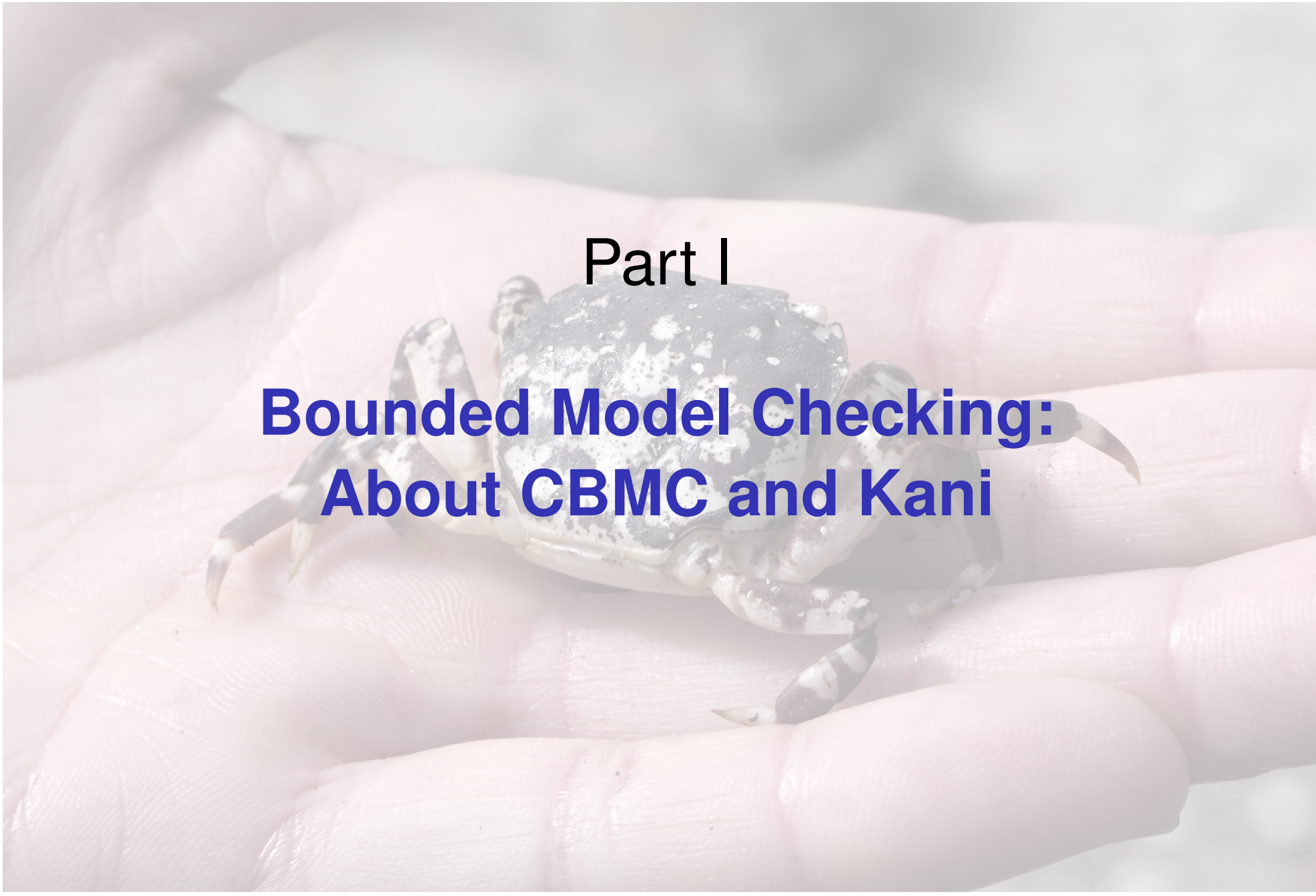
Software Testing, Quality Assurance & Maintenance—Lecture 15

Patrick Lam
University of Waterloo

March 2, 2026



Part I

Bounded Model Checking: About CBMC and Kani

Seen on the Internet

Bounded model checking is unit testing with superpowers.

