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

1. **Selective Symbolic Execution Engine**

* **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:**

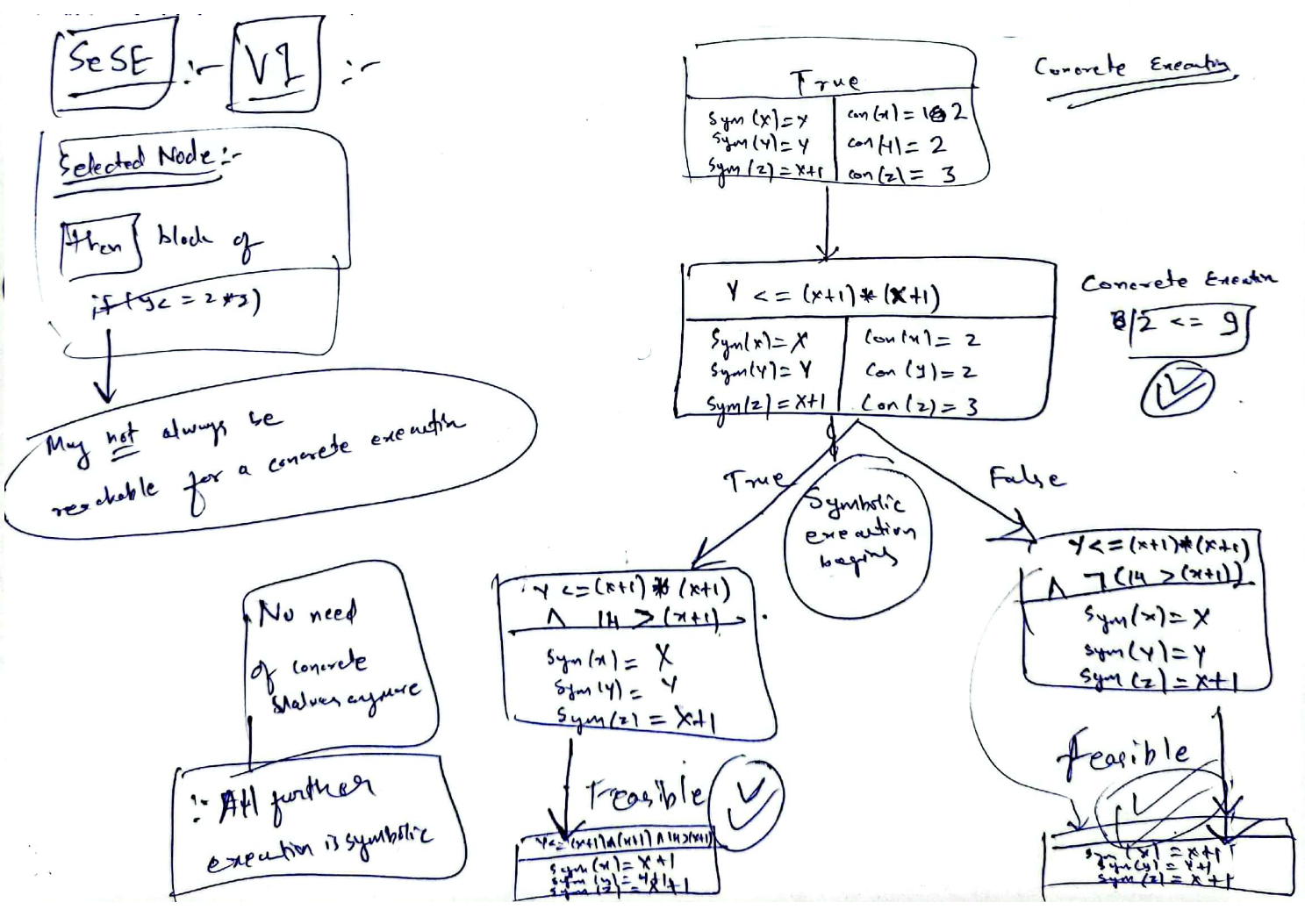
|  |
| --- |
| **selective\_symbolic\_execution (current\_node, curr \_state, selected\_ast\_node, execution\_type):**  **Input:** **current\_node:** Abstract Syntax Tree of the program/block under test  **curr\_state:** Current state in the form of a tuple (path\_condition, symbolic\_environment,  concolic\_environment). 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 concolically.  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 = **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:**

