Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 16 — March 6, 2025

Patrick Lam *version 1*

Here's a different example[1], which I'll present in both Dafny and Kani.

Here's a Rust Rectangle implementation.

```
#[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;
}
```

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

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);
}
```