Testing and Dataflow Analysis on PIG Programming Language

Course Project of CSC4001 Software Engineering 2024 Spring

Qingshuo Guo
qingshuoguo@link.cuhk.edu.cn

Qiuyang Mang
qiuyangmang@link.cuhk.edu.cn

1 INTRODUCTION

This project covers two main topics: **(a)** *Testing* (50 pts) and **(b)** *Dataflow Analysis* (50 pts). You are required to implement code (80 pts) and write only **one** report (20 pts) for all parts. We also provide a bonus task (20 pts) for the first topic. You may earn extra points from the bonus task, but your final score won't exceed 100 pts.

2 PRELIMINARIES

In this project, we introduce a simple programming language called *PIG*, which will serve as the foundation for all tasks. It is imperative to thoroughly understand the following instructions regarding the *PIG* programming language before proceeding with the tasks.

2.1 Overview

The *PIG* programs consist of a series of statements, each appearing on a separate line. Each statement is a sequence of tokens, with a space between two adjacent tokens. Except for the blank line at the end of the code, there should be no blank lines in the *PIG* code.

2.1.1 Tokens. To formalize the syntax of the *PIG* language, we define the following tokens in Table 1.

2.1.2 Data Type. The only data type in *PIG* language is bit vector of 4 different lengths ("bv8", "bv16", "bv32" and "bv64"). They are used to simulate unsigned integers of different value ranges ($[0, 2^8 - 1]$, $[0, 2^{16} - 1]$, $[0, 2^{32} - 1]$ and $[0, 2^{64} - 1]$).

2.1.3 Expression. Expressions in the PIG language are defined as sequences of tokens that can be evaluated into a value of a certain type. The formal definition of expressions involves the use of binary operators, denoted by `Bop`, which include `ADD`, `SUB`, `AND`, and `OR` as shown in listing 1. **Please refer to context-free grammar[1] if you are not familiar with the listing here**.

Listing 1: binary operators

```
Bop  →  ADD
     →  SUB
     →  AND
     →  OR
```

Expressions can be defined recursively and may consist of the following elements as shown in listing 2.

- A constant enclosed within parenthesis. The value of this expression corresponds to the value of the constant, with the type being a bit vector of the same length as the constant.
- A variable enclosed within parenthesis. The value and type of the expression are identical to those of the variable.
- Two expressions combined using a binary operator enclosed within parenthesis. The determination of the value and type of this expression will be elaborated later.
- A not operator followed by an expression enclosed within parenthesis. The determination of the value and type of this expression will be elaborated later.

Listing 2: expressions

```
Exp  →  LP  CONSTANT  RP
     →  LP  VAR  RP
     →  LP  Exp  Bop  Exp  RP
     →  LP  NOT  Exp  RP
```

NOT. In the PIG language, the "NOT" operator (`NOT Exp`) performs a bitwise complement operation on the expression `Exp`. The resulting value is the complement of `Exp` with the same data type.
For example:
the value of `NOT ( 11110000 )` is `00001111`.
the value of `NOT ( 1010101010101010 )` is `0101010101010101`.

AND. In the PIG language, the "AND" operator (`Exp AND Exp`) performs a bitwise AND operation on the two expressions `Exp`. If the two expressions have different types, the value of the expression with the shorter bit vector type will be extended to match the length of the longer one, with leading zeros.
For example:
the value of `( 10101010 ) AND ( 11001100 )` is `10001000`.
the value of `( 1111000010101010 ) AND ( 11001100 )` is `0000000010001000`.

OR. In the PIG language, the "OR" operator (`Exp OR Exp`) performs a bitwise OR operation on the two expressions `Exp`. If the two expressions have different types, the value of the expression with the shorter bit vector type will be extended to match the length of the longer one, with leading zeros.
For example:
the value of `( 10101010 ) OR ( 11001100 )` is `11101110`.
the value of `( 1111000010101010 ) OR ( 11001100 )` is `1111000010001000`.

ADD. In the PIG language, the "ADD" operator (`Exp ADD Exp`) performs a bitwise addition operation on the two expressions `Exp`. If the two expressions have different types, the value of the expression with the shorter bit vector type will be extended to match the length of the longer one, with leading zeros. The resulting value of the addition operation is the sum of the two expressions, and any overflowed bit will be omitted.
For example:
the value of `( 01001111 ) ADD ( 01001111 )` is `10011110`.
the value of `( 10000000 ) ADD ( 10000000 )` is `00000000`.
the value of `( 10000000 ) ADD ( 0000000010000000 )` is `0000000100000000`.

