2IMP10 Program Verification Techniques Assignment I Report

Yu Yang
Student-No. 2051362
y.yang5@student.tue.nl

Dmytro Mudragel
Student-No. 2022060
d.mudragel@student.tue.nl

I. Part I - System modeling and analysis

We have written a set of temporal logic properties that describe the behavior we would expect from the railroad system. The file error-case.smv contains 10 properties we come up with to ensure the safety and liveness of this system. The properties are stated below:

- The signal shows either green or red.
- The signal shows red if passing the signal is unsafe due to occupied tracks or points that are not locked.
- The signal is not stuck on red forever.
- The system never issues conflicting commands.
- The points always follow the given commands.
- Trains always make progress.
- There is a trace in which track T3A is occupied.
- There is a trace in which trains pass each other in the middle.
- orresponding points must always be adjacent to each other.
- Corresponding points must remain in curved positions if a train passes through them.

After running the execution sequence with NuSMV tool we can watch the demonstrated CTL counterexample in which 2 trains move from both sides. The train from the left moves to track T1A when signal S1A is green. The train from the right moves first from track T1B to track T2B when corresponding signals appear green and points P1B and P2A are locked straight. Then the right train goes to track T3B when S2B is red and S3B is green. But in the next step, the right train goes forward through P1B and P2A even when they are still in a state of moving to lock curved and T1A is also occupied. The model also does not consider the time it takes a train to move through the points in a curved state. This problem was fixed in the delay-case.smv file by changing the points module in such a way that the movement between positions takes an arbitrary finite amount of steps (Fig. 1).

In the delay-case scenario, we have another execution sequence demonstrating a CTL counterexample. During the execution, it appears that when the right train moves through T1B when S1B is green to T2B when P1B is locked into a straight state and then moves to T3B when S3B is green and the left train moves through T1A, T2A, and T3A when the corresponding signals S1A, S2A, and S3A are green, the execution sequence shows that the train cannot be on tracks

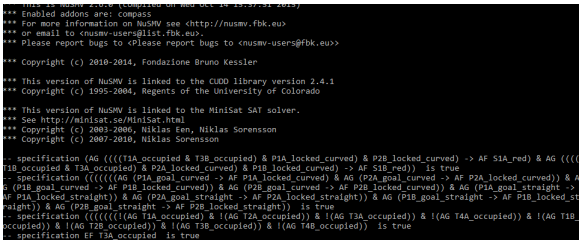

Fig. 1. The cmd.

T3A and T3B correspondingly at the same time. The slightly corrected version of the model is contained in the full-case.smv file.

II. Part II - Model checking software

A. Abstraction-Refinement

a) Add assertions: The modified .c file for this task is "set-list-modified-a.c", with proper assertions added.

The checking is done by iterating over the elements using nested for-loops. Each element is compared with every other element, and the "__VERIFIER_assert" function is used to assert that no two elements in the list are the same. The added code is shown in Fig. 2.

```
// Check that each element is unique
int i, j;
for (i = 0; i < n; i++) {
  for (j = i + 1; j < n; j++) {
    __VERIFIER_assert(list[i] != list[j]);
  }
}
return 0;
```

Fig. 2. The assertions added.

After running the script using the command "./cpa.sh -predicateAnalysis /yy/set-list-modified-a.c", the Linux console reports "Verification result: TRUE. No property violation found by chosen configuration", which means literally.

More details about the verification run can be found in the directory "./output". And a graphical representation of the report is included in the file "./output/Report.html", in which the most useful figure, for now, is a graph of the Control Flow

Automaton (CFA). In it, the nodes represent statements in the program (such as assignments, conditional judgments, etc.), while the edges represent the control flow paths of program execution (such as the path executed when the condition is true/false).

b) Check for all possible insertions: The modified .c file for this task is "set-list-modified-b.c". In this version, the element insertion part is modified by using the "__VERIFIER_nondet_int" function to generate non-deterministic integer values for insertion into the set.

The console reports again "Verification result: TRUE. No property violation found by chosen configuration" after running the script. There are still details that may be useful in the future in the directory "./output".

c) Expand size range: The modified .c file for this task is "set-list-modified-c.c". In this version, The size variable in the program is now generated by the "__VERIFIER_nondet_int()" function, and an assumption about the size is expressed using the "__VERIFIER_assume" function, ensuring that size is within the range of 1 to 4.

