Analyzing Java Function Output Range: Randomized vs. Concolic Testing

Becker, Paul Nelson
Technical University of Denmark
s194702@student.dtu.dk

Onorato, Andrea
Technical University of Denmark
s233336@student.dtu.dk

Jørgensen, Thor-Haakon Howatson Wichmann
Technical University of Denmark
s194754@student.dtu.dk

Tuda, Enrico
Technical University of Denmark
s232438@student.dtu.dk

***Abstract*—The following implements random testing and concolic analysis to analyze whether a given Java function with an expected output range can return values outside that expected range. The two approaches are tested and compared using multiple example functions. Further loop skipping in the concolic analysis is evaluated in an attempt to accelerate the tool.**

Keywords: Concolic Analysis · Random Testing · Method Verification

I. Introduction

In the realm of software testing and quality assurance, the critical task of ascertaining whether a Java function, equipped with an expected output range, ever produces values that go beyond this predefined output range has drawn our attention. This paper explores this topic using concolic analysis and compares that to random testing.

Sometimes, when engineers write code, they might miss potential errors that can happen when different inputs interact and cause unexpected outcomes. In the subsequent function, there will be highlighted a procedure for determining the efficiency class of a wall based on four input variables. While the programmer conscientiously considered that the quantity, color class, and exposure variables must maintain positive values, they inadvertently omitted the consideration that certain calculations may result in negative efficiency values. The ensuing project is dedicated to the examination of output ranges, specifically focusing on the identification of input combinations capable of producing results outside said output range.

```
public static int calculateEfficiency(int
    quantityOfInsulant, int
    classEfficiency, int colorGrade, int
    exposure) {
   // Ensure the input parameters are
       within valid ranges.
   assert 1 <= colorGrade && colorGrade <=
       10;
   assert 1 <= exposure && exposure <= 10;
   assert quantityOfInsulant > 0;
   assert classEfficiency > 0;
   int efficiency = quantityOfInsulant *
       classEfficiency;
   int correction = Math.min(colorGrade,
       exposure);
   if (efficiency <= 120) {
      // In the first efficiency class [0,
          120], additional correction may
          be required.
      efficiency = efficiency -
          correction; //this return can
          give a negative number
   } else {
      // For higher efficiency levels,
          apply a different correction.
    efficiency = efficiency - correction *
        2; //this return will always give a
        positive number
   }
   return efficiency;
}
```

A **static analysis** approach would not be suitable for this problem. Static analysis focuses on the internal structure of the program, not on specific input, variable, or output values. Our evaluation is focused on specific output values, which are more easily evaluated by running the program (dynamic analysis).

Random testing involves generating a large number of random inputs for the Java function and then checking the output against the expected range. The main cons for random testing are that it may not guarantee full coverage or detect subtle issues, it can be time-consuming and resource-intensive and there's a possibility of missing edge cases if not enough random inputs are generated.

Concolic analysis is a more advanced approach that combines symbolic execution with concrete execution to analyze the behavior of a program. Practically, concrete input values are generated and the function is executed using them while keeping track of the symbolic path constraints arising as *if* statements are evaluated. Iterative new inputs, that differ in at least one of the previous path constraints are generated, such that eventually it is not possible to generate inputs anymore. At this point, all possible paths in the function have been explored.

The ability to explore all possible execution paths and thereby uncover all potential edge cases of concolic analysis

appears advantageous compared to random testing. Moreover, concolic analysis offers a more formal approach to analyzing code behavior. However, it presumably comes at the cost of more overlay and the simplicity of random testing might be useful for simple methods. The following question shall be addressed with the analysis program.

R1 Is concolic testing more effective than random testing for verifying a function's output range?

After answering our first research question (see the rest of this report), we endeavor to further improve our solution by expanding our research questions. To find further areas of exploration we look at limitations of our current approach. We find that the largest issue with our approach is how it handles loops.

A concolic analysis, as it follows concrete values through the execution, is k-bound by nature. Exploring all possible paths on all programs is not feasible because paths grow exponentially, and thus we set a limit for depth of exploration, k. Loops create a large number of long execution paths in the concolic analysis, often exceeding k iterations in depth, which often means that we do not explore the entire loop behavior or any program behavior that occurs after a loop is executed. This presents a limitation to our approach for reasoning about ranges of outputs.

To explore whether this problem can be solved or improved upon we develop a second research question:

R2: Bonus Research Question: Can we improve the concolic analysis to verify a function's output range by skipping loops in the analysis?

