Retrofitting Typestates into Rust

José Duarte
jmg.duarte@campus.fct.unl.pt
NOVA School of Science and Technology
NOVA LINCS
Portugal, Lisbon

António Ravara
aravara@fct.unl.pt
NOVA School of Science and Technology
NOVA LINCS
Portugal, Lisbon

ABSTRACT

As software permeates our lives, bugs become increasingly expensive; the best way to reduce their cost is to reduce the number of bugs. Of course, this is easier said than done and, at best, we can go after their root causes to mitigate them. One of such causes is *state*, whether it is the state of a light bulb (i.e. on/off), or the state of a complex protocol, reasoning about state is a complex process which developers are required to do with subpar tools.

Ideally, we want to specify constraints and have the computer reason for us; typestates enable developers to describe states using the type system and allow the compiler to reason about them.

We propose an approach to bring typestates to Rust, without any external tools, leveraging only Rust's type and macro systems. Our approach provides a macro-based domain-specific language which enables developers to easily express and implement typestates, along with certain state machine safety guarantees, it is open-source and available at https://github.com/rustype/typestate-rs.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Software libraries and repositories**; **Preprocessors**.

KEYWORDS

Behavioral Typing, Domain-specific Languages, Macros, Typestates, Rust, Session Types, Protocol Compliance

ACM Reference Format:
José Duarte and António Ravara. 2021. Retrofitting Typestates into Rust. In *25th Brazilian Symposium on Programming Languages (SBLP'21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3475061.3475082

1 INTRODUCTION

Context. Both as users and developers, we are accustomed to bugs in the software we use; most of these will be annoying, degrading our experience as users; however, in critical software, bugs are much more impactful, possibly putting the user's life at risk.

In 2019 and 2020, the Boeing 737 Max was grounded after several crashes having been attributed to faulty software[1,2,3]; more recently, as COVID-19 cases grew and contact tracing apps were used to mitigate the virus' impact, the UK's National Health Service app failed to ask users to self-isolate due to a software bug[4].

These incidents are not isolated; as software becomes more prevalent in our lives, the cost of bugs will inevitably rise. While the language and nature of bugs differ on a case-by-case basis, there is no silver bullet and our best alternative is to mitigate them — building tools and abstractions that enable developers to increase safety guarantees.

Systems Programming. Unsafe languages like C and C++ have dominated the systems programming landscape for years, their biggest strength is also their biggest weakness, the amount of control offered to the developer. As the popular phrase states — "*with great power, comes great responsibility*"; in the case of these unsafe languages, this means that developers get low-level control over their systems, even when not required, leading to bugs that may crash or leave systems vulnerable to malicious attacks.

Being built with safety in mind, Rust addresses memory-related problems through its ownership system and borrow checker; allowing the developer to be confident that no memory related errors will arise. However, this is not enough, bugs can also be found in business logic, protocols and others (e.g. off-by-one errors); cases outside the borrow checker's scope.

Typestates, in their most basic form, are finite state machines described at the type-level [11]. They belong to behavioral types, a flow-centric approach to static typing [2, 5] and aim to tame stateful computations. Typestates enable the compiler to reason about state and by extension, function call ordering.

Consider the light bulb, as a simple state machine; it can either be on or off, and trying to turn it off twice produces no effect. By design, light switches forbid us of turning it off twice; when writing code, the compiler is the light switch, stopping developers from calling functions at "*the wrong time*".

Just like the light switch provides us confidence that no "*incorrect*" behavior reaches the bulb, a typestate-enabled compiler provides developers with ability to restrict certain uses of their API's, alleviating developers from the responsibility of bookkeeping their systems' runtime state during development.

The approