Test synthesis for java programs

J. Lichman, T. Gross

February 14, 2019

Abstract

Achieving full code coverage is difficult task even with help of the code coverage tools. Untested regions of code are prone to suffer from bugs which can lead to crucial security flaws. In our approach we try to automatically synthesize tests which are able to execute specified region of code. Therefore our tool can increase code coverage by creating tests that can reach untested regions of code. Concrete implementation of the theory is for java program but can be easily applied to other programming languages.

1 Introduction

Software testing is crucial part of every software development cycle. Software bugs not detected by tests but later exploited by hackers already costed companies billions of dollars. Therefore creation of reliable test set is top goal of every company that wants to deliver highly reliable software.

Test coverage, used as indicator of test adequacy, became standard in software testing. Regions of code not covered by any test are more likely to contain vulnerabilities and therefore it is in testers best interest to have code coverage as high as possible. Difficulty of testing increases with program size since every new conditional branch in program brings risk for guarded region of code to not be reached by any test. However after software deployment attacker that is motivated enough can succeed in reaching this region of untested code and find vulnerabilities there that could be of high confidentiality.

In our approach we automatically create tests that can reach specified regions of code and detect unexpected behaviour before software release. One usage of our tool could be to first collect all not covered regions of code and subsequently run our tool for each of them to create test that covers it or to detect unreachable part of the code which can be in refactoring phase erased or there can actually be a bug in the condition guarding it which makes it unreachable.

The paper is organized as follows. Chapter 2 contains theory background needed to understand our approach. It includes control flow graph, static single assignment form as well as SMT solving description. In chapter 3 we present our approach. We begin with detailed problem description together with simplifications we had to make. Next subsection presents notion of conditional definitions. Following two subsections contain core of the work which is actual encoding into a SMT formula and describe every component of our solution

in great detail together with the examples. Chapter 4 presents analysis of our approach together with the performance measures and chapter 5 discusses possible support for the loops.

2 Theory

In order to present our approach we need to describe data structures, algorithms and techniques we built the tool on.

2.1 Control Flow Graph

In our approach we are doing the analysis on the top of the Control Flow Graph. It is a program representation using graph notation where nodes are parts of the program always executed together and directed edges connect nodes with the other nodes that can be executed after them. CFG models all paths that might be traversed during execution. We can see example of control flow graph in the figure 1 where original code is partitioned into the basic blocks and connected by the directed edges which indicates the order of the execution. Number of possible executions is equal to the number of distinct paths between entry and exit nodes in the graph.

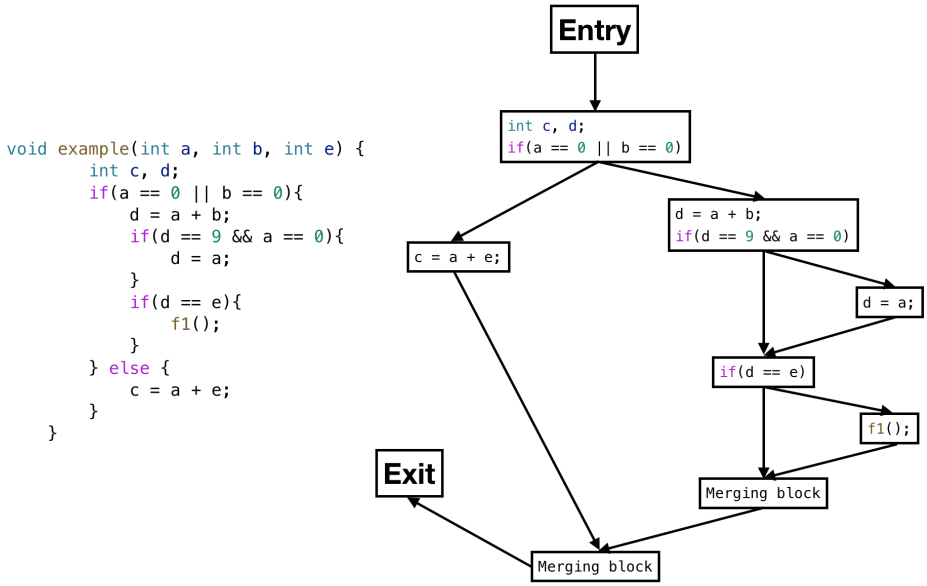

Figure 1: Example of Control flow graph.

