15-414: Bug Catching: Automated Program Verification

Lecture Notes on Bounded Model Checking

Matt Fredrikson

Carnegie Mellon University
Lecture 17
Thursday, March 24, 2022

1 Introduction

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.

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 tools like Z3 can be applied to implement bounded model checking over formulas of bit vector arithmetic, providing counterexamples in cases where the program cannot be verified.

2 Bounded Model Checking

Now that we have seen how the validity of a formula over machine integers can be reduced to propositional satisfiability, we will take the discussion a step further and show how verification problems are cast as validity over machine integers. The approach that we consider is called *bounded model checking*, and is one of the original, and still among the most prevalent, applications of bit-blasting.

Bounded model checking essentially computes an underapproximation of the set of states that a given program may be able to reach, and checks these states against a given "goal" formula that describes the specification. In this lecture, we will focus on specifications given as Hoare triples, as discussed in Lecture 11. Given a program α, check that the postcondition Q is true in any reachable final state assuming that the precondition P is true in any initial state:

$$P\,\{\alpha\}\,Q$$

In practice, bounded model checking is often employed to check other types of properties, such as invariants which require that *any* reachable state models a given formula.

2.1 Computing reachable states

Recall from Lecture 11 the strongest postcondition:

1. $P\{\alpha\}(\mathsf{sp}(\alpha)P)$ (it is a postcondition, or: it is a necessary consequence of P)
2. If $P\{\alpha\}R$ then $\mathsf{sp}(\alpha)P \to R$ (it is a *strongest* postcondition for P, or: is it sufficient for all consequences of P)

In other words, given a precondition P, the strongest postcondition for P is a formula that describes *all* of the possible consequences of executing α starting in P. For any other possible formula R describing the set of final states from executing α starting in P, we know that $\mathsf{sp}(\alpha)P \to R$.

If our pre and postconditions are given in the theory of bit vector arithmetic, then the strongest postcondition provides a means of checking a Hoare triple $P\,\{\alpha\}\,Q$ using a SAT solver.

1. Compute the strongest postcondition of the precondition P.
2. Check the satisfiability of $\mathsf{sp}(\alpha)P \wedge \neg Q$.
   a) If the result is `sat`, then the Hoare triple is not valid.
   b) If the result is `unsat`, then the Hoare triple is valid.

To understand why this procedure is correct, first consider the case where the solver returns `sat`. A satisfying assignment to $\mathsf{sp}(\alpha)P \wedge \neg Q$ will consist of a set of values for variables in the final state of α, that does *not* satisfy the goal postcondition Q; in other words, this case yields a *counterexample* to the Hoare triple $P\,\{\alpha\}\,Q$. On the other hand,

if the solver returns `unsat`, then it means that for all possible final states described by $\mathsf{sp}(\alpha)P$, none of them is consistent with $\neg Q$. In other words, *all* of the final states of α starting in P satisfy Q, so the Hoare triple is valid.

So far, this is promising given that we know how to compute strongest postconditions, and we know how to decide the satisfiability of bit vector arithmetic. As you might have guessed, the problem with this approach is the iteration command, for which the strongest postcondition has a recursive definition:

$$\mathsf{sp}(\alpha^*)P = P \vee \mathsf{sp}(\alpha^*)(\mathsf{sp}(\alpha)P)$$

The way that bounded model checking resolves the issue is to simply impose an upper-bound on the number of times the iterative strongest postcondition is applied recursively. This is tantamount to syntactically "unwinding" each of the loops in the program a finite number of times, and checking the resulting loop-free program.

