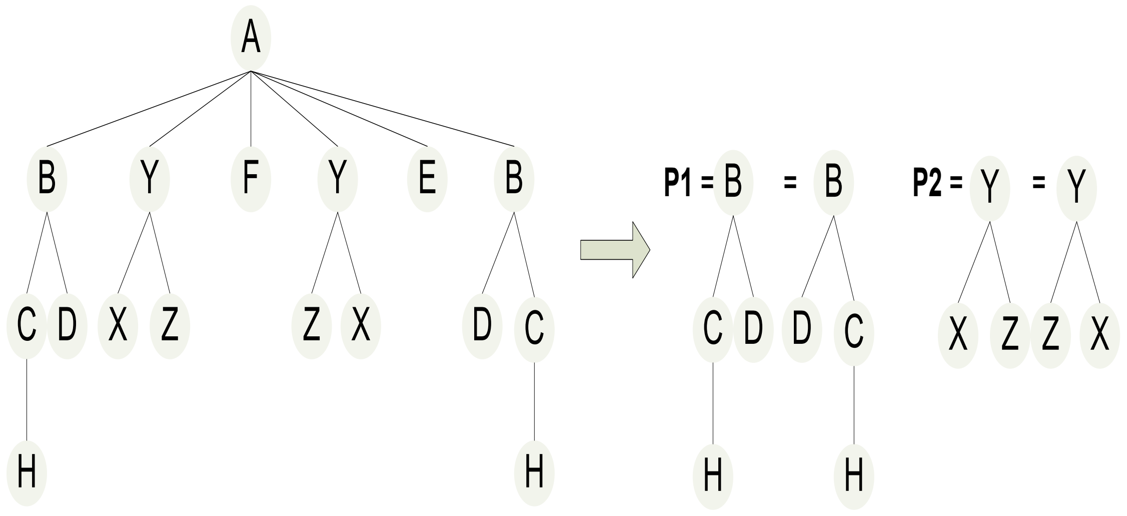


Figure 3.6 Example of detecting patterns using the 'ordering' criterion

C. Depth-limiting

Using this criterion, two subtrees are considered similar if they have the same method calls with the same order at specific depth. The rest of the methods that go beyond this level are ignored and not taken into account. Figure 3.7 shows how this criterion can be used to consider the subtrees B as similar if the depth of comparing sequences of calls is limited to level 1.

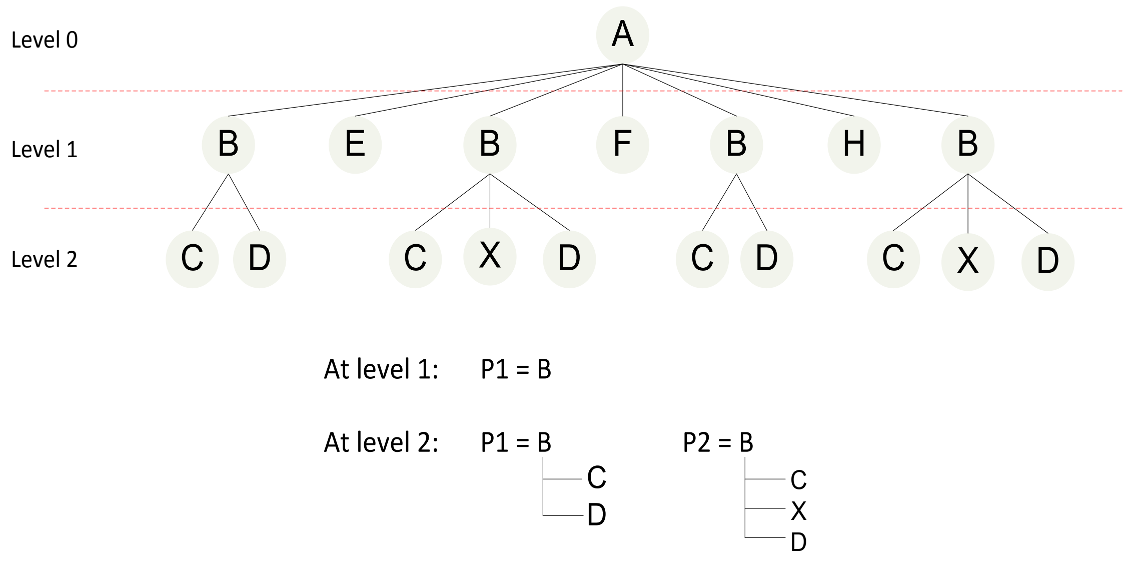


Figure 3.7 Detecting Patterns using Depth-Limiting Criterion

D. Set

The set matching criterion is about ignoring the order of method calls in a specific subtree as well as assuming each method is called only once even if it is called many times in the subtree; all redundancies are ignored. Figure 3.8 shows an example where two patterns can be detected rooted at B and E respectively by treating the sequences of calls rooted at these nodes as a set.

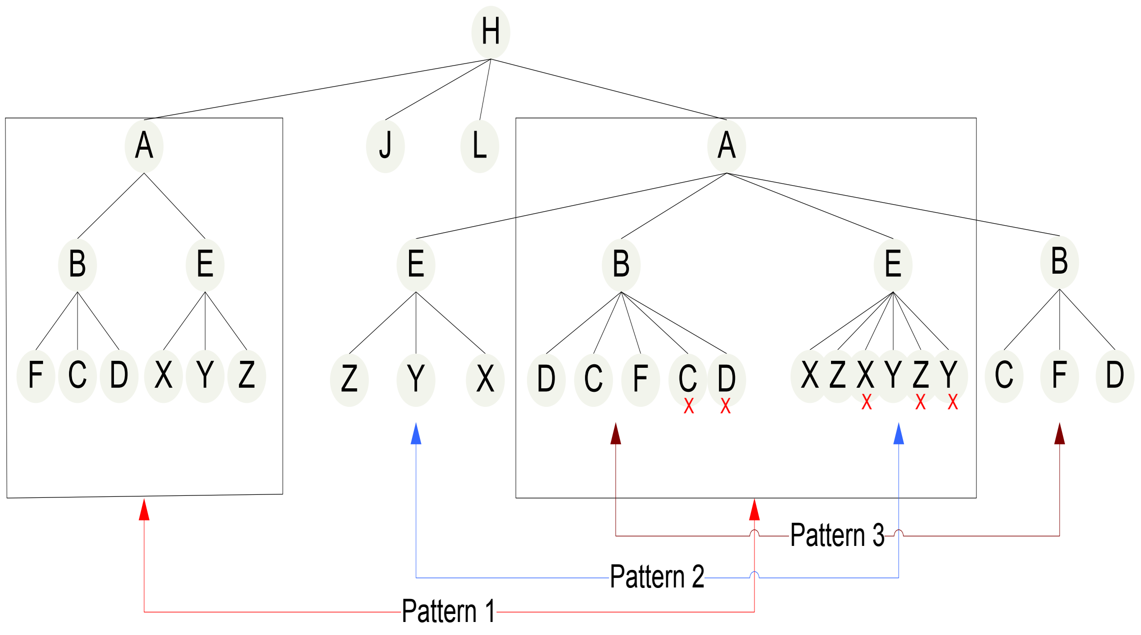


Figure 3.8 Extracting Patterns using Set Criterion

E. Flattening

The flattening criterion ignores the hierarchical structure of the sequences of calls to be compared [De Pauw 02]. It lines up all the method calls in a linear structure where the routine call is occurred and mentioned only once regardless of the number of repetitions for each routine invoked in the subtree. This is perhaps the most extreme way to group sequences into instances of the same pattern. Figure 3.9 shows an example of using the flattening criterion.