2.2 Static Single Assignment

Static single assignment form is very useful in many optimization techniques or in the program verification. One can find it used in all major compilers or software verification tools. We use it as well since it is very useful in value propagation. Major property of SSA is that

it requires each variable to be assigned exactly once, and every variable is defined before it is used. Example in the listing 4 will look in the SSA form as in the listing 2, where Phi expression is merging point of the two variable versions. Phi node, however, gives us no more information. Only that variable $x3$ can take either value of $x1$ or $x2$. In our analysis we need to get more information about the variable definition such that we can decide when each definition is actually applied. We will describe exact usage of the SSA for our purposes in later chapters.

Figure 2: Example 4 in SSA form

```
int x1, x2, x3;
x1 = 0;
if(a < 0)
    x2 = 1;
x3 = Phi(x1, x2);
read(x3);
```

2.3 SMT solving

Satisfiability modulo theories (SMT) problem is a decision problem for logical formulas from multiple theories. In our approach we are using only integer and Boolean theory ($=, \neq, <, \leq, \geq, >, \%, +, -, *, /, \Rightarrow, \wedge, \vee, \Leftrightarrow$). Moreover, SMT formulas can contain quantifiers, uninterpreted functions and more types.

SMT Solver is a program that takes SMT formula as an input and checks if formula is satisfiable. In case it is, solver provides model satisfying it as well. If formula is not satisfiable then there does not exists a model that would satisfy the formula. We can think of a formula as the one in the first order logic, for instance: $x > 0 \wedge y > 9 \Rightarrow 6 = x * 2$. In our approach we do not use quantifiers and predicates since we do not need them but in general SMT Solver can handle them.

Even though SMT (and underlying SAT) solving is NP complete problem, recent years brought significant improvements in this area by employing many heuristics, conflict learning and search space pruning. Therefore for sizes of programs we consider SMT solver is far from being the bottleneck.

3 Approach

In this section we are going to first present the problem description followed by the subsection about reaching definitions which are basic block in our solution. Later subsections present core algorithms about program encoding into one large SMT formula together with the examples for easier understanding.

3.1 Problem description

Because of the time constraints of the project we decided to simplify our original problem description which considered input to be a complex java program and output a text file.

First simplification we made is that we will not consider file as our output but rather an assignment to function parameters. The reason for this simplification is that in general we need to know the grammar of the input file in order to map it to program variables. Therefore an input to our program should be java function with 1 to n parameters. They all have to be numbers (integer, double, short etc.). The second input to the program is block of code we want to reach inside the function. We decided for this approach because understanding input grammar is not goal of ours and input in form of parameters is very clear and therefore we can focus on test synthesis rather than input understanding.

Second simplification is that the input function can contain only mathematical operations, assignments and if statements. Loops, fields and function calls are for now not considered. The reason for supporting only primitive types is aliasing which destroys our assumptions that one variable represents unique region of memory. For future extensions we consider function calls which can be handled by extending our approach. Example of a supported program can be seen in the figure 3.

```
int x = 1, y = 2, z = 3, w = 4;
if(x == b)
    z = 2;
else {
    if(b == 2 || c == 2){
        x = 4;
        y = 3;
    } else {
        z = 3;
    }
    w = 1;
}

if(c == y && y > 2){
    w = 8;
} else {
    w = 3;
}

if(x + y + z + w > 10)
  ● error("ex16");
```

Variable	Condition	Value
x	x == b || (x != b && b != 2 && c != 2)	1
x	x != b && (b == 2 || c == 2)	4
z	x == b	2
z	x != b && b != 2 && c != 2	3
z	x != b && (b == 2 || c == 2)	3
y	x != b && (b == 2 || c == 2)	3
y	x == b || (x !=b && b != 2 && c != 2)	2
w	c == y && y > 2	8
w	c != y || (c == y && y <= 2)	3

Figure 3: Example of the supported program together with its conditional definitions. Note that in this case we can get along without SSA but in general such a program is first converted to the SSA form and conditional definitions are done on the top of it. Red dot indicates block of the code for which are conditional definitions created.

