Applied Soft Computing 11 (2011) 3494–3514

Contents lists available at ScienceDirect

Applied Soft Computing

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

Evolutionary repair of faulty software

Andrea Arcuri

Simula Research Laboratory, P.O. Box 134, Lysaker, Norway

ARTICLE INFO

Article history:
Received 30 March 2009
Received in revised form 2 August 2010
Accepted 16 January 2011
Available online 28 January 2011

Keywords:
Repair
Fault localization
Automated debugging
Genetic programming
Search Based Software Engineering
Coevolution

ABSTRACT

Testing and fault localization are very expensive software engineering tasks that have been tried to be automated. Although many successful techniques have been designed, the actual change of the code for fixing the discovered faults is still a human-only task. Even in the ideal case in which automated tools could tell us exactly where the location of a fault is, it is not always trivial how to fix the code. In this paper we analyse the possibility of automating the complex task of fixing faults. We propose to model this task as a search problem, and hence to use for example evolutionary algorithms to solve it. We then discuss the potential of this approach and how its current limitations can be addressed in the future. This task is extremely challenging and mainly unexplored in the literature. Hence, this paper only covers an initial investigation and gives directions for future work. A research prototype called JAFF and a case study are presented to give first validation of this approach.

1. Introduction

Software testing is used to reveal the presence of faults in computer programs [1]. Even if no fault is found, testing cannot guarantee that the software is fault-free. However, testing can be used to increase our confidence in the software reliability. Unfortunately, testing is expensive, time consuming and tedious. It is estimated that testing requires around 50% of the total cost of software development [2]. This is the reason why there has been a lot of effort spent to automate this expensive software engineering task.

Even if an optimal automated system for software testing existed, to be able to fix the faults we would still need to know *where* they are located. Automated techniques can help the tester in this task [3–5].

Although in some cases it is possible to automatically locate the faults, there is still the need to modify the code to remove the faults. *Is it possible to automate the task of fixing faults?* This would be the natural next step if we seek for full automated software engineering. And it would be particularly helpful in the cases of complex software in which, although the faulty part of code can be identified, it is difficult to provide a patch for the fault. This would also be a step forward to achieve corporate visions as for example IBM's *Autonomic Computing* [6].

There has been work on fixing code automatically (e.g. [7–10]). Unfortunately, in those works there are heavy constraints on the type of modifications that can be automatically done on the source code. Hence, only limited classes of faults can be addressed. The reason for putting these constraints is that there are infinite ways to do modifications on a program, and checking all of them is impossible.

In this paper we investigate whether it is possible to automatically fix faults in source code without putting any particular restriction on their type. Because the *space* of possible modifications cannot be exhaustively evaluated, we model this task as a *search problem* [11,12].

The reformulation of software engineering as a search problem (i.e., *Search Based Software Engineering*) has been widely studied in the recent years. Many software engineering tasks have been modelled in this way with successful results (e.g., testing [13]). In many software engineering cases, search algorithms seem to have better performance than more traditional techniques (e.g. [14,15]). This was our motivation for applying search algorithms on the task of automatically repairing faulty software.

Given as input a faulty program and a set of test cases that reveal the presence of a fault, we want to modify the program to make it able to pass all the given test cases. To decide which modifications to do, we use a search algorithm. Note that we want to correct the source code, and not the state of the computation when it becomes corrupted (as for example in [16]).

The search space of all possible programs is infinite. However, "programmers do not create programs at random" [17]. Therefore, it is reasonable to assume that in most cases the sequences of modifications to repair software would not be very long. This assumption makes the task less difficult.

We discussed the idea of fixing software with search algorithms in a doctoral symposium paper [18], and we then presented very preliminary results on a sorting routine using a limited Lisp-like

E-mail address: arcuri@simula.no

1568-4946/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.asoc.2011.01.023

programming language [19]. In this paper, we present a novel prototype that is able to handle a large sub-set of the Java programming language. The case study is based on realistic Java software.

Different types of search algorithms could be used. In this initial investigation, we consider and compare three search algorithms. We use a random search as baseline. Then we consider a single individual algorithm (i.e., a variant of a Hill Climbing (HC)) and a population based algorithm (i.e., Genetic Programming (GP) [20]).

To improve the performance of these algorithms, we present a novel search operator that is based on current fault localization techniques. This operator is able to narrow down the search effort to promising sub-areas of the search space. Besides providing an empirical validation, we also theoretically analysed the conditions for which this operator is helpful.