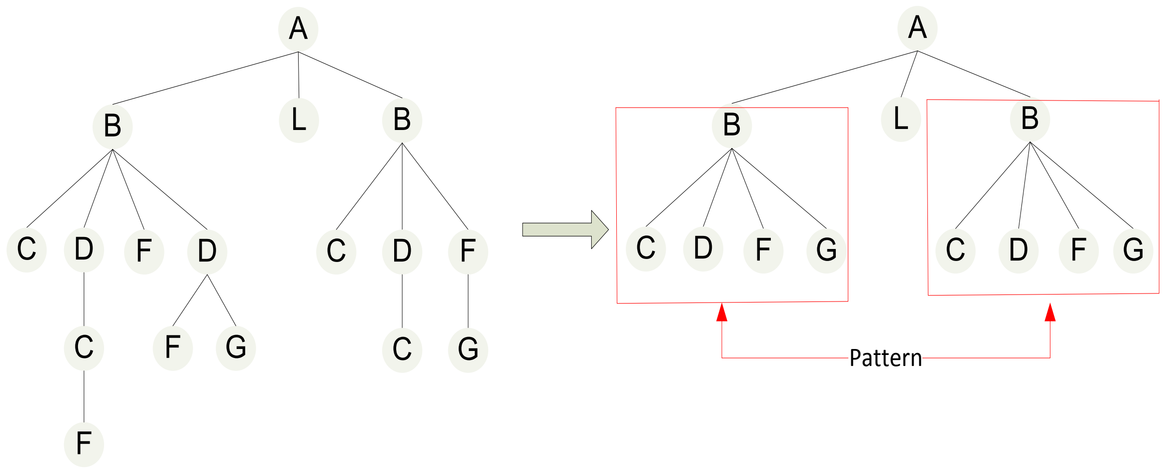


Figure 3.9 Extracting Patterns using Flattening Criterion

F. Combination of Matching Criteria

The above matching criteria can be combined in various ways. In this thesis, we propose using two new criteria. The first one is "Set-Depth" which is a combination of the set and depth-limiting criteria. The second one is "Ordering-Depth" which is a combination of ordering and depth-limiting. The set-depth criterion considers two sequences as similar using the set matching criteria applied to a certain depth. Figure 3.10 shows an example. In this figure, depending on the depth, we can distinguish different patterns. At level 1, we can see one pattern rooted at B with three sequences. At level 2, we have also one pattern with only two sequences. The subtree B in the middle cannot be considered as a sequence of this pattern using the set-depth criterion by setting the depth to 2.

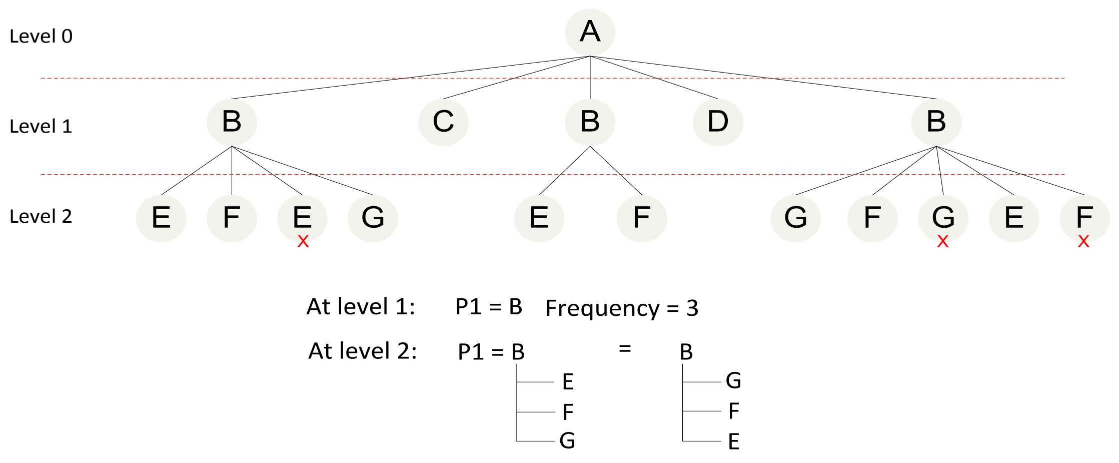


Figure 3.10 Extracting Patterns using Set-Depth Criterion

The ordering-depth criterion is similar to the depth-limiting criterion except that instead of treating the calls as a set we only ignore the order of calls. Figure 3.11 shows an example of applying this criterion.

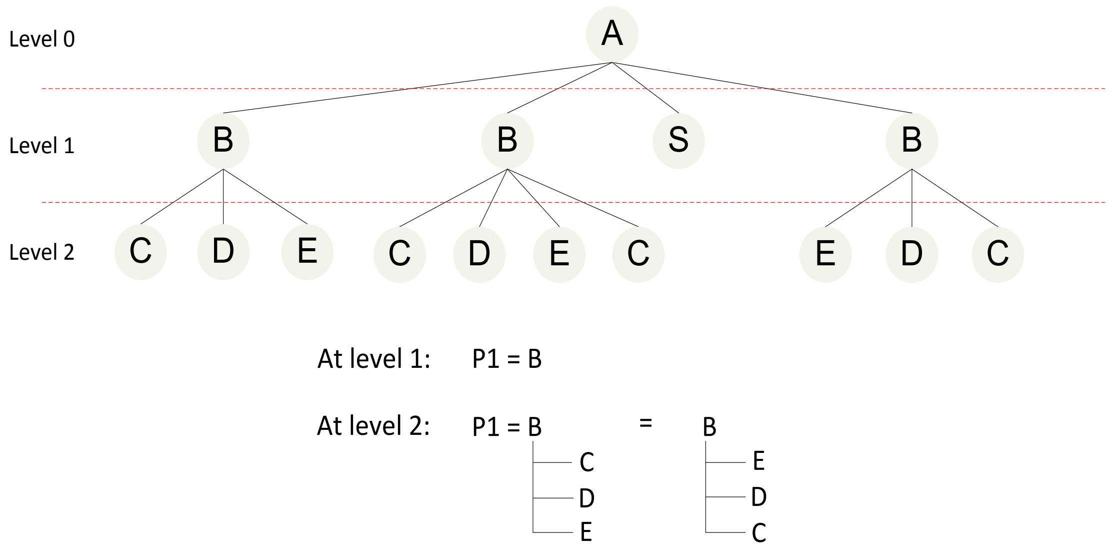
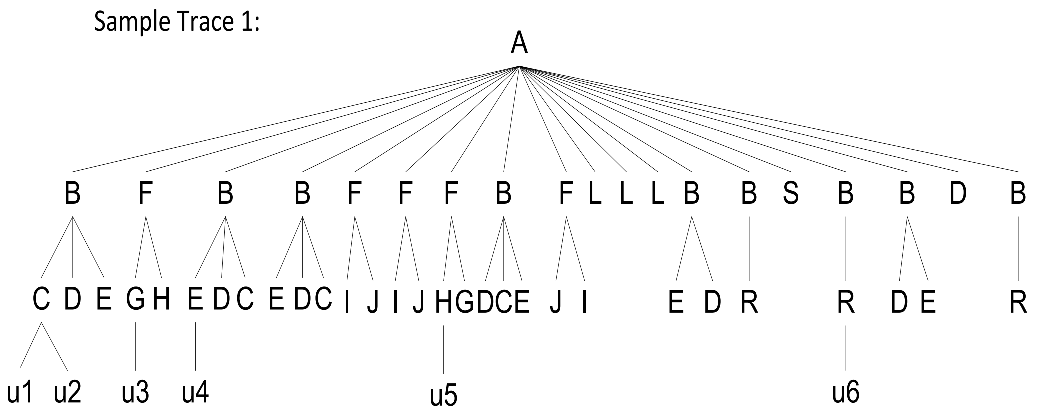