— Karl Schultheisz,
https://kdsch.org/post/cbmc-technique/

About Bounded Model Checking

Bounded Model Checking (BMC): a way to statically verify code properties.

Tools: CBMC (for C) and Kani (for Rust).

More lightweight than full program verification (Dafny).

- Don’t need loop invariants.
- Don’t necessarily get a guarantee.
- More exhaustive than testing.

Goals

Be able to apply modern model checking tools (CBMC/Kani);

Understand how they work and when they fail;

Know what guarantees they are giving you.

Part II

Exploring CBMC

The simplest possible program

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

Let's try to verify it.

Verifying the simplest possible program

```
$ 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
```

It is, indeed, what we would expect; we can also change the program and make it pass.

The symbolic execution example (in C)

```
// cbmc div-by-zero.c --function foo
unsigned int foo(unsigned int x, unsigned int y) {
  if (x > y)
    return x / y;
  else
    return y / x;
}
```

Finding possible division by 0

```
$ cbmc div-by-zero.c --function foo
[...]
** 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 `--show-properties`:

```
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)
```

Verification Conditions

A verification condition (VC) is a logical formula—an implication—that implies that a desired program property holds.

Showing Verification Conditions

Here's the VC for 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)
```

CBMC automatically puts in an assertion checking for division by zero.

Using symbolic execution, it knows the conditions under which the y/x branch is executed,
recording these conditions in variable guard1—
true when $\neg(y \geq x)$.

What's Needed to Verify

```
{-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)
```

Verification succeeds on this branch if one can establish that input `x` is nonzero, given that guard1 is false (which was necessary to reach this branch).

What CBMC can check

CBMC can check:

- 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,
- pointer validity.

Undefined behaviour example

(https://model-checking.github.io/cbmc-training/cbmc/overview/debugging.html)

```
#define SIZE 20
char buffer[SIZE];

char read_buffer(int i)
  { return buffer[i];     }
char read_pointer(int i)
  { return *(buffer + i); }

int main() {
  int index; // uninitialized
  read_buffer(index);
  read_pointer(index);
}
```

Focussing

```
int index; // uninitialized
buffer[index]; // read at uninitialized index
```

C says this is undefined behaviour; anything can happen.

Don't write such code!

CBMC can analyze such code as if there is an arbitrary value in `index`:

a core strength of symbolic execution.

Attempting to Verify

CBMC reports bounds (upper and lower—`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 bu
[read_buffer.array_bounds.2] line 4 array 'buffer' upper bound in bu

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

Under the hood: CBMC adds assertions about the compile-time known array size (20) and compares `index` to the array size.

These assertions fail, given an unconstrained index, and so the SMT solver flags a violation.

Learning more

CBMC `--trace` tells you about the violated property and values that lead to it.

```
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
----------------------------------------------------
  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)
```

i.e. if `index` were initialized to 2147483647, we exceed buffer size (= 20).