Consider the following program.

$$(x := x + 1)*$$

Suppose that we wish to apply bounded model checking with an iteration bound of $k = 2$, i.e., we will only recursively apply the strongest postcondition twice.

$$\begin{array}{ll}
\mathsf{sp}((x := x + 1)*)P(x) & = \\
P(x) \vee \mathsf{sp}((x := x + 1)*)(\mathsf{sp}(x := x + 1)P(x)) & = \\
P(x) \vee \mathsf{sp}((x := x + 1)*)(\exists x_1 . x = x_1 + 1 \wedge P(x_1)) & = \\
P(x) \vee (\exists x_1 . x = x_1 + 1 \wedge P(x_1) \vee \mathsf{sp}((x := x + 1)*)(\mathsf{sp}(x := x + 1)(\ldots))) &
\end{array}$$

At this point we have applied the strongest postcondition twice: once on the original program, and again on the body of the iteration. The result still has a disjunctive term corresponding to the strongest postcondition:

$$\mathsf{sp}((x := x + 1)*)(\mathsf{sp}(x := x + 1)(\exists x_1 . x = x_1 + 1 \wedge P(x_1)))$$

We do not wish to apply the strongest postcondition again, as we have reached our bound of $k = 2$. We have two choices of how to remove this term from the formula.

Ignorance is bliss The first option is to simply ignore it. The identity for disjunction is **false**, so we simply replace the remaining appearance of strongest postcondition with $\bot$ in the formula, and proceed with satisfiability checking. The resulting formula in this case would be:

$$P(x) \vee (\exists x_1 . x = x_1 + 1 \wedge P(x_1))$$

But notice that this formula only accounts for two of the infinite set of possible ways to execute the original program $(x := x + 1)*$: executing the body either zero or one time. This means that any time we have a postcondition Q that could only be violated after executing the body more than once, we will remain ignorant of the corresponding

bugs. For example, if $P(x) \equiv x = 0$ and $Q(x) \equiv x > 1$, then the formula that bounded model checking will ultimately check satisfiability of:

$$(x = 0 \vee (\exists x_1 . x = x_1 + 1 \wedge x_1 = 0)) \wedge \neg(x > 1) \text{ is unsatisfiable}$$

Thus, by taking this approach we remain blissfully ignorant of any bugs that manifest on deeper executions of the program.

Conservative, but safe The fact that bounded model checking unwinds iteration only a pre-determined finite number of times means that it will always fall short of being able to uncover bugs that arise deep in execution. But in some cases, it is possible to verify that the procedure has unwinded iteration enough to find any bugs that *could* arise. Thus, if bounded model checking returns `unsat`, we can conclude that the Hoare triple is valid.

This approach relies on *unwinding assertions*, which are named as such because they can be understood in terms of adding assertions at certain points in the unwinding procedure. Consider the following program, which will only ever execute its loop body once.

```
1    x := 0;
2    while(x < 1)
3      x := x + 1;
```

Nonetheless, if we were to attempt to verify the Hoare triple $\{true\}\,\alpha\,\{x < 2\}$ using bounded model checking for $k = 1$, we would imagine unwinding the loop once, and applying bounded model checking to the result:

```
1    x := 0;
2    if(x < 1)
3      x := x + 1;
```

This would lead to the following satisfiability check, which has been simplified for clarity; the guard from the conditional does not appear, nor does the "else" branch, because the initial assignment to x makes both tautological.

$$(\exists x_1 . x = x_1 + 1 \wedge x_1 = 0) \wedge \neg(x < 2)$$

This is unsatisfiable, as expected, because the program clearly satisfies the postcondition: there is no execution that ends in a final state where $x \geq 2$. But bounded model checking does not actually let us draw such a conclusion, because it only promises to find violations up to the depth that we allowed it to unwind the loop, so the fact that this is unsatisfiable does not let us conclude that a deeper unwinding will also be bug-free.

To take advantage of unwinding assertions, suppose that we added the following assertion once the unwinding had reached its depth bound:

```
1    x := 0;
2    if(x < 1) {
3      x := x + 1;
4      assert(x ≥ 1);
5    }
```

If this assertion were to fail when the program were executed, then it would tell us that deeper executions are possible in the program that we unwound, and to rule out bugs with bounded model checking, we should increase the depth bound.

We can incorporate this reasoning into bounded model checking by "simulating" the assertion when we compute the strongest postcondition. We add a special variable to signal that an assertion has been violated:

```
1    x := 0;
2    if(x < 1) {
3      x := x + 1;
4      if(x < 1)
5        error := 1
6    }
```

Then we use bounded model checking to verify the triple $\{\texttt{error} = 0\}\,\alpha\,\{\texttt{error} = 1\}$. Note that we do not literally need to rewrite the program by adding a conditional statement to it. Whereas with the "ignorance is bliss" approach we equated the remaining strongest postcondition term in a recursive unrolling with **false**, here we instead replace with the strongest postcondition of the simulated assertion. In the example, the formula that we return would be:

$$\exists x_1.x = x_1 + 1 \wedge (x_1 = 0 \wedge \exists \texttt{err}_1.\texttt{err} = 1 \wedge \texttt{err}_1 = 0)$$

We would then check whether the following formula is satisfiable:

$$\exists x_1.x = x_1 + 1 \wedge (x_1 = 0 \wedge \exists \texttt{err}_1.\texttt{err} = 1 \wedge \texttt{err}_1 = 0) \wedge \texttt{err} = 1$$

Using unwinding assertions is conservative in the sense that if this formula is satisfiable, then it does not necessarily mean that there is an execution of the program that violates the postcondition. It simply means that the depth of the bounded check is not sufficient to rule out potential violations, and that a larger unwinding bound may uncover additional bugs.

3 Implementation with Z3

At this point, we have learned how formulas over bit vectors can encode constraints involving machine integers, as well as an approach for finding bugs, and in some cases proving their absence, using bounded model checking. Now we will put these pieces together, and see how to perform bounded model checking with a tool called Z3 [DMB08].

You are likely already somewhat familiar with Z3, as it is one of the three decision procedures what Why3 uses to discharge verification conditions. Here we will see how to use Z3's Python API to implement a strongest postcondition generator, check the satisfiability of formulas needed to implement bounded model checking, and examine its results to gain insight into bugs that the procedure discovers. You may already have Z3 on your system because it is used by Why3. If you use a virtual machine or Docker container to run Why3, and would like to run Z3 natively, pre-built binaries for

all major platforms can be obtained at https://github.com/Z3Prover/z3/releases. The Python API can be installed using `pip`:

```
pip install z3-solver
```

Further instructions are available at https://github.com/Z3Prover/z3.

To make use of Z3's Python API, we will import its entire namespace.

```
from z3 import *
```

The full documentation for this namespace is available at:

https://z3prover.github.io/api/html/namespacez3py.html.

3.1 Defining programs and formulas

Before we can get begin to implement the strongest postcondition generator, we need a way to represent programs and formulas. Note that in these notes and in the live-coding demos from lecture, we make use of syntax that is new to Python 3.10, but the code linked from the course webpage uses syntax that is compatible with earlier versions of Python 3.

We will assume that terms (expressions) appearing in programs and formulas are either integer constants, variables, and sums or differences of terms.

```
@dataclass
class Const:
    value: int

@dataclass
class Var:
    name: str

@dataclass
class Sum:
    left: Term
    right: Term

@dataclass
class Difference:
    left: Term
    right: Term

Term = Const | Var | Sum | Difference
```

Next we define the syntax of formulas. We suffix the constructors with a capital `F`. This avoids naming collisions with objects imported from the Z3 API, i.e., Z3 already exports `Not`, `Or`, `Implies`, etc., and we do not want to override those names.

```
@dataclass
class TrueC:
    _: None

@dataclass
```

```
class FalseC:
    _: None

@dataclass
class LtF:
    left: Term
    right: Term

@dataclass
class EqF:
    left: Term
    right: Term

@dataclass
class NotF:
    q: Formula

@dataclass
class AndF:
    p: Formula
    q: Formula

@dataclass
class OrF:
    p: Formula
    q: Formula

@dataclass
class ImpliesF:
    p: Formula
    q: Formula

Formula = TrueC | FalseC | LtF | EqF | NotF | AndF | OrF | ImpliesF
```

