Fixing Real Bugs in Real Programs using Evolutionary Computing

ThanhVu Nguyen*
University of New Mexico
tnguyen@cs.unm.edu

Westley Weimer
University of Virginia
weimer@virginia.edu

Claire Le Goues
University of Virginia
legoues@virginia.edu

Stephanie Forrest
University of New Mexico
forrest@cs.unm.edu

Abstract

We present an automatic method to create software repairs using Evolutionary Computing (EC) techniques. 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 EC process. To preserve the core functionalities of the program, we evolve programs only from code in the original program. Positive and negative testcases are used in the fitness function to determine the correctness of the programs. In addition, various techniques are considered to speed up the process and delta debugging with structural differencing algorithms are applied to optimize the repair found. Early experimental results show our EC approach is able to fix various program defects in reasonable time.

Keywords: evolutionary computing, genetic programming, software repair, software engineering.

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 [2].

In this paper, we propose an Evolutionary Computing (EC) [4] based 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, our EC 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 EC algorithms is the potential exponential-size search space that the program must explore. To address this, we constrain our algorithm 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[1]. 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.

2 Software Repair using EC (SREC)

The below program fragment (Figure 1) shows the embarrassing bug causing Microsoft Zune media players to freeze on December 31st, 2008 [3]. Throughout this section we refer to this example to describe how our algorithm works.

2.1 Preprocessing

To obtain the execution path, we use the C Intermediate Toolkit [8] 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, e.g., `zunebug(365)`, `zunebug(1000)` and a *negative* testcase as one that causes an error, e.g, `zunebug(366)`, `zunebug(10593)`. 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 statements occurring in the negative execution path (e.g., lines $1-9$ using the negative testcase `zunebug(366)`). In addition, we also prefer lines that are unique,i.e., only happen in

[1] though these fixes might not be first-choice corrections for the programmer.

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365)  {
4      if (isLeapYear(year)){
5        if (days > 366)  {
6          days -= 366;
7          year += 1;
8        }
9      }
10     else {
11       days -= 365;
12       year += 1;
13     }
14   }
15   printf("current year is %d\n", year);
16 }
```

Figure 1. Zunebug: infinite loop on the last day of leap year (i.e., when the *days* variable has value 366)

Lines	Statement
4	`int isLeapYear(int){ ... }`
6	`days -= 366 ;`
7	`year += 1 ;`
5-8	`if (days > 366){...}`
11	`days -= 365 ;`
12	`year += 1 ;`
4-13	`if isLeapYear(year){...} else{...}`
3-14	`while(days > 365){...}`
2	`int year = 1980;`
15	`printf("current year is %d\n", year);`

Figure 2. Code-Bank for zunebug

the negative execution path and not on the positive one. To preserve the contents of the original program, we hash its statements into a code repository called C-Bank (Figure 2) and evolve new programs from there.

2.2 Evolutionary Computing

Our Software Repair using Evolutionary Computing (SREC) algorithm (Figure 3) follows the traditional Evolutionary 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 recombination operators. The program stops upon reaching a terminating criterion.

The **fitness function** (Figure 4) 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

Input: Program V to be repaired
Input: Set of positive T_P and negative T_n testcases
Output: Repaired version V' if found

1: $W_{\text{path}} := \text{preprocess}(V, P_n,)$
2: $Pop := \text{initial\_population}(V, \text{pop\_size})$
3: $gen := 0$
4: **repeat**
5: $F := \text{eval\_fitness}(Pop, T_p, T_n)$
6: **if** $\exists\, V' \in Pop\ .\ V'_f = \text{max\_fitness}$ **then**
7: **return** V'
8: **end if**
9: $Pop := \text{select}(Pop, F)$
10: **for all** $\langle p_1, p_2 \rangle \in Pop$ **do**
11: $\langle c_1, c_2 \rangle := \text{crossover}(p_1, p_2)$
12: $Pop := Pop \cup \{c_1, c_2\}$
13: **end for**
14: **for all** $P \in Pop$ **do**
15: $P := \text{mutate}(P)$
16: **end for**
17: $gen = gen + 1$
18: **until** $gen > max\_gen$
19: **return** $\text{minimize}(V, P, PosT, NegT)$

Figure 3. SREC algorithm.

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 scores equal to or less than *zero* are immediately excluded. From here we have a mating pool ready to be modified by the recombination and mutation genetic operations.

Our first **recombination** (Figure 5) method adheres to the conventional 1-point *crossover* in EC strategies 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 block. These statements are chosen uniformly at random regardless of their weights w.

Our second implementation called *crossback* 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 recombination 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.

We consider each statement in the negative execution

Input: Program V.
Input: Set of positive T_p and negative T_n testcases.
Output: V_f, fitness score of V

