Automatic Program Repair for Extended While3Addr

Benedict Ozua

https://github.com/BozeBro/Program-Repair

1 Abstract

The While3Addr (W3A) is a very tedious programming language with very strict semantics on how to write programs. It's an even greater problem to debug any existing W3A code. This paper delves into the process of automating the W3A debugging process, using the SemFix Paper as inspiration for implementation. Creating Automatic Repair is also an interesting topic to study as it solves a tedious issue, while encompassing different ideas in Program Analysis, namely path constraint generation and Program Synthesis. This version of program repair aims to fix singular assignment lines in a .w3a file, and replace values in assignments with new expressions with the help of the SMT solver, Z3. Different from SemFix, we will use Concolic Execution to generate path conditions to pass to a constraint solver rather than pure Symbolic Execution. This paper will also walk through the steps of implementation details of repair, new extended features, and overall structure of implementation code.

2 Background

The While programming language is a simple toy language originally used for discussing and proving concepts about code. It has basic features such as while loops, assignments, operations, and expressions that consist of one or more arithmetic operators. The While3Addr (W3A) programming language is a simpler version of the While language that makes it simpler to prove facts and build on top of other program analysis ideas. This language is restricted in that there are no while loops and 1 use of arithmetic opertaions are allowed per line. For example, in the While language, we are allowed to express $1 + 2 + 3 + x$ in a single line, but W3A would require the programmer to write each addition on a separate line. The simpleness of this language's semantics and features makes automatic program repair simple to implement. The current interpreter for both W3A has been extended with array types and basic array functions to manipulate arrays. These semantics include updating arrays, indexing into arrays, and creating arrays. The compiler to compile from While into W3A has also been extended to allow us to write buggy programs in While and compile down to W3A since manually writing W3A code can be a time consuming process.

3 Program Repair

Automatic Program Repair is the process of programmatically fixing bugs in a program using techniques similar to how a real person would debug a program. A bug is classified as an expression in a file that causes a test in a test suite to fail rather than pass. A program fix will find this expression and change it to a new expression that causes the program to pass all test cases. Repair, generally, has a three step process: fault localization, generating candidate programs, and verifying the candidate program. In fault localization, repair will rank lines by suspiciousness of causing a bug. The SemFix paper uses a formula heuristic to calculate these lines given here.

$$susp(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

s is a statement in a file, $failed(s)$ is the number of times s is executed in all failing tests, and $passed(s)$ is the number of times s is executed in all passing tests. To generate candiate repairs, we use a technique in program repair called constraint-based program repair. This technique generates constraints that need to be true for a statement s in each test in the test suite. For example in this example W3A program

```
1: num := input 0
2: a := num / 2
3: b := a * 2
4: c := expression
5: if c = 0 then res := 1 else res := 0
6: print res
```

we have a simple program for seeing if an input to the file is either even or odd, in which the program prints 1 if even and 0 if odd. If we wanted to make constraints for the assignment to replace expression in line number 4, the constraints would say that c is 0 when num is even, and nonzero otherwise. Note, that integer division discards any remainder on integers. With these constraints, the repair will pass them to a constraint solver, like z3, and give us a new expression if the expression is satisfiable. If the expression is satisfiable, we can replace expression with a better expression. Afterwards, verifying a candidate program involves rerunning the test inputs, and see if the desired outputs are reached. If all the tests pass, the repair stop, otherwise, the repair goes to the next line in the suspiciousness ranking and reruns repair again.

