# **Comparing Different Types of Symbolic Execution Techniques**

## **Abstract**

Symbolic execution has come a long way since its very introduction. Today, we have many sub-categories of symbolic execution techniques aimed at overcoming the challenges and difficulties associated with Static Symbolic Execution (SSE). In this report, we discuss these challenges and how 4 types of symbolic execution techniques, namely – Static Symbolic Execution (SSE), Selective Symbolic Execution (SeSE), Dynamic Symbolic Execution (DSE) – Breadth-first + Depth-first style, and Backward Symbolic Execution (BSE) compare against each other. To complement this discussion, we have partly implemented SeSE and Depth-first style DSE with WLANG as the artifact, using the same structure we had used for creating a Static Symbolic Execution Engine in one of the assignments. In the process, we also discuss the design principles, examples and applications of each of these techniques.

## **Introduction**

Symbolic execution is one of the most widely used techniques to aid software testing. It helps generate test cases for feasible paths in the program, so that useful program paths can be tested instead of just random testing.[4] Symbolic execution traverses the program in consideration, in terms of symbolic expressions, so that multiple feasible paths can be found and respective test cases or satisfying assignments generated.

**Why Symbolic Execution?** Symbolic execution was introduced in the mid 70’s[2] to combat the challenges associated with finding good and meaningful test cases for programs under test. Even though we have other techniques such as random testing, fuzzing, etc; majority of those techniques tend to under-approximate the feasible program properties [2]. Also, some of those techniques, tend to find cases which are either infeasible or take the same execution path as the other test cases (i.e., path duplication). We further discuss how symbolic execution helps overcome these problems so that each new test case is guaranteed to cover a unique path.

**Key Idea** The main characteristic of symbolic execution is that it maintains the explored path in terms of symbolic expressions, such that all feasible combinations of non-concretized variables can be taken into consideration when deciding whether a particular path can be taken, instead of deciding on the basis of random concrete value for the variables, as is the case with most other random testing techniques. A state in symbolic execution is maintained as a tuple – [2]

either **<path-condition, symbolic-memory-mapping>**

or **<path-condition, symbolic-memory, concrete-memory-mapping>**

where ‘path-condition’ is a first-order Boolean formula that checks for path satisfiability and is updated each time a branch-split (if-then-else, while-do, case, etc), assertion or assumption is encountered. ‘Symbolic-memory-mapping’ maps each defined variable to its symbolic counterpart. Similarly, ‘Concrete-memory-mapping’ maps each defined variable to a concrete assignment, such that the assignments satisfy the path-condition for that state.

Checking path satisfiability and finding the corresponding model (i.e., the satisfying assignment) is mostly done using a ‘Satisfiability-Modulo-Theory (SMT)’ solver. One of the most popular ones in use today in markets is z3-solver by Microsoft [6]

**First-hand results** Overall, performing symbolic execution on a program, results in an acyclic directed control flow graph, such that each of the leaf nodes have a satisfiable path condition - with a model, to be used as one of the test cases. [4]

We further discuss the many design principles, associated challenges, search strategies, types of symbolic execution engines, etc; in detail. Also, a small implementation of 3 symbolic execution engines can be found in the project directory, the results of which have been discussed at the end of this report.

## **How Symbolic Execution finds bugs? [7]**

As discussed in the introduction, Symbolic Execution traverses feasible paths and generates respective satisfying assignments which can be utilized as test cases. When the test suite containing these cases is run, the bugs that exist in the respective path are triggered and hence the hence recorded in the test results. [7] These are bugs that are found as a result of testing performed, after symbolic execution.

However, certain bugs can be directly found while we are performing symbolic execution. This requires some extra effort. Some oracles such as assertions are compiled into conditionals; such that, if the path condition and assertions are together VALID, then there are no bugs. Otherwise, even if there is one case, where the combined boolean formula may not hold; then it can be treated as a potential bug in the program. [7]

In general, ‘**assert (condition)’** can be expressed as **‘if not(condition) then raiseError else proceed’** [7]

The assertions can be used for a safety property such as divide by zero, array index out of bounds, nullPointers, etc.; where the implementation can be explicit (provide assertions or checks before the safe operation) or implicit (added at runtime by the symbolic execution engine – which requires the engine to be created/implemented with that aspect in mind.) [7]. Note that, not all execution engines may support the implicit implementation.

## **Properties of Symbolic Execution [2]**

Any version of Symbolic execution tends to share the following common properties:

**1] Progress:** For every CPU resource and second of time that is consumed by the Symbolic Execution Engine, is justified by progress in terms of – new paths, update to memory stores, removal of infeasible paths or reporting assertion failure. [2]

**2] Uniqueness:** Every new path that is explored should be different from the rest of the path that have been explored to that point.

**3] Symbolic update:** Symbolic expressions are updated (if applicable) at each step of the program that changes the state; irrespective of whether that part of the program is explored concretely or symbolically.

**4] Maintain History in Path Condition: ‘**Path condition’ at each new step in the execution, must contain all the constraints that had been explored to reach that node.