Finally, we define the syntax of programs. Following previous lectures, and in particular Lecture 11, we will define programs to be assignment, sequential composition, test, nondeterministic choice, and nondeterministic repitition (iteration).

```
@dataclass
class Asgn:
    left: Var
    right: Term

@dataclass
class Seq:
    alpha: Prog
    beta: Prog

@dataclass
class Test:
    q: Formula

@dataclass
class Choice:
```

```
    alpha: Prog
    beta: Prog

@dataclass
class Iter:
    alpha: Prog

Prog = Asgn | Seq | Test | Choice | Iter
```

Now we have all the constructors that we need to encode simple programs and formulas. For example, we can encode a program that increments `x` as follows:

```
Asgn(Var('x'), Sum(Var('x'), Const(1)))
```

Slightly more involved, a loop that increments `x` until it reaches 5:

```
Seq(
    Iter(
        Seq(
            Test(LtF(x, Const(5))),
            Asgn(x, Sum(x, Const(1)))
        )
    ),
    Test(NotF(LtF(x, Const(5))))
)
```

We did not provide constructors for some terms and formulas, e.g. multiplication, bitwise operations, or quantifiers. These are straightforward to add, and are left as an exercise.

3.2 Encoding terms and formulas

Moving on, we now consider how to encode instances of `Term` and `Formula` using the Z3 API. We wish to make use of Z3's theory of bit vectors, which are represented by the type `BitVecRef`. The two primary functions of interest to us will be `BitVecVal` and `BitVec`:

```
BitVecVal(value: int, width: int) -> BitVecRef
BitVec(name: str, width: int) -> BitVecRef
```

`BitVecVal` takes a Python integer value, and a bit vector width, and returns a `BitVecRef` that represents the value. `BitVec` takes a string and bit vector width, and returns a `BitVecRef` that represents a bit vector variable of the given width. Z3 treats the type of `BitVecRef` instances as being indexed on their width; in general, attempting to perform operations on `BitVecRef`'s of different widths will result in a runtime exception.

Now, we define a function `term_enc` that maps `Term` objects to their corresponding representation as `BitVecRef`. We define the global constant `bit_width` to ensure that all `BitVecRef` objects have a consistent width.

```
bit_width = 32

def term_enc(e: Term) -> BitVecRef:
    match e:
```

```
        case Const(value):
            return BitVecVal(value, bit_width)
        case Var(name):
            return BitVec(name, bit_width)
        case Sum(left, right):
            return term_enc(left) + term_enc(right)
        case Difference(left, right):
            return term_enc(left) - term_enc(right)
```

Observe that `BitVecRef` objects overload the infix addition and subtraction operator. Likewise, multiplication, division, bitwise and/or/xor/complement/shift, unary negation ("minus"), and relational predicates (==, <, <=, >, >=) are overloaded for these objects as well.

Moving on, we define a similar encoding function for `Formula` objects. The relevant type for these is `BoolRef`. Similar to how `BitVecRef` objects are created, the Z3 API exposes `BoolVal` for constructing Boolean constant values, and `Bool` for Boolean-valued variables.

```
def fmla_enc(p: Formula) -> BoolRef:
    match p:
        case TrueC(_):
            return BoolVal(True)
        case FalseC(_):
            return BoolVal(False)
        case LtF(left, right):
            return term_enc(left) < term_enc(right)
        case EqF(left, right):
            return term_enc(left) == term_enc(right)
        case NotF(p):
            return Not(fmla_enc(p))
        case AndF(p, q):
            return And(fmla_enc(p), fmla_enc(q))
        case OrF(p, q):
            return Or(fmla_enc(p), fmla_enc(q))
        case ImpliesF(p, q):
            return Implies(fmla_enc(p), fmla_enc(q))
```

