The Journal of Systems and Software 90 (2014) 45–60

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Combining mutation and fault localization for automated program debugging

Vidroha Debroy[a], W. Eric Wong[b,*]

[a] Microsoft, Redmond, WA, USA
[b] Department of Computer Science, University of Texas at Dallas, USA

ARTICLE INFO

Article history:
Received 5 February 2013
Received in revised form 17 October 2013
Accepted 18 October 2013
Available online 7 November 2013

Keywords:
Program debugging
Mutation
Fault localization
Fault-fixing
Software testing

ABSTRACT

This paper proposes a strategy for automatically fixing faults in a program by combining the ideas of mutation and fault localization. Statements ranked in order of their likelihood of containing faults are mutated in the same order to produce potential fixes for the faulty program. The proposed strategy is evaluated using 8 mutant operators against 19 programs each with multiple faulty versions. Our results indicate that 20.70% of the faults are fixed using selected mutant operators, suggesting that the strategy holds merit for automatically fixing faults. The impact of fault localization on efficiency of the overall fault-fixing process is investigated by experimenting with two different techniques, Tarantula and Ochiai, the latter of which has been reported to be better at fault localization than Tarantula, and also proves to be better in the context of fault-fixing using our proposed strategy. Further experiments are also presented to evaluate stopping criteria with respect to the mutant examination process and reveal that a significant fraction of the (fixable) faults can be fixed by examining a small percentage of the program code. We also report on the relative fault-fixing capabilities of mutant operators used and present discussions on future work.

1. Introduction

Much progress has been made in software development and testing over the recent years. However, in spite of this, even software that is of the highest quality may still contain faults; which in turn may cause the software to fail. Debugging a program,[1] when failure is observed, is essentially a two-step process that consists of first determining the location and nature of a suspected fault, and then fixing the fault itself (Myers, 2011). While the debugging process as a whole is extremely time-consuming and tedious, the first step in particular, referred to as 'fault localization', has been reported to be one of the most expensive of the program debugging activities (Vessey, 1985). Even in an educational setting, based on interactions with students, it has been reported that finding faults in code is an extremely difficult task (Fitzgerald et al., 2010). With this realization, several techniques have been proposed in the recent past, which aim to reduce the manual effort spent by developers/programmers to locate faults.

Broadly speaking, fault localization techniques can be classified as either static or dynamic. Static fault localization techniques essentially only make use of information that is derived from program source code or one of its abstractions, whereas dynamic fault localization techniques analyze abstractions of program execution traces collected at runtime, and then contrast successful executions with those that have failed. The general idea is for a fault localization technique to create a ranking of most probable faulty components such that these components may then be examined by programmers in order of their suspiciousness (likelihood of containing faults) until a fault is found (Abreu et al., 2009; Cleve and Zeller, 2005; Jones and Harrold, 2005; Liblit et al., 2005; Liu et al., 2006; Renieris and Reiss, 2003; Wong et al., 2010, 2012b; Wong and Qi, 2009; Zheng et al., 2006). A good technique should rank a faulty component toward the top (if not at the very top) of its ranking such that a fault is discovered early on in the examination of the ranking.

However, even if a fault has been detected and located, the burden of fixing the fault is still left solely to the programmers. This paper focuses on addressing this issue and tries to take a step in the direction of fixing faults automatically; the motivation being not just that the manual fixing of faults is time-consuming and tedious, but it is also error prone (Goel, 1985; Xie and Yang, 2003).

A common problem faced by fault localization studies is a lack of suitable data sets on which to evaluate the effectiveness of their techniques. It is often the case that even though subject programs are readily available; there are not enough faulty versions of those programs to conduct a comprehensive analysis. Recently

* Corresponding author. Tel.: +1 9728836619; fax: +1 9728832399.
E-mail addresses: vdebroy@microsoft.com (V. Debroy), ewong@utdallas.edu (W.E. Wong).

[1] In this paper the terms 'software' and 'program' are used interchangeably. Also 'fault' and 'bug' are used interchangeably.

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.10.042

researchers (not just in the area of fault localization) have relied on *mutation* (which is to be described in detail subsequently) to address this issue, where each mutant of a correct program represents a faulty version suitable for study. The rationale behind this approach is that the mutants so generated can represent 'realistic faults' and when used in experiments yield trustworthy results (Andrews et al., 2005; Do and Rothermel, 2006; Liu et al., 2006; Namin et al., 2006). However, this leads us to a very intriguing question: if mutating a correct program can result in a realistic fault, can mutating a faulty program result in a realistic fix for some faults? Let us assume that it is possible. Then, we need to answer additional questions. Are we to generate every mutant for the entire program? Where do we start and how do we know when we have fixed the fault? We provide useful guidance by combining this concept with that of fault localization.