## **Common Challenges with Symbolic Execution**

Symbolic execution is a very exhaustive process; and even though it produces very good results for the sample programs that we discuss in this report, the same may not hold for programs of bigger size. Following are some of the common challenges faced with all types of symbolic execution techniques, when used with industry and other real-world applications

**1] Memory Aliasing:** [5]

Having multiple reference variables pointing to the same address in memory, is something that can be very difficult for an SMT solver to manage. As we had already discussed, SE uses symbolic expressions for the variables and mostly does not take dynamic memory allocation into account. [5]

**2] Collections:** [5]

Real-world applications tend to use a lot of collections as arrays, lists, maps, sets, etc. In such cases, the Symbolic Execution Engine has to make a choice between using a single v/s multiple expressions - to reference the elements of the collection. And a single collection element can be only be referenced dynamically for a concrete iterator [5].

**3] Constraint Satisfiability:** [2]

Satisfiability is an NP complete problem. Hence, the time taken by an SMT solver at each branch point, to determine satisfiability, could be exponential.

**4] Environment:** [5]

Most real-world applications, specially IoT ones, tend to have a hight amount of interaction with the environment or operating system. In such cases, the SE can run into consistency issues (for example, when a system call is made), as SE would no longer have control over the execution. [5]

These challenges are addressed differently in different situations and depend a lot on the context it is used in. While addressing them, SE may have to give up on completeness; but proper assumptions, filters, etc (specific to the context) can be figured out to ensure that the best possible subset of feasible paths are returned by the symbolic execution engine – for the specified resource and time constraints.

## **Types of Symbolic Execution Engines in focus:**

### **1] Static Symbolic Execution (SSE)**

Static symbolic execution is the same as Classic Symbolic execution that we discussed in the introduction. The only point to consider is that the state is a tuple of the first kind that we had discussed:

**<path-condition, symbolic-memory-mapping>**

* **Algorithm:**

|  |
| --- |
| **static\_symbolic\_execution (current\_node, curr \_state):**  **Input:** **current\_node:** Abstract Syntax Tree of the program/block under test  **curr\_state:** Current state in the form of a tuple (path\_condition, symbolic\_environment).  Initially (in the first recursive call) both are empty/NULL.  **Output:** List of feasible states    **Algorithm:**  list\_feasible\_states = **perform\_node\_specific\_execution** (current\_node, curr\_state, selected\_ast\_node, execution\_type)  if current\_node.has\_next():  new\_states = [ ]  for state in list\_ feasible \_states:  results = **static\_symbolic\_execution** (current\_node -> next(), curr\_state, selected\_ast\_node, execution\_type)  new\_states.extend(results)    return new\_states  else:  return list\_feasible\_states |
| Here, the method ‘**perform\_node\_specific\_execution()’** performs the execution for that specific node (if-then-else, while, assert, assume, assignment, etc) and may also involve a call to ‘**static\_symbolic\_execution()’** (if the specific node involves a block execution). |
| * The implementation of this algorithm can be found in **‘wlang/SSE\_v1.py’.** (and is the same implementation from Assignment 2. We use it here only for comparison against the implementations of Selective Symbolic Execution (SeSE) and Dynamic Symbolic Execution (DSE) engines, which have been freshly implemented). * The examples/tests discussed in this report can be found in **‘wlang/test\_SSE\_DSE\_SeSE.py’.** |

* **Benefits:**

a] Static symbolic execution is sound: [2]

- By soundness, we mean that all feasible unsafe assignments are guaranteed to be found using symbolic execution; another way to put it is – ‘SE prevents false negatives’ [2].

b] SSE is complete: [2]

- By completeness, we mean that the assignments, which were deemed unsafe, will actually be realised as so with an error when the assignment is used in a test case; i.e. ‘SE prevents false positives’. [2]

* **Drawbacks:**

1] Path Explosion: [2][5]

Real world and industry applications have large amounts of code, and symbolically enumerating each feasible path is not practical most of the times, considering the amount of CPU resources & time at hand

* The number of paths grow exponentially when each branch point is reached.
* Some loops whose iteration depends on inputs to the program, may be caught in an infinite loop.
* Procedures, parallel behaviour, etc – all add to the complexity. [7]
* **Potential Solutions:** Selective Symbolic Execution

2] Complex Operations or code: [7]

Operations or code which involve function pointers, hash functions, system calls, etc; can be very hard to execute symbolically [7]. A path condition involving such operations is very difficult for an SMT solver to check

* **Potential Solutions:** Dynamic Symbolic Execution

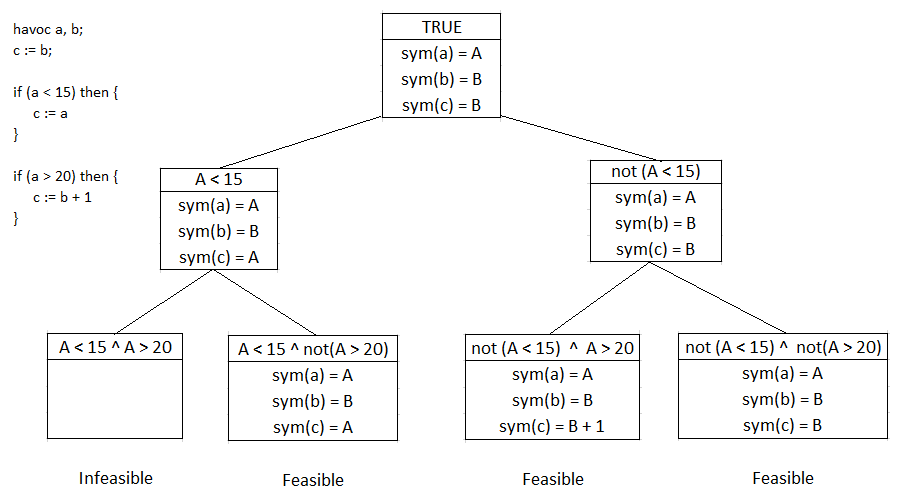
3] Constraint Satisfiability: [2]

* Satisfiability is an NP complete problem. Hence, the time taken by an SMT solver at each branch point, to determine satisfiability, could be exponential.
* **Applications:**

1] Ideal for small program, which achieve baseline functionality.

