Using Execution Paths to Evolve Software Patches

ThanhVu Nguyen*, Westley Weimer†, Claire Le Goues† and Stephanie Forrest*
*Department of Computer Science, University of New Mexico, {tnguyen,forrest}@cs.unm.edu
†Department of Computer Science, University of Virginia, {weimer,legoues}@virginia.edu

Abstract

We present an evolutionary approach using Genetic Programming (GP) to automatically create software repairs. By concentrating the modifications on regions related to where the bug occurs, we effectively minimize the search space complexity and hence increase the performance of the GP process. To preserve the core functionalities of the program, we evolve programs only from code in the original program. Early experimental results show our GP approach is able to fix various program defects in reasonable time.

1. Motivation

Detecting and fixing software bugs remains a major burden for the programmers even after the project is released. Despite promising results and efforts in developing automated debugging techniques, these methods still rely on results from rigorous automated testings to locate the errors and require manual modifications from the programmers to fix bugs [1].

In this paper, we propose a Genetic Programming (GP) approach to automate the task of repairing program bugs in existing software. Programs are evolved and evaluated until one is found that retains the functionality of the original program and fixes the bug that occurred. We first process the source code of the program to produce a path containing traces of execution procedures. This allows us to obtain a *negative* execution path when an error occurs which contains the list of executed statements. Next, the GP algorithm creates new programs by modifying the original code with more bias toward statements that occurred during the negative execution path. Additional tests are incorporated into the fitness function to retain the functionalities of the program.

A major impediment for evolutionary algorithms like GP is the potential exponential-size search space that the program must explore. To address this, we constrain our GP to only operate on the regions of the program relevant to the error rather than the entire program; that is, we concentrate on statements executed in the negative execution path. Moreover, we restrict the algorithm to produce changes that are based on other parts of the original program. We hypothesize that most errors in the original program can be replaced using statements found in other locations in the program. Combining these ideas with other optimization techniques, such as caching programs and minimizing the results, we have created an automated software patcher that creates repairs for more than ten real C programs in feasible time.

This extended abstract first reviews the main ideas and results reported in [2], [3]. Additional details and experimental results are then described, especially concerning the GP implementation technique.

2. The GP Algorithm

2.1. Preprocessing

To obtain the execution path, we use the C Intermediate Toolkit [4] to assign unique ID's to statements in a C program source file. An *execution path* is a record of statement ID's that were called when the program runs against some inputs or *testcases*. We define a *positive* testcase as one that results in expected behavior and a *negative* testcase as one that causes an error. One of the key ideas in our approach is to focus on the regions where the error occurs. To do this we assign a weight w to each statement in the program, with heavier weight on statements occuring only in the negative execution path. To preserve the contents of the original program, we hash its statements into a code repository and evolve new programs from there.

2.2. Genetic Programming

Our GP approach follows the traditional GP algorithm structure. The algorithm maintains a population of chromosomes (programs), selects a pool of individuals based on their fitness, and modifies them with mutation and crossover operators. The program stops upon reaching a terminating criterion.

The **fitness function** takes a program source code, compiles it, and runs against the set of positive and negative testcases. Finally it returns a score indicating the acceptability of that program. The negative test reproduces the bug in the original program that needs to be fixed and the positive testcases preserve the core functionalities of the program. The fitness score of a program is the weighted sum of the testcases that the program passes. We assign the fitness score of *zero* to programs that do not compile and

those with runtime exceeding a preset time threshold (e.g., five seconds).

A subset of the population is **selected** for reproduction using either stochastic universal sampling or tournament selection. Those with fitness score *zero* are immediately excluded. From here we have a mating pool ready to be modified by the crossover and mutation genetic operations.

Our first **crossover** preserves the contents of the original program and concentrates on regions in the negative execution path. A single cutoff point c is chosen randomly for both input parents. Then all statements with ID's larger than c are selected for crossover based on their weight values w. The contents of the selected statement s from both parents are replaced by the contents of statement s in the code repository.

Our second implementation adheres to the conventional 1-point crossover in GP by exchanging a statement from one parent with another. Our representation of the program allows a statement to contain sub-statements, e.g., conditional and loop code contains all statements within that code blode. These statements are chosen uniformly at random regardless of their weights w.

We consider each statement in the negative execution path for **mutation** with more bias toward those that are unique (i.e., only happen in the negative execution path and not on the positive one). The selected statement s is modified with one of the three operations: *delete* the contents of s, *replace* the contents of s with another one from code repository, or *insert* a statement from the code repository after s.

The GP **terminates** when an acceptable solution (i.e., one passing all the testcases) is found or it has exceeded the maximum number of preset generations.

2.3. Bloat Control and Other Optimizations

Our GP approach deals with code bloat in several ways. The algorithm evolves programs very similar to the original by limiting the modifications to regions in the negative execution path and only uses code from the original program. Moreover, the selection routine disregards non-working programs. Hence, programs that are not well-formed or deviate greatly from the original have a low chance of being selected. Finally our GP process stops when a candidate passes all the testcases, it doesn't keep evolving to find better solutions.

To improve the performance of our algorithm, we cache the program (its *md5sum* result) and the associated fitness score. Only programs not in the cache are evaluated by the fitness function.

In addition, we apply ideas from structural differencing algorithms [5] and delta debugging [6] to minimize the repair found. Our technique generates a 1-minimal patch that, when applied to the original program, repairs the defect without sacrificing required functionality.

3. Experimental Results

Our GP has been shown to fix several real-world defects. *Zune-bug* is our implementation of the 366^{th} day bug which occurs in Microsoft Zune music players. Our repair adds in a new exit condition breaking out of the loop when the variable days is the last day of a leap year. The *look* function in svr4.0 1.1 has an infinite binary search when the dictionary file is not sorted. Our program adds a new exit condition to this loop. For the segfault caused by the *look* dictionary function in ultrix 4.3, our fix changes the handling of the command-line arguments, avoiding the cases of buffer overrun in the function getword. *flex*, a lexical analyzer generator, has a bug causing segfault in version 2.5.4a when the yytext variable points to an unterminated user input. Our repair changes one of the uncontrolled input fragments held by yytext. In *atris*, a graphical Tetris game, a local stack buffer exploit happens due to an incorrect use of sprintf to construct user-defined variables. Our program removes the sprintf call, leaving all users with the default global preferences.

Overall, our algorithm has successfully fixed defects in more than ten programs, including security vulnerabilities in *lighthttpd* (a webserver) and *wu-ftpd* (an ftp server). The success of finding a patch ranges from 4% to 99% with average running time ranging from half a second to ten minutes.

References

[1] A. Arcuri, "On the automation of fixing software bugs," in *Proceedings of the Doctoral Symposium of the IEEE International Conference on Software Engineering*, 2008.

[2] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering (in press)*, 2009.

[3] S. Forrest, W. Weimer, T. Nguyen, and C. L. Goues, "A genetic programming approach to automated software repair," in *The Genetic and Evolutionary Computation Conference (in press)*, 2009.

[4] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: An infrastructure for C program analysis and transformation," in *International Conference on Compiler Construction*, Apr. 2002, pp. 213–228.

[5] R. Al-Ekram, A. Adma, and O. Baysal, "diffX: an algorithm to detect changes in multi-version XML documents," in *Conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2005, pp. 1–11.

[6] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Foundations of Software Engineering*, 1999, pp. 253–267.