Open access • Proceedings Article • DOI:10.1145/1858996.1859059

Automated program repair through the evolution of assembly code — Source link

Eric Schulte, Stephanie Forrest, Westley Weimer

Institutions: University of New Mexico, University of Virginia

Published on: 20 Sep 2010 - Automated Software Engineering

Topics: Source code, Dead code, Static program analysis, Unreachable code and Redundant code

Automated Program Repair through the Evolution of Assembly Code

Eric Schulte
Computer Science
University of New Mexico
Albuquerque, NM
eschulte@cs.unm.edu

Stephanie Forrest
Computer Science
University of New Mexico
Albuquerque, NM
forrest@cs.unm.edu

Westley Weimer
Computer Science
University of Virginia
Charlottesville, VA
weimer@virginia.edu

ABSTRACT

A method is described for automatically repairing legacy software at the assembly code level using evolutionary computation. The technique is demonstrated on Java byte code and x86 assembly programs, showing how to find program variations that correct defects while retaining desired behavior. Test cases are used to demonstrate the defect and define required functionality. The paper explores advantages of assembly-level repair over earlier work at the source code level—the ability to repair programs written in many different languages; and the ability to repair bugs that were previously intractable. The paper reports experimental results showing reasonable performance of assembly language repair even on non-trivial programs.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.2 [**Software Engineering**]: Macro and Assembly Languages

General Terms

Experimentation, Languages

Keywords

program repair, evolutionary computation, fault localization, assembly code, bytecode, legacy software

1. INTRODUCTION

Automating the process of software repair is an important issue for software engineering, one that is under active exploration by several groups (e.g., [11, 2, 10, 11]). This paper describes a generic method for automatically repairing important classes of software defects in legacy assembly programs, compiled from different programming languages in the absence of formal specifications.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

In earlier work, Evolutionary Computation (EC) techniques successfully repaired bugs in legacy C programs by representing and manipulating the abstract syntax tree (AST) of C source code at the statement level [3, 11]. The earlier work does not presuppose a formal specification of the program, relying instead on test cases, both for fault localization and as a proxy for evaluating correctness. The current paper extends this earlier work, demonstrating the feasibility of repairing programs by manipulating the assembly code associated with existing software products. Although direct manipulation of assembly code is not the most obvious way to approach software repair for programs written in high-level languages, the paper demonstrates a number of benefits:

- **Generality:** The technique is potentially applicable to any program that compiles to an intermediate assembly code, even without access to source code.
- **Expressive Power:** The earlier work was limited to statement-level repairs and could not directly repair, for example, transposed arguments to a function, incorrect type declarations for a variable, or incorrect value assignments to a constant. All of these repairs are possible at the finer-grained assembly code level.
- **Representation:** The assembly code representation features a small alphabet of primitives (each instruction is a primitive in our representation) and many more lines of code than their source code counterparts. These properties ensure that our method (which does not invent new code) has access to a large portion of the space of possible instructions.

2. BACKGROUND

Evolutionary Computation (EC) is a domain-independent method of guided random search which borrows both mechanisms and terminology from the biological process of evolution. Both *Genetic Algorithms* (GA) and *Genetic Programming* (GP) [5] are sub-fields of EC in which the candidate solutions are represented either as linear arrays of parameters and features (GA) or as program trees (GP). In this work we blur the distinction between GA and GP by evolving linear arrays of assembly code instructions, and we therefore adopt the more general term of evolutionary computation.

In EC, a population of candidate solutions is generated for a given problem. In our application, these candidates are computer programs. Each candidate solution is referred to as an *individual*, *genome*, or *variant*, and we use those

terms interchangeably. EC evaluates each individual to assess how well it solves the problem at hand, referred to as its *fitness*. In this paper, fitness is assessed by test cases. Only high-fitness programs (i.e., those that pass many test cases) are retained in the population. Computational analogs of biological mutation and crossover produce variations of the high-fitness programs, and the process iterates until a high-fitness program is found.

The small amount of previous work on EC program repairs falls into two categories. The first guarantees that the code-modification operators (*genetic operators*) can produce only valid programs [8, 9], while the second uses general operators and defers the determination of validity to the compiler and execution engine [6]. Using the first approach, Orlov and Sipper evolved machine code compiled from short programs written in Java (on the order of 100 lines of code for simple mathematical manipulations) [9]. Here we demonstrate applicability to legacy programs that are both longer and more complex using the latter approach, finding that assembly code is surprisingly robust under naïve modifications. In earlier work, we demonstrated that EC could repair legacy C programs by operating on statement-level ASTs parsed from C source code [3, 11]. The earlier work was limited to programs written in C and to bugs that can be fixed with only statement-level manipulations. This paper addresses both limitations by operating at the assembly code level.

