15-414: Bug Catching: Automated Program Verification

Lecture Notes on Bounded Model Checking

Ruben Martins

Carnegie Mellon University
Lecture 20
Tuesday, April 2, 2024

1 Introduction

In the previous lecture, we have seen that we can use SMT to show the equivalence between two programs. In this lecture, we will show how we can use SAT solvers to either verify that some program is correct or find a counterexample that shows inputs to the program that may trigger some bug. One approach that can leverage SAT technology is through *bounded model checking*. There are several challenges when trying to verify programs, foremost among them the fact state-space of programs may be infinite. Bounded model checking computes an *under*approximation of the reachable state-space by assuming a fixed computation depth in advance, and treating paths within this depth limit symbolically to explore all possible states. While this approach has its limitations, it can be effectively used in practice and it is a useful technique to have in our collection of verification techniques.

Learning Goals

- **Bounded Model Checking** (BMC) considers an **underapproximation** of all possible traces of a program.
- BMC handles **loops** by **unrolling** them with a predefined depth. If the loop is completely unrolled then it is possible to prove verification conditions. Otherwise, BMC is more useful as a **bug catching** procedure.
- **Counterexamples** given by BMC can be helpful for programmers to **fix potential bugs**.

- **Programs can be encoded into propositional logic** by unrolling loops and using SSA to guarantee that each variable is only assigned once.
- **CBMC is a powerful BMC tool** for C programs that can handle arbitrary C programs.

2 Bounded Model Checking

Bounded model checking considers an *underapproximation* of all possible traces of a program. In particular, not all possible traces will appear in the approximation, but all those that do appear are certain to be in the true trace semantics. In principle Bounded Model Checking (BMC) can be used to verify arbitrary properties, but it is most commonly used to check reachability invariants of the form $\Box terminated \to P$, and we will focus on this case for the remainder of these lecture notes.

2.1 Trace Semantics

Let first formalize the notion of trace semantics of a program.

Definition 1 (Trace semantics of programs)**.** The *trace semantics, $\tau(\alpha)$, of a program α*, is the set of all its possible traces and is defined inductively as follows:

1. $\tau(x := e) = \{(\omega, \nu) \ : \ \nu = \omega$ except that $\nu(x) = \omega[\![e]\!]$ for $\omega \in \mathcal{S}\}$
2. $\tau(?Q) = \{(\omega) \ : \ \omega \models Q\} \cup \{(\omega, \Lambda) \ : \ \omega \not\models Q\}$
3. $\tau(\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta) = \{\sigma \in \tau(\alpha) \ : \ \sigma_0 \models Q\} \cup \{\sigma \in \tau(\beta) \ : \ \sigma_0 \not\models Q\}$
4. $\tau(\alpha;\beta) = \{\sigma \circ \varsigma \ : \ \sigma \in \tau(\alpha)\,,\ \varsigma \in \tau(\beta)\}$;
   the composition of $\sigma = (\sigma_0, \sigma_1, \sigma_2, \dots)$ and $\varsigma = (\varsigma_0, \varsigma_1, \varsigma_2, \dots)$ is
   $$\sigma \circ \varsigma := \begin{cases} (\sigma_0, \dots, \sigma_n, \varsigma_1, \varsigma_2, \dots) & \text{if } \sigma \text{ terminates in } \sigma_n \text{ and } \sigma_n = \varsigma_0 \\ \sigma & \text{if } \sigma \text{ does not terminate} \\ \text{not defined} & \text{otherwise} \end{cases}$$
5. $\tau(\texttt{while}(Q)\,\alpha) = \{\sigma^{(0)} \circ \sigma^{(1)} \circ \dots \circ \sigma^{(n)} \ :$ for some $n \geq 0$ such that for all $0 \leq i < n$: ① the loop condition is true $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]$ and ③ $\sigma^{(n)}$ either does not terminate or it terminates in $\sigma_m^{(n)}$ and $\sigma_m^{(n)} \not\models Q$ in the end$\}$
   $\cup \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \dots \ :$ for all $i \in \mathbb{N}$: ① $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]\}$
   $\cup \{(\omega) \ : \ \omega \not\models Q\}$
   That is, the loop either runs a nonzero finite number of times with the last iteration either terminating or running forever, or the loop itself repeats infinitely often and never stops, or the loop does not even run a single time.
6. $\tau(\alpha^*) = \bigcup_{n\in\mathbb{N}} \tau(\alpha^n)$ where $\alpha^{n+1} \overset{\text{def}}{\equiv} (\alpha^n;\alpha)$ for $n \geq 1$, and $\alpha^1 \overset{\text{def}}{\equiv} \alpha$ and $\alpha^0 \overset{\text{def}}{\equiv} (?true)$.

