In Verus Veritas: Lessons from Verifying TOCK OS Timer Code

Eric Mugnier[1] and Chengsong Diao[1]

[1]*UC San Diego*

1 Introduction

Timers are hard to get right. They involve an infinite metric–time–, and significant concurrency while relying on limited amount of hardware to keep track of it. The Tock operating system is no exception. As a preemptive embedded operating system, the timer is a core component of the system. It is also one of the parts with the highest number of reported bugs and issues.

To address this, we have started working on the functional verification of the timer. Our main insight is that, while Tock is a complex system, it is also highly modular, written in Rust, with a single threaded execution model, making it a good candidate for verification.

While the primary goal of this project is to verify the timer and prove the absence of previously fixed bugs, this report serves as an experience report highlighting the lessons learned for adapting real system code, originally written without verification in mind, to work effectively with a proving tool like Verus. In particular, we aim to facilitate the integration of verification in the development process. First, we kept the implementation as close as possible to the original code. Second, we strive to integrate proofs and code in a way that maintains their coherence and readability.

2 Timer Architecture

In Tock, the timer functionality is distributed across multiple parts of the codebase. However, the most complex component is the `VirtualAlarm` capsule, responsible for multiplexing applications alarms on the hardware timer.

```
pub trait Alarm<'a> {
    fn get_alarm(&self) -> Self::Ticks;
    fn disarm(&self) -> Result<(), ErrorCode>;
    fn is_armed(&self) -> bool;
}

// The RTC driver implements the Alarm trait
// It represents the hardware timer
pub struct Rtc {
    dt_reference: Ticks32,
}

// The VirtualAlarm implements the Alarm trait
// Each application has one VirtualAlarm
pub struct VirtualAlarm<'a> {
    base: &'a MuxAlarm<'a>,
    dt_reference: Ticks32,
    next: ListLink<'a, VirtualAlarm<'a>>,
}

pub struct MuxAlarm<'a> {
    rtc: &'a Rtc,
    virtual_alarms: List<'a, VirtualAlarm<'a>>,
}
```

In this simplified code snippet, the `MuxAlarm`, is responsible for multiplexing multiple `VirtualAlarm` instances onto the hardware timer `Rtc` with both implementing the `Alarm` trait. To manage all the alarms, Tock uses a linked list in which the alarms are sorted by their expiration time.

This logic has faced several issues in the past, from which we derived our invariants, including:

- for every VirtualMuxAlarm in the linked list, its expiration time must be well-defined and maintain a strict order relative to other VirtualMuxAlarm instances
- Alarms cannot be more precise than MINDELAY [1]
- The number of enabled alarms should reflect the number of alarms waiting to fire [2]

We adopted a step-by-step approach to verifying the timer. First, we reformatted the code to only use Rust features supported by Verus and explicitly declared all the trusted code. Next, we modeled the concept of Ticks along with its different operators. Then, we verified the correctness of the linked list implementation. Currently, we are focused on verifying the main `VirtualAlarm` capsule logic. **As of now, we have successfully verified the invariants for the linked list but none of the VirtualAlarm invariants.**

3 Minimizing implementation changes

One of our primary goal is to integrate the timer verification into the upstream Tock repository. To achieve this, it is essential to minimize deviations from the original codebase as much as possible. Keeping the changes small not only validates the correctness of the original implementation but also facilitates the review process.

However, certain modifications are necessary to enable verification with Verus:

1. Adapting to verification-aware structures, such as converting `Cell` into `PCell`, the verified version of interior mutability.
2. Retrofitting the code to use only Verus-supported Rust features, such as replacing `while` loops and avoiding complex iterators

3.1 Converting Cell into PCell

Standard `Cells` that provide interior mutability are not supported by Verus, but Verus provides `PCells` as a verified alternative.

```
1 let (pcell, mut points_to) = Pcell:new(1);
```

When initializing a `Pcell`, the API returns a `Pcell` structure that contains only a `CellID` structure along with a permission structure `PointsTo` which maps the `CellID` to its associated interior value. The fundamental difficulty lies in the fact that while `PCells` have interior mutability in the execution mode as `Cells`, their corresponding permissions `PointsTos` do not. To access actual values inside `PCells` in specification and proof code, we must interact with `PointsTos` instead of `PCells`. To maintain the immutable borrows for `PCells` as `Cells` in the original code, we need to store `PointsTos` in some separate ghost-state structs, distinct from those containing `PCells`. and pass references of `PointsTos` Additionally, we must pass references to `PointsTos`, with the correct mutability, as arguments to functions as argument to the setter and getter of the `Cell`. It creates several challenges:

- Keeping track of the `PointsTos` separately from the structs containing the corresponding `PCells` is difficult. It requires to store them in complex ghost states that contain all the program's `PointsTo` and is passed everywhere.
- Having to deal with the ownership of `PointsTo` is repetitive and in some ways can defeat the purpose of interior mutability. A solution is to create a setter and getter APIs that leverage this complex ghost state and deal with ownership internally.

