EIE4001 Software Engineering

Project Report

Name: Zhang Yixuan

Student ID: 120010058

Date: Apr. 15th

The Chinese University of Hong Kong, Shenzhen

① Differential Testing

1. Differential Testing

Overview. To find bugs in a buggy PIG-interpreter through differential testing [1, 2], you'll need to write a generator for the PIG language and a correct version of PIG-interpreter. You're required to submit two Python files for this task: "gen.py" (your generator) and "pig.py" (your interpreter). For each buggy interpreter, we'll repeat the testing process with 100 iterations:

(1) Run your generator "gen.py", which should save the program output to a file named "input.pig" under the same directory.

(2) Use "input.pig" as input and run your interpreter "pig.py", then save the output to a file named "1.out" under the same directory.

(3) Use "input.pig" as input again, this time with the buggy PIG interpreter, and save the output to a file named "2.out" under the same directory.

(4) Compare "1.out" and "2.out". Any inconsistency between them will serve as a bug report for your testing.

1.1 Code logic

1.1.1 gen.py:

Variable pre-defined:

var_nums: a int variable randomly selected between 40 and 100, meaning the number of variables that should be defined.

list_var_bit: a list to store the number of bits for each defined variable. Index corresponds to the variable name. For removed variables, bit data does not become 0 but will be updated to a newly endowed bit data.

list_var_name: a list to store the variable names.

line_count: an int type variable to store the amount of lines printed.

cycle: an int type variable to control steady printing of instructions.

deleted_var: a list to store the removed variables. Once a variable is redefined, it will be removed from the list.

Function defined:

find_exist_var(deleted_var): the function is used to generate a existed variable for an expression to use. There is a while loop that repeatedly generate a random variable and once the variable is not in **deleted_var**, the loop will break and return the variable name.

gen_line_D(var_name, var_types): the function is to generate the Define instructions. A random variable type is generated and it will be combined with 'D' and variable name as input to get the instruction. The instruction and bit data of the defined variable will be returned.

gen_exp(list_var_bit, iteration_time, deleted_var): the function is to generate expression using iteration. A variable **exp_type** will be used to decide which kind of expression it is. If an **iteration_time** reaches 3, the expression type will be forced to be 0 or 1, that is **LP CONSTANT RP** or **LP VAR RP**, otherwise, the type can be **LP EXP BOP EXP RP** or **LP NOT EXP RP**. In each case, expression is constructed according to generated constant values, variable names, and sub-expressions. When generating constant, I first choose a length (8, 16, 32, 64) and randomly generate all "0" except 1 "1" or all "1" except 1 "0" string as value. If directly generate number, there might be some problems in branching. While generating variable names, a existed variable will be generated. For the following conditions containing sub-expression, iteration is utilized with **iteration_time** +1. Operation type except NOT is randomly generated. Expression is combined based on correct grammar rules.

gen_line_A(var_name, exp): the function is to generate the Assign instruction according to variable name and generated expression.

gen_line_B(rand_line, exp): the function is to generate the Branch instruction according to randomly selected line within given range and generated expression.

gen_line_R(var_name): the function is to generate Remove instruction according to randomly selected variable name that is not in **deleted_var**.

gen_line_O(var_name): the function is to generate Output instruction according to variable name as input.

Main part:

The main part contains a while loop that regularly checks printed lines' number and print instructions in each cycle. Cycle count starts from 0, meaning that the first cycle is when **cycle = 0**. The limit, **line_count**, is set to be smaller than 998 because in the last cycle, an A instruction and an O instruction will be printed, leading to 2 lines.

In the beginning, several variables are defined. For cycle number fewer than **var_nums**, a D instruction will be printed in each cycle. After receiving the return value of **gen_line_D**, bit and variable name information will be stored in lists.

There is a **if** statement that define **removed** variables back when it is the 41st or the 421st cycle, and when there is deleted variable. There are two conditions to define back because I insert 2 R instructions later.

The following is the generation of an A instruction and an O instruction which will run in every cycle.

