Assignment 6
Catch Me If You Can

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Tuesday, April 13, 2021
90 pts

This assignment is due on the above date and it must be submitted electronically on Gradescope. Please carefully read the policies on collaboration and credit on the course web pages at http://www.cs.cmu.edu/~15414/assignments.html.

What To Hand In

You should hand in the following files on Gradescope:

- Submit the file `asst6.zip` to Assignment 6 (Code). You can generate this file by running `make handin`. This will include your code `ubarray.c` and the output of CBMC for the different tasks with names `ubarray-*.txt` (more details in Section 2).
- Submit a PDF containing your answers to the written questions to Assignment 6 (Written). You may use the file `asst6-sol.tex` as a template and submit `asst6-sol.pdf`.

  Make sure your code and your PDF solution files are up to date before you create the handin file.

Using LaTeX

We prefer the answer to your written questions to be typeset in LaTeX, but as long as you hand in a readable PDF with your solutions it is not a requirement. We package the assignment source `asst6.tex` and a solution template `asst6-sol.tex` in the handout to get you started on this.

1 Theory Solving (30 pts)

Task 1 (15 pts). **Positive reasoning**. The DPLL(T) algorithm given on page 8 of lecture 16 constructs a conjunctive T-formula to send to the theory solver using an interpretation returned by the SAT solver as follows:

$$\psi \equiv B^{-1}\left(\bigwedge_{i=1}^{n} P_i \leftrightarrow I(P_i)\right)$$

A potential optimization would be to replace this step with one that only sends the conjunction of theory literals that the SAT interpretation assigns *true*:

$$\psi \equiv B^{-1}\left(\bigwedge_{\{i:I(P_i)=\mathit{true}\}} P_i\right)$$

Unfortunately, this is not sound. Give an example formula for which this optimization would reduce the number of times DPLL(T) must iterate, but would cause it to yield an incorrect answer.

Task 2 (15 pts). **Congruence closure**. Use the congruence closure algorithm for T_E to determine the satisfiability of the following Σ_E-formulae:

1. $f(g(x)) = g(f(x)) \land f(g(f(y))) = x \land f(y) = x \land g(f(x)) \neq x$
2. $f(f(f(a))) = f(a) \land f(f(a)) = a \land f(a) \neq a$

2 Bounded Model Checking (60 pts)

In this assignment, we ask you to use the C Bounded Model Checker (CBMC) for verifying safety properties of C code up to some underapproximation. We provide a buggy implementation of unbounded arrays (`ubarray.c`). Your job is to write a contract for the functionality of each function. You can use the model checker to check if this contract is satisfied and to identify other bugs (e.g., out-of-bounds accesses). You should then repair these bugs and show that these repairs are sufficient and that CBMC does not report any more bugs. This assignment makes use of a new tool. Although it should not require as much time to become familiar with it as it did learning to use Why3, it is important to take some time before attempting the problems in this assignment learning how to use the tool.

2.1 Setting up the model checker

You will need to download CBMC and install it in your system. We recommend downloading the binaries for either x86 Linux, Windows, or MacOS that are available at `https://www.cprover.org/cbmc/`. If you have any issue running CBMC in your system, then we recommend using the VM we provided and installing CBMC in that VM instead. Alternatively, you can also download the x86 Linux version to your AFS area, which can be accessed on an SSH session to `linux.andrew.cmu.edu` or `linux.gp.cs.cmu.edu`, or on a CS cluster machine. You can download CBMC with a wget command "wget `https://www.cprover.org/cbmc/download/cbmc-5-11-linux-64.tgz`". These machines are not meant to be used for resource-intensive jobs but running CBMC for this assignment should take less than 1 minute per task. If you have problems setting up CBMC, please contact the course staff as soon as possible.

2.2 Using CBMC

In lecture 15, we went over different examples of using CBMC. Here, we cover some relevant features of CBMC.

Non-determinism. To prove that a program is correct for any given input, one must consider all possible values. This can be achieved using non-deterministic variables and one way of doing this with CBMC is by declaring a function with no body in the source file being analyzed, i.e., an external function. To deal with external code without making unwarranted assumptions, CBMC assumes that any values returned from such code can take any value. In the `ubarray.c` file, we already include two functions `nondet_int()` and `nondet_char` that will return an integer and a char, respectively. These functions should be used to simulate non-deterministic values.

Unwinding loops. CBMC works by unrolling each loop to a given depth. To unroll all loops with a given depth you can use the command line option `--unwind N`. This will unroll each loop `N` times. Furthermore, if we tell CBMC to insert unwinding assertions by passing the command-line argument `--unwinding-assertions`, then we can conclude that there are no bugs up to the given unwinding depth, and that the unwinding depth is sufficient to cover all possible paths of the programs. **Note:** If you don't specify an unwinding depth then CBMC will loop forever! If this happens, just terminate CBMC with CTRL+C.

Requires and ensures. CBMC does not have support for pre- and post- conditions like Why3. However, two keywords are useful for writing your contract and requiring some restrictions on the value of variables.

- `__CPROVER_assume` lets you assume that a given variable has some restrictions. For instance, let's say that you declare a nondeterministic integer variable i and want to impose that this variable can only have values between 0 and 10. Then you could write:

  ```
  __CPROVER_assume(i >= 0 && i <= 10);
  ```

  CBMC will interpret this line of code and restrict the possible values of i. Note that you can write these assume statements in any part of your code.

