SUTD - 50.053 Software Testing and Verification - Homework 2

Spring 2025

Author

Atul Parida (1006184)

Date of Submission

28th March 2025

Answers for Homework

Just for reference, I've re-written the code for the function required.

Code Outside Box (Human Programmer's Input):

```
int search_key_in_array(int search_list[], int key) {
    int index;
```

Code Inside Box (Code Completion System's Output, which we can't control):

```
    for (index = 0; index < length(search_list); index++) {
    if (search_list[index] == key)
        return index;
   }
   return -1;
}
```

Question 1

a) To test the correctness of the code completion system, we can use some of the metamorphic transformations and corresponding metamorphic relations that we've identified before. 5 of them are as follows:

1. Variable Renaming:
   - Transformation: Change the variable name `search_list` to `list_to_search`.
   - Relation: The output of the function should remain the same, as the variable name does not affect the logic of the code.
2. Control Flow Reordering:
   - Transformation: We can add comments to the code to imply the AI code completion system can reorder the control flow statements. For instance, we can add a comment below `int index;` to say `// Hint: What loop do you use here?`.

- Relation: The output of the function should remain the same, as the logic of the code is not affected by the order of control flow statements, such as using different control flow statements.

3. Input Reordering:
   - Transformation: Change the order of the elements in the `search_list` array passed into the function.
   - Relation: The output of the function should remain the same, as the order of elements in the array does not affect the logic of the code, if the code is generated correctly.
4. Dead Code Insertion into the Prompt:
   - Transformation: Add a line of code that does not affect the logic of the function, such as `int dummy_variable = 0;` before the loop.
   - Relation: The output of the function should remain the same, as the added line of code does not affect the logic of the function. For instance, `dummy_variable` shouldn't be used in the function at all, and it should be removed by the AI code completion system.
5. Type Casting the Prompt:
   - Transformation: Change the type of the `key` variable from `int` to `float` in the function signature, and ensure there's an assertion to check if the `key` is an integer before using it in the comparison.
   - Relation: The output of the function should remain the same, as the logic of the code is not affected by the type of the variable, as long as the AI code completion system handles the type casting correctly. A valid assertion should be added to check if the `key` is an integer before using it in the comparison. For instance, `assert(is_integer(key));` could be added before the loop.

b) Assuming we have a set of seed inputs to test the AI code completion system, we can use the following pseudocode based off of the CGF algorithm to generate the test cases:

```
Input: SeedQ = {prompt1, prompt2, ...}
Output: FailureQ = {}

while (testing_time_remaining):
    t = ChooseNext(SeedQ)
    base_output = AI_completion(t)
    E = AssignEnergy(t)
    for i from 1 to E:
        transformation = RandomChoice({VariableRenaming, ControlFlowReordering, InputReorder
        t_prime = Apply(transformation, t)
        output_prime = AI_completion(t_prime)
        if (base_output != output_prime):
            add t_prime to FailureQ
        else if IsInteresting(t_prime):
            add t_prime to SeedQ
        end if
```

```
    end for
end while
```

For this, the seed inputs needed would be a set of valid code prompts, such as simple function stubs in a format like:

```
int function_name(int arg1, int arg2) {
    // Function body
}
```

The seed inputs should be diverse enough to cover different types of functions, such as sorting algorithms, searching algorithms, and mathematical operations. The AI code completion system should be able to generate correct code completions for these prompts.

First five iterations, for instance, could be as follows: 1. Iteration 1 - Transformation: Variable Renaming, where we get code like:

```
int search_key_in_array(int list_to_search[], int key) {
    int index;
```

```
- Output: `0` (indicating the index of the key in the list)
- Interesting: Yes, as the output is correct and the variable name change does not affect th
- Add to SeedQ: Yes, as it is an interesting transformation.
```

2. Iteration 2
    - Transformation: Control Flow Reordering, where we get code like:

```
int search_key_in_array(int search_list[], int key) {
 int index;
 // Hint: What loop do you use here?
```

    - Output: `0` (indicating the index of the key in the list)
    - Interesting: Yes, as the output is correct and the control flow reordering does not affect the logic of the code.
    - Add to SeedQ: Yes, as it is an interesting transformation.
3. Iteration 3
    - Transformation: Input Reordering, where we get code like:

```
int search_key_in_array(int search_list[], int key) {
 int index;
 // Input list: [3, 1, 2, 4]
```

    - Output: `2` (indicating the index of the key in the reordered list)
    - Interesting: Yes, as the output is correct and the input reordering does not affect the logic of the code.
    - Add to SeedQ: Yes, as it is an interesting transformation.
4. Iteration 4
    - Transformation: Dead Code Insertion, where we get code like:

```
int search_key_in_array(int search_list[], int key) {
 int index;
 int dummy_variable = 0; // This line is dead code
```

    - Output: `0` (indicating the index of the key in the list)

- Interesting: Yes, as the output is correct and the dead code insertion does not affect the logic of the code.
- Add to SeedQ: Yes, as it is an interesting transformation.

5. Iteration 5
   - Transformation: Type Casting, where we get code like:

```
int search_key_in_array(int search_list[], float key) {
 int index;
```

   - Output: `0` (indicating the index of the key in the list, meaning the code's successfully casted the key to an integer)
   - Interesting: Yes, as the output is correct and the type casting does not affect the logic of the code.
   - Add to SeedQ: Yes, as it is an interesting transformation.

c) Assuming we already accounted for robustness testing in the form of smaller changes in input, we can modify our existing pseudocode to find more erroneous outputs, typically due to larger changes in the code or more logical errors. The modified pseudocode would look like this:

```
Input: SeedQ = {valid code prompts}
Output: FailureQ = {}

while (testing_time_remaining):
    t = ChooseNext(SeedQ)
    base_output = AI_completion(t)
    E = AssignEnergy(t)
    for i from 1 to E:
        // Apply an aggressive mutation that performs subtree parsing and structural additio
        t_prime = Apply_BigMutation(t)
        output_prime = AI_completion(t_prime)
        if (base_output != output_prime):
            add t_prime to FailureQ
        else if IsInteresting(t_prime):
            add t_prime to SeedQ
        end if
    end for
end while
```

`Apply_BigMutation` would be a function that performs more aggressive mutations, such as changing the structure of the code, adding new functions, or changing the logic of existing functions metamorphically. This could be done using techniques like EMI parsing, where we apply parsing transformations on the AST level. The pseudocode is as follows:

```
function Apply_BigMutation(prompt):
    // Parse the prompt into an Abstract Syntax Tree (AST)
    ast = ParseToAST(prompt)

    // Select a random subtree from the AST representing a code block or parameter list
```

```
selected_subtree = ChooseRandomSubtree(ast)

// Generate a new subtree using allowed production rules (e.g., subtree addition or repl
new_subtree = GenerateSubtreeUsingGrammar()

// Insert or replace the selected subtree with the new subtree
mutated_ast = ReplaceSubtree(ast, selected_subtree, new_subtree)

// Convert the mutated AST back into source code
return GenerateCode(mutated_ast)
```

Using this, we can find more complex errors, potentially logical errors that the code completion system may not be handling in its outputs.