For the printing of B instruction, the condition is set to be the 151st or the 401st cycle. When it is the 151st cycle, branching line is a random number from 400 + **var_nums** to 500 + **var_nums**. When it is the 401st cycle, branching line is a random number from 200 + **var_nums** to 300 + **var_nums**. There is also a parameter i which is randomly set to 0 or 1. 0 means that the value of expression is 0 while 1 means that the expression is randomly generated using **gen_exp**. This operation increases the probability of 0 value of expression, avoiding always branching to branch line.

Here I specifically design a part which can positively check the bugs affecting branch instructions which branch for many times and finally not branch. The part happens at the 100th cycle. I first choose an existed variable to get its **var_name** so that its bit number is 8 or 16 which I can easily design special A instruction for it. The 8 or 16 bits ensures better hit rate of bugs. In order to make everything simple, I use simple ADD logic, letting the variable add either '10000000' for 16-bit-variable or '00000001' for 8-bit-variable constantly. The rest choice is to not branch directly, just to ensure a normal operation of following codes, preventing from resulting in too many 'too-many-lines' cases. Two choices are to eliminate the probability of invalid checking. After the A instruction, the value of the variable will be output. The third instruction is to branch to the A instruction by checking the value of the variable. This part successfully check pig3.

The final section is to generate R instructions during the 21st and the 61st cycle. A variable name is selected from existed variables (in **list_var_name** but not in **deleted_var**). Then **gen_line_R** function is used to generate the instructions.

For each **print(stmt, file=f)** operation, **line_count** is updated by adding 1. Cycle updates after each round.

1.1.2 pig.py:

Variable pre-defined:

list_bit: a list variable storing defined var.

list_remove: a list variable storing removed var.

index: an int variable recording the index of current line in input.pig file.

executed_lines: an int variable recording the amount lines executed.

Function defined:

add_binary_str(string1, string2): the function is used to do ADD calculation between two strings. The length of longer string will be recorded for future synchronization. The two strings will be transformed into decimal form to calculate the **sum** and the **sum** is then transformed into binary form using **bin()** function. By contrasting the length of sum string and the length of longer string, '0's are added or some bits are omitted. Result after aligning bits is returned.

sub_binary_str(string1, string2): the function is used to do SUB calculation between two strings. The length of longer string will be recorded for future synchronization. The two strings will

be transformed into decimal form to calculate the **difference**. The subtracted number will add a **2^n** value first if it is smaller than subtraction (**n** should be 8, 16, 32, or 64 according to longer length). The operation is to ensure subtraction between decimal numbers. The **difference** is then transformed into binary form using **bin()** function. By contrasting the length of sum string and the length of longer string, '0's are added or some bits are omitted. Result after aligning bits is returned.

and_binary_str(string1, string2): the function is used to do AND calculation between two strings. The length of longer string will be recorded for future synchronization. The two strings should add '0's before them if their lengths are smaller than longer length. By contrasting two strings bit by bit, an empty string adds '1' only when both bits of strings are '1'. Otherwise, the empty string will add '0'. The result will be returned.

or_binary_str(string1, string2): the function is used to do OR calculation between two strings. The length of longer string will be recorded for future synchronization. The two strings should add '0's before them if their lengths are smaller than longer length. By contrasting two strings bit by bit, an empty string adds '0' only when both bits of strings are '0'. Otherwise, the empty string will add '1'. The result will be returned.

not_binary_str(string1): the function is used to do NOT calculation of a string. The string is read bit by bit. If the bit is '1', an empty string will add '0'. If the bit is '0', the empty string will add '1'. The result will be returned.

search_last(list, element): the function is to search the last '(' in a list. The function will return the index of the last ')'.

exp_cal(split_line, list_bit, var_name, **vars): the function is to calculate the expression in a **Assign** instruction. **bin_exp_cal(split_line, **vars)** will be called for expression calculation and the result will adapt the length of **var**. Result after adaption will be returned.

bin_exp_cal(split_line, **vars): the function is used to calculate the expression of **Branch** instruction without length adaption. Expression and dictionary storing values of vars are used as input. A **list** is created used as a stack. A for loop is used to check every elements in expression. As long as the element is not ')', the element will be appended in the **list**. When the element is ')', elements after the last '(' will be popped out, using to calculate. Since the types of **EXP** in pig file is defined, we can check the number of elements between **LP** and **RP** to do calculation. If the number is 2, it is **NOT** calculation. **not_binary_str(string1)** will be called. If the number is 3, it is the other 4 binary operations where corresponding functions will be called. If the number is 1, the type of **EXP** can be a variable name or simply a value. If it is a variable name (the first character is 'v'), the stored value of the variable will be used. After the calculation of simple **EXP**, we have a binary string as the result. The result will be appended to the list for further calculation. Based on correct grammar, the for loop will end after the last **RP** and a single binary string will be the only element in the **list**. The element will be returned.