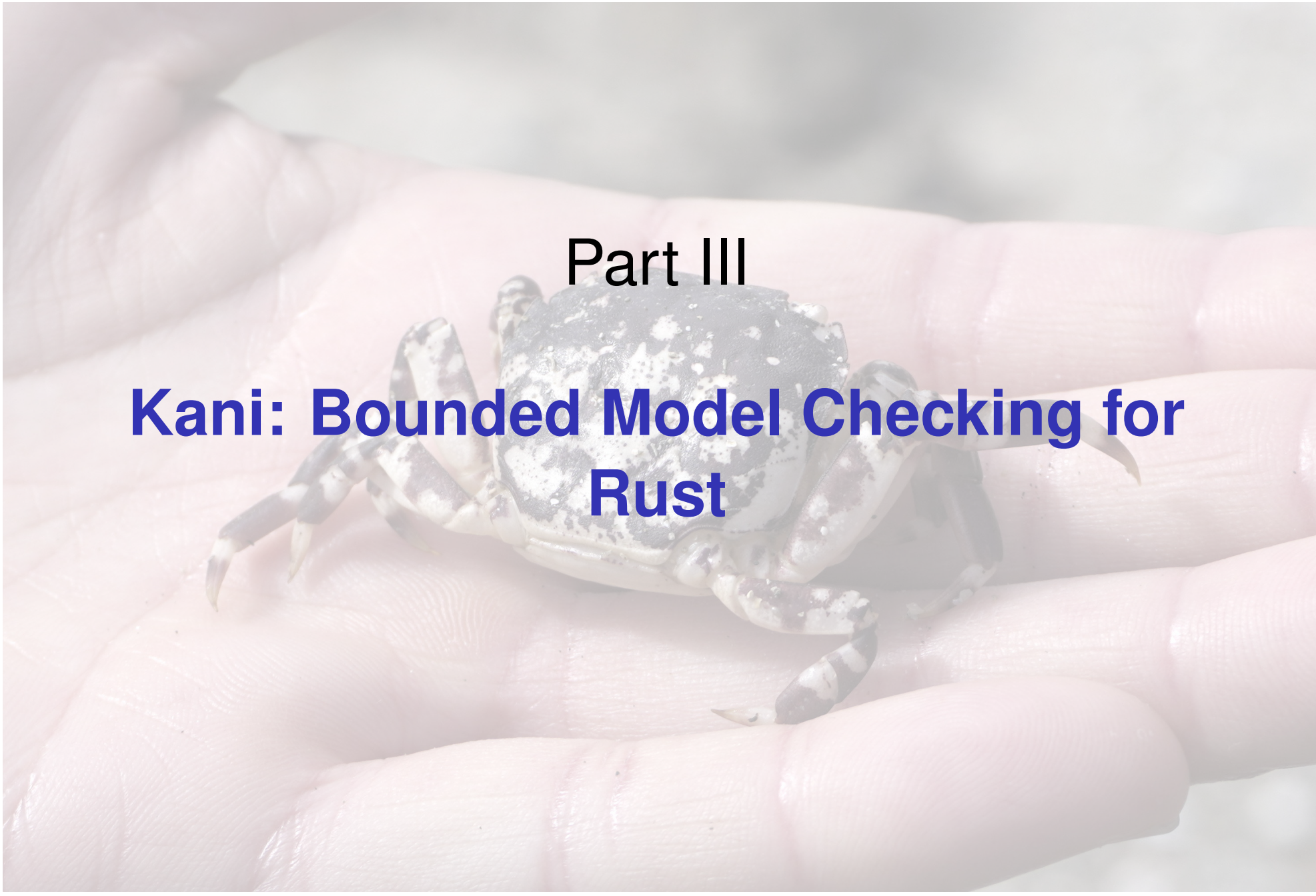
CBMC & Concurrency?

CBMC is supposed to support concurrency.

I tried examples from the Internet on CBMC 6.6.0, and get errors.

Concurrency is a key application of model checking: it can check all execution interleavings.

SE students: You might enjoy using some model checker for CS 343 next term, though it may not work with μC++.



Part III

Kani: Bounded Model Checking for Rust

Why Kani?

Kani does model checking of Rust code using CBMC.

Kani syntax shows some concepts clearly.

You shouldn't need to understand Rust to use Kani.

Rust: attributes

(code in `code/L15/rust-tests`)

Here we have the “test” attribute.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }
}
```

Use `cargo test` to run tests.

Rust: standard symbolic execution demonstration

```
fn estimate_size(x: u32) -> u32 {
    if x < 256 {
        if x < 128 {
            return 1;
        } else {
            return 3;
        }
    } else if x < 1024 {
        if x > 1022 {
            panic!("Oh no, a failing corner case!");
        } else {
            return 5;
        }
    } else {
        if x < 2048 {
            return 7;
        } else {
            return 9;
        }
    }
}
```

One test input (out of 4 billion) will trigger the panic.

Proptest won't find the panic

When talking about fuzzing, we also talked about property testing.

proptest crate does Rust property testing.

A property test is quite unlikely to find this 1-in-4-billion bug:

