Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 15 — March 2, 2026

Patrick Lam *version 1*

About CBMC and Kani

Continuing from the previous lecture's discussion on symbolic execution, we are going to talk about bounded model checking, as implemented in the CBMC and Kani tools. CBMC is for C, and Kani adapts it to Rust.

Bounded model checking (BMC) is another way of statically verifying code properties. BMC should be more lightweight than Dafny, which we'll talk about next—in particular, you don't need to write loop invariants when using a bounded model checker, but on the other hand, you might not get a guarantee from a successful BMC run.

My goal in talking about bounded model checking is for you to be able to apply modern bounded model checking tools like CBMC and Kani, understanding (at a high level) how they work and when they are likely to fail, and understanding what they are telling you when they succeed.

Someone on the Internet writes:

> Bounded model checking is unit testing with superpowers.
> — Karl Schultheisz, `https://kdsch.org/post/cbmc-technique/`

CBMC

So, let's get into it. Here is a C program, which we'll run CBMC with.

```
1 int main()
2 {
3   int a = 5;
4   __CPROVER_assert(a == 2, "a is not 2");
5   return 0;
6 }
```

We can verify it with CBMC:

```
$ cbmc explicit-assert.c
CBMC version 6.6.0 (cbmc-6.6.0) 64-bit x86_64 linux
Type-checking explicit-assert
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Starting Bounded Model Checking
Passing problem to propositional reduction
converting SSA
```

```
Running propositional reduction
SAT checker: instance is SATISFIABLE

** Results:
explicit-assert.c function main
[main.assertion.1] line 4 a is not 2: FAILURE

** 1 of 1 failed (2 iterations)
VERIFICATION FAILED
```

Indeed, 2 is not 5, and CBMC reports an assertion failure. If you initialize `a` to 2 instead of 5, then the verification succeeds.

We can also run the example from the previous lecture, now converted to C:

```
1 // cbmc div-by-zero.c --function foo
2 unsigned int foo(unsigned int x, unsigned int y) {
3   if (x > y)
4     return x / y;
5   else
6     return y / x;
7 }
```

and we do get failures for possible division by zero:

```
** Results:
div-by-zero.c function foo
[foo.division-by-zero.1] line 4 division by zero in x / y: FAILURE
[foo.division-by-zero.2] line 6 division by zero in y / x: FAILURE
```

Running CBMC with the `--show-properties` flag shows the properties it is verifying:

```
Property foo.division-by-zero.1:
  file div-by-zero.c line 4 function foo
  division by zero in x / y
  !(y == 0u)

Property foo.division-by-zero.2:
  file div-by-zero.c line 6 function foo
  division by zero in y / x
  !(x == 0u)
```

and with `--show-vcc` shows the relevant required verification conditions, as we discussed last time. Here's the report from the `y / x` division.

```
{-10} goto_symex::\guard#1 <=> ¬(foo::y!0@1#1 >= foo::x!0@1#1)
{-11} goto_symex::return_value::foo!0#1 = foo::x!0@1#1 / foo::y!0@1#1
|--------------------------------
{1} ¬goto_symex::\guard#1 => ¬(foo::x!0@1#1 = 0)
```

We can see that CBMC is going to automatically put in an assertion checking for division by zero, and it knows the conditions under which the y/x branch is executed, recording the conditions in variable guard1, which is true when $\neg(y \geq x)$. Verification succeeds on this branch if one can establish that input `x` is nonzero, given that guard1 is false (necessary to reach this branch).

CBMC checks. It's not just division by zero. From `cbmc --help`, CBMC includes checks for: bounds, pointers, memory leaks, memory cleanups, division by zero (ints and floats), (signed and unsigned) overflow and underflow, pointer arithmetic overflow and underflow, non-meaningful type conversions and shifts, floating-point overflow and NaN, enum ranges, and pointer validity.