Main part:

After several variables defined, a while loop will be used to get information of each line in **input.pig** to execute, ensuring the execution will end after the last line is executed. At the beginning of the while loop, the number of line executed is checked so that the loop breaks after executing 5000 lines.

The first **if** statement checks D instructions. The instruction is first split in list so that the variable names and their values can be extracted to use. The dictionary **vars** will store the initial value 0 according to the name and length. The next check is important because it recognizes the type of D instruction (first define or redefine). For first define, bit information will be directly stored. For redefine, bit information will be updated and the variable will be removed from **list_remove**. Correct grammar ensures correct record.

The second **if** statement checks A instructions. The variable name and expression will be extracted from split line and **exp_cal(split_line, list_bit, var_name, **vars)** is called. The value of the variable will be assigned to **vars**.

The third **if** statement checks B instructions. After calculating the value of expression using **exp_cal(split_line, list_bit, var_name, **vars)**, the value will be checked. If the value does not equal to 0, the **index** will become the index of destination line. Otherwise, the **index** will add 1 to be the index of next line.

The fourth **if** statement checks R instructions. The variable to be removed will be appended to **list_remove**, then its bit information will be changed to 0 to show that the variable does not exist.

The last **if** statement check O instructions. **print(vars[line[2:6]], file=g)** will be directly used to print the variable values stored in dictionary **vars** into file.

After executing each line, **index** will be adjusted (adding 1 or branch) and **executed_lines** will always add 1.

1.2 Test cases

Test times: 100

Buggy Interpreter hit:

Pig1: 100

Pig2: 100

Pig3: 25 (not branch: 54, '10000000': 21)

The result shows that pig1 and pig2 have hit rates of 100% while pig3 has a hit rate of 25%. The miss of checking usually happens when the special part in gen.py has a random number of 1 and use '10000000' in the expression of branch, making it trivial for change in variable value to reach the error operation standard of pig3. Another probability is that the B instruction simply choose to not branch. In general, the code performs well in detecting bugs. The ratio between not branch and branch is about to be 1:1 by random process. The ratio between 16 and 8 bits is about 1:1 as well.

Sample output:

```
csc4001@csc4001:~/Test/DifferentialTesting/Project$ sudo ./pig1 < input.pig > 2.out
csc4001@csc4001:~/Test/DifferentialTesting/Project$ diff 1.out 2.out -b
17c17
< 11111111111111111111111111111111111111111111111111111111111111
---
> 00000000000000000000000000000000000000000000000000000000011111100
21,22c21,22
< 11111111111111111111111111111111111111111111111111111111111111
< 1111111111111101
---
> 00000000000000000000000000000000000000000000000000000000011111100
> 0000000011111010
30c30
< 00000000000000000000000000000000000000000000000011111111111111101
---
> 00000000000000000000000000000000000000000000000000000000011111010
35c35
< 0000000000000000
---
> 0000000000000001
43c43
< 0000000000000000
---
> 0000000000000001
54c54
< 00000000000000000000000000000000
---
```

Fig. 1 Difference of outputs after implementing pig1

```
csc4001@csc4001:~/Test/DifferentialTesting/Project$ sudo ./pig2 < input.pig > 2.out
csc4001@csc4001:~/Test/DifferentialTesting/Project$ diff 1.out 2.out -b
15c15
< 00000011
---
> 00000010
19c19
< 00000001
---
> 00000000
26c26
< 00000000000000000000000000000000000000000000000000000000000000001
---
> 00000000000000000000000000000000000000000000000000000000000000000
30c30
< 00000000000000000000000000000000000000000000000011111111111111101
---
> 00000000000000000000000000000000000000000000000000000000000000000
32,33c32
< 11111111111111111111111111111110
< 0000000000000000
---
> 00000000000000000000000000000000
35a35
> 0000000000000000
39c39
< 00000000
---
> 00000100
```