The output of our program should be a function call with concrete values whose execution should lead to the execution of a predefined block of code. In the example 3 it can for instance be the call to the error function or some other arbitrary statement. The output of our program for the function above should be assignment: $b = 0, c = 2$. This assignment is one of the many possible assignments leading to the execution of the function error. Our goal is to find one such assignment or say no such assignment exists.

3.2 Conditional Definitions

In order to synthesize an input to a function, we need to understand the structure of a input program. We start by obtaining control flow graph which shows how the flow goes and which values local variables can take. As mentioned in the SSA section we have unique definition for every variable version in a program but we do not know under which conditions such a variable definition hold. Therefore we introduce notion of conditional definitions that enhance well known reaching definitions technique used for instance in constant propagation by adding condition to each definition. We can think of them as a set of tuples $(x, c, v) where$:

$x \in$ **program variables**, $c = c_1 \wedge c_2 \wedge ... c_n$ and $v =$ **expression**. In other words x is a variable that is defined under conjunction of conditions c with value v where v is program expression, i.e. constant, variable, binary or unary expression. Set of tuples with the same variable x is then called conditional definition of the variable x. All such sets represent definition of every variable inside a function except for the function parameters whose definition is not constrained and therefore spans the whole space.

In the following example we have a parameter a and a local variable x. Its definition in read statement is ambiguous since there are exactly two definitions of x and techniques like constant propagation therefore cannot be used. However with conditional definitions we want to understand the data flow and ambiguity is not an issue for us. Following program can be represented in the conditional definitions as $\{(x, a < 0, 1), (x, a >= 0, 0)\}$.

Figure 4: Simple example for conditional definitions

```
int x = 0;
if(a < 0)
    x = 1;
read(x);
```

Conditional Definitions are computed with forward branched flow analysis. It iterates CFG until fix point is reached. In our case only one iteration is needed since we do not support loops. There are 2 important events in the analysis that we need to handle properly. They are:

1. **Merge**
   In merge event we are merging flow from two branches. We can imagine this point to be after every if or if-else statement. Action we need to take in the case of Conditional Definitions is to store both definitions together with their conditions. More formally, let x be the variable to have more than one definition coming, D be the set of conditional definitions and c_1, v_1, c_2, v_2 conditions and values coming from branch 1 and 2. Our action can be then formally defined as: $D := D + +(x, c1, v1) + +(x, c2, v2)$.

2. **Flow through**
   Flow through is an event that is called on every statement. We need to update the flow that is "flowing through" the statement from the previous one by information that current statement provides. In case of our analysis we are interested in 2 types of statements:

- **If Statement** forks the current branch into two. Let CC be the variable that holds "current condition" in every program point. In other words CC is a conjunction (set) of conditions that needs to be met such that current program point can be reached. The action that needs to be taken in case of this statement can be formally defined as: Let a be the condition inside an if statement, then we propagate $CC + +a$ to the branch satisfying the condition a and $CC + +\neg a$ to the other branch. For instance in example 4 CC of the line 1 is $\{True\}$ because this statement will always be executed. However, for the line 3 CC is $\{a < 0\}$ since this condition needs to hold if $x = 1$ wants to be executed. If statement in our example would propagate $\{a < 0\}$ to the branch satisfying the condition and $\{a >= 0\}$ to the one that does not.
- **Phi** statement merges different versions of the same variable. This expression occurs whenever variable was modified in one or more branches. If we encounter such an expression we need to create a new conditional definition for the new version of the variable (variable that is being assigned with the Phi node). More formally, let D be again the set of all conditional definitions, Phi the set of merging variables and x the new version of the variable then the statement would look like $x = Phi(local_1, local_2, ..., local_i)$ and merging algorithm can be defined as follows:

```
for local in Phi:
    for _, c, v in D.select(t|t.first = local):
        D := D ++ (x, c, v)
```

After running the analysis with the rules described above we have conditional definition set for every block in the program.

