15-414: Bug Catching: Automated Program Verification

Lecture Notes on Bounded Model Checking

Ruben Martins

Carnegie Mellon University
Lecture 15
Tuesday, March 30, 2021

1 Introduction

In the previous lectures, we have introduced decision procedures for propositional logic. 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. [1]

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

[1] This lecture was partially written by Matt Fredrikson and André Platzer and adapted by Ruben Martins for the current semester. Last updated on April 8, 2021.

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. $\tau(x := e) = \{(\omega, \nu) \ : \ \nu = \omega \text{ except that } \nu(x) = \omega[\![e]\!] \text{ for } \omega \in \mathcal{S}\}$
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. $\tau(\texttt{while}(Q)\,\alpha) = \{\sigma^{(0)} \circ \sigma^{(1)} \circ \dots \circ \sigma^{(n)} \ : \ \text{for some } n \geq 0 \text{ such that for all } 0 \leq i < n\text{:}$
   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$\}$
   $\cup \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \dots \ : \ \text{for all } i \in \mathbb{N}\text{: ① } \sigma_0^{(i)} \models Q \text{ and ② } \sigma^{(i)} \in [\![\alpha]\!]\}$
   $\cup \{(\omega) \ : \ \omega \not\models Q\}$
   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 $\alpha^0 \overset{\text{def}}{\equiv} (?\mathit{true})$.

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. Recall from a much earlier lecture the [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 $\le$ x $<$ N) {
3       i := i - 1;
4       x := x + 1;
5     }
```

Suppose that we want to check that $\Box terminated \rightarrow 0 \leq i$ 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 $\le$ 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 [if], [;], and [:=] 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.

```
i := 3;
while(0 $\le$ x $<$ 3) {
  i := i - 1;
  x := x + 1;
}
```

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 $k = 2$.

```
i := 3;
if(0 $\le$ x $<$ 3) {
  i := i - 1;
  x := x + 1;
  if(0 $\le$ x $<$ 3) {
    i := i - 1;
    x := x + 1;
    assert($\lnot$(0 $\le$ x $<$ 3));
  }
}
```

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 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 $\{-1, 0, 1\}$.

```
1    int a;
2    int b;
3    if (a < 0) b = -a;
4    else b = a;
5    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/15-bmc/cbmc-examples.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:

```
void f03 (int x, int i) {
  if (i >= 0 && i < 10) {
    if (x > 0) {
      while (i < 10) {
        x += i;
        ++i;
      }
    }
  } else {
    x = 42;
  }
  assert(x != 1);
}
```

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.

```
void f14 (int16_t x, int16_t y) {
  int16_t a = x;
  int16_t b = y;
  assert(a*b == x*y);
}
```

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 Finding bugs and proving verification conditions

CBMC can be used to find bugs in your code or prove that certain verification conditions hold on all paths without reaching the unwinding assertions. Consider that you have the following sorting algorithm in the file sort-bug.c.

```
1  void sort (int * a, int size) {
2    int i, j;
3    int element;
4    for (i = 2; i <= size; i++) {
5      element = a[i];
6      j = i - 1;
7      while (j >= 0 && a[j] > element) {
8        a[j+1] = a[j];
9        j = j - 1;
10     }
11     a[j+1] = element;
12   }
13   a[0] = 0;
14 }
```

This implementation has some bugs and CBMC can help us to find them.
If you try to run CBMC directly over this function with the command:

```
$ cbmc --function sort sort-bug.c
```

Then CBMC will just unwind forever without terminating:

```
Unwinding loop sort.0 iteration 1 file sort-bug.c line 7 function sort thread 0
Unwinding loop sort.0 iteration 2 file sort-bug.c line 7 function sort thread 0
Unwinding loop sort.1 iteration 1 file sort-bug.c line 4 function sort thread 0
...
```

To make the analysis of CBMC bounded you must specify an unwind depth. For instance, you can limit the unwind depth to 4 and run the command:

```
$ cbmc --function sort sort-bug.c --unwind 4
```

CBMC will return `VERIFICATION SUCCESSFUL`.

```
Generated 0 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
```

This means that CBMC did not generate any verification conditions, since there is no property to be checked. CBMC includes some properties that be checked with additional command line options such as `--pointer-check`. This option implicitly introduces assert statements in the code to guarantee that all pointers are valid. If we run CBMC again with this option:

```
$ cbmc --function sort sort-bug.c --unwind 4 --pointer-check
```

CBMC will find several failures:

```
[sort.pointer_dereference.1] line 5 dereference failure:
   pointer NULL in a[(signed long int)i]: FAILURE
...
** 32 of 38 failed (4 iterations)
VERIFICATION FAILED
```

Since we just call CBMC directly over the sort function, CBMC does not know anything about the possible values of `*a`. Therefore, it could be the case that this pointer is NULL which would lead to many pointers dereference failures. If you know that `*a` will never be NULL then you can use the keyword `__CPROVER_assume` to make this assumption by putting the following line at the beginning of the function:

```
__CPROVER_assume(a != NULL);
```

This would fix any pointer dereference failure that is related to `a` being `NULL`. However, CBMC would still report problems with pointer outside of object bounds.

```
[sort.pointer_dereference.6] line 12 dereference failure:
  pointer outside object bounds in a[(signed long int)i]: FAILURE
  ...
  ** 3 of 38 failed (2 iterations)
VERIFICATION FAILED
```

Since we do not know the size of the array, it could be possible to get a pointer outside of object bounds if the variable `size` is too large. We can have more control over the assumptions we can make if we build a harness test for this function. This will also allow checking the functionality of the function before and after calling this function.