[1] https://en.wikipedia.org/wiki/Context-free_grammar

token	regular expression	explanation
D	D	*a single character “D”*
A	A	*a single character “A”*
B	B	*a single character “B”*
O	O	*a single character “O”*
R	R	*a single character “R”*
ADD	+	*a single character “+”*
SUB	-	*a single character “-”*
AND	&	*a single character “&”*
OR	|	*a single character “|”*
NOT	!	*a single character “!”*
LINE	$[0-9]^3$	*a three-digit number*
CONSTANT	$[01]^8\vert[01]^{16}\vert[01]^{32}\vert[01]^{64}$	*a binary string of specified length*
LP	(	*a single character “(”*
RP	)	*a single character “)”*
TYPE	bv(8|16|32|64)	*a string starting with “bv” and followed by a specified number*
VAR	v$[0-9]^3$	a *string starting with “v” and followed by a three-digit number*

Table 1: Tokens of *PIG* language

SUB. In the PIG language, the "SUB" operator (`Exp SUB Exp`) performs a bitwise subtraction operation on the two expressions `Exp`. If the two expressions have different types, the value of the expression with the shorter bit vector type will be extended to match the length of the longer one, with leading zeros. Overflow can occur in subtraction as well. For instance, if a bit vector with all bits set to 0 is subtracted by 1, it will result in a bit vector with all bits set to 1. For example:
the value of `( 01000000 ) SUB ( 00100000 )` is `00100000`.
the value of `( 00000000 ) SUB ( 00000001 )` is `11111111`.
the value of `( 00000000 ) SUB ( 0000000000000001 )` is `1111111111111111`.

2.2 Statement

2.2.1 Variable Declaration. A variable declaration statement in the *PIG* language is used to declare a new variable and initialize it with a value of 0. It follows the syntax:

```
D TYPE VAR
```

This statement will declare a new variable with the name `VAR` of type `TYPE`. In *PIG* language, we only consider variable types for unsigned integers of different bit lengths. In this case, the “bv8", “bv16", “bv32", “bv64", indicate a bit vector of this length to represent an unsigned integer. The variable name **must not** duplicate with an existing variable name.

The following code gives an example variable declaration statement:

```
D bv8 v001
```

This statement will declare a new variable called “v001” of type “bv8”. The value of “v001” will be initialized to 0.

2.2.2 Variable Assignment. A variable assignment statement in the *PIG* language is used to assign a new value to an **existing** variable. It follows the syntax:

```
A VAR Exp
```

This statement will assign `VAR` with the value returned by `Exp`. A variable **must** exist before assigning value to it. If the value is of a longer bit vector type than the variable. The leading bits will be omitted.

The following code gives an example variable assignment statement:

```
A v001 ( 00000101 )
```

This statement assigns variable “v001” with the value of 5.

2.2.3 Branching. A branching statement in the *PIG* language is used to alter the flow of control of *PIG* program. It follows the syntax:

```
B LINE Exp
```

By default, the *PIG* language executes the code line by line. The line index starts from 0. Initially, a program will execute the statement on line 0.

After executing the statement on line i. The program will execute the statement on $i+1$ if line i is not a branch statement (the program will terminate if the statement on line i is the last statement of the code). If line i is a branch statement, the program will evaluate the value of `Exp` and execute the statement on line $i+1$ if the value of `Exp`. If the value of the `Exp` is not 0 code will execute the statement on line `LINE`. Otherwise, it will execute the statement of on line $i+1$. The value of `LINE` should be **within** the maximum line index of the program.

The following code gives an example branch statement:

```
B 001 ( v000 )
```

This statement branches to line 001 if the value of variable “v000” is not 0.

2.2.4 Output. The output statement in the *PIG* language is used to output the value of an **existing** variable to the user. It follows the syntax:

```
O VAR
```

This statement will output the value of VAR as a binary string with its type length.

The following code gives an example output statement:

```
O v001
```

This statement will output the value of “v001”. If v001 is of type “bv8” with a value of 5. The output will be “00000101”.

2.2.5 Remove Variable. The remove variable statement in the *PIG* language is used to destroy an **existing** variable. It follows the syntax:

```
R VAR
```

This statement will destroy the variable VAR.

The following code gives an example remove a variable statement:

```
R v001
```

This statement will destroy the variable “v001”. After executing the statement, the variable “v001” no longer exists. When the program defines the variable “v001” next time, its value will be reinitialized as 0.

2.3 Examples

To help you better understand the *PIG* language, we will provide examples of the *PIG* program and their explanation. These examples can be found in the project file [2].

