1st Assignment 2IMP0 - Model checking based verification

I. Makantasis 1002480, Timon Heuwekemeijer 1003212

November 2021

Part 1 - System modeling and analysis

Exercise 1

The source code of our leader election protocol for three nodes is shown in code listing 1. Each node is represented as an instance of a module. This module, the Node module, is instantiated with the unique ID of that node. Each node keeps track of whether it declared itself as leader (`leader`), what their ID is (`ownID`), what the value received last was (`recievedID`) and what the largest value encountered so far is (`maxID`). In the main module, the nodes are initialized, and only message passing is handled. All other logic (processing the received value), is handled by the node.

Listing 1: Leader Election Protocol for 3 Nodes

```
MODULE Node(ID)
        DEFINE
                ownID := ID;
        VAR
                leader: boolean;
                maxID: 0..3;
                receivedID: 0..3;

        ASSIGN
                init(maxID) := ownID;
                init(leader) := FALSE;
                init(receivedID) := 0;

                next(leader) :=
                        case
                            receivedID = ownID: TRUE;
                            TRUE: leader;
                        esac;

                next(maxID) :=
                        case
                            receivedID > maxID : receivedID;
                            TRUE: maxID;
                        esac;

MODULE main
        VAR
                node1 : Node(2);
                node2 : Node(3);
                node3 : Node(1);

        ASSIGN
                next(node2.receivedID) := node1.maxID;
                next(node3.receivedID) := node2.maxID;
                next(node1.receivedID) := node3.maxID;
```

Because almost all logic is handled by the Node module, scaling the model to a higher node count, only involves expanding the main module to instantiate more nodes and properly pass messages among the nodes.

The message passing is done as simply as possible. Each node simply communicates the maximum value it has encountered so far to the next node in the ring. It is all done in sequence to minimize the number of states necessary to model this, and thus avoid state explosion.

Exercise 2

The properties used to verify the model shown in listing 1 can be seen in listing 2. Both of these listings were part of the same file, and the properties were checked by running NuSMV on the file non interactively (`NuSMV LeaderElection.smv`).

Listing 2: Verification Properties for 3 Nodes

```
1  --Eventually there is always a leader
2  SPEC AF (node1.leader | node2.leader | node3.leader);
3  --Not every state has a leader
4  LTLSPEC !G (node1.leader | node2.leader | node3.leader);
5
6  --ReceivedID is initialized to 0 until ReceivedID is maxID of previous node
7  SPEC A [(node1.receivedID = 0 & node2.receivedID = 0 & node3.receivedID = 0)
8      U (node1.receivedID = node3.maxID &
9          node2.receivedID = node1.maxID &
10         node3.receivedID = node2.maxID)];
11
12 --If node1 has the largest ID, it will always eventually become the leader
13 SPEC node1.ownID > node2.ownID & node1.ownID > node3.ownID -> AF node1.leader;
14 --If node2 has the largest ID, it will always eventually become the leader
15 SPEC node2.ownID > node1.ownID & node2.ownID > node3.ownID -> AF node2.leader;
16 --If node3 has the largest ID, it will always eventually become the leader
17 SPEC node3.ownID > node1.ownID & node3.ownID > node2.ownID -> AF node3.leader;
18
19 --If one node is leader, the others are not
20 LTLSPEC G node1.leader -> !node2.leader & !node3.leader;
21 LTLSPEC G node2.leader -> !node1.leader & !node3.leader;
22 LTLSPEC G node3.leader -> !node2.leader & !node1.leader;
```

- The first CTL property ensures that the model ends up with a leader in all possible executions. It is crucial for this model that a leader is eventually chosen.
- The second LTL property ensures that not every possible state has a leader. It is a liveness property since it has an infinite counterexample. This property ensures that the model is not simply initialized with a leader.
- The next CTL property is a lot longer. This ensures both the initialization is correct and that the right values are passed between nodes eventually.
- The three CTL properties on lines 13-17 ensure that the node with the maximum ID eventually becomes the leader.
- The last three LTL properties ensure that at most one node is the leader at any time. These properties are all safety properties, as they have a finite counterexample.

These properties ensure that the model always ends up with a leader, that there is at most one leader, and that the correct leader is chosen. It also ensures that after the initial values, the correct values are sent from node to node.