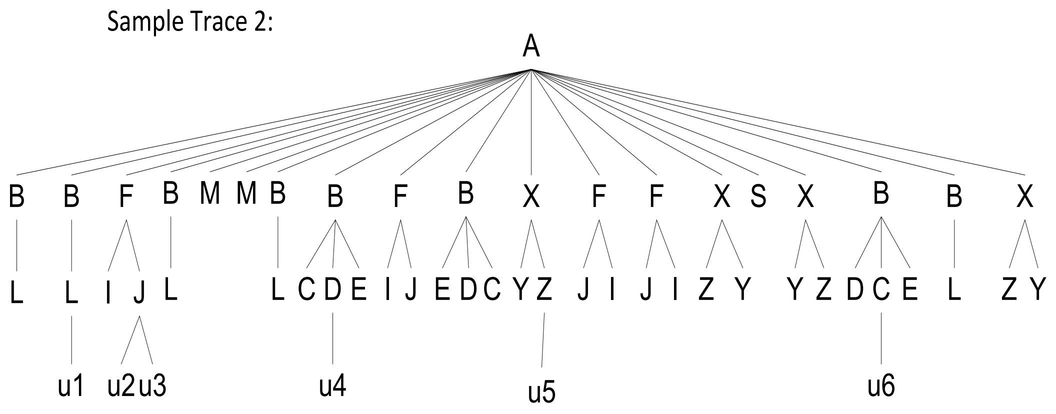
Figure 3.11 Extracting Patterns using Ordering-Depth Criterion

In [Hamou-Lhadj 03b], 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 technique adopted the idea of the algorithm with some modifications and improvements. These improvements include providing the ability for using and applying more than one matching criteria to extract the similar patterns as desired. Besides this, we implemented the two new combined matching criteria introduced earlier. For example, we can use the combined matching criteria at a time or two separate criteria one after another to gain the precise similar patterns from the list of similar patterns resulting from applying the first criterion.

An example of application of the pattern extraction technique on the refined sample traces 1 and 2 of Figure 3.13 are shown in Tables 3.1 and 3.2. Figure 3.12 shows the same sample traces, but before we apply the first phase of our approach, namely, pre-processing (raw traces). In this example, one matching criterion is used to detect and extract the similar patterns from the refined ones, which is ignoring the order of calls (Ordering matching criterion).

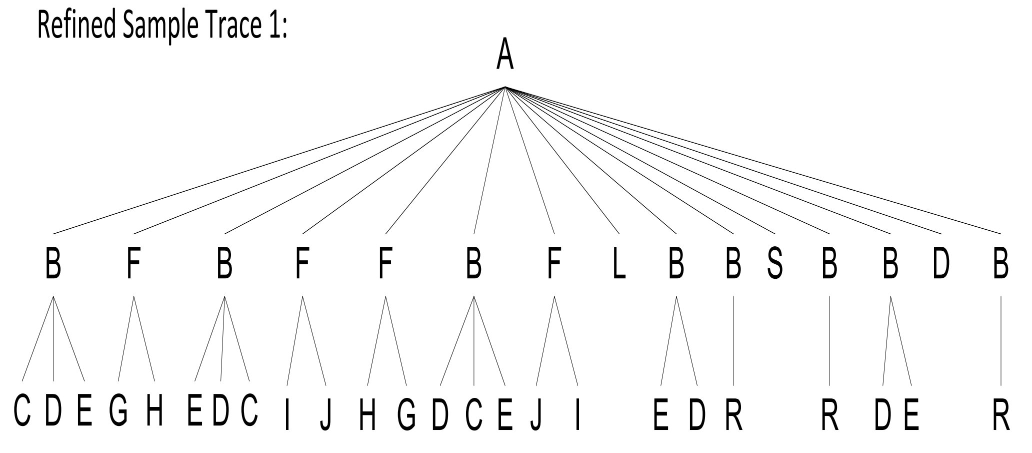


Sample Trace 1:
A
B F B B F F F B F L L L B B S B B D B
C D E G H E D C E D C I J I J H G D C E J I E D R R D E R
u1 u2 u3 u4 u5 u6



Sample Trace 2:
A
B B F B M M B B F B X F F X S X B B X
L L I J L L C D E I J E D C Y Z J I J I Z Y Y Z D C E L Z Y
u1 u2 u3 u4 u5 u6

Figure 3.12 Two Raw Sample Routine (method) Call Traces Example


Refined Sample Trace 1:
A
B F B F F B F L B B S B B D B
C D E G H E D C I J H G D C E J I E D R R D E R

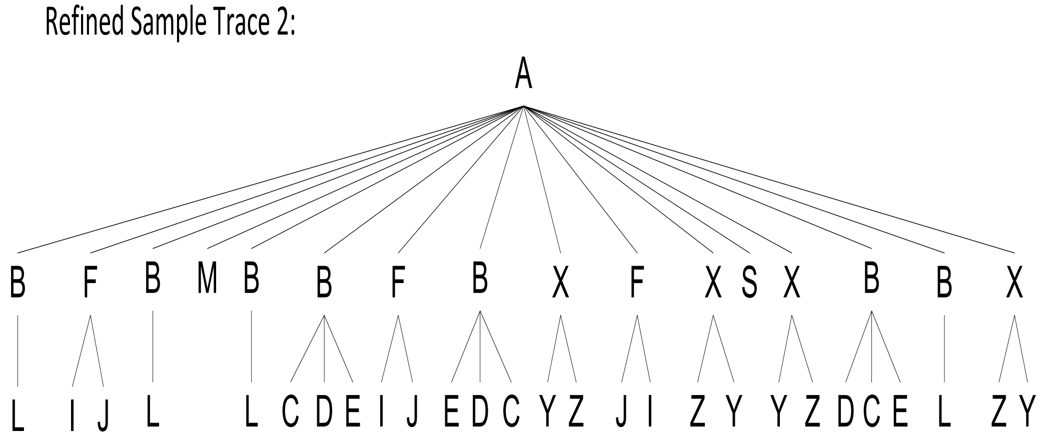

Refined Sample Trace 2:
A
B F B M B B F B X F X S X B B X
L I J L L C D E I J E D C Y Z J I Z Y Y Z D C E L Z Y

Figure 3.13 Two Refined Sample Routine (method) Call Traces Example

Table 3.1 Similar Patterns extracted from Sample Trace 1 of Figure 3.13

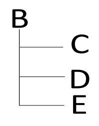
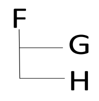
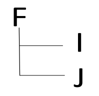
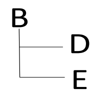
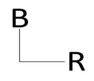
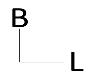
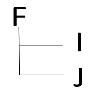
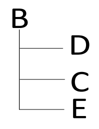
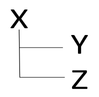
Pattern Number	Trace 1 Pattern Content	Frequency
1		3
2		2
3		2
4		2
5		3

Table 3.2 Similar Patterns extracted from Sample Trace 2 of Figure 3.13

Pattern Number	Trace 2 Pattern Content	Frequency
1		4
2		3
3		3
4		4

The outputs of this step are two sets of extracted patterns from two execution traces. These pattern sets will be used in the next step to measure the similarity between these sets. We explain the trace correlation metrics step in the next section.

3.2.2.2 Trace Correlation Metrics

In this section, we present two 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: Non-weighted trace correlation metric and the weighted trace correlation metric.

Non-weighted Trace Correlation Metric:

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

$$NW\_TCM(T1, T2) = \frac{\left(\frac{CPtrnN}{T1TotalPtrnN}\right) + \left(\frac{CPtrnN}{T2TotalPtrnN}\right)}{2}$$

Where:

- CPtrnN: Total Number of Common Similar Patterns of both Traces.
- T1TotalPtrnN *and* T2TotalPtrnN: Total Number of Patterns of Trace 1 and Trace 2 respectively.