2] Ideal in situations, where you have different small modules in an application with very low dependency.

* **Example:**



### **2] Dynamic Symbolic Execution (DSE) [1]**

Dynamic Symbolic Execution is an approach based on concolic execution, which combines both symbolic execution and concrete execution to help guide the engine, when the branch conditions involve operations that are difficult or practically not possible for an SMT solver to check against. Another way to put it is – ‘Concrete execution drives Symbolic Execution’ **[2]**.

A state in Dynamic Symbolic Execution takes the following form:

**<path-condition, symbolic-environment, concrete-environment>**

Every time a branch condition is reached, one of the branch is executed concretely by checking if the path condition holds for concrete values of variables. This technique was introduced to overcome the problems associated with static symbolic execution with respect to complex operations such as a hash function. It also helped avoid using the solver for 1 branch condition (although it may lead to incompleteness).

Today, many different execution engines follow a dynamic approach with different search strategies, with Depth-first search being the most commonly used one.

Following are the search strategies that can or are employed in Dynamic Symbolic Execution:

**a] Depth-first Style Dynamic Symbolic Execution:** [1]

Depth first search (DFS) was one of the first search strategies to be used in Dynamic Symbolic Execution. This strategy makes use of a memory-based model, to track back to the last branch condition. The general steps followed in DFS-based Dynamic Symbolic Execution are as follows:

1. Get or find concrete environment values.
2. Concretely keep executing one of the branches, every time a branch condition is reached. Keep doing so, until either – both the branch conditions are infeasible (with concrete values) or until the leaf node has been reached.
3. Check the last visited branch in memory.
4. If the other branch condition involves any complex operations, then compute and replace them concretely in the branch condition. Also, concretize all variables involved in performing the complex operations.
5. Symbolically check if Path-condition (PC) is SATISFIABLE, using an SMT solver.
6. If PC is SATIFIABLE, then execute the node, else go to step (ii).
7. Repeat steps (ii) to (iv) until no more branches are left to be covered.

**Note:** the symbolic environment and path condition is always updated with symbolic expressions at each step, irrespective of whether the branch was executed concretely or symbolically.

* **Example**
* Covered in the Section (……..) of implementation
* **Benefits:**

a] Helps solve the issue associated with Complex operations. This makes constraint solving easier for that branch.

b] The no. of paths no longer increase in exponential size. Hence, helps reduce path explosion to some extent.

* **Drawbacks:**

a] Loses completeness: May not be able to cover all feasible paths.

b] May get trapped in loops, where the condition does not rely on symbolic inputs. [1]

c] Size of constraint also increases with growth in execution path, and can potentially make the job of constraint solver harder.

* **Applications**

a] Ideal for security applications, which involve a lot of complex cryptographic operations.

* **Examples of real Engines in Industry**

a] DART (Directed Automated Random Testing) [8]

b] CUTE (Concolic Unit Testing Engine) [9]

**b] Breadth-first Style Dynamic Symbolic Execution:** [1]

Breadth first search (BFS), although not as widely used as DFS, is till utilized by some symbolic execution engines. In BFS, the concrete and symbolic branches are executed in parallel, instead of entirely exploring a particular branch (as in DFS).

The general steps followed in BFS-based Dynamic Symbolic Execution are as follows:

1. Get or find concrete environment values.
2. Concretely execute one branch, whichever is feasible with concrete values.
3. If the other branch condition involves any complex operations, then compute and replace them concretely in the branch condition. Also, concretize all variables involved in performing the complex operations.
4. In parallel, symbolically execute the other branch.

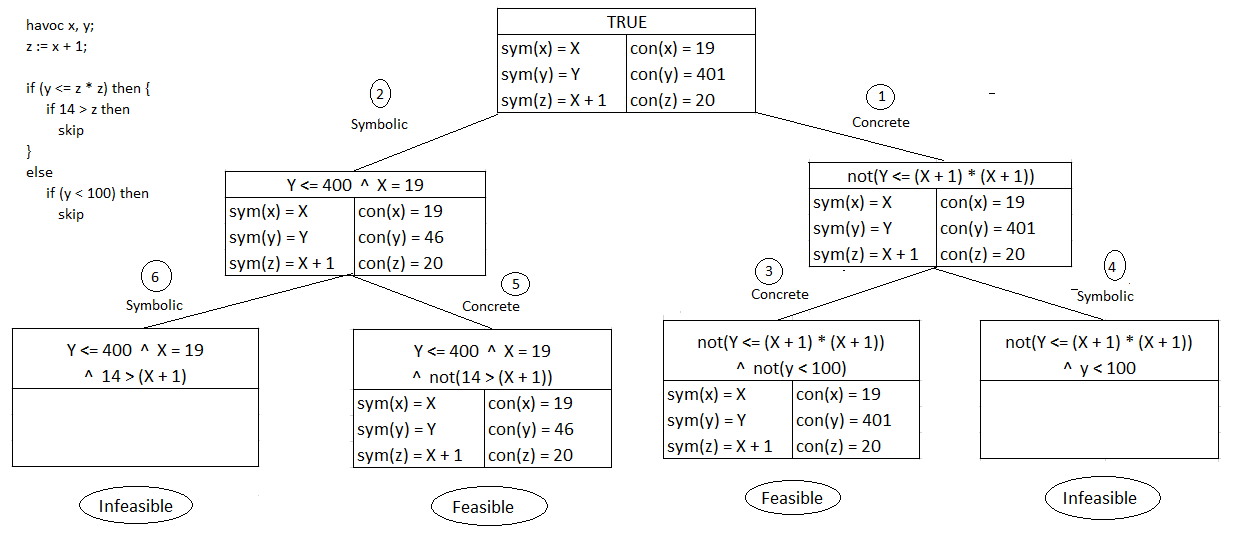
- Symbolically check if Path-condition (PC) is SATISFIABLE, using an SMT solver.