Clearview [10] addresses a problem similar to ours, automatically detecting and patching assembly-level errors in deployed software. Clearview monitors specific features of a program at runtime, learns invariants that characterize normal behavior, and subsequently flags violations for repair. Candidate patches that satisfy the implicated invariant are generated and tested dynamically. Clearview repairs only those errors that are relevant to preselected monitors. Our method is more generic, providing a single approach to repair multiple classes of faults without the need for specific monitors, and we do not require continual runtime monitoring (and the incumbent slowdown) to create and deploy repairs.

3. TECHNICAL APPROACH

Our technique takes as inputs an assembly-language program and its test suite. The test suite contains both *positive* test cases, which the program currently passes, and one or more *negative* test cases, which the program currently fails, demonstrating the bug. The program is represented abstractly (Section 3.1) and manipulated using EC operators (Section 3.2) to generate new candidate repairs. This process iterates until a repair is found that passes all tests (Section 3.3).

3.1 Representation

Individuals (program variants) are represented as linear arrays of assembly code instructions, either x86 assembly code or Java bytecode. In the case of x86 assembly, each element in the array represents an entire line taken from a file of assembly code, such as that generated by `gcc -S`. For Java the "Bytecode Engineering Library" (BCEL) [1] is used to obtain a list of bytecode instructions from a compiled `class` file.

Assembly code instructions include labels, single commands, and commands with arguments. We never alter the internal elements of an assembly code instruction, and no distinction is made between instructions based on their form or content. This simplifies and generalizes the operators (because there are no hard-coded rules about x86 assembly structure, for example). It also restricts the search to programs that can be constructed from the program's existing assembly code instructions Given the relatively small alphabet of assembly code instructions and the large number of such instructions in even small programs, this limitation seems reasonable.

3.2 Genetic Operators

Our genetic operators permute, copy, delete and recombine existing sequences of instructions. Each instruction is assigned a positive and negative weight indicating it's relevance to the execution path of the positive and negative test cases respectively. There are three *mutation operators* for altering a single variant and one *crossover operator* for recombining two variants into a single offspring. *Mutate-insert* selects an instruction with probability proportional to its positive weight, copies and inserts it in a position selected proportionally to the negative weights. *Mutate-delete* selects an instruction proportionally by negative weight and deletes it. The *mutate-swap* operator selects two instructions proportionally by negative weight and swaps them. Finally, *Crossover* begins with two program variants, selects a single crossover point for each of them (proportionally by negative weight) and exchanges all instructions after the crossover point between the two variants, creating two new program variants.

3.3 Fitness Evaluation

The quality of a program variant is assessed using a fitness function. Our fitness function uses the test suite of the original program (i.e., the positive tests) and the negative tests which exercise the bug. The negative tests allow us to determine when a program variant has successfully repaired the bug, and the positive tests ensure that a variant has retained required program behavior.

To calculate the fitness of a program variant we assemble and link it to an executable binary (x86) or a `class` file (Java bytecode). Programs that fail to assemble or link are assigned a fitness score of 0. Otherwise, we run the program against all positive and negative test cases, recording how many it passes correctly. The fitness score is equal to the weighted sum of the number of positive and negative test cases passed. In general passing a negative test case is worth more, both to compensate for the fact that there are generally more positive than negative test cases, and to bias the search towards buggy code.

4. EXPERIMENTAL RESULTS

We compared the assembly code and C AST representations [11], using the benchmarks of the original work. The benchmarks range from standard software engineering errors (e.g., crashes or infinite loops) to security vulnerabilities (e.g., remote buffer overflows). The programs themselves are generally Unix system utilities, but include one webserver. Each program typically includes five positive test cases and one negative test case. Two programs, atris and flex, are not included, because the testing rig and test cases provided for them did not sufficiently isolate runaway and ill-behaved variants.

All experiments were run using populations of 40 program variants, and terminated after a maximum of 10 genera-

tions. We generate 90% of each generation's new program variants using mutation, and 10% using crossover, in both tournaments of size 3 are used to select fit individuals for reproduction.

