COBOLd: Gobblin' Up COBOL Bugs for Fun and Profit

squaresLab*
Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Mr. squaresLab SpouseMan†
The Private Sector
Pittsburgh, PA 15213

ABSTRACT

The cost and ubiquity of software bugs has motivated research in automated program repair (APR), a field dedicated to the automatic triage and patching of software defects. APR has typically focused on popular languages such as C and Java, but COBOL developers have suffered six decades of neglect and deserve a repair tool too. We present COBOLd, the first tool for automatically repairing buggy COBOL programs. We demonstrate COBOLd's effectiveness on a COBOL reimplementation of the infamous Zune leap-year bug. We also argue that "we got this" and other researchers should stay away so that we can fix all the COBOL bugs ourselves and thus be filthy, stinking rich.

CCS CONCEPTS

•Software and its engineering →Software evolution; Maintaining software; Search-based software engineering; •COBOL repair →$$$$$$$$$$$$$$$$$$$$$$$$$$$$;

KEYWORDS

COBOL, SBSE, Software Evolution, Genetic Programming, $$$$

ACM Reference format:
squaresLab and Mr. squaresLab SpouseMan. 2018. COBOLd: Gobblin' Up COBOL Bugs for Fun and Profit. In *Proceedings of SIGBOVIK, Pittsburgh, PA USA, March 2018 (SIGBOVIK'18),* 5 pages.
DOI: 10.475/123_4

1 INTRODUCTION

1.1 Automatic Program Repair (APR)

Automatic program repair (APR) is all about fixing bugs automatically because software maintenance is expensive. APR techniques take as input (1) a program with a defect and (2) a mechanism to validate correct and incorrect behavior in that program. The validation mechanism is typically a test suite: a bug to be repaired corresponds to one or more tests that is failing; one or more passing tests guard against the removal of desireable functionality. The goal is to produce a patch that modifies the original program such that failing test cases pass without breaking any of the originally passing test cases.

In this work, we automatically fix bugs in COBOL programs using COBOLd, which is just GenProg [10, 11, 17, 18] directly applied to COBOL programs. GenProg uses genetic programming to traverse a space of possible patches to a given buggy program. GenProg uses the test results to define an objective function that informs an evolutionary search [4, 8] over the space of candidate patches. We reuse GenProg because it is well-established and also we know its source code very, very well, having written a large proportion of it. It turns out that, given the appropriate configuration options, the code we already have can totally fix bugs in COBOL programs. Who knew?

2 BACKGROUND

Don't know COBOL? Don't worry. This section has all you need.

2.1 COBOL: Is it reasonable?

2.1.1 Yes. It is incredibly difficult to create a program that contains bugs in COBOL. This is because the design of the language makes it incredibly difficult to write *any* program in COBOL. The lack of advanced features such as *passing parameters to procedures* means that programmers are forced to think very carefully before authoring COBOL programs. Bugs can only exist if programs exist.

2.1.2 Counterpoint: Absolutely Not. The first specification of the language (COBOL-60) contained many logical errors and was impossible to interpret unambiguously [2]. This is unsurprising: no academic computer scientists were on the design committee. The committee invented a new metalanguage to define its syntax because no committee members had heard of Backus-Naur form [14].

Dijkstra once declared: "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence." [3]

Also, the "Hello World" program for COBOL is considered non-compliant.[1]

2.2 (Inexplicably Recent) Prior Work on COBOL

Unfortunately for Dijkstra, COBOL is still lurking in academic conferences and college curriculums (These citations are all real‼! I know. We were also surprised!). Research papers on COBOL in the 21st century cover aspect oriented programming [9, 15] and a (2017‼!) study of code clones [6]. Professors disagreeing with Dijkstra, or openly admitting to torturing their students, have published

*Chris Timperley, Deby Katz, Zack Coker, Rijnard van Tonder, Mauricio Soto, Afsoon Afzal, Cody Kinneer, Jeremy Lacomis, and Claire Le Goues
†Adam Brady