The main contributions of this paper are:

- We analyse in detail the task of repairing faulty software in an automatic way. In particular, we characterise the search space of software repair and we explain for which types of faults our novel approach can scale up.
- To improve the performance of software repair with search algorithms, we present a novel search operator. This operator is not limited to the software repair problem. It can be extended to other applications in which programs are evolved. Theoretical and empirical analyses confirm that this novel operator eases the task of repairing software.
- We present a Java prototype called JAFF (Java Automatic Fault Fixer) for validating our automatic approach for repairing faulty software. Differently from the work in the literature (e.g. [7,21,22]), no particular constraint is imposed on the type of modifications that JAFF can apply to fix faulty software.
- We carried out a large empirical study that confirms that faulty software can be repaired automatically. It was possible to automatically fix types of faults that current techniques in the literature are not able to handle (e.g. [7,21,22]). Furthermore, our analyses show that for this task GP is better than HC and random search.

The paper is organised as follows. Section 2 gives a brief overview of the automation of the debugging activity. Section 3 describes how software repair can be modelled as a search problem. The analysed search algorithms are described in Section 4. The novel search operator is presented in Section 5. Our research prototype JAFF is presented in Section 6. The case study on which the proposed framework is evaluated follows in Section 7. Section 8 outlines the limitations of repairing software automatically. Future directions of research are discussed in Section 9. Finally, Section 10 concludes the paper.

2. Related work

Debugging consists of two separated phases. First, we need to *locate* the parts of the code that are responsible for the faults. Then, we need to *repair* the software to make it correct. This means that we need to modify the code to fix it. These changes to the code are often called *patch*.

Several different techniques have been proposed to help software developers to debug faulty software. We briefly discuss them. For more details about the early techniques, old surveys were published in 1993 [3] and 1998 [4]. A more updated and comprehensive analysis of the debugging problem can be found in Zeller's book [5] and partially in Jones's thesis [23].

2.1. Fault localization

One of the first techniques to help to locate faults is *Algorithmic debugging* [24,25]. Using a divide-and-conquer strategy, the computation tree is analysed to find which sub-computation is incorrect. This approach has two main limitations. First, an oracle for each sub-computation is required. This is often too expensive to provide. Second, the precision of the technique is too coarse-grained.

A *slice* [26] is a set of code statements that can affect the value of a particular variable at a certain time during the execution of software. Debugging techniques can exploit these slices to focus on only the parts of the code that can be responsible for the modification of suspicious variables [27–29].

In *delta debugging* [30–33] the trace of a passing execution is compared against a similar (from the point of view of the execution flow) one that is instead failed. A binary search is done on the memory states of these two executions to narrow down the inspection of suspicious code. The memory states of the failing execution are altered to see whether these alterations remove the failure. This technique is computationally very expensive. Finding two test cases with nearly identical execution path, but one passing and the other failing, can be difficult. If all the provided test cases fail, then this technique cannot be applied.

Software developers often make common mistakes that are practically independent from the semantics of the software. Typical example is opening a stream and then not closing it. Another is sub-classing a method with a new one that has very similar name (doing this has the wrong result of having a new method instead of sub-classing the previous one). Many of these mistakes can be found by statically analysing the source code without running any test case. A set of *bug patterns* can be defined and used to see whether a program has any of this known mistakes [34–36]. On one hand, this technique has the limitation that it can find only faults for which a pattern can be defined. On the other hand, it is a very cheap technique that does not require any test case. It can be easily applied on large real-world software and it can point out many possible sources of faults. This type of static analysis can be improved with data mining techniques applied to real-world source code repositories [37].

In large real-world software, it is common that parts of code result from *copy-and-paste* activities. This has been shown to be very prone to introduce faults, because for example often the developers forget to modify identifiers. If the software does not give any compiling error, then it is very difficult to find this type of fault. Data mining techniques to identify copy-and-paste faults have been proposed [38].

If the behavioural model of software is available (expressed for example with a finite state machine), one black-box approach is to identify which components of the model are wrongly implemented in the code [39]. Similar to mutation testing [17], the idea is to mutate the model with operators that mimic common types of mistakes done by software developers. *Confirming sequences* are then generated from the mutated models and validated against the tested program [39]. The mutated models represent hypothesis about the nature of the faults.

To understand the reason why a fault appears, software developers speculate about the possible reasons. This translates to questions about the code. Tools such as *Whyline* [40] automatically present to the user questions about properties of the output, and then they try to give explanations/answers based on the code and the program execution.

Given a set of test cases, coverage criteria can be used as heuristic to locate faults [41]. On the one hand, parts of code that are executed only by passed test cases cannot be responsible for the faults. On the other hand, code that is executed only by the failing test cases is