This analysis will be done on Java bytecode. For simplicity, the analysis is limited to integer values.

Fundamentally the concolic analysis tool should arrive at the correct result, meaning if an out of range value for the example function is possible, the tool should find it. Furthermore, the tool's performance will be compared against randomized testing. The performance of random testing depends greatly on the probability of hitting an input causing an error in the range considered when generating random inputs. To nevertheless compare the two approaches, the probability needed for random testing to, on average, find a wrong input, as fast as the concolic analysis takes, is calculated.

The concolic analysis with loop skipping is compared against the plain concolic analysis in terms of performance, by testing their various runtimes on example functions with and without loops. Furthermore, loop skipping is evaluated based on the precision lost by it. If loop skipping results in losing soundness it will be considered unfeasible.

II. Approach

The Java code is compiled into bytecode to be used in the analysis. The foundation for our approach is a concolic analysis, which will examine all possible execution paths for different input constraints within the function. For each execution path that results in a return statement, the returned value is symbolically related to the function's inputs. In combination with the input constraints of a specific execution path, we can then calculate if this specific path can return out of range values.

An analysis example for the java function in the introduction can be seen in Table I.

TABLE I
EXAMPLE ANALYSIS

Inputs	Path Constraints	Symbolic Return	Return Check
$q = 0$ $cE = 0$ $cG = 0$ $e = 0$	$\neg(cG \geq 1$ $\wedge cG \leq 10)$	no return	
$q = 100,$ $cE = 3$ $cG = 2$ $e = 1$	$cG \geq 1 \wedge cG \leq 10$ $\wedge e \geq 1 \wedge e \leq 10$ $\wedge q > 0 \wedge cE > 0$ $\wedge e \leq cG$ $\wedge\neg(qC * cE \leq 120)$	$qC * cE - 2e$	$\neg(qC * cE - 2e >= 0)$
$q = 30$ $cE = 3$ $cG = 2$ $e = 1$	$cG \geq 1 \wedge cG \leq 10$ $\wedge e \geq 1 \wedge e \leq 10$ $\wedge q > 0 \wedge cE > 0$ $\wedge e \leq cG$ $\wedge qC * cE \leq 120$	$qC * cE - e$	$\neg(qC * cE - e >= 0)$

The analysis tool receives a list of conditions for the output of the function, e.g. > -10, < 100, $! = 0$. We then build an expression for each of these conditions, such that they evaluate the symbolic return statement. These expressions are each negated and combined using logical OR. We then check if it is possible to generate inputs fulfilling all path constraints as well as the just created OR expression. So inputs that lead to the same execution path and therefore the same symbolic return statement, but also violating at least one of the output conditions. If we can generate such inputs, we have found an out of range output, and the analysis is stopped. If we can not generate such inputs, the analysis is continued and the next execution path is explored. If there are no execution paths left to explore, the analyzed method will never return out of range values.

Approach Bonus Research Question

In order to **skip loops** we need to reason about their behavior without exploring all loop iterations. Essentially we must determine how they will transform the variables without running the entire loops themselves.

For some simple programs, this proves doable. An example is this simple loop algorithm:

Algorithm 1: Simple Loop

```
1 a = 0 ;
2 b = 0;
3 while b < 10 do
4     b = b + 1;
5     a = a + 2;
6 end
7 return a;
```

We can reason about the output of this algorithm by looking at the transformation on b in one loop iteration and using this to determine how many iterations until b fulfills the

requirements to break the while loop. Once we have the number of iterations, we can reason about how a will be transformed: a is increasing by 2 for each iteration, so we can see that the final value for a is $a(initial) + 2 * 10$.

However, as we explore more complex loops, reasoning about their behavior becomes more difficult. Eg;

Algorithm 2: More Complex Loop

```
x = 0 ;
y = 01;
while x < 100 do
    x = x + y;
    y = y + 1;
end
return x;
```

Determining the behavior of such a loop proves difficult. The behavior of the loop is non-linear, as x grows in an unpredictable manner. In fact, as we increase loop complexity, we find that determining the behavior of some loops without running the loops themselves is impossible. We will discuss why in the $Discussion$ section.

Having found that it is not possible to both skip loops and determine their outputs, we settle for the next best thing, which is to **predict** loop outputs. This leads us to an approach involving **linear regression**.

We start by developing a linear regression approach for predicting outputs of simple loop algorithms with the intention of expanding in order to accommodate more complex loops.