3.3 Encoding to SMT formulas

In order to execute specific piece of code we need to reach it by satisfying all conditions guarding it. For instance in example 3 we need to satisfy $x + y + z + w > 10$ if we want to execute error function call. Simple idea would be to encode it directly to a SMT formula. However some variables in guarding conditions can be locals which have their own interpretations. They are not completely unknown to us since we know their definitions. Therefore we need to recursively replace all locals in the guarding formulas until they contain only variables that are parameters. If we later create conjuction of all such resolved formulas and provide it to the SMT Solver we would be able to obtain parameter assignment that executes part of the code we want.

It can be observed that final set of guarding conditions can be expressed as a conjunction of conditions taken from all if statements that are on the path to the region of the code we want to reach. We can then process each statement separately and conjoin all the results into one final formula.

We start the resolving process by traversing AST of every expression in the guarding set. For each expression we recursively go down to the leafs first. Leaf node can be local variable, constant or parameter. For each node in the AST we return set of implications to its parent.

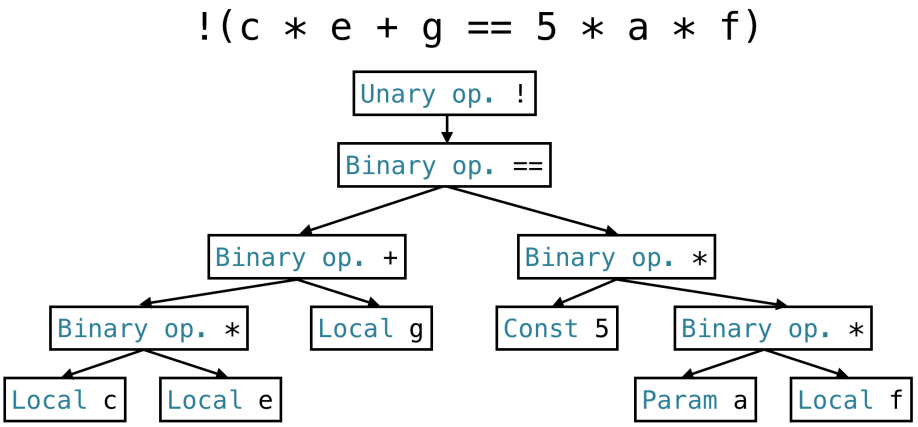

Figure 5: Example of Abstract Syntax Tree for sample expression with 4 local variables c, e, f, g, one parameter a and one constant 5.

For constant node with the constant value c we return $true \Rightarrow c$. In other words we return set of size 1 with one implication where right side is the constant. This is not valid SMT formula but later on it will become valid. For parameter p we apply the same pattern and therefore return $true \Rightarrow p$.

For locals it gets complicated since they have their conditional definitions and they can contain other locals that need to be replaced. Replacing is however not trivial since some local variable can have more than one definition and hence we cannot simply keep replacing variables with their definitions because we need to take into account every definition that can be applied during execution of the program. Therefore we present algorithm 6 for resolving guarding conditions. Usage of SSA is very useful here because every variable is assigned only once and relevant definitions can be therefore obtained in constant time. Remember that variable can be assigned to some expression and hence have one definition or a Phi node in which case it has two or more definitions.

Conditional definitions help a lot here because we know under which condition corresponding definition hold. Furthermore conditional definitions can be think of as conjunction of implications. For instance definition of x in example 2 can be converted to $a < 0 \Rightarrow 1 \wedge a >= 0 \Rightarrow 0$. If we want to make the whole formula $true$ we have to satisfy one of the left sides and therefore corresponding right side as well. Here the conjunction is again not valid formula since the right side of both implications is only a number not a logical expression. However in the merging steps described below we will see how this values will substitute values in Boolean expressions and therefore become valid expressions.