- If PC is SATIFIABLE, then execute the node, else treat the path as infeasible (hence, not explored any further).

1. Repeat steps (ii) to (iv) for all the feasible paths in memory – and execute them all in parallel

**Note:** the symbolic environment and path condition is always updated with symbolic expressions at each step, irrespective of whether the branch was executed concretely or symbolically.

* **Example**



* **Benefits:**

a] Helps solve the issue associated with Complex operations.

b] Can use the multiprocessor capability of modern systems to traverse each path in parallel.

* **Drawbacks:**

a] Loses completeness: May not be able to cover all feasible paths.

b] Memory Utilization is poor: Since DSE has to keep switching the states. [1]

c] Exploration space of the program grows quickly. [2]

* **Applications**

a] Useful for security applications, which involve a lot of complex cryptographic operations.

* **Examples of real Engines in Industry**

a] EXE [10]

b] KLEE [11]

**c] Heuristic Search Strategies:** [1]

DFS and BFS are classical search strategies. Heuristic search strategies, instead, do the following: [1]

i] Obtains information about the context.

ii] Selects the coverage metric.

iii] Computes the weight associated with each feasible path.

Some of the widely used Heuristic Search strategies have been compared in the below table:

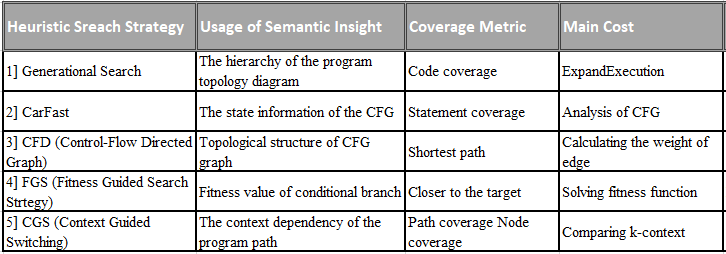


Table 1: Heuristic Search Strategies [1]

### **3] Selective Symbolic Execution (SeSE) [2][3]**

Sometimes, a developer or team may want to test only part of the program. In such cases, it’s not useful to run symbolic execution on the entire program. All of the Symbolic execution techniques which we have discussed so far, do not allow us this option. What we want in this case, is a SE engine which takes input from the developer, as to which part of the program he/she intends to perform testing on. And depending on this input, the engine performs symbolic execution on only the specified part of the program and return feasible test cases for the respective paths that have been found. The rest of the program can traversed concretely.

A state in Selective Symbolic Execution takes the following form:

**<path-condition, symbolic-environment, concrete-environment>**

SSE creates an illusion of full symbolic execution [3]. An important factor to consider here is – “once we encounter and finish performing symbolic execution on the module or part of interest, do we continue symbolically executing the rest of the program, or should we switch back to concrete execution?”

We will consider both the cases:

a] The former (**Version 1**) makes sense as the code encountered after the part in contention, would have also been affected by whatever changes were made to that part (perhaps as part of maintenance or update). A proper way to implement this case would be as follows:

1. Take concrete input and module/part of interest from the developer/user.
2. Run SeSE engine concretely till the module\_of\_interest has been encountered.
3. Once module\_of\_interest is encountered, execute the rest of the program symbolically.

b] The latter (**Version 2**) also makes sense in some cases, where the code in contention is meant for specific functionality and has no affect on the rest of the program. A proper way to implement this case would be as follows:

1. Take concrete input and module/part of interest from the developer/user.
2. Run SeSE engine concretely till the module\_of\_interest has been encountered.
3. Once module\_of\_interest is encountered, execute the entire module\_of\_interest symbolically.
4. Once the module\_of\_interest has been fully traversed, execute the rest of the program concretely.

**Note:** a] In the latter’s case, switching back and forth between symbolic and concrete execution can be a challenging task. We have successfully implemented both the cases with WLANG as he artifact. The results of our implementation have been discussed in detail in section (……….).

b] In Version 1 - the symbolic environment and path condition is always updated with symbolic expressions at each step, irrespective of whether the branch was executed concretely or symbolically. However, in Version 2, we can stop maintaining the symbolic environment, once the module\_of\_interest has been fully traversed.

Another major challenge in both the cases is – “how to provide the module/part/code of interest”. There are many different ways to do this. One of the most effective one is to provide the node in AST (Abstract Syntax Tree), corresponding to that module. And this is exactly the approach that we have followed in our implementations of the 2 versions.