```
use proptest::prelude::*;
proptest! {
  #![proptest_config(ProptestConfig::with_cases(10000))]
  #[test]
  fn doesnt_crash(x: u32) {
      estimate_size(x);
  }
}
```

Running proptest

Almost always, you'll see:

```
$ cargo test
   Compiling kani-example v0.1.0 (/home/plam/courses/stqam-2026-working-no
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.49s
     Running unittests src/main.rs (target/debug/deps/kani_example-9e52195

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
```

(Rust proptest crate starts with a random value, but that's unlikely to get the needed value.

It can also "shrink" the obtained value—not helpful here.)

Proof Harnesses

Idea: a proof harness is like a test case, but there is (bounded) proving.

```
#[cfg(kani)]
#[kani::proof]
fn check_estimate_size() {
    let x: u32 = kani::any();
    estimate_size(x);
}
```

Proptest vs proof harness

Let's compare.

```
#[cfg(kani)]
#[kani::proof]
fn check_estimate_size() {
    let x: u32 = kani::any();
    estimate_size(x);
}
```

```
#[test]
fn doesnt_crash(x: u32) {
    estimate_size(x);
}
```

...pretty similar!

Looking at the Proof Harness

```
#[cfg(kani)]
#[kani::proof]
fn check_estimate_size() {
    // nondeterministic value
    let x: u32 = kani::any();
    estimate_size(x);
}
```

Instead of relying on proptest to call us, explicitly create a nondeterministic value.

Proof Harnesses as Tests

A proof harness is like a test.

Arrange:

```
let x: u32 = kani::any();
```

Act:

```
estimate_size(x);
```

Assert:

not present here.

More about proof harnesses and assertions

For a proof harness:

Assert phase still consists of assertions; but, may not be possible to write concrete assertions given abstract input.

Assertions summarize what the act phase did.
May be related to the postcondition.

Running the proof harness

Run proof harnesses with “cargo kani”; 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
```

Kani finds that there exists some `x` that reaches the `panic!`.

`kani::any()` and symbolic execution

`kani::any()` represents all possible bit-values valid for `u32` (here, because it's the type of `x`)—like symbolic `X`.

If you assign `kani::any()` to some other type, it takes on all values of that type.

Kani supports most Rust primitive types and some from the standard library.

Concrete test cases from Kani

Experimental feature `--concrete--playback` shows 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_632329396826141692 3() {
    let concrete_vals: Vec<Vec<u8>> = vec![ // 1023
        vec![255, 3, 0, 0], ];
    kani::concrete_playback_run(concrete_vals, check_estimate_size);
}
```

Kani searches permissible bit patterns for `x`.

Here: array of u8s containing the bit pattern representing 1023: [255, 3, 0, 0].

If we call `check_estimate_size()` with that bit pattern, then we trigger the panic.

Kani inventory example: more sophisticated state

We've now seen bounded model checking on `u32`s.

What about more sophisticated types?

Here's a user-defined type, which uses a (non-stdlib) `VecMap`.

```
pub type ProductId = u32;

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

There is also an invariant about the `VecMap`.

`Inventory` implementation

It delegates.

```
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()
    }
}
```

An `Inventory` proof harness

Adds to the inventory and checks that the quantity matches:

```
// cargo kani --harness safe_update
#[kani::proof]
pub fn safe_update() {
    // Empty to start (deterministic)
    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 must be nonzero.");

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

How the proof harness works

Arrange: (Symbolically) generate all valid values for `id` and `quantity` using symbolic execution;

Act: call `update`;

Assert: check that the quantity associated with `id` is the quantity inserted.

Proof harness notes

I omitted an unwinding limit from the example (more soon on that).

`assert!(quantity.get() != 0)` unnecessary by construction, since `NonZeroU32` is non zero.

Part IV

BMC: Boundedness & Unwinding

Kani vs Dafny

Recall:

- Kani = bounded model checking
- Dafny = auto-active verification

Key difference: how to handle loops.

Otherwise: at some level, both generate a formula and get it solved.

(Another difference: BMC uses proof harnesses to focus efforts.)

Kani example: loop