Consider this program[1]:

```
1  #define SIZE 20
2  char buffer[SIZE];
3
4  char read_buffer(int i)  { return buffer[i];     }
5  char read_pointer(int i) { return *(buffer + i); }
6
7  int main() {
8    int index; // uninitialized
9    read_buffer(index);
10   read_pointer(index);
11 }
```

C defines reading from uninitialized variables to be undefined behaviour, so anything can happen here. It is malpractice to write such code. Nevertheless, you can write such code, and execute it, and something happens. Most likely, the result is that there will be an arbitrary value in the variable (but undefined behaviour is allowed to do worse, due to compiler optimizations).

CBMC, therefore, goes ahead and analyzes such code as if there is an arbitrary value, which is easy to do with symbolic execution—indeed, this is the core strength of symbolic execution.

It will therefore report bounds (upper and lower—because `index` can be negative) and pointer dereference failures:

```
$ cbmc memory-safety.c --bounds-check --pointer-check
** Results:
memory-safety.c function read_buffer
[read_buffer.array_bounds.1] line 4 array 'buffer' lower bound in buffer[(signed long int)i]: FAILURE
[read_buffer.array_bounds.2] line 4 array 'buffer' upper bound in buffer[(signed long int)i]: FAILURE

memory-safety.c function read_pointer
[read_pointer.pointer_dereference.1] line 5 dereference failure: pointer outside object bounds in
     buffer[(signed long int)i]: FAILURE
```

Under the hood, CBMC is inserting assertions about the compile-time known array size (20) and comparing `index` to the array size. These assertions fail, given an unconstrained index, and so the SMT solver flags a violation, which you get to hear about.

If you run CBMC with `--trace` it will also tell you about the violated property and values that lead to it, e.g.

```
State 32 file memory-safety.c function main line 8 thread 0
----------------------------------------------------
  index=2147483647 (01111111 11111111 11111111 11111111)

State 35 file memory-safety.c function main line 9 thread 0
----------------------------------------------------
```

[1] https://model-checking.github.io/cbmc-training/cbmc/overview/debugging.html

```
  i=2147483647 (01111111 11111111 11111111 11111111)

Violated property:
  file memory-safety.c function read_buffer line 4 thread 0
  array 'buffer' upper bound in buffer[(signed long int)i]
  !((signed long int)i >= 20l)
```

Here, we see that if `index` was initialized to 2147483647, we would exceed buffer size, which is 20.

Concurrency. In principle CBMC is supposed to support concurrency, but when I try examples that I find on the Internet with CBMC 6.6.0 I just get an error message. Concurrency is one of the key applications of model checking, though, because it can check all execution interleavings. You might enjoy using some model checker for CS 343 next term, though it may not work with μC++.

Introducing Kani

Kani is a model checking tool for Rust code that uses CBMC underneath the hood. I've chosen to talk about it here because it has some nice syntactic sugar for allowing you to express some concepts. These concepts can also be expressed in CBMC, but it's messier.

We aren't really going to use Rust in this course (take ECE 459 if you want to use Rust), but I'll show off Rust features. The Rust build system is "cargo", and you can get it to run your tests for you. Attributes are great. You can label test methods in Rust like this:

```
1 mod tests {
2     use super::*;
3
4     #[test]
5     fn test_add() {
6         assert_eq!(add(1, 2), 3);
7     }
8 }
```

If you run "cargo test", then cargo runs all of the test cases in your project.

Now, consider the following Rust function[2], which fits the pattern of a fairly standard symbolic execution demonstration.

```
1   fn estimate_size(x: u32) -> u32 {
2     if x < 256 {
3         if x < 128 {
4             return 1;
5         } else {
6             return 3;
7         }
8     } else if x < 1024 {
9         if x > 1022 {
```

[2] https://model-checking.github.io/kani/tutorial-first-steps.html

```
10              panic!("Oh no, a failing corner case!");
11          } else {
12              return 5;
13          }
14      } else {
15          if x < 2048 {
16              return 7;
17          } else {
18              return 9;
19          }
20      }
21  }
```