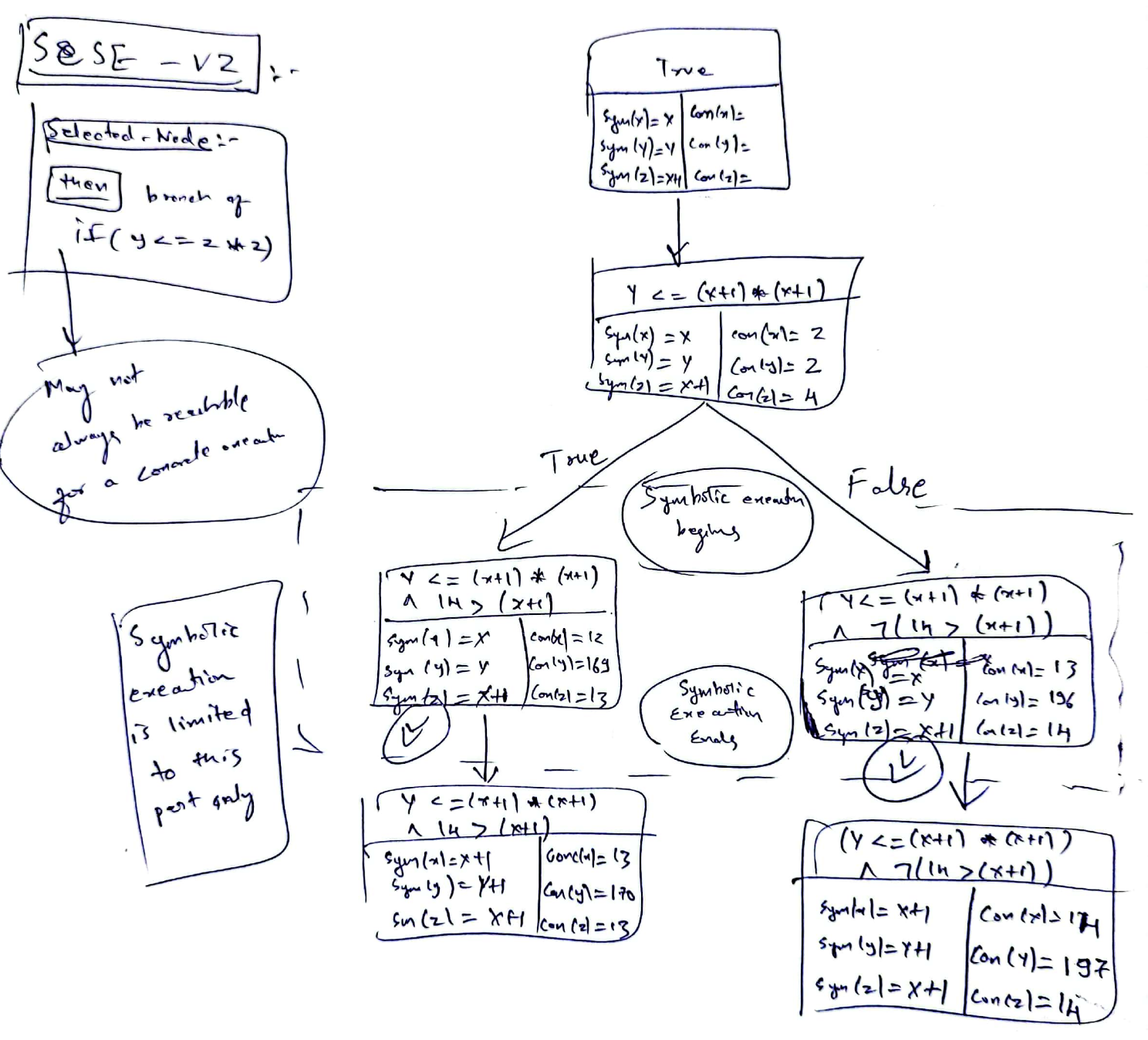
|  |
| --- |
| **selective\_symbolic\_execution (current\_node, curr \_state, selected\_ast\_node, execution\_type):**  **Input:** **current\_node:** Abstract Syntax Tree of the program/block under test  **curr\_state:** Current state in the form of a tuple (symbolic\_pathCondition, symbolic\_environment,  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 concolically.  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 = **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:**