In our approach, the loop is first executed some amount of times n. As we are looking at bytecode, loops are not directly visible. But actually, we can just consider if (and ifz) opcodes, as they are where the program splits into different execution paths. So we keep track of the execution states at each if instruction throughout the analysis.

When an *if*-opcode is reached, we check if we already collected n states for this specific program count. Additionally, we keep track of all previous if statement results, such that we can now check that the loop has been running steadily. Given the loop executed steadily n times, we then try to predict how many more iterations of the loop we need to run for the considered *if* statement to change value.

Here the collected states are used as we do a linear regression on the top value(s) of the stack, which are the values relevant for the *if*-opcode, in all states matching the *if* statements program count. As we limited the analyzed methods to only use addition and subtraction inside loops, we only fit a first order polynomial. Later, we will explore using more advanced mathematical models to increase the coverage. With the fitted model we can then solve for the number of iterations until the *if* statement changes value.

After all *if*-opcodes involved in the loop have calculated the additional number of iterations needed to change their value, the lowest of these, i, is chosen. The state is then updated according to the prediction linear regression makes on the variable values after i additional iterations, and the analysis continues. If none of the *if* statements in the loop changes value based on the prediction of the regression, we exit the analysis stating that we have discovered an infinite loop.

As we are missing path constraints for the iterations of the loop skipped, we also add an expression $f\forall x, where 0 > x > i$ for each involved *if* statement describing that we assume it to stay constant for the iterations skipped. This is crucial to not generate inputs the next time around that leads us on the same execution path. Similarly, without it, we might calculate inputs to return out of range values that would have actually reached a different symbolic return expression.

After predicting outputs for simple loops, we attempt to develop our solution to predict outputs for more complex loops using an enhanced linear regression function. In this way, we can account for higher-order polynomials and other predictable linear functions that may arise from loops. We will explain further how we approached this in the $Implementation$ Section. Although our new approach does expand on which functions we can predict, we discover that we still can not predict the majority of loops, and this approach is therefore incomplete. We will discuss this further in the $Discussion$ section.

III. Implementation and Setup

Comparing the efficiency of concolic analysis to random testing in detecting when a function generates outputs outside the expected range requires several steps. These steps included creating the tests function for data comparison, creating the Bytecode, creating a Bytecode interpreter, writing a concolic analysis, and creating a tool for Random Testing.

- **Creating the Test Functions:** The choice of testing functions, including "CalculateEfficiency", "no loop", "short loop" and "long loop" was made to provide a diverse set of scenarios for evaluating the performance of concolic analysis and random testing. Each function represents different aspects of a Java program and presents unique challenges and characteristics that influence the comparative analysis.
  The testing functions were written in Java. The "short loop" function simulates a financial transaction, where the cost of products is deducted from an initial amount of money. It includes a loop that repeats the transaction for a fixed number of times. The function contains assertions to check that certain conditions are met, ensuring the inputs are valid. The loop and conditional statements within the function introduce paths that may or may not lead to out-of-range outputs. This function is useful for evaluating how well each method handles loops and conditional statements. The function's out-of-range output is 0. The "CalculateEfficiency" function involves calculations based on parameters such as the quantity of insulant, class efficiency, color grade, and exposure. It includes conditional statements with assertions to ensure that input parameters are within certain bounds. This function introduces complexity through conditional

statements and multiple input parameters. The function's output values must be zero or positive as they describe efficiency. Similar to the "short loop" example, the function simplifies a real-world scenario, however, it captures essential aspects of program behavior that are relevant for testing methodologies.

- **Creating the Bytecode:** From the .java files of the test functions, using the command "javac file.java" we create the file .class and through the help of jvm2json the class files are transformed into Bytecode in .json format with the command "jvm2json -s file.class -t file.json" (jvm2json repository).

- **Creating a Bytecode Interpreter:** The interpreter is coded in Python. The idea is to determine the set of instructions the bytecode would support. Each instruction should perform a specific operation, like adding two numbers or jumping to a different part of the code.

- **Writing the Concolic Analysis:** The goal is to explore different paths through the program by manipulating the symbolic values and collecting constraints on these values. The analysis is carried out with the help of Z3 (satisfiability modulo theories (SMT) solver developed by Microsoft), which is a solver for symbolic logic. It helped us resolve the constraints to explore all the possible paths and provide concrete values to our variables. The concolic analysis was timed using the Python function time().