Exercise 3

A

a For this step, to construct the models for the 6 and 9 nodes in the leader election process, the original model for the 3 nodes was modified. Thus, for 6 nodes, the code would be the following (only the changed parts are visible):

Listing 3: Leader Election Protocol for 9 Nodes

```
MODULE Node(ID)
        VAR
                maxID: 0..6;
                receivedID: 0..6;

MODULE main
        VAR
                node1 : Node(1);
                node2 : Node(2);
                node3 : Node(3);
                node4 : Node(4);
                node5 : Node(5);
                node6 : Node(6);

        ASSIGN
                next(node2.receivedID) := node1.maxID;
                next(node3.receivedID) := node2.maxID;
                next(node4.receivedID) := node3.maxID;
                next(node5.receivedID) := node4.maxID;
                next(node6.receivedID) := node5.maxID;
                next(node1.receivedID) := node6.maxID;
```

Similarly, for the case of 9 nodes in the leader election procedure, we have:

Listing 4: Leader Election Protocol for 6 Nodes

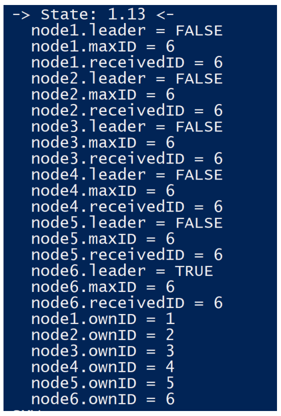
```
MODULE Node(ID)
        VAR
                maxID: 0..9;
                receivedID: 0..9;

MODULE main
        VAR
                node1 : Node(1);
                node2 : Node(2);
                node3 : Node(3);
                node4 : Node(4);
                node5 : Node(5);
                node6 : Node(6);
                node7 : Node(7);
                node8 : Node(8);
                node9 : Node(9);

        ASSIGN
                next(node2.receivedID) := node1.maxID;
                next(node3.receivedID) := node2.maxID;
                next(node4.receivedID) := node3.maxID;
                next(node5.receivedID) := node4.maxID;
                next(node6.receivedID) := node5.maxID;
                next(node7.receivedID) := node6.maxID;
                next(node8.receivedID) := node7.maxID;
                next(node9.receivedID) := node8.maxID;
                next(node1.receivedID) := node9.maxID;
```

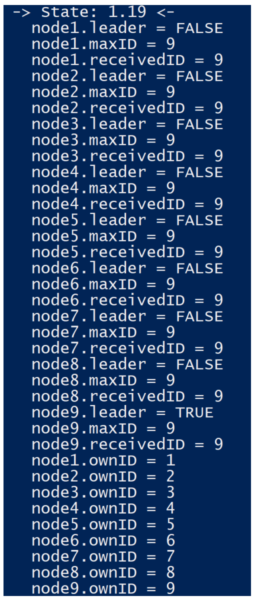
As our original procedure was already optimized, we were in a position of verifying all of our properties, both *LTL* and *CTL*. The commands `NuSMV LeaderElection_6_nodes.smv` and `NuSMV LeaderElection_9_nodes.smv` were used. Moreover, when inspecting the properties for both procedures, no distinguishable delay was found, as the properties were verified to be true nearly instantly.

b To find out the state space for all three models, the command `print_reachable_states` was used. We first provide the table containing both the reachable state space, and the whole state space of each model.



```
-> State: 1.13 <-
  node1.leader = FALSE
  node1.maxID = 6
  node1.receivedID = 6
  node2.leader = FALSE
  node2.maxID = 6
  node2.receivedID = 6
  node3.leader = FALSE
  node3.maxID = 6
  node3.receivedID = 6
  node4.leader = FALSE
  node4.maxID = 6
  node4.receivedID = 6
  node5.leader = FALSE
  node5.maxID = 6
  node5.receivedID = 6
  node6.leader = TRUE
  node6.maxID = 6
  node6.receivedID = 6
  node1.ownID = 1
  node2.ownID = 2
  node3.ownID = 3
  node4.ownID = 4
  node5.ownID = 5
  node6.ownID = 6
```