Fig. 2 Difference of outputs after implementing pig2

```
csc4001@csc4001:~/Test/DifferentialTesting/Project$ sudo ./pig3 < input.pig > 2.out
csc4001@csc4001:~/Test/DifferentialTesting/Project$ diff 1.out 2.out -b
103d102
< 00000000
2472a2472
> 00000000000000000011000000110001
csc4001@csc4001:~/Test/DifferentialTesting/Project$
```

Fig. 3 Difference of outputs after implementing pig3

Testing factors:

Environment: MacOS M1 - UTM - Linux x86_64 (python3.8.10)

Instructions:

(1) python3 gen.py

(2) python3 pig.py

(3) sudo ./pig1 < input.pig > 2.out (use pig1 as example)

(4) diff 1.out 2.out -b

Attention: Since running x86_64 architecture on Arm64 machine is often slower, it usually pause for a moment when doing the first two steps. But everything goes well while running locally.

2. Dataflow

Overview. You are required to implement a Python code “da.py” to count the number of lines that may use undeclared variables in PIG code through dataflow analysis. In particular, you need to treat each branch statement as it may both branch or not branch to the target line number every time it is executed no matter what the expression is.

2.1 Code

2.1.1 da.py:

Variable pre-defined:

list_var: a list to store the variable names.

list_remove: a list to store the deleted variables.

list_index: a list to store the line numbers that represents lines using undeclared variables.

jumped_index: a list to store the branch destination of B instructions.

index: a int value recording the line number

Function defined:

check_exp(list_var, exp): the function is to check whether an expression contains variable names (starts with 'v') that are not defined yet (or have been removed).

jump_binary(index, lines, list_var, list_remove, list_index, jumped_index): the function is to create a fork allowing codes to operate in a branch case. All storing list as perimeters need to be copied to ensure the data changed will not affect the father level. By using while loop checking whether index is smaller than length of lines, pig file is operated line by line. By checking the first char of a line, I split instructions into different conditions. When it is 'D', it's an instruction defining a variable. Redefine operation will be ignored but if the variable has not been defined, it will be added to **list_var** and remove it from **list_remove** if it is in **list_remove**. If the first char is 'A', **check_exp(list_var, exp)** will check the whole line to see if there are some undeclared variables. If there is, the index of the line will be appended to **list_index** if it is not in it. The logic is used in the following part too. For the 'B' condition, things become interest. Since we need to consider both directions, we do not care about the value of expression but we do use **check_exp(list_var, exp)** to find undeclared variables. Again, the index of problematic lines will be appended to **list_index** if it is not in it. I use iteration here to ensure a start point will pass through all forks and reaches the end. The logic is, inputting branch line into iteration and continue to run the line after B instruction, which can be considered as Depth-first Search. The encapsulation and heritage of lists ensures the comprehensiveness of the logic. Next, the 'R' instructions. Here the variable names are extracted directly to see whether they have been removed (we cannot remove again if a variable is already in **list_remove**). No use of undeclared variable will lead to removing from **list_var** and appending to **list_remove**. For the 'O' instructions condition, the logic is the same as that in 'R' part without removing variables.

Main part:

The main part contains a while loop that regularly checks printed lines' number. Each line will be checked and the whole loop without iteration equals to all branch instructions having expression value 0.

The whole logic of code in main part is similar to that in function **jump_binary(index, lines, list_var, list_remove, list_index, jumped_index)**. I repeat the logic to ensure the normal operation of iterations.

The whole code can then detect all forks and pass through all possible route to check lines using undeclared variables. But then I discover that **list_remove** is redundant, so I delete it and reconstruct if statement in checking A instructions.