```
1:  V_f := 0
2:  if V is in fitness_cache then
3:     V_f := retrieve_from_fitness_cache(V)
4:  else
5:     V_exec := compile(V)
6:     if V_exec is valid then
7:        pass_p := exec(V_exec, T_p)
8:        pass_n := exec(V_exec, T_n)
9:        V_f := w_pos * |pass_p| + w_neg * |pass_n|
10:    end if
11:    cache(V, V_f)
12: end if
13: return V_f
```

Figure 4. Fitness evaluation. w_{pos} is set by default at 1.0 and w_{neg} is set by default at 10.0 (higher fitness is given to programs passing the negative testcases)

path for **mutation** (Figure 6) with more bias toward those with heavier weights. 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 bank, or *insert* a statement from the code bank after s.

The algorithm **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

Unlike traditional code evolving methods such as Genetic Programming, our EC approach does not suffer from code *bloats* due to several reasons. The algorithm does not add additional nodes (or branches) to existing structure. Inserting a statement j to statement i appends j to i, i.e., $i = \{i; j\}$, but does not separate i and j into two distinct nodes. Moreover, SREC 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. The selection routine also 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 EC 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.

Input: Two programs V_1 and V_2.
Output: Two new programs C_1 and C_2.

```
1:  C_1 := copy(V_1)
2:  C_2 := copy(V_2)
3:  if method = crossover then
4:     stmt_i := V_1[rand]
5:     stmt_j := V_2[rand]
6:     swap(stmt_i, stmt_j)
7:  else if method = crossback then
8:     cutoff := rand(|W_path|)
9:     for all stmt_i ∈ |W_path| do
10:       if i > cutoff then
11:          if random(stmt_i^w) then
12:             V_1[i] := stmt_i
13:             V_2[i] := stmt_i
14:          end if
15:       end if
16:    end for
17: end if
18: return C_1, C_2
```

Figure 5. Recombination operator. $|W_{path}|$ is the length of the weighted path. stmt_i^w is the weight assigned to statement i.

In addition, we apply ideas from structural differencing [1] algorithms and delta debugging [10] 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 SREC Performance and Scalability

SREC has been shown to fix several real-world defects. Our *zunebug* repair (Figure 3) executes the statement `days -= 366;` when *year* is a leap year, thus guarantees termination to the loop. 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.

Table 8 provides the information on programs that are successfully repaired with SREC. For each program, we run

Program	Version	Types of Bug	Stmts/ LoC	Pos/Neg Testcases	Negative Path Weight	Success Rate		
						crossover	crossback	size
`zune`	example	Infinite loop	14 / 28	5/2	1.1	58%	71%	4
`uniq`	ultrix 4.3	Segfault	81 /1146	5/1	81.5	100%	100%	4
`look-u`	ultrix 4.3	Segfault	90 /1169	5/1	213.0	100%	99%	11
`look-s`	svr4.0 1.1	Infinite loop	100 /1363	5/1	32.4	100%	100%	3
`units`	svr4.0 1.1	Segfault	240 /1504	5/1	2159.7	5%	7%	4
`deroff`	ultrix 4.3	Segfault	1604 /2236	5/1	251.4	97%	97%	3
`indent`	1.9.1	Infinite loop	2022 /9906	5/1	1435.9	34%	7%	2
`flex`	2.5.4a	Segfault	3635 /18775	5/1	3836.6	4%	5%	3
`atris`	1.0.6	Buffer exploit	6470 /21553	2/1	34.0	82%	82%	3

Figure 8. Experimental Results

Input: Program V.
Output: V_m, a mutated version of V

1: $V_m := copy(V)$
2: **for all** $\mathsf{stmt}_i \in W_{path}$ **do**
3: **if** $\mathsf{random}(\mathsf{stmt}_i^w) \wedge \mathsf{random}(mut\_rate)$ **then**
4: $\mathsf{op} := \mathsf{choose}(\{\mathsf{insert}, \mathsf{replace}, \mathsf{delete}\})$
5: **if** $\mathsf{op} = \mathsf{replace}$ **then**
6: $\mathsf{stmt}_j := \text{C-Bank}[\mathsf{rand}]$
7: $\mathsf{stmt}_i := \mathsf{stmt}_j$
8: **else if** $op = \mathsf{insert}$ **then**
9: $\mathsf{stmt}_j := \text{C-Bank}[\mathsf{rand}]$
10: $\mathsf{stmt}_i := \{\mathsf{stmt}_i; \mathsf{stmt}_j\}$
11: **else if** $op = \mathsf{delete}$ **then**
12: $\mathsf{stmt}_i := \{\}$
13: **end if**
14: **end if**
15: **end for**
16: **return** V_m

Figure 6. Mutation operator. The global variable mut_rate is set by default at 0.06.

