CSC4001 Project Report

Yueyan WU 121090620

1. Differential Testing

1.1. Differential Testing Overview

Differential Testing is a software testing technique that aims to find errors by comparing differences in behavior between different software implementations or different versions of the same software. This approach is particularly useful for detecting subtle errors that are difficult to detect with traditional testing methods, such as the implementation of compilers, interpreters, and standard libraries. In this testing approach, it is common to run multiple software implementations or versions with the same set of test inputs and compare their outputs, with any inconsistencies likely to indicate a defect in at least one implementation. In this problem, the goal is to find errors in the PIG language interpreter by using differential testing methods.

1.2. Generator

1.2.1 Global variable
I define a global variables “all_length”, to tracks the number of lines of code generated to control the length of the overall code generation.

1.2.2 function
1. random_bits(value_type)
Function: This function generates a random binary string, the length of which is determined by the number of bits specified by the value_type argument. This is useful for creating test inputs or random data values in the PIG language, which is based on bit vectors.
2. random_expression(variables, variable_types, depth, max_depth)
Function: Recursively generate random mathematical or logical expressions. It have ADD, OR, NOT, and so on. It generates random expressions based on variables and their types. It recursively creates complex expressions using both monadic and binary operations, which is crucial for testing how the PIG interpreter handles various computational and logical operations.

These two function used in generate the expressions of A and B.

3. declare(n, var_types, variables_dict, variable_types)
Function: Randomly declare n variables, randomly select the type, and remove these variables from the available variable dictionary. Record each variable name and its corresponding type. It returns a tuple containing two elements, the first being a list of declaration statements and the second being the number of variables actually declared.
4. assign(variables, variable_types)

Function: Generate an assignment statement for randomly selected variables, and the assignment content is a random expression.
5. output(variables)
Function: Generate an output statement for a random variable.
6. remove(variables, variable_types, variables_dict, removed_variables)
Function: Removes variables from the specified variable list and updates the related dictionary.
7. B(actual, n, variables, variable_types)
Function: Generate a jump statement for non-linear jumps of the control flow.

These six function are represented by D,A,O,R. The function is simple. We define them as a function which are convenient to use later.

8. if_block, for_block, basic_block
Function: These functions generate control blocks (if blocks, for loop blocks, basic code blocks) that are suitable for creating complex nested logic.
In if_block, we design a structure that allows Branch comes first, then, follow a simple block that branch could reach any of line in this block to realize if condition.
In for_block, we design a structure that allows a simple block comes first, then, follow a Branch. In the situation, that branch could reach any of line in this block to realize for condition.
Basic block is design for recursion. In this block, it can go to any of other block.

9. generate_program(file_path, max_lines, max_length)
Function: Drive the whole code generation process, manage file output. It call other functions to generate code blocks and write the final result to the specified file.

1.3. Interpreter

This program is used to interpret and execute a PIG language that handles variable declarations, assignments, conditional jumps, output, and variable deletion operations. The program reads an input file (input.pig), executes the instructions therein, and outputs the results to another file (1.out).

1.3.1design analysis

1） File processing: The program reads instructions from the input.pig file and processes them line by line, with the final result written to the 1.out file.
2） Instruction Execution: It supported instructions include variable declaration (D), assignment (A), output (O), conditional jump (B), and variable deletion (R).
3） Control flow management: It use line numbers to control program flow, and support conditional jumps to achieve loops and conditional branches.
4） Variable management: it use dictionary vars to store the value of all current variables, and another dictionary, var_lengths, to store variable bit lengths.
5） Expression processing: it convert complex expressions, such as logical and arithmetic

expressions to postfix expressions which is better to calculate by computer.
6） Error handling: The program handles undefined variables or jumps to illegal line numbers by throwing an exception.
7）Performance monitoring: It set a loop limit (5001 iterations) to avoid infinite loops and output "too-many-lines" if the limit is reached.

2. metamorphic testing

2.1. metamorphic overview

Metamorphic Testing is an effective testing method in software testing, especially for those situations where it is difficult to directly determine the expected results. This approach is based on a core concept, Metamorphic Relations (MR), which are pre-defined rules describing how the output of the program should change after changes are made to the program input.
There are lots of matamorphic relations:
Value flip: For binary values, flip all bits (from 0 to 1, from 1 to 0).
Type extension: Change a variable from a smaller bit-wide type to a larger bit-wide type, such as bv8 to bv16, while leaving the value unchanged or properly populated.
Order change: Changes the order in which variables are declared and assigned.

In my program, I choose the Order change.

2.2. generator

I'll start by putting the generator from the previous problem here. The modified part has defined variables that can no longer be defined. Then, I add a new function.
It first read the original file and store each line of the file into the list lines. Then it initialize class lists and auxiliary dictionaries:
d_statements is used to store all declaration statements starting with'D '.
r_statements are used to store all remove statements starting with 'R '.
other_statements is used to store all statements except declarations and removals.
The branch_updates dictionary is used to record the original line numbers of branch (conditional or loop) statements and their target line numbers.
The original_line dictionary appears to be unused.
It walk through each line of code, sorting them into the corresponding list according to their opening character. For branch statements (starting with 'B '), additionally record their target line numbers into the branch_updates dictionary.
It reorganize the contents of a file in a new order. First all declaration statements, then all other statements, and finally all remove statements. This order may be to ensure that all necessary variables are declared before any operation is performed, and that cleanup or removal operations are performed last.
Next, it mapping original line numbers to new line numbers. Create a dictionary original_to_new_line_numbers to map the line numbers in the original file to the line numbers in the new content list. This is critical for updating the target line number of the

branch statement. And For each branch statement, update its target line number based on its position in the new file content. This ensures that branch statements correctly point to the new target row even after the content is reorganized.
Finally, write to new file.

