Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 18 — April 2, 2025

Patrick Lam *version 1*

This final lecture is going to be about bounded model checking (BMC), which is another way of statically verifying code properties. The goal with BMC is to be more lightweight than Dafny—in particular, you don't need to write loop invariants when using a bounded model checker, but also, you might not get a guarantee from a successful BMC run. It also happens that BMC works with languages like C and Rust.

Let's consider some code again[1]. This is in Rust, and is from the Kani tutorial. Kani is a bounded model checking tool for Rust, currently under active development.

```
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;
        }}}
```

You have seen at least two techniques that can find the panic here—symbolic execution and Dafny's formal verification. Neither of these techniques is lightweight.

Kani is another technique which can find this panic. As I alluded to above, one way of thinking about it is "like Dafny, but doesn't require loop invariants". For completeness, the following Kani *proof harness* will find this panic.

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

But let's introduce Kani using a different example[2], which I'll present in both Dafny and Kani.

Here's a Rust Rectangle implementation.

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

[2]https://model-checking.github.io/kani-verifier-blog/2022/05/04/announcing-the-kani-rust-verifier-project.html

```
#[derive(Debug, Copy, Clone)]
struct Rectangle {
    width: u64,
    height: u64,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }

    fn stretch(&self, factor: u64) -> Option<Self> {
        let w = self.width.checked_mul(factor)?;
        let h = self.height.checked_mul(factor)?;
        Some(Rectangle {
            width: w,
            height: h,
        })
    }
}
```

Pretty standard code. I can also provide the Dafny version:

```
class Rectangle {
    var width: int
    var height: int
}

predicate can_hold(self: Rectangle, other: Rectangle)
    reads self, other
{
    self.width > other.width ∧ self.height > other.height
}

method stretch(self: Rectangle, factor: nat) returns (rv : Rectangle) {
    rv := new Rectangle;
    rv.width := self.width * factor;
    rv.height := self.height * factor;
    return rv;
}
```

It would probably be more idiomatic object-oriented code to put stretch on the Rectangle, but it doesn't really matter for our purposes. In terms of proofs, we could state and maintain class invariants, which you should have seen in CS 247.

Going back to Rust, we can write a test case:

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

    #[test]
    fn stretched_rectangle_can_hold_original() {
        let original = Rectangle {
            width: 8,
            height: 7,
        };
        let factor = 2;
        let larger = original.stretch(factor);
        assert!(larger.unwrap().can_hold(&original));
    }
}
```

This is a test case which checks one specific input to can_hold. It succeeds, which is nice, but doesn't tell us that much.

We also know how to write Dafny test cases:

```
method {:test} TestStretchedRectangleCanHoldOriginal() {
    var original := new Rectangle;
    original.width := 8; original.height := 7;
    var factor := 2;
    var larger := stretch(original, factor);
    expect can_hold(larger, original);
}
```

We can replace the expect with an assert, but that won't verify, because there is no postcondition for method stretch. If we add a postcondition, then the proof for the test case goes through, but that's still not a general result. It's just about that particular rectangle.

There is a Rust crate (library) that does property-based testing, where it generates thousands of test cases. We can run both of these tests using cargo test.

```
#[cfg(test)]
mod proptests {
    use super::*;
    use proptest::prelude::*;
    use proptest::num::u64;

    proptest! {
        #[test]
        fn stretched_rectangle_can_hold_original(width in u64::ANY,
            height in u64::ANY,
            factor in u64::ANY) {
            let original = Rectangle {
                width: width,
                height: height,
            };
            if let Some(larger) = original.stretch(factor) {
                assert!(larger.can_hold(&original));
            }
        }
    }
}
```

This succeeds too. But we can go one step further and verify that the property that we've tested holds for any input, by running cargo kani --harness stretched_rectangle_can_hold_original.

```
#[cfg(kani)]
mod verification {
    use super::*;

    #[kani::proof]
    pub fn stretched_rectangle_can_hold_original() {
        let original = Rectangle {
            width: kani::any(),
            height: kani::any(),
        };
        let factor = kani::any();
        if let Some(larger) = original.stretch(factor) {
            assert!(larger.can_hold(&original));
        }
    }
}
```

This proof harness tells Kani to test the behaviour for *any* width, height, and factor. When we invoke Kani, we see that it runs a whole bunch of tests, looking mostly for safety violations (in unsafe Rust, of which there is none here). However, Kani also reports that the assertion fails. Why?

It turns out that we're missing some preconditions. Kani does support contracts like those we've seen in Dafny, but we'll just encode them directly in the test harness.

```
        kani::assume(0 != original.width);  //< explicit requirements
        kani::assume(0 != original.height); //<
        kani::assume(1 < factor);           //<
```

With these assumes (effectively preconditions), Kani verifies the code.

Similarly, in Dafny:

```
method stretch(self: Rectangle, factor: nat) returns (rv : Rectangle)
    ensures rv.width = self.width * factor ∧ rv.height = self.height * factor
{
    rv := new Rectangle;
    rv.width := self.width * factor;
    rv.height := self.height * factor;
    return rv;
}

method stretched_rectangle_can_hold_original(original: Rectangle, factor: nat)
    requires original.width > 0 ∧ original.height > 0 ∧ factor > 1
{
    var larger := stretch(original, factor);
    assert can_hold(larger, original);
}
```

Other things that Kani does. We've seen Kani find assertion violations. For languages like C, we also want to verify the absence of undefined behaviour. Safe Rust doesn't have undefined behaviour. (Unsafe Rust does, and Kani can detect some of that). Kani statically finds some errors that would occur dynamically, when you provide it with the correct proof harness. This includes bounds errors and overflows. (Overflows are panics in debug mode and wrap in release mode.)

Loops and Kani

The really important thing to understand about bounded model checking, and how it differs from Dafny, is how it handles loops. Otherwise, it generates a formula and passes it to a theorem prover.

Let's consider an example from the Kani book.

```
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;
    }
}
```

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

```
#[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);
}
```

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—like you did in the symbolic execution assignment.

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 an assertion will fail. 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.

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.

The Kani book suggests a three-step approach to doing bounded proofs such as these.

- bound the problem size, in the proof harness (here done with 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 it 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.