

## Introduction

Systems programming is one of the most demanding domains in computer science where bugs and their respective consequences come at a high cost to both service providers and consumers. Languages like C/C++ have dominated the systems programming landscape for years and one of the main problems with both is the lack of memory management. Leaving such responsibility to the developer has proven to be *a less than ideal* solution<sup>1,2</sup>.

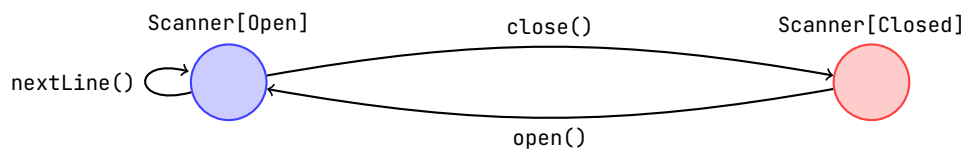
To address such problem, several tools and languages have been and continue to be developed, so far, Rust has been the only one to achieve *mainstream* status. Rust aims to provide memory safety without affecting performance or productivity, to achieve such ambitious goal, Rust validates code with the borrow checker, which then enforces memory safety rules.

**Why Rust?** One of Rust's core values is safety, manifesting itself in the form of memory safety, and the provision of tools to prevent concurrency problems. The *safe* mindset is also imbued in the community, an example would be Sealed Rust<sup>3</sup>, an effort to bring Rust to the safety critical domain.

While Rust addresses memory safety, not all bugs are memory-related. Consider a programmer implementing a protocol, even with the specification's aid the resulting implementation may contain bugs due to human error. Since we can express tpestates as state machines, we can then leverage the latter to materialize the relationships between states and operations, as shown by Figure 1.

Tpestates enable the programmer to track the state of an object throughout its lifetime with help of the type system and while Rust has no direct support for tpestates, its type system is powerful enough to allow them to be implemented within it, effectively blurring the line between specification and implementation by baking the possible state transitions in the type system.

Figure 1: The Scanner tpestate state machine.



**What are tpestates?** In a nutshell, tpestates can be thought as a mechanism to constrain APIs as the program state evolves. More formally, tpestates belong to the behavioral types category and are built on the idea of lifting state to the type level since state becomes part of the type system, the compiler should be able to reason about state, effectively helping the developer track state and validate certain assumptions.

**How are tpestates useful?** Diving deeper on how do tpestates help the developer, we provide a simple yet classic example. Consider a stream, whether it be a file or a socket, to be read, the stream must first be open before being read and finally closed.

Snippet 1: Scanner misuse example.

```

1 Scanner s = new Scanner(System.in); // open the stream
2 s.nextLine();                       // read
3 s.close();                          // close the stream
4 s.nextLine();                       // IllegalStateException
  
```

The code in Snippet 1 tries to read a line after closing the stream, it will crash during runtime with a `IllegalStateException` since you cannot read from a closed stream.

The fact that this code compiles without warnings (even when `-Xlint:all` is used) is problematic, since the error will lead to a crash during runtime (equally, if line 3 was `s = null`, no warnings would issued). While the presented example is simple, production-code is not, and code paths that raise runtime errors may be untested until it reaches the hands of the user<sup>4</sup>.

<sup>1</sup><https://git.io/JLdDc>

<sup>2</sup><https://www.chromium.org/Home/chromium-security/memory-safety>

<sup>3</sup><https://ferrous-systems.com/blog/sealed-rust-the-plan/>

<sup>4</sup><https://github.com/redis/jedis/issues/1747>

Using tpestates solves the above problem by establishing a distinction between the open and closed `Scanner`, consider the code in [Snippet 2](#).

Snippet 2: Tpestated `Scanner` usage example.

```
1 Scanner[Open] s = new Scanner(System.in); // open the stream
2 s.nextLine(); // read
3 Scanner[Closed] s = s.close(); // close the stream
4 s.nextLine(); // compile-time error
```

The compiler is now able to provide the developer with an error at compile-time since it now knows that the `Scanner` is closed and thus, it does not have a function `nextLine`.

**Aliasing Control.** Besides the pro-safety mentality of Rust, another key detail which makes Rust a solid candidate for tpestates is its aliasing control.

By definition, tpestates are incompatible with aliasing due to the fact that if an object is being used by  $N$  clients, when a client mutates an object, all other clients guarantees are broken. Back to the stream example, if a client closes the stream, all other clients may crash since they may try to read from a stream which is now closed.

While Rust is unable to enforce a truly linear type system (in which objects must be used *exactly once*), it is able to enforce an affine type system (or *at most once* usage), as demonstrated in [Snippet 3](#).

Snippet 3: Rust move semantics example.

```
1 let x = 0;
2 let y = x;
3 println!("{}", x); // error, value moved in line 2
```

Such type system allows us to emulate tpestates by *forcing a move* whenever a state transition is required to happen, this way, the previous state is “*destroyed*” and cannot be mutated.

## Typestating Rust

Rust previously had a tpestate system which was removed in version 0.4 since it “did not pull its own weight”. The idea we propose is not aimed at bringing back the old tpestate system but rather at leveraging Rust’s type system to allow for tpestated structures. In short, we propose a DSL built with procedural macros which takes advantage of Rust’s generics and affine type system capabilities.

**The Tpestate Pattern.** As of now, it is possible for a developer to make use of tpestates in Rust the key idea is to write functions which consume the current object and return a new object the function signature should be similar to `fn transition(self: OldState, args...) → NewState`.

Each state can then carry information, or not, being a simple 0-sized structure. States can be grouped into sets using traits, these sets can be further restricted using the sealed trait pattern<sup>5</sup>. An example implementation of the tpestate pattern is available in <https://github.com/rustype/http-parser>.

**Proposal.** The code necessary to implement the tpestate pattern requires a lot of boilerplate furthermore, we want to be able to prove tpestate properties (e.g. reachability and termination). The former challenge can be solved through the use of macros, and the latter is not novel in the literature to address it we just need to extract the state machine from the code and apply existing algorithms.

However, to do so without writing a static checker from the ground up we are required to use Rust’s procedural function-like macros, which enable the creation of a DSL for tpestates, thus solving both the boilerplate problem and the requirement for the ability to prove certain properties.

Existing work on the Rust ecosystem regarding tpestate-like mechanisms is limited, we have reviewed several automata crates and only one provided strong compile-time guarantees, such crate was the `state_machine_future` crate, however it requires a runtime such as Tokio due to being centered around asynchronous computations.

Our approach aims to extend the ideas behind `state_machine_future` with an expressive DSL and the possibility to be used without a runtime in a purely static approach. Currently, a prototype is under development and can be found in <https://github.com/rustype/tpestate-rs>.

---

<sup>5</sup><https://git.io/JLbry>