The console reports again "Verification result: TRUE".

d) The modified implementation of multi-sets: The implementation .c file ismset-list.c". In order to adjust the original C program according to the task requirements, the following steps were done:

- Introduce a Counter Array: It is the same size as the array "list", storing the number of occurrences of the corresponding "list[i]" element.
- Modify Insertion Logic: Just as the task requires, the insertion logic will be updated so that if an element is not present in the list, it is added, and "counter[i]" is set to 1. If the element is already in the list, the corresponding "counter[i]" is incremented.
- Verification Logic: Add a logic to verify that the sum of all elements in the counter array is equal to the total number of elements inserted, represented by the "size" variable.
- Handle Arbitrary Inputs: The program will be modified to handle arbitrary input elements, not just the value 1.
- Adjust the Old Method: The old function used to check for the presence of 'value' in the array 'list' is adjusted. Now it returns the index if found, and returns -1 if not found.

The console reports again "Verification result: TRUE".

B. Bounded Software Analysis

a) The SAT-formula: The result "simpleSAT.txt" is shown below (and also the file is in the submitted .zip), with basic comments explaining the procedure:

```
; Starting Z3
(echo "Starting Z3 for multiplyBy3
    ↪ verification...")

; Declaration of variables
(declare-const y Int)
(declare-const res Int)
(declare-const i Int)

; Initialization of variables
; x is always 3 in multiplyBy3 function
(assert (= res 0))
(assert (= i 3))

; Loop Iterations
; Since x is 3, the loop runs 3 times. We
    ↪ unroll the loop manually.

; First iteration
(assert (= res y)) ; res = res + y, which
    ↪ is initially 0 + y
(assert (= i 2))   ; i = i - 1

; Second iteration
(assert (= res (+ res y)))
(assert (= i 1))

; Third iteration
(assert (= res (+ res y)))
(assert (= i 0))

; Final Assertion
; The assertion res == 3 * y
(assert (= res (* 3 y)))

; Check for satisfiability
(check-sat)

; If satisfiable, get the model
(get-model)
```

Here is a detailed explanation of how the original program corresponds to the SAT formula:

- Variable Declaration: In "simple.c", the function "multiplyBy3" takes an integer "y" and uses local variables "res" and "i". These are declared in "simpleSAT.txt" as (declare-const y Int), (declare-const res Int), and (declare-const i Int).
- Initialization: The original program initializes res = 0 and i = 3 (as x is always 3). This is mirrored in the SAT formula with (assert (= res 0)) and (assert (= i 3)).
- Loop Unrolling: The function in simple.c contains a loop that adds y to res a total of x (which is 3) times. This loop is unrolled in the SAT formula: after each iteration, res is incremented by y, and i is decremented by 1. This is represented by the series of assertions for each iteration.
- Final Assertion: The program asserts that res == 3 * y at the end of the function. This assertion is represented in the SAT formula by (assert (= res (* 3 y))).
- Checking Satisfiability: Finally, the SAT formula uses (check-sat) to determine if there is a possible assignment

of values to the variables that makes all the assertions true. If so, it means the original program's assertion holds under all conditions.

The Z3 answers' meaning with respect to this verification problem are:

- sat (Satisfiable): If Z3 returns "sat", it means that there is at least one assignment of values to the variables in the original program (y, res, i) that makes all the assertions in the SAT formula true. In other words, the program's logic, as represented by the SAT formula, is consistent, and the assertion res == 3 * y in the multiplyBy3 function holds true under some conditions.
- unsat (Unsatisfiable): If Z3 returns "unsat", it indicates that there are no possible assignments of values to the variables that can make all the assertions in the SAT formula true simultaneously. This implies a logical inconsistency or error in the program's logic. Specifically for the multiplyBy3 function, it means the assertion res == 3 * y cannot be satisfied, indicating a potential bug or flaw in the implementation of the function.

b) Verification of the function "multiply": The command we use is: "cbmc simple.c −−function multiply".

Observe the output after executing the command. The first few lines are the normal setup and preparation phase. When the loop unwinding phase begins, the program first outputs "Unwinding loop multiply.0 iteration 1 file simple.c line 20 function multiply thread 0", then repeatedly outputs this, with the count after "iteration" increasing by 1 each time. Without stopping in a short period, we can only manually terminate the process.

The fact that the loop is continually unwound without reaching a verdict (either proving or disproving the correctness of the assertion) means the analysis is inconclusive. CBMC has not found a counterexample (a case where the assertion is violated), but it also hasn't proven that the assertion holds in all possible cases.

