15-414: Bug Catching: Automated Program Verification

Lecture Notes on Model Checking

Matt Fredrikson

Carnegie Mellon University
Lecture 17
Thursday, April 1, 2025

1 Introduction

In this lecture, we will show how we can use 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 such approach is called *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.

In this lecture, you will learn:

- How bounded model checking verifies an under-approximation of a program's semantics against a contract given by a Hoare triple, by leveraging the *strongest postcondition* introduced in Lecture 11.
- A key limitation of bounded model checking, i.e. the fact that it cannot prove the absence of all bugs, can be partially mitigated with *unwinding assertions*.
- How to formalize specifications of correctness that involve time and sequencing as Linear Temporal Logic (LTL).

2 Bounded Model Checking

Bounded Model Checking computes an *underapproximation* of a program's semantics 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. Recall from a much earlier lecture the following axiom, which allows us to replace a loop with a conditional statement, within which is a copy of the original loop.

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

The axiom 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 it 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 $\le$ 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 the axiom twice to the loop. When we stop, we replace the remaining loop with an empty statement.

```
1    i := N;
2    if(0 $\le$ x $<$ N) {
3      i := i - 1;
4      x := x + 1;
5      if(0 $\le$ 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 other terminal axioms 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.

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

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 $\le$ 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, 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 $\le$ x $<$ 3) {
3       i := i - 1;
4       x := x + 1;
5       if(0 $\le$ x $<$ 3) {
6         i := i - 1;
7         x := x + 1;
8         assert($\lnot$(0 $\le$ x $<$ 3));
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 $\le$ x $<$ 3) {
  i := i - 1;
  x := x + 1;
  if(0 $\le$ x $<$ 3) {
    i := i - 1;
    x := x + 1;
    if(0 $\le$ 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 Linear Temporal Logic

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 $\Box \mathit{terminated} \to P$, and we will focus on this case for the remainder of these lecture notes.

3.1 Trace Semantics (Extra, Optional)

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. $\tau(x := e) = \{(\omega, \nu) \ : \ \nu = \omega \text{ except that } \nu(x) = \omega[\![e]\!] \text{ for } \omega \in \mathcal{S}\}$
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. $\tau(\texttt{while}(Q)\,\alpha) = \{\sigma^{(0)} \circ \sigma^{(1)} \circ \cdots \circ \sigma^{(n)} \;:$ 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$\}$
   $\cup\, \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \ldots \;:$ 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 $\alpha^0 \overset{\text{def}}{\equiv} (?true)$.

3.2 LTL

Now that we have a set of traces such as the ones $\tau(\alpha)$ generated by a program α, we have more temporal information about the sequence of states that happened during the run of the program. That enables us to talk more about the way how truth-values change over time along such a trace.

Definition 2 (LTL)**.** The formulas of linear temporal logic (LTL) with atomic propositions p are defined by the following grammar:

$$P, Q ::= p \mid \neg P \mid P \wedge Q \mid \mathbf{X}P \mid \Box P \mid \Diamond P \mid \mathbf{U}PQ$$

The formula $\Box P$ means that P is always true in the future. The formula $\Diamond P$ means that P is sometimes true in the future, meaning at least at one point. The formula $\mathbf{X}P$ means that P is true in the next state. And the formula $\mathbf{U}PQ$ means that P is true until Q is true (which also will be true at some point).

The suffix of a trace σ starting at step $k \in \mathbb{N}$ is denoted σ^k and only defined if the trace has at least length k. That is

$$(\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{k-1}, \sigma_k, \sigma_{k+1}, \sigma_{k+2}, \ldots)^k = (\sigma_k, \sigma_{k+1}, \sigma_{k+2}, \ldots)$$

In particular σ^0 is the same as σ. Also $(\sigma_0) \circ \sigma^1 = \sigma$ if the trace has at least length 1 so that σ^1 is defined.

Definition 3. The truth of LTL formulas in a trace σ is defined inductively as follows:

1. $\sigma \models p$ iff $\sigma_0 \models p$ for atomic propositions p provided that $\sigma_0 \neq \Lambda$
2. $\sigma \models \neg P$ iff $\sigma \not\models P$, i.e. it is not the case that $\sigma \models P$
3. $\sigma \models P \wedge Q$ iff $\sigma \models P$ and $\sigma \models Q$
4. $\sigma \models \mathbf{X}P$ iff $\sigma^1 \models P$
5. $\sigma \models \Box P$ iff $\sigma^i \models P$ for all $i \geq 0$
6. $\sigma \models \Diamond P$ iff $\sigma^i \models P$ for some $i \geq 0$

