A Capable Crossover Technique on Automatic Program Repair

Ryotaro Kou*, Yoshiki Higo* and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University
Email: r-kou, higo, kusumoto@ist.osaka-u.ac.jp

***Abstract*—In software development, debugging is indispensable to guarantee the reliability. However, debugging is becoming more difficult because software is becoming larger and more complex. Thus, techniques for supporting debugging, especially automatic program repair techniques based on genetic programming distinguish themselves due to their capability. Genetic programming produces many modified programs by three operations: selection, mutation, and crossover. In this research, we focus on crossover that can bring a large modification at one operation, and we are conducting research on capable crossover by selecting several modified programs not randomly but using the guidance indicating their properties. In this paper, we propose a new crossover technique based on comparing modified programs' properties.**

I. INTRODUCTION

Debugging is one of the most important processes in software development since any software requires reliability and safety. Debugging denotes finding faults, locating bugs, fixing bugs, and validating repair. However, debugging is hard work since software becomes larger and more complex, e.g. Baker [1] reports that debugging occupies half of the software development. Thus, many techniques supporting debugging are proposed. Automatic program repair techniques support locating bugs, fixing bugs, and validating repair automatically. Those techniques have been attracted.

Recently, automatic program repair techniques based on genetic programming [2] such as GenProg [3], [4] distinguish themselves [5]. Genetic programming is a search algorithm resembling a biological evolution. Genetic programming is composed of three operations: selection, mutation, and crossover. Selection is to pick out several individuals by evaluating their fitness. Mutation is to generate new individuals by modifying each individual. Crossover is to generate new individuals by combining two individuals. GenProg generates many fixed program candidates as patches through evaluating and modifying them by regarding programs as biological individuals.

There are other automatic program repair techniques that do not use genetic programming such as RSRepair [6], PAR [7], SemFix [8], DirectFix [9], and more [10], [11], [12]. RSRepair generates a fixed program based on random search. PAR modifies programs with prepared patterns. SemFix and DirectFix modify programs by semantic analysis.

In recent years, many research studies are conducted on automatic program repair, but few of them focus on crossover on techniques using genetic programming. We consider there is a large room for improvement on crossover because crossover of existing techniques are mainly conducted by random algorithms. In this research, we focus on crossover which can bring a large modification in one operation.

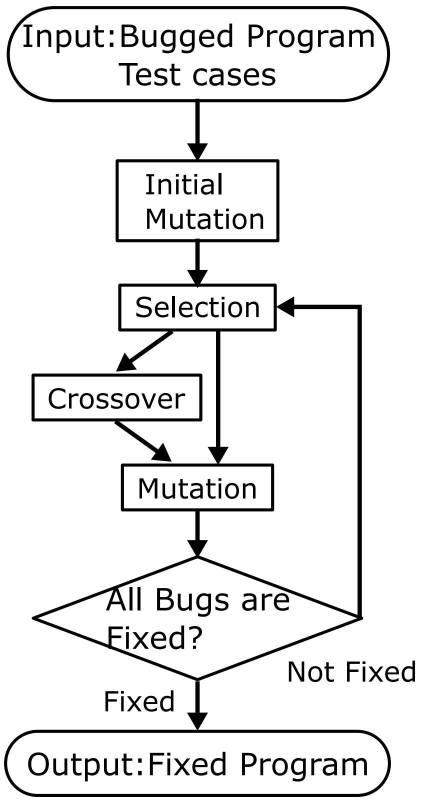


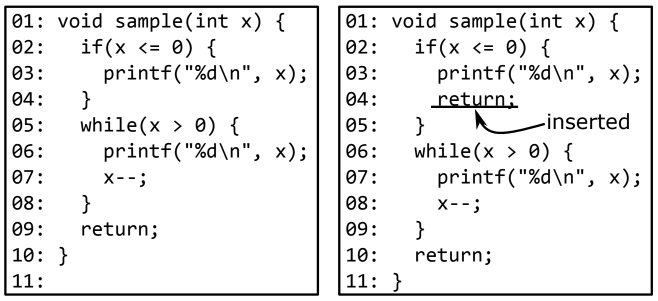
Fig. 1. Abstract of GenProg

We propose a new crossover technique, which selects target programs by their properties and combines programs using a couple of strategies. We also implement our technique as a tool, Marriagent. In the experiments, we compared Marriagent with GenProg and confirmed that Marriagent outperformed GenProg in capability to bug fix.