Weighted Trace Correlation Metric:

The weighted trace correlation metric, W_TCM, improves over 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:

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

Where:

- T1CPtrnFreqN and T2PtrnFreqN: The frequency of occurrence in each trace of the patterns shared between traces T1 and T2.
- CPtrnN: Total number of similar patterns contained in both Traces.
- T1TotalPtrnN and T2TotalPtrnN: Total number of patterns of Trace T1 and Trace T2 respectively.
- T1TotalFreqN and T2TotalFreqN: Total number of frequencies of Trace T1 Patterns and Trace T2 patterns respectively.

Both correlation metrics range between 0 and 1. The traces are similar if the metric converges to 1. They are completely dissimilar if the metric is close to 0. The number of common similar patterns will never exceed the total number of all patterns in each trace. The same applies to the total number of frequencies of common similar patterns which is always less than or equal to the total number of frequencies of patterns related to each trace.

We illustrate the application of these metrics on the examples of Figure 3.12. After we performed the patterns detection algorithm, we obtained two sets of patterns, among which the common patterns are extracted. Figure 3.13 shows the patterns extracted from Traces 1 and 2 as well as the common patterns between the two traces. Notice that we used in this step the "Ordering" matching criterion.

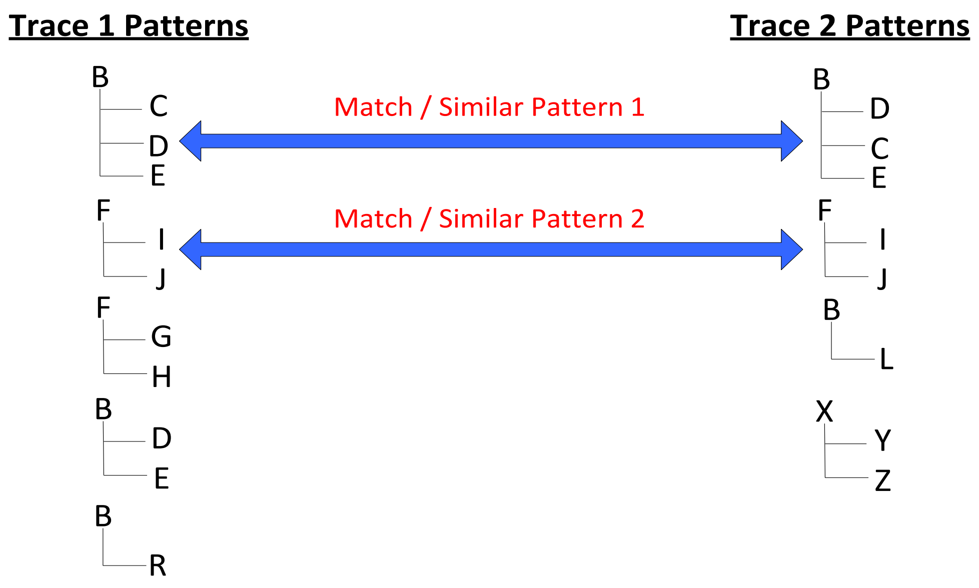


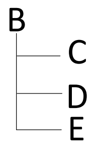
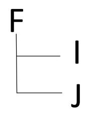
Figure 3.14 Extracting the Final Set of Similar Patterns of Sample Traces T1 and T2 (two sets of extracted patterns) of Figure 3.13

Now, all the relevant information needed to perform and compute the two correlation metrics is available to be used in the next process. Table 3.3 shows the number of patterns and their frequencies in the traces of Figure 3.12.

Table 3.3 The Properties of the Two Sample Traces of Figure 3.13

Properties / Sample Traces	Total Number of Patterns	Total Number of Pattern Frequencies
Trace 1	5	12
Trace 2	4	14

Table 3.4 Final Set of Common Patterns of Two Sample Traces of Figure 3.13

Pattern Number	Content of Similar Pattern	Frequency	
		Trace 1	**Trace 2**
1		3	3
2		2	3
Total number of frequencies of similar patterns		**5**	**6**

The results of applying the non-weighted and weighted correlation metrics are as follows:

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

$$W\_TCM(T1,T2) = \frac{\left(\frac{5}{12} \times \frac{2}{5}\right) + \left(\frac{6}{14} \times \frac{2}{4}\right)}{2} = 0.19 = 19\%$$

In view of the fact that it is almost half of each sample trace patterns considered similar in this example, the result obtained by applying NW_TCM seems to be reasonable. The outcome of applying this metric has resulted in 45% similarity between the feature traces Trace 1 and Trace 2. Using the W_TCM metric, we take the frequency into account as well. The result obtained was 19% due to the fact that the frequency of the common patterns is much less than the total number of frequencies of the Traces 1 and 2 patterns. Moreover, the number of common patterns is less than the half in Trace1.

3.3 Summary

In this chapter, we presented our approach of comparing two traces based on their main behavioural patterns. We discussed the various matching criteria used to measure the extent by which sequences of events can be deemed similar. This is because identical matching alone would result in many patterns that differ only slightly. We also introduced two metrics that measure the similarity between two traces based on the number of common patterns they have, the weighted correlation metric and the non-weighted correlation metric, which vary whether the frequency of the patterns is taken into account or not. In the next chapter, we show the applicability of our approach on traces generated from a real system.

Chapter 4 Evaluation

4.1 Target System

We have applied the proposed trace correlation algorithm to traces generated from two versions of the same Java-based software system called Weka [WEKA]. The versions of Weka that have been selected for this case study are versions 3.4 and 3.7. Weka is 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. Weka version 3.4 is comprised of 55 packages, 732 classes, 8980 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).

We selected the Weka system because it is popular (well-known) and also well documented. The Weka system framework and its components including packages and the most important classes are documented in a book dedicated to the tool and machine learning in general [Witten 99]. In addition, Weka enjoys an active online community which resulted in many documents and tutorials made available on the Weka official website [WEKA]. Some of these documents contain overviews and detailed description of the Weka architecture to help developer use and improve the tool if need be. In this research, we used Weka's documentation to validate some of the results obtained by applying our approach.

4.2 Usage Scenario

4.2.1 Feature Selection

We have applied our trace correlation techniques to a specific software feature supported in both versions of Weka, which is the use of the J48 classification algorithm used for machine learning to construct efficient decision trees. Our choice of J48 was motivated by the fact that it is a feature available in both versions and that it does not require extensive knowledge on how to trigger it. The application of our approach aims to reveal the similarities or differences in the way each version implements the J48 algorithm and if there are any major change in the newer version of Weka with respect to this algorithm.

4.2.2 Generation of Feature-Traces

In order to generate the execution traces that correspond to the selected feature for this case study, we instrumented Weka using TPTP Eclipse plug-in (the Eclipse Test and Performance Tool Platform Project). TPTP is an open source platform which allows the software developers to build test and performance tools. The detailed description of this tool can be found on the website and the entire information of the plug-in and its download is provided on [Eclipse TPTP]. Probes were inserted at each entry and exit method (including constructors) of the intended system in order to instrument it including all the invoked routines that are specific to the scenario chosen to examine Weka.

For the feature discussed in the previous section, we generated two execution traces, which correspond to the same selected feature, by executing the two instrumented versions of Weka. We used a sample input data provided in the documentation and the source code package (folder) of Weka system to exercise the J48 feature in Weka versions 3.4 and 3.7.

4.3 Applying the Trace Correlation Algorithm

The first step of the algorithm is preprocess the traces by filtering out utilities such as get and set methods as well as removing contiguous repetitions. We also removed from each trace the methods responsible for generating the graphical interface and initializing the Weka environment. The removal of parts of the trace that are concerned with initializing the Weka environment was necessary so as to focus on only parts of the traces concerned with the implementation of the J48 algorithm, since the objective of the study is to understand the variation that may occur in both versions of Weka with respect to this algorithm.