Figure 1: Results of the ring election procedure for 6 nodes.



```
-> State: 1.19 <-
  node1.leader = FALSE
  node1.maxID = 9
  node1.receivedID = 9
  node2.leader = FALSE
  node2.maxID = 9
  node2.receivedID = 9
  node3.leader = FALSE
  node3.maxID = 9
  node3.receivedID = 9
  node4.leader = FALSE
  node4.maxID = 9
  node4.receivedID = 9
  node5.leader = FALSE
  node5.maxID = 9
  node5.receivedID = 9
  node6.leader = FALSE
  node6.maxID = 9
  node6.receivedID = 9
  node7.leader = FALSE
  node7.maxID = 9
  node7.receivedID = 9
  node8.leader = FALSE
  node8.maxID = 9
  node8.receivedID = 9
  node9.leader = TRUE
  node9.maxID = 9
  node9.receivedID = 9
  node1.ownID = 1
  node2.ownID = 2
  node3.ownID = 3
  node4.ownID = 4
  node5.ownID = 5
  node6.ownID = 6
  node7.ownID = 7
  node8.ownID = 8
  node9.ownID = 9
```

Figure 2: Results of the ring election procedure for 9 nodes.

Models	Reachable States	Out of
3 nodes	7 (2^2.80735)	32768 (2^15)
6 nodes	13 (2^3.70044)	2^39.68803
9 nodes	19 (2^4.24793)	2^68.7947

Table 1: The difference in the state space between the three models

The graph presenting the changes to the whole state space when comparing the models for 3, 6 and 9 nodes can be seen in Figure 3.

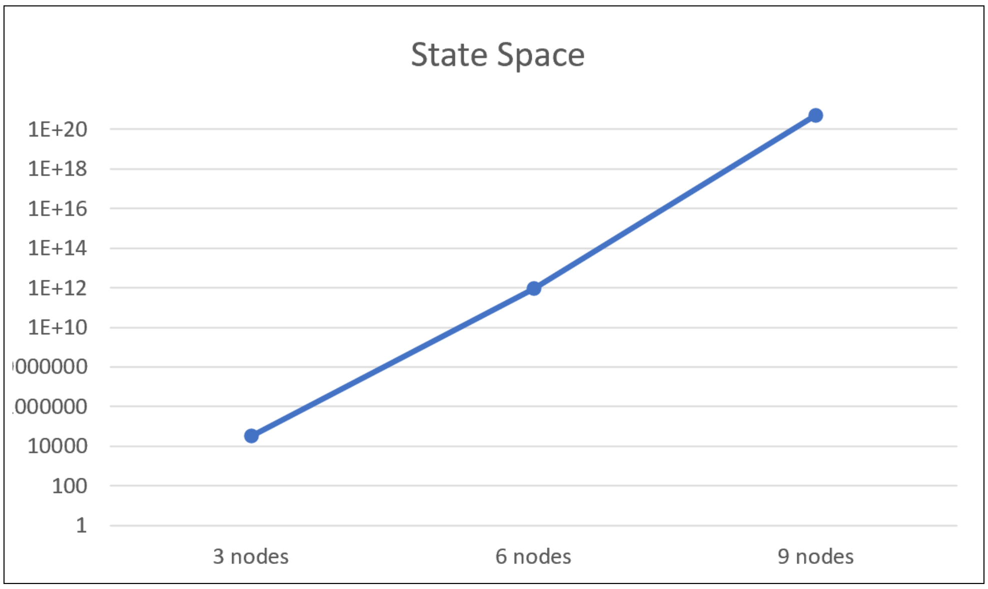


Figure 3: Linechart for the state space of the three models

Due to the sheer number of the reachable states, a logarithmic scale with base 10 was used in Figure 3 to represent them. It is in line with what we thought would be the case, as the number of possible choices is incremented every time the number of nodes increases, which is double and triple the initial number of nodes in our first model. However, looking at the Table 1, we can see that our implementations are indeed optimal as only a mere fraction of those states is actually been reached.

c For this part, we created two models for the procedure using 8 nodes. Since our original implementation creates an optimized model from the start, we also created a variation of it, where we deliberately created a state explosion in the reachable states. This was the result of giving the nodes a 50% chance of sending the ID that they have to the next node or skipping their turn.
Thus, for the optimised version, the code used is the following

Listing 5: Leader Election Protocol for 8 Nodes (Optimised)

```
MODULE Node(ID)
        VAR
                maxID: 0..8;
                receivedID: 0..8;

MODULE main
        VAR
                node1 : Node(1);
                node2 : Node(2);
                node3 : Node(3);
                node4 : Node(4);
                node5 : Node(5);
                node6 : Node(6);
                node7 : Node(7);
                node8 : Node(8);

        ASSIGN
                next(node2.receivedID) := node1.maxID;
                next(node3.receivedID) := node2.maxID;
                next(node4.receivedID) := node3.maxID;
                next(node5.receivedID) := node4.maxID;
                next(node6.receivedID) := node5.maxID;
                next(node7.receivedID) := node6.maxID;
                next(node8.receivedID) := node7.maxID;
                next(node1.receivedID) := node8.maxID;
```