II. PRELIMINARIES

A. Genetic Programming

Genetic programming is a search algorithm resembling biological evolution. This algorithm aims at a solution by performing the following three kinds of operations iteratively.



```
01: void sample(int x) {
02:   if(x <= 0) {
03:     printf("%d\n", x);
04:   }
05:   while(x > 0) {
06:     printf("%d\n", x);
07:     x--;
08:   }
09:   return;
10: }
11:
```

(a) Original Program

```
01: void sample(int x) {
02:   if(x <= 0) {
03:     printf("%d\n", x);
04:     return;
05:   }
06:   while(x > 0) {
07:     printf("%d\n", x);
08:     x--;
09:   }
10:   return;
11: }
```

(b) Modified Program

Fig. 2. Example of the Modification

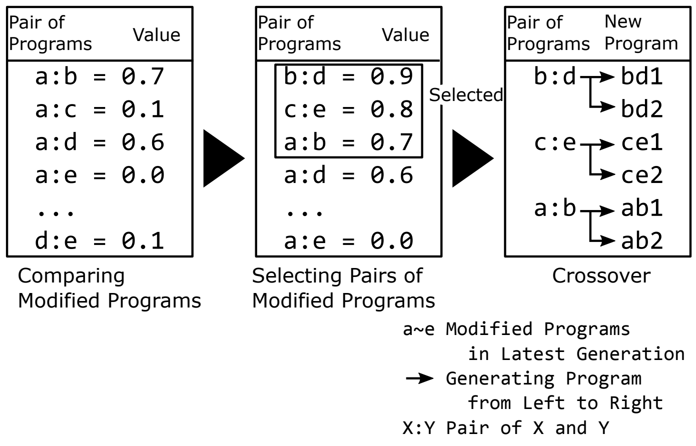


Fig. 3. Flow of Proposed Approach

Selection This operation evaluates the fitness of each individual, and then it picks out several survivable individuals. Fitness is a value which indicates the closeness to the solutions.

Mutation This operation generates new individuals by modifying each old individual.

Crossover This operation generates new individuals by combining two old individuals. Each new individual has half of both old individuals' elements.

B. Related Work

GenProg [3] is an automatic program repair technique based on generic programming. This technique regards fixed program candidates as individuals, and then populations is a set of them. GenProg takes a buggy program and test cases as input, and then it iterates the generations of fixed program candidates as shown in Fig. 1. If a fixed program candidate passes all the test cases, it is output as a fixed program. GenProg executes each operation of genetic programming as follows.

Selection This operation executes every fixed program candidate with all the test cases to evaluate its fitness. Then, this operation picks up several programs having high fitness.

Mutation This operation adds a modification to each program. A mutation operation is either of inserting a program statement existing in the program to the faulty code region, deleting a program statement in the faulty code region, or replacement, which is a combination of insertion and deletion.

Crossover This operation swaps half of modifications of two programs.

In GenProg, each element of the individual is not program's statement or token, but a modification. Fig. 2 shows an example of modification. Fig. 2b is a modified version of the program shown in Fig. 2a. GenProg regards this modification such as "Add the 9th line to behind 3rd line." Hence, the original program is an individual having no element. In addition, modifications are no particular order, altogether those can change their orders without any effect on program behavior.

Several techniques related to GenProg have been proposed. RSRepair [6] introduces random search instead of genetic programming. It generates many fixed program candidates with only a modification without iterating generations. It is suited for bugs which can be fixed by only a single modification.

PAR [7] modifies programs using some fix patterns which have been prepared manually, e.g. altering method parameters, adding a null checker, and changing a branch condition. Modifications by patterns are adaptable to many bugs in general programs and modified programs have good understandability.

SemFix [8] generates repair constraints using symbolic execution with test cases, and then it modifies programs so as to satisfy all the constraints. A remarkable feature of SemFix is that it does not reuse existent statements but generates a new statement that does not exist.

DirectFix [9] is an extending technique of SemFix. It can fix programs having more simplicity and understandability.

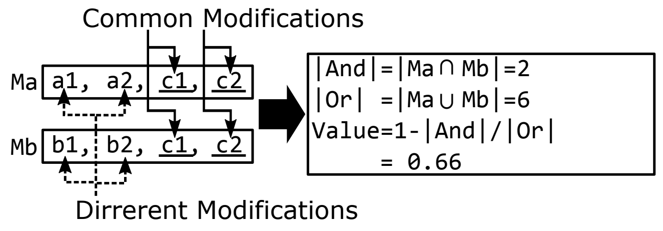
SPR [10] generates fixed program by the algorithm named Staged Program Repair, which is composed of parameterized transformation schemes, target value search, and condition synthesis. It efficiently searches for rich search spaces and then it generates fixed programs correctly.