2.2 Underapproximation of Trace Semantics

BMC computes an *underapproximation* of $\tau(\alpha)$ by assuming that all loops in the program are unrolled to some fixed, pre-determined finite depth k. There are two useful ways to think about this operation. The first, which might have occurred to you naturally before having taken this course, is to transform the original program, which may contain loops, into a loop-free program using the bound k. Consider the following [unwind] axiom, which allows us to replace a loop with a conditional statement, within which is a copy of the original loop.

([unwind]) $[\texttt{while}(Q)\,\alpha]P \leftrightarrow [\texttt{if}(Q)\,\{\alpha;\texttt{while}(Q)\,\alpha\}]P$

Axiom [unwind] tells us that it is perfectly acceptable when reasoning about a safety property to replace `while` statements with `if` statements in this way. To perform bounded model checking, we first apply [unwind] to each loop in the program up to k times. When we are finished, we replace any remaining loops with `skip` statements (or equivalently, $?Q$).

Let's see an example. Consider the following program, which doesn't do anything useful but is simple enough to illustrate the key ideas here.

```
1   i <- N;
2   while(0 <= x < N) {
3     i <- i - 1;
4     x <- x + 1;
5   }
```

Suppose that we want to check that $\Box \mathit{terminated} \rightarrow 0 \leq i$ holds, up to a bound of $k = 2$. We begin by applying [unwind] twice to the loop. When we stop, we replace the remaining loop with an empty statement.

```
1   i <- N;
2   if(0 <= x < N) {
3     i <- i - 1;
4     x <- x + 1;
5     if(0 <= x < N) {
6       i <- i - 1;
7       x <- x + 1;
8     }
9   }
```

With all of the loops removed from the program, verification is straightforward using the deductive techniques covered earlier in the semester: the formula we need to prove is just $[\alpha]0 \leq i$. In particular, we can apply [if], [;], and [$\leftarrow$] repeatedly until we are left with a term containing no modalities and literals involving only integer operations. In the current example, we have the following after applying the necessary steps.

$$\begin{array}{ll} & (\neg(0 \leq x < N) \rightarrow 0 \leq N) \\ \wedge & (0 \leq x < N \rightarrow \neg(0 \leq x+1 < N) \rightarrow 0 \leq N-1) \\ \wedge & (0 \leq x < N \rightarrow 0 \leq x+1 < N \rightarrow 0 \leq N-2) \end{array}$$

If this formula is valid (which it is not), then the original property holds. Notice that there are three clauses in this formula, one for each possible path through the program after unwinding at $k = 2$. What bounded model checking essentially does is to "symbolically" evaluate each path through the program up to the unwinding depth. Each path corresponds to a conjunctive clause so that if the formula is not valid, there will be a clause that the model checker can identify as being at fault. The corresponding path gives a counterexample and a satisfying solution to its negation a valuation of the input variables that will violate the property.

In the example above, we see that the first clause is already invalid. We negate it to look for a satisfying solution:

$$\neg(\neg(0 \leq x < N) \rightarrow 0 \leq N) \leftrightarrow (\neg(0 \leq x < N) \wedge \neg(0 \leq N))$$

A satisfying solution to the above is $x = 0, N = -1$. Notice that if we run the original program starting in a state that matches this assignment, then it terminates immediately without executing the loop, leaving $i = -1$.

Limitations Because bounded model checking is an underapproximation, it might not consider some traces that are in the trace semantics of the program. This means that if it does not find a property violation, we cannot necessarily conclude that the program is bug-free. However, in some cases, we can. Consider the following variation of the above example.

```
1    i <- 3;
2    while(0 <= x < 3) {
3      i <- i - 1;
4      x <- x + 1;
5    }
```

While a bound of $k = 2$ is insufficient to conclude that there are no bugs in this program, setting $k = 3$ is in fact sufficient. Furthermore, we can modify the unwinding process slightly so that if no bugs are found up to a particular depth, *and* we've chosen a sufficiently large enough k, we will conclude as much. Likewise, if no bugs are found but we chose an inadequately large k, we'll know that to be the case as well.