Consider the function `harness_sort` that we will use to verify the functionality of the sort function.

```
#define MAX_ARRAY_BOUND 3
int nondet_int(void);

int *create_array(int n) {
  if(n <= 0 || n > MAX_ARRAY_BOUND)
    return NULL;

  int *a = malloc(n*sizeof(int));

  return a;
}

void harness_sort() {
  int n = nondet_int();
  int *array = create_array(n);
  __CPROVER_assume(array != NULL);
  for (int i = 0; i < n; ++i){
    __CPROVER_assume(array[i] >= 0 && array[i] <= 16);
  }

  sort (array, n);
}
```

First, in order to perform a bounded analysis, we need to bound the maximum size of the array. We can do this by defining a variable `MAX_ARRAY_BOUND` and set it to a given bound. Next, we need to create an array before using it in the sort function.

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. For instance, the function `nondet_int()` will return a non-deterministic integer.

Using non-deterministic variables, we can create an array of arbitrary size. However, since we want to bound the size of the array, in the `create_array` function, we can impose restrictions to the possible values of n. Since after the array is created we assume that it is not `NULL` then it is guaranteed that the array will have a size between 1 and `MAX_ARRAY_BOUND`. We use assume statements to populate the array with non-deterministic values between 0 and 16 (we impose a small domain to speedup verification but you can verify this for larger integer values).

If we run CBMC on the `harness_sort` function with the command:

```
$ cbmc --function harness_sort sort-bug.c --unwind 4 --pointer-check
  --drop-unused-functions --unwinding-assertions --trace
```

Note that we used a few extra options to check if we are unwinding completely all loops (note that the unwind depth depends on `MAX_ARRAY_BOUND` in this case) and to obtain a trace if a bug was found. CBMC will return a trace where we can observe that `i` would be equal to `size` and this would result in a pointer outside dynamic object bounds error. We could fix this by changing line 4:

```
for(i=2;i <=size;i++) -> for(i=2;i<size;i++)
```

After doing this, we can rerun CBMC and we will get `VERIFICATION SUCCESSFUL`. This means that all the pointer checks are valid but does this code have the correct functionality? Let's write a verification contract that can be automatically checked by CBMC. We can do this after we call the function `sort` by checking if the array is ordered:

```
1 // check if the array is sorted
2   for (int i = 0; i < n-1; i++){
3     assert (array[i] <= array[i+1]);
4   }
```

If we rerun CBMC then we will see that this assert fails. Analyzing the trace, we can see where the algorithm fails. Line 4 still has an issue since it starts with $i = 2$. This can be fixed for instance by setting $i = 0$:

```
for(i=2;i<size;i++) -> for(i=0;i<size;i++)
```

After fixing this bug, if we rerun CBMC then we get `VERIFICATION SUCCESSFUL`. This means that we prove that the array will always be ordered. However, our verification contract may still be too weak. For instance, did we guarantee that all elements that appear before still appear afterward? To fix this issue, you will need to write additional assert statements and may need to write some helper code. We start by defining a `copy_array` function that will copy the array before we sort it.

```
1  int* copy_array(int *a, int n)  {
2    int *a_copy = malloc(n*sizeof(int));
3    for (int i = 0; i < n; i++)
4      a_copy[i] = a[i];
5
6    return a_copy;
7  }
8
9  ...
10 void harness_sort() {
11
12   ...
13   int *array_orig = copy_array(array, n);
14
15   sort (array, n);
16
17   // check if each element that appears before still appears
18   for (i = 0; i < n; i++){
19     int element = array_orig[i];
20     int ok = 0;
21     for (int j = 0; j < n; j++){
22       if (array[j] == element)
23         ok = 1;
24     }
25     assert (ok == 1);
26   }
```

We can now compare the old array with the new array and see if all elements in the old array appear in the new array. If we run CBMC after writing these assert statements we are able to find a bug on line 13 of the original code whereby mistake the programmer set `a[0] = 0`. Since the array is being initialized with values between 0 and 16, setting the first element to 0 will still make the array ordered. However, this is not the intended behavior and to fix the bug we simply remove that line from the code. After this step, CBMC will say that the current contract is satisfied. Note that this contract is still incomplete since we do not check the elements appear the same number of times before and after sorting. We leave this as an exercise to the interested student.

4.3 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.

5 Summary

- **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.