For our second, unoptimised version we used the following variation

Listing 6: Leader Election Protocol for 8 Nodes (Unoptimised)

```
MODULE Node(ID)
        VAR
                maxID: 0..8;
                receivedID: 0..8;
                sendBit: 0..3;

        ASSIGN
                --1 in 2 chance of sending maxID next iteration
                next(sendBit) := { 2, 3};

MODULE main
        VAR
                node1 : Node(1);
                node2 : Node(2);
                node3 : Node(3);
                node4 : Node(4);
                node5 : Node(5);
                node6 : Node(6);
                node7 : Node(7);
                node8 : Node(8);

        ASSIGN
                next(node2.receivedID) :=
                        case
                                node1.sendBit = 3: node1.maxID;
```

```
                    TRUE: node2.receivedID;
            esac;
        next(node3.receivedID) :=
            case
                    node2.sendBit = 3: node2.maxID;
                    TRUE: node3.receivedID;
            esac;
        next(node4.receivedID) :=
            case
                    node3.sendBit = 3: node3.maxID;
                    TRUE: node4.receivedID;
            esac;
        next(node5.receivedID) :=
            case
                    node4.sendBit = 3: node4.maxID;
                    TRUE: node5.receivedID;
            esac;
        next(node6.receivedID) :=
            case
                    node5.sendBit = 3: node5.maxID;
                    TRUE: node6.receivedID;
            esac;
        next(node7.receivedID) :=
            case
                    node6.sendBit = 3: node6.maxID;
                    TRUE: node7.receivedID;
            esac;
        next(node8.receivedID) :=
            case
                    node7.sendBit = 3: node7.maxID;
                    TRUE: node8.receivedID;
            esac;
        next(node1.receivedID) :=
            case
                    node8.sendBit = 3: node8.maxID;
                    TRUE: node1.receivedID;
            esac;
```

In Table 2, we present the difference in the reachable states found in the two version of the model containing 8 nodes. Once again, the command used was `print_reachable_states`.

Models	Reachable States	Out of
8 nodes optimized	17	2^58.7188
8 nodes unoptimized	228608	2^74.7188

Table 2: The difference in reachable states between the two models

As we said before, this explosion to the reachable states is attributed to the fact that each node has to non-deterministically choose on whether to send their ID to the next node or not.
The properties are checked in the following order

Properties	8 Nodes (Opt)	8 Nodes (Unopt)
ReceivedID is initialized to 0, until ReceivedID is maxID of previous node	True(Instantly)	False (00:04:10,27)
If node1 has the largest ID, it will always eventually become the leader	True(Instantly)	True (00:01:25,56)
If node2 has the largest ID, it will always eventually become the leader	True(Instantly)	True (00:01:26,56)
If node3 has the largest ID, it will always eventually become the leader	True(Instantly)	True (00:01:25,82)
If node4 has the largest ID, it will always eventually become the leader	True(Instantly)	True (00:01:26,29)
If node5 has the largest ID, it will always eventually become the leader	True(Instantly)	True (00:01:26,75)
If node6 has the largest ID, it will always eventually become the leader	True(Instantly)	True (00:01:26,38)
If node7 has the largest ID, it will always eventually become the leader	True(Instantly)	True (00:01:26,55)
If node8 has the largest ID, it will always eventually become the leader	True(Instantly)	True (00:06:45,26)
Eventually there is always a leader	True(Instantly)	True (00:06:40,25)
Not every state has a leader	True(Instantly)	True (Instantly)
If one node is leader, the others are not	True(Instantly)	True (Instantly)
Total Time	8.23 sec	00:27:41,55

Table 3: Verification time for each property in both models

Looking at the two tables, it is clear that our choices affected not only the state space of the model but also the time required to verify them. In Table 3, verifying the optimized model happens almost instantly, approximately only 8.23 seconds were needed, while the unoptimized version required almost half an hour to go through all the properties. Moreover, for the unoptimized version, the first property failed, which was expected as in this case, a node has 50% chance of sending its ID to the next one, and a counterexample was provided by *NuSMV*.