In Table 4.1, we show statistical information regarding the size of the traces before and after the preprocessing stage. We can see the removal of contiguous repetitions and utilities reduces considerably the size of raw traces. But the resulting traces are still in the order of thousands of calls, which are still hard for humans to comprehend manually. The size of the initialization part turned out to be very small compared to the size of the raw traces.

Table 4.1 The Execution Traces of Weka System for Versions 3.4 and 3.7

Weka Version / Properties of Execution Traces	Weka V. 3.4	Weka V. 3.7
Original (raw) Trace Size	35,974	103,009
Original Trace Size after Removing Contiguous Repetitions	6,850	26,978
Initialization Trace Size	5,919	17,534
Initialization Trace Size after Removing Contiguous Repetitions	682	1,288
Original Trace Size after Removing Initialization Phase (Initialization Trace)	5,510	24,700

Note that the information reported in this table is in ordered steps where we can see that the original traces went through several processes starting by removing the contiguous repetitions, followed with the removal of the initialization part. Since we eliminated all contiguous repetitions out of original traces, we did the same thing for the generated initialization traces as well.

Also, we can see in Table 4.1 that the size of the J48 trace in Weka 3.7 is considerably higher than the size of the J48 trace generated from the older version Weka 3.4. This indicates that new enhancements have been made to this algorithm in the newer version. We further exploited this aspect using our pattern detection algorithm.

The second step was to apply the pattern detection algorithm. We used the “ordering” matching criterion during the extraction process. Future work should focus on experimenting with other matching criteria to study their impact on the final result. The last step is to apply the correlation metrics to measure the differences between the two pattern sets extracted from the Weka traces. In the next subsequent sections, we present the quantitative and qualitative analysis of the results.

4.3.1 Quantitative Analysis

Table 4.2 shows the number of extracted patterns from each trace and the total number of similar patterns in both traces.

Table 4.2 Behavioural Patterns of Two Execution Traces of Two Weka Versions

Weka Version	**Matching Criteria**	**Number of Patterns**	
		All Extracted Patterns	**Similar Patterns of Two Versions**
Weka 3.4	Ordering (Ignore Order)	162	64
Weka 3.7	Ordering (Ignore Order)	299	

As we see in Table 4.2, the total number of patterns that belong to Weka 3.7 is almost the double the total number of patterns of Weka 3.4. This result shows that the implementation of the J48 algorithm in Weka has undergone several changes from Weka 3.4 to Weka 3.7. Moreover, the similar patterns of both traces which are 64 patterns are less than the half of the total patterns relevant to the used versions of target system (i.e. all extracted patterns of Weka 3.4 and 3.7 that are 162 and 299, respectively). This number of similar patterns compared with the whole patterns influences the final results of the correlation metrics.

Table 4.3 shows the results of applying the correlation metrics to the patterns of both traces of Weka. The results show that both traces are considerably different (NW_TCM = 31%, W_TCM = 5%).

Table 4.3 Results of Running Trace Correlation Metrics on Execution Traces

Trace Correlation Metrics	Results
NW_TCM (t1,t2)	**30.45%**
W_TCM (t1,t2)	**5%**

To be able to justify these differences, we examined the patterns that are not common between the two traces by exploring the source code of the two Weka versions. This qualitative analysis is presented in the next section.

4.3.2 Qualitative Analysis

The dissimilarity between the two versions in terms of the total number of all extracted 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 is 656, whereas the number of distinct methods in the trace generated from Weka 3.7 has 1024 distinct methods. This has led to the generation of many patterns that are in one trace and not in another trace (patterns triggered by the new methods). By exploring the source code of both versions, we found that many of these methods have been introduced in newer versions of Weka starting from Weka 3.7. Table 4.4 shows an example of methods that we either newly invoked only in the trace of Weka 3.7 (new in scenario), and not in Weka 3.4 but they existed in the source code of both versions of the system or new methods that were introduced in the source code of Weka 3.7 and therefore did not exist in Weka 3.4 (new in source code and scenario of Weka 3.7).

Table 4.4 New Method Samples of Weka 3.7 in scenario or scenario and source code

Method Number	New in Scenario	Method Number	New in Source Code and Scenario
1	`weka.classifiers.evaluation.ThresholdCurve.makeInstance`	1	`weka.gui.explorer.ClassifierPanel.updateCapabilitiesFilter`
2	`weka.classifiers.evaluation.ThresholdCurve.makeHeader`	2	`weka.core.Capabilities.clone`
3	`weka.core.Memory.isOutOfMemory`	3	`weka.core.AbstractInstance.numClasses`
4	`weka.core.Utils.checkForRemainingOptions`	4	`weka.core.DenseInstance.value`
5	`weka.core.Utils.splitOptions`	5	`weka.core.WekaEnumeration.nextElement`
6	`weka.classifiers.evaluation.NominalPrediction.distribution`	6	`weka.classifiers.Evaluation.weightedFalsePositiveRate`

After we gathered all the information regarding the patterns including the distinct methods and the independent patterns related to each trace, two inspection strategies have been achieved to validate the results of our approach. The first one is for the similar patterns in two versions with respect to the parent roots and the second one is for the new patterns of Weka 3.7. For the first inspection strategy, we detected various patterns in both pattern sets that can be deemed to be similar in terms of having the same first method call but their content is different. In other words, we investigated these similar patterns which only share the equivalent parent node of their subtrees.

We studied some of these patterns and discovered that many refactoring has 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 or moving the classes and routines to other existing or new classes and components. For example, the `size()` method in `FastVector` class of Weka 3.4 is changed to be considered as utility routine in Weka 3.7 and replaced by the `size()` method implemented in the `Collection` interface which is a built-in class in Java package `java.util`. There are many `size()` methods in Java classes that are invoked according to the type of the predefined object that calls the right one according to its type that can be List, ArrayList, etc. Thus, the `size()` method did not appear in the extracted patterns of Weka 3.7 since it is not traced and logged in the corresponding execution trace while it is considered as a utility method of the system components and external routine to the Weka project.

Another example of what we have observed in examining the patterns and the source code of both versions is that some invoked methods have been moved to new classes introduced in Weka 3.7. The method named `hasMoreElements()` was in the `FastVectorEnumeration` class of the old version while it was shifted to a new class called `WekaEnumeration` in the new version. `AbstractInstance` and `DenseInstance` are other examples of new classes added to the `Core` package of the new version that contain new methods as well. Table 4.5 shows few samples of these detected patterns that share the same parent node but different implementation due to refactoring of the code.

As we can see in Table 4.5, each pair seems similar according to the first method call but their contents are different. In the following, we explain each pair of patterns in addition to their responsibilities in the source code.

Table 4.5 Samples of Patterns with the Same Parent Nodes

Pattern Number	Pattern	
	V 3.4	V 3.7
1	• **weka.core.Instances.attribute** • weka.core.FastVector.elementAt	• **weka.core.Instances.attribute** • weka.core.Instances.numAttributes • weka.core.Instances.attribute • weka.core.Attribute.name • weka.core.Instances.attribute
2	• **weka.classifiers.trees.J48.Distribution.add** • weka.core.Instance.classValue • weka.core.Instance.classIndex • weka.core.Instances.classIndex • weka.core.Instance.value • weka.core.Instance.weight	• **weka.classifiers.trees.J48.Distribution.add** • weka.core.AbstractInstance.classValue • weka.core.AbstractInstance.classIndex • weka.core.Instances.classIndex • weka.core.DenseInstance.value • weka.core.AbstractInstance.weight
3	• **weka.classifiers.Evaluation.makeDistribution** • weka.core.Instance.isMissingValue	• **weka.classifiers.Evaluation.makeDistribution** • weka.core.Utils.isMissingValue
4	• **weka.classifiers.trees.J48.C45Split.weights** • weka.core.Instance.isMissing	• **weka.classifiers.trees.J48.C45Split.weights** • weka.core.AbstractInstance.isMissing • weka.core.DenseInstance.value • weka.core.Utils.isMissingValue

The first pattern in version 3.4 consists of two method calls while its matching pattern in version 3.7 includes five method calls. In version 3.4, the `attribute()` method has one parameter of integer data type that invoked another routine called `elementAt()` located in the class `FastVector`. By referring to the source code of the two versions, the `attribute()` method that was called is not the same as the one invoked in its corresponding pattern of version 3.4 since its parameter is of string data type. It calls many other methods to achieve its task starting with `numAttributes()` and then `attribute()` which is the same one as in the corresponding pattern in the old version. It continues with the `name()` procedure and ends with calling again the `attribute()` method. The responsibility of the version 3.4 pattern is to return an attribute while the pattern of version 3.7 responsibility is to return an attribute given its name. If there is more than one attribute with the same name, it returns the first one, otherwise it returns null if the attribute cannot be found.