* **Examples**

Examples can be found in section (……)

* **Benefits**

a] Helps reduce the issues associated with path explosion.

b] Time associated with constrained solving in now reduced.

* **Drawbacks:**

a] Since the part before module\_of\_interest is executed concretely, there is a strong possibility that the module may never be reached. (This can be taken care of, if the developer provides good concrete values instead of using random ones.)

### **4] Symbolic Backward Execution (SBE) [2]**

Symbolic Backward Execution, as the name suggests, is a reverse implementation of the traditional Dynamic Symbolic Execution (DSE). The traversal begins from the leaf node (which is the target) and the constraints of the path condition are collected along branches, in the reverse direction. In SBE, many paths can be traversed at a time. Feasibility is checked periodically, such that unsatisfiable paths are discarded and SBE backtracks. [2]

**Requirements of SBE:** Inter-procedural control-flow graph should be made available so that the backward traversal is possible [2]. But even then, it is very difficult to put this into practice.

* **Examples**
* **Benefits:**

a] Can help find a test case that can trigger a specific line in the program. [2]

b] Many paths can be traversed in parallel. [2]

* **Drawbacks:**

a] Very difficult to put into practice.

b] Can be very expensive. [2]

## **Small Implementations of 2 Symbolic Execution Engines in WLANG:**

1. **Selective Symbolic Execution Engine (SeSE)**

* **Introduction**

For implementing this technique, as discussed in the above sections, we are required to provide a reference to the block or code that we wish to execute symbolically. We have used WLANG as our artifact and provide the AST node as the reference to the selected block.

We have implemented 2 versions of the Selective Symbolic Execution techniques:

**a] Version 1:** **Concretely executes till the selected block is reached. Symbolically executes the entire code, from the point where the selected block begins.**

- As we had learnt earlier, Selective Symbolic Execution can be used to cover only the part that has been changed in the most recent update.  
- Hence, it is possible that the rest of the execution is also affected due to the changed block. Hence, we symbolically executed the rest of the code, once the selected node or block has been reached.

**b] Version 2:** **Symbolically executes only the block or node that was selected. Rest of the code, before and after the selected block is executed concretely.**

**-** We often come across changes or updates in code, which do not have a great deal of effect on the rest of the implementation. Also, sometimes it might be computationally and/or financially very expensive to symbolically execute the code after the selected/updated block.

**-** Hence, in such cases, it is ideal to symbolically execute only the node that has been selected; and leave the rest to be implemented concretely by the CPU (Central Processing Unit) concretely.

* **Pre-set**
* We are using WLANG as the artifact
* Selection of block of code to be tested, is done by reference to ‘AST node’ of the corresponding block.
* To simplify the implementation, loop invariants have been supressed.
* A state is a (path\_condition, symbolic\_environment, concolic\_environment)

where: **i] path\_condition:** List of contraints in symbolic form

**ii] symbolic\_environment:** A dictionary which maps each defined variable to its symbolic value.

**iii] concrete\_environment:** A dictionary which maps each defined variable to its concrete value.

* **Algorithms (Version 1 v/s Version 2)**

**a] Version 1:**

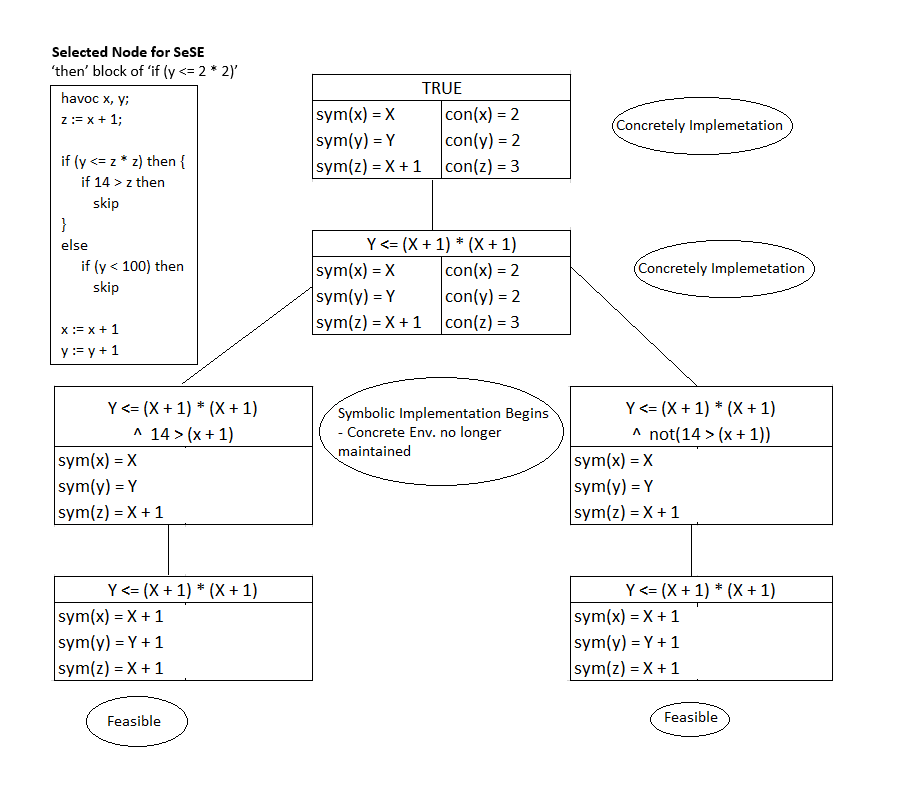
|  |
| --- |
| **version1\_selective\_symbolic\_execution (current\_node, curr \_state, selected\_ast\_node, execution\_type):**  **Input:** **current\_node:** Abstract Syntax Tree of the program/block under test  **curr\_state:** Current state in the form of a tuple (path\_condition, symbolic\_environment,  concolic\_environment). Initially (in the first recursive call) all 3 are empty/NULL.  **selected\_ast\_node:** Node to be executed symbolically  **execution\_type:** Determines whether the implementation should proceed symbolically or concretely.  Initially (in the first recursive call) the value is ‘concrete’.  **Output:** List of feasible states    **Algorithm:**  if reference (current\_node) == reference (selected\_ast\_node); then:  execution\_type = ‘symbolic’  list\_feasible\_states = **perform\_node\_specific\_execution** (current\_node, curr\_state, selected\_ast\_node, execution\_type)  if current\_node.has\_next():  new\_states = [ ]  for state in list\_ feasible \_states:  results = **version1\_selective\_symbolic\_execution** (current\_node -> next(), curr\_state, selected\_ast\_node, execution\_type)  new\_states.extend(results)    return new\_states  else:  return list\_feasible\_states |
| Here, the method ‘**perform\_node\_specific\_execution()’** performs the execution (symbolic or concrete – as specified in the argument) and my also involve a call to ‘**selective\_symbolic\_execution()’.**  Note here, that once we find the ‘selected\_node’, the rest of the execution proceeds symbolically. |
| * The implementation of this algorithm can be found in **‘wlang/SeSE\_v1.py’.** * The examples/tests discussed in this report can be found in **‘wlang/test\_SSE\_DSE\_SeSE.py’.** * Some more interesting test cases can be found in **‘wlang/test\_ SeSE\_v1.py’.** |