B

a For implementing lossy communication channels, we modified the main module as can be seen in listing 7.

Listing 7: Main Module with Lossy Communication

```
MODULE main
        VAR
                node1 : Node(2);
                node2 : Node(3);
                node3 : Node(1);

        ASSIGN
                next(node2.receivedID) :=
                        case
                                --Left to right: normal behaviour,
                                --  repeated message, broken/zero value
                                TRUE: {node1.maxID, node2.receivedID, 0};
                        esac;

                next(node3.receivedID) :=
                        case
                                --Left to right: normal behaviour,
                                --  repeated message, broken/zero value
                                TRUE: {node2.maxID, node3.receivedID, 0};
                        esac;

                next(node1.receivedID) :=
                        case
                                --Left to right: normal behaviour,
                                --  repeated message, broken/zero value
                                TRUE: {node3.maxID, node1.receivedID, 0};
                        esac;
```

In this model, each node has a 1 in 3 chance of receiving one of the following values: the correct message (the `maxID` of the previous node), a repeat of the last value received, or no value at all (a 0).

b To verify this, we used the same properties as seen in listing 2. Most of these properties fail. Because without fairness, it is possible that all normal messages are lost, which in this case means that only repeats and 0 values can be sent. This causes the model to never have a leader.

c The fairness properties added ensured that the correct communication behavior occurs infinitely often, as shown in listing 8. The only property that does not hold with fairness in place is the one specified on lines 7 through 10 in listing 2, which does not hold because the normal communication behavior is no longer guaranteed. However, with the addition of the fairness properties, all other properties hold, showing that the model always eventually finds on the leader, which is the node with the highest ID.

Listing 8: Fairness Properties Lossy Communication

```
--Fairness for lossy channel
FAIRNESS (node1.receivedID = node3.maxID);
FAIRNESS (node2.receivedID = node1.maxID);
FAIRNESS (node3.receivedID = node2.maxID);
```

Part 2 - Model checking software

Abstraction-Refinement

(a)

The assertions added to set-structure.c can be seen in lines 22-26 in listing 9. This code checks whether any inserted element in the set is the same as any of its predecessors. For this, the program keeps track of how many elements are inserted through the variable `setSize`.

Listing 9: main Function of set-structure.c With Added Assertions

```
1  int main( ) {
2    int n = 0;
3    int set[ SIZE ];
4    int setSize = 0; //only incremented when an element is inserted
5    int x;
6    int y;
7
8    // insert elements in the array
9    int v;
10   for ( v = 0 ; v < SIZE ; v++ ) {
11     // check if the next element to insert exists , if not insert it .
12     int element = 0;
13     if ( !elem_exists( set , n , element ) ) {
14       // parametes are passed by reference
15       set[n] = element;
16       n++;
17       setSize++; //amount of inserted items + 1, so size increases by 1
18     }
19   }
20
21   //code to verify correctness
22   for(x = 0; x < setSize; x++){
23     for(y = 0; y < x; y++){
24       __VERIFIER_assert(set[x] != set[y]);
25     }
26   }
27
28   return 0;
29 }
```

(b)

CPAchecker reports an assertion violation. The counterexample provided can be seen in listing 10. The first 12 lines are initialization. The loop in main starts at line 13. The predicate in line 15 equates to the predicate in the for loop, it signifies that the loop will execute since `v` < 3. Then element is initialized, and `elem_exists` is called. This returns 0, since as can be seen on line 22, the predicate in the for loop in `elem_exists` equates to false, so the for loop is not executed. Because 0 is returned, in lines 24-27 `n` is added to the set, and on line 29 v is incremented.

Then in lines 30-45, the loop is executed again. Notable in this iteration though, is that in line 37 the for loop is executed, and a duplicate is found, but `elem_exists` still returns 0, so n is again added to the set. Lines

45-61 contain another execution of the loop, which shows the same behavior as the previous one. The predicate in line 62 shows that the loop is not executed another time, since `v` is no longer less than 3.

On line 63 the nested assertion loops start. On the first execution of the outer loop, the inner loop (which is first executed on line 66) is not executed, since the for loop predicate is false as can be seen on line 68. The second outer loop execution the inner loop is executed on line 71. Here the assertion sees that the first and second values in the set are both 0 on line 74. This is a violation of the assertion and the end of the trace.