In algorithm 6 we first iterate over all conditional definitions of variable to resolve. For each definition we first create working queue, then iterate until there is nothing to process and in every iteration we pop current implication to resolve from the queue, find first local inside it and if there is no such local i.e. all variables are parameters then we append it to the resulting set. Otherwise we iterate all conditional definitions of the found local and for each one we clone current implication, replace found local with the current definition, append the current condition to the left side of the cloned implication and append such a modified implication to the working queue.

Figure 6: Algorithm for resolving single local variable x. In this algorithm we need to take into account that direct definitions of x can contain values that need to be resolved as well and they can contain other locals so we need to be generic enough to resolve it all the way down to the parameters.

```
def resolve(x: Var) : Set{Implies}
    resolved : Set{Implies}
    for _x, c, v in D where _x = x:
        q : Queue
        q.push(Implies(c, v))
        while not q.empty():
            curr : Implies := q.pop()
            toResolve : Var := find(curr)
            if toResolve = null:
                resolved := resolved ++ curr
            else:
                for x_t, _c, _v in D where toResolve = x_t:
                    curr_clone := clone(curr)
                    // Replaces toResolve with _v in curr_clone
                    replace(curr_clone, toResolve, _v)
                    curr_clone.left := curr_clone.left ++ _c
                    q.push(curr_clone)
    return resolved
```

With this algorithm we always accumulate all preconditions on the left side of a implication and on the right side we always have values or expressions that resolved variable can take if preconditions on the left side are met. We treat set of these implications as their conjunction. In such a conjunction we can never make all left sides of implications false because there would be a contradiction since one expression is always complementary to the conjunction of the others.

3.4 Merging

When leaf node is resolved i.e. set of implications described in previous subsection is created, we need to propagate it to its parent. Parent can be either binary operator or unary operator as can be seen in figure 5. Consequently binary or unary operator can have these leaf nodes as children but it can also have other binary or unary operators. Therefore to be generic enough every method processing some node in the AST returns set of implications representing node converted into a SMT formula.

Merging of two children can therefore occur only in binary operator and there are two sets of implications that needs to be merged into one by binary operator. Formally we can define the merging algorithm for binary operation as in listing 7, where we first collect sets returned by the children and then create Cartesian product of their left sides with binary operation which has children replaced by the right sides of the corresponding implications.

For example, let's assume we want to resolve condition $x > y$ and definition set of x is $\{a < 0 \Rightarrow 1, a \geq 0 \Rightarrow 0\}$ and the one of y is $\{a < 0 \Rightarrow 4, a \geq 0 \Rightarrow -1\}$. Logically corresponding region of code is in example 8.

We can see that we have two definitions for both x and y. Result of merging definitions for

Figure 7: Algorithm for merging two sets of generated implications into the one which represents current binary operation applied on cross product of its two children.

```
def resolve(bop: BinaryOp) : Set{Implies}
    resolved, left, right : Set{Implies}
    left := resolve(bop.left)
    right := resolve(bop.right)
    for l in left:
        for r in right:
            resolved := resolved ++ Implies(l.left ++ r.left,
               Binop(l.right, r.right, bop.type))
    return resolved
```

two local variables in the same expression is conjuction of implications that have on the right side formula with replaced local variables and on the left side conjuction of preconditions corresponding to the definitions that replaced original locals. In case of x and y the result would be: $\{a < 0 \Rightarrow 1 > 4,\ a < 0 \wedge a \geq 0 \Rightarrow 1 > -1,\ a \geq 0 \wedge a < 0 \Rightarrow 0 > 4,\ a \geq 0 \Rightarrow 0 > -1\}$.

Figure 8: Example for demonstration of binary operation resolver.

```
int x = 0, y = -1;
if(a<0){
    x = 1;
    y = 4;
}
if(x > y)
    // region to reach
```

The resulting formula can be provided to the SMT solver. It will return SAT/UNSAT (in our case SAT) together with a model in the SAT case. For formulas above it will be number in range $< 0, \infty)$. The reason for it is that formulas 2 and 3 have false left sides and therefore will become irrelevant for the solver. Remaining formulas have complementary left hand sides so solver has to decide which side it wants to make true. It will decide for the one in the formula 4 since this is the only way how to satisfy the whole formula.

