Assignment 2: Concolic Testing

Milan Lagae

Institution/University Name:	Vrije Universiteit Brussel
Faculty:	Sciences and Bioengineering Sciences
Course:	Software Quality Analysis

Contents

1 Intro

This report will discuss an implementation for the assignment “Assignment 2: Concolic Testing” for the course: Software Quality Analysis. Complete code snippets can be found in the Appendix section §6 of the respective section.

2 Discussion Point 1

This section will discuss the results of the first discussion point.

2.1 Manual Execution

2.1.1 Legend

The meaning of the values in the table(s) below:
- / program did not enter this line
- || value is the same as the previous value

2.1.2 Iteration 1

Code	**Concrete Store**	**Symbolic Store**	**Path Conditions**
main ( ) {			
var x , y , z ;			
x = input ;	[x -> 0] (input = 0)	[x -> a]	
z = input ;	[x -> 0, z -> 0] (input = 0)	[x -> a, z -> b]	
y = &x ;	[x -> 0, z -> 0, y -> x]	[x -> a, z -> b, y -> x]	
if (x > 0) {	[x -> 0, z -> 0, y -> x]	[x -> a, z -> b, y -> x]	[!(a > 0)]
y = &z ;	/	/	||
} else {	||	||	||
*y = input ;	[x -> 0, z -> 0, y -> x] (input = 0)	[x -> a, z -> b, y -> x]	||
}	||	||	||
*y = *y + 7 ;	[x -> 7, z -> 0, y -> x] (7 + 0)	[x -> a, z -> b, y -> x]	||
if (2 > z) {	||	||	[!(a > 0) and !(2 > (b + 7))]
if (*y == 2647) {	/	/	/
error 1 ;	/	/	/
}	/	/	/
}	/	/	/
return *y ;	7	||	||
}	-	-	[!(a > 0) and !(2 > (b + 7))]

With the above result, negate the first condition $\neg(a > 0)$ from the PC: $\neg(a > 0) \wedge (b + 7)$. The input for the next iteration is as follows:
- $x = 1$
- $z = 0$

2.1.3 Iteration 2

Code	Concrete Store	Symbolic Store	Path Conditions
main ( ) {			
var x , y , z ;			
x = input ;	[x -> 1] (input = 1)	[x -> a]	
z = input ;	[x -> 1, z -> 0] (input = 0)	[x -> a, z -> b]	
y = &x ;	[x -> 1, z -> 0, y -> x]	[x -> a, z -> b, y -> x]	
if (x > 0) {	||		[(a > 0)]
y = &z ;	[x -> 1, z -> 0, y -> z]	[x -> a, z -> b, y -> z]	
} else {	/		
*y = input ;	/		
}	/		
*y = *y + 7 ;	[x -> 1, z -> 7, y -> x] (7 + 0)	[x -> a, z -> b, y -> x]	||
if (2 > z) {			[(a > 0) and !(2 > (b + 7))]
if (*y == 2647) {			
error 1 ;			
}			
}			
return *y ;	7		
}			

With the extended path condition now consisting of: $(a > 0) \land \neg(2 > (b + 7)))$. The previous input resulted in the first 2 clauses of the PC to be true. For the next iteration the following input values will be chosen:

- $x = 1$
- $z = -5$

Keeping the value of x the same, but changing the value of z, as to negate the second clause of the PC.

2.1.4 Iteration 3

Code	Concrete Store	Symbolic Store	Path Conditions
main ( ) {			
var x , y , z ;			
x = input ;	[x -> 1] (input = 1)	[x -> a]	
z = input ;	[x -> 1, z -> −5] (input = −5)	[x -> a, z -> b]	
y = &x ;	[x -> 1, z -> −5, y -> x]	[x -> a, z -> b, y -> x]	
if (x > 0) {	||		[(a > 0)]
y = &z ;	[x -> 1, z -> −5, y -> z]	[x -> a, z -> b, y -> z]	
} else {	/		
*y = input ;	/		
}	/		
*y = *y + 7 ;	[x -> 1, z -> 2, y -> x] (7 + −5)	[x -> a, z -> b, y -> x]	||
if (2 > z) {			[(a > 0) and !(2 > (b + 7))]
if (*y == 2647) {			
error 1 ;			
}			
}			
return *y ;	2		
}			

