VBNConcolicGen: Concolic Java Unit Test Generation Using Soot and Z3

Viet-Hung Nguyen
Simon Fraser University
vhn@sfu.ca

Brendan Saw
Simon Fraser University
bsaw@sfu.ca

Nathan Tsai
Simon Fraser University
nta41@sfu.ca

Abstract

Concolic execution (**con**crete + sym**bolic**) has been used by Sen et al. to generate unit tests for both C and Java programs through CUTE and jCUTE respectively. Those libraries were written almost 20 years ago, with older dependencies [1]. This project aims to recreate portions of the CUTE library using dependencies and versions of Java. This project recreation will be substituting Sen et al.'s jCUTE usage of JDK 1.4 with JDK 11, and will use Z3 as the SMT solver, offered by Microsoft Research, rather than lp_solve. Our project can be found on GitHub, here: VBNConcolicGen.

1 Introduction

1.1 Problem

Unit testing can be seen as a chore for developers [2], which is why it is often neglected in the development process [3]. However, unit testing, or testing in general can increase the safety of new code deployments. A good test suite can detect issues caused by new code being introduced before the new logic makes its way into production code - if the tests are ran as part of a regression test suite. Since coupling of functional code to test code increases proportionally to the granularity to the tests (i.e., unit tests would be the most highly coupled to the implementation logic), unit tests must be modified a significant number of times if the logic that they correspond to is modified. If there were a tool that were to automatically generate unit tests based on simply the format of a program, it would accelerate the development process by allowing more trivial test cases to be created, leaving more complex and important test generation to developers, and increasing the velocity of any software development process using these tools.

2 Objective

Our project aims to implement Java unit test generation capabilities, essentially creating a Java version of CUTE using newer technologies and updated versions of libraries. We aim to perform basic instrumentation on Java code, be able to run the instrumented program multiple times, and generate solved inputs such that every branch of a Java program is explored. We will be using Java JDK 1.11, Soot 4.4.1 to instrument the code, and the Z3 SMT solver to solve constraints that will allow us to re-execute program such that we reach alternative branches. We are hoping that we can leverage Z3 instead of lp_solve to improve the performance of how fast inputs are solved.

3 Related Research

3.1 CUTE

CUTE, published in 2005 by Koushik Sen and his team, focuses on creating unit tests for C. It had the core idea of generating unit tests to maximize branch coverage. CUTE used CIL to instrument C programs, and used lp_solve - software for solving linear, integer and mixed integer programs - rather than an SMT solver, since solvers like Z3 hadn't been invented at the time). CUTE implemented a bounded depth-first search strategy to prevent combinatorial explosion when run on larger programs. CUTE came to be known as a concolic unit testing framework, which utilized symbolic execution with constraint solving to generate test inputs that provided both line and branch coverage. CUTE provided the ability to automatically test sequential C programs, including pointers [1].

3.2 jCUTE

After the publication of CUTE, jCUTE was developed. jCUTE focused on bringing CUTE's functionality to Java, and added the ability to test concurrent programs. Since their

inceptions, jCUTE and CUTE have been used a variety of research projects [5]. Additionally, CUTE's methodologies have been integrated into other tools currently used by developers at Microsoft [6] through the development and adoption of Microsoft Pex. jCUTE in particular has been shown to successfully detect concurrency bugs within Sun Microsystem's Java system code [4].

While the research that has been created regarding CUTE & jCUTE has been widely adopted by other sources (both in academia and industry), both the CUTE and jCUTE automatic unit test generators use old versions of C and Java respectively. jCUTE in particular uses older libraries such as lp_solve 5.1 for their constraint solving, and Soot version 2.2.3 for symbolic execution instrumentation of Java code. jCUTE also does not have a build system - using a script to manually include dependencies, and in order to run jCUTE on a 64-bit system, the user must build the distribution archive from the sources in the publicly available jCUTE GitHub. The reason for these older dependencies is attributed to the fact that jCUTE was invented by Koushik Sen around 2006, the paper for jCUTE published in 2013, and no public repository indicating further changes from that period onward [8].