The correctness of a function depends on whether it behaves as intended in all expected scenarios. The assertion in the function (assert(res == x*y)) is a claim that the result of the function (res) always equals the mathematical product of x and y. Without a complete analysis from CBMC, we cannot be certain that this assertion holds in every case.

In short, both CBMC and us can't guarantee the correctness of this function.

Modify the simple.c file to include assumptions, as is shown in Fig. 3 (also a file "simple-assum.c" in the submitted .zip):

Then we use the command "cbmc simple-assum.c –function multiply –unwind 11" to verify the results. This command instructs CBMC not to exceed 11 iterations when unwinding the loop (since the maximum value of x is 10, 11 iterations are sufficient to cover all possible iterations). The result is showned in the Fig. 4.

c) Fix and verify "primessieve.c": To analyze this program for bugs using CBMC, first we need to review the entire source code to understand its logic and structure. This will help in identifying potential areas where bugs might exist. Then if necessary, modify the source code to add assertions. Finally, use CBMC commands to analyze the program, and analyze the output of CBMC to understand the nature of any bugs or issues it reports (involve looking at counterexamples provided by CBMC, which show how a particular assertion can fail).

```
int multiply(int x, int y)
{
  // Assumptions about the inputs
  __CPROVER_assume(x >= 0 && x <= 10);
  __CPROVER_assume(y >= 0 && y <= 10);

  int res = 0;
```

Fig. 3. The assumptions added.

```
** Results:
simple-assum.c function multiply
[multiply.assertion.1] line 28 assertion res == x*y: SUCCESS

simple-assum.c function multiplyBy3
[multiplyBy3.assertion.1] line 13 assertion res == x*y: SUCCESS

** 0 of 2 failed (1 iterations)
VERIFICATION SUCCESSFUL
```

Fig. 4. The verification result.

To be honest, most of the process of finding bugs in this file still relies on manual analysis of the code, rather than just constantly adjusting the assertion for CBMC and relying on counter-example to guide modifications. Therefore, the intermediate process is difficult to be shown, by completely using only cbmc commands and reasoning. The final debug results are as follows:

- The primes array is dynamically allocated in main with a size of 100. This should be 101 since the program is intended to handle prime numbers up to and including 100. Otherwise, there's a risk of buffer overflow.
- The loop conditions and indices in the getPrimes function need to be carefully checked. The condition i*j¡limit might lead to missing the last number (limit) in some cases.
- There's a call to free(primes) inside the getPrimes function. This is problematic because the memory is freed before its contents are used in main. The free should be called in main after the primes array is no longer needed.
- In main, the loop starts from 1 (for (int i=1;i¡=100;i++)). However, the primes array is indexed from 2, so this loop should ideally start from 2.

After modifying the code and adding assertions, the modified .c file is "primessieve-m.c", with comments explaining the specific modification.

Use the CBMC command "cbmc primessieve.c –unwind 102 –trace ". The result is shown in the Fig. 5

This result indicates that all the assertions and memory management checks in your program passed successfully. There were no issues detected within the bounds and conditions tested, suggesting that the program is free from common errors

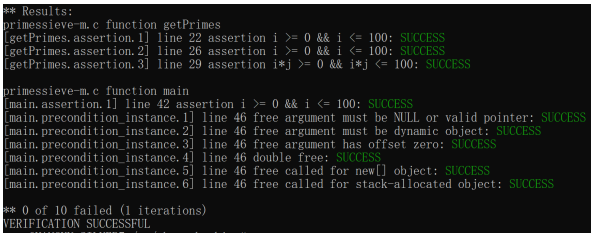

Fig. 5. The bug finding result.

like buffer overflows and invalid memory accesses in those areas. Overall, the verification was successful.

d) About primes.c: To verify that these two functions give the same results using CBMC and non-determinism, we first modify the main function to include a non-deterministic integer input and assertions to check if both functions return the same result for this input (shown in primes-m.c).