Table 1 shows the results of these two representations. The repair process is stochastic; the '% Success' column gives the percentage of random trials that yield a valid repair. The dominant time cost of the repair process is evaluating the fitness of a variant on the test suite; the 'Fitness per Success' column gives the average number of fitness evaluations required to find a repair. Since each run is independent (and, indeed, the fitness evaluation of each variant is independent), runs are conducted in parallel, terminating at first success. Thus if the success rate of a repair is 25% and it requires 100 test suite evaluations, we say that it takes on average 400 test suite evaluations before that repair is discovered.

The average number of fitness evaluations required to produce a repair is 63.6 for C and 74.4 for assembly: only 17% more work is required, on average, for assembly level repairs. These performance results were obtained using EC parameters taken from previous work for the purpose of direct comparison. EC applications more typically use population sizes and number of iterations in the hundreds or thousands. When we select parameters more suited to assembly-level repair, performance improves significantly with the success rate of `ultrix-look` rising from 60% to 100%, and of `ultrix-deroff` rising from 1% to 21%.

This comparison shows that evolution at the assembly code level has reasonable performance when fixing bugs in legacy software tools. Even at the scale of thousands of lines of code, and tens of thousands of assembly code instructions, the solutions were normally found with very few runs (under 40 fitness evaluations on average across every program in Table 1). This supports earlier results showing that EC program repair time scales with the size of the weighted path (fault localization) [3].

4.1 Generality to Multiple Languages

The algorithm shown in Figure 1 calculates the greatest common denominator of two numbers. This algorithm contains a bug: When $a = 0$ and $b \neq 0$ the program enters an infinite loop in lines 4 to 10. This algorithm was coded by the authors in C, Haskell, and Java, and repairs were generated for the assembly code compiled from each program. The results are shown in Table 2, and demonstrate that it is indeed possible to use a single setup to repair defects in a variety of programming languages. 'Program length' refers to the total number of assembly-code instructions of the baseline individual. 'Unique Solutions' is the total number of unique solutions found in 500 total runs with populations of 40 program variants.

```
Input:  Integer a
Input:  Integer b.
Output: gcd(a, b)
  or infinite loop
 1: if a = 0 then
 2:    print a
 3: end if
 4: while b ≠ 0 do
 5:    if a > b then
 6:       a ← a − b
 7:    else
 8:       b ← b − a
 9:    end if
10: end while
11: print a
```

Figure 1: A buggy Euclid's algorithm.

Table 2: GCD repair results for 3 source languages.

	C	Haskell	Java
Program Length	79	885	33
Unique Solutions	2	10	1

5. DISCUSSION AND FUTURE WORK

Section 4.1 shows that linear arrays of assembly code instructions obtained from legacy programs can be repaired automatically using EC. Previous EC work focused on hundred-line assembly programs; we present results on thousands of lines (e.g., 15428 lines for `indent`). The fine-grained expressive power of operations at the assembly code instruction level can, in some cases, effect repairs that are currently not possible at the source-code statement level. The generality of assembly code extends the potential reach of this method to a wide range of programming languages, greatly expanding its applicability compared to previous automated bug correction techniques. The large number of assembly code statements present in even trivial programs ensures that the genetic operations have a full library of existing code from which to construct new program variants.

We were surprised to discover that most of the repairs described in this paper required a small number of genetic operations, in some cases, only one swap of assembly instructions. We experimented with more complex operators, which not only rearranged assembly code instructions, but also altered the values of arguments and labels. These finer-grained alterations rarely improved our results, and generally slowed down search time. We conjecture that our results could be improved significantly by optimizing and enhancing the EC parameters and operators (e.g. context-aware mutation and improved crossover), and we leave the question of carefully evaluating repair quality to future work.

Our methodology is automatable but not yet a turnkey solution. The configuration of the testing rig is more complex compared to previous work. Also, existing program behavior that is not protected by positive test cases could be compromised, although that has not been a problem in practice. Finally, it is not yet clear how many complex bugs can be fixed without inventing any new code.

5.1 Fitness Evaluation and Safety

Evaluating programs created by randomly mutating assembly code is not trivial. For example, `gcc` or `as` would often stall indefinitely without throwing an exception when transforming mutated x86 assembly into object files. We adopted an eight second timeout on the process and all test cases.