So the chosen input did not show any new branches, therefore revert back to the previous input with:

- $x = 0$
- $z = 0$

and

- $y = 2640$

2.1.5 Iteration 4

Code	Concrete Store	Symbolic Store	Path Conditions
main ( ) {			
var x , y , z ;			
x = input ;	[x -> 0] (input = 0)	[x -> a]	
z = input ;	[x -> 0, z -> 0] (input = 0)	[x -> a, z -> b]	
y = &x ;	[x -> 0, z -> 0, y -> x]	[x -> a, z -> b, y -> x]	
if (x > 0) {	[x -> 0, z -> 0, y -> x]	[x -> a, z -> b, y -> x]	[!(a > 0)]
y = &z ;	/	/	||
} else {	||	||	||
*y = input ;	[x -> 2640, z -> 0, y -> x] (input = 2640)	[x -> a, z -> b, y -> x]	||
}	||	||	||
*y = *y + 7 ;	[x -> 2647, z -> 0, y -> x] (7 + 2640)	[x -> a, z -> b, y -> x]	||
if (2 > z) {	||	||	[!(a > 0) and (2 > (b + 7))]
if (*y == 2647) {	/	/	[!(a > 0) and (2 > (b + 7)) and (a == 2647)]
error 1 ;	Error		
}			
}			
return *y ;			
}	-	-	

Found failure sequence for input: $x = 0, z = 0, y = 2640$.

This concludes the concolic testing for this example, other input's can be found but they do not change the behaviour of the program or find a new failure sequence. For example; for y a negative value my traverse the false branch, but does not significantly change the program output.

2.2 Comparison

When looking at the output of the concolic testing using TIP, the input's are chosen randomly. While in contrast, using manual concolic execution, values can be chosen more targeted, in the sense arriving more easier to failure sequences.

But the cost of keeping track of all values throughout execution is difficult for larger programs.

In the end both execution should arrive at the 'same' values, with values depending on the program accepted range of input.

2.3 TIP Pointer Support

TIP Pointer support can be found in the Interpreter.scala file. In Listing 1 & Listing 2, the TIP implementation for pointer referencing (&z) and dereferencing (*y) is shown.

2.3.1 Reference

Match the AExpr on the class AVarRef, to set a variable reference.

```
case AVarRef(e: AIdentifier, _) =>
  // DP1 - &z value position
  log.debug("Variable reference: " + exp)
  (env(e.declaration), store)
```

Listing 1: Interpreter - Pointer Reference

2.3.2 Dereference

Dereference is done by matching on AUnaryOp & DerefOp, retrieving the ReferenceValue from the result of applying semright on the expression.

```
case e: AUnaryOp =>
  val (sub, s1) = semeright(e.subexp, env, store)
  val cval = e.operator match {
    case DerefOp =>
      // DP1 - *z value position
      log.debug("Dereference Operator: " + exp)
      sub match {
        case crv: ReferenceValue =>
          s1.getOrElse(crv, errorUninitializedLocation(exp.loc, s1))
        case _ => errorDerefNotPointer(exp.loc, sub, s1)
      }
  }
  (cval, s1)
```

Listing 2: Interpreter - Pointer Dereference

3 Discussion Point 2

This section will discuss the implementation of the second discussion point. The modified files are those with starting annotations: SymbolicValues.scala, Interpreter.scala and SMTSolver.scala.

3.1 Implementation

This subsection will discuss the implementation of concolic testing on arrays in TIP. In order of sequence will the operations be discussed: Interpreter, SymbolicValues and last SMTSolver.

3.1.1 Interpreter

The purpose of the interpreter statements is updating/ accessing the store with the correct reference values of the array & concrete values stored in the array.

Array Creation

The created array is iterated and the store is updated by use of a fold. Each concrete value is given a reference value, with the reference value kept within the symbolic array.