- `__CPROVER_assert` (or just `assert`) lets you assert that a condition holds. When combined with non-deterministic variables, it is able to show that this condition holds for all possible inputs. For instance, you can write:

  ```
  assert(i >= 0); (or __CPROVER_assert(i >= 0);)
  ```

  to prove that i will always be larger or equal to 0 at that location of your program.

These are the main keywords that you will use to model your verification conditions.
Useful command line options. Other useful command line options that you should use for this assignment:

- `--function function_name` runs the verification just for the specified function. Otherwise, CBMC will try to run the verification for the `main` function.
- `--pointer-check` enable pointer checks. This option implicitly creates contracts that will verify if pointer checks are done correctly.

- `--trace` gives a counterexample trace for failed properties. This will be useful to debug the current code and fix the bug.
- `--drop-unused-functions` drop functions trivially unreachable from main function which makes it easier to read the output.
- `--slice-formula` remove assignments unrelated to property which makes verification faster.

CBMC has many other options and you are free to explore them but for this assignment, we recommend you to run the following command line:

```
$ cbmc --function function_name ubarray.c --unwind N --unwinding-assertions
       --pointer-check --trace --slice-formula --drop-unused-functions
```

You should replace `function_name` with the corresponding function you want to analyze and `N` with a suitable unwind depth.

Saving the output. To save the output of CBMC, just redirect the output to a file. For instance, for Task 4 you should redirect the output of CBMC with the buggy trace to a file called `ubarray-4.txt`. If you want to save multiple traces, just name your files `ubarray-4-1.txt`, `ubarray-4-2.txt`, etc.

To redirect the output to a file just use `cbmc ... > filename`. For instance:

```
$ cbmc --function ubarray_harness_init ubarray.c --unwind N --unwinding-assertions
  --pointer-check --trace --slice-formula --drop-unused-functions > ubarray-4.txt
```

WARNING: Be careful when redirecting files since if the file already exists it will be overwritten.

Running CBMC. To make easier to run CBMC, we provide a simple bash script `cbmc.sh` (for Linux and MacOS systems) that will run the previous command with all the described options.

```
Usage: ./cbmc.sh function_name unwind_depth output_file
       function_name = name of the function to analyze
       unwind_depth  = number of times each loop is unwind
       output_file (optional) = save output of CBMC to this file
```

This script assumes that you have a cbmc binary in the current directory. If you have it installed somewhere else, then you can change the variable `CBMCLOC` in the script.

2.3 Gotta Catch 'Em All

A buggy implementation of unbounded arrays is given in `ubarray.c`. A constant `MAX_ARRAY_BOUND` has been defined and set to 4. This small limit speeds up verification but in practice CBMC would be able to solve larger limits. For this assignment, do not modify this constant. We provide some helper functions `valid_ubarray` and `copy_data`. You may write any additional helper functions that you think are useful. Your fixes should aim to satisfy the following informal specification: whenever the array structure given to a function satisfies `valid_ubarray`, then the function should behave as expected (without terminating the program), returning a correct result as expected and a `valid_ubarray` where applicable. Rather than terminating the program, your implementation should return an error code whenever possible, or NULL where appropriate.

Task 3 (5 pts). What is the minimum unwind depth that is sufficient for verifying all paths without reaching unwinding assertions? Explain your reasoning. This part should be submitted to the written assignment.

In general, we recommend you start by running CBMC on each function (e.g., `uba_add`) to find any bugs that are related to pointer checks associated with these functions.

Task 4 (15 pts). `ubarray_harness_init`:

- Write a test harness for the initialization of an unbounded array with *exactly one* nondeterministic element. [1]
- Write a contract to verify the functional correctness of this initialization.
- Fix any bugs that you may find in `ubarray_new` or `uba_add` when running CBMC. Iterate until no more bugs are reported by CBMC.
- Save the output of CBMC with the counterexamples that you found to `ubarray-4.txt`.

Task 5 (10 pts). `ubarray_harness_resize:`

- Write a test harness for the `uba_resize` function.
- Write a contract to verify the functional correctness of this function.
- Fix any bugs that you may find in `uba_resize` when running CBMC. Iterate until no more bugs are reported by CBMC.
- Save the output of CBMC with the counterexamples that you found to `ubarray-5.txt`.

Task 6 (10 pts). `ubarray_harness_get:`

- Write a test harness for the `uba_get` function.
- Write a contract to verify the functional correctness of this function.
- Fix any bugs that you may find in `uba_get` when running CBMC. Iterate until no more bugs are reported by CBMC.
- Save the output of CBMC with the counterexamples that you found to `ubarray-6.txt`.

Task 7 (10 pts). `ubarray_harness_set:`

- Write a test harness for the `uba_set` function.
- Write a contract to verify the functional correctness of this function.
- Fix any bugs that you may find in `uba_set` when running CBMC. Iterate until no more bugs are reported by CBMC.
- Save the output of CBMC with the counterexamples that you found to `ubarray-7.txt`.

Task 8 (10 pts). `ubarray_harness_rem:`

- Write a test harness for the `uba_rem` function.

[1] In practice, a more general harness could be done by initializing the unbounded array with a non-deterministic number of elements. However, verification time will grow with the number of elements. To keep things simple and fast for verification, we restrict the initialization to a single element.

- Write a contract to verify the functional correctness of this function.
- Fix any bugs that you may find in `uba_rem` when running CBMC. Iterate until no more bugs are reported by CBMC.
- Save the output of CBMC with the counterexamples that you found to `ubarray-8.txt`.