Note that in `fmla_enc`, we used the Z3 API to construct negation, conjunction, and disjunction. While the API overloads Python syntax for many operations on `BitVecRef` objects, this is not the case for `BoolRef`. So, for example, `And(P, Q)` returns a Z3 `BoolRef` object that represents the conjunction of `P` and `Q`, whereas `P and Q` raises an exception:

```
---------------------------------------------------------
Z3Exception             Traceback (most recent call last)
Input In [27], in <cell line: 2>()
      1 P, Q = Bools('P Q')
----> 2 P and Q

File z3.py:375, in AstRef.__bool__(self)
    373  return self.arg(0).eq(self.arg(1))
    374 else:
--> 375  raise Z3Exception(...)
```

```
11
12 Z3Exception: Symbolic expressions cannot be cast to
13               concrete Boolean values.
```

3.3 Implementing strongest postcondition

We are now in a good position to implement the strongest postcondition generator. The basic signature that we would like to implement matches our formal definition of strongest postcondition, taking a program and precondition, a maximum iteration bound, and returning a new formula:

```
1 spost(alpha: Prog, P: BoolRef, max_depth: int) -> BoolRef
```

We might have considered implementing a slightly different signature:

```
1 spost(alpha: Prog, P: Formula) -> Formula
```

Pragmatically, this makes little difference, as we can always apply `fmla_enc` to `Formula` objects to obtain a `BoolRef`. However, defining our generator to both accept and return `BoolRef` objects directly will make the code more concise.

For most of the statement forms, the implementation of strongest postcondition is straightforward, relying primarily on recursive calls to `spost`. Recall that the strongest postcondition of sequential composition $\alpha; \beta$ first computes the strongest postcondition of α, and uses the result as the precondition to compute the strongest postcondition of β:

$$\mathsf{sp}(\alpha; \beta)\, P = \mathsf{sp}(\beta)(\mathsf{sp}(\alpha)\, P)$$

This translates easily to the following implementation.

```
1 case Seq(alpha, beta):
2   return spost(beta, spost(alpha, P), max_depth)
```

Similarly, the definitions for nondeterministic choice and repetition depend on making the appropriate composition of results obtained via recursive calls:

$$\begin{array}{lcl} \mathsf{sp}(\alpha \cup \beta)\, P & = & \mathsf{sp}(\alpha)\, P \vee \mathsf{sp}(\beta)\, P \\ \mathsf{sp}(\alpha^*)\, P & = & P \vee \mathsf{sp}(\alpha; \alpha^*)\, P \end{array}$$

This is again an easy translation to code, but we first decrement the maximum iteration depth before recursing to ensure eventual termination:

```
1 case Choice(alpha, beta):
2     return Or(spost(alpha, P, max_depth), spost(beta, P, max_depth))
3
4 case Iter(alpha):
5     return Or(P, spost(Seq(alpha, Iter(alpha)), P, max_depth-1))
```

The test command does not require a recursive call to `spost`, as the strongest postcondition in this case is the conjunction of the guard formula and the given precondition:

$$\mathsf{sp}(?Q)\, P = P \wedge Q$$

Translated to code, the only thing to be careful about is ensuring that the conjunction is over two `BoolRef` objects:

```
case Test(Q):
    return And(fmla_enc(Q), P)
```

The only remaining statement is assignment, which involves variable substitution.

$$\mathsf{sp}(x := e(x))\, P(x) = \exists x'.x = e(x') \wedge P(x')$$

Fortunately, Z3 provides a function for performing substitution on formulas. Assuming a helper function `fresh_var` that returns a fresh new variable name, we implement this as follows:

```
case Asgn(left, right):
    next_var = fresh_var(left)
    right_sub = substitute(term_enc(right),
                           [(term_enc(left), term_enc(next_var))])
    P_sub = substitute(P, [(term_enc(left), term_enc(next_var))])

    return Exists(term_enc(next_var),
                  And(term_enc(left) == right_sub, P_sub))
```