Prophet [11] learns a probabilistic model from successful patches written by humans, and it generates fixed programs and validates them using the model. It has the good ability to generate successful programs.

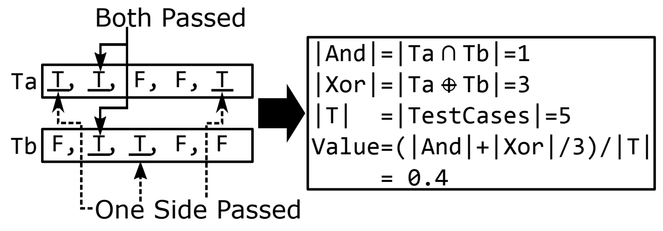
A technique proposed by Le, et. al. [12] mines the fix patterns from the history of many projects and it uses them to fix. The fixed programs based on the history have good-quality as compared to the baselines.

C. Feature of Crossover

The techniques shown in Subsection II-B improve selection or mutation by focusing on their potential. For instance, RSRepair improved selection, and PAR improved mutation. On the other hand, crossover also has a large room for improvement. In this research, we focus on crossover and its features. By comparing crossover and mutation, crossover has following features.



(a) Similarity



(b) Test Cases

Fig. 4. Comparing Modified Programs

- It can bring a large modification with one operation.
- It can remove modifications added in old generations.
- It combines existent modifications.

We considered that crossover was able to generate programs including required modifications by selecting programs to make use of above features. However, GenProg often wastes them: e.g. it may crossover similar programs and generate programs with little difference. Hence, we propose a new crossover technique based on comparing modified programs' properties.

III. PROPOSED TECHNIQUE

In the proposed technique, selection and mutation are the same as the ones described for GenProg's algorithm. Fig. 3 shows three steps in crossover of our technique as follows:

Comparing Modified Programs This step compares each pair of latest modified programs' properties and sets values of them.

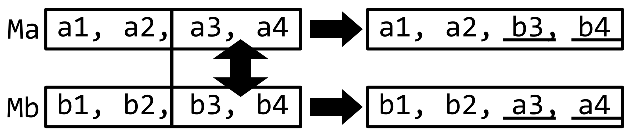
Selecting Pairs of Modified Programs This step selects several pairs in order of high value.

Crossover This step generates new modified programs by crossover selected pairs.

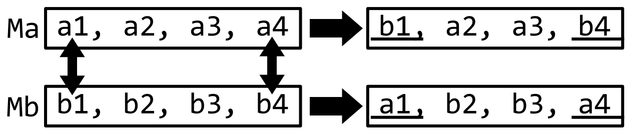
In the reminder of this section, we explain the details of the algorithm for each step.

A. Comparing Modified Programs

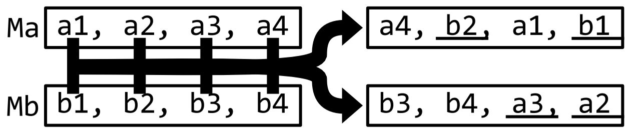
Fig. 4 shows our proposed algorithm, which compares modified programs' properties. It is proper to select the programs having different modifications to bring a large modification. Thus, the algorithm shown in Fig. 4a evaluates a similarity between programs. The similarity is calculated by Jaccard coefficient shown in exp. (1), because it can be calculated very quickly. Then proposed algorithm calculates a quantitative value of the similarity shown in exp. (2). The less similarity indicates the better pair. Its value ranges from 0 to 1.



(a) One Point Crossover



(b) Uniform Crossover



(c) Random Crossover

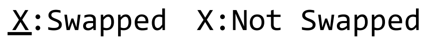


Fig. 5. Crossover Strategy

$$JaccardCoef. = \frac{|M_a \cap M_b|}{|M_a \cup M_b|} \quad (1)$$

$$ValueofSimilarity = 1 - \frac{|M_a \cap M_b|}{|M_a \cup M_b|} \quad (2)$$

- M_a, M_b: a set of modifications in programs a or b.
- $A \cap B$: common modifications between A and B.
- $A \cup B$: union modifications between A and B.
- $|A|$: the number of elements in A.