We can see that there is one test input (out of 4 billion) that will trigger the panic. We have talked about property testing recently, when we were talking about fuzzing. Rust has a crate which does property testing, proptest[3]. A property test is quite unlikely to find this 1-in-4-billion bug:

```
1  use proptest::prelude::*;
2  proptest! {
3      #![proptest_config(ProptestConfig::with_cases(10000))]
4      #[test]
5      fn doesnt_crash(x: u32) {
6          estimate_size(x);
7      }
8  }
```

and so when you run it, you are almost surely going to get:

```
$ cargo test
   Compiling kani-example v0.1.0 (/home/plam/courses/stqam-2026-working-notes/notes/code/L15/kani-example)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.49s
     Running unittests src/main.rs (target/debug/deps/kani_example-9e52195d8a3f4738)

running 3 tests
test tests::test_bad_add ... ignored
test tests::test_add ... ok
test doesnt_crash ... ok

test result: ok. 2 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out; finished in 0.10s
```

(The Rust proptest crate starts with a random value and can "shrink" the value, but that's unlikely to get the needed value.)

Proof Harnesses

Rust has good syntax for so-called proof harnesses. You can write proof harnesses in C as well, but it's ugly and relies on conventions, so I'm teaching the easy-to-understand version.

```
1  #[cfg(kani)]
```

[3] https://altsysrq.github.io/proptest-book/

```
2 #[kani::proof]
3 fn check_estimate_size() {
4     let x: u32 = kani::any();
5     estimate_size(x);
6 }
```

This is conceptually exactly like the proptest. In practice, we explicitly create a variable `x` and we assign it the special value `kani::any()`, which is a symbolic execution thing—specifically, `x` gets a nondeterministic value. Instead of using the `proptest!` macro, we have a method that is conditionally compiled only by Kani (`#[cfg(kani)]`) and is labelled as a Kani proof harness (`#[kani::proof]`).

A proof harness is like a test, especially a property-based test. Remember "AAA": arrange, act, and assert. You still *arrange* conditions to call the system under test, in accordance with its preconditions, but you are now arranging with the help of symbolic values. *Act* is as normal, even if it's not acting with concrete values. And the *assert* phase (not present in the example above) still consists of assertions, but it may not be possible to write concrete assertions given an abstract input. The assertions may contain refinements of the postcondition that are supposed to be true due to what gets passed to the system in the arrange phase.

We can run proof harnesses with "cargo kani", which in turn invokes CBMC.

```
$ cargo kani
[...]
RESULTS:
Check 1: estimate_size.assertion.1
  - Status: FAILURE
  - Description: "Oh no, a failing corner case!"
  - Location: src/main.rs:18:13 in function estimate_size


SUMMARY:
 ** 1 of 1 failed
Failed Checks: Oh no, a failing corner case!
 File: "src/main.rs", line 18, in estimate_size
```

Bounded model checking's use of symbolic execution manages to find that there exists a value of `x` that reaches the `panic!`.

To elaborate, `kani::any()` represents all possible bit-value combinations valid for, in this case, the `u32` type of variable `x`. If you assign a `kani::any()` to some other type, it's supposed to mean all values of that type. Kani supports most Rust primitive types and some from the standard library.

There is an experimental feature, `--concrete--playback`, which shows you a counterexample test case.

```
$ cargo kani --concrete-playback=print -Z concrete-playback
[...]
```
```
/// Test generated for harness `check_estimate_size`
///
/// Check for `assertion`: "Oh no, a failing corner case!"

#[test]
fn kani_concrete_playback_check_estimate_size_6323293968261416923() {
    let concrete_vals: Vec<Vec<u8>> = vec![
        // 1023
```

```
        vec![255, 3, 0, 0],
    ];
    kani::concrete_playback_run(concrete_vals, check_estimate_size);
}
```

Remember, Kani searches the permissible bit patterns for `x`. Here, we have an array of u8s that contains the bit pattern representing 1023: [255, 3, 0, 0]. If we call `check_estimate_size()` with that bit pattern, then we trigger the panic.

More sophisticated state. From the Kani tutorial, another example where there is a bit more going on[4], which I've copied to `code/L15/kani-inventory-example`.