4 Methodology

We started by examining the original CUTE research paper, deconstructing the pseudocode and data structures denoted in the paper originally meant for unit test generation in C [1]. Unfortunately, due to a lack of an available paper illustrating how jCUTE was adapted from CUTE to handle the Java programming language, we had to make inferences in how the data structures were altered when handling the migration to a language that doesn't have pointers. We ended up removing all references to pointers and memory manipulation since Java handles that information on behalf of us. We also were mainly focused on handling primitives, narrowing our project scope to that similar to CUTE rather than jCUTE. Our program is comprised of four distinct components:

- Instrumentation
- Runtime Library
- Concolic Runner
- Constraint Solver

4.1 Instrumentation

4.1.1 Overview

To have symbolic execution running side by side with program execution, we first focused our efforts on the instrumentation of the code. We utilized the `BodyTransformer` class from the Soot library to instrument the Jimple intermediate representation (IR) of the Java source code. The instrumenting program transforms each method in the Jimple program, iterates through each statement and inserts method invocations to our defined runtime library class called `Call`.

4.1.2 Jimple Data Type

Based on the Soot source code and the Jimple IR, we deduced the data types using a simplified representation of the method body in the Jimple program. The following pseudocode in this section utilizes pattern matching for greater clarity. However, Java does not support pattern matching. To achieve that in Java, we utilized the provided `Switch` classes from the Soot library to instrument the program units accordingly.

```
Statement :=
    Assignment (Variable left, Value right) |
    Return (Value s) |
    ReturnVoid |
    Invoke (Method m) |
    Identity (Variable v)
    If (BinaryExpression expr) |

Operand := Constant c | Variable v
Expression := BinaryExpression (
        Operand v1,
        BinaryOp op,
        Operand v2
    ) |
    UnaryExprssion (
        UnaryOp op,
        Operand v
    ) |
    Method (MethodName name, Operand[] args)

Value := Operand op | Experssion expr
```

4.1.3 Instrumentation Algorithm

For simplification in the pseudocode, we used double quotes ("") in to wrap the variables to indicate extracting variable names from the Jimple program. Since a Jimple method body is a list of statements, we just simply iterate through each statement and apply the function `instrumentStatement`.

```
instrumentStatement(Statement s):
  match s:
    Assignment (left, right):
      instrument(right, s)
      insert Call.finalizeStore("left", left) after s
    Return s:
      insert Call.pushArg("s", s) before s
    Invoke v:
      instrument(right, s)
    Identity v:
      insert Call.popArg("v", v) after s
    If expr:
      instrument(right, s)
```

```
      insert (
        if (expr) Call.trueBranch()
        else Call.falseBranch()
        Call.finalizeIf()
      ) before s

instrumentValue(Value v, Statement s):
  switch v:
    Constant c:
      insert pushConstant(c) before s
    Variable var:
      insert pushSymbol("var", var) before s
    BinaryExpression (left, op, right):
      instrument(left, s)
      instrument(right, s)
      insert Call.applyOperand(op) before s
    UnaryExpression (op, value):
      instrument(value, s)
      insert Call.applyOperand(op) before s
    MethodCall (name, args):
      for arg in reverse(args):
        switch arg:
          Constant c:
            insert Call.pushArgConst(c) before s
          Variable p:
            insert Call.pushArg("p", p) before s
      insert Call.beginFunction(name) before s
      insert Call.afterFunction(name) before s
```

4.2 Runtime Library

4.2.1 Overview

The runtime library `Call` is for interacting with the global symbolic state during the program execution. It followed a similar pattern to jCUTE. Specifically, we accept calls of `push*()`'s, `applyOp(" + ")`, and `finalize*()`, then generate constraints accordingly. (See Appendix B for more details.) Based off of `finalizeStore` and `finalizeBranch`, we would generate a stack of constraints.

While this approach has high temporal coupling, it is reasonable because we rely on our instrumentation (not humans) to adhere to it. We also apply safety checks to ensure our instrumented code adheres to the invariant. Specifically, we clear the temporary data structures after generating the constraints and check that all data is present (constraints + operand always, evaluated result in the case of a branch) before generating a constraint.

