© 2016 IJSRSET | Volume 2 | Issue 2 | Print ISSN : 2395-1990 | Online ISSN : 2394-4099
Themed Section: Engineering and Technology

A Survey Paper on GenProg : A Genetic Technique for Software Repair

Chandrashekhar S. Pawar
Department of Computer Technology, R.C.P.I.T., Shirpur, Maharashtra, India

ABSTRACT

GenProg is a mechanized technique for repairing defects in off-the-rack, legacy programs without formal particulars, program explanations or exceptional coding practices. GenProg utilizes a stretched out type of genetic programming to develop a program variation that holds required usefulness however is not susceptible to a given defect, utilizing existing test suites to encode both the imperfection and required usefulness. GenProg might be connected either to program techniques for consequently detecting software defects source or modules.

Keywords: GenProg, Fitness, Mutation

I. INTRODUCTION

Genetic programming is consolidated with program analysis techniques to repair bugs in the different programs. In Genetic Programming (GP) has not supplanted human software engineers, It create, keep up, and repair computer programs to a great extent by hand. In GP can be used in genetic technique program examination strategies to repair legacy programs. It expect that have entry to the C source code, a negative experiment that activities the deficiency to be repaired, and a few positive experiments that encode the required conduct of the program. This program for use developmental calculation is a promising strategy for automating time-consuming and costly programming support assignments, including bug repair. This GenProg concentrating on:[1]

(1) Representation of individual variations
(2) Crossover plan
(3) Mutation operators
(4) Search space definition.

II. LITRATURE SURVEY

W. Weimer et al.[1] are used genetic programming to develop program variations until one was found that both hold required usefulness. Standard experiments are utilized to practice the deficiency and to encode program prerequisites. After an effective repair has been found, it is minimized utilizing basic differencing calculations and delta investigating.

Arcuri et al. [2, 3, 4] are utilized GP to automate the repair of software bugs, demonstrating the design on a hand-coded model of the bubble sort algorithm.

Demsky et al. [5] proposed a strategy for data structure repair. Given a formal detail of data structure consistency, run-time observing code is embedded that "patches up" conflicting state so that a surrey system can keep on executing if the data structures ever ended up conflicting.

W. Weimer et al.[7] discribe combined program analysis strategies with transformative calculation to consequently repair bugs in off-the-shelf legacy C programs.

III. CONCEPTS

A. Genpro Concepts

This method also to solve the bug problem in software they use in evolutionary methods have been used to repair programs automatically, with promising results but this fitness function for repair software bugs . However, the wellness capacity used to accomplish these outcomes depended on a couple of basic

experiments and is likely excessively oversimplified for bigger programs and more complex bugs. It focus here on two parts of wellness assessment: effectiveness and exactness. This describes and evaluates Genetic Program Repair ("GenProg"), a technique that uses existing test cases to automatically generate repairs for real-world bugs legacy applications. It utilize the expressions "repair" and "patch" interchangeably. GenProg does not require formal particulars, program annotations, or exceptional coding practices. We introduce three key innovations to address this longstanding problem:-

1. GenProg works at the statement level of a program's **abstract syntax tree** (AST) expanding the search granularity.
2. GenProg to discover the weighted path comprising of a list of program statements each connected with a weight based on that statement's event in different experiment execution follows.
3. GenProg operates **critical Fault** confinement is, when all is said in done, a hard and unsolved issue. It describe "GenProg" to use in a software quality to remove the bugs in a program and it is very good method to remove the bugs and getting the good result. It is also saving the time and solves the problem in software. Fixing bugs is a difficult, time-consuming, and manual process. Some reports place programming maintenance, generally characterized as any alteration made on a framework after its delivery, at 90% of the aggregate expense of a typical software project. This technique takes as data a program, an arrangement of successful positive experiments that encode required project conduct, and a failing negative test case that shows an imperfection .In GenProg uses various techniques for solving bugs in a program.

B. Software Repair Concept

It uses 3 function for repairing the software program. This method is very useful and fast process to solve the bug problem. GenProg may provide utility as a debugging aid alternately by incidentally tending to bugs that would some way or another take days to patch or require inconvenient temporary solutions, a utilization case we investigated in our closed loop repair model. GenProg uses a representation that combines abstract syntax trees with weighted violating paths; these bits of knowledge permit our search to scale to vast programs. In GenProg uses 3 steps to software repair:-

1) Negative test case to Positive test case

GenProg takes as input source code containing a defect and a set of test cases, including a failing negative test case that exercises the defect and a set of passing positive test cases that describe requirements. A program passes a test case if it produces the expected output when run on the input, as defined by the oracle comparator; otherwise, it fails the test case. A positive experiment is a standard (relapse) experiment that encodes right program conduct; the program's current test suite comprises the positive experiments. A negative experiment is a program data that shows the bug and a comparator that identifies it.