2.2 Test case

For Test1 to Test6, the running time is always less than 0.1s. The time-costing condition happens when running Test7. At first, the running time was almost out of the range (10s). However, after implementing several strategies to da.py, I successfully reduce the running time of Test7 to around

6.3s. The strategies include updating if statement and reduce redundant data processing that are unnecessary. The last case happens often when checking a line. For example, when checking an A instruction, if it is already recognized as using undeclared variables, it will be passed to next line. Repeatedly checking all lines for 5000 accumulates unnecessary running time.

```
[Running] python3 -u "/Users/viegozhang/Desktop/Testing/DifferentialTesting/Dataflow/da.py"
66

[Done] exited with code=0 in 6.289 seconds

[Running] python3 -u "/Users/viegozhang/Desktop/Testing/DifferentialTesting/Dataflow/da.py"
66

[Done] exited with code=0 in 6.283 seconds

[Running] python3 -u "/Users/viegozhang/Desktop/Testing/DifferentialTesting/Dataflow/da.py"
66

[Done] exited with code=0 in 6.319 seconds
```

Fig. 4 Running time of Test7

After further optimization by giving up **list_remove**, the running time decreases to about 5.9s.

3. Metamorphic Testing

Overview. In contrast to differential testing, we will not implement another version of the target system to detect bugs. Instead, we utilize the relationship between the system outputs of the given inputs related. To utilize metamorphic testing for PIGinterpreters, your task is to implement a generator that can produce two related inputs and a checker for their outputs, which are named “gen_meta.py” and “checker.py” respectively. In this part, we have 4 buggy interpreters. For each buggy interpreter, we’ll repeat the testing process with 100 iterations.

3.1 Code

3.1.1 gen_meta.py

Variable pre-defined:

var_nums: a int variable randomly selected between 40 and 100, meaning the number of variables that should be defined.

list_var_bit: a list to store the number of bits for each defined variable. Index corresponds to the variable name. For removed variables, bit data does not become 0 but will be updated to a newly endowed bit data.

list_var_name: a list to store the variable names.

line_count: an int type variable to store the amount of lines printed.

cycle: an int type variable to control steady printing of instructions.

deleted_var: a list to store the removed variables. Once a variable is redefined, it will be removed from the list.

Functions:

find_exist_var(deleted_var): the function is used to generate a existed variable for an expression to use. There is a while loop that repeatedly generate a random variable and once the variable is not in **deleted_var**, the loop will break and return the variable name.

gen_line_D(var_name, var_types): the function is to generate the Define instructions. A random variable type is generated and it will be combined with 'D' and variable name as input to get the instruction. The instruction and bit data of the defined variable will be returned.

gen_exp(list_var_bit, iteration_time, deleted_var): the function is to generate expression using iteration. A variable **exp_type** will be used to decide which kind of expression it is. If an **iteration_time** reaches 3, the expression type will be forced to be 0 or 1, that is **LP CONSTANT RP** or **LP VAR RP**, otherwise, the type can be **LP EXP BOP EXP RP** or **LP NOT EXP RP**. In each case, expression is constructed according to generated constant values, variable names, and sub-expressions. When generating constant, I first choose a length (8, 16, 32, 64) and randomly generate all "0" except 1 "1" or all "1" except 1 "0" string as value. If directly generate number, there might be some problems in branching. While generating variable names, a existed variable will be generated. For the following conditions containing sub-expression, iteration is utilized with **iteration_time** +1. Operation type except NOT is randomly generated. Expression is combined based on correct grammar rules.

gen_line_A(var_name, exp): the function is to generate the Assign instruction according to variable name and generated expression.

gen_line_R(var_name): the function is to generate Remove instruction according to randomly selected variable name that is not in **deleted_var**.

gen_line_O(var_name): the function is to generate Output instruction according to variable name as input.

Main part:

input.pig1 and **input.pig2** are opened as new file and with "w" operation, each as **f** or **g**.

The main part contains a while loop that regularly checks printed lines' number and print instructions in each cycle. Cycle count starts from 0, meaning that the first cycle is when **cycle = 0**. The limit, **line_count**, is set to be smaller than 998 because in the last cycle, an A instruction and an O instruction will be printed, leading to 2 lines.

In the beginning, several variables are defined. For cycle number fewer than **var_nums**, a D instruction will be printed in each cycle. After receiving the return value of **gen_line_D**, bit and variable name information will be stored in lists.

There is a **if** statement that define **removed** variables back when it is the 41st or the 421st cycle, and when there is deleted variable. There are two conditions to define back because I insert 2 R instructions later.

The following is the generation of an A instruction and an O instruction which will run in every cycle. I defined several functions so that expression we have in **input.pig1** can be used to calculate result directly. The calculated value is used as expression part in **input.pig2**.