When the instrumented code either terminates normally or on an `Exception`, we write the constraints to an external data store. Similar to CUTE, the external data store is simply a file. We serialize the state, then write to file. Afterwards, the state is read by the concolic runner and parsed.

4.2.2 Pseudocode

The general approach of the runtime library is:

```
// 1. Receive 1+ pushSymbol() or pushConstant()
// 2. If 2+ values pushed, require applyOperand()
// 3. If a branch, require pushTrue/FalseBranch()
// 4. Receive finalizeBranch() or finalizeStore()

// ... (repeat the above steps)

// 5. If error, require terminateOnError()
// 6. If not error, require terminateNormally()
```

See Appendix A for the full pseudocode.

Notes: The instrumentation library cannot handle reassignments consistently. Thus, we apply the heuristic of when there are less than 2 symbols, we assume it is a reassignment. Also, while we have pseudocode to handle function invocations, we have not implemented this feature due to lack of time.

4.3 Concolic Runner

4.3.1 Overview

To explore all branches of a program's execution, we need to generate different input variables for different paths of a program. The concolic runner of VBN handles this.

Before VBN can run a program, we must configure it with the number of inputs it requires and their data types. Currently, we store this configuration in a lookup map.

For the first iteration of concolic execution, we generate random values for each input, according to their data type. The seed for the random generation can be set manually for deterministic testing, with a default value of the system's epoch timeline in milliseconds.

A random input passed into the program would then be run, of which we would then wait for the program to finish executing, then retrieve the constraints that it had encountered during execution - these include commands like branching (if statements) and assignments. We obtain the constraints that the program encountered by reading an external data store generated from the runtime.

Based on the constraints explored and which side of the branch that the program actually explored with the initial random input, we can then negate the last constraint, and solve for a new input using Z3. These solved constraints are then passed into the program again to explore another branch. The process of checking the path that the execution took and finding another branch continues until there are no more branches to explore.

4.3.2 Pseudocode

```
execute(programName):
    instrument(programName)
    progInputStructure <- lookupMap[programName]
    programInput <- randInput(progInputStructure)
    runInstrumented(programName, programInput)
```

```
instrumentedState <- readIO()
constraints <- instrumentedState.constraints

while (!constraints.empty()):
    constraints <- negateConstraints(constraints)
    programInput <- z3Solve(constraints)
    runInstrumented(programName, programInput)
    instrumentedState <- readIO()
    constraints <- instrumentedState.constraints
```

4.4 Constraint Solver

4.4.1 Overview

To interpret our custom data types and convert them into the Z3 Java API, we created a wrapper, informally known as the "Z3Solver". The Z3Solver is passed a list of constraints, which it then needs to find a satisfiable assignment for. Some of our constraints have an "assigned" value, which represents assignment (ex: `x = y + z`). We must keep track of assignment to determine satisfiability within the program over multiple variables. We can track data dependence by observing every assignment and branch constraint. Other constraints may not have an assignment, such as if they are normal if statements (ex: `x \< y`). Both unary constraints and binary constraints must be handled. Unary constraints refer to expressions such as `-x`, while binary constraints are of the form `x == y`.

4.4.2 Pseudocode

```
z3Solve(constraints):
  z3Exprs: list
  for (constraint : constraints):
    switch constraint:
      UnaryConstraint (assigned, op, expression) ->
        switch op:
          NOT -> (z3Expr <- z3.mkNot(expression))
          NEG -> (z3Expr <- z3.mkMul(expression, -1))
        if assigned:
          z3Expr <- z3.mkEq(z3Expr, assigned)
      BinaryConstraint (assigned, left, op, right) ->
        switch op:
          AND -> (z3Expr <- z3.mkAnd(left, right))
          OR -> (z3Expr <- z3.mkOr(left, right))
          ...
          ADD -> (z3Expr <- z3.mkAdd(left, right))
          ...
          LT -> (z3Expr <- z3.mkLt(left, right))
          ...
        if assigned:
          z3Expr <- z3.mkEq(z3Expr, assigned)
    z3Exprs.add(z3Expr)
  z3.solve(z3Exprs)
```

5 Differences from CUTE's Approach

5.1 Symbolic State Representation

In our implementation, we treated variable names in Java program as symbolic entities and enforced the equality constraints on the assignment statements. We also attached a reference counter to the symbolic entity so that it will increase when the corresponding variable is reassigned. This is very different from the Sen's approach, where each variable in the program maps to a symbolic expression. When variable is reassigned, Sen just remapped the variable name to a different symbolic expression.