4 Generating Path Constraints

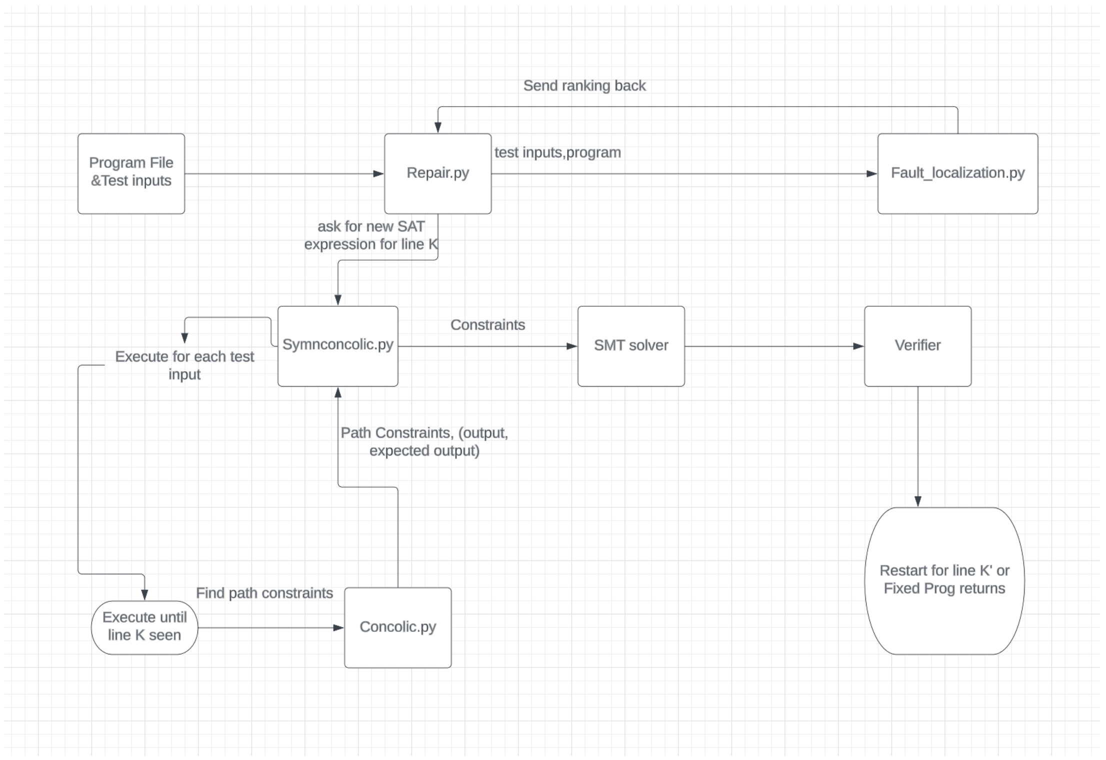
To generate path constraints on some test inputs, while evaluating a new expression for a suspiscious line, the repair will evaluate a program concretely until the suspicious line number is reached. Here, the SemFix paper and my project implementation make a slight deviation for the remaining of generating path constraints. SemFix will use symbolic execution, executing with symbolic expressions, until an output is reached. My project uses concolic execution, while having assignment on the suspicous line map to a symbolic value.

$$C_i := (\bigvee_{j=1}^{m} (pc_j \wedge O_j == O(t_i))) \wedge (f(\xi_i) == \tau_i)$$

The image above from the SemFix paper shows a formal specification of how the constraint will look like for an individual test input. pc_j is the path condition for one of m paths. O_j is the output of a path at the end of execution. $O(t_i)$ is the desired output. the f is a function that takes in the current environment and output a value that will replace value expression on a line number.

Symbolic execution is not directly implemented in W3A, and a concolic executor has been made for a past homework assignment in the 17355 class. In additional, modifying the concolic executor executor to allow arrays and different modifications seemed to require less time to implement than figuring a solution for pure symbolic execution for W3A. The drawback to concolic execution for generating path constraints is that in while loops, some imputs that we use may cause infinite loops to occur. Thus, we restrict for instructs to be executed for some amount of time X. Afterwards, we will cut off execution, and say the repair produced some output $None$

5 Project Structure


Send ranking back
Program File &Test inputs
Repair.py
test inputs,program
Fault_localization.py
ask for new SAT expression for line K
Symnconcolic.py
Constraints
SMT solver
Verifier
Execute for each test input
Path Constraints, (output, expected output)
Restart for line K' or Fixed Prog returns
Find path constraints
Execute until line K seen
Concolic.py

The files used for interpreting, compilg, and repairing are located within the *src* folder. This folder contains the 3 key folders, while, repair, and while3addr. The while, and while3addr folders contains code for interpreting .while and .w3a programs using Ocaml to run the interpreter. The repair module is located in the repair folder. The repair module contains four key files that implement the 3 phases of constraint-based program repair. *repair.py* is the entrypoint into the repair module. It requires 2 commandline arguements, the testfile, and the path/name of the new file if a succesful program. The program will call functions in the $fault\_localization.py$ file to rank the executed lines by suspiciousness. Note, we throw away lines that are not assignment statements, since we only focus on fixing values in assignments. Afterwards, We iterate through each line i in the ranking, and call a $findlines()$ located in the *symconcolic.py* file to generate a candidate program. *symconcolic* will execute each test until we see i, where we will execute concolic execution and generate path constraints. With these path constraints, we assert that the output of the file is equal to the expected value. We do this for reach test input in the Test suite, and $\wedge$ all the constraints together. Afterwards, we pass this large constraint into a z3 SMT solver, and have it synthesize an expression that supposdely passes all the tests. Because we have mixed concrete execution into our path finding, we may run into false positives that force us to check if the generated expression is actually correct. We return the program to *repair*, if upon verification, all the tests pass, we return, otherwise, we retry again. The image above shows how this process looks on the high level visually.