2) **Fitness Function**

In GP, the fitness function is an objective function used to evaluate variants. The fitness of a person in a program repair task should assess how well the program maintains avoids the the program bug while as yet doing "everything else it is supposed to do."

It use test cases to measure fitness. In fitness function encodes software requirements at the test case level: negative test cases encode the fault to be repaired, while positive test cases encode functionality that cannot be sacrificed.:-
WPosT :- It successful positive test is weighted by the global parameter

WNegT:- It successful negative test is weighted by the global parameter

The fitness function is thus simply the weighted sum

$$\text{Fitness}(P) = WPosT * |\{t \in PosT \mid P \text{ passes } t\}| + WNegT * |\{t \in NegT \mid P \text{ passes } t\}|$$

3) Mutation Function:-

It has a little risk of changing a specific statement along the weighted path Changes to statements.

C. Genpro Analysis Concept

When user wants to solve the bug in the program then GenProg to solve this problem. Some keywords into a GenProg then related this keyword, there are many bug are problem to be created in a particular program.

This technique that uses genetic programming to evolve a version of a program that retains required functionality while avoiding a particular error. GenProg may provide utility as a debugging aid or by temporarily addressing bugs that would otherwise take days to patch alternately require unfavourable makeshift arrangements, a utilization case we investigated in our closed loop repair model. GenProg utilizes a representation that consolidates abstract syntax trees with weighted damaging ways; these bits of knowledge permit our search to scale to vast programs.

In the long term, the technique have described leaves considerable space for future examination concerning the repair of new types of bugs and programs and the effects of automatic repair on program readability, maintainability, and quality. In future to add this method Investigate Mutational Robustness, Improve GP Technique, on-Repair Evolution. It is an automatic repair may provide a first step toward the automation of many aspects of the program development procedure.

Program	Positive Tests	|Path|	Initial Repair				Final Repair			
			Time	Fitness	Success	Size	Time	Fitness	Size	Effect
gcd	5x human	1.3	153 s	45.0	54%	21	4 s	4	2	Insert
zune	6x human	2.9	42 s	203.5	72%	11	1 s	2	3	Insert
uniq	5x fuzz	81.5	34 s	15.5	100%	24	2 s	6	4	Delete
look-u	5x fuzz	213.0	45 s	20.1	99%	24	3 s	10	11	Insert
look-s	5x fuzz	32.4	55 s	13.5	100%	21	4 s	5	3	Insert
units	5x human	2159.7	109 s	61.7	7%	23	2 s	6	4	Insert
deroff	5x fuzz	251.4	131 s	28.6	97%	61	2 s	7	3	Delete
nullhttpd	6x human	768.5	578 s	95.1	36%	71	76 s	16	5	Both
openldap	40x human	25.4	665 s	10.6	100%	73	549 s	10	16	Delete
ccrypt	6x human	18.01	330 s	32.3	100%	34	13 s	10	14	Insert
indent	5x fuzz	1435.9	546 s	108.6	7%	221	13 s	13	2	Insert
lighttpd	3x human	135.8	394 s	28.8	100%	214	139 s	14	3	Delete
flex	5x fuzz	3836.6	230 s	39.4	5%	52	7 s	6	3	Delete
atris	2x human	34.0	80 s	20.2	82%	19	11 s	7	3	Delete
php	3x human	30.9	56 s	15.5	100%	139	94 s	11	10	Delete
wu-ftpd	5x human	149.0	2256 s	48.5	75%	64	300 s	6	5	Both
average		573.52	356.5 s	33.63	77.0%	67.0	76.3 s	8.23	5.7	

Figure 1. Results on 120K lines of program or code

This figure.1 shows the averages for 100 random trials. The “Positive Tests” column describes the positive tests. The “Path” columns give the weighted path length. “Initial Repair”. It gives the average performance for one trial, as far as "Time" (the normal time taken for each effective trial), "fitness" (the normal number of fitness assessments in a successful trial), "Success" (what number of the arbitrary trials brought about a repair). "Size" reports the normal UNIX diff size between the first source and the essential repair, in lines. "Final Repair" reports the same data for the creation of a 1-minimal repair from the first initial repair found; the minimization prepare dependably succeeds. "Effect" depicts the operations performed by a demonstrative last patch: A patch may insert code, erase code, or both insert, and erase code.

D. Repair Minimization Concept

Once a variation is found that passes through the greater part of the experiments. It minimizes the repair before displaying it to engineers. Because of the irregularity in the mutation and crossover calculations, it is likely that the effective variation will incorporate insignificant changes that are hard to examine for rightness. It wishes to deliver a patch, a rundown of alters that, when connected to the first program, repair the imperfection without giving up required usefulness. It joins experiences from delta debugging and tree-organized separation measurements to minimize the repair. Naturally, we create a substantial patch by taking the difference between the variation and the original, and afterward dispose of all aspects of that patch while as yet passing all test cases.