By contrast, it is also proper to select the programs having the good fitness to generate a good program. Thus, the algorithm shown in Fig. 4b evaluates whether each program passes each test case. Then, the proposed technique calculates the values of test cases shown in exp. (3). That value also ranges from 0 to 1.

$$ValueofTestCases = \frac{|T_a \cap T_b| + |T_a \bigoplus T_b|/3}{|TestCases|} \quad (3)$$

- T_a, T_b: a set of booleans whether programs a and b pass each test case or not.
- $A \cap B$: true in common between A and B.
- $A \bigoplus B$: true in sole either A and B.
- $|A|$: the number of elements in A.

It is possible to use both values of similarity and test cases because those are independent on each other. In the experiments, we also evaluated a pattern multiplying both of them, its value also ranges from 0 to 1.

B. Selecting Pairs of Modified Programs

In this step, the proposed technique selects several program pairs in the descending order of values calculated in the last section. The number of conducted crossover is shown in exp. (4), which is an expected value of the number of crossover by GenProg. The reason we use it is to prevent too much increasing of programs by crossover. Besides, the same program can be chosen more than once such as program "b" shown in Fig. 3.

$$NumberOfCrossover = \frac{|Programs| \times Probability}{2} \quad (4)$$

$Programs$ Modified programs in the latest generation.

$Probability$ The probability of crossover. GenProg sets it as a configuration option; default value is 0.5.

C. Crossover Strategy

Fig. 5 shows three strategies for swapping modifications, which are listed as follows.

One Point this crossover splits sets of modifications at center and swaps the latter halves.

Uniform This crossover swaps each modification on the same indices with 50% probability. If the surplus modifications exist, those are transfered with 50% probability.

Random This crossover combines all modifications and distribute them to programs randomly. The generated programs have the same number of modifications.

GenProg uses one point crossover, which suffers from hitchhiking [13]., which is an issue that adjoining elements cannot be divided into different next-generation programs. In Fig. 5, if element *a1* is suitable and *a2* is harmful, it is necessary to divide them to generate a fixed program. Uniform crossover and random crossover can divide them, but one point crossover cannot. Thus, GenProg is not good at dividing adjoining elements.

IV. EXPERIMENTS

We implemented the technique as a tool, Marriagent, which is and extension of GenProg. We compared Marriagent and GenProg by investigating rates of success and their runtime to fix.

TABLE I
SUBJECT PROGRAMS

Version	LOC	Fault(s)	Description for Fault
version 9	173	1	Assignment statement
version 43	175	1	Control statement
version 44	175	2	Assignment statement and Control statement

A. Subject Programs

We use the subject program tcas, which is a traffic collision avoidance system, from Software-artifact Infrastructure Repository (SIR) [14]. The versions we used are shown in Table I. Version 9 has a fault on assignment statement, which is easy to fix. Version 43 has a fault on control statement, which is hard to fix. Version 44 has two faults on assignment statement and control statement, which cannot be fixed by one modification.

B. Experimental Setup

We executed GenProg and Marriagent on the subject programs 50 times, and then we recorded the number of success within 15 minutes and runtime to fix. In this experiment, we defined fix as generating the program which passes all test cases, but do not consider its validity in the real world.

Note that some executions fixed programs by initial mutation, which mean no crossover was performed. Hence, we also recorded the results excluded such execution because it does not show the capability of our proposed technique.

Table II shows the configurations of GenProg and Marriagent. The configurations not shown in Table II are default settings of GenProg. Seeds of random number generation are set in from 1 to 50 at each execution. Generations are iterated unlimitedly during 15 minutes. In addition, Table III shows the unique configurations of Marriagent. Each column is shown in Subsection III. In the case of Both Uses, Marriagent multiplies two values of Similarity and Test Cases.

In this experiment, we ran on Ubuntu 14.04 for 64-bit machine, and with 2.00 GHz Intel (R) Dual-Core CPU and 40

TABLE II
EXPERIMENTAL SETUP

Column	Value	Column	Value
Positive Tests	100	Negative Tests	9
Populations	10	Probability of Crossover	0.5
Seeds	Variable	Generations	Infinity

TABLE III
EXPERIMENTAL SETUP FOR MARRIAGENT

Pattern	Comparison	Crossover
MarriagentSO	Similarity	Single point
MarriagentSU		Uniform
MarriagentSR		Random
MarriagentTO	Testcases	Single point
MarriagentTU		Uniform
MarriagentTR		Random
MarriagentBO	Both Uses	Single point
MarriagentBU		Uniform
MarriagentBR		Random

GB of memory. Both techniques are build on OCaml 3.12.1 and gcc 4.8.