Further, we used different grammars to define the symbolic expression and constraints. Specifically, Sen used a simple grammar to generate arithemetic constraints:

```
<op> := + | - | * | /
<cmp> := >= | <= | < | > | == | !=
<symbolic> := <var> | <constant>
<expr> := <expr> | <expr> <op> <symbolic>
<constraint> := <expr> <cmp> <expr>
```

where as we distinguished between symbolic variable types (number or boolean) resulting more complex grammar.

```
<arithmetic_op> := + | - | * | /
<relation_op> := >= | <= | < | > | == | !=
<num_value> := <num_symbol> | <num_constant>
<bool_value> := <bool_symbol> | <bool_constant>
<arithemtic_expr> := - <num_value> |
        <num_value> <arithmetic> <num_value>
<bool_expr> := <num_value> <relation_op> <num_value>
<constraint> :=  <int_symbol> == <arithemtic_expr> |
    <bool_symbol> == <bool_epxr> |
    <bool_expr>
```

5.2 Concolic Runner

In the papers covering CUTE and jCUTE, mis-approximations were possible, and he had a way to check if the branch that was traversed, was the actual branch that the program had predicted the previous input would take it down. These errors did not happen often, though they also were necessary in the event that approximations were incorrect - at times, the bounded-depth first search can lead to branch prediction errors. In event of a prediction error, he would restart the concolic execution. For our VBNConcolicGen, we did not consider handling mis-approximations due to time constraints and limited sizes of test cases not requiring the bound set.

Sen and his team replace symbolic expression by concrete value when symbolic expression becomes 'unmanageable' - in particular, when non-linear statements appear. For our execution, we do not execute symbolically and replace with concrete values, but rather we directly run the programs with

solved concrete values - representing the negated branch currently traversed - each time.

5.3 Constraint Solver

The original constraint solver that was used in CUTE was `lp_solve`. The difference in choice of constraint solver led to a difference in the way that the constraints could be solved, and different related speeds. While there aren't measurements provided by Koushik Sen specifically, in one of his talks he talked about how it would be more difficult to solve non-linear equations [7]. The inability to solve non-linear equations is easily overcome as part of the functionality included with Z3, and non-linear equations are handled as if they were any other type of equation.

6 Details Not Mentioned in CUTE

6.1 Instrumentation

The CUTE paper mentioned about function instrumentation, but did not have sufficient details on the instrument methodology, such as handling and transferring arguments from one stack frame to the other. To resolve this issue, we deduced the methodology for instrumenting function from the jCUTE source code [8]. We ended up having an argument stack to transfer symbolic arguments between the execution stack frames.

6.2 Runtime Library & Constraint Solver

The CUTE paper did not mention how they handled floats and doubles. While `lp_solve` (the constraint engine used by CUTE) has a `set_scaling` function, we could not identify any usage in jCUTE. It is likely CUTE does not handle floats. In VBN, we convert floats into doubles and treat them as reals in Z3Solver.

Specifically, we used an approach by Eric listed at https://stackoverflow.com/a/1657688 to convert doubles into rational numbers, then create a Z3 real using a numerator and denominator - Z3 creation of a real number only accepts, integers, longs, and a pair of integers representing the numerator and denominator approximation of the real number.

6.3 Concolic Runner

The CUTE paper did not mention what do when constraints are unsolvable. In such a case, we should simply skip that iteration and continue up the trace of the branches.

The CUTE paper also did not mention that for each iteration of concolic execution, the instrumented code stores the state into a file to pass it back to the CUTE's concolic runner.

7 Issues Encountered

One issue we faced when implementing VBN was handling Jimple's `cmpg` and `cmpl` commands (there is also `cmp` but that symbol never appeared in our test cases). To address this issue, we approximated `cmpg` and `cmpl` with Z3's `GTE` and `LTE` respectively.