The approach uses what are called *unwinding assertions*. Whereas before when we finished applying [unwind], we replaced the remaining loop with an empty statement, now we will replace it with a statement that violates safety if the unwinding is insufficient. In the above example, we would have the following for $k = 2$.

```
1    i <- 3;
2    if(0 <= x < 3) {
3      i <- i - 1;
4      x <- x + 1;
5      if(0 <= x < 3) {
6        i <- i - 1;
7        x <- x + 1;
8        assert(0 > x || x >= 3); //conditional negation
9      }
10   }
```

Although we haven't talked about assertions before, we can model them using existing constructs and safety properties. To check that an assertion isn't violated, we replace the `assert` statement with a corresponding conditional, which makes an assignment to a special variable whenever its condition is true.

```
error <- 0;
i <- 3;
if(0 <= x < 3) {
  i <- i - 1;
  x <- x + 1;
  if(0 <= x < 3) {
    i <- i - 1;
    x <- x + 1;
    if(0 <= x < 3) error <- 1;
  }
}
```

We can then check the validity of the formula $[\alpha]\texttt{error} = 0$. In this case, the formula would be invalid, because x is at most 2 on the path containing the `assert`. This means that the unwinding assertion fails to hold, and so we should not conclude that the program is bug-free by unwinding up to $k = 2$.

3 From Program to Propositional Logic

In the previous section, we have seen the intuition behind extraction a formula from a program. In practice, these formulas are converted into satisfiability problems and given to a decision procedure. Consider the following program that computes the absolute number. For simplicity, assume that our integers can only take values $\{-1, 0, 1\}$.

```
int a;
int b;
if (a < 0) b = -a;
else b = a;
assert (b >= 0 && (b == a || b == -a));
```

To encode this program to propositional logic, we need to encode integers into propositional logic. For this example, we will use a unary encoding where each possible integer value for each variable will be represented by a Boolean variable:

- $a = \{a_{-1}, a_0, a_1\}$, where $a_i = i$ iff $a = i$ with $-1 \leq i \leq 1$
- $b = \{b_{-1}, b_0, b_1\}$, where $b_i = i$ iff $b = i$ with $-1 \leq i \leq 1$

Since exactly one (EO) variable a_i and b_i can have assigned to true, we must encode this property into CNF as follows.

- $EO(a_{-1}, a_0, a_1)$: $(a_{-1} \vee a_0 \vee a_1) \wedge (\neg a_{-1} \vee \neg a_0) \wedge (\neg a_{-1} \vee \neg a_1) \wedge (\neg a_0 \vee \neg a_1)$
- $EO(b_{-1}, b_0, b_1)$: $(b_{-1} \vee b_0 \vee b_1) \wedge (\neg b_{-1} \vee \neg b_0) \wedge (\neg b_{-1} \vee \neg b_1) \wedge (\neg b_0 \vee \neg b_1)$

Now we need to encode the operations that involve these integer variables, in particular:

- `if (a < 0) b = -a`: $(\neg a_{-1} \vee b_1)$
- `if (a >= 0) b = a`: $(\neg a_0 \vee b_0) \wedge (\neg a_1 \vee b_1)$

Finally, in order to prove the assert statement, we must negate it and show that the resulting propositional logic formula is valid, i.e. that the formula is unsatisfiable. For simplicity, let's suppose that we want to prove that b is always larger than 0 at the end of the procedure. To ensure that this is the case, we just need to add the unit clause (b_{-1}) to the formula and show that the resulting formula is unsatisfiable. In the end, we would have formula φ that would encode the semantics of this program as:

$$\begin{aligned}\varphi =&(a_{-1} \vee a_0 \vee a_1) \wedge (\neg a_{-1} \vee \neg a_0) \wedge (\neg a_{-1} \vee \neg a_1) \wedge (\neg a_0 \vee \neg a_1) \wedge \\ &(b_{-1} \vee b_0 \vee b_1) \wedge (\neg b_{-1} \vee \neg b_0) \wedge (\neg b_{-1} \vee \neg b_1) \wedge (\neg b_0 \vee \neg b_1) \wedge \\ &(\neg a_{-1} \vee b_1) \wedge (\neg a_0 \vee b_0) \wedge (\neg a_1 \vee b_1) \wedge \\ &(b_{-1})\end{aligned}$$

Even though this is a very simple example, it gives the intuition on how to transform a program to propositional logic. Two main challenges when performing these transformations are loops and variables that are assigned more than once. Loops can be transformed into a straight-line program with the unrolling technique described previously. For variable assignment, we can introduce fresh variables to guarantee that each variable is assigned *exactly once*. This transformation is called *static single assignment* (SSA). We will not go over the details of SSA in this lecture but if you are interested in knowing more we refer you to the lecture notes of the "15-411 Compiler Design".