If a fault localization technique has already ranked program components such as statements in order of how likely they are to contain faults, these statements may then be mutated in the same order until a possible fix (one of the mutants) is found. Also, under the assumptions that the fault localization technique is of sufficient quality and ranks a faulty statement as highly suspicious, and the fault is fixable using the mutant operators[2] applied; starting from the most suspicious statement we would only need to generate and execute a fraction of the total number of possible mutants, thereby reducing the overhead involved in the fault-fixing process significantly. Furthermore, this can be done automatically without the need for human intervention. Our previous study (Debroy and Wong, 2010) has provided a good starting point showing how the proposed strategy worked on a set of 8 programs using the Tarantula fault localization technique (Jones and Harrold, 2005). Based on that, we further investigate the usefulness and viability of this idea by virtue of case studies performed on multiple sets of 19 programs and two fault localization techniques. More importantly, we also discuss in depth various important aspects of the proposed idea and present useful insights on its efficiency and general applicability.

The remainder of this paper is organized as follows: Section 2 presents some background information to help readers better understand this paper followed by Section 3 which, via examples and a visual outline, describes our proposed fault-fixing strategy in detail. Section 4 reports on the case studies undertaken to evaluate the proposed strategy, while also including details on the experimental design, environment, data collection, etc. Sections 5 and 6 present in-depth discussions (and experimental data) on the importance of the fault localization technique, and the relevance of the mutant operators, respectively. Subsequently, Section 7 discusses some special aspects of the proposed fault-fixing strategy; presents a detailed discussion on how it might be extended; and discusses the threats to validity. Related work is overviewed in Section 8, and finally our conclusions appear in Section 9.

2. Preliminaries: mutation and fault localization

In order to better understand the work presented in this paper, we provide some background knowledge on the two important components of our proposed strategy. We first overview mutation, and then, move on to discuss fault localization, by demonstrating the use of the Tarantula technique (Jones and Harrold, 2005). Note that we discuss Tarantula in this section purely to illustrate how fault localization works. Our study also includes the Ochiai technique which is discussed in Section 5. Other fault localization techniques can be easily applied in the same way.

[2] The term "mutant operators" is also referred to as "mutation operators" in other published literature.

2.1. Mutation: an overview

Mutation is typically used to assess the fault detection effectiveness of a test set for a particular program, by introducing syntactic code changes into a program, and then observing if the test set is capable of detecting these changes (Budd, 1980; DeMillo et al., 1978; Do and Rothermel, 2006; Wong and Mathur, 1995a).

First, a mutant operator (say m) is applied to a program P to generate a mutant P'. If P' is different from P by exactly one change, then P' is a first-order mutant. Otherwise, P' is a higher-order mutant if there are at least two changes between P and P'. In this paper, we only consider first-order mutants. It is likely that the application of m to P shall not just lead to the generation of one such P', but rather several similar yet distinct mutants. If the program consists of more than one location where m may be applied, then m is applied one by one to each location, producing a distinct mutant each time.

Given a test set T, each mutant P' can be executed against every test case in T. If there exists a test case (say t) in T such that $P(t) \neq P'(t)$ (i.e., the output or behavior of P' on test case t is different from that of P), then P' is said to have been *killed* (or *distinguished*) by t. In other words, the fault in P' has been discovered by t, as P' fails to produce the expected output. This is consistent with the taxonomy in Avizienis et al. (2004) where a failure is defined as an event that occurs when a delivered service deviates from the correct service. A mutant that is not killed by any of the test cases in T is said to be *live* (with respect to T), i.e., the fault in this mutant is not discovered.

It is important to note that a mutant P' may be functionally equivalent to the original program P and therefore, no test case can kill it. Such mutants are called *equivalent* mutants. Thus, a live mutant may not be killed by any test case in a test set T either because it is an equivalent mutant or because the test set T is insufficient to kill the mutant. A mutation score can be assigned to a test set which is the percentage of *non-equivalent* mutants killed by test cases in this set. A test set is said to be *mutation adequate* (with respect to a set of mutant operators and a program) if its mutation score is 100%. Mutation can also be automated via tools such as Proteum for C (Maldonado et al., 2000) and Mujava (Ma et al., 2005) for Java, to name a few.

2.2. Fault localization: the Tarantula technique

For the purposes of this paper, we focus on dynamic fault localization techniques that make use of information collected as a result of test case executions against the program under test. More precisely, data on the execution result (success or failure) and the coverage (which statements[3] are covered/executed by each test case and which are not) is utilized to assign a suspiciousness value to each statement, which is interpreted as the likelihood of that statement being faulty. Once the suspiciousness value for each statement is computed, all the statements can then be sorted in descending order from most suspicious to least suspicious to produce a ranked list that can be examined in order by programmers for locating faults.

Tarantula is based on the intuition that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are executed by successful (or passed) test cases (Jones and Harrold, 2005). Tarantula also allows some tolerance for faults to be occasionally executed by successful test cases and they find that this tolerance often provides for more effective

[3] In this paper we consider 'statements' as the program components of interest with the understanding that without loss of generality, fault localization techniques, and the proposed fault-fixing strategy are equally applicable when considering other program components such as functions, blocks (Agrawal et al., 1995), predicates, etc.