SIGBOVIK'18, Pittsburgh, PA USA

DOI: 10.475/123_4

[1] The `DISPLAY` statement (i.e., `print`) is considered a dangerous pattern according to a COBOL static analyzer: https://rules.sonarsource.com/cobol/RSPEC-1279

```
1   IDENTIFICATION DIVISION.
2   PROGRAM-ID. ZUNEBUG.
3
4   DATA DIVISION.
5   WORKING-STORAGE SECTION.
6   01 WS-YEAR PIC 9(9) VALUE 1980.
7   01 WS-DAYS PIC 9(9).
8   01 WS-Q PIC 9(9).
9   01 WS-R1 PIC 9(9).
10  01 WS-R2 PIC 9(9).
11  01 WS-R3 PIC 9(9).
12
13  PROCEDURE DIVISION.
14  MAIN-PARAGRAPH.
15      DISPLAY 1 UPON ARGUMENT-NUMBER.
16      ACCEPT WS-DAYS FROM ARGUMENT-VALUE.
17      PERFORM WITH TEST BEFORE UNTIL WS-DAYS <= 365
18  *       LEAP YEAR COMPUTATION
19          DIVIDE WS-YEAR BY 4 GIVING WS-Q REMAINDER WS-R1
20          DIVIDE WS-YEAR BY 100 GIVING WS-Q REMAINDER WS-R2
21          DIVIDE WS-YEAR BY 400 GIVING WS-Q REMAINDER WS-R3
22          IF (WS-R1 = 0 AND WS-R2 NOT = 0) OR WS-R3 = 0 THEN
23              IF WS-DAYS IS GREATER THAN 366 THEN
24                  SUBTRACT 366 FROM WS-DAYS
25                  ADD 1 TO WS-YEAR
26              END-IF
27          ELSE
28              SUBTRACT 365 FROM WS-DAYS
29              ADD 1 TO WS-YEAR
30          END-IF
31      END-PERFORM.
32      DISPLAY WS-YEAR.
33      STOP RUN.
```

(a) Buggy code: An infinite loop happens when `WS-YEAR` is a leap year, and `WS-DAYS` is less than 366.

```
1   IDENTIFICATION DIVISION.
2   PROGRAM-ID. ZUNEBUG.
3
4   DATA DIVISION.
5   WORKING-STORAGE SECTION.
6   01 WS-YEAR PIC 9(9) VALUE 1980.
7   01 WS-DAYS PIC 9(9).
8   01 WS-Q PIC 9(9).
9   01 WS-R1 PIC 9(9).
10  01 WS-R2 PIC 9(9).
11  01 WS-R3 PIC 9(9).
12
13  PROCEDURE DIVISION.
14  MAIN-PARAGRAPH.
15      DISPLAY 1 UPON ARGUMENT-NUMBER.
16      ACCEPT WS-DAYS FROM ARGUMENT-VALUE.
17      PERFORM WITH TEST BEFORE UNTIL WS-DAYS <= 365
18  *       LEAP YEAR COMPUTATION
19          DIVIDE WS-YEAR BY 4 GIVING WS-Q REMAINDER WS-R1
20          DIVIDE WS-YEAR BY 100 GIVING WS-Q REMAINDER WS-R2
21          DIVIDE WS-YEAR BY 400 GIVING WS-Q REMAINDER WS-R3
22          IF (WS-R1 = 0 AND WS-R2 NOT = 0) OR WS-R3 = 0 THEN
23                  SUBTRACT 366 FROM WS-DAYS
24              IF WS-DAYS IS GREATER THAN 366 THEN
25                  ADD 1 TO WS-YEAR
26              END-IF
27          ELSE
28              SUBTRACT 365 FROM WS-DAYS
29              ADD 1 TO WS-YEAR
30          END-IF
31      END-PERFORM.
32      DISPLAY WS-YEAR.
33      STOP RUN.
```

(b) Zune bug, repaired by COBOLd (yes, seriously). Lines 23 and 24 are swapped in the original source

Figure 1: COBOL implementation of the infamous Zune bug and repaired program generated by COBOLd. First six columns omitted to fit on one page.

the colorfully titled papers "Teaching a language many CS students didn't want to learn: COBOL" [7] and "Bringing COBOL back to the college IT curriculum" [12]. Some of our favorite excepts from these papers' abstracts include *"As shocking as this may sound to some, there are many reasons beyond great need to bring COBOL back into the CIS curriculum"* [12] and *"Over 80 percent of error diagnosis was found to be inaccurate. Such feedback is not optimal for users, particularly for the learning user of Cobol."* [13].