According to the second pair of patterns, if we try to analyze it, we will find that both of them have the same first method call, namely, `add()` with same parameters as well which are integer and Instance data types. `Add()` invokes the method `classValue()` in both versions but with one difference which is in the old version the `classValue()` is located in class `Instance` while it is shifted to new class called `AbstractInstance` in the new one. Then, `classValue()` calls `classIndex()` from class `Instance` in version 3.4 while it is called from the class `AbstractInstance` in version 3.7. The subsequent methods `value()` and `weight()` are retrieved in both patterns where their place in Weka 3.4 is the class `Instance` while they are placed in the new classes in Weka 3.7; `DenseInstance` and `AbstractInstance`, respectively. Both of them have the same responsibility (functionality) in the system which is to add a given instance to a given bag.

Now, if we take a look at the third pair of patterns in the source code of Weka versions, we notice that their parent method which is `makeDistribution()` consists of one parameter with the data type double. It invokes the method `isMissingValue()`

which is located in the class `Instance` regarding the old Weka while it is moved to another class in Weka 3.7 called `Utils`.

We finally examined the last matching patterns in the above table and found that the method `weights()` with one parameter of Instance data type invokes the routine `isMissing()` from the class `Instance` in old version when it is invoked from the new class `AbstractInstance` in the new Weka. In version 3.4, another procedure has been called by the routine `isMissing()` but not traced since it is a routine of built-in class in Java named `isNaN`. The routine `isMissing()` in the new version retrieves `isMissingValue()` which is a procedure of class `Utils`. It invokes another method which is `value()` located in the new class `DenseInstance`. Their functionality is to return weights if instance is assigned to more than one subset and return null if instance is only assigned to one subset.

We also studied the patterns that were introduced in Weka 3.7 and not in Weka 3.4. Table 4.6 shows the results of these patterns and their corresponding source code methods.

As shown in this table, many patterns are triggered by methods that are new in Weka 3.7. The responsibilities of the same new pattern samples presented in the above table are shown in Table 4.7. Notice that each pattern is summarized into only one method call which is the first (parent) method of its entire routine calls.

Table 4.6 Samples of New Patterns in Execution Trace of Weka 3.7

Pattern Number	Pattern	Method Calls			
		New in Source Code of Weka 3.7	New in Scenario of Weka 3.7	Existed in both Scenarios of Weka versions (3.4 , 3.7)	Existed in both Source Code of Weka Versions (3.4 , 3.7)
1	`weka.classifiers.evaluation.ThresholdCurve.makeInstance`		✓		✓
	`weka.core.DenseInstance.<init>`	✓	✓		
	`weka.core.AbstractInstance.<init>`	✓	✓		
2	`weka.classifiers.evaluation.ThresholdCurve.makeInstance`		✓		✓
	`weka.core.Utils.missingValue`	✓	✓		
	`weka.core.DenseInstance.<init>`	✓	✓		
	`weka.core.AbstractInstance.<init>`	✓	✓		
3	`weka.core.AbstractInstance.classIsMissing`	✓	✓		
	`weka.core.AbstractInstance.classIndex`	✓	✓		
	`weka.core.Instances.classIndex`			✓	✓
	`weka.core.AbstractInstance.isMissing`	✓	✓		
	`weka.core.DenseInstance.value`	✓	✓		
	`weka.core.Utils.isMissingValue`	✓	✓		
4	`weka.core.DenseInstance.copy`	✓	✓		
	`weka.core.DenseInstance.<init>`	✓	✓		
	`weka.core.AbstractInstance.<init>`	✓	✓		
	`weka.core.Memory.isOutOfMemory`		✓		✓
	`weka.core.AbstractInstance.weight`	✓	✓		
5	`weka.core.DenseInstance.freshAttributeVector`	✓	✓		
	`weka.core.DenseInstance.toDoubleArray`	✓	✓		
6	`weka.classifiers.evaluation.ThresholdCurve.makeHeader`		✓		✓
	`weka.core.FastVector.<init>`			✓	✓
	`weka.core.Attribute.<init>`			✓	✓
	`weka.core.ProtectedProperties.<init>`			✓	✓
	`weka.core.Attribute.<init>`			✓	✓
	`weka.core.FastVector.addElement`			✓	✓
	`weka.core.Instances.<init>`			✓	✓
	`weka.core.Attribute.name`			✓	✓
	`weka.core.Instances.numAttributes`			✓	✓
	`weka.core.Instances.attribute`			✓	✓
	`weka.core.Instances.numAttributes`			✓	✓

Table 4.7 The Responsibilities of New Pattern Samples of Table 4.6

Pattern Number	Pattern	Responsibility
1	`weka.classifiers.evaluation.ThresholdCurve.makeInstance`	Creates an instance out of the given data.
2	`weka.classifiers.evaluation.ThresholdCurve.makeInstance`	Creates an instance out of the given data.
3	`weka.core.AbstractInstance.classIsMissing`	Tests whether the class of an instance is missing or not (returns true if the instance's class is missing).
4	`weka.core.DenseInstance.copy`	Generates a shallow copy of the instance as well as it has an access to the dataset as well.
5	`weka.core.DenseInstance.freshAttributeVector`	Clones the attribute vector of the instance and overwrites it with the clone.
6	`weka.classifiers.evaluation.ThresholdCurve.makeHeader`	Generates the header of an instance.

To support and validate the result obtained from the W_TCM, we retrieved all the frequencies that are related to the similar patterns in both traces as well as the total frequency numbers for all the extracted patterns of the whole traces; one for each Weka version. The frequencies for each version are shown in Table 4.8.

Table 4.8 The Frequency Values of both Traces of Weka Versions 3.4 and 3.7

Weka Version	Total Number of Frequency	
	Similar Patterns	**All Extracted Patterns**
Weka 3.4	411	1939
Weka 3.7	581	8389

The significant discrepancy in the frequency of patterns in both versions combined with the number of patterns that differ from one system to another contributes to the very low value of W_TCM (5%) obtained by comparing the traces of both systems.

4.4 Summary

In this chapter, we applied our trace correlation algorithm to traces generated by exercising two subsequent versions of the Weka systems using the same software feature. Our metrics revealed that the versions differ significantly in the way this feature is implemented. We showed by examining the source code that the newer version of the system contained many new methods that did not exist in previous versions. In addition, many changes were made to the methods of the older version. The changes consist of the emergence of new methods and/or the removal of existing ones introduces and validates these variant correlation results since many components and classes have been newly added or updated in the source code.

One of our metrics, NW_TCM, can be used to provide engineers with an estimation on the differences between the two scenarios. When applied to the Weka trace, it showed that the two traced scenarios were only 30% similar from one version to another. This can be exploited by software engineers to develop additional tests, or account for additional maintenance tasks. It also gives them an initiative that the source code of both selected versions of the system has many changes due to many activities such as refactoring techniques concerning the intended feature. These activities apparently influence the result obtained from the NW_TCM. The second metric W_TCM is very sensitive to the way the scenario is triggered (e.g. input data, etc) since the frequency of patterns often depends on the number processing needed to process the input data. We only recommend using it for stable scenarios that are not sensitive to the input data. W_TCM can be beneficial for software engineers who are into enhancing and working on the performance of the target system since it takes into account the number of repetitions for all the events occur in the execution trace of the user-specific scenario.