2.3. Checker

We check whether the two output the same results.

3. Data analysis

3.1. Overview

The dataflow analysis can be used for detecting some features of a code. We use Reaching Definitions Analysis to detect the potential undeclared variables in this program.

3.2 coding

3.2.1 Overall process

1） Input read: Read a series of program instructions from standard input until you encounter an EOF (end of file flag).
2） Initialize definition: Initialize variables according to instructions to define state.
3） Find Leader instructions and build base blocks: Identify the starting instructions (leaders) of the base block and group the instructions into the base block based on those leaders.
4） Build Control Flow Diagram (CFG) : Build CFG based on basic blocks and jump relationships between them.
5） Data flow analysis: Analyzes the program's data flow, calculating the set of entries and exits for each base block.
6） Detecting the use of undeclared variables: counts the number of lines of code that use undeclared variables.
7） Output result: Print the number of lines involving the use of undeclared variables.

3.2.2 function

1.find_leaders:
Function: Find all leader instructions (start instructions of the basic block). The leader includes the first instruction of the program, the jump target instruction, and the next instruction of the jump instruction.
Implementation: Iterate through instructions, check branch ('B') instructions, add jump targets and next instructions to the leader set.
2.build_blocks:
Function: Group instructions into basic blocks based on leader instructions.
Implementation: Iterate instructions, starting a new base block when a leader is encountered, and add subsequent instructions to the current base block until the next

leader.

3.build_cfg:

Function: Build a control flow diagram that records the successors and predecessors of each base block.

Implementation: check the last instruction of each basic block, if it is a branch instruction, then record the jump target; If not, direct the process to the next logical block.

These three function build a complete cfg, which is helpful in the following steps.

4.compute_gen_kill:

Function: Generate (gen) and kill (kill) sets for each base block, but mainly deal with defining status updates in the provided code.

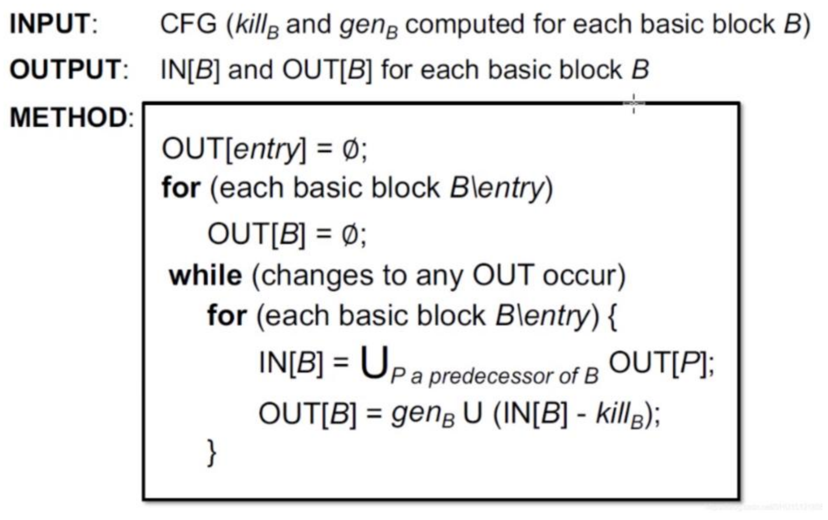
Implementation: Iterate through the instructions of each basic block and update the defined state of local variables.

5.analyze_dataflow:

Function: Perform data flow analysis, calculating the set of entry (IN) and exit (OUT) for each base block.

Implementation: Using the data flow analysis algorithm, the IN set of the current block is calculated according to the OUT set of the previous block, and then the OUT set is updated according to the operation of the current block.

These two function calculate the IN and OUT set, which following the Reaching Definitions Analysis.



INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

```
OUT[entry] = $\emptyset$;
for (each basic block $B \backslash entry$)
    OUT[B] = $\emptyset$;
while (changes to any OUT occur)
    for (each basic block $B \backslash entry$) {
        IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];
        OUT[B] = $gen_B \cup (IN[B] - kill_B)$;
    }
```

6.count_undeclared_variable:

1) To initialize counts and undeclared use collections:

· undefined_use_count is initialized to 0 and is used to count the number of rows that use undeclared variables.

· undeclared_uses is a collection that stores instructions that use undeclared variables and their location.

2) Walk through each basic block:

For each base block, the function first copies the block's entry (IN) set (that is, the declared state of each variable) to local_defs.
3) To process each instruction:
Walk through each instruction in the basic block, processing it differently depending on the instruction type.
4) Update variable declaration status:
For the declare ('D') and remove ('R') directives, update the variable state in local_defs to reflect the variable declaration when the program executes to the current location.
5) Detect undeclared variables using:
If an instruction is found to reference a variable whose status is undeclared (False) in local_defs, that line is added to the undeclared_uses collection and undefined_use_count is added.
Returns the total number of rows that are not declared to be used.