As far as static analysis tools, SonarQube[2] provides static analysis tools for COBOL. However, SonarQube can't automatically fix COBOL programs. But we can. So watch this and don't blink.

3 APR FOR COBOL

3.1 Generate-and-validate.

One of primary classes of approaches for APR is known as *heuristic* or *generate-and-validate repair* (G&V), which uses search-based software engineering techniques [5, 16] to generate many candidate patches for a bug, and then validate them using indicative workloads or test suites.

Algorithm 1 illustrates our novel G&V approach for fixing COBOL programs. The COBOLd function takes as input a textual COBOL program p. State of the art G&V APR techniques typically operate on the abstract syntax tree (AST) of a program, rather than the program text itself. This helps limit the generation stage to syntactically valid programs, and allows more advanced reasoning about program structure. Though this works for reasonable programming languages, COBOL is not specified with a real grammar (cf. Section 2.1.2), and the creation of a parser would require for-real work. So, the COBOLd function invokes GenProg [11] to modify the program text of p directly. This is particularly fortunate because we would hate to implement a parser for a language with *550 reserved words, are you #kidding me?*[3]

Algorithm 1 A simplified G&V algorithm for automated COBOL repair.

```
1: function COBOLd(p)
2:     cands ← GenProg.getInitialPopulation(p)
3:     while !oracle(cands) do
4:         cands ← GenProg.getNextPopulation(cands)
5:     end while
6:     ??????????????????????????????
7:     PROFIT!
8: end function
```

The brunt of the work is done by GenProg, which produces candidate programs *cands* in the initial state (Line 1) and performs

[2] https://www.sonarqube.org/

[3] http://www.math-cs.gordon.edu/courses/cs323/COBOL/COBOL-reservedWords.html

genetic mutation to generate additional candidates (Line 4) until the oracle condition is satisfied (Line 3). The validation function ORACLE runs the candidate COBOL program on test inputs. If the program produces the expected output, ORACLE returns 1 and we end up with a 100% absolutely correct^{TM} COBOL program. Otherwise, we keep looping until we find a candidate satisfying ORACLE.

3.2 Mutation operators.

COBOLd currently modifies COBOL code by applying one of three different *mutation operators*: *delete*, which deletes a line of code, *add*, which selects a random line of code and inserts a copy of it at a random program point, and *swap*, which exchanges two lines of code. In our exploratory study, these operators were sufficient to generate patches. Future versions of COBOLd will include more COBOL-specific mutation operators such as *delete-a-full-stop*, which randomly deletes a ".", and adaptations, such as only mutating code inside of a procedure.

4 SERIOUSLY: HERE'S AN EXAMPLE

Figure 1 shows a COBOL implementation of the infamous Zune bug [1]. The Microsoft Zune was a popular portable music player that stored the current time as the number of days and seconds since 1 January, 1980. The (buggy) function in Figure 1a was used to compute the current year. An infinite loop occurs when `WS-YEAR` corresponds to a leap year, and `WS-DAYS` is less than 366. In this case, the `SUBTRACT` statement on line 24 is never executed, so the value of `WS-DAYS` is never changed. COBOLd is able to successfully repair this bug, producing the code shown in Figure 1b. To produce this patch, COBOLd swaps the `IF` statement on line 23 and the `SUBTRACT` statement on line 24.