We have a struct `Inventory` with delegating methods implementing functionality:

```
pub type ProductId = u32;

pub struct Inventory {
    /// Every product in inventory must have a non-zero quantity
    pub inner: VecMap<ProductId, NonZeroU32>,
}

impl Inventory {
    pub fn update(&mut self, id: ProductId, new_quantity: NonZeroU32) {
        self.inner.insert(id, new_quantity);
    }

    pub fn get(&self, id: &ProductId) -> Option<NonZeroU32> {
        self.inner.get(id).cloned()
    }
}
```

We can write a proof harness that adds to the inventory and checks that the quantity is the same as was added:

```
// cargo kani --harness safe_update
#[kani::proof]
pub fn safe_update() {
    // Empty to start
    let mut inventory = Inventory { inner: VecMap::new() };

    // Create non-deterministic variables for id and quantity.
    let id: ProductId = kani::any();
    let quantity: NonZeroU32 = kani::any();
    assert!(quantity.get() != 0, "NonZeroU32 is internally a u32 but
        it should never be 0.");

```

[4] https://model-checking.github.io/kani/tutorial-nondeterministic-variables.html

```
12      // Update the inventory and check the result.
13      inventory.update(id, quantity);
14      assert!(inventory.get(&id).unwrap() == quantity);
15  }
```

The example on the webpage has an unwinding limit, but that doesn't seem to do anything, so I omitted it. The assertion about the NonZeroU32 is also unnecessary by construction: the type NonZeroU32 cannot be 0, and so `kani::any()` will not generate a value of 0 for it.

The symbolic execution is going to symbolically generate all valid values for `id` and `quantity` and check that the quantity associated with `id` is the quantity inserted.

Boundedness & loop unwinding

The really important thing to understand about bounded model checking, and how it differs from program verification e.g. with Dafny, is how it handles loops. Otherwise, at some level, both approaches generate a formula and passes it to a theorem prover—though the use of a proof harness for BMC also restricts its scope.

Let's consider an example from the Kani book.

```
1  fn initialize_prefix(length: usize, buffer: &mut [u8]) {
2      // Let's just ignore invalid calls
3      if length > buffer.len() {
4          return;
5      }
6
7      for i in 0..=length {
8          buffer[i] = 0;
9      }
10 }
```

This code has an off-by-one error. Also a loop. Here's a proof harness for the code.

```
1  #[cfg(kani)]
2  #[kani::proof]
3  #[kani::unwind(1)] // deliberately too low
4  fn check_initialize_prefix() {
5      const LIMIT: usize = 10;
6      let mut buffer: [u8; LIMIT] = [1; LIMIT];
7
8      let length = kani::any();
9      kani::assume(length <= LIMIT);
10
11     initialize_prefix(length, &mut buffer);
12 }
```

This proof harness specifies an *unwinding bound*. Since there is no loop invariant (Kani supports, but does not require, invariants), then Kani tries to cope with the loop by unwinding it some finite

number of times. Unwinding is a key concept for bounded model checking.

If you try to run this proof harness, then Kani complains:

```
Check 73: initialize_prefix.unwind.0
        - Status: FAILURE
        - Description: "unwinding assertion loop 0"
        - Location: src/lib.rs:7:5 in function initialize_prefix
```

Kani is copying-and-pasting the loop body N times (here $N = 1$) and after the Nth iteration Kani adds an assertion, which will fail in this case. If the unwinding bound is large enough such that this assertion never fails, then we have explored enough of the loop behaviour for our purposes. Here there is an explicit `LIMIT` on the buffer size, so we are still only checking for buffers of size up to 10.

Specifically, here is a C-style loop:

```
1 _Bool f();
2
3 int main() {
4   for(int i=0; i<100; i++) {
5     if(f()) break;
6   }
7   assert(0);
8 }
```

which gets unwound (under a limit of 2) to:

```
1 int main(int argc, char **argv) {
2   if(cond) {
3     BODY CODE COPY 1
4     if(cond) {
5       BODY CODE COPY 2
6       assert(!cond); // unwinding assertion
7     }
8   }
9 }
```

For the Rust example, we can increase the unwinding limit, and then the unwinding assertion doesn't fail anymore, but a new assertion does.

```
Check 2: initialize_prefix.assertion.1
        - Status: FAILURE
        - Description: "index out of bounds: the length is less than or equal to the given index"
        - Location: src/lib.rs:8:9 in function initialize_prefix
```

We can correct the off-by-one error with the length check in initialize_prefix and then Kani's bounded model checking succeeds. So we know that this code is correct for buffers of length at most 10.