C. Experimental Results

Table IV shows results of the experiments. In version 44, some of the patterns could not fix even once. In such cases, average times in Table IV are expressed as "No Fixed". Described in Subsection IV-B, some executions finished by initial mutation without crossover, and then the results excluded those executions are expressed in column Exclude 0 Gene.

Table IV indicates that Marriagent especially using random crossover and comparison by test cases outperforms GenProg in the number of the fix at version 9 and 43. GenProg did not fix version 44 even once, but Marriagent occasionally fixed by using some patterns. The pattern using random crossover and comparison by test cases is the best in all versions.

In version 9, the average runtime of Marriagent was almost as much as GenProg. By contrast in version 43, the average runtime of Marriagent is various, some are half of runtime of GenProg and others are twice as GenProg. In version 44, only Marriagent fix the bug.

V. DISCUSSION

A. Comparison of Modified Programs

In version 9 and 43, our technique using comparison by similarity achieved good results. We consider that an idea for bringing a large modification is suitable.

In the case of using comparison by test cases, our technique using random crossover is the best, but the others are producing worse results overall. We consider the reason is that our technique often selects the same programs more than once because it compares programs by test cases. Hence, a new program generated by one point crossover or uniform crossover tends to be similar to its parents. In contrast, random crossover can generate a new program which is dissimilar to its parents frequently.

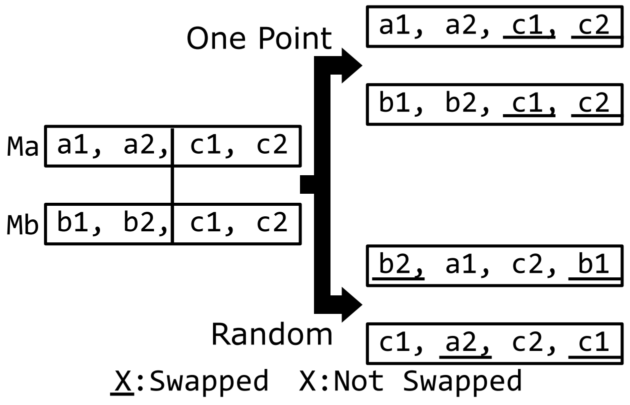


Fig. 6. An example of crossover on similar programs

Besides, our technique using comparison by both similarity and test cases are not good in the number of the fix. Thereby we consider these properties are unsuited to each other. In genetic programming, more iterated generations, more similar individuals tend to be generated. In particular, our technique using comparison by test cases more generates similar programs. For this reasons, comparison by similarity is hard to evaluate programs. However, average times of that pattern are fast. Then we think it is strong in early generations because few programs are similar to each other.

From the above, both properties have a merit. However, a combination of them requires the plan: e.g. using weight, switching by generations or fitness.

B. Crossover Strategy

Our technique achieved good results by using random crossover. We consider the reason is that random crossover can make a large modification even for the similar programs. An example of crossover for similar programs shown in Fig. 6. Programs *Ma* and *Mb* contain the common modifications *c1*

TABLE IV
EXPERIMENTAL RESULTS

	Version 9				Version 43				Version 44			
Approach	Fixed Count	Exclude 0 Genes	Average time (s)	Exclude 0 Gene.	Fixed Count	Exclude 0 Genes	Average time (s)	Exclude 0 Gene.	Fixed Count	Exclude 0 Gene.	Average time (s)	Exclude 0 Gene.
GenProg	18	13	190.17	251.54	7	4	147.22	234.92	0	0	No Fixed	
MarriagentSO	27	22	167.98	199.26	12	9	145.06	183.24	0	0	No Fixed	
MarriagentSU	27	22	207.21	248.06	8	5	90.69	128.60	0	0	No Fixed	
MarriagentSR	27	22	179.31	213.17	12	9	215.69	278.54	0	0	No Fixed	
MarriagentTO	25	20	180.67	218.07	8	5	183.28	276.81	1	1	844.01	844.01
MarriagentTU	22	17	169.98	210.99	8	5	181.82	274.30	0	0	No Fixed	
MarriagentTR	34	29	216.29	248.36	18	15	275.89	325.39	3	3	377.42	377.42
MarriagentBO	20	15	159.93	203.18	8	5	266.80	409.93	0	0	No Fixed	
MarriagentBU	22	17	172.54	214.43	8	5	99.34	141.16	0	0	No Fixed	
MarriagentBR	28	23	165.34	194.69	11	8	113.95	145.43	1	1	371.64	371.64

and *c2* in latter halves. By using one point crossover, generated programs are the same as their parents because of swapping the same modifications. In contrast, by using random crossover, generated programs can be different from their parents.