4 Bounded Model Checking in Practice

Several tools implement efficient BMC procedures and that can be used in practice. One of the most known BMC tools is CBMC, which performs bounded model checking for C code and it is available at https://www.cprover.org/cbmc/.

4.1 Getting started

We will show some examples of using CBMC to prove verification conditions or to find a counterexample when the program is buggy. These examples are available at https://www.cs.cmu.edu/~15414/lectures/20-bmc/cbmc-example.c.

```
1  void f00 (int8_t x, int8_t y, int8_t z) {
2    if (x < y) {
3    int8_t firstSum = x + z;
4    int8_t secondSum = y + z;
5    assert(firstSum < secondSum);
6  }
7 }
```

CBMC can be run on the program `f00` with the following command line:

```
$ cbmc --drop-unused-functions --function f00 cbmc-example.c
```

And will be able to detect a potential arithmetic overflow when summing x and z. You can ask CBMC for a trace that corresponds to the counterexample and that will assign values to x and z that will trigger the arithmetic overflow by running the following command line:

```
$ cbmc --drop-unused-functions --function f00 cbmc-example.c --trace
```

To fix this issue, one can change the type of `firstSum` and `secondSum` to have a larger bit width, e.g. changing it to a 16 bit integers (`int16_t`).

When handling `for` loops, CBMC is able to determine how much the loop needs to be unroll to fully replace the loop of the program equivalent straight-line code. However, for programs with `while` loops, CBMC is unable to determine the necessary unroll depth. Consider the following program:

```
1  void f03 (int x, int i) {
2    if (i >= 0 && i < 10) {
3      if (x > 0) {
4        while (i < 10) {
5          x += i;
6          ++i;
7        }
8      }
9    } else {
10     x = 42;
11   }
12   assert(x != 1);
13 }
```

The user must specify how many times the loop must be unrolled. This can be achieved with the option `--unwind n`. To check if the loop was completely unrolled, the user can additionally use the option `--unwinding-assertions`. For instance, the command:

```
$ cbmc --drop-unused-functions cbmc-example.c --unwind 10 --function f03
       --unwinding-assertions
```

This would show that the unwinding assertion will be triggered since the loop must be unrolled 11 times.

CBMC uses a SAT solver by default but it can also use Satisfiability Modulo Theory (SMT) solvers instead. In the next lecture, we will introduce SMT and talk about decision procedures to solve a combination of theories. However, we show an example here that illustrates some differences when handling multiplication.

```
1  void f14 (int16_t x, int16_t y) {
2    int16_t a = x;
3    int16_t b = y;
4    assert(a*b == x*y);
5  }
```

SAT encodings of multipliers are not very efficient. When considering this program, it takes more than 10 minutes to solve this verification problem on a common laptop when invoking CBMC with a SAT solver:

```
$ cbmc --drop-unused-functions cbmc-example.c --function f14
```

However, if CBMC is called using the SMT solver Z3 (with the option `-z3`) then it takes less than 1 second for any bit-width since Z3 uses equivalence reasoning and reduces the problem to $a \times b = a \times b$.

4.2 Useful command line options

CBMC has a lot of command line options, but here we list some common options.

- `--function name` set main function name to be analyzed.
- `--trace` give a counterexample trace for failed properties. This option is helpful to trace the bug which will give you insights on how to fix it.
- `--bounds-check` enable array bounds checks. This option implicitly creates assert statements to ensure correct array bounds.
- `--pointer-check` enable pointer checks. This option implicitly creates assert statements to ensure correct pointer access.
- `--signed-overflow-check` enable signed arithmetic over- and underflow checks. This option implicitly creates assert statements to ensure that there will not be signed overflows.
- `--drop-unused-functions` drop functions trivially unreachable from main function which makes it easier to read the output.
- `--unwind nr` unwind nr times. Each loop is unwind nr times.
- `--slice-formula` remove assignments unrelated to property which makes verification faster.
- `--unwinding-assertions` generate unwinding assertions.
- `--z3` use Z3 to solve the formula instead of a SAT solver.