5 WE ARE GOING TO BE VERY, VERY RICH

5.1 No, Really

According to the Cobol Cowboys[4], COBOL is 65% of active code used today, runs 85% of all business transactions. IBM cites that 200 BILLION lines of COBOL code is still in use today.[5] All of Google's code comes in at about 1 percent of this number, at a mere 2 billion lines of code.[6] This is music to our ears. Rate of adoption for our COBOL repair tool will strictly increase with existing and proliferating COBOL code. Figure 2 conservatively projects our income.

5.2 Treat. Yo. Self

We prepared a shortlist of things to treat ourselves with.

- Clothes
- fragrances
- massages
- mimosas
- fine leather goods
- supercars

[4]http://cobolcowboys.com/
[5]Not a joke: https://www-03.ibm.com/press/us/en/pressrelease/41095.wss
[6]Not a joke: https://informationisbeautiful.net/visualizations/million-lines-of-code/

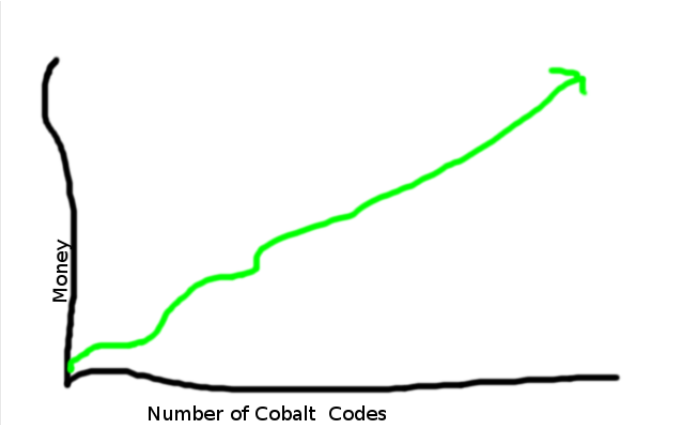

Figure 2: COBOL Repair is going to make us rich.

👀 👌 👀 👌 💯 if i do 5a*y* so my sel f 💯 i say so 💯 thats what im talking about right there right there (chorus: ʳᵢght tₕeᵣe) mMMMMMM 💯 👌 👌

It's a modest start, we're new to this. We realize our enormous purchasing power is likely to raise difficult questions. Like, we want private islands too,[7] but is Australia an island? Is it for sale?

We do plan to use very smol portion of our profits[8] for the greater good. Our #1 priority is to eradicate video and audio autoplay on all websites. We are willing to pay good money to remove the `autoplay` attribute from the HTML spec, please get in touch if you know someone. We also plan to resurrect geocities sites. We really miss 88x31 buttons. Click the one below to get the COBOLd demo (or Netscape Now! we're not really sure).

ACKNOWLEDGMENTS

We would like to thank Nalia Soto Gutierrez (Figure 3); Ryan, Kyle (Figure 6), Kevin (Figure 5), and Layla Lacomis (Figure 7); Aries (Figure 12), Cheese (Figure 10), Snickers (Figure 4), and Ampersand LeBrady (Figure 11); Penelope Surden (Figure 8); and Millie Timperley (Figure 9) for all of their love and support, without them this research would not be possible.

We'd also like to more seriously thank Grace Hopper for her work in compilers, for guiding the development of machine-independent programming languages like COBOL, and for being an awesome woman in science. And the ENIAC programmers, who do not get enough recognition.

REFERENCES

[1] BBC News. 2008. Microsoft Zune affected by 'bug'. In *http://news.bbc.co.uk/2/hi/technology/7806683.stm.*
[2] Bob Bemer. 1971. A View of the History of COBOL. *Honeywell Computer Journal* (1971).
[3] Edsger W. Dijkstra. 1975. How do we tell truths that might hurt? (June 1975). published as EWD:EWD498pub.

[7]https://www.privateislandsonline.com
[8]which is still going to be like, hundreds of millions

[4] Stephanie Forrest. 1993. Genetic Algorithms: Principles of Natural Selection Applied to Computation. *Science* 261 (Aug. 1993), 872–878.

