Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools

Shin Hwei Tan*, Jooyong Yi†, Yulis*, Sergey Mechtaev*, Abhik Roychoudhury*

*National University of Singapore
{shinhwei,yulis,mechtaev,abhik}@comp.nus.edu.sg

†Innopolis University
j.yi@innopolis.ru

***Abstract*—Several automated program repair techniques have been proposed to reduce the time and effort spent in bug-fixing. While these repair tools are designed to be generic such that they could address many software faults, different repair tools may fix certain types of faults more effectively than other tools. Therefore, it is important to compare more objectively the effectiveness of different repair tools on various fault types. However, existing benchmarks on automated program repairs do not allow thorough investigation of the relationship between fault types and the effectiveness of repair tools. We present Codeflaws, a set of 3902 defects from 7436 programs automatically classified across 39 defect classes (we refer to different types of fault as defect classes derived from the syntactic differences between a buggy program and a patched program).**

***Keywords*-automated program repair; defect classes; empirical evaluation; benchmark;**

I. Introduction

Bug-fixing is a time-consuming software maintenance activity. Various automated repair tools (e.g., GenProg [1], PAR [2], *relifix* [3], SemFix [4], DirectFix [5], Angelix [6], SPR [7], and Prophet [8] etc.) have been introduced to save the time and effort spent in bug-fixing. Although these repair tools are designed to fix many classes of software faults, different repair tools may fix certain faults more effectively than other tools. Prior work [9] alluded that the failure to identify the target fault types is an important pitfall of automated program repair research. Unfortunately, prior evaluations of repair tools only perform monolithic comparison of repair tools (where two tools are compared on a set of subject programs without considering defect classes) [6, 8]. As the existing benchmarks are not designed specifically for the study of types of repairable defects, it is difficult to evaluate repair tools using the existing benchmark. We specify the following criteria for a benchmark that allows extensive evaluation of repair tools:

C1: Diverse types of real defects.
C2: Large number of defects.
C3: Large number of programs.
C4: Programs that are algorithmically complex
C5: Large held-out test suite for patch correctness verification

TABLE I: The Basic Statistics of Subject Programs in Codeflaws

Measurement	Total/Range	Average
No. of Programming Contests	548	-
No. of Programming Problems	1284	-
No. of Programs	7436	-
No. of Defects	3902	-
Size of Repair Test Suite	2–8	3
Size of Held-out Test Suite	5–350	40
Source Lines of Codes	1–322	36

Prior evaluations on program repair tools [3, 6, 7, 10] have been conducted on the GenProg benchmark [1], which is later expanded into the ManyBugs and IntroClass benchmarks [11]. Although the ManyBugs and IntroClass benchmarks contain 185 and 998 defects, respectively (i.e., satisfy **C2**), they only contain 9 and 6 subject programs, *not* satisfying **C3**. Meanwhile, IntroClass has only simple programs (such as computing the median of 3 given numbers) submitted by students of an introductory programming class and small held-out test suites (i.e., not satisfying **C4** and **C5**). Since existing benchmarks for automated program repairs do not fulfill the listed criteria, we derive a new benchmark, called *Codeflaws*, to facilitate future study of repairable defect classes.

The Codeflaws benchmark consists of 7436 programs in the Codeforces[1] online database. Table I lists the information about the subject programs in Codeflaws. Each *programming contest* consists of multiple *programming problems* (3–5 problems) with various difficulty levels. Each *program* represents one user submission for a specific problem to Codeforces. These programs are submittted by 1653 users with diverse level of expertise. Each *defect* is represented by a rejected submission and an accepted submission. To our best knowledge, in automatic program repair evaluation, our benchmark has the largest number of defects obtained from the largest number of subject programs to date.

To ease the usage of Codeflaws for future experiments on automated repair tools, we provide all the required scripts to run four state-of-the-art repair tools (GenProg, SPR, Prophet and Angelix) in our website: http://codeflaws.github.io/.

[1] http://codeforces.com/

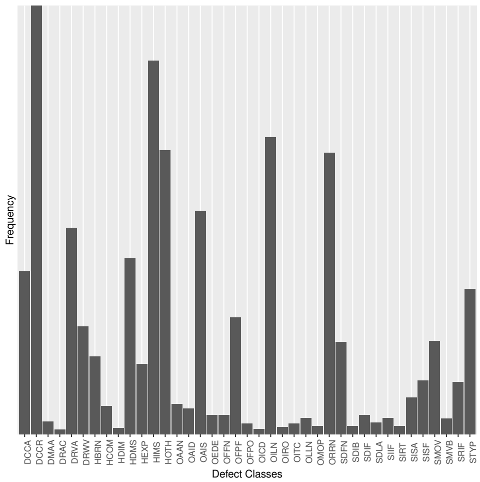

Fig. 1: Distribution of defect classes.