Listing 10: Counterexample Provided by CPAchecker

```
1  INIT GLOBAL VARS
2  void abort();
3  void __VERIFIER_assert(int cond);
4  int __VERIFIER_nondet_int();
5  void __VERIFIER_assume(int);
6  int elem_exists(int set[], int size, int value);
7  int main();
8    int n = 0;
9    int set[3];
10   int x;
11   int y;
12   int v;
13   for
14   v = 0;
15   [v < 3]
16   int element = 0;
17   int __CPAchecker_TMP_0;
18   elem_exists(set, n, element)
19     int i;
20     for
21     i = 0;
22     [!(i < size)]
23     return 0;
24   [__CPAchecker_TMP_0 == 0]
25   set[n] = element;
26   int __CPAchecker_TMP_1 = n;
27   n = n + 1;
28   __CPAchecker_TMP_1;
29   v = v + 1;
30   [v < 3]
31   int element = 0;
32   int __CPAchecker_TMP_0;
33   elem_exists(set, n, element)
34     int i;
35     for
36     i = 0;
37     [i < size]
38     [(set[i]) == value]
39     return 0;
40   [__CPAchecker_TMP_0 == 0]
41   set[n] = element;
42   int __CPAchecker_TMP_1 = n;
43   n = n + 1;
44   __CPAchecker_TMP_1;
45   v = v + 1;
46   [v < 3]
47   int element = 0;
48   int __CPAchecker_TMP_0;
49   elem_exists(set, n, element)
50     int i;
51     for
52     i = 0;
53     [i < size]
54     [(set[i]) == value]
55     return 0;
56   [__CPAchecker_TMP_0 == 0]
57   set[n] = element;
58   int __CPAchecker_TMP_1 = n;
59   n = n + 1;
60   __CPAchecker_TMP_1;
61   v = v + 1;
62   [!(v < 3)]
63   for
64   x = 0;
65   [x < 3]
66   for
67   y = 0;
```

```
    [!(y < x)]
    x = x + 1;
    [x < 3]
    for
    y = 0;
    [y < x]
    __VERIFIER_assert((set[x]) != (set[y]))
        [cond == 0]
        Label: ERROR
```

(c)

The program is fixed by changing the return value in the `elem_exists` loop on line 14 of the program to 1, meaning that `true` is returned when a duplicate value is found. After this fix `cpa-def set-structure.c` reports that the verification result is `TRUE`.

(d)

Instead of assigning 0 to `n` every time, `__VERIFIER_nondet_int()` is assigned to `n` every time. This allows us to check for all possible integer values for `n`. CPAchecker again reports that no assertion violations are found.

Bounded Software Analysis

(i)

We begin our analysis by first running the following command `cbmc simple.c -function add3 -show-vcc`, as specified in the description. With that, we get the following result,

Listing 11: Verification Steps of add3

```
{-1} __CPROVER_alloca_object#1 = NULL
{-2} __CPROVER_dead_object#1 = NULL
{-3} __CPROVER_deallocated#1 = NULL
{-4} __CPROVER_malloc_failure_mode_assert_then_assume#1 = 2
{-5} __CPROVER_malloc_failure_mode_return_null#1 = 1
{-6} __CPROVER_malloc_is_new_array#1 <=> false
{-7} __CPROVER_max_malloc_size#1 = 36028797018963968
{-8} __CPROVER_memory_leak#1 = NULL
{-9} __CPROVER_new_object#1 = NULL
{-10} __CPROVER_next_thread_id#1 = 0
{-11} __CPROVER_next_thread_key!0#1 = 0
{-12} __CPROVER_pipe_count#1 = 0
{-13} __CPROVER_rounding_mode!0#1 = 0
{-14} __CPROVER_thread_id!0#1 = 0
{-15} __CPROVER_thread_key_dtors!0#1 = array_of #source_location=""(NULL)
{-16} __CPROVER_thread_keys!0#1 = array_of #source_location=""(NULL)
{-17} __CPROVER_threads_exited#1 = array_of #source_location=""(false)
{-18} __CPROVER__start::y!0@1#2 = nondet_symbol #source_location=""
identifier="symex::nondet0"
{-19} add3::y!0@1#1 = __CPROVER__start::y!0@1#2
{-20} add3::1::x!0@1#2 = 3
{-21} add3::1::res!0@1#2 = add3::y!0@1#1
{-22} add3::1::i!0@1#2 = 3
{-23} add3::1::res!0@1#3 = add3::1::res!0@1#2 + 1
{-24} add3::1::i!0@1#3 = 2
{-25} add3::1::res!0@1#4 = add3::1::res!0@1#3 + 1
{-26} add3::1::i!0@1#4 = 1
{-27} add3::1::res!0@1#5 = add3::1::res!0@1#4 + 1
{-28} add3::1::i!0@1#5 = 0

{1} add3::1::res!0@1#5 = 3 + add3::y!0@1#1
```