- **Creating a Random Testing Tool:** The tool has been built using Java, the program takes as input the name of the function to be tested and the number of inputs the function expects. Additionally, it prompts the user to specify the lower and upper bounds of the out-of-range values. Once the user inputs are provided, the program dynamically loads the target program class using Java reflection. It then searches for the specified test function within this class. The testing process begins by generating random input values for the function within the specified range, this is when the runtime timer starts. Of course, it is noted that as the input range increases, the tool will require more time to find an out-of-range output. The tool repeatedly invokes the function with these random inputs until the result falls outside the user-defined range. The number of attempts (tries) is counted during this process. The timing of the testing process is recorded using System.nanoTime() from the library System, measuring the elapsed time between the start and end of the testing loop. If the program finds a result, it outputs the elapsed time, the number of tries, the out-of-range result, and the input values that caused the function to go out of range. The tool does not contain a memory to save all the inputs it tries because it has been noticed a significant decline in performance saving the inputs used in each iteration or using algorithms to create completely new input combinations each time. As for any randomized testing tool, the effectiveness of the tool depends on the randomness of the generated inputs and the characteristics of the tested function.

- **Exploring advanced linear regression techniques for skipping loops:** For enhanced linear regression we use a provided linear regression function from Python's Sci-kit-learn library. We felt that developing our own enhanced linear regression in Python would be beyond this project's scope and that we would be better served with an already-built function. The regression function provided by Python allows us to fit more complex mathematical functions, beyond the first-order polynomials of our initial solution.
  In the end, we find that this approach becomes too complicated and ineffectual for us to implement in our final solution, which we will elaborate on in the $Discussion$ section.

Source Code on GitHub: ProgramAnalysis-Project

IV. Evaluation and Results

The empirical evaluation aimed to **compare the efficiency of concolic analysis and random testing** in analyzing if a Java method with the desired output range will ever return out of that range. A diverse set of Java functions has been examined, each equipped with specified output boundaries, to assess which testing method would better identify input values outside the predefined output intervals.

The construction of random testing, including the size of the range for the generated inputs and the possibility of trying the same input combinations more than one time (since it does not have a memory which would significantly increase the runtime of the tool), importantly impacts performance. The **execution environment**, including machine specifications, introduced additional variability, therefore we opted to compare the different runtimes in the same Virtual Machine (VM) trying to reproduce the same running environment for all the test functions and testing methods. The VM was created using Oracle VM, the machine had Windows 10 OS installed with 8GB of RAM memory allocated for its use. The processor was a quad-core, Intel i5 with a CPU speed of 2.40GHz. Also, after executing some tests on different machines we noticed that machines with higher clock speeds generally lead to faster execution. Larger cache sizes and newer cache hierarchies can reduce memory access times, enhancing overall program performance. These machine-specific characteristics should be considered when interpreting results.

The evaluation is split up into the following parts. First random testing is compared to the concolic approach using methods that output out of range, and methods that do not,

separately. Afterwards, the performance of skipping loops is evaluated. Hereby the potential speed-up as well as differences in correctness are compared.

For execution times of the concolic tool, enough iterations have been averaged, such that the variance was below 0.001. For the random testing, since the numbers are generated uniformly at random, the sample variance was lowered below 0.001 as well.

In cases where the analyzed method can return out of range values, the performance of random testing is dependent on the time it takes to run the analyzed function as well as the density of wrong inputs, in the inputs space for random testing, producing out of range values. The density of wrong inputs in the input space for random testing depends on the method analyzed and the size of the input space used. Therefore it is not possible to say the concolic approach is faster than random testing on a certain analyzed method. Instead, a statement can be made on the required density d_r of wrong inputs in the input space for random testing to be equally fast as the concolic analysis. In particular, the following equation can be derived by leveraging the concept that the expected number of attempts until success is $\frac{1}{p}$, where p is the probability of success and is equal to d_r in this case.

$$t_c = \frac{1}{d_r} \times \frac{\Delta t_r}{10^8}$$

Where t_c is the time taken by the concolic analysis to find an out of range value in the test function analyzed and Δt_r is the time taken by random testing to try out 10^8 random inputs. This equation can be rearranged to:

$$d_r = \frac{\Delta t_r}{t_c \times 10^8}$$

The lower the d_r value, the better the random testing performs on that test function compared to the concolic approach, as it is able to handle a lower density of wrong inputs to find a wrong input in the time taken by the concolic approach.