II. METHODOLOGY

We modify *Codeforces-crawler* [12] for our customized crawler to extract data from Codeforces. Starting from a seed page with a list of programming problems sorted by the number of submissions in Codeforces [13], our crawler systematically extracted the information about all submissions for each problem, including the submitter's expertise information, the submission time, the programming problem statement and the test cases used for each submission. We only crawl C programs because most existing program repair tools [1, 5–7] specialize in repairing C programs. Overall, we crawled over 10000 webpages. For each rejected submission r, we find another accepted submission a by the same user for the same programming problem in our crawled data. Each fault is represented by the submission pair (r, a). In total, we obtain 5544 defects. We further exclude 924 defects due to inadequate held-out tests, 677 defects due to non-reproducible bugs, and 41 defects due to a known CIL bugs[2] in handling variable sized multidimensional array.

Software defects can be classified by various criteria (e.g., the symptoms of defects, the causes of defects, and fix operations) [9]. Compared to the ManyBugs and IntroClass Benchmarks [11] that classify software defects based on the defect symptoms, we use a more fine-grained defect class classification based on the syntactic differences between the buggy program and the patched program. We choose this classifications because (1) it allows automatic classification of defect classes, which is essential for handling our large dataset, (2) it is commonly deployed in the literature [14–20], and (3) it enables extensive evaluation of different repair tools.

We modify GumTree [21] to extract syntactic differences at the AST level. AST-level syntactic differences express the AST nodes that are changed (i.e, added/deleted/replaced). Figure 1 shows the distribution of each defect class in Codeflaws. According to Figure 1, some of the most common defect classes are DCCR (replacing constant) and HIMS (insert non-branches), OILN (replace logical operators).

[2]https://sourceforge.net/p/cil/mailman/message/26922529/

TABLE II: Our defect classes and example of each defect class

AST Type	Defect Type	Defect Class	Example
Statement	Control flow	(SDIF) Delete if, else, else if, for or while	– if (lines[i].y1 == last->y1)
		(SIIF) Insert if, else, else if, for or while	+ if(l)
		(SRIF) Replace if, else, else if, for or while	– if(a==b) + if(mask(a)==b)
		(SIRT) Insert return	+ return 0;
		(SDIB) Delete/Insert break or continue	– break;
	Data flow	(SDLA) Delete assignment	– answer+=((i-1)*dif);
		(SISA) Insert assignment	+ t=0;
	Function call	(SDFN) Delete function call	– printf("%s %s\n",s1,s2);
		(SISF) Insert function call	+ scanf("%d", &n);
	Type	(STYP) Replace variable declaration type	– int a; + long a;
	Move	(SMOV) Move statement	– scanf("%d",&i); scanf("%s", &a); + scanf("%d",&i);
		(SMVB) Move brace up/down	– } printf("%d",c); + }
Operator	Control flow	(ORRN) Replace relational operator	– if(sum>n) + if(sum>=n)
		(OLLN) Replace logical operator	– if((s[i] == '4') && (s[i] == '7')) + if((s[i] == '4') || (s[i] == '7'))
		(OILN) Tighten condition or loosen condition	– if(t%2==0) + if(t%2==0 && t!=2)
		(OEDE) Replace = with == or vice versa	– else if(n=1 && k==1) + else if(n==1 && k==1)
		(OICD) Insert a conditional operator	– printf ("%d\n", i); + printf ("%d\n", 3 == x ? 5 : i);
	Arithmetic	(OAAN) Replace arithmetic operator	– v2-=d; + v2+=d;
		(OAIS) Insert/Delete arithmetic operator	– max += days%2; + max += (days%7)%2;
		(OAID) Insert/Delete/Replace ++ or −−	+ i++;
		(OMOP) Modify operator precedence	– ans=max(ans,l-arr[n]*2); + ans=max(ans,(l-arr[n])*2);
	Function call	(OFFN) Alternative function call	– fflush(stdin); + getchar();
		(OFPF) Replace print format	– printf("%d\n",l); + printf("%lld\n",l);
		(OFPO) Modify function parameter order	– if(strcmp(c[i],b)>0) + if(strcmp(b,c[i])>0)
	Pointer	(OIRO) Insert/Delete Reference Operator	– printf("%d",&t); + printf("%d",t);
	Type	(OITC) Insert type cast operator	– ((p2m/p1m)*t+1 + ((float)p2m/p1m)*t+1;
Operan**D**	Constant	(DCCR) Replace constant with variable/constant	– for(i=n+1;i<=9000;i++) + for(i=n+1;i<=10000;i++)
	Variable	(DRVA) Replace a read variable with a variable/constant	– for (i=0;i<l;i++) + for (i=0;i<m;i++)
		(DRWV) Replace a write variable with a variable	– b=0; + a=0;
	Array	(DMAA) Insert/Replace array access	– out[l] = '\0'; + out[l−−] = '\0';
		(DRAC) Replace constant of array initialization	– int ex[2]={0,2}; + int ex[2]={0,3};
		(DCCA) Modify array size	– int x[100] + int x[100000];
Higher order	Non-branch	(HDMS) Delete multiple non-branch statements	– freopen("input.txt", "r", stdin); – freopen("output.txt", "w", stdout);
		(HIMS) Insert multiple non-branch statements	+ freopen("input.txt", "r", stdin); + freopen("output.txt", "w", stdout);
		(HDIM) Delete and insert multiple non-branch statements	– break; + count=0;
	Branch stmt	(HBRN) Delete/Insert branch and non-branch statements	+ if(len%slov!=0){printf("NO"); + return 0;}
	Expressions	(HEXP) Delete/Insert/Replace operators & operands	–if(m*9>=s && s) +if((m*9>=s && !s) || (m==1))
	Combination	(HCOM) Insert/Replace statements and expressions	– rep(i,n) + for(i=n-1;i>=0;i−−)
	Others	(HOTH) Other higher order defect classes	– scanf("%s",h); + for(i=0;i<71;i++) + scanf("%c",&h[i]);

