

TODO Add a name

Kevin Chen

February 20, 2026

1 Introduction

TODO insert introduction

2 Symbolic Execution

Symbolic execution is a way of executing a program abstractly, such that on abstract execution covers multiple possible inputs of the program that share a particular execution path through the code [1]. Each input of symbolically executed code is represented as a symbol rather than assigning it a concrete value. When the value is referenced such as for an if statement, the execution path branches, where one path the symbolic value is evaluated as true while the other path the symbolic value is evaluated as false.

One of the key uses for symbolic execution is in testing. Most unit tests, require the programmer to hardcode in values to test, resulting in various unit tests to cover all branches the code can tranverse. On the other hand, a symbolic execution would execute simultaneously on a family of inputs. Consider the following example:

Algorithm 1 Algorithm 1

Require: $x, y \in \mathbb{Z}$

```
1:  $a \leftarrow x + y$ 
2: if  $a \bmod 2 = 0$  then
3:   return error “ $a$  is even”
4: else
5:   return  $a$ 
6: end if
```

With a glance, we can see that using any combination of numbers such that their sum results in an even number will result in an error. There is no way to cover all cases of failure using unit tests, as there are an infinite number of way which we can stumble onto this error.

Instead of testing with unit tests such as $x = 1, y = 1$, symbolic execution represents the inputs as symbols $x = x_0, y = y_0$. Whenever, the symbols are required to return a value, such as in a branch, symbolic execution splits into two executions, with each execution recording the branching condition in its execution path. Consider a symbolic execution of the algorithm above. We start with representing each input symbolically:

line	Path Condition	Translation
0	true	$x \rightarrow x_0, y \rightarrow y_0$
1	true	$x \rightarrow x_0, y \rightarrow y_0, a \rightarrow x_0 + y_0$
2	$(x_0 + y_0) \bmod 2 = 0$	$x \rightarrow x_0, y \rightarrow y_0, a \rightarrow x_0 + y_0$
2	$\neg((x_0 + y_0) \bmod 2 = 0)$	$x \rightarrow x_0, y \rightarrow y_0, a \rightarrow x_0 + y_0$
3	$(x_0 + y_0) \bmod 2 = 0$	Error
5	$\neg((x_0 + y_0) \bmod 2 = 0)$	Return $x_0 + y_0$

Figure 1: Symbolic execution trace showing path conditions and variable translations

In this example, we see that the inputs values start symbolic as symbols which has no concrete value. Following the execution, we translate every instance of x with x_0 with the same case for y . Where it gets interesting is when the the execution has a branching path. In the **if** statement, the value of a is required to be assigned a value, which breaks the symbolic abstraction. Instead, the symbolic execution branches into two different executions, each of which assuming a different condition to satisfy the branching statement. Below we can visualize the different execution paths the execution takes via a simple tree (see Figure 2):

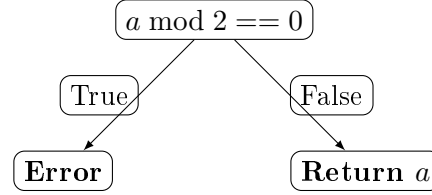


Figure 2: Execution tree showing the branching paths in symbolic execution

In each paths, we accumulate the different path conditions until we either reach the end (return statement) or a failure (error). In this case, we find that there exists a set of inputs (when $x_0 + y_0 \bmod 2 = 0$), which causes an error. Now imagine a different example; using the same coding example, but with a twist. Prior to running this program with symbolic execution, one can add constraints to the path. Below, an example where we start with a given constraint that $x_0 + y_0 = 1$:

line	Path Condition	Translation
0	$x_0 + y_0 = 1$	$x \rightarrow x_0, y \rightarrow y_0$
1	$x_0 + y_0 = 1$	$x \rightarrow x_0, y \rightarrow y_0, a \rightarrow x_0 + y_0$
2	$\neg((x_0 + y_0) \bmod 2 = 0)$	$x \rightarrow x_0, y \rightarrow y_0, a \rightarrow x_0 + y_0$
2	$x_0 + y_0 = 1 \wedge (x_0 + y_0) \bmod 2 = 0$	$x \rightarrow x_0, y \rightarrow y_0, a \rightarrow x_0 + y_0$
3	False	Error
5	$\neg((x_0 + y_0) \bmod 2 = 0)$	Return $x_0 + y_0$

Figure 3: Modified version of execution with existing constraints

With the constraint, the path condition which leads to the error is contradictory. This illustrates that the path is impossible. With this constraint, we see that no path leads to any errors. In industry, programmers, can create tests with given preconditions (constraints) and run until the program terminates. Instead of throwing an error, we can use assertion statements as post-conditions. If it terminates without an error, the program is robust across all set of inputs. Otherwise, a symbolically executed program will return a concrete set of conditions which will give an error or violate a post-condition.