6 Extending the Interpreters and Compiler

The interpreter for While and W3A along with the compiler to compile While to W3A exist in the past homework 3 github. The project modifies these tools to allow for arrays and compile the extended While language into the extended W3A language. Both of these modules are written in Ocaml. To get information about a W3A program for our python repair moduel, I've implemented a second interpreter for W3A written in python that can also supply instructions at a line in a program to allow for pythonic concolic execution. The Ocaml interpreters and compiler are only used for compiling While programs into W3A code. The python interpreter is used for line information for the repair module while also being the main interpreter for running W3A programs with inputs. The Ocaml interpreters, can run the same semantics as python except it does not implement inputs to the file. Inputs must be hardcoded into the file.

7 Reproduction Steps

The code can be located here. The simplest way to obtain the code is to use the command 'git clone https://github.com/BozeBro/Program-Repair ProgRepair'. change directory into ProgRepair To get the environment running for Ocaml, the project requires opam, core, dune, merlin, ounit2, libraries to be installed. See Opam Installation for installation guide.

```
opam init
opam switch create homework 4.14.0
eval $(opam env)$
opam install core dune merlin ounit2
```

This will activate the opam environment and install the rest of the library dependencies needed for the project. For the python environment, we need z3 to be installed, and we need the python version to be at least 3.10. The way this project goes about installing the z3 package is through the builtin environment venv. Run 'python -m venv env' to create the virtual environment named env. Afterwards activate with source ‘env/bin/activate’ if on Unix machines or ’env\Scripts\activate.bat’ on Windows. See venv instructions for more guidance. After activating the environment, use ‘pip install z3-solver’ to install z3 for python3. This is an example Reproduction steps for Mac/Linux users

```
python3 -m venv env
source env/bin/activate
pip3 install z3-solver
```

Afterwards, run make to generate the compiler, Ocaml while interpreter, and the Ocaml W3A interpreter.

To write a test file suite along with the file we want to fix, create a file with the following specification. The first line needs to be the path of the buggy program that we want to repair. Every line afterwards follows the format inputs : output. This assumes that each w3a program only has 1 output and possibly many inputs. Inputs are space separated. To create array inputs and outputs, it is the same as python list semantics but remove any spaces within the list [1,2,3,4] is a valid list input but [1, 2, 3, 4] is an invalid array input. 5 [1,2,3,4] is would be a list of two arguements. Create a text file named test.txt file and put the following text

buggy_progs/even.w3a
10 : 1
11 : 0
12 : 1
100 : 1
1 : 0

This text file contains a test suite for a even W3A program. From the test cases, can we guess that the programmer wants to print 1 whenever the input is even, and 0 if it is odd. If you take a look at the even.w3a code, you’ll notice that it will always print 1 regardless of the input, and that is the bug that we want to fix.

To run a program, use the interpreter.py file in src/repair. We can invoke it via‘ python3 src/repair/interpreter.py file argList’ where file is the path to the W3A program and args either takes an 1 arguement for an int argument or a space separated list of integers for an array arguement. To see what even.w3a outputs on some int input, run ‘python3 src/repair/interpreter.py buggy_progs/even.w3a 1’ to have 1 as an input.

To compile a while program into a W3A program, use the commandline arguments ./compile < path/to/whileProgram > path/to/new/w3aProgram.

To run a .while program, use the semantics ./while < path/to/whileprogram. Note that ./while binary does not implement any semantics for inputs to a file, so we cannot run the instruction input i in Ocaml interpreter binaries.

To generate a fixed program for the buggy even program, run the command

'python3 src/repair/repair.py buggy_progs/even.w3a fixed.w3a' to generate a fixed even function called fixed.w3a.

To try it for yourself, run 'python3 src/repair/interpreter.py fixed.w3a 11' to see that the output is 0. You can also try it on the other outputs in the test suite to see that all the outputs are expected.

8 Limitations

Some issues arise in dealing with while loops. Since we are running concollicaly, some test inputs that we use to go to a new path may cause infinite loops to occur. To handle these, we put an upper bound on the number of times an input can execute. This upper bound will lead to some input files to be unable to be solved by this project. Additionally, there is currently still buggy behavior around asserting constraints with arrays. This problem is due to the fact that array issues sometimes need more than one line changed to be fixed, and since we cannot change update statements in this project since it is not an assignment expression, it leads to limiations on actually fixing it. Additionally, we only try to change 1 line instead of inserting lines or changing multiple lines. This reason is due to scalability, and small programs can even cause infinite loops to occur, which proved to be a hard problem to get around.