Chapter 5 Conclusion

5.1 Research Contributions

In this dissertation, we presented a new approach for comparing the implementation of same or different features in subsequent versions of the same system - Finding the similarity between two traces generated from two different versions of the same software system for the same feature under study. In particular, we focused on calculating the trace correlation and similarity based on all the behavioural patterns extracted from the execution traces using one or more matching criteria.

We introduced two metrics for measuring the similarity between two traces based on trace patterns, W_TCM, NW_TCM, which vary whether the number of occurrences of patterns is taken into account or not.

Our approach is comprised of two main phases: trace preprocessing and trace correlation technique. In the first phase, we prepared and refined the generated traces by removing utility methods invoked during the generation process as well contiguous repetitions due to the presence of loops and recursion.

The second phase, namely, trace correlation technique contains two steps: pattern detection and trace correlation metrics. In patterns detection step, we extracted all the behavioural trace patterns based on applying one or more of the existing and/or new matching criteria to the input traces. Many matching criteria have been introduced to generalize patterns instead of counting on identical matching which often results in large and yet similar patterns. In addition, we proposed some new matching criteria in this thesis to expand the area of study for more opportunities and give more options to extract different kinds of patterns upon request once we increase the number of matching criteria (see Chapter 3 – section 3.3). Once the patterns are extracted from both traces, they are compared using the trace correlation metrics (second step of this phase). In this step, all the similar patterns in both sets were extracted after we selected the matching criteria to be used to compare the two pattern sets of both versions. Consequently, all the similar patterns are essentially required to participate with their frequencies as the main factor in the correlation metrics to have the final similarity percentage.

Finally, we applied our approach to traces generated from two different versions of the same object-oriented software system, named Weka. The results obtained from the correlation metrics showed the differences in the implementation of the two features in each system. We validated our results by examining the Weka source code and documentation and found that the metrics reflected the differences that exist in the traces. We want to note that our approach is not complex and can be supported by tools since it does not need a lot of human intervention.

5.2 Opportunities for Further Research

Many ideas and directions are available to be employed for future research in the context of trace correlation approach. This future work includes the need to conduct more experiments with other different software systems that have more than one version to further assess the efficiency of our approach. Moreover, there is a need to apply the techniques on different types of execution traces such as statement-level traces that tend to be considerably larger than routine call traces. On the other hand, instead of focusing on the internal aspects of the feature under analysis, a study can be conducted to work on the external functional attributes of the feature whereas the similarity percentage might be completely different while we study the feature from different point of view. For example, considering all the method calls with their body (statement-level traces) will considerably increase the size of the generated traces and introduce new types of patterns that are completely different than those extracted from the routine call traces. Hence, while some patterns extracted by our techniques from routine call traces are considered different regarding the refactoring activities, they might be considered similar as one pattern since we go to another level of abstraction, a statement-level.

Finally, there is a need to compare our results with other software comparison techniques and perhaps end up with combined techniques that can reveal similarities and dissimilarities among feature implementation.

5.3 Closing Remarks

Understanding an entire software system is a significant challenge that maintainers and developers face when they develop, maintain and add new features to the target software system. Locating some features under study by referring to the source code could be very costly in terms of time and resources, specially, when the subject systems become more complex due to the increased code size. The main objective of trace correlation approach is to assist developers when trying to understand the difference between various versions of the same system using tracing techniques. This can help in many maintenance tasks such as estimating the cost and effort needed to perform maintenance task on newer versions of the system.

Bibliography

ANSI/IEEE Std	ANSI/IEEE Standard 729-1983.
Arnold 89	R. S. Arnold," Software restructuring", Proceedings of the IEEE 77, pp. 607–617, 1989.
Ball 99	T. Ball, "The concept of dynamic analysis", In Proc. 7th European Software Engineering Conference and ACM SIGSOFT Symp. on the Foundations of Software Engineering(ESEC/FSE), pp. 216-234, Springer, 1999.
Bennett 00	K. H. Bennett, and V. T. Rajlich, "Software maintenance and evolution: a roadmap", In Proc. of the Conference on the Future of Software Engineering, pp.73-87, 2000.
Biggerstaff 89	T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse", IEEE Computer, Volume 22, Issue 7, IEEE Computer Society, pp. 36-49, 1989.
Biggerstaff 93	T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding", In Proceedings of the 15th International Conference on Software Engineering, pp. 482-498, IEEE C.S., 1993.

Boehm 00	B. Boehm, C. Abts, and S. Chulani, “Software Development Cost Estimation Approaches – A Survey”, Annals of Software Engineering, 10: pp. 177-205, 1-Em, 2-Rganejwbfpnnthby, 3-Re, 4-Nr, 5-No, 2000.
Boehm 81	B.W. Boehm, “Software Engineering Economics”, Prentice-Hall, Englewood Cliffs, N J, 1981.
Boehm 96	B. W. Boehm, and Hoh In, “Identifying Quality- Requirements Conflicts”, IEEE Software, pp. 25-35, 1996.
Brooks 83	R. Brooks, “Towards a theory of the comprehension of computer programs”, International Journal of Man-Machine Studies, 18(6), pp. 542-554, 1983.
Brooks 95	F. Brooks, “The Mythical Man-Month”. Addison-Wesley, 1975 & 1995. ISBN 0-201-00650-2 & ISBN 0-201-83595-9.
Canfora 96	G. Canfora, L. Mancini, and M. Tortorella, “A workbench for program comprehension during software maintenance”, in: Fourth Workshop on Program Comprehension, IEEE Computer Society Press, Silver Spring, MD, pp. 30±39, 1996.
Chan 03	A. Chan, R. Holmes, G. C. Murphy, and ATT. Ying, “Scaling an object-oriented system execution visualizer through sampling”. In Proc. 11th Int. Workshop on Program Comprehension (IWPC), IEEE, pp. 237–244, 2003.

Chikofsky 90 E. J. Chikofsky, and J. H. Cross, “Reverse engineering and design recovery: A taxonomy”, IEEE Software, 7(1), pp. 13–17, 1990.

Coleman 94 D. M. Coleman, D. Ash, B. Lowther, and P.W. Oman, “Using Metrics to Evaluate Software System Maintainability,” Computer, vol. 27, no. 8, pp. 44-49, 1994.

Cornelissen 08 B. Cornelissen, and L. Moonen, “On Large Execution Traces and Trace Abstraction Techniques”, Delft: Software Engineering Research Group, ISSN 1872-5392, 2008.

De Pauw 02 W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the Execution of Java programs", In Proc. of the International Seminar on Software Visualization, LNCS 2269, Springer-Verlag, pp. 151-162, 2002.

De Pauw 93 W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, “Visualizing the Behaviour of Object-Oriented Systems”, In Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages,and Applications, ACM Press, pp. 326-337, 1993.

De Pauw 98 W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman, “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.

Dugerdil 07	P. Dugerdil, “Using trace sampling techniques to identify dynamic clusters of classes”, In proc. 2007 conference of the center for advanced studies on Collaborative research, pp. 306-314, 2007.
Dunsmore 00	A. Dunsmore, M. Roper, and M. Wood, “The role of comprehension in software inspection”, J. Syst. Softw. 52, 2-3, 121-129, 2000.
Eclipse TPTP	http://www.eclipse.org/tptp/
Erdös 98	K. Erdos, H. M. Sneed, "Partial Comprehension of Complex Programs (enough to perform maintenance)", IEEE Proceedings - Sixth International Workshop on Program Comprehension, pp. 24-26, 1998.
Fjeldstad 83	R. K. Fjeldstad, and W. T. Hamlen, “Application Program Maintenance Study: Report to Our Respondents,” In Proceedings GUIDE 48, Philadelphia, PA, Tutorial on Software Maintenance, G. Parikh and N. Zvegintozov, Eds., IEEE Computer Society, 1983.
Fowler 99	M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley, 1999.
Greevy 05	O. Greevy, and S. Ducasse, “Correlating features and code using a compact two-sided trace analysis approach”, In Proceedings of

CSMR 2005 (9th European Conference on Software Maintenance and Reengineering, pp. 314-323, 2005.