2.1 While loops

To represent a while loop in symbolic execution, we unroll it into a recursive function with no loops. Take the following code for instance:

Algorithm 2 Count Number of 1's in a Bit Vector

Require: v is a bit vector of length n

```
1:  $count \leftarrow 0$ 
2:  $i \leftarrow 0$ 
3: while  $i < n$  do
4:   if  $v[i] == 1$  then
5:      $count \leftarrow count + 1$ 
6:   end if
7:    $i \leftarrow i + 1$ 
8: end while
9: return  $count$ 
```

We can rewrite it as a recursive function:

Algorithm 3 Recursively Count Number of 1's in a Bit Vector

Require: v is a bit vector of length n

```
1: function COUNT_ONES( $v, i, n$ )
2:   if  $i == n$  then
3:     return 0
4:   else if  $v[i] == 1$  then
5:     return  $1 + \text{COUNT\_ONES}(v, i + 1, n)$ 
6:   else
7:     return  $\text{COUNT\_ONES}(v, i + 1, n)$ 
8:   end if
9: end function
10: return  $\text{COUNT\_ONES}(v, 0, n)$ 
```

Now that we have rewritten the while loop as if statements, we simply create a new branch for every if statement. In practice, this may lead to state explosion, resulting in thousands of branches in a just a few iterations. Programmers either set a recursion limit, terminating the branch which gets stuck, or find a different way to represent the while loop, using the loop invariant. Below is a tree of the branching paths created from a symbolic execution of Algorithm 3.

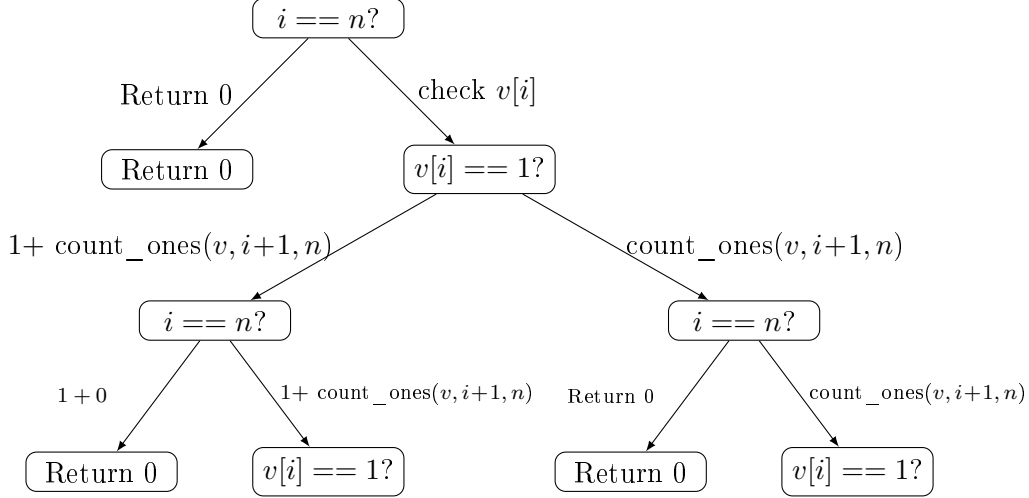


Figure 4: Symbolic execution tree for the recursive `count_ones` algorithm showing branching at each condition

3 Hardware Language

4 Patronus

4.1 Expressions

Patronus[2] represents expressions under the class *Expr* and *ExprRef*. Each expression is an object which contains the type of expression and a list of *ExprRef* to other expressions. For example, the expression `if(x == 4, y + = 3, z + = 4)` can be represented as `ite(ExprRef[1], ExprRef[2], ExprRef[3])`. The expressions are stored all in an Context.