```
fn initialize_prefix(length: usize,
                     buffer: &mut [u8]) {
    // Let's just ignore invalid calls
    if length > buffer.len() {
        return;
    }

    for i in 0..=length {
        buffer[i] = 0;
    }
}
```

There's an off-by-one error.

Also a loop.

Proof harness for initialize_prefix

```
#[cfg(kani)]
#[kani::proof]
#[kani::unwind(1)] // deliberately too low
fn check_initialize_prefix() {
    const LIMIT: usize = 10;
    let mut buffer: [u8; LIMIT] = [1; LIMIT];

    let length = kani::any();
    kani::assume(length <= LIMIT);

    initialize_prefix(length, &mut buffer);
}
```

any/assume idiom

```
let length = kani::any();
kani::assume(length <= LIMIT);
```

This is a standard idiom for letting `length` be anything, except that it is no greater than `LIMIT`.

What about loops?

Symbolic execution needs to reason about all paths.

But when there's a loop,
then there is an unbounded number of paths!

One strategy: loop invariants.

Here, there are no loop invariants
(Kani supports but does not require them).

So, Kani unwinds the loop some bounded number of times.

Unwinding is a key concept for bounded model checking.

Unwinding bound

Let's set the unwinding bound to 2:

```
#[kani::unwind(2)] // deliberately too low
```

which applies to the loop in initialize_prefix:

```
for i in 0..=length {
    buffer[i] = 0; }
```

to yield

```
i = 0;
if i <= length {
  buffer[i] = 0;
  i += 1;
  if i <= length {
    buffer[i] = 0;
    i += 1;
    assert (!(i <= length));
  }
}
```

What happens?

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
```

That is, the *unwinding assertion* failed:

```
assert (!(i <= length));
```

because the loop could execute for more iterations than the bound.

When unwinding might work

Remember that we had a LIMIT:

```
const LIMIT: usize = 10;
let mut buffer: [u8; LIMIT] = [1; LIMIT];
```

If the unwinding bound is large enough (i.e. bigger than LIMIT), then the unwinding assertion doesn't fail.

Trying again with a bigger bound

We can increase the unwinding bound
(this dial goes to 11).
Then we get a new error:

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

This was the off-by-one error. Fix it:

```
if length >= buffer.len() {
```

and verification succeeds:

```
SUMMARY:
 ** 0 of 73 failed

VERIFICATION:- SUCCESSFUL
Verification Time: 0.19064504s
```

So this code is correct for buffers of length at most 10.

About boundedness

Bounded in bounded model checking means:

1. can check finite # of loop iterations, asserting that the loop never loops more than that;
2. data structures are of at most a bounded size.

Boundedness of data structures is why BMC doesn't report unconditional truth: maybe there is a counterexample, but it needs a bigger bound.

Unlike with the unwinding assertion, you don't know anything about bigger sizes.

Large bound gives more confidence, but not a guarantee.

`kani::any()` and unbounded data structures

Per the documentation, basically:

- instead use `BoundedArbitrary` and specify a bound; or,
- add to data structure a bounded # of times.

Kani quickly becomes slow as the bound grows—exponential growth bites again.

Part V

CBMC: PID controllers & test generation

PID controllers

SE students will encounter these in SE 380 (controls) next fall.

CBMC documentation shows test generation for them.

This code runs in a loop, so we'll use loop unwinding.

PID controller: one iteration

```
void climb_pid_run()
{
  float err=estimator_z_dot-desired_climb;

  float fgaz=climb_pgain*(err+climb_igain*climb_sum_err)+
                CLIMB_LEVEL_GAZ+CLIMB_GAZ_OF_CLIMB*desired_climb;

  float pprz=fgaz*MAX_PPRZ;
  desired_gaz=((pprz>=0 && pprz<=MAX_PPRZ)
                ? pprz : (pprz>MAX_PPRZ ? MAX_PPRZ : 0));

  /** pitch offset for climb */
  float pitch_of_vz=(desired_climb>0)
                ? desired_climb*pitch_of_vz_pgain : 0;
  desired_pitch=nav_pitch+pitch_of_vz;

  climb_sum_err=err+climb_sum_err;
  if (climb_sum_err>MAX_CLIMB_SUM_ERR) climb_sum_err=MAX_CLIMB_SUM_ERR;
  if (climb_sum_err<-MAX_CLIMB_SUM_ERR) climb_sum_err=-MAX_CLIMB_SUM_ERR;
}
```