[5] Mark Harman. 2007. The Current State and Future of Search Based Software Engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 342–357. https://doi.org/10.1109/FOSE.2007.29

[6] Tomomi Hatano and Akihiko Matsuo. 2017. Removing Code Clones from Industrial Systems Using Compiler Directives. In *International Conference on Program Comprehension (ICPC '17)*. 336–345.

[7] Stephen M. Jodis. 1995. Experiences and Successes in Teaching a Language Many CS Students Didn't Want to Learn: COBOL. In *Southeast Regional Conference (ACM-SE 33)*. 273–274.

[8] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

[9] Ralf Lämmel and Kris De Schutter. 2005. What Does Aspect-oriented Programming Mean to Cobol?. In *International Conference on Aspect-oriented Software Development (AOSD '05)*. 99–110.

[10] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *AMC/IEEE International Conference on Software Engineering (ICSE)*. Zurich, Switzerland, 3–13.

[11] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering (TSE)* 38 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[12] Ed Lindoo. 2014. Bringing COBOL Back into the College IT Curriculum. *Journal of Computing Sciences in Colleges* 30, 2 (Dec. 2014), 60–66.

[13] Charles R. Litecky and Gordon B. Davis. 1976. A Study of Errors, Error-proneness, and Error Diagnosis in Cobol. *Commun. ACM* 19, 1 (Jan. 1976), 33–38.

[14] Schneiderman. 1985. The Relationship Between COBOL and Computer Science. *Annals of the History of Computing* 7, 4 (1985), 348–352.

[15] Hideaki Shinomi and Yasuhisa Ichimori. 2010. Program Analysis Environment for Writing COBOL Aspects. In *International Conference on Aspect-Oriented Software Development (AOSD '10)*. 222–230.

[16] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *International Conference on Software Engineering (ICSE)*. Florence, Italy.

[17] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. 2010. Automatic program repair with evolutionary computation. *Communications of the ACM Research Highlight* 53, 5 (May 2010), 109–116.

[18] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. Vancouver, BC, Canada, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

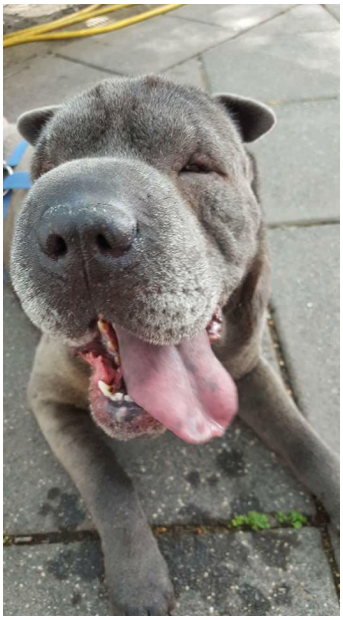

Figure 3: Naila: Was a cat in her previous life. Also, loves pizza more than life itself

Figure 4: Snickers: THE PRETTIEST KITTY

Figure 5: Kevin: Kind of an idiot

Figure 6: Kyle and Ryan: Soft pile

Figure 7: Layla: *Actually* the prettiest kitty

Figure 8: Madam Penelope, Queen of the Apartment

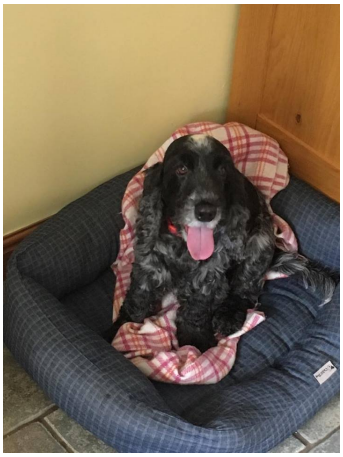

Figure 9: Millie: Stealer of food and hearts

Figure 10: Cheese: SPIRIT OF A CHAMPION

Figure 11: Ampersand: Stone Cold Killer

Figure 12: Aries: Is a bird