About boundedness. Taking a step back: the *bounded* in bounded model checking means that: (1) it can check a finite number of loop iterations, and assert that the loop never takes more than

that number of iterations; (2) data structures are of at most a bounded size. The boundedness of data structures is why BMC doesn't report answers that are unconditionally true about the code. Maybe there is a counterexample, but it requires a larger bound than the one you checked. Unlike with the unwinding assertion, you don't know anything for sure about behaviour outside the verified regime. If your bound is large, then the possibility of an error may be unlikely, but it's not impossible.

There are technical considerations involved with using `kani::any()` in the context of unbounded data structures. The documentation says more, but basically: either you instead use `BoundedArbitrary` and specify a bound; or you add to your data structure a bounded number of times. Kani is going to quickly become slow as the bound grows: it's an exponential growth problem.

CBMC: Test generation for PIDs

Now that we've talked about loop unwinding, here's a detour back to CBMC. Next term, in SE 380, you will encounter PID controllers, which can be implemented in software. One might want to verify, or at least test, such software. The CBMC page talks about generating test suites for a PID controller using CBMC: `https://www.cprover.org/cprover-manual/test-suite/`.

An implementation of one iteration of the PID controller looks like this:

```
1  void climb_pid_run()
2  {
3    float err=estimator_z_dot-desired_climb;
4
5    float fgaz=climb_pgain*(err+climb_igain*climb_sum_err)+
        CLIMB_LEVEL_GAZ+CLIMB_GAZ_OF_CLIMB*desired_climb;
6
7    float pprz=fgaz*MAX_PPRZ;
8    desired_gaz=((pprz>=0 && pprz<=MAX_PPRZ) ? pprz : (pprz>MAX_PPRZ ?
         MAX_PPRZ : 0));
9
10   /** pitch offset for climb */
11   float pitch_of_vz=(desired_climb>0) ? desired_climb*
        pitch_of_vz_pgain : 0;
12   desired_pitch=nav_pitch+pitch_of_vz;
13
14   climb_sum_err=err+climb_sum_err;
15   if (climb_sum_err>MAX_CLIMB_SUM_ERR) climb_sum_err=
        MAX_CLIMB_SUM_ERR;
16   if (climb_sum_err<-MAX_CLIMB_SUM_ERR) climb_sum_err=-
        MAX_CLIMB_SUM_ERR;
17 }
```

and it has to be verified in the context of a driver, which provides constrained symbolic inputs to the function. Here, `__CPROVER_assume` constrains the input, while `__CPROVER_input` and `__CPROVER_output` mark the indicated variables as inputs and outputs. There is no real documentation that I could find about what `__CPROVER_output` actually does.

```
1  int main()
2  {
3
4    while(1)
5    {
6      /** Non-deterministic input values */
7      desired_climb=nondet_float();
8      estimator_z_dot=nondet_float();
9
10     /** Range of input values */
11     __CPROVER_assume(desired_climb>=-MAX_CLIMB && desired_climb<=
           MAX_CLIMB);
12     __CPROVER_assume(estimator_z_dot>=-MAX_CLIMB && estimator_z_dot
           <=MAX_CLIMB);
13
14     __CPROVER_input("desired_climb", desired_climb);
15     __CPROVER_input("estimator_z_dot", estimator_z_dot);
16
17     climb_pid_run();
18
19     __CPROVER_output("desired_gaz", desired_gaz);
20     __CPROVER_output("desired_pitch", desired_pitch);
21   }
22
23   return 0;
24 }
```

This is an infinite loop, so CBMC can never fully unwind it. It can unwind it to a fixed depth, though I'm a bit suspicious about what happens when I do that.