5 Implementation

5.1 Symbolic Execution

One of the key challenges of implementing symbolic execution is being able to distinguish whether or not a path is feasible or not. Consider the execution from Figure 3 row 3. The question are we suppose to tell that this path represents is infeasible?

As always in computer science, one of the best ways to model something is to make it someone else's problem; take an answer to an existing hard solution and leverage it to solve your problem. Here, we leverage a SAT solver to answer these ambiguous questions for us. Now, all we need to do is translate each branching statement into a series of binary equations.

We represent integers as bit-vectors of their binary representation. A standard 32-bit integer would instead be represented as size 32 vector of booleans.

We represent the state of an symbolic execution with the following tripe: (P, M, C) , where P holds the path conditions, M is the mapping which translates variables to symbols, and C is the current command we are processing. Suppose that the command we are trying to process is an if statement with the form $C(c, t, f)$, where c is the condition, t is the return value if the condition is true, and false being the return value if the condition is false. To update the state, we run the following queries in the SMT Solver:

1. $P \wedge c = SAT \wedge P \wedge \neg c = SAT$: The SMT solver says that both paths are feasible. In this case, we create two branching executions one with $P' = P \wedge c$ and one with $P' = P \wedge \neg c$.
2. $P \wedge c = SAT \wedge P \wedge \neg c = UNSAT$: The SMT solver says that the path such that $\neg c$ is impossible. This means that given the information of P , $\neg c$ is infeasible. As such, we do not need to add $\neg c$ into the path, as the information is already recorded in the path. You can think of this as a node with one branch. It has no other choice so no branching executions are created. Consider $P = (x == 10) \wedge (y == 11), c = (x + y > 12)$. There is no need to add c to the execution path because the value is already included in P .
3. $P \wedge c = UNSAT \wedge P \wedge \neg c = SAT$: The SMT solver says that the path such that c is impossible. Similar to the previous case, we do not need to add c onto the path as the information for deducing $\neg c$ is already included in P .
4. $P \wedge c = UNSAT \wedge P \wedge \neg c = UNSAT$: The result states that regardless of c the execution is infeasible. This must mean that P is infeasible, which means the constraints were infeasible.

5.2 Initialize

You start in the initiation step. Each state variable will start with an symbolic value. However, if it has an initial value, the state variable will be assigned to that instead. For simplicity, values assigned at each timestep will have a timestep added as part of its symbol. This is to distinguish between symbols added and important for identifying different symbols for the input.

5.3 Simplification

The goal of simplification is to have values as simplest terms. This means that each expression must contain basic terms, such as numbers or symbols, things that cannot be further simplified.

Assume you have all the basic rules for simplification. One example is if you add two integers, the result should just be the sum of the integers. Another is if you have an if statement and the condition is true, then it should simplify to just the true branch.

To simplify expressions and expression references, we need to start by simplifying the inner most layer. This is implemented with a stack. Each child of an expression gets added to a stack and the last children added is popped and simplified. Table 5 traces this algorithm on the expression $If(true, If(add(0, 1), 4, 5), -1)$: the first column is the step (number of pops), the second is the current result overall (the full expression with all completed simplifications applied so far), the third is the current simplification target (the expression just popped), and the fourth is the stack (top to bottom) after pushing that target's children, if any. The expression simplifies to 4: -1 , 4 , and 5 do not simplify; $add(0, 1)$ simplifies to 1 ; $If(1, 4, 5)$ simplifies to 4 (condition is true); $If(true, 4, -1)$ simplifies to 4 .