PID controller: driver

```
int main() {
 while(1) {
  /** Non-deterministic input values */
  desired_climb=nondet_float();
  estimator_z_dot=nondet_float();

  /** constrain range of input values */
  __CPROVER_assume(desired_climb>=-MAX_CLIMB && desired_climb<=MAX_CLIMB);
  __CPROVER_assume(estimator_z_dot>=-MAX_CLIMB && estimator_z_dot<=MAX_CLI
  __CPROVER_input("desired_climb", desired_climb);
  __CPROVER_input("estimator_z_dot", estimator_z_dot);

  climb_pid_run();

  __CPROVER_output("desired_gaz", desired_gaz);
  __CPROVER_output("desired_pitch", desired_pitch);
 }

 return 0;
}
```

What CBMC can do with the PID controller driver

Infinite loop, so CBMC can't fully unwind. Can unwind to a fixed depth, allegedly.

CBMC can generate test cases (concrete inputs satisfying specified criteria).

CBMC doco's example achieves MC/DC (criterion used for avionics), but we won't talk about that.

Can ask CBMC to use e.g. branch coverage as well.

MC/DC example

```
$ 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=-6
desired_climb=-2.097152e+6f, estimator_z_dot=-2.097149e+6f, desired_climb=
[...]
```

We get concrete inputs for `climb_pid_run()`, satisfying `assume`s.

Can put into regression tests;
need oracle to know if outputs correct.

Part VI

Back to Kani: Contracts

Contracts: the foundation of verification

```
#[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) }
}
```

A contract includes a `requires` clause (precondition) and an `ensures` clause (postcondition).

About contracts

```
#[kani::requires(min != 0 && max != 0)]
#[kani::ensures(|result| *result != 0 && max % *result == 0 &&
                min % *result == 0)]
```

Verification proof obligation:

assuming the precondition,
show that the postcondition must hold after the method execution terminates.

Dafny always uses contracts; Kani can use contracts.

The gcd 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) }
}
```

Implementation is straightforward recursive.

Precondition: OK to call with any nonzero 8-bit uints.

Postcondition: result is divisor of both `min` and `max`.
missing a detail: incomplete but not wrong.

Contract checking with Kani

```
$ cargo kani -Z function-contracts
[...]
SUMMARY:
 ** 0 of 368 failed (4 unreachable)

VERIFICATION:- SUCCESSFUL
Verification Time: 12.000412s
```

Can conclude that `gcd` meets its spec, for 8-bit integers.

Tried for 16-bit integers. Takes too long. Gave up.

Using contracts

Having verified the `gcd` contract, can then tell Kani to use the contract when verifying any functions that call `gcd`.

Don't need to re-prove the wheel 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.

Part VII

Unsafe Rust

Teaser: Kani and Unsafe Rust

Will probably say more.

Have seen Kani find assertion violations.
For C, must also verify the absence of undefined behaviour.

Mostly one uses Safe Rust,
    with no undefined behaviour.
But, for performance & interop, may have to use Unsafe Rust,
    with undefined behaviour.

Kani and Unsafe Rust

Kani can ensure absence of undefined behaviour in Unsafe Rust.
Other tools can do this too.

For Kani: need a good proof harness to ensure no undefined behaviour.
Harness must rule out bounds errors and integer overflows.

Part VIII

Using Bounded Model Checking

Three-step approach from Kani book

- bound problem size in proof harness (define a `LIMIT`);
- guess an `unwind` bound, increase as necessary;
- if Kani takes too long, decrease the `LIMIT`.

Probably if you can prove with a reasonably small bound, your code is fine. But that's not a proof.

This does not require invariants.

Works well for many problems, but not for parsing.

Also you can find good practices for verification-friendly C.