Verus — SMT-based verification of low-level systems code

github.com/secure-foundations/verus

Andrea Lattuada – ETH Zurich
Chris Hawblitzel – Microsoft Research
Bryan Parno, Yi Zhou, Chanhee Cho, Travis Hance – Carnegie Mellon University
Jon Howell – VMware Research

This **30 minute** talk will introduce Verus, an SMT-based tool we are building to verify full functional correctness of systems code written in Rust. I am planning to demonstrate Verus via live demos interspersed in the talk: the examples in this proposal are samples of the snippets that I will verify with Verus in the demos.

Introduction. The Rust programming language, with its combination of high-level features, memory safety, and low-level "unsafe" primitives has proven to be an excellent tool to write high-performance systems software, with direct interactions with the hardware.

When looking at Rust in the context of software verification we observed that functions using the safe subset of Rust without interior mutability are pure, deterministic transformations, even when they manipulate mutable references (as they can be logically rewritten as consuming the initial value of the reference, and returning the new value). By statically preventing aliased mutable references Rust relieves the burden on the developer of reasoning about data races; when verifying imperative code in languages that do not restric aliasing, the developer similarly has to explicitly manage the heap and potential aliasing. In our experience, a Rust-like linear type system can be an excellent aid to Floyd-Hoare SMT-based verification of executable code.

We leverage these intuitions in the design of our new tool for SMT-based verification of Rust code, Verus. We are building Verus to efficiently verify full functional correctness of low-level systems code written in a safe Rust dialect that supports expressing specifications and proofs.

Figure 1 shows a snippet of a Rust function Ⓖ verified with Verus.

Verus' design. Verus leverages the fact that safe Rust has a natural, direct encoding in SMT (for code without interior mutability, which we need to address separately). Algebraic data types (ADTs) and sound generics complement safe Rust to make it a solid basis for a mathematical language for expressing specifications and proofs. In Figure 1, the `fibo` function Ⓐ is a pure mathematical definition of the n-th Fibonacci number. When functions represent theorems (like `lemma_fibo_is_monotonic` Ⓑ), expressing complex proofs like lemmas-by-induction is straightforward via a simple recursive call Ⓔ.

```
#[spec]
fn fibo(n: nat) -> nat { Ⓐ
  decreases(n);
  if n == 0 { 0 } else if n == 1 { 1 }
  else { fibo(n - 2) + fibo(n - 1) }
}

#[proof]
fn lemma_fibo_is_monotonic(i:nat, j:nat) { Ⓑ
  requires(Ⓒ i<=j);
  ensures(Ⓓ fibo(i) <= fibo(j));
  decreases(j-i);

  if i<2 && j<2 { } else if i==j { }
  else if i==j-1 {
    reveal_with_fuel(fibo, 2);
    lemma_fibo_is_monotonic(i, j-1); Ⓔ
  } else {
    lemma_fibo_is_monotonic(i, j-1);
    lemma_fibo_is_monotonic(i, j-2);
  }
}

#[spec]
fn fibo_fits_u64(n: nat) -> bool { Ⓕ
  fibo(n) <= u64::MAX }

#[exec]
fn fibo_impl(n: u64) -> u64 { Ⓖ
  requires(Ⓗ fibo_fits_u64(n));
  ensures(Ⓘ |result: u64| result == fibo(n));
  if n == 0 { return 0; }
  let mut prev: u64 = 0; let mut cur: u64 = 1;
  let mut i: u64 = 1;
  while i < n {
    invariant([Ⓙ
      i > 0, i <= n,
      fibo_fits_u64(n as nat), fibo_fits_u64(i as nat),
      cur == fibo(i), prev == fibo(i as nat - 1),
    ]);
    let new_cur = cur + prev;
    prev = cur; cur = new_cur; i = i + 1;
    lemma_fibo_is_monotonic(i, n);
  }
  cur
}
```

Figure 1: A proof of correcntess of a function computing the n-th Fibonacci number. We use circled letters, similar to Ⓩ, to mark points of interest in the code.

Other tools, like Dafny, need complex encodings in SMT to support the semantics of the language, and in our experience this can be the cause of unpredictable prover performance. We choose to prioritize efficient, straighforward encoding of Rust to Z3 in Verus (without an intermediate language like Boogie). Thanks to Rust's linear typing guarantees we can often encode potentially

mutable, complex objects, as immutable SMT datatypes, rather than having to rely on complex rules and axioms. Even objects with interior mutability that expose an "immutable" interface can often be directly encoded.