```
1   void zunebug_repair(int days) {
2     int year = 1980;
3     while (days > 365) {
4       if (isLeapYear(year)){
5         if (days > 366) {
6           // days -= 366; // repair deletes
7           year += 1;
8         }
9         days -= 366;       // repair inserts
10      } else {
11        days -= 365;
12        year += 1;
13      }
14    }
15    printf("current year is %d\n", year);
16  }
```

Figure 7. Final Zunebug repair after minimization

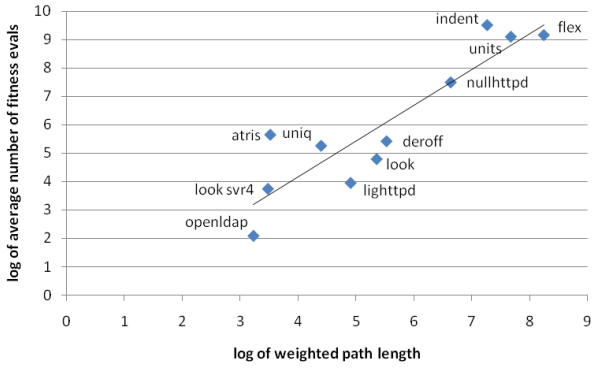

Figure 9. EC Search Time Scales with Execution Path Weight. Data are shown for 11 programs successfully repaired by GP. The x-axis is the natural logarithm of the weighted path length, and the y-axis shows the natural logarithm of the total number of fitness evaluations performed before the primary repair is found (averaged over 100 runs).

the algorithm for 100 trials using two mutation parameters 0.06 and 0.03. If the first parameter does not work (i.e., gives no repair), the second parameter is used. Column "Stmt/LoC" gives the approximate size of the test programs in terms of number of statements and lines of code. The "Pos/Neg Testcases" column lists the number of positive and negatve testcases used. The "Negative Path Weight" column gives the weighted length of the negative execution path. The last three columns provide statistics when running SREC on these test programs. Columns "crossover" and "crossback" respectively show the success rates of SREC when using different recombination methods. Finally, the "size" column shows the difference in lines of code between the original program and the repair found.

In order to assess the practicality of our EC approach,

we need to know how the algorithm scales with problem size and the expected size of the problems we would want to solve. Figure 9 plots weighted path length against search time (measured as the average number of fitness evaluations until the first repair). On a *log-log* scale, the relationship is roughly linear with slope 1.26 (90% confidence: [0.90, 1.63]), including the two outliers (**atris** and **uniq**). Although we do not have enough data to draw strong conclusions, the plot suggests that search time may scale as a power law of the form $y = ax^b$ where b is the slope of the best fit line (1.26) and $b = 1$ would indicate that search time grows linearly. This is encouraging because it suggests that search time grows as a small polynomial of the weighted execution path and not as an exponential.

Overall, our algorithm has successfully fixed defects in more than ten programs, including security vulnerabilities in *lighthttpd*, *nullhttpd* (opensource webservers), *openldap* (opensource directory server), and *wu-ftpd* (an ftp server). The success of finding a patch ranges from 4% to 100% with average running time ranging from half a second to ten minutes. Due to space limitations, we save the details and statistics of these results for other publications [5, 6, 9, 7].

3.1 Limitations and Future Work

Our technique builds programs only from existing code and lacks of the ability to introduce new code. One remedy for this is to add more commonly used statements C-Bank, the code repository. The code can be manipulated directly by changing operators such as $+, *, /, <, >, \&\&, |||$. We also rely on unique statements occured on the negative paths to reduce search space; thus may encounter difficulty if the statements on negative execution paths overlap greatly with those on positive execution paths.

In addition, our future plans include trying differrent genetic operators (e.g., different types of crossover, mutation) and selection techniques (e.g., tournament selection). Parallel models are also promising to our work. For instance we can take advantage of Shared-Memory systems (e.g., SMP, Multicore) to parallelize the fitness evalulation to speed up the process. Distributed Memory systems can also be used to raise different populations in parallel and have them communicate periodically. These ideas will be investigated more thoroughly in future studies.

4 Conclusions

We introduced a Evolutionary Computing based algorithm that automatically creates fixes for software defects. Our approach is based on several novel ideas including constraining the search space to regions relevant to the defects and reusing the original program as a code repository for the repairs. To guide the EC process, our fitness function combines standard testcases with negative testcase for the respective purposes of retaining the required functionalities of the original program and avoiding the defect. Early experiments show the algorithm works on a variety of programs in feasible time.

References

[1] 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*, pages 1–11. IBM Press, 2005.

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

[3] BBC News. Microsoft zune affected by 'bug'. In *http://news.bbc.co.uk/2/hi/technology/7806683.stm*, Dec. 2008.

[4] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[5] 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.

[6] C. L. Goues, T. Nguyen, W. Weimer, and S. Forrest. Closed-loop repair of security vulnerabilities. In *USENIX Security (submitted)*, 2009.

[7] C. L. Goues, T. Nguyen, W. Weimer, and S. Forrest. Using execution paths to evolve software patches," search-based software testing. In *Search-Based Software Testing (in press)*, 2009.

[8] 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*, pages 213–228, Apr. 2002.

[9] 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.

[10] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.