8 Evaluation

To evaluate our project, we originally were going to test our code on the test cases in testcomp23. However, our project was implemented enough to be able to handle these more complex test cases.

Thus, we looked for the test cases originally handled, but CUTE, but could not find them. Instead, we compared our implementation against test cases in CREST [9].

We didn't want to use the jCUTE tests since the jCUTE program was quite complex, handing similar features we haven't implemented (e.g. function calls) and quite complex ones (e.g. concurrency). CREST had more tests related to primitives and basic operations that we had built our program to handle.

Ultimately, we evaluated our program with CREST unit tests and our own custom tests to detect missing features and identify where the shortcomings of the program exist at a finer grain detail. The metric we will be looking for will be branch coverage, as the original CUTE paper examined that metric - albeit at a larger scale while looking at execution of the SGLIB library [1].

9 Results

9.1 Overview

Our project ended up being able to handle branching through if-statements, and was able to accommodate boolean, int, float, and double primitive types. Array fields are also able to be written to and read from, by characterizing each array field as if it was its own variable. Features that are missing from the project include looping, due to bugs regarding how we handle reassignment of variables, and object field accessing which fails due to us not handling nested classes correctly during instrumentation. Function calls are also slightly flaky, and fail if variable names are the same across functions. A tabulated form of the program capabilities can be found in Table 1.

9.2 Running Example Branch Coverage

We ran our code on a variety of personally created custom tests. Here is an example Java test (`vbn.examples.Test_07_If_Multiple_Diff_Types`) that we ran our code on:

```
package vbn.examples;
```

```
public class Test_07_If_Multiple_Diff_Types {
    static int x;
    static int y;
    static int z;
    static boolean t;

    public static void main(String[] args) {
        x = Integer.parseInt(args[0]);
        y = Integer.parseInt(args[1]);
        z = Integer.parseInt(args[2]);
        t = Boolean.parseBoolean(args[3]);
        if (t) {
            if (x > y) {
                System.out.println("Path 1");
                if (y > z) {
                    System.out.println("Path 1.1");
                } else {
                    System.out.println("Path 1.2");
                }
            } else {
                System.out.println("Path 2");
                if (x > z) {
                    System.out.println("Path 2.1");
                } else {
                    System.out.println("Path 2.2");
                }
            }
        } else {
            System.out.println("Path 3");
        }
    }
}
```

Running the program on this, resulted in these inputs:

```
[472998880, 37529119, -572350825, false]  // path 3
[0, 0, 0, true]   // path 2.2
[0, 0, -1, true]  // path 2.1
[0, -1, -1, true] // path 1.2
[1, 0, -1, true]  // path 1.1
```

These found program inputs correctly cover all possible branches in the program.

9.3 Metrics

As shown in Table 2, we covered all the branches that encapsulated purely the functionality shown in our personally created programs that we tested our concolic unit test generator on. The programs that we were unable to instrument, such as concrete return and unsigned were due to the original CREST tests being in C, which had different syntax from Java, including the ability to have unsigned integers, or assign a printf to an int (`concrete_return`) [10].

Unfortunately, we weren't able to get our code to run on many of jCUTE's tests - they often instrumented programs that had objects, loops and employed multiple threads. Evaluating our program on testcomp23 also ended up with suboptimal results - many of the tests would use arrays for their algorithms.

Custom Test Name	Our Results
Basic, no branching	Pass
Single If Statement Between Two Input Variables	Pass
Nested If Statements Between Input Variables	Pass
Nested If Statements With Multiple Inputs With Varying Types	Pass
Single If Statement Between Integer Input and Concrete Integer Value	Pass
Single If Statement Between Double Input and Concrete Double Value	Pass
Single If Statement Between Float Input and Concrete Float Value	Pass
Ability to Handle Incrementing Variables	Pass
Calls to other functions	Fails (incorrect results)
Looping	Fail (reassignment issues)
Object field accesses	Fail (nested classes not handled)
Array field accesses	Pass

Table 1: Results of instrument our custom programs to determine functionality of our concolic unit test generator

10 Future Work

In the future, we would like to build upon the progress that we have achieved in this project. This involves fixing the bugs causing incorrect results when run on programs.