2.3.1 example I. This example is named “e1.pig”, with corresponding output “e1.out”.

```
D bv8 v000
D bv8 v001
A v000 ( 00000000 )
A v001 ( 00000000 )
A v001 ( ( v001 ) + ( 00000001 ) )
A v000 ( ( v000 ) + ( v001 ) )
B 004 ( ( v001 ) - ( 00001000 ) )
O v000
```

In this code, “v001” is used to iterate from 1 to 8, and “v000” counts the sum from 1 to 8. Hence the value of v000 is 36 and the output should be “00100100”.

2.3.2 example II. This example is named “e2.pig”, with corresponding output “e2.out”.

```
D bv8 v000
O v000
A v000 ( 1111111100000001 )
O v000
R v000
D bv16 v000
A v000 ( 1111111100000001 )
O v000
```

The output for this code is “00000000”, “00000001”, “1111111100000001” respectively.

2.3.3 More examples. You can find more examples in the materials. Due to the length of those examples, the explanations are omitted here.

3 TESTING

In this section, we’re dealing with black-boxed PIG-interpreters that hide bugs. These bugs will even yield incorrect outcomes for input code without syntax and semantic errors. Your task is to employ two testing methodologies in this course, Differential Testing and Metamorphic Testing, to automatically detect bugs within each PIG-interpreter.

Note that, in this part, all interpreters (including yours) will only execute at most 5,000 statements. When trying to run the 5,001-st statement, the interpreter should output a line of “too-many-lines” (without quotes) and stop the program.

3.1 Differential Testing (50 pts)

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.

Note that, your Python code should include the file operations (see the released example).

Requirements. Your generator should output a **non-empty** PIG program with no more than 1,000 lines each time, and there should not be more than 1,000 characters in each line. In addition, the step (1) and step (2) should be finished within 1 second. Meanwhile, we will use a standard interpreter to check whether your produced inputs follow the grammar rules and whether the bug reporting is robust. The generated program should follow the standard grammar and will not cause any crashes for either the standard interpreter or your interpreter. If any such crashes and exceeding time limits occur, we will regard this testing iteration as a failure.

For a successful testing iteration, we will consider its state as one of the following four:

1. **TP**: The result of the buggy interpreter is different from yours and the standard interpreter’s (no matter whether your result is correct). It means you successfully find a bug-inducing test case. Congratulations!
2. **TN**: The result of your interpreter is the same as the buggy interpreter and the standard interpreter.

[2] CSC4001Project/Interpreter/Examples/

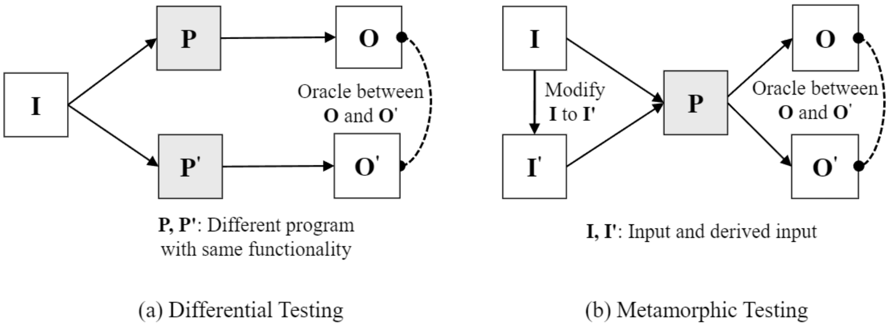

Figure 1: Two testing methods

(3) **FN**: The result of your interpreter is the same as the buggy interpreter but different from the standard interpreter.
(4) **FP**: The result of your interpreter is different from the buggy interpreter but the buggy interpreter's result are the same as the standard result.

Grading. (**40 pts for coding**) We have 8 buggy interpreters in total, where 3 of them are released (without source code) and the remaining are hidden. For each interpreter, the task has **5 pts** in total:

(1) Test case validity (**1 pts**): you will receive (number of successful testing iterations / 100) pts.
(2) Bug detection (**2 pts**): you will receive 2 pts if there is a TP bug report in the 100 testing iterations.
(3) No false alarms (**1 pts**): you will receive (number of successful testing iterations / 100) pt if there is no FP bug report in the 100 testing iterations and there is at least one TP bug report.
(4) No missing bugs (**1 pts**): you will receive (number of successful testing iterations / 100) pt if there is no FN bug report in the 100 testing iterations and there is at least one TP bug report.