For the unary operation no merging is needed since it has only one child and therefore algorithm 9 is used to deal with an unary operator.

Figure 9: Algorithm for applying unary operator on all children implications.

```
def resolve(bop: UnOp) : Set{Implies}
    resolved, child : Set{Implies}
    child := resolve(bop.op)
    for i in child:
        resolved := resolved ++
            Implies(i.left, Unop(i.right, bop.type))
    return resolved
```

where all we need to do is loop over child's implications and wrap right side with the unary operator.

In order to resolve all guarding conditions we need to apply resolving procedure described above on every individual condition and append all returned implications into one set. Resulting set can be treated as one huge conjunction which needs to be encoded with SMT syntax and provided to the SMT solver.

4 Analysis

In this section we are going to present results we obtained by running our tool on concrete functions.

4.1 Testing

Together with the test synthesis tool we developed automatic testing framework which can automatically run the synthesis, for each test get the output from the SMT Solver, create function call with synthesized parameters, execute it and check for the expected output. We designed our tests such that in every targeted block we output the name of the function the block is in. We have one target block per function and after running analysis we check whether name of every executed function was printed. In case targeted block is unreachable we get UNSAT from the SMT solver and in this case we need to check manually whether the block is really unreachable. The framework enabled us to test the program faster and with the highest possible accuracy.

4.2 Performance

It is very hard to express performance in terms of some program property like lines of code, number of if statements or number of parameters. What makes tests more difficult is "depth" of variable which is how many versions it has together with depth of variables that guard these versions i.e. variables that occur inside if statements. Therefore in our performance table we present only test names together with their execution times and not performance plots since performance is dependant on many factors inside a function which are hard to express. You can also find attached source codes of the tests provided in the table. The whole test set can be found here. We run every test 15 times and report average, standard deviation and 95% confidence intervals.

Func. name	Average	Std. dev.	95% conf. interval
ex13	0.00418843	0.00176147	(0.00321295, 0.00516390)
ex19	0.20716976	0.10732012	(0.14773786, 0.26660166)
lon	0.29144768	0.11125415	(0.22983718, 0.35305818)
deep	0.14047423	0.13014127	(0.06840440, 0.21254406)
superLong$_s$	8.64035899	1.87805459	(7.60032702, 9.68039096)
superLong	–	–	– run out of memory –

It is important to point out that the execution time and memory usage grow exponentially with the number of conditional definitions of variable that can reach the final guarding if

statement. Therefore test "$superLong_s$" which is subset of test "$superLong$ is executed in reasonable amount of time while execution of "$superLong$" terminated with out of memory exception since it reached 2GB memory limit after running for more than 2 minutes. We are aware of some optimization that could reduce size of generated implications in runtime by evaluating their left sides to false (contradicting conditions etc.) and therefore cutting off their further resolving. We propose these optimization as future work since goal of our work was to come up with an correct approach which can be sub-optimal.

5 Loops

As we mentioned at the beginning of the paper, loops are completely omitted in our setting. The reason for it is not only time constrain but also because loops are generally considered to be a big problem in the static analysis field. Therefore we want to discuss in this chapter possible approaches for partial loop support which is not ideal but can work at least in some cases.

The main problem of the loops is that they are in general statically unbounded. In other words, we cannot decide at compile time how many iterations will the loop have. This fact implies that the code region after statically unbounded loop can be reached via infinitely many paths. In case of our test synthesis, we need to find at least one such path vie which the region after the loop can be reached in a state that will further on lead to the execution of the targeted block. However, this problem is undecidable since we can have a non terminating loop before the block and detecting infinite loop statically is generally impossible. Therefore we can only come up with approaches that will succeed in some cases but not for every one.