To check this using $Z3$, only the lines starting from 19 until the end were considered. The previous ones are not considered relevant for this exercise and are thus left out. As a result, we get the following program

Listing 12: showSat.txt

```
(declare-const y1 Int)
(declare-const y Int)
(declare-const x Int)
(declare-const i2 Int)
(declare-const i3 Int)
(declare-const i4 Int)
```

```
(declare-const i5 Int)
(declare-const res2 Int)
(declare-const res3 Int)
(declare-const res4 Int)
(declare-const res5 Int)
(assert (= y1 y))
(assert (= x 3))
(assert (= res2 y1))
(assert (= i2 3))
(assert (= res3 1))
(assert (= i3 2))
(assert (= res4 (+ res3 1)))
(assert (= i4 1))
(assert (= res5 (+ res4 1)))
(assert (= i5 0))
(assert (= res5 (+ 3 y1)))
(check-sat)
```

We then continue by inputting this code to the provided *Z*3 tutorial. As output, we receive *sat*, which is interpreted as there is a setting of variables that can satisfy all clauses, along with the given assertion.

(ii)

We begin by running the same command as in the previous exercise. However, in this case, it appears that *CBMC* runs into the issue of repeatedly unwinding loops. Thus we proceed by bounding the unwinding loop, while also providing an upper and lower bound to x, ensuring that the loop will end. After using the command `cbmc simple.c --unwind 31 --bounds-check --unwinding-assertions`, we receive a `SUCCESS` message, showing that our assertions are indeed correct.

Listing 13: The modified add method

```c
int add(int x, int y)
{
  __CPROVER_assume(0 <= x && x <= 30);
  int res = y;
  int i = x;
  while(i != 0){
    res ++;
    i--;
  }
  assert(res == x+y);
  return res;
}
```

(iii)

Since this program uses memory allocation, we decided to check for memory leaks using `cbmc primessieve.c --memory-leak-check`. All these checks passed.
Next we checked whether there were any problems with pointers, so we ran the commmand `cbmc primessieve.c --pointer-check`, and it reported that in multiple places `i` got out of bounds of `primes`. By printing traces (through adding the `--trace` argument), it became clear that number 100 was attempted to be accessed, even though `primes` only contained 0..99. Therefore, line 33 was changed to allocate 101 bools instead of 100, fixing this issue. Also, `primes` was deallocated before being processed in `main()`, so the line `free(primes);` was moved to the end of `main`.
As a next step, we added `assert(isPrime(i))` to be run on every value of `i` that is printed, and `assert(!isPrime(i))` to be run on every value of `i` that is not printed in `main()`. since we can assume that `isPrime` is correct. The traces of this verification step showed us that the loop for printing the primes should start on 2 instead of 1, since 2 is the first prime and 0 and 1 are not checked in `getPrimes`.
With these assertions in place, we can check every number considered for output for whether the program handles it correctly. The program also has a set amount of iterations and is not dependent on the input. Therefore, this allows us to verify all output of this program and thus verify this program fully.

(iv)

The assertion used to verify `primes.c` can be found in listing 14. Inspecting the traces that were the result of verifying this program showed us that it indeed contained bugs.

Listing 14: Main Method of primes.c

```c
int main(){
    int number = nondet_int();
    assert(isPrimeFast(number) == isPrime(number));
}
```

The traces of cases where this assertion failed showed us that input numbers with a factor of 17 caused the assertion to fail. This is because instead of skipping 15, `isPrimeFast` skipped checking 17 instead. Resetting third to 1 instead of to 0 after `third == 2` had been true fixed this problem.
After fixing that problem we found that the assertion was false if the input number equalled 2. This was fixed by putting brackets in the predicate of the second if statement of `isPrimeFast`: `num % 2 == 0 || num % 3 == 0 && num > 3` became `(num % 2 == 0 || num % 3 == 0) && num > 3`.
For finding out how much we could unwind the loops, we used the following observations: if `isPrime` needed n iterations, the amount of iterations of `isPrimeFast` is upper bounded $\frac{\sqrt{n}}{2}$ iterations, since `isPrimeFast` only checks until the root of the input number and takes steps of two. Note, this does not take into account the skipping of numbers when `third == 2`, but this is still a useful approximation.
Using this observation, we ran `cbmc primes.c --unwindset isPrime.0:400 --unwindset isPrimeFast.0:10` which executed in around 1 minute and 30 seconds. The next step, `cbmc primes.c --unwindset isPrime.0:484 --unwindset isPrimeFast.0:11` needed around 2 minutes and 10 seconds.