To protect host systems, fitness is evaluated in a virtual machine. Because the genetic operators do not include safety checks, it is possible for arbitrary code to be executed on the machine used for evaluation. "Stack Smashing" errors were common and program variants were routinely terminated by security measures built into the host Linux operating system. Similarly, some ill-behaved program variants would not respond to standard termination signals. The use of safety and isolation measures built in to the processor and OS to selectively cull unfit program variants is a standard feature of EC approaches [6].

Table 1: Performance comparison between C AST Assembly ('ASM') code program representations: Program size and fault localization are expressed in terms of statements for C and assembly instructions with non-zero negative weights for the Assembly programs. '% Success' gives the number of runs out of 100 that produced a successful repair. 'Fitness per Success' gives the average number of times the test suite was evaluated per successful repair. 'ASM Operations per Repair' shows the frequency with which each genetic operator produced a successful repair.

	Program Size		Fault Localization		% Success		Fitness per Success		ASM Operations per Repair				
Program	C	ASM	C	ASM	C	ASM	C	ASM	total	swap	del	ins	cros
ultrix-uniq	1146	486	81.5	3	100	75	9.5	30.8	1	21	26	28	0
ultrix-look	1169	541	213.0	12	99	60	11.1	27.6	1	37	15	8	0
look	1363	565	32.4	6	100	98	8.5	39.0	1	37	37	24	0
units	1504	1494	2159.7	125	7	2	55.7	15.5	1	2	0	0	0
ultrix-deroff	2236	7041	251.4	13	97	1	21.6	23.0	1	1	0	0	0
nullhttpd	5575	6933	768.5	13	36	1	79.1	27.0	1	1	0	0	0
indent	9906	15428	1435.9	80	7	1	95.6	14.0	1	0	0	1	0
average	3271	4641	705.1	36	63	34	40.1	25.3	1	14	11	8	0

5.2 Future Work

The functional properties of common programs (as defined by test suites) are remarkably robust to program manipulation at the assembly level, a property known as *mutational robustness*. Preliminary investigations of programs compiled from different source languages suggests significant differences in the mutational robustness of their assembly representation. We hope to examine both the surprising overall level of robustness of program functionality to mutational changes, and the differences in robustness in similar programs compiled from different source languages.

More speculatively, we believe that this technique could have practical applications beyond software repair. First, seemingly innocuous machine-specific environmental factors can have surprisingly large impacts on the performance of common utility programs [7]. A variation of our technique could be used to optimize such utilities to each machine on which they are used. A second potential application is disruption of software monocultures [4]. In security settings, randomization is often inserted into compiled programs to prevent malicious attacks. Assembly code evolution could be used for widely distributed programs to add diversity to deployed software.

5.3 Conclusion

This work presents a simple, powerful, and general mechanism for automatically repairing legacy software by evolving assembly code programs. We presented empirical results comparing our approach to the previous state-of-the-art in EC repair, showing that assembly level EC is nearly as efficient as source code EC. Although some may object to attempting automated assembly code repair at all, we believe that it will increase the class of repairs that can be addressed by EC and is an important steps toward making automated repair practical.

5.4 Acknowledgments

The authors gratefully acknowledge the support of NSF grants CCF 0621900 (SF), CCR-0331580 (SF), CNS 0716478 (WW), CNS 0905373 (WW), and SHF-0905236; AFOSR MURI grant FA9550-07-1-0532, and the Santa Fe Institute.

6. REFERENCES

[1] M. Dahm. Byte code engineering with the BCEL API. Technical report, Freie Universität Berlin, 2001.

[2] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Automated Software Engineering*, 2009.

[3] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computing Conference*, 2009.

[4] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. Pfleeger, J. Quarterman, and B. Schneier. Cyber insecurity: The cost of monopoly. Technical report, Computer & Communications Industry Association, 2003.

[5] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT, 1992.

[6] F. Kühling, K. Wolff, and P. Nordin. Brute-force approach to automatic induction of machine code on CISC architectures. In *European Conference on Genetic Programming*, pages 288–297, 2002.

[7] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Architectural support for programming languages and operating systems*, pages 265–276, 2009.

[8] P. Nordin, W. Banzhaf, and F. D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In *Advances in Genetic Programming 3*, pages 275–299. June 1999.

[9] M. Orlov and M. Sipper. Genetic programming in the wild: evolving unrestricted bytecode. In *Genetic and evolutionary computation*, pages 1043–1050, 2009.

[10] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles*, pages 87–102, October 2009.

[11] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.