Array Select

Retrieve the location of the concrete value by accessing the content field on the array, using the integer value of the idxv. Once the location is retrieved the concrete value can be accessed by accessing the s2 variable or the updated store.

```
// MOD-DP2
// Array Access - since we are on the right side of an expression
case AArrAcc(arr, idx, _) =>
  // Evaluate the idx
  val (idxv, s1) = semeright(idx, env, store)
  // Using the new store, evaluate the arr
  val (arrv, s2) = semeright(arr, env, s1)
  // Get the value and return the latest store
  (idxv, arrv) match {
    case (idxv: IntValue, arrv: ArrValue) =>
      log.debug("[IP] - Selecting value")
      // Get reference value
      val location = arrv.content(idxv.i)
      // Access concrete value from store
      val concreteValue: IntValue = s2.getOrElse(location, null) match {
        case intValue: IntValue => intValue
      }
      // Return symbolic value & updated store
      (spec.arraySelect(arrv, idxv, concreteValue), s2)
  }
```

Listing 3: Interpreter - Select Array

Apply the arraySelect method to generate the symbolic value of the array operation. Return the symbolic value and updated store.

Array Store

First generate the symbolic values of the array updated. Retrieve the concrete reference for the array (arrayReference) by accessing the envusing the id which is a location type.

Accessing the reference for the updated value (valueReference)is done by accessing the content field on the array. Return a tuple of the env and updated store with concrete references to both the array & concrete value updated.

```
case AAssignStmt(left: AArrAcc, right: AExpr, _) =>
 // Left index
 val (idxv, s1) = semeright(left.idx, env, store)
 // Left array
 val (arrv, s2) = semeright(left.arr, env, s1)
 // Value to place in array
 val (rhsv, s3) = semeright(right, env, s2)
 (idxv, arrv, left.arr, rhsv) match {
   // MOD-DP2
   case (idxv: IntValue, arrv: ArrValue, id: AIdentifier, rhsv: IntValue) =>
     log.debug("[IP] - Storing array value")
     // Get updated array
     val updatedArray: ArrValue = spec.arrayStore(arrv, idxv, rhsv)
     // Get reference value array from env
     val arrayReference = env.getOrElse(id, null)
     // Get reference value value
     val valueReference = updatedArray.content(idxv.i)
     // Update store with reference for array & value
     (env, s3 + (arrayReference -> updatedArray) + (valueReference -> rhsv))
 }
```

Listing 4: Interpreter - Store Array

3.1.2 SymbolicValues

Based on the operation performed in the interpreter, created the symbolic value, to later pass to the SMTSolver.

Array Creation

Iterate over the list of vals, retrieve the symbolic value of each and store in a sequence of AExpr. Return a SymbArrValue, containing the vector of refs and AArrOp element as second field, with symbolicArray as first element.

```
// MOD-DP2
def arrayValue(refs: Vector[ReferenceValue], vals: Vector[IntValue]): ArrValue = {
 log.debug("[SV] - Creating array value")
 // Create a sequence of symbolic expressions
 val symbolicArray: Seq[AExpr] = vals.map {
   case SymbIntValue(_, symbolic) => symbolic
 }

 log.debug("[SV] - Array content: " + symbolicArray)
 SymbArrValue(refs, AArrOp(symbolicArray, noLoc))
}
```

Listing 5: SymbolicValues - Create Array

Array Select

```
// MOD-DP2
def arraySelect(arr: ArrValue, idx: IntValue, v: IntValue): IntValue = {
 (arr, idx) match {
   case (arr: SymbArrValue, idx: SymbIntValue) =>
     log.debug("[SV] - Selecting value in array")
     SymbIntValue(v.i, AArrAcc(arr.symbolic, idx.symbolic, noLoc))
 }
}
```

Listing 6: SymbolicValues - Select Array

Array Store

```
// MOD-DP2
def arrayStore(arr: ArrValue, idx: IntValue, v: IntValue): ArrValue = {
 (arr, idx, v) match {
   case (arr: SymbArrValue, idx: SymbIntValue, v: SymbIntValue) => {
     log.debug("[SV] - Setting value in array")
     SymbArrValue(arr.content, AArrUpdate(arr.symbolic, idx.symbolic, v.symbolic, noLoc))
   }}}}}
```

Listing 7: SymbolicValues - Store Array

3.1.3 SMTSolver

Array Creation

A constant array in Z3 is created by the following statement: ((as const (Array Int Int) 0). If there are no elements in the array, return just the array starting value. If there are elements present, create the populated array from the base values with a chained list of store statements populating it.

```
// MOD-DP2
// ((as const (Array Int Int) 0)
case AArrOp(elems, _) =>
  log.debug("[SMT] - Array")
  // Z3 Starting array value
  val startingArray = "((as const (Array Int Int)) 0)"
  if (elems.isEmpty) {
    // If no elements, return the empty array
    startingArray
  } else {
    // Create the constant array from individual store statements
    val populatedArray = elems.zipWithIndex.foldLeft(startingArray) {
      case (acc, (elem, idx)) =>
        s"(store $acc ${idx.toString} ${expToSexp(elem)})"
    }
    populatedArray
  }
```

Listing 8: SMTSolver - Create Array

The select & store expand the array with the elements, as shown above, appending the new operation to it.