One possible approach for partially dealing with loops is via loop unrolling. In this approach one needs to unroll the loop until it is left with desired state or some defined threshold is met and we terminate the process as unsuccessful. This approach works well for the loops with small number of iterations since in this case it is feasible to explore them all statically. However, in practice loop bodies are executed many times and loop unrolling becomes ineffective technique. For instance in function $ex1$ it would take more than 1000 loop unrolling iterations to find out that $a \geq 1002$ in order to reach the $error()$ block while in $ex2$ we would need to unroll the loop only four times to find out that $a = 5$ which would be impossible to achieve with the second technique we present.

```
void ex1(int a) {
    int i = 0;
    while(i < a) {
        if(1000 < i && i == a-1)
            error();
        i++;
    }
}

void ex2(int a) {
    int x = 2;
    int i = 1;
    while(i < a) {
        x = i*x; // 2 4 12 48
```

```
        i++;
    }
    if(x == 48)
        error();
}
```

Another approach for partial loop support could be to try encode a loop into a SMT formula as we did it with if and assignment statements. This approach does not work for all loops but at least some fraction of them can be encoded and solved more effectively than with the loop unrolling approach. One loop that we think cannot be encoded into a SMT formula is the one in $ex2$ where we would need to have some recursive definitions on x.

There are three possible relative positions of a targeted basic block to the loop. If the basic block is before the loop we do not need to deal with the loop. If it is inside the loop without any condition guarding it we need to fulfill loop condition at least once. In case it is guarded by some additional condition inside the loop we need to satisfy this condition as well. If the condition contains loop counter, we need to identify its bounds or set of values that the variable can take and create implication by implying the condition with the variable bound. In this case we will treat loop variable as unknown. Lastly, if targeted basic block is after the loop we need to check loop termination and find such state after the loop that will satisfy all conditions guarding the basic block after the loop.

In order to resolve $ex1$ with the second approach, we need to first identify the bounds for i. They are $[0, a)$ i.e. in this case it is an interval. We need to satisfy $i < a$ in order to enter the loop and $1000 < i \wedge i = a - 1$ to satisfy guarding condition inside the loop. The final encoding that would generate model reaching error() function will therefore be $1000 < i \wedge i < a \wedge i = a - 1$.

This approach covers again only limited subset of loops. It assumes that they don't carry some additional state except for the loop counter and furthermore assumes that loop counter exists. Interesting research direction would be to come up with a heuristics based on which we would decide which approach to try. This template based approach would increase types of loops covered but would still not guarantee test generation for any kind of loop.

6 Conclusions

In this paper we presented our approach to the automatic test synthesis based on the program analysis and SMT solving. The tool is able to create function calls that lead to the execution of a specified basic block or say that the basic block is unreachable. Our approach works well in the constrained program settings and we only encountered issues related to the performance and not the correctness. As the future work we therefore propose performance improvements together with the partial loop support mentioned in the previous chapter. The performance of the tool can be enhanced by runtime checking of resolving implications. One approach would be to check for every newly created implication if its left side is false. In case it is we will not append it to the queue since it will not be considered by the SMT solver anyway. Another proposal for performance improvement would be to apply memoization for variable resolution.

Further research dimensions include overlapping memory support which happens in java

when two program variables are aliased and function call support where are many issues related to the recursive calls of the same function as well as indirect recursions.

References

[1] P. Müller, M. Schwerhoff, and A. J. Summers "Viper: A Verification Infrastructure for Permission-Based Reasoning", Verification, Model Checking, and Abstract Interpretation (VMCAI), **41-62**, (2016),

[2] American fuzzy loop, lcamtuf.coredump.cx/afl/historical_notes.txt, visited on 20. 10. 2018

[3] P. Godefroid, M. Y. Levin, D. Molnar, "Automated Whitebox Fuzz Testing"

[4] Á. Einarsson and J. D. Nielsen, "A Survivors Guide to Java Program Analysis with Soot", (2008)

[5] H. Liang, X. Pei, W. Shen, "Fuzzing: State of the Art", IEEE Transactions on reliability, vol. 67, no. 3, (2018)

[6] H. Peng, Y. Shoshitaishvili, M. Payer", T-Fuzz: fuzzing by program transformation", 2018 IEEE Symposium on Security and Privacy