The final section is to generate R instructions during the 21st and the 61st cycle. A variable name is selected from existed variables (in **list_var_name** but not in **deleted_var**). Then **gen_line_R** function is used to generate the instructions.

For each **print(stmt, file=f)** and **print(stmt, file=g)** operation, **line_count** is updated by adding 1. Cycle updates after each round.

3.1.2 checker.py

It's quite similar to the file in Example because we maintain the output to be the same theoretically but with different expressions. If there are some bugs, 2.out can be used as correct output.

3.2 Testing

Test time: 10

Correct output (1 in res.out): 10

Output sample:

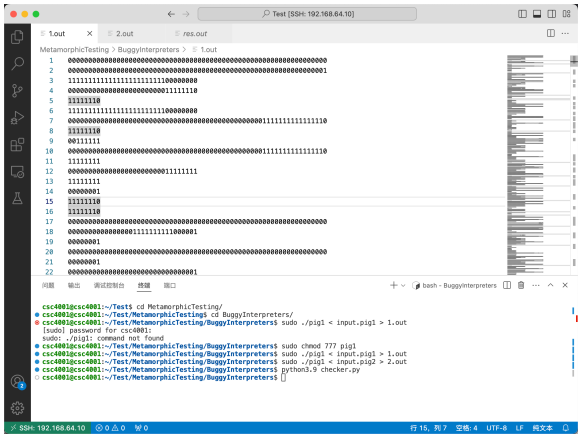

Fig. 5 Sample 1.out in Metamorphic Testing

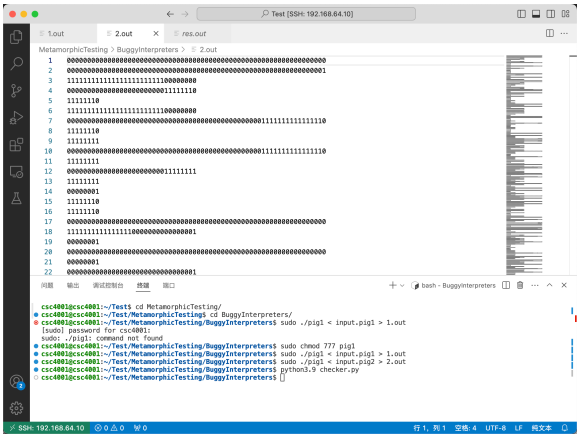

Fig. 6 Sample 2.out in Metamorphic Testing

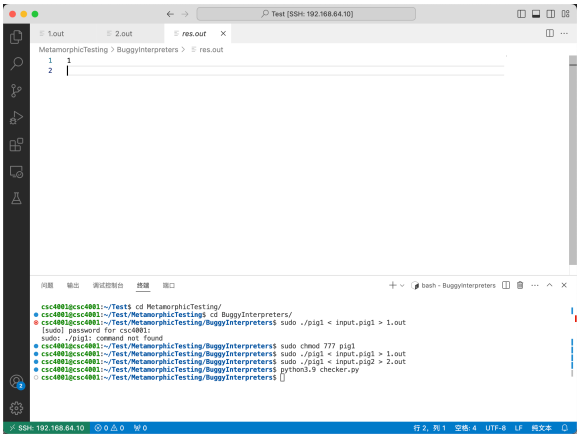

Fig. 7 Sample res.out in Metamorphic Testing

Testing method: (ensuring buggy interpreter executable)

python3.9 gen_meta.py to generate **input.pig1** and **input.pig2**

sudo ./pig1 < input.pig1 > 1.out and **sudo ./pig1 < input.pig1 > 1.out** to generate 2 results **1.out** and **2.out**

Python3.9 checker.py to generate **res.out** containing 0 or 1

Again, running x86 linux system virtual machine on M1 chip Mac make execution of python file slow. Running locally can ensure a running time within1 second.

3 Conclusion

In a word, gen.py and pig.py produce pig file normally without grammar errors. And they can detect abnormal bugs in a comparatively high rate. For da.py, it has a high efficiency to deal with a large number of branch instructions. In Metamorphic Testing, the gen_meta.py ensures the same output but a different process of two input.pig. Since buggy interpreters can only affect the file with complex process, input.pig2 can ensure a correct result and the difference between the result indicate the bugs.