Note that, your Python code should avoid importing any third-party libraries, which may cause you to get **0 pts**. We highly recommend you only use "import random" in your code when importing libraries. In addition, any disallowed file operations may also lead you to get **0 pts**.
(**10 pts for report**) You will receive full grades for the report as long as you demonstrate comprehension of the core concept of differential testing and your code. So do not worry about it.

Materials. We have released an example generator and interpreter for your reference[3]. The generator can correctly save the output file and generate a PIG program without syntax errors. The interpreter can correctly read and execute the generated program, as well as save the correct output file. However, they are not strong enough to pass the task, you need to implement your versions.
You can also find the 3 released buggy interpreters in the project file[4].

Hints.

(1) A possible way (may not sound) to evaluate the validity of your test cases is by feeding them into released buggy interpreters and seeing whether there are any crashes.
(2) You may evaluate your test case quality by testing the code coverage on your interpreter since the released interpreters are black-boxed.

3.2 Metamorphic Testing (bonus 20 pts)

Overview. In contrast to differential testing [1, 2], 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 [3] for PIG-interpreters, 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.

(1) Run your generator "gen_meta.py", which should save two files named "input1.pig" and "input2.pig".
(2) Use "input1.pig" as input and run the buggy PIG-interpreter, then save the output to a file named "1.out".
(3) Use "input2.pig" as input and run the buggy PIG-interpreter, then save the output to a file named "2.out".
(4) Run your checker, which should read "1.out" and "2.out" and then write a single integer in $\{0, 1\}$ to the file "res.out", where 1 indicates the buggy-interpreter output at least one incorrect result for the two inputs.

Note that, your Python code should include the file operations (see the released example).

Requirements. Similar to Section 3.1, your generator should output two **non-empty** PIG programs with no more than 1,000 lines each time, and there should not be more than 1,000 characters in each line. The step (1) and step (2) still should be finished within 1 second. We will use a standard interpreter to check whether your produced inputs follow the grammar rules and whether the bug reporting is robust. The generated program should follow the standard grammar and will not cause any crashes for either the standard interpreter or your interpreter. If any such crashes and exceeding time limits occur, we will regard this testing iteration as a failure. In this part,

[3] "CSC4001Project/DifferentialTesting/Examples/"
[4] "CSC4001Project/DifferentialTesting/BuggyInterpreters/"

we only further consider the successful iteration when your checker reports “1”. There are two states:

(1) **TP**: Any of the two results of the buggy interpreter is different from yours and the standard interpreter’s when the checker reports “1”. It means you successfully find a bug-inducing test case. Congratulations!
(2) **FP**: All results of the buggy interpreter are the same as standard’s when the checker reports “1”.

Grading. (**16 pts for coding**) We have 4 buggy interpreters in total, where 1 of them is released (without source code) and the remaining are hidden. For each interpreter, the task has **4 pts** in total:

(1) Test case validity (**1 pts**): you will receive (number of successful testing iterations / 100) pts.
(2) Bug detection (**2 pts**): you will receive 2 pts if there is a TP bug report in the 100 testing iterations.
(3) No false alarms (**1 pts**): you will receive (number of successful testing iterations / 100) pt if there is no FP bug report in the 100 testing iterations there is at least one TP bug report.

Note that, your Python code should avoid importing any third-party libraries, which may cause you to get **0 pts**. We highly recommend you only use “import random” in your code when importing libraries. In addition, any disallowed file operations may also lead you to get **0 pts**.
(**4 pts for report**) You will receive full grades for the report as long as you demonstrate comprehension of the core concept of metamorphic testing and your code. So do not worry about it.

Materials. We have released an example generator and checker for your reference [5]. The generator can correctly generate two PIG programs without syntax errors. The checker can check whether there is a bug by comparing the results of the two generated programs. However, they are not strong enough to pass the task, you need to implement your versions.
You can also find a released buggy interpreter in the project file[6].

Hints.

(1) A possible way (may not sound) to evaluate the validity of your test cases is by feeding them into released buggy interpreters and seeing whether there are any crashes.
(2) Equivalent code rewriting.

4 DATAFLOW ANALYSIS

In this section, we are dealing with some manually-written or machine-generated *PIG* codes. A user may want to know some properties (i.e. use of undefined variables) of the code. However, it may be very costly for the user to execute the code and verify this property. In this case, dataflow analysis, as a static analysis approach, is very helpful.

4.1 Use of Undefined Variable in *PIG* Code (50 pts)

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.