By using the command "cbmc primes-m.c –trace –unwind <bound>" and set the bound to 10, we obtain the following result (Fig. 6):

```
** Results:
primes-m.c function main
[main.assertion.1] line 49 assertion result1 == result2: FAILURE

Trace for main.assertion.1:

State 21 file primes-m.c function main line 43 thread 0
------------------------------------------------------
  x=0 (00000000 00000000 00000000 00000000)

State 22 file primes-m.c function main line 43 thread 0
------------------------------------------------------
  x=2 (00000000 00000000 00000000 00000010)

Assumption:
  file primes-m.c line 44 function main
  x >= 0

State 24 file primes-m.c function main line 46 thread 0
------------------------------------------------------
  result1=FALSE (00000000)

State 27 file primes-m.c function main line 46 thread 0
------------------------------------------------------
  num=2 (00000000 00000000 00000000 00000010)

State 33 file primes-m.c function main line 46 thread 0
------------------------------------------------------
  result1=TRUE (00000001)

State 34 file primes-m.c function main line 47 thread 0
------------------------------------------------------
  result2=FALSE (00000000)

State 37 file primes-m.c function main line 47 thread 0
------------------------------------------------------
  num=2 (00000000 00000000 00000000 00000010)

State 43 file primes-m.c function main line 47 thread 0
------------------------------------------------------
  result2=FALSE (00000000)

Violated property:
  file primes-m.c function main line 49 thread 0
  assertion result1 == result2
  result1 == result2
```

Fig. 6. The result.

The CBMC trace indicates a discrepancy between the outputs of isPrime and isPrimeFast when x=2. Specifically:

- State 33: isPrime(2) returns TRUE, indicating that 2 is correctly identified as a prime number by isPrime.
- State 43: isPrimeFast(2) returns FALSE, suggesting that 2 is incorrectly identified as a non-prime number by isPrimeFast.

This inconsistency is the reason for the assertion failure at line 49 (assert(result1 == result2)).

Going back to the code, we discover that in the isPrimeFast function, the condition if (num % 2 == 0 —— num % 3 == 0 num > 3) is problematic. For num = 2, this condition incorrectly returns false, as 2 % 2 == 0. However, 2 is a prime number and should not be discarded by this condition.

By adding if (num == 2) returns true; (shown in primes-m-1.c), we ensure that the function correctly returns true for 2, aligning with the behavior of isPrime. After making this change, we run CBMC again to verify that the assertion holds for the new range of values, and the result is "primes-m-1.c function main [main.assertion.1] line 50 assertion result1 == result2: SUCCESS".

As the result suggests: "Runtime decision procedure: 0.160104s" by using "–unwind 10". Then to limit the time to two minutes, we should set the unwind bound to about 750.

e) About numbers.c: Test Cases for number_test:

- Prime Numbers:
  Input: 7 (Prime)
  Expected Output: Prime flag set, messages indicating it's prime.
  Input: 4 (Not Prime)
  Expected Output: Prime flag not set, messages indicating it's not prime.
- Square Numbers:
  Input: 16 (Square)
  Expected Output: Square flag set, messages indicating it's square.
  Input: 17 (Not Square)
  Expected Output: Square flag not set, messages indicating it's not square.
- Perfect Numbers:
  Input: 28 (Perfect)
  Expected Output: Perfect flag set, messages indicating it's perfect.
  Input: 29 (Not Perfect)
  Expected Output: Perfect flag not set, messages indicating it's not perfect.
- Mersenne Numbers:
  Input: 31 (Mersenne)
  Expected Output: Mersenne flag set, messages indicating it's Mersenne.
  Input: 32 (Not Mersenne)
  Expected Output: Mersenne flag not set, messages indicating it's not Mersenne.
- Fibonacci Numbers:
  Input: 13 (Fibonacci)

Expected Output: Fibonacci flag set, messages indicating it's Fibonacci.
Input: 14 (Not Fibonacci)
Expected Output: Fibonacci flag not set, messages indicating it's not Fibonacci.

- Edge Cases:
  Input: 1 (Special Case)
  Expected Output: Proper handling of edge case.
  Input: 100 (Boundary)
  Expected Output: Test for boundary condition.

These test cases are mainly figured out by manually review the code, and only a few steps with the CBMC assistance. Unfortunately, the specific cbmc command is quite similar with the former ones, and didn't give back a good result, so omitted here.

Bugs discovered and modified in "numbers-m.c", with one bug not knowing what to do with it.

For this program, a combination of branch coverage and boundary value analysis would be suitable. Branch coverage ensures that each conditional branch is executed at least once, while boundary value analysis tests the limits of input values (like the smallest prime number, the edge cases of Fibonacci numbers, etc.).

Comparatively, statement coverage might not be sufficient as it would not necessarily cover all logical branches of the code. For instance, it might miss testing a number that is both a Fibonacci and a prime number.

Achieving 100% coverage might be challenging due to the complexity and variety of mathematical properties being tested. Some logical branches might be difficult to reach with typical test cases, such as a number that is both prime and square, which the program asserts should not happen.