Four different functions that can potentially return out of range values have been created. Hereby two functions are solely using if-statements, one function contains a short loop and one function contains a long loop. The results are shown in the table below.

TABLE II
RELATIVE PERFORMANCE OF CONCOLIC VS RANDOM

Function Name	Δt_r	tc	d_r
no loop	5.977	0.183	$\frac{327}{10^9}$
calculateEfficiency	7.698	0.034	$\frac{2264}{10^9}$
short loop	4.187	0.035	$\frac{1196}{10^9}$
long loop	4.084	1.098	$\frac{37}{10^9}$

Interestingly random testing performs significantly better on the first function compared to the concolic analysis than on the second function. This could be because the first function has four input variables, while the second function has three, which probably does not influence the Δt_r much, but makes the analysis for the concolic approach more complex.

The two functions with loops have similar execution speeds, as the long loop is likely heavily optimized when executed, but the concolic analysis takes a lot longer on the long loop which can be explained by the if cases inside the loop that on each loop iteration create path constraints and therefore exponentially more execution paths.

The actual runtime of random testing now depends on the range used to generate random inputs and on the actual density d_a of inputs causing an out of range output in the individual methods. In the case of the two functions with loops, the input's product needs to be 50 for an out of range value to be returned. As there is only a finite amount of integer pairs whose product is 50, d_a gets lower with a higher input range used. However, if it was every *even* input causing an error, d_a would stay constant with higher input ranges used. To put the derived d_r values into perspective, let's consider the *long loop* example, which is also the one performing best on random testing compared to the concolic analysis. There are six input combinations in total whose product is 50 and that therefore return out of range. To achieve the d_r value, the input size used by random testing can at most be 160 million, corresponding to a range of e.g. -6000 to 6000 for each input. So only this range per input can be tried out for random testing to average the same time as the concolic analysis for finding an input that returns out of range.

TABLE III
ABSOLUTE PERFORMANCE OF CONCOLIC VS RANDOM TESTING

Function Name	t_r	tc
no loop	11.293	0.183
calculateEfficiency	9.848	0.034
short loop	0.801	0.035
long loop	0.696	1.098

Table III shows the amount of time each testing method needs to find out of range values. Looking at the times, it can be noticed that increasing the number of inputs for the test functions (in the last two functions there are only two inputs, in the first two there are more) will lead to a significant increase in time for the random testing tool to find out of range values. This is because each integer input was generated using the same 0 to 10000 range in this example. All the functions have very few input combinations for going out of range, so therefore the functions with more input variables have an input space larger by a factor of 10000 which decreases d_a dramatically.

However, the first two functions have similar t_r values, regardless of *calculateEfficiency* taking fewer inputs than *no loop*. This points to *no loop* having a significantly lower density of inputs causing errors in the input space used.

Another interesting takeaway when using *no loop* is that to go out of range, the first two inputs need to be close to 0 and the last two inputs need to be close to 9000. This makes

it difficult in practice to decrease the input space as there would need to be different ranges for the different inputs to still have the wrong inputs included.

Even though random testing can, in the right conditions, be quicker than the concolic analysis, it has the major disadvantage that it can never tell the user that there are no inputs causing returning out of range (it is not complete). The following equation is used to calculate the range sizes for each individual input random testing is able to try out in the time taken by the concolic analysis to declare that there are no inputs causing out of range values.

$$i_r = \sqrt{t_c \times \Delta t_r \times 10^8}$$

Where i_r is the individual input range size for each input, Δt_r is the time taken by random testing to try out 10^8 values and t_c is the time used by the concolic analysis. The square root is used because the functions each take two inputs.

TABLE IV
CONCOLIC VS RANDOM WHEN THE FUNCTION CANNOT GO OUT OF RANGE

Function Name	Δt_r	tc	i_r
short loop (fixed)	4.099	0.518	14571
long loop (fixed)	4.136	134.443	235808

While the concolic analysis is relatively fast on the short loop example, it takes much longer on the long loop example allowing random testing to try out large ranges for the inputs in the same time.

The data in the tables above showed that **concolic testing generally demonstrates faster analysis times compared to random testing** if there a very few inputs causing out of range values. Moreover, in the case of test functions that could not go out of range, concolic analysis was able to determine this and give an answer, while random testing was just trying infinitely times to find an out of range value without giving any output.

The concolic analysis has proven very powerful in finding out of range values in methods without large loops or methods for which a majority of execution paths can return out of range. However, it struggles with methods that can not return out of range with big loops having if-cases inside, as then all of the many execution paths need to be evaluated.