**Version 2:**



* 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 contraints in symbolic form

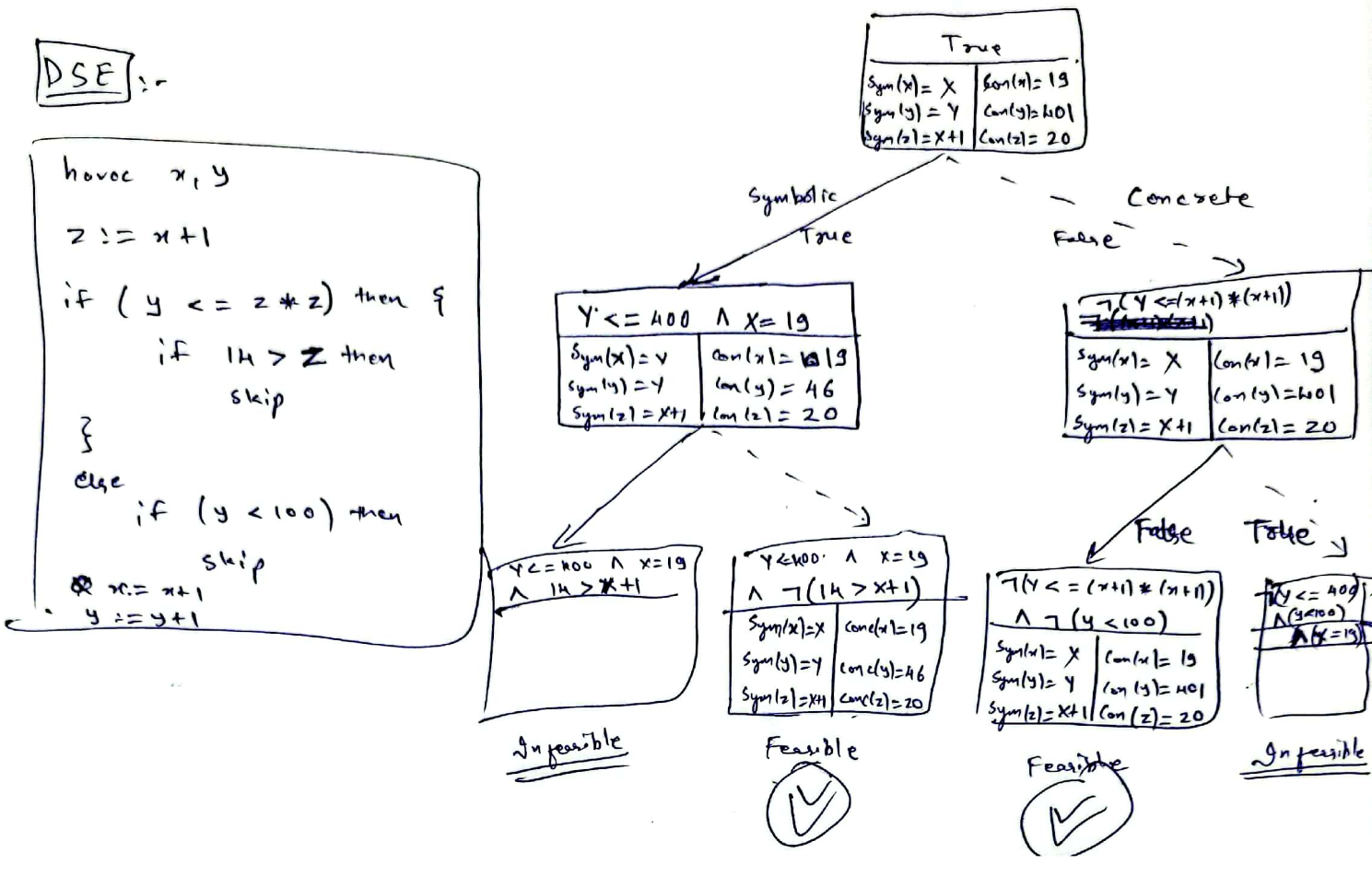
**ii] symbolic\_environment:** A dictionary which maps each defined variable to its symbolic value.

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

* **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 |