Our technique using one point crossover or uniform crossover is worse than using random crossover. Uniform crossover is tolerant of hitchhiking as well as random crossover, but uniform crossover is outperformed by random crossover in experiments results. Hence, we think hitchhiking affects the capability of our technique slightly. We consider the reason is that a program including harmful modifications is to be deleted in selection because its fitness is low. In other hand, random crossover has a merit described above, and that is why the difference appears between random crossover and uniform crossover.

VI. Threats to Validity

We proposed a crossover technique, which compares programs' properties. We used Jaccard coefficient to evaluate similarity because its calculation is light, but we did not check other measures for programs' similarity: e.g. Levenshtein distance. In addition, we used exp. (3) to evaluate test cases, its values were set by our intuition. Hence, there may be a better expression using other values: e.g. 1/3 to 1/2 or 1/4.

In experiments, we compared Marriagent and GenProg to evaluate our technique. GenProg has many configurations: e.g. probability of mutation, weight to test cases, and compiler used to evaluate programs. If we set different values to such populations or limit for generations, a result might differ to above because our technique depends on the pair of programs.

Besides, we used three versions of tcas, which have various faults, to subject program in the experiments. However, their LOC are less than two hundred. If we use the program having thousands of LOC or more, the result of our technique, especially using similarity may differ. In addition, our technique is capable because tcas has 109 test cases. If we use the program having a few test cases, the result of our technique may differ. Therefore, we need to evaluate the scalability of our technique and generalize the results.

VII. Conclusion

We propose a crossover technique extending GenProg. Our technique operates crossover by selecting programs based on comparison their properties. Our experimental results show our technique, using comparison by test cases and strategy as random crossover, generates fixed programs more often than GenProg.

In the future works, we plan to develop new strategies in comparison of programs or operating crossover, moreover combining existing techniques related to selection or mutation. Besides, we also plan the experiments using more subjects such as the large software including thousands of LOC to evaluate scalability and generalize experimental results.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers 25220003, 24650011, and 24680002.

References

[1] J. Baker, "Experts battle £192bn loss to computer bugs," Feb. 2012, accessed:2015-12-07, http://www.cambridge-news.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html.

[2] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Dec. 1992.

[3] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "Systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *In 34th International Conference on Software Engineering (ICSE) 2012.*, June 2012, pp. 3–13.

[4] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devandu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering (TSE) 2015*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.

[5] X. Le, T. Le, and D. Lo, "Should fixing these failures be delegated to automated program repair," in *In 26th IEEE International Symposium on Software Reliability Engineering (ISSRE) 2015*, nov. 2015.

[6] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *In 36th International Conference on Software Engineering (ICSE) 2014.*, June 2014, pp. 254–265.

[7] D.Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *In 35th International Conference on Software Engineering (ICSE) 2013.*, May 2013, pp. 802–811.

[8] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *In 35th International Conference on Software Engineering (ICSE) 2013.*, May 2013, pp. 772–781.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *In 37th International Conference on Software Engineering (ICSE) 2015.*, May 2015, pp. 448–458.

[10] L. Fan and R. Martin, "Staged program repair with condition synthesis," in *In 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE) 2015*, sep. 2015, pp. 166–178.

[11] ——, "Automatic patch generation by learning correct code," in *In 43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) 2016*, Jan. 2016, (In appear).

[12] X. Le, T. Le, D. Lo, and C. L. Goues, "History driven automated program repair," in *In 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2016.*, mar. 2016, (In appear).

[13] S. Forrest and M. Mitchell, "Relative building block fitness and the building block hypothesis," in *Foundations of Genetic Algorithms 2*, D. Whitley, Ed. Morgan Kaufmann, Feb. 1993, pp. 109–126.

[14] "Sir: Software-artifact infrastructure repository," http://sir.unl.edu/portal/index.html.