**b] Version 2:**

|  |
| --- |
| **version2\_selective\_symbolic\_execution (current\_node, curr \_state, selected\_ast\_node, execution\_type):**  **Input:** **current\_node:** Abstract Syntax Tree of the program/block under test  **curr\_state:** Current state in the form of a tuple (symbolic\_pathCondition, symbolic\_environment,  concolic\_environment). Initially (in the first recursive call) all 3 are empty/NULL.  **selected\_ast\_node:** Node to be executed symbolically  **execution\_type:** Determines whether the implementation should proceed symbolically or concretely.  Initially (in the first recursive call) the value is ‘concrete’.  **Output:** List of feasible states    **Algorithm:**  flag\_turned = false  if reference (current\_node) == reference (selected\_ast\_node); then:  execution\_type = ‘symbolic’  flag\_turned = true  list\_feasible\_states = **perform\_node\_specific\_execution** (current\_node, curr\_state, selected\_ast\_node, execution\_type)  if flag\_turned == true:  execution\_type = ‘concrete’    if current\_node.has\_next():  new\_states = [ ]  for state in list\_ feasible \_states:  results = **version2\_selective\_symbolic\_execution** (current\_node -> next(), curr\_state, selected\_ast\_node, execution\_type)  new\_states.extend(results)    return new\_states  else:  return list\_feasible\_states |
| Here, the method ‘**perform\_node\_specific\_execution()’** performs the execution (symbolic or concrete – as specified in the argument) and my also involve a call to ‘**selective\_symbolic\_execution()’** (for any of the nested nodes - which are a block). ‘**perform\_node\_specific\_execution()’** also assigns model values to concrete variables (for feasible paths), when branch nodes are reached.  Note here, that once we find the ‘selected\_node’, only the ‘node-specific execution’ is performed fully symbolically.  And the rest of the implementation proceeds concretely. |
| * The implementation of this algorithm can be found in **‘wlang/SeSE\_v2.py’.** * The examples/tests discussed in this report can be found in **‘wlang/test\_SSE\_DSE\_SeSE.py’.** * Some more interesting test cases can be found in **‘wlang/test\_ SeSE\_v2.py’.** |

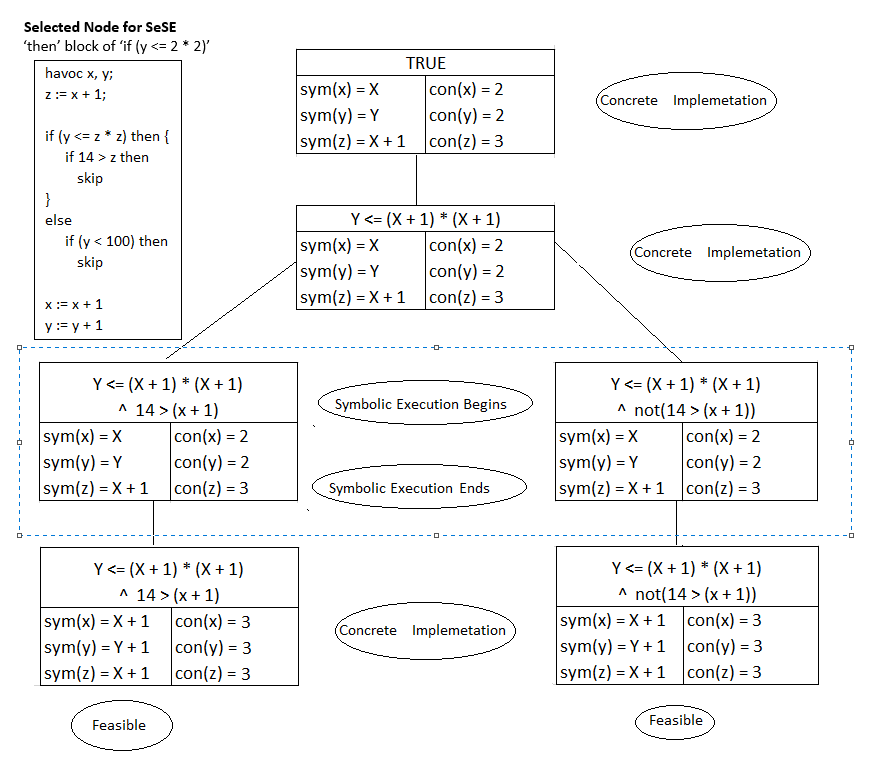
* **Examples (Version 1 v/s Version 2)**