Through testing the two different methods the **complexity of the Java program** emerged as a critical factor influencing the relative performance of concolic analysis and random testing. For straightforward programs, where paths are limited, concolic testing is very fast. However, as a program's complexity increases, the number of execution paths rises exponentially making the concolic analysis slower.

Random testing, on the other hand, demonstrated effectiveness through simplicity and suitability for more intricate code structures. Its probabilistic nature introduces uncertainty on its efficiency, but after using it multiple times for the test functions it became apparent that it can sometimes be faster than concolic analysis. Additionally, it can be noticed that in scenarios where a multitude of potential input combinations may result in out-of-range outputs, the efficiency of random testing surpasses that of concolic analysis. If it is likely to find an out of range input the simplicity of random testing comes into play, making it much faster than concolic analysis.

In safety-critical systems, where the consequences of software failure can be severe, prioritizing precision becomes crucial. Precision ensures that the testing method thoroughly explores potential edge cases, boundary conditions, and critical paths, reducing the likelihood of overlooking subtle vulnerabilities. Concolic analysis, with its systematic exploration of feasible paths, can be advantageous in such scenarios. In situations where testing resources (time, computational power) are constrained, achieving comprehensive coverage becomes a pragmatic choice. Random testing, being simpler and less resource-intensive, can be more suitable. While it may not guarantee the total exploration of all paths, it provides a quick assessment of the program's behavior across a broad spectrum of inputs.

In any empirical study, acknowledging **threats to validity** is crucial for ensuring the reliability and generalizability of the experiment.
There are important factors that should be addressed when discussing threats to validity:

- **Selection of Java Functions**: The effectiveness of concolic analysis and random testing is closely tied to the characteristics of the analyzed Java functions. The selection of functions for the evaluation may introduce bias, as certain functions might favor one testing method over the other. Many different functions were analyzed for diversity in function complexity and structure to mitigate this threat.

- **Quality of Random Inputs**: The effectiveness of random testing is intricately tied to the choice of the random number generation function, particularly within the context of Java's System library. This library relies on the linear congruential generator (LCG) algorithm to generate pseudo-random numbers, offering a pseudo-random number generation mechanism suitable for a myriad of applications, but it's important to note that the numbers generated by LCG are deterministic and repeatable if the same seed is used.
- **Execution Environment Variability**: The performance of both testing approaches is influenced by the characteristics of the machine on which the analysis is conducted. Variabilities in execution environments, such as differences in processing power and available resources, may introduce errors in the results. Tests were conducted on a Virtual Machine, without any heavy program in the background. Before starting each test the CPU usage was 10% and it was needed only for the basic applications to make the Windows OS work, so the test could use the remaining 90% for its computations.

Evaluation Bonus Research Question

TABLE V
DIFFERENCE OF PERFORMANCE BETWEEN NORMAL ANALYSIS AND LOOP SKIPPING ANALYSIS OF BYTECODE

Function Name	tc	tc-skip loops
no loop	0.183	0.195
calculateEfficiency	0.034	0.033
short loop	0.035	0.072
long loop	1.098	0.375
short loop (fixed)	0.518	1.726
long loop (fixed)	134.443	84.989

In Table V we see runtime comparisons between our normal method for concolic testing and the implementation where we include loop skipping. We can see that loop skipping is faster for longer loops and slower for shorter loops. This makes sense as longer loops generally take a long time to run through, but with loop skipping we can avoid that, however, this involves an initial time loss to skip the loop, which we can see in the longer runtimes for short loops.

V. DISCUSSION OF THE TECHNIQUE AND ALTERNATIVES

Over time, numerous methodologies have emerged for the evaluation of programs, each tailored to specific contexts. The selection of an appropriate testing method is crucial to prevent inefficiencies, as different tests exhibit distinct characteristics such as speed, soundness, maintainability, and completeness. It is imperative to align the testing strategy with the specific objectives to avoid any compromise in the aforementioned aspects.