Array Select

```
// MOD-DP2
// (select ((as const (Array Int Int) 0) x)
case AArrAcc(arr, idx, _) =>
  log.debug("[SMT] - Select")
  s"(select ${expToSexp(arr)} ${expToSexp(idx)})"
```

Listing 9: SMTSolver - Select Array

Array Store

```
// MOD-DP2
// (store ((as const (Array Int Int) 0) x y)
case AArrUpdate(arr, idx, v, _) =>
  log.debug("[SMT] - Update")
  s"(store ${expToSexp(arr)} ${expToSexp(idx)} ${expToSexp(v)})"
```

Listing 10: SMTSolver - Store Array

3.2 Execution Analysis

The program execution performed 36 runs, of which 27 where successful and 9 contain errors or unsatisfiable path.

TODO

4 Discussion Point 3

This section will discuss the implementation of the third discussion point. As described in the assignment, two priorities where implemented. Both strategies where implemented in the ConcolicEngine.scala file.

4.1 Bread-First Search Priority

The first priority based BFS strategy is implemented by the method: nextExplorationTargetBFS. The search starts from the the root of the execution tree. The complete method implementation for the BFS Strategy can be found in Listing 13.

The first step is creating a mutable.PriorityQueue, by default the queue retrieves the elements with the highest priority, the algorithm requires the element with the lowest priority (shortest distance from root), for this the ordering is reversed on queue creation.

After the empty queue is created, the children of the root are iterated and for each a tuple, is created containing the child node: (child, 1), with the initial distance set to 1.

The next step in the algorithm is to iterate the queue until it is empty. On entering the queue, the first element highest priority/shortest distance from root is removed.

Using the below match case statement from the existing nextExplorationTarget method, the true & false branches are checked for the status of amount of explored branches.

```
// Check if node has been explored
val nextTarget: Option[(Branch, Boolean)] = node match {
  case b: Branch =>
    (b.branches(true), b.branches(false)) match {
      // True branch unexplored
      case (_: SubTreePlaceholder, _) if b.count(true) == 0 => Some((b, true))
      // False branch unexplored
      case (_, _: SubTreePlaceholder) if b.count(false) == 0 => Some((b, false))
      // None are unexplored
      case (_, _) => None
    }
  case _ => None
}
```

Listing 11: BFS - Is Branch Explored

If the count of explored branches of either value is 0, the existing node is returned. In case both branches have already been explored, the None value is returned.

The final step in the algorithm, is the if case, which can be found in the listing below: Listing 12.

```
if (nextTarget.isDefined) {
  // Return the found target node
  log.debug("[DP3-BFS] - Found next target node, distance from root: " + distance)
  return nextTarget
} else {
  // Append the children of node that has been explored
  log.debug("[DP3-BFS] - Target node has been explored, appending node children, and checking other nodes first")
  // Ensure elements append first (lower distance) are visited first, bread first search
  queue = queue ++ node.children.map(child => (child, distance + 1))
}
```

Listing 12: BFS - Append Children

If there is a value defined the chosen node is used as the `nextTarget`. If no value is defined, all children of the node are added to the queue, with the distance increased by one: `distance + 1`.

4.2 Random Priority

The second search strategy is based on assigning a random priority to the node, the implementation method can be found in method: `nextExplorationTargetRNDM` and in listing: Listing 14.

The only changed required for this strategy is importing the `scala.util.Random` package. When populating the queue with the initial values of the `root`, use the `rand.nextInt(Int.MaxValue)` to assign a random value.

The method will proceed as before, the only remaining difference is the case when the node has already been explored. When appending the children of the node, instead of increment the distance value as before, a random value is assigned with the function described as before.

5 Discussion Point 4

This section will discuss the implementation of the different search strategies and the effect on the number of runs & resulting errors cases.