**Version 1:**



**Note that:** there is a possibility that the selected node may not be reached.

**Version 2:**



**Note that:** there is a possibility that the selected node may not be reached.

* Explanation
* Observations

1. **Depth-first-style Dynamic Symbolic Execution Engine**

* **Introduction**

Here, we are implementing a dynamic approach to symbolic execution in a depth-first manner.

As we have already learnt, the major benefit of dynamic symbolic execution is faster processing using CPU for concrete part of the execution; and in security applications – where checking satisfiability of a path condition involving complex operations may be difficult for an SMT solver.

We cover both these aspects in our implementation.

We use WLANG as the artifact. There is no concept pf methods in WLANG. Instead, for our implementation, we are considering multiplication (\*) and division (/) to be complex operations, which most SMT solvers struggle with.

* **Pre-set**
* Using WLANG as the artifact
* Complex operations are - multiplication (\*) and division (/).
* To simplify the implementation, loop invariants have been supressed.
* A state is a (path\_condition, symbolic\_environment, concolic\_environment)

where: **i] path\_condition:** List of constraints in symbolic form

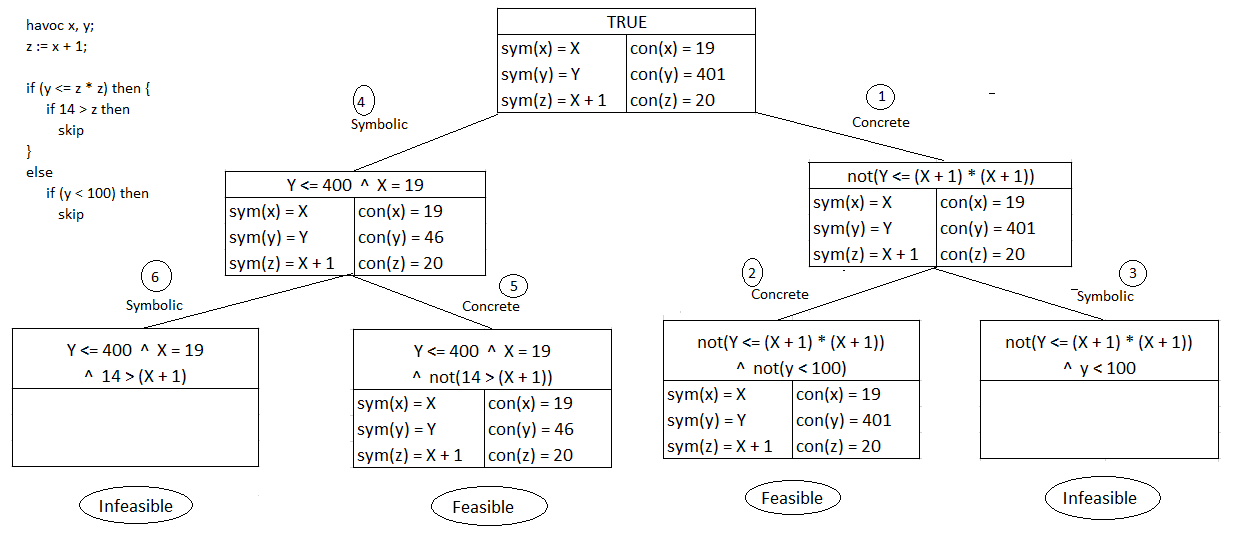
**ii] symbolic\_environment:** A dictionary which maps each defined variable to its symbolic value.