E. Genetic Operator

In GenProg to use various methods for evolutionary computing, particularly genetic programming can optimize software and software engineering, including evolving test benchmarks, look meta-heuristics, conventions, making web administrations, enhancing hashing and trash gathering, repetitive programming and even consequently settling bugs. Frequently there are numerous potential approaches to adjust usefulness with asset utilization.

Be that as it may, a human software engineer can't attempt all of them. Also the optimal trade off may be different on each hardware platform and it could vary over time or as usage changes. It might be genetic programming can consequently recommend diverse exchange offs for each new market.

Program	Fault	|Path|	Time	Fitness	Success	Initial	Final
gcd	infinite loop	1.3	149 s	41.0	54%	21	2
zune	infinite loop	2.9	42 s	203.5	72%	11	3
uniq	segmentation fault	81.5	32 s	9.5	100%	24	4
look-u	segmentation fault	213.0	42 s	11.1	99%	24	11
look-s	infinite loop	32.4	51 s	8.5	100%	21	3
units	segmentation fault	2159.7	107 s	55.7	7%	23	4
deroff	segmentation fault	251.4	129 s	21.6	97%	61	3
nullhttpd	remote heap overflow	768.5	502 s	79.1	36%	71	5
indent	infinite loop	1435.9	533 s	95.6	7%	221	2
flex	segmentation fault	3836.6	233 s	33.4	5%	52	3
atris	local stack overflow	34.0	69 s	13.2	82%	19	3
Average		801.56	171.7 s	52.0	59.9%	55.4	3.9

Figure 2. Results on 63 kLOC.

It report averages for 100 random trials. The 'jPathj' column gives the weighted path length, which approximates search space size. 'Success' reports the percentage of random trials that resulted in a repair. "Time" gives the normal divider clock time, and "Fitness" the normal number of fitness assessments for a successful trial; neither incorporates minimization time. 'Initial' and 'Final' report the normal UNIX difference size for both the initial and the minimized repair.

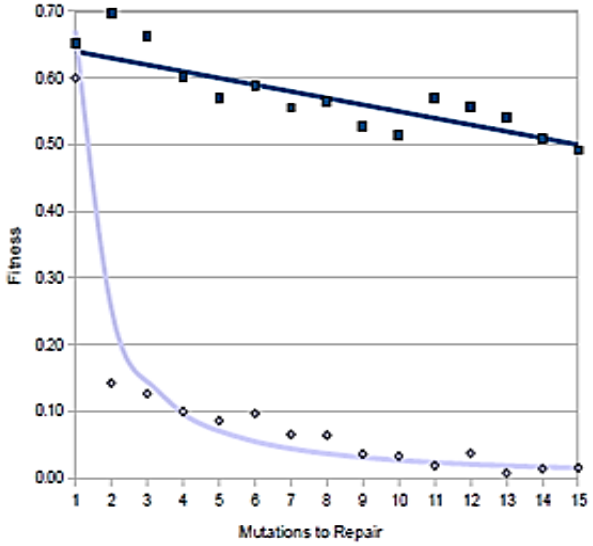

Figure 3. Fitness function performance

IV. DISCUSSION AND CONCLUSION

There are different methods used for GenProg which are fitness function, crossover function, and Mutation operations to solve the bug problem**.** GenProg to use in a software quality to remove the bugs in a program and it is very good method to remove the bugs and getting the good result. It is also saving the time and solves the problem in software. Fixing bugs is a difficult, time-consuming, and manual process. Some reports place programming support, generally characterized as any change made on a framework after its conveyance, at 90% of the aggregate expense of a normal software project.

V. REFERENCES

[1] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically Finding Patches Using Genetic Programming," in Proceedings of International Conference Software Eng., pp. 364-367, 2009.

[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] A. Arcuri, D. R. White, J. Clark, and X. Yao, "Multi-objective improvement of software using co-evolution and smart seeding," in Proceedings of the International Conference on Simulated Evolution And Learning, pages 61–70, 2008.

[4] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in IEEE Congress on Evolutionary Computation, 2008

[5] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in International Symposium on Software Testing and Analysis, pages 233–244, 2006.

[6] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues, "A Genetic Programming Approach to Automated Software Repair," in Proceedings of Genetic and Evolutionary Computing Conference, 2009.

[7] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic Program Repair with Evolutionary Computation," *Comm. ACM,* vol. 53, no. 5, pp. 109-116, May 2010.

[8] W. Weimer, "Patches as Better Bug Reports," in Proceedings of Conference on Generative Programming and Component Eng., pp. 181-190, 2006.