Requirements. Your program should read a *PIG* code from standard input. The input is a string representing a *PIG* code. Each input code contains up to 1,000 lines with each line containing up to 1,000 characters. Besides, we guarantee the syntax correctness of the input code. However, there may exist systematic errors in the input code (i.e. redefining of existing variables or use of undefined variables). The output is simply an integer indicating the number of lines that may use undeclared variables. Your code should output this integer to the standard output.

Examples. To help you better understand this problem, we provide some examples here.
Example A

```
D bv8 v000
A v000 ( v001 )
```

The output for this code should be 1 as line 1 uses an undeclared variable “v001”.
Example B

```
D bv8 v000
A v000 ( ( v001 ) + ( v002 ) )
```

The output for this code should be 1 as line 1 uses undeclared variables “v001” and “v002”.
Example C

```
D bv8 v000
D bv8 v001
A v000 ( v001 )
O v002
```

The output for this code should be 1 as line 3 uses an undeclared variable “v002”.
Example D

```
D bv8 v000
D bv8 v001
R v000
A v000 ( v001 )
```

The output for this code should be 1 as “v000” is destroyed in line 2 and used in line 3 afterward.
Example E

```
D bv8 v000
B 004 ( 00000001 )
A v000 ( v001 )
A v001 ( v000 )
A v000 ( v000 )
```

The output for this code should be 2 as both line 2 and line 3 may use undeclared variable “v001” as we treat each branch statement as it may both branch or not branch to the target line number no matter what the expression is.
Example VI

```
D bv8 v000
D bv8 v001
B 004 ( 00000001 )
R v001
A v000 ( v001 )
```

[5] “CSC4001Project/MetamorphicTesting/Examples/”
[6] “CSC4001Project/MetamorphicTesting/BuggyInterpreters/”

The output for this code should be 1 as when we execute line 4 the variable “v001” may have been destroyed.
Example VII

```
D bv8 v000
D bv8 v001
B 004 ( v002 )
R v001
R v002
A v000 ( v001 )
D bv8 v001
B 002 ( 00000001 )
A v000 ( v001 )
A v000 ( v002 )
```

The output for this code should be 4 (line 2, line 4, line 5, and line 9). We do not care if the code may never halt.

Grading. (**40 pts for coding**) We have 10 manually written or machine-generated *PIG* codes, where 7 of them are released and the remaining are hidden. The hidden test case may have a more complex structure and longer length compared with the released test cases. For each *PIG* code, the task has **4 pts**. We will use automatic tools to test your code. Note that, your Python code should avoid importing any third-party libraries, which may cause you to get **0 pts**. You will receive grades for each test case if your program outputs the correct result. You will not receive grades for a test case if your program:

- Crashes or raises an error
- Have no output or output an incorrect result
- Can not finish within 10 seconds
- Use more than 1GB of memory

(**10 pts for report**) You will receive full grades for the report as long as you demonstrate comprehension of the core concept of dataflow anylsis and your code. So do not worry about it.

Materials. We will release 7 test cases [7].

Hints.

(1) Consider how to abstract an input *PIG* code into a CFG.
(2) Give a proper definition of the IN and OUT state and identify their relationship.
(3) Use the result of dataflow analysis to calculate the required output.

5 SUBMISSION

The deadline for this project is April 28th 23:59. Late submissions will receive a grade of 0 for the whole project. Your submission should include the following 4-6 files named as follows. Please submit them in a single submission without zipping through Blackboard. All code grading will be under Ubuntu20.04 and Python3.9 basic environment.

- “gen.py” your generator to generate *PIG* code for differential testing.
- “pig.py” your interpreter for *PIG* code.
- “da.py” your dataflow analysis tool to check the use of undefined variables in *PIG* code.
- (optional) “gen_meta.py” your generator to generate *PIG* code for metamorphic testing.
- (optional) “checker.py” your checker for metamorphic testing.
- “report.pdf” your report for the whole project.

Please note that, TAs may ask you to explain the meaning of your program, to ensure that the codes are indeed written by yourself. Please also note that we would check whether your program is similar to your peers’ code using plagiarism detectors. If you have any questions regarding this project, please send an email to TAs or the USTFs[8] for this project.

6 TUTORIAL SESSION FOR THIS PROJECT

We will conduct Q&A sessions during the tutorials scheduled for the week of April 8th to April 12th. Please feel free to ask your questions that time.

[7] CSC4001Project/DataflowAnylsis/Pretests

[8] qingshuoguo@link.cuhk.edu.cn and qiuyangmang@link.cuhk.edu.cn

REFERENCES

[1] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. 549–552.

[2] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[3] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.