Guimaraes 83 T. Guimaraes, “Managing Application Program Maintenance Expenditure”, Comm. ACM, vol. 26, no. 10, pp. 739-746, 1983.

Hamou-Lhadj 03a A. Hamou-Lhadj, and T. Lethbridge, “Techniques for Reducing the Complexity of Object-Oriented Execution Traces”, In Proc. of VISSOFT, pp. 35-40, 2003.

Hamou-Lhadj 03b 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 (WODA), Portland, Oregon, USA, 2003.

Hamou-Lhadj 04 A. Hamou-Lhadj, and T. C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques”, In Proc. of CASCON 2004, 67 IBM Press, ACM Digital Library , Toronto, Canada, pp. 42-54, 2004.

Hamou-Lhadj 05a A. Hamou-Lhadj, T. Lethbridge, and L. Fu, “SEAT: A Usable Trace Analysis Tool”, International Workshop on Program Comprehension (IWPC), IEEE CS, St. Louis, Missouri, USA, pp. 157-160, 2005.

Hamou-Lhadj 05b A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, “Recovering behavioral design models from execution traces”, In

	Proceedings of the 9th European Conference on Software Maintenance and Reengineering, pp. 112-121, 2005.
Hamou-Lhadj 06	A. Hamou-Lhadj, and T. C. Lethbridge, "Summarizing the content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", In Proc. of the 14th IEEE International Conference Program Comprehension, pp. 181-190, 2006.
ISO/IEC	ISO/IEC 14764:2006, 2006.
Jensen 83	R. Jensen, "An Improved Macrolevel Software Development Resource Estimation Model", Proceedings 5th ISPA Conference, pp. 88-92, 1983.
Jerding 97a	D. Jerding, J. Stasko, and T. Ball, "Visualizing Interactions in Program Executions", in Proceedings of the 19th ICSE, Boston, Massachusetts, pp. 360-370, 1997.
Jerding 97b	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.
Jerding 98	D. F. Jerding, and J. T. Stasko, "The Information Mural: A Technique for Displaying and Navigating Large Information Spaces", In Proceedings of the IEEE Transactions on Visualization and Computer Graphics, vol. 4, pp. 257-271, 1998.

Jones 97	C. Jones, "Applied Software Measurement", 2nd Ed., McGraw-Hill, NY, 1997.
Kerievsky 04	J. Kerievsky, " Refactoring to Patterns", Addison-Wesley, 2004.
Kuhn 06	A. Kuhn, and O. Greevy, "Exploiting the analogy between traces and signal processing", 22nd IEEE International Conference on Software Maintenance (ICSM'06), 2006.
Lange 97	D. B. Lange, Y. Nakamura, "Object-Oriented Program Tracing and Visualization", IEEE Computer, Volume 30, Issue 5, pp. 63-70, 1997.
Lientz 78	B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance", Commun. ACM 21, pp. 466-471, 1978.
Lientz 80	B. P. Lientz, and E. B. Swanson, "Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations", Addison-Wesley, 1980.
Low 90	G. C. Low, and D. R. Jeffery, "Function points in the estimation and evaluation of the software process", IEEE Transactions on Software Engineering, 16(1), pp. 64-71, 1990.

Malony 91	A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski, “Traceview: A Trace Visualization Tool”, In the Proceedings of IEEE Software”, Illinois University, pp. 19-28, 1991.
Martin 83	J. Martin, and C. Mcclure, “Software Maintenance: The Problem and its Solutions”, Prentice-Hall: Englewood Cliffs NJ, 1983.
Mayrhauser 95	A. von Mayrhauser, and A. M. Vans, “Program comprehension during software maintenance and evolution”, IEEE Computer, pp. 44-55, 1995.
Nelson 96	M. L. Nelson, “A Survey of Reverse Engineering and Program Comprehension”, ODU CS-551 Software Engineering Survey, 1996.
Ng 04	D. Ng, D. R. Kaeli, S. Kojarski, and D. H. Lorenz, “Program Comprehension Using Aspects”, In Proceedings of the ICSE Workshop on Directions in Software Engineering Environments (WoDiSEE’2004), Edinburgh, Scotland, 2004.
Parnas 94	D. L. Parnas, “Software Aging”, In Proc. of the 16th International Conference on Software Engineering, pp. 279-287, 1994.
Pfleeger 98	S. L. Pfleeger, “Software Engineering: Theory and Practice”, Prentice Hall, Englewood Cliffs, New Jersey, 1998.

Pigoski 97 T. M. Pigoski, “Practical Software Maintenance: Best Practices for Managing Your Software Investment”, John Wiley and Sons, New York NY, pp. 384, 1997.

Pirzadeh 10 H. Pirzadeh, A. Agarwal, and A. Hamou-Lhadj, “An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension”, In Proc. of the 8th International Conference on Software Engineering Research, Management & Applications (SERA 2010), Montreal, Canada, 2010.

Putnam 92 L. H. Putnam, and W. Myers, “Measures for Excellence: Reliable Software on Time, Within Budget”, Englewood Cliffs, Yourdon Press Computing Series, 1992.

Renieris 99 M. Renieris, and S. P. Reiss, “Almost: Exploring Program Traces”, In Proceedings of the 1999 Workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM international conference on Information and knowledge management, pp. 70-77, United States, 1999.

Robson 91 D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro, “Approaches to Program Comprehension”, The Journal of Systems and Software, Elsevier North Holland, 14, pp. 79-84, 1991.

Rohatgi 08 A. Rohatgi, “An Approach towards Feature Location Based on Impact Analysis”, Master’s Thesis, Department of Computer Science, Concordia University, 2008.

Rugaber 95	S. Rugaber, “Program Comprehension”, Encyclopedia of Computer Science and Technology, 35(20), 341-368.Technical report, College of Computing, Georgia Institute of Technology, 1995.
Standish 84	T. A. Standish, “An essay on software reuse”, IEEE Trans. Software Engineering, SE-10(5), pp. 494-497, 1984.
Suri 07	P. K. Suri, and B. Bhushan, “Simulator for Time Estimation of Software Development Process”, IJCSNS, Vol 7 No. 7, 2007.
Systä 00	T. Systä, “Understanding the Behaviour of Java Programs”, In Proc. of the 7th Working Conference on Reverse Engineering, IEEE Computer Society, pp. 214-223, 2000.
Walker 98	R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Swanson, and J. Isaak, “Visualizing Dynamic Software System Information through High-level Models”, In Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, pp. 271-283, 1998.
WEKA	Weka 3: Data Mining Software in Java http://www.cs.waikato.ac.nz/ml/weka/
Whaley 00	J. Whaley, “A portable sampling-based profiler for Java virtual machines”, In Proc. of the ACM 2000 conference on Java Grande, pp. 78-87, 2000.

Wilde 03	N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds, “A comparison of methods for locating features in legacy software”, Journal of Systems and Software, 65(2), pp.105-114, 2003.
Witten 99	I. H. Witten, and E. Frank, “Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations”, Morgan Kaufmann, 1999.
Yau 80	S. S. Yau and J. S. Collofello, “Some Stability Measures for Software Maintenance”, IEEE Transactions on SofhYare Engineering, 6, pp. 545-552, 1980.