Step	Current result overall	Current simplification target	Stack
0	$If(true, If(add(0, 1), 4, 5), -1)$	$If(true, If(add(0, 1), 4, 5), -1) \rightarrow ?$	$true, If(add(0, 1), 4, 5), -1$
1	$If(true, If(add(0, 1), 4, 5), -1)$	$-1 \rightarrow -1$	$true, If(add(0, 1), 4, 5)$
2	$If(true, If(add(0, 1), 4, 5), -1)$	$If(add(0, 1), 4, 5) \rightarrow ?$	$true, add(0, 1), 4, 5$
3	$If(true, If(add(0, 1), 4, 5), -1)$	$5 \rightarrow 5$	$true, add(0, 1), 4$
4	$If(true, If(add(0, 1), 4, 5), -1)$	$4 \rightarrow 4$	$true, add(0, 1)$
5	$If(true, If(1, 4, 5), -1)$	$add(0, 1) \rightarrow 1$	$true, If(1, 4, 5)$
6	$If(true, 4, -1)$	$If(1, 4, 5) \rightarrow 4$	$true$
7	4	$true \rightarrow true$	\emptyset

Figure 5: Stack-based simplification trace for $If(true, If(add(0, 1), 4, 5), -1)$. At each step the target is popped and its children (if any) are pushed. -1 , 4 , 5 , 0 , 1 , and $true$ do not simplify; $add(0, 1) \rightarrow 1$, then $If(1, 4, 5) \rightarrow 4$, then $If(true, 4, -1) \rightarrow 4$. Result: 4 .

5.4 Finding nested if-statements

Using the idea for simplification, the same method can be used to traverse through nested if-statement. Start by implementing a stack which holds all the expressions, a stack which holds all the traversed paths, and a map which holds the position of each expression reference. For each expression, add each child to the mapping, and then explore each children. If any of the children are an ITE expression, we should simplify the path to either the true or false expression and add this to the path stack. Then, with the assumption that the ite is true or false, we can use simplification to simplify the expression and add it to the expression stack. Do this for each path as well and eventually we will have a list of paths which represent all feasible combinations of paths and, for each path, the new states.

Consider the expression $add(ite(x, 1, 2), ite(y, 2, 3))$ where x and y are unconstrained symbolic booleans. There are two nested ITE sub-expressions, one inside each argument of add , producing four feasible execution paths. Figure 6 shows the resulting path-exploration tree.

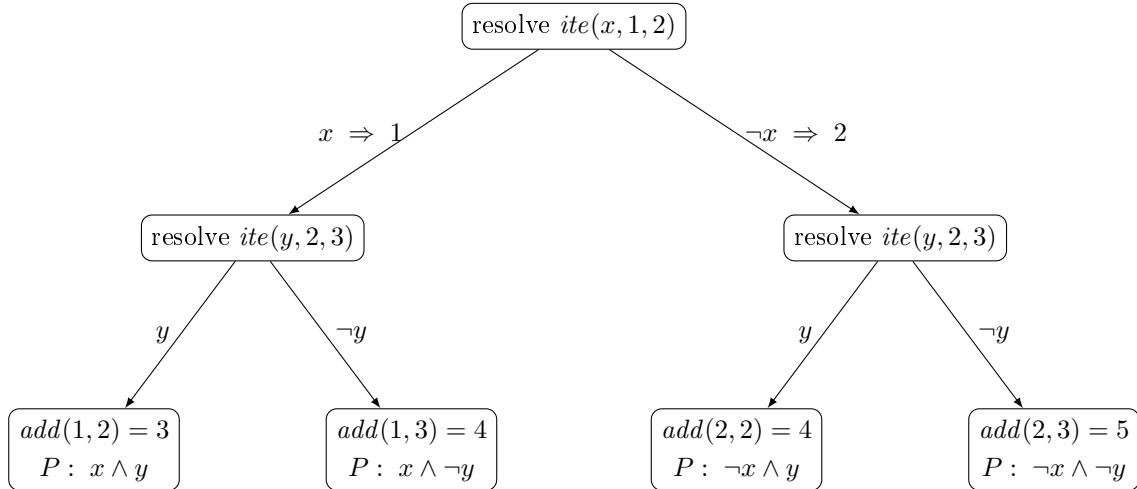


Figure 6: Path-exploration tree for $add(ite(x, 1, 2), ite(y, 2, 3))$. Each internal node shows an ITE being resolved; edges are labelled with the branch condition and its concrete value. Leaves show the fully resolved result and the accumulated path conditions P .

6 Conclusion

References

- [1] Jonathan Aldrich. Symbolic execution. Lecture notes 14, Course 17-355, Carnegie Mellon University, 2018. Accessed: 2025.
- [2] Kevin Laeuer. Patronus: work in progress, playing around with btor2 in rust. GitHub repository, 2024. Accessed: 2025.