**Iii] concrete\_environment:** A dictionary which maps each defined variable to its concrete value.

* **Algorithm**

|  |
| --- |
| **dynamic\_symbolic\_execution (current\_node, curr \_state):**  **Input:** **current\_node:** Abstract Syntax Tree of the program/block under test  **curr\_state:** Current state in the form of a tuple (symbolic\_pathCondition, symbolic\_environment,  concolic\_environment). Initially (in the first recursive call) all 3 are empty/NULL.    **Output:** List of feasible states    **Algorithm:**  branch\_node\_type = [‘if-then-else’, ‘while’, ‘assertion’]    list\_feasible\_states = [ ]  if ‘current\_node.type’ in ‘branch\_node\_type’:  concrete\_branch, symbolic\_branch = **find\_concrete\_symbolic** (current\_node, curr\_state)  result\_concrete = **concretely\_perform\_node\_specific\_execution** (concrete\_branch, curr\_state)  result\_symbolic = **symbolically\_perform\_node\_specific\_execution** (symbolic\_branch, curr\_state)    list\_feasible\_states.append (result\_concrete, result\_symbolic)  else:  list\_feasible\_states = **perform\_node\_specific\_execution** (current\_node, curr\_state)  if current\_node.has\_next():  new\_states = [ ]  for state in list\_feasible\_states:  results = **dynamic\_symbolic\_execution** (current\_node -> next(), curr\_state)  new\_states.extend(results)    return new\_states  else:  return list\_feasible\_states |
| - **find\_concrete\_symbolic ():**  Finds the branch that can be satisfied with concrete values and assigns the other branch as symbolic.  - **concretely\_perform\_node\_specific\_execution** **():**  Concretely executes branch that holds under the concrete values.  1] If any such path exists, then proceeds with that branch-specific execution.  - **symbolically\_perform\_node\_specific\_execution** **():**  Symbolically executes the given branch.  1] If: ‘Path\_Condition’ involves a complex operation;  - then: a] Compute the operation concretely and directly replace its value in the ‘path\_condition’  b] Concretize the symbolic variables, which were involved in the complex operation.  - else: Proceed to ‘Step-2’  2] Check satisfiability.  3] If: Satisfiable; then proceed to ‘Step-3’  - else: Mark as infeasible and return.  4] Get the model from SMT solver; and assign model values to ‘concrete variables’.  5] Proceed with the branch specific execution.  - The last 2 methods may also involve a call to ‘**selective\_symbolic\_execution()’** (for any of the nested nodes - which are a block) |
| * The implementation of this algorithm can be found in **‘wlang/DSE.py’.** * The examples/tests discussed in this report can be found in **‘wlang/test\_SSE\_DSE\_SeSE.py’.** * Some more interesting test cases can be found in **‘wlang/test\_DSE.py’.** |

* Examples



* Explanation
* Observations
* **Comparison & Observations**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Example 1**  havoc x, y;  z := x + 1;  if (z\*z >= y) then      {          if 14 > z then          skip      }  else      if (y < 100) then          skip  **Selected Node for SeSE (Both Versions):**  ‘then’ block of ‘if (y <= 2 \* 2)’ | | **Example 2**  havoc x, y;  if x + y > 15 then      { x := x + 7; y := y - 12}  else      { y := y + 10; x := x - 2 };  x := x + 2;  if 2 \* (x + y) > 21 then      {          x := x \* 3;          y := y \* 2      }  else      {          x := x \* 4;          y := y \* 3 + x      };  skip  **Selected Node for SeSE (Both Versions):**  1st ‘if-then-else’ block | | **Example 3**  r := 0;  havoc x;  if x > 8 then {      havoc x;      r := x - 7  };  if x / 1 < 5 then      r := x - 2  **Selected Node for SeSE (Both Versions):**  2nd ‘if-then’ block | |
| **Time** | **# of feasible paths** | **Time** | **# of feasible paths** | **Time** | **# of feasible paths** |
| **Static Symbolic Execution** |  | 4 |  | 3 |  | 4 |
| **Dynamic Symbolic Execution** |  | 2 or 3 |  | 2 |  | 2 |
| **Selective Symbolic Execution – Version 1** |  | 1 or 2 |  | 3 |  | 4 |
| **Selective Symbolic Execution – Version 2** |  | 1 or 2 |  | 2 |  | 2 |

**References:**

1. A Survey of Search Strategies in the Dynamic Symbolic Execution - Yu LIU\*, Xu ZHOUa and Wei-Wei GONGb
2. A Survey of Symbolic Execution Techniques - ROBERTO BALDONI, EMILIO COPPA, DANIELE CONO D’ELIA, CAMIL DEMETRESCU, and IRENE FINOCCHI, Sapienza University of Rome
3. Selective Symbolic Execution - Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea School of Computer and Communication Sciences ´E cole Polytechnique F´ed´erale de Lausanne (EPFL), Switzerland
4. <https://www.tutorialspoint.com/software_testing_dictionary/symbolic_execution.htm>
5. <https://en.wikipedia.org/wiki/Symbolic_execution>
6. <https://www.microsoft.com/en-us/research/project/z3-3/>
7. <https://git.uwaterloo.ca/stqam-1225/pdfs/-/raw/master/W05-SymExec.pdf>
8. DART: Directed Automated Random Testing - Patrice Godefroid Nils Klarlund, Koushik Sen

<https://web.eecs.umich.edu/~weimerw/590/reading/p213-godefroid.pdf>

1. CUTE: A Concolic Unit Testing Engine for C - Koushik Sen, Darko Marinov, Gul Agha

<https://mir.cs.illinois.edu/marinov/publications/SenETAL05CUTE.pdf>

1. EXE: Automatically Generating Inputs of Death - Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler

<https://dl.acm.org/doi/10.1145/1455518.1455522>

1. <https://klee.github.io/>
2. <https://git.uwaterloo.ca/stqam-1225/pdfs/-/raw/master/W06-DSE.pdf>
3. SAGE: Whitebox Fuzzing for Security Testing - Patrice Godefroid, Michael Y. Levin, David Molnar, Microsoft

<https://queue.acm.org/detail.cfm?id=2094081>

1. Directed symbolic execution - Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks

<https://www.cs.umd.edu/~mwh/papers/dse-sas11.pdf>