Note that because we intend to use `spost` *positively* in a satisfiability check, we do not need the existential quantifier in the case for assignment. We can instead implement the strongest postcondition for assignment as:

```
case Asgn(left, right):
    next_var = fresh_var(left)
    right_sub = substitute(term_enc(right),
                           [(term_enc(left), term_enc(next_var))])
    P_sub = substitute(P, [(term_enc(left), term_enc(next_var))])

    return And(term_enc(left) == right_sub, P_sub)
```

The primary advantage of this implementation is that in cases where the formula $\mathsf{sp}(\alpha)\, P \wedge \neg Q$ is satisfiable, i.e., the Hoare triple is not valid, Z3 will return a satisfying assignment over all of the intermediate assignments leading up to a counterexample. With the implementation that uses existential quantification, the variables representing intermediate values are not free in the formula checked by Z3, so they are not included in the models that it produces.

3.4 Putting it all together

Now that we have our strongest postcondition generator, we can apply it to bounded model checking. We'll start with a simple test case to make sure everything works as expected, verifying the triple:

$$\{x = 0\}\, x := x + 1\, \{x = 1\}$$

Recall that to check this triple, we derive $\mathsf{sp}(x := x + 1)(x = 0)$, conjoin the result with the negated postcondition, $x \neq 1$, and check satisfiability.

```
alpha = Asgn(Var('x'), Sum(x, Const(1)))
P = EqF(Var('x'), Const(0))
Q = EqF(Var('x'), Const(1))

s = z3.Solver()
s.add(post(alpha, fmla_enc(P)))
s.add(fmla_enc(NotF(Q)))
s.check()
```

As expected, the result is `unsat`.

Now let's try a slightly more involved example on an invalid triple, and see how the approach finds bugs:

$$\{x = 0\}\,(?(x < 10); x := x + 1)^*; ?(x \geq 5)\,\{x = 5\}$$

The test in the iteration is not sufficient to ensure the postcondition, and there should be four counterexamples that demonstrate the problem. A counterexample is an assignment to the variables in the program that violates the triple. When the solver returns `sat` (recall, this means that the triple is not valid), we can extract a counterexample from the *model*, or satisfying assignment, that it found. enumerate all of the counterexamples by iteratively adding *blocking clauses* to the solver's set of asserted constraints.

```
s = z3.Solver()
s.add(post(alpha, fmla_enc(P)))
s.add(fmla_enc(NotF(Q)))

while s.check() == sat:
    x_val = s.model().evaluate(term_enc(Var('x')),
                               model_completion=True)
    blocking_clause = Not(term_enc(Var('x')) == x_val)
    s.add(blocking_clause)
    print('counterexample: x = {}'.format(x_val))
```

As shown in this example, blocking clause is a formula that prevents the solver from returning a previously-encountered satisfying assignment. In this context, we just assert that `x` is not equal to the current counterexample. Note that we passed the keyword argument `model_completion` to `s.model().evaluate`. This is to ensure that Z3 produces a value for the variable we are interested in, as in general there is no guarantee that the solver will need to assign every variable to return `sat`.

Running this code produces the four possible counterexamples:

```
counterexample: x = 6
counterexample: x = 8
counterexample: x = 9
counterexample: x = 7
```

When extracting counterexamples, we need not limit ourselves to the final value of `x`. Because we did not use the existential quantifier in the strongest postcondition for assignment, we can see the corresponding intermediate values for x by invoking `s.model()`.

```
[x_1 = 0,
 x_2 = 1,
 x_3 = 2,
 x_4 = 3,
 x_5 = 4,
 x_6 = 5,
 x = 6,
 x_7 = 2147483642,
 x_8 = 2147484766,
 x_9 = 2147482530]
```

Note that the values for `x_7`-`x_9` are arbitrary, as the solver effective chose an execution of the program that only iterates through the loop body six times.

References

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, Berlin, Heidelberg, 2008. Springer-Verlag.