In the following paragraph, diverse testing methodologies will be compared, including the one utilized to reach the goal of finding inputs that would lead to out of range output values. A possible alternative technique to reach our goal could be using abstract interpretation, also called symbolic execution which is a **static analysis**, because it examines the program source code and estimates what it might do at run time. **Abstract interpretation** simulates program execution, but each expression and each variable evaluates to a property, also known as abstract value. It computes an abstract state at every program point, most of the time the abstract state is a mapping from each variable to an abstract value. Such an approach can handle large methods with loops much better as it terminates once it finds a fix-point, implying a state where the analysis doesn't change with further iterations. In other words, it reaches a stable condition. Terminating at a fixed point helps avoid unnecessary iterations and speeds up the analysis process. In our case, two possible types of abstraction techniques could be used:

- **Sign Abstraction:** Sign abstraction refers to a technique where the focus is on approximating the sign (positive, negative, or zero) of variables in a program. This abstraction is particularly useful in reasoning about numerical computations and conditions in programs. In the test functions that we are analyzing it would be fast and efficient when for example the out of range values are all the negative numbers because it would approximate our variables and give as output a value that could be positive or negative, corresponding to in range or out of range. However, it can only be used for this problem if out of range values have a specific sign that the in range values never have.
- **Range abstraction:** Range abstraction means approximating the possible values that variables can take with a range they could be in. Such an abstraction would be very flexible in modeling output ranges, but for fix-points to occur, the abstract ranges need to have a maximum or minimum (e.g. < -1000,> 1000), which might prove difficult to define in practice.

Despite the advantages, the approach is acknowledged as unsound because it over-approximates the outputs. This means it might provide results that are broader or less precise than the actual outcomes. However, it will maintain complete when applied to this problem, so it would always find an out of range value if it exists. Further, it might be difficult to suggest to the user which specific input combinations cause problems when using abstraction.

Moving to a **dynamic analysis approach**, random testing and concolic testing are the leaders of the dynamic environment. As explained before the decision of choosing concolic analysis besides random testing was because concolic analysis synergizes the strengths of concrete and symbolic testing, combining the advantages of both. With the goal of finding out of range inputs given a test function, concolic analysis seems to be sound and complete. Furthermore, its efficiency extends to small out of range output intervals, showcasing its robustness across a spectrum of scenarios, because it focuses on visiting all branches within the execution tree to find out of range values without taking as much time as random testing.

Discussion Bonus Research Question

In our attempts to solve the loop problem, we have learned that it is impossible to determine the behavior of deterministic loops without running the loops themselves. Upon further inspection, it becomes clear that we are asking ourselves a version of **The Halting Problem**. The Halting problem asks whether, given an algorithm, we can determine whether the algorithm will halt or not. The answer is no, whether an algorithm will halt cannot always be determined without running the algorithm itself. Loops are programs themselves, and thus are subject to the halting problem. When we look at our problem in this light, we can see why our goal of skipping loops in analysis cannot be achieved without losing accuracy: just as we cannot determine whether our loops will halt, we can not either determine when they will halt (on which iteration), nor their effects on the input variables.

In retrospect, this conclusion seems obvious; why perform any sort of program analysis if we are capable of entirely determining a program's behavior without running it?
Having learned that it was impossible to determine all loop effects, we moved on to prediction, which is what our implementation focused on. We found that this was doable for extremely simple algorithms, which reflect a predictable linear pattern. In our evaluation, we see that our loop skipping function is capable of producing correct outputs and decreasing runtime for these types of loops. This approach, however, is ultimately crude, and it was necessary to explore solutions for predicting more complex loops.
When looking at more complex loops, the Sci-kit-learn regression function proved effective at modeling more complex loop behavior as mathematical functions. It also allows us to access bias with the $intercept$ method, which is important for our prediction. It is also possible with this method, to expand to multiple linear regression, which means we could also use relationships between the variables to predict output, as well as the relationships between variables and iterations.
Although in theory, we could now evaluate some more complex loops, it became clear that we still could not accurately predict the majority of loops, and for this reason, we abandoned the attempt to incorporate more complexity. We are still limited to loops that map to linear functions, we begin to have trouble with nested loops, and we struggle to determine whether our prediction is at all accurate.
Overall, even when we settle for predicting loop behavior, we are still extremely limited. Firstly, the loop must map to a linear mathematical function, in the case of our final implementation, this must be a first-order polynomial. Any dynamic loop behavior will not be mapped correctly and will either be ignored or will have the effect of producing an inaccurate regression model and thus inaccurate predictions.
Positively, we have managed to decrease runtime for some simple loops when incorporating loop skipping, however, our solution then becomes unsound and incomplete.

VI. Conclusion

In conclusion, the research showed the comparison between concolic analysis and random testing in finding inputs that would lead to out of range output values. The paper discusses which technique is faster, which is sound, and complete, and in which cases it's better to use one or the other. The evaluation of both methods across various test functions with different complexities revealed their strengths and limitations.