7. $\sigma \models \mathbf{U}PQ$ iff there is an $i \geq 0$ such that $\sigma^i \models Q$ and $\sigma^j \models P$ for all $0 \leq j < i$

In all cases, the truth-value of a formula is, of course, only defined if the respective suffixes of the traces are defined.

For example, $\mathbf{X}P$ only has a truth-value in trace σ if σ^1 is defined, which means that the trace σ has length 1 (recall that this means it has at least 1+1 states). So $\mathbf{X}P$ is neither true or false but simply meaningless in a trace such as σ_0 that does not actually have a next state. Likewise, $\mathbf{X}p$ is meaningful (and either true or false depending on whether p is true in σ_1) in a trace $\sigma = (\sigma_0, \sigma_1)$, but $\mathbf{XX}p$ is not meaningful in the same trace because it's not long enough to have a successor of a successor.

Note that the meaning of the box and diamond modalities of LTL is quite analogous to the meaning that the box and diamond modalities already have in dynamic logic. The only difference is that dynamic logic modalities range over the runs of a concrete program while the modalities of LTL range over time (along a fixed trace of something). This is not a coincidence. Both are versions of modal logics, which differ in terms of what the box and diamond modalities range over but are otherwise built similarly.

For the cases $\mathbf{X}P, \Box P, \Diamond P, \mathbf{U}PQ$ It is, of course, very important to retain the entire suffix of the trace for the semantics (not just a single state) in case the subformulas P and Q themselves mention further temporal operators. For example, LTL formula

$$\Box\Diamond P$$

expresses that P is true infinitely often when referring to an infinite trace. On a finite trace, it merely means that P is true in the last (non-failure) state.

The LTL formula

$$\Diamond\Box P$$

expresses that P is eventually true all the time (so is true almost always, so except at finitely many exception states) when referring to an infinite trace. On a finite trace, it also merely means that P is true in the last (non-failure) state.

4 LTL Formulas on Program Traces

The following very clever program solves the issue of subtracting from negative numbers by first turning them into positive numbers and then adding, while ultimately flipping the sign again.

$$\begin{aligned} x &:= -x; \\ x &:= x + 7; \\ x &:= -x; \end{aligned}$$

This program does correctly subtract 7 from a negative number as witnesses by a corresponding proof of the following dynamic logic formula:

$$x = x_0 \rightarrow [x := -x; x := x + 7; x := -x]\, x = x_0 - 7$$

This formula means that whenever x_0 equals the initial value of variable x then *after* running the program, the resulting value of x will be the result of subtracting 7 from x_0, which, since it didn't change, still is the initial value of x. The program also satisfies the property that if x is initially negative then x is finally negative:

$$x < 0 \rightarrow [x := -x; x := x + 7; x := -x]\, x < 0$$

But it *does not* satisfy that x is negative always at all times while running the program, because the whole point is that the first assignment flips the sign of x into a positive number. In fact, all traces of this program are of the following form:

$$\tau(x := -x; x := x + 7; x := -x) = \{(\omega, \omega_x^{-\omega(x)}, \omega_x^{-\omega(x)+7}, \omega_x^{-(\omega(x)+7)}) \ : \ \omega \text{ is any state}\}$$

Consequently, if $\sigma \in \tau(x := -x; x := x + 7; x := -x)$ is a trace of this program starting in an initial state σ_0 with negative initial value of x so $\sigma_0(x) < 0$, then the LTL formula $\Box(x < 0)$ is *not* true for it even if it is true initially and in the end. Indeed, all traces $\sigma \in \tau(x := -x; x := x + 7; x := -x)$ of the program satisfy:

$$\sigma \not\models x < 0 \rightarrow \Box(x < 0)$$

That is, the following condition is false for σ:

> if $x < 0$ is true (initially, because there's no temporal operator on the left hand side of the implication), then $x < 0$ is true always in the future (of σ).

But what is, indeed, true for all traces σ of the program is:

$$\sigma \models x < 0 \rightarrow \Box(x \neq 0)$$

That is, if x starts negative then it will always be nonzero at every point in time throughout the entire trace σ.

5 Summary

- A trace is either a finite sequence of states of a given length or an infinite sequence of states;
- A trace terminates iff it is finite and its last state is not a failure state (Λ);
- The trace semantics of a program α ($\tau(\alpha)$) is the set of all its possible traces;
- Linear temporal logic is defined by the following grammar:

$$P, Q ::= p \mid \neg P \mid P \wedge Q \mid \mathbf{X}P \mid \Box P \mid \Diamond P \mid \mathbf{U}PQ$$

  - The formula $\Box P$ means that P is always true in the future;
  - The formula $\Diamond P$ means that P is sometimes true in the future, meaning at least at one point;
  - The formula $\mathbf{X}P$ means that P is true in the next state;
  - The formula $\mathbf{U}PQ$ means that P is true until Q is true (which also will be true at some point).