CBMC can also generate test cases by printing out a set of concrete inputs that satisfy the specified criteria. The example on the page that I've linked above prints out inputs achieving MC/DC, a coverage criterion that I don't think is worth talking about in this class. MC/DC does come up in avionics software, which is also what this PID controller does. You can also ask CBMC to use other criteria, e.g. branch coverage.

```
$ cbmc pid.c --cover mcdc --show-test-suite --unwind 6
[...]

** coverage results:
[...]
** 36 of 37 covered (97.3%)

Test suite:
desired_climb=2.097152e+6f, estimator_z_dot=2.097149e+6f, desired_climb=-65538.0f, estimator_z_dot=-6.612582e+7f, desired_climb=
desired_climb=-2.097152e+6f, estimator_z_dot=-2.097149e+6f, desired_climb=-65538.0f, estimator_z_dot=-6.612582e+7f, desired_clim
[...]
```

We can see that CBMC prints out concrete input values for the `climb_pid_run()` function that satisfy the assumptions. These can be put into regression tests, for instance. As discussed earlier, one would need an oracle to validate the computed outputs.

Contracts

Dafny verification is heavily based on contracts. Kani can also do contracts[5]; the code is available in `code/L16/kani-contracts`.

Here is a gcd implementation, along with a contract.

```
#[kani::requires(min != 0 && max != 0)]
#[kani::ensures(|result| *result != 0 && max % *result == 0 && min % *result == 0)]
#[kani::recursion]
fn gcd(mut max: u8, mut min: u8) -> u8 {
    if min > max {
        std::mem::swap(&mut max, &mut min);
    }

    let rest = max % min;
    if rest == 0 { min } else { gcd(min, rest) }
}
```

The implementation is a straightforward recursive implementation which should be familiar from MATH 135. The contract says that it's allowed to call this function with any non-zero unsigned 8-bit integers; the result will be a divisor of both `min` and `max`. For technical reasons, it also declares the recursion in the function. This is a partial contract—it's missing a detail—but what it says is appropriate, as far as it goes.

You can trigger exhaustive checking of the implementation against the contract using the invocation `cargo kani -Z function-contracts`. Even though it's only 8-bit integers, it still takes 11 seconds on my laptop. But we then can conclude that `gcd` is correct. I tried to run it for 16-bit integers but it takes a lot longer and I gave up.

Having verified a contract for `gcd`, you can then tell Kani to use its contract when verifying any functions that subsequently call `gcd`, instead of verifying it from scratch every time:

```
// Assume foo() invokes gcd().
// By using stub_verified, we tell Kani to replace
// invocations of gcd() with its verified contracts.
#[kani::proof]
#[kani::stub_verified(gcd)]
fn check_foo() {
    let x: u8 = kani::any();
    foo(x);
}
```

This is called stubbing in the Kani world.

[5] `https://model-checking.github.io/kani/reference/experimental/contracts.html`

Unsafe Rust

I may go into more detail on this if time permits, but for now I'll just comment on unsafe Rust.

We've seen Kani find assertion violations. For languages like C, we also want to verify the absence of undefined behaviour.

One would usually write programs in the subset of Rust known as Safe Rust. Life is great in Safe Rust and there is no undefined behaviour. However, for performance and for interoperability with other languages, one may have to use Unsafe Rust, which enables a few additional (unsafe) features, and also opens the door to undefined behaviour.

It is possible to use Kani, and other tools, to ensure that even Unsafe Rust code is free of undefined behaviour and respects its contract. One can start with the tools that we've seen here. Kani would require a good proof harness to ensure the absence of undefined behaviour. This includes bounds errors and overflows. (Overflows are panics in debug mode and wrap in release mode.)

Using Bounded Model Checking

The Kani book suggests a three-step approach to doing bounded proofs[6]:

- bound the problem size, in the proof harness (e.g. define a constant LIMIT);
- guess an unwind bound and increase it until there is no more unwinding assertion failure; and
- if Kani takes too long with the bound you've guessed, decrease the problem size.

Probably most of the time if you can prove what you need for a reasonably small bound, you'll find most of the errors. But this is not a proof. But you also don't have to specify invariants. Apparently this works well for many problems, but notoriously not for parsing.

I'm not going to summarize it here, but there are some good practices for writing verification-friendly (and good) C code at `https://model-checking.github.io/cbmc-training/management/Code-for-verification.html`.

[6] `https://model-checking.github.io/kani/tutorial-loop-unwinding.html`