We would like to allow for functions to be called within the main function, allow for looping, and enable objects to be created with their fields accessible to be instrumented.

Currently, we only supports symbolic values for primitive types. Supporting symbolic values for Java Object types could be a potential expansion for our project. Furthermore, we could add supports for Java object reference constraints, inspired from the pointer constraints in CUTE paper [1].

We would also like to be able to instrument the code to automatically detect input variables and their types - we are currently using a map to store this information. It would also be great to automatically generate the unit tests in the style of

Test Name	Results	Reason for Fail
cfg_search_test	Solver fails	Loops
cfg_test	Solver fails	Function calls
concrete_return	Cannot instrument	Unsuppported assignment
function	100% branch cov.	N/A
left_shift	Instrument fails	Shift operator
math	100% branch cov.	N/A
shift_cast	Instrument fails	Shift operator
simple	100% branch cov.	N/A
struct_return	Execution fails	Objects
struct_test	Instrument fails	Objects
table_test	Solver fails	Loops
unary	100% branch cov.	N/A
uniform	100% branch cov.	N/A
unsigned	Cannot instrument	Unsigned int

Table 2: Results from running VBNConcolicGen on CREST tests. Some tests could not be transferred to Java (e.g. using unsigned int), so we manually evaluated them.

a JUnit syntax, instead of generating merely the inputs to the programs.

11 Conclusion

In conclusion, the development of this application has proven to be a much larger endeavor than we (Viet, Brendan, and Nathan) had ever expected. The experience of examining Sen and his team's work, and re-implementing the application from the ground up has given us a greater appreciation for the tools that use this technology today, enabling our software to be safer and better tested. While we weren't able to implement all of the core features that the original CUTE program offered, we were happy with our progress of implementing many of the primitive types within the time we were allocated. Overall, we learned a great deal about concolic execution, and its efficacy in unit test generation.

12 Acknowledgements

We would like to thank Dr. Nick Sumner for his supervision and insight into the design of VBNConcolicGen.

References

[1] Sen, K., Marinov, D., & Agha, G. (2005). CUTE: A concolic unit testing engine for C. ACM SIGSOFT Software Engineering Notes, 30(5), 263–272. https://doi.org/10.1145/1095430.1081750

[2] Leman, G. (2022, May 11). Unit Tests Slow Me Down. CodeX. https://medium.com/codex/unit-tests-slow-me-down-98a6bac41462

[3] Bannister, J. (2020, March 12). Why Developers Don't Write Unit Tests. LinkedIn. https://www.linkedin.com/pulse/why-developers-dont-write-unit-tests-justin-bannister/

[4] Sen, K., & Agha, G. (2006). CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In T. Ball & R. B. Jones (Eds.), Computer Aided Verification (pp. 419–423). Springer. https://doi.org/10.1007/11817963_38

[5] Anand, S., & Harrold, M. J. (2011). Heap cloning: Enabling dynamic symbolic execution of java programs. 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 33–42. https://doi.org/10.1109/ASE.2011.6100071

[6] Tillmann, N., & de Halleux, J. (2008). Pex–White Box Test Generation for .NET. In B. Beckert & R. Hähnle (Eds.), Tests and Proofs (pp. 134–153). Springer. https://doi.org/10.1007/978-3-540-79124-9_10

[7] CUTE: A Concolic Unit Testing Engine for C Technical Report Koushik SenDarko MarinovGul Agha University of Illinois Urbana-Champaign. - ppt download. (n.d.). Retrieved April 23, 2023, from https://slideplayer.com/slide/6791414/

[8] Sen, K. (2013). JCUTE [Java]. Open Systems Laboratory. https://github.com/osl/jcute

[9] Software. (2018). Open Systems Laboratory. http://osl.cs.illinois.edu/software/index.html

[10] Burnim, J., & Sen, K. (2008). Heuristics for Scalable Dynamic Test Generation. 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 443–446. https://doi.org/10.1109/ASE.2008.69