(v)

For this exercise, in the file `numbers.c`, we test five methods namely, `isPrime(), isSquare(), isPerfect(), isMersenne(), isFibonacci()`.
To start with bug hunting, we first made some assumptions regarding the bounds of n to be between 99 and 1000, to get a smaller range of numbers that we could manually check. However, we saw the despite our assumptions and the bounds already consider from the code ($n > 100$), the first two numbers generated were always either too small or too big.
Initially, the command we used was `cbmc --cover location -unwind 50 -function number_test --show-test-suite numbers.c`. *Location* was first selected as our coverage criteria as we wanted to see whether there were numbers that could satisfy all five methods. Moreover, we took the liberty of selecting the unwinding bound to be 50. As with more loop iterations, it would take a lot of time for the program to be executed.
With the test coverage of the initial version of the code being close to 80%, we were certain that there were bugs in almost all of the methods and thus we started fixing them. The following list contains all the corrections that were made.

- isPrime(): changed the conditions of the `for-loop`, from `i*i < num` to `i*i <= num`
- isMercenne(): changed the resulted value from `n <= res`, to `n == res`, to ensure that a Mercenne number is generated.
- isFibonacci(): chaned the body of the `while-loop`. In more detail, the new value of `a` instead of being assigned to the new value of b, is now assigned to the old one. Thus we now have `a = b - a`.

With those fixes performed, we decided to run again some tests, this time using the following command `cbmc --cover mcdc -unwind 100 -function number_test --show-test-suite numbers.c`. *MCDC* was chosen as the coverage criteria as it can check both the satisfiability of the methods as well as the provided assertions. From the test suite, the following list of numbers was created $-200654992, 1607458931, 936, 937, 939, 511, 225, 233, 127, 144, 610, 496$ in around 15 minutes. We consider the first two numbers as outliers and thus focus on the rest of them. With a manual computation, we got the following results

- 936: is not prime, is not square, is not mercenne,is not perfect, is not fibonacci
- 937: is a prime, is not square, is not perfect, is not mercenne, is not fibonacci
- 939: is not prime, is not square, is not perfect, is not mercene, is not fibonacci
- 511: is not prime, is not square, is not perfect, is mercenne, is not fibonacci
- 225: is not prime, is square, is not perfect, is not mercene, is ot fibonacci
- 233: is a prime, is not square, is not perfect, is not mercenne, is fibonacci
- 127: is a prime, is not square, is not perfect, is mercenne, is not fibonacci
- 144: is not a prime, is square, is not perfect, is not mercene, is fibonacci

- 610: is not a prime, is not square, is not perfect, is not mercene, is fibonacci
- 496: is not a prime, is not square, is perfect, is not mercene, is not fibonacci

Finally, using the following assertions we were able to verify that the program was able to pass all of its tests.

Listing 15: The assertions used

```c
int main(int argc, char *argv[]){
  if(argc < 2){
    printf("Need an argument\n");
    return 1;
  }
  int n = atoi(argv[1]);
  number_test(n);
  assert(number_test(-200654992) == -1);
  assert(number_test(1607458931) == 0);
  assert(number_test(936) == 0);
  assert(number_test(937) == 1);
  assert(number_test(939) == 0);
  assert(number_test(511) == 8);
  assert(number_test(225) == 2);
  assert(number_test(233) == 17);
  assert(number_test(127) == 9);
  assert(number_test(144) == 19);
  assert(number_test(610) == 16);
  assert(number_test(496) == 4);

}
```

An interesting remark is that despite all of the obvious bugs being fixed in the five methods, our test coverage using the `MCDC` option, was again 83.3%. We theorize this has to do with the first two numbers of our set, which are outside the predefined boundaries. This was a recurring issue as we saw in several tests, and had we had more time, it would be interesting to see why it happens.