4 Code readability

Managing the proof, spec and implementation readability is complex as not only there is more code to maintain, but also the code must be written in a way that is compatible with Verus and its tooling.

Unsupported Rust features Some Rust features and standard library functionalities are not supported by Verus[3]. For example, in our verification attempt for Tock, we found that `Iterator`

is not supported by Verus. It impacted our verification, as iterators are extensively used in Tock especially when going over the linked list. In some cases we were able to reimplement them using while loops, but in other cases we had to refactor the code to avoid using iterators which made it less idiomatic.

Recursive types must have non-recursive variants. The Verus compiler rejects types that have fields containing recursive references without a non-recursive variant, as illustrated in the following code snippet.

```
struct A<'a> {
    value: i32,
    ref: &'a A,
}
```

To pass the compiler checks, the simplest solution is to wrap `ref` in an `Option`, so that the type of `ref` becomes `Option<&'a␣A>`, where `None` is the non-recursive variant. It is not only inelegant because `None` but it also complicates the implementation of the getter function for 'ref'. Without marking implementations as `external`, we cannot use `unwrap` or `unreachable!`, since these contain `panic`s which are not supported by Verus. If we choose to use a placeholder value, then the lifetime checks will fail, whether we create a new instance in the 'None' branch of matching 'ref' or use a static variable.

Different Verus desugaring. To avoid repeating the same specification multiple times, we defined specification function where we check if a structure is well-formed. However, we found that the Verus compiler desugars the code differently and verification can fail when copying the complex postconditions to a specification function. This forced us to temporarily keep those common specification code inlined and repeated across multiple functions which impacted readability. We plan to investigate this issue further to avoid spec duplication.

IDE support. As of now, the IDE support for Verus is limited, especially when compared to tools such as `rust-analyzer`. The developers of `verus-analyzer` also stated that it works fine on small, standalone verification code, but is likely to break for large projects. In our case, it could not load the Tock project. Thus, we had to fall back to `rust-analyzer`, which could not understand the spec and proof code and made the development process more tedious since we had to look manually at the methods and fields of multi-level structures.

5 Results and experiments

5.1 Verus bugs

During verification, we found several bugs in Verus and reported some of them [4]–[6].

For instance, we discovered a situation where the Verus compiler may unexpectedly crash when the logical NOT (`!`) operator is used. To work around this issue, we had to restructure our code by removing logical NOT operators and swapping code blocks between branches.

Another example is the `old(a)` function, meant to refer to the previous value of a mutable reference a in specification and proof code. When passing it directly as an argument to a specification function, e.g. `spec_fn(old(a))`, the Verus compiler will complain that a should be a mutable reference while it indeed is. The underlying issue stems from the fact that spec functions only accept immutable references as function arguments, not mutable ones. As a result, the compiler coerces the type of a to an immutable reference within `old`, due to its `T->T` type signature. To make it work, we need to explicitly write as `&*old(a)`.

5.2 Impact on code size

To assess the effort involved in refactoring and verification, along with the readability of the code, we measured the number of lines in the specifications, proofs, and executable code.

Each file differs significantly in the ratio of specification, proof, and executable code due to variations in complexity and the number of properties verified. The `Virtual Alarm` only contains specification and proof related to the `PCell` and the linked list structure, without verifying any

Table 1. Specifications, Proofs, and Executable Code Lines

File	Specification	Proof	Executable
Linked list	246	174	102
Virtual Alarm	38	32	419
Ticks	40	4	348
Total	331	229	934

functional properties at the moment. For the moment, no functional properties are verified. The `Ticks` file contains multiple verified ticks operations that can be automatically proven by Verus. The `Linked list` is fully verified and required non-trivial proofs that involve universal quantifiers requiring annotations to guide Verus. While the ratio for this file is close from 2:1:1, which is considered good for verified code, it is important to note that even the implementation code has increased in size compared to the non-verified version, which was only 87 lines of code, likely due to the `PCell` struct.

References

[1] *Tock issue 1651*, https://github.com/tock/tock/pull/1651.

[2] *Tock pr 59*, https://github.com/tock/tock/pull/59.

[3] *Supported rust features*, https://verus-lang.github.io/verus/guide/features.html.

[4] *Verus issue 1267*, https://github.com/verus-lang/verus/issues/1267.

[5] *Verus issue 1334*, https://github.com/verus-lang/verus/issues/1334.

[6] *Verus issue 1268*, https://github.com/verus-lang/verus/issues/1268.