5.1 Overview

The results of the concolic testing can be summarized and are visible in Table 1.

Strategy	Runs	Failures
DFS	21	1
BFS	21	1
Random	21	1

Table 1: Concolic Testing - Search Strategies

Since the purpose of concolic testing, is the exploration of all branches, changing the strategy of which branches are explored first does not inherently change the result of the execution. All satisfiable & unsatisfiable branches will be explored eventually.

6 Appendix

6.1 Discussion Point 3

```scala
// DP3 - Priority - BFS
def nextExplorationTargetBFS(root: ExecutionTreeRoot): Option[(Branch, Boolean)] = {
  log.info("[DP3-BFS] - Using Bread First Search, with priority queue & distance from root")

  // Initiate the queue, with ordering based on the lowest value (shortest distance) to root node
  var queue: mutable.PriorityQueue[(ExecutionTree, Int)] = mutable.PriorityQueue()(Ordering.by[(ExecutionTree, Int), Int](_._2).reverse)
  // Populate the queue with the list of children from the root, and set the distance immediately to 1
  queue = queue ++ root.children.map(child => (child, 1))

  while (queue.nonEmpty) {
    // Get the first element from the queue
    val (node, distance) = queue.dequeue()

    // Check if node has been explored
    // - if been explored = add children to queue with distance + 1
    // - else return that node as next exploration target
    val nextTarget: Option[(Branch, Boolean)] = node match {
      case b: Branch =>
        (b.branches(true), b.branches(false)) match {
          // True branch unexplored
          case (_: SubTreePlaceholder, _) if b.count(true) == 0 => Some((b, true))
          // False branch unexplored
          case (_, _: SubTreePlaceholder) if b.count(false) == 0 => Some((b, false))
          // None are unexplored
          case (_, _) => None
        }
      case _ => None
    }

    if (nextTarget.isDefined) {
      // Return the found target node
      log.debug("[DP3-BFS] - Found next target node, distance from root: " + distance)
      return nextTarget
    } else {
      // Append the children of node that has been explored
      log.debug("[DP3-BFS] - Target node has been explored, appending node children, and checking other nodes first")
      // Ensure elements append first (lower distance) are visited first, bread first search
      queue = queue ++ node.children.map(child => (child, distance + 1))
    }
  }
  None
}
```

Listing 13: Bread-First Search Strategy

```
// DP3 - Priority - Random
def nextExplorationTargetRNDM(root: ExecutionTreeRoot): Option[(Branch, Boolean)] = {
  log.info("[DP3-RNDM] - Using Bread First Search, with random priority")
  val rand = new scala.util.Random

  // Initiate the queue, with ordering based on the lowest value (shortest distance) to root node
  var queue: mutable.PriorityQueue[(ExecutionTree, Int)] = mutable.PriorityQueue()(Ordering.by[(ExecutionTree, Int), Int](_._2).reverse)
  // Populate the queue with the list of children from the root
  queue = queue ++ root.children.map(child => (child, rand.nextInt(Int.MaxValue)))

  while (queue.nonEmpty) {
    // Get the first element from the queue
    val (node, priority) = queue.dequeue()

    // Check if node has been explored
    // - if been explored = add children to queue with distance + 1
    // - else return that node as next exploration target
    val nextTarget: Option[(Branch, Boolean)] = node match {
      case b: Branch =>
        (b.branches(true), b.branches(false)) match {
          // True branch unexplored
          case (_: SubTreePlaceholder, _) if b.count(true) == 0 => Some((b, true))
          // False branch unexplored
          case (_, _: SubTreePlaceholder) if b.count(false) == 0 => Some((b, false))
          // None are unexplored
          case (_, _) => None
        }
      case _ => None
    }

    if (nextTarget.isDefined) {
      // Return the found target node
      log.debug("[DP3-RNDM] - Found next target node, priority: " + priority)
      return nextTarget
    } else {
      // Append the children of node that has been explored
      log.debug("[DP3-RNDM] - Target node has been explored, appending node children, and checking other nodes first")
      // Ensure elements append first (lower distance) are visited first, bread first search
      queue = queue ++ node.children.map(child => (child, rand.nextInt(Int.MaxValue)))
    }
  }
  None
}
```

Listing 14: Random Priority Strategy