A Runtime Library

Below are the functions of the runtime library.

```
static state

static valSt = Stack()
static op = ENum
static evalResult = null

pushSym(varName, value):
    sy = state.updateOrCreateSymbol(varName, value)
    valSt.push(sy)

pushConst(value):
    constant = createConst(value)
    valSt.push(constant)

applyOperand(operand):
    op = operand

finalizeStore(varName, value, lineNum):
    // handleReassignmentEdgeCases()

    assigned = state.updateOrCreateSymbol(varName, value)

    if 1 == SIZE(valStack)
        sym = valSt.pop()
        c = BinaryC(assigned, EQUALS, sym, lineNum)
    if 2 == SIZE(valStack)
        right = valSt.pop()
        left = valSt.pop()
        c = BinaryC(left, op, right, lineNum)
```

```
        c.assignedSymbol = assigned

    state.addConstraint(c)

branchEvalToTrue():
    evalResult = true

branchEvalToFalse():
    evalResult = false

finalizeBranch(lineNum):
    c = cg.genBranch(lineNum)

    if 1 == SIZE(valStack)
        sy = valSt.pop()
        c = UnaryC(op, symbol, lineNum)
    if  2 == SIZE(valStack)
        right = valSt.pop()
        left = valSt.pop()
        c = BinaryC(left, op, right, lineNum)

    c.isBranch = true
    c.evaluatedResult = evalResult

    state.addConstraint(c)

terminateProper():
  storeStateInExternalDatastore()

terminateOnError(err):
  state.addError(err)
  saveToExternalDatastore(state)
```

B Runtime Approach

In the OSL jCUTE repository, we see how CUTE instruments the code in the /src/cute/instrument/CuteInstrumenter.java file.

	CUTE	VBN

1.1. Sy+Sy	loadAddress(y);
	loadValue(y);
	loadAddress(z);
	loadValue(z);
	applyOp("+");
	store(x);
	x = y + z;
		pushSym("y", y)
		
		pushSym("z", z)
		
		applyOp(" + ")
		
		x = y + z;
		finalizeStore("x", x, ##)

1.2. Sy+C	loadAddress(y);
	loadValue(y);
	loadValue(4);
	applyOp("+");
	store(x);
	x = y + 4;
		pushSym("y", y)
		
		pushConstant(4)
		applyOp(" + ")
		
		x = y + 4;
		finalizeStore("x", x, ##)

Table 3: Examples of code instrumentation for assignment. (Sy = Symbolic, C = Constant).

	CUTE	VBN

2.1. f(S, S)	popAll()
	pushArg(y);
	pushArg(z);
	funBegin();
	f(y,z);
	funEnd();
		
		pushArg("y", y)
		pushArg("z", z)
		beforeInvokeFunc()
		f(y,z);
		afterInvokeFunc()

2.2. f(S, C)	popAll()
	pushArg(y);
	pushArg();
	funBegin();
	f(y,4);
	funEnd();
	// Fixed typos
		
		pushArg("y", y)
		pushArgConst(4)
		beforeInvokeFunc()
		f(y,4);
		afterInvokeFunc()

2.3. void f()	void f(int x,int y){
	popStore(x);
	popStore(y);
	
	return;
	}
		void f(int x,int y){
		popArg("x", x)
		popArg("y", y)
		
		return;
		}

3.1. int f(S, S)	popAll()
	pushArg(y);
	pushArg(z);
	funBegin();
	x = f(y,z);
	funEnd();
	popStore(x);
		
		pushArg("y", y)
		pushArg("z", z)
		beforeInvokeFunc()
		x = f(y,z);
		afterInvokeFunc()
		popArg("x", x)
		finalizeReturn("x", x, ##)

3.2. int f(S, C)	popAll()
	pushArg(y);
	pushArg();
	funBegin();
	x = f(y,4);
	funEnd();
	popStore(x);
		
		pushArg("y", y)
		pushArgConst(4)
		beforeInvokeFunc()
		
		afterInvokeFunc()
		popArg("x", x)
		finalizeReturn("x", x, ##)

3.3. int f()	int f(int x,int y){
	popStore(x);
	popStore(y);
	
	popAll();
	pushArg(z);
	return z;
	}
		int f(int x,int y){
		popArg("x", x)
		popArg("y", y)
		
		pushArg(z);
		return z;
		}

Table 4: Examples of code instrumentation for functions. (S = Symbolic, C = Constant).