Science of Computer Programming 78 (2013) 1099–1118

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Stratified sampling of execution traces: Execution phases serving as strata

Heidar Pirzadeh [a,*], Sara Shanian [b], Abdelwahab Hamou-Lhadj [a], Luay Alawneh [a], Arya Shafiee [a]

[a] Department of Electrical and Computer Engineering, Concordia University, Montreal, QC, Canada
[b] Department of Computer Science, Laval University, Quebec City, QC, Canada

ARTICLE INFO

Article history:
Received 12 April 2011
Received in revised form 5 November 2012
Accepted 6 November 2012
Available online 3 December 2012

Keywords:
Trace analysis
Program comprehension
Sampling techniques
Stratified sampling
Execution phases

ABSTRACT

The understanding of the behavioral aspects of a software system is an important enabler for many reverse engineering activities. The behavior of software is typically represented in the form of execution traces. Traces, however, can be overwhelmingly large. To reduce their size, sampling techniques, especially the ones based on random sampling, have been extensively used. Random sampling, however, may result in samples that are not representative of the original trace. In this paper, we propose a trace sampling technique that not only reduces the size of a trace but also results in a sample that is representative of the original trace by ensuring that the desired characteristics of an execution are distributed similarly in both the sampled and the original trace. Hence, the insights gained from analyzing the sample trace could be extrapolated to the original execution trace. Our approach is based on stratified sampling instead of random sampling and uses the concept of execution phases as strata. We define an execution phase as a part of a trace that represents a specific task of the traced system. We also present an approach for the automatic detection of execution phases from a trace. Finally, we show the effectiveness of our sampling technique through two case studies.

1. Introduction

Analyzing the content of execution traces can be a challenging task due to the large size of typical traces but, if done properly, the benefits are numerous. Traces can reveal important information about the way a system behaves and help answering questions on why it behaves in a certain way. This is useful for many maintenance tasks such as understanding how a particular feature is implemented or uncovering places where a bug occurs [6,25,38,39,11]. Trace analysis techniques can also be used to correlate traces generated from subsequent versions of a system to understand important variations that can in turn help maintainers estimate the effort required to maintain evolving systems. In fact, the understanding of the behavioral aspects of software systems goes beyond maintenance to include recent research areas, like security and autonomic computing [47,16], where traces are used to characterize the normal behavior of a system and detect any deviations from normalcy due to attacks, design faults, or changes in the environment.

Investing in trace analysis techniques (or what we prefer to call software behavior analysis) is therefore an important research thread that is expected to have a significant impact. There exist today several studies that focus on ways to make traces smaller while (ideally) keeping as much of their essence as possible (e.g., [17,35]). Several trace simplification

* Corresponding author. Tel.: +1 514 750 9710.
E-mail addresses: pirzadeh@ieee.org, pirzadehc@gmail.com, s_pirzad@ece.concordia.ca (H. Pirzadeh), Sara.Shanian@ift.ulaval.ca (S. Shanian), abdelw@ece.concordia.ca (A. Hamou-Lhadj), l_alawne@ece.concordia.ca (L. Alawneh), ar_s@ece.concordia.ca (A. Shafiee).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.11.002

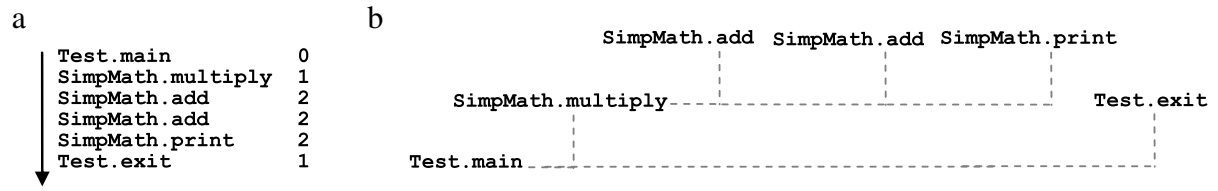

Fig. 1. (a) A trace of method calls and (b) its corresponding tree representation.

and abstraction techniques have emerged over the years to help software engineers explore the content of large traces faster. These techniques vary in their design and range from the use of visualization techniques (e.g., [36,35]) to advanced filtering techniques based on the removal of utilities [17]. These techniques should not be confused with data compression methods as defined in Information Theory. The purpose of compression algorithms is to make data as small as possible; a decompression process is required to reconstitute the data to use it for any purpose. In contrast, the purpose of trace abstraction is to make the data somewhat smaller by eliminating unneeded data, keeping the result intelligible and useful without the need for 'decompressing'. Despite the significant progress that trace analysis has seen in recent years, the general consensus is that more research in this area is needed.

In this paper, we present an approach for reducing the size of traces that is based on the sampling of the trace content. The main drawback with existing trace sampling approaches is that there is no guarantee that the resulting sample is representative of the original trace. That is, using common sampling methods, we might not be able to make correct inference about a trace based on a sample from that trace. To overcome this limitation, we propose the stratified sampling of execution traces. We first divide the trace into trace segments that we refer to as execution phases. We define an execution phase as a part of a trace that performs a specific computation [30]. In other words, a phase denotes a group of similar events[1] that together perform an essential step of the general execution. A trace can then be seen as a sequence of exhaustive and non-overlapping execution phases rather than a mere flow of events. By using execution phases as strata, we ensure that a certain number of events will be selected from each execution phase to yield a sample that is representative of the original trace. Our approach provides users with the freedom to choose their desired sample size and set a threshold for detecting phases.

The remaining part of this paper is organized as follows. In Section 2, we review some background concepts and related work. In Sections 3–6 we present our approach and its components. In Section 7, we discuss the tool support. In Section 8, we validate the effectiveness of our approach in two case studies, followed by a list of threats to validity in Section 9. We conclude the paper in Section 10.

2. Background and related work

2.1. Background concepts

Execution Trace: An execution trace is a sequence of events (e.g., method calls, invoked objects, system calls, etc.) resulted from exercising one or more features[2] of a software system. The content of an execution trace (i.e., the events) is a representation of the functionalities triggered by the user. Each event can have a number of attributes (e.g., entry time, code line number, etc). Our focus, in this paper, is on traces where the events are method calls. Such traces are commonly used in the field of software maintenance [20,43]. A trace of method calls can be represented as a tree structure. An example of interactions among two objects of the classes Test and SimpMath, which implements multiplication of numbers using repeated addition, is shown as a trace of method calls in Fig. 1(a). The integer value at the right-hand-side of each method call indicates the value of the nesting level attribute of that method call. Fig. 1(b) shows the corresponding tree representation of the trace in Fig. 1(a).

Population and Sampling: A population can be defined as a group of elements (people, plants, animals, cars, numbers, etc.) about which we want to make judgments. Studying an entire population may be slow and expensive. Sampling is a process through which we select parts of a population for analysis instead of analyzing the entire population. To be able to generalize the results of the analysis on a sample to the population, the sample has to be representative of the population. Sample representativeness means that the characteristics of the sample closely match those of the population. Thus, the goal in sampling is to find a representative sample of the population.

Execution Trace Sampling: In trace sampling, the population is the trace under study. We refer to this trace as the *original trace*. A *sampled trace* is a trace generated through the sampling of an original trace. Similar to other fields, in trace sampling, the aim is to generate a sampled trace that is representative of the original trace. Given that an original trace represents the functionalities triggered by the user, a sampled trace is representative of its original trace if the sampled trace can represent similar functionalities triggered in the original trace.

1 In this paper, we focus on traces of method calls.

2 A feature is an observable functionality triggerable by a user [13].