The results indicate that concolic testing generally demonstrates faster analysis times compared to random testing, particularly in scenarios where the set of out of range values is small, making it difficult for random testing to find one. Its systematic approach ensures comprehensive path exploration, providing an advantage in safety-critical systems where precision is crucial. Moreover, contrary to random testing, concolic analysis proves completeness in identifying whether there are inputs that lead to out-of-range values or not.

The complexity of the test functions emerged as a critical factor influencing the relative performance of concolic analysis. For straightforward programs with limited paths, the concolic analysis can be very fast, but as program complexity increases and execution paths grow exponentially, concolic analysis slows down more than random testing does.

In summary, the choice between concolic analysis and random testing depends on the specific characteristics of the functions tested and the testing objectives, precision requirements, and available resources. If a function is rather simple and wrong input values are rare, concolic analysis is the better choice. When testing a complex function with many wrong input values, random testing can be considered. If completeness is required, only concolic analysis can be delivered.

Conclusion Bonus Research Question
Our second research question asks whether our concolic approach can be improved by adding a loop-skipping function. We have concluded that the answer is no. Firstly, it is impossible to skip loops and retain accuracy in the system, it is necessary then that such a loop-skipping function be based on prediction. Although we have managed to decrease runtime for some simple loops, the sacrifice to soundness and completeness are not worthy trade-offs. When looking at more advanced loops, complexities arise that render the solution impractical and ineffectual. In conclusion, we can see that loop skipping is at best crude and overall a hindrance to the reliability of the analysis.

VII. Appendix

```
public static int no_loop(int coeff1, int
    coeff2, int coeff3) {
  //legal output range: (0,inf)

  assert coeff1>1;
  assert coeff2>1;
  assert coeff3>1;
  assert coeff3<9005;
  assert coeff1<=400;
  assert coeff2<=400;

  int calculated = coeff1*coeff2*coeff3;
  int temp = coeff1 + coeff2 + coeff3;
  int sum = temp*4;
  if (calculated < 27000) {
    calculated = calculated + 4500;
    return calculated;
  }else {
    calculated = calculated - sum;
    return calculated;
  }
}
```

```
public static int short_loop_fixed(int
    CostPerProduct, int QuantityOfProducts) {
   //legal output range: !=0

   assert CostPerProduct>0;
   assert QuantityOfProducts>0;
   int cost =
       CostPerProduct*QuantityOfProducts;
   int money = 500;

   for (int transaction=0; transaction<10;
       transaction++) {
      if (money<-100) {
         return money; }
      else if ((money-cost) == 0) { // If we
          get into this the function will
          never go out of range - the money
          will never be 0
         money = money-cost-1; }
      else {
         money = money - cost; }
   }
   return money;
}
```

```
public static int short_loop(int
    CostPerProduct, int QuantityOfProducts) {
   //legal output range: !=0

   assert CostPerProduct>0;
   assert QuantityOfProducts>0;
   int cost =
       CostPerProduct*QuantityOfProducts;
   int money = 500;

   for (int transaction=0; transaction<10;
       transaction++) {
      if (money<-100) {
         return money; }
      else if ((money-cost) == 1) {
         money = money-cost-2; }
      else {
         money = money - cost; }
   }
   return money;
}
```

```
public static int long_loop_fixed(int
    CostPerProduct, int QuantityOfProducts) {
   //legal output range: !=0

   assert CostPerProduct>0;
   assert QuantityOfProducts>0;
   int cost =
       CostPerProduct*QuantityOfProducts;
   int money = 50000;

   for (int transaction=0; transaction<1000;
       transaction++) {
      if (money<-100) {
         return money; }
      else if ((money-cost) == 0) { // If we
          get into this the function will
          never go out of range - the money
          will never be 0
         money = money-cost-1; }
      else {
         money = money - cost; }
   }
   return money;
}
```

```
public static int long_loop(int
    CostPerProduct, int QuantityOfProducts) {
   //legal output range: !=0

   assert CostPerProduct>0;
   assert QuantityOfProducts>0;
   int cost =
       CostPerProduct*QuantityOfProducts;
   int money = 50000;

   for (int transaction=0; transaction<1000;
       transaction++) {
      if (money<-1000) {
         return money; }
      else if ((money-cost) == 1) {
         money = money-cost-2; }
      else {
         money = money - cost; }
   }
   return money;
}
```