III. CONCLUSION

This paper presents the Codeflaws benchmark that aim to facilitate future empirical study in automated program repair. Given the diverse defects classes and the large number of programs in Codeflaws, developers of new program repair tools could have more objective measurement of the relative effectiveness of their tools compared to other existing tools. We believe that Codeflaws is a step towards the evaluation of program repair tools against multiple dimensions with defect classes being one such dimension.

The Codeflaws benchmark and the 39 defect classes proposed in this paper can be used for systematic study of coding defects in future testing and debugging research. Concrete possibilities include targeted testing/repair techniques for intelligent tutoring systems that teach programming in an interactive fashion, or for a targeted evaluation across defect classes for the efficacy of different test generation strategies.

References

[1] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13.

[2] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *ICSE*, 2013, pp. 802–811.

[3] S. H. Tan and A. Roychoudhury, “relifix: Automated repair of software regressions,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 471–482.

[4] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *ICSE*. IEEE Press, 2013, pp. 772–781.

[5] S. Mechtaev, J. Yi, and A. Roychoudhury, “Directfix: Looking for simple program repairs,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 448–458.

[6] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, pp. 691–701.

[7] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 166–178.

[8] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’16. New York, NY, USA: ACM, 2016, pp. 298–312.

[9] M. Monperrus, “A critical review of ”automatic patch generation learned from human-written patches”: Essay on the problem statement and the evaluation of automatic software repair,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 234–242.

[10] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, “Anti-patterns in search-based program repair,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 727–738.

[11] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The manybugs and introclass benchmarks for automated repair of c programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[12] Z. Zi. (2016) codeforces-crawler. [Online]. Available: https://github.com/Nymphet/codeforces-crawler

[13] (2016) Problemset - Codeforces. (Retrieved 7 June, 2016). [Online]. Available: http://codeforces.com/problemset/?order=BY_SOLVED_DESC

[14] K. Pan, S. Kim, and E. J. Whitehead Jr, “Toward an understanding of bug fix patterns,” *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[15] H. Osman, M. Lungu, and O. Nierstrasz, “Mining frequent bug-fix code changes,” in *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, 2014, pp. 343–347.

[16] C. Liu, Y. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, “An ast-based approach to classifying defects,” in *IEEE International Conference on Software Quality, Reliability and Security, QRS 2015*, 2015, pp. 14–21.

[17] B. Kidwell, J. H. Hayes, and A. P. Nikora, “Toward extended change types for analyzing software faults,” in *International Conference on Quality Software*, 2014, pp. 202–211.

[18] C. Liu, J. Yang, L. Tan, and M. Hafiz, “R2fix: Automatically generating bug fixes from bug reports,” in *ICST*, 2013, pp. 282–291.

[19] M. Martinez, L. Duchien, and M. Monperrus, “Automatically extracting instances of code change patterns with AST analysis.”

[20] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction,” *IEEE Trans. Software Eng.*, vol. 33, no. 11, pp. 725–743, 2007.

[21] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324.