Bounded model checking C code

1 A brief introduction to the C Bounded Model Checker

1.1 Setting up the model checker

You do not need to install any software to complete this lab. The course staff has set up an installation of CBMC on AFS, 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. The binary is located at `/afs/cs.cmu.edu/academic/class/15414-f17/cbmc/cbmc`, and should run as-is. We recommend adding `/afs/cs.cmu.edu/academic/class/15414-f17/cbmc` to your `PATH` variable, so that you can run the model checker by simply invoking `cbmc`.

Before getting started with the lab, please test the command. If you do not see the following output, let the course staff know immediately.

```
andrewid@linux1:~$ cbmc
CBMC version 5.8 64-bit x86_64 linux
Please provide a program to verify
```

Installing CBMC on your own machine. While it is possible to install CBMC on your own machine, we do not recommend doing so as this may lead to delays in your completing the assignment. Because an installation is available on AFS, the course staff will not be able to spend time helping you debug a failed installation on a personal machine. Binaries and installation instructions for Windows, MacOS, and Windows are available at `http://www.cprover.org/cbmc/`. However, if you attempt to use these and installation does not work on your machine immediately, please revert to using the AFS installation.

1.2 Using CBMC

We will illustrate the use of CBMC to find bugs or verify their absence by applying it to the toy example from Lecture 19, which is shown in Figure 1. The first thing to note is that we have placed the code in the `main` function. Being a C function, CBMC expects the entry point of the program to reside in `main`. If the file given to CBMC has no `main`, and an alternate entry point isn't provided with the `function` command-line argument, then CBMC will finish without verifying anything.

```
int N, x;
int main() {
  int i = N;
  while(0 <= x && x < N) {
    i = i - 1;
    x = x + 1;
  }
  __CPROVER_assert(0 <= i, "postcondition");
}
```

Figure 1: Toy example program from Lecture 19

The next thing to note is the call to `__CPROVER_assert`. This is the primary form of user-defined specification supported by CBMC. When the model checker is invoked, it will attempt to verify that the condition given as the first argument holds on all paths up to a specified bound. If no bound is given on the command line, CBMC will attempt to infer an upper bound on the program's execution depth, and verify the program after unwinding. The second argument to `__CPROVER_assert` is a diagnostic string that will be reported in the results if CBMC finds a counterexample for the assertion.

Let's run the model checker on this example. For now, we will not specify an unwinding bound, and let CBMC try to infer the bound on its own. The results are shown in Figure 2. Surprisingly, we see that CBMC concluded with a `VERIFICATION SUCCESSFUL` message! This is contrary to what we saw in class when we worked this example out, where we found a counterexample at $\texttt{N} = -1, \texttt{x} = 0$. Why didn't CBMC find this bug? Notice that `N` and `x` are not initialized. Because they are static globals, CBMC assumes that they are initialized to 0 by default. This is not necessarily a safe assumption to make, and there are two primary ways to address it.

The first approach is to pass the command-line argument `nondet-static`, which tells CBMC to assume that any variable with static lifetime is initialized to a nondeterministic value. The second approach is to introduce the nondeterminism ourselves. We can do this 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. Figure 3 is updated

```
andrewid@linux1:~$ cbmc toy1.c
CBMC version 5.8 64-bit x86_64 macos
Parsing toy1.c
Converting
Type-checking toy1
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 38 steps
simple slicing removed 0 assignments
Generated 1 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
```

Figure 2: CBMC output on toy example program from Figure 1

```
int nondet_int();
int N, x;
int main() {
  N = nondet_int();
  x = nondet_int();
  int i = N;
  while(0 <= x && x < N) {
    i = i - 1;
    x = x + 1;
  }
  __CPROVER_assert(0 <= i, "postcondition");
}
```

Figure 3: Toy example with nondeterministic initialization.

to reflect this approach, by declaring an external function `nondet_int` and calling it to initialize `N` and `x` at the beginning of `main`.

If we run the model checker again on the updated program, we see a large amount of output that fails to terminate.

```
andrewid@linux1:~$ cbmc toy1.c
CBMC version 5.8 64-bit x86_64 linux
...
Unwinding loop main.0 iteration 1785 file toy1.c line 7 function main thread 0
Unwinding loop main.0 iteration 1786 file toy1.c line 7 function main thread 0
Unwinding loop main.0 iteration 1787 file toy1.c line 7 function main thread 0
...
```

This is due to the fact that we did not specify an unwinding depth; CBMC attempts to find a bound on the depth of the loop, but is unable to do so because `N` is initialized nondeterministically to take any integer value. We address this by passing `--unwind 3` on the command line, telling CBMC to unroll the loop at most three times. We now see the following (note that some of the output has been omitted to save space).

```
andrewid@linux1:~$ cbmc toy1.c --unwind 3
Solving with MiniSAT 2.2.1 with simplifier
1019 variables, 3727 clauses
SAT checker: instance is SATISFIABLE
Runtime decision procedure: 0.006s
** Results:
[main.assertion.1] postcondition: FAILURE
** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```

This is the result we expected to see, knowing that the assertion should not always hold. We see that CBMC generated a SAT instance with 1019 variables and 3727 clauses, and found it to be satisfiable. This corresponds to a violation of the assertion labeled `postcondition`, which is the property we wished to check.

In order to see a counterexample for this bug, we pass the command-line argument `--trace`. The output is shown below.

```
andrewid@linux1:~$ cbmc toy1.c --unwind 3 --trace
Trace for main.assertion.1:

State 20 file toy1.c line 4 function main thread 0
----------------------------------------------------
  N=-1073741824 (11000000000000000000000000000000)
State 21 file toy1.c line 5 function main thread 0
----------------------------------------------------
  x=-1073741825 (10111111111111111111111111111111)
State 22 file toy1.c line 6 function main thread 0
----------------------------------------------------
  i=0 (00000000000000000000000000000000)
State 23 file toy1.c line 6 function main thread 0
----------------------------------------------------
  i=-1073741824 (11000000000000000000000000000000)

Violated property:
  file toy1.c line 11 function main
  postcondition
  0 <= i
```

Among other things, the counterexample trace tells us that `N` is initialized to -1073741824 on line 4 in `main`, `i` is initialized to 0 on line 6, and subsequently updated to the same value as `N` on the same line. The trace then ends with the violated property, bypassing the loop entirely. Note that `i` is updated twice; the first instance corresponds to the declaration of `i`, where it is given the default value 0. The second corresponds to the initialization to `N`, which is the source of the bug.

Environment assumptions. Before moving on, we introduce the specification primitive `__CPROVER_assume(Q)`. Like `__CPROVER_assert`, the argument to `__CPROVER_assume` is a Boolean condition. CBMC interprets a call to this function slightly differently: any path that does not satisfy the condition passed to `__CPROVER_assume` is discarded from the analysis. Importantly, if such a path later contains a safety violation, it is not reported in the results. This can be useful when modeling assumptions about the environment, for example if we have reason to believe that the arguments passed to a function will always satisfy certain conditions.

In the present example, if we place a call to `__CPROVER_assume(0 <= N && N < 3);` immediately after the intitialization of `N` on line 4, then CBMC will return `VERIFICATION SUCCESSFUL`. 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 for exhaustive verification.

```
andrewid@linux1:~$ cbmc toy1.c --unwind 3 --unwinding-assertions
** Results:
[main.assertion.1] postcondition: SUCCESS
[main.unwind.0] unwinding assertion loop 0: SUCCESS
** 0 of 2 failed (1 iteration)
```

Array bounds and pointer checking. This lab will have you verify the absence of memory errors in two C programs. It is possible to do this by inserting appropriate calls to `__CPROVER_assert` before array and pointer accesses, as in the following example.

```
char buf[100];
int i = get_index();
__CPROVER_assert(0 <= i && i < 100, "array bounds check");
printf("%d", buf[i]);
```

However, CBMC will automatically insert these checks for you when given the command-line arguments `--bounds-check` and `--pointer-check`. You may want to use the command line argument `--slice-formula` to speed model checking. `--trace` provides extra information for any bugs found.Before starting the lab, please read the tutorial at `http://www.cprover.org/cprover-manual/cbmc.shtml`, which provides more information about the use of these arguments. Further documentation is available at `http:`

//www.cprover.org/cprover-manual/.