Additionally, Rust's memory management story removes a significant challenge of other tools, like Dafny, that need to encode and manage the heap to handle potential aliasing. Heap-reasoning is notoriously costly in SMT-based tools. For example, in our experience building and verifying VeriBetrKV, a verified crash-safe high-performance key-value store, we were able to reduce proof burden (in terms of LOCs and verification time) by up to 30% by integrating a linear type system into a fork of Dafny.

This design principle of prioritizing efficient Z3 queries by relying on safe Rust and the linear type system comes with a necessary compromise: we do not intend to support all Rust features and libraries, instead excluding those that would require complicated encoding. [1] We do, however, plan to provide verification-aided safe alternatives for patterns where "unsafe" would otherwise be required in Rust.

Borrow-checking proofs. As part of recent work, we recognized the value of tracking ownership of ghost variables, which do not appear in the emitted code, but aid verification. For example, in concurrent code, linear ghost variables can facilitate tracking of ghost permissions to access non-atomic memory, represent a thread's knowledge of a concurrent, shared-memory protocol state, or represent limited aliasing domains, for example as a mechanism to encapsulate and prove non-linear data-structures that require aliased references such as `Rc`. To enable using linearity to track these assertions, Verus supports three modes for variables and functions, `#[spec]`, `#[proof]`, and `#[exec]`, as shown in Figure 1 and Figure 2.

Code and variables in `#[spec]` **mode** are always ghost and make up the pure mathematical language used to write specifications. Examples of `spec` are the predicates Ⓐ, Ⓕ, and the expressions in `requires` Ⓒ Ⓗ, `ensures` Ⓓ Ⓘ, and `invariant` clauses Ⓙ. Code in this mode is not checked for linearity and borrow rules, and is always erased before compilation.

Code in `#[proof]` **mode** is ghost code used to define lemmas (e.g. Ⓑ) and prove postconditions (e.g. Ⓔ). To support using linear ghost variables to track permissions and other singly-owned tokens that can aid verification, `proof` code supports linear types and is checked by the borrow checker; it is also erased, like `spec`. Figure 2 shows an example of the borrow-checker preventing Ⓜ an illegal use Ⓝ of a linear ghost permission Ⓚ whose ownership had been transferred Ⓛ.

```
#[exec]
fn increment(
  // a shared memory location
  counter: &PCell<u64>,
  // a linear ghost permission to access it
  #[proof] permission: &mut Permission<u64>) {
  requires(...); ensures(...);

  let cur = counter.borrow(permission); // read
  counter.put(cur + 1, permission);     // write
}

#[exec]
fn release(#[proof] permission: Permission<u64>)
{ ... }

#[exec]
fn start_thread(
  counter: &PCell<u64>,
  #[proof] mut permission: Permission<u64> Ⓚ) {
  requires(...);

  release(permission); Ⓛ

  // error: borrow of moved value: `permission` Ⓜ
  increment(counter, Ⓝ &mut permission);

  assert(...); // that the counter was incremented
}
```

Figure 2: An example of a linear "ghost" permission used to access a shared memory counter. This code is rejected by the borrow checker because `permission` is a linear "ghost" value.

Executable code (`#[exec]` mode) is used for the actual implementation (e.g. Ⓖ), and abides by the same rules as regular Rust code.

Tailored solvers. In our experience verifying large scale system software, we needed the flexibility to write proofs in a variety of ways (e.g. by induction, using ghost variables, using linearity). Verus enables switching the solver's strategy for some proofs, for example by selectively leveraging the SMT solver's bit-vector theory while using the integer theory for the rest of the system. We are also experimenting with support for dedicated solvers (in addition to Z3) to dispatch certain verification conditions for which they are more effective.

Well-engineered foundation. An explicit goal of the project is to build a solid, well-engineered foundation with excellent diagnostics, to aid onboarding of new users, in the spirit of the Rust's compiler error reporting (I will demonstrate this in the demos). A well-engineered foundation should also allow us and others to use Verus as the basis to experiment with new verification language features and techniques.

[1] In the talk, I will highlight which benefits and limitations this may bring in contrast with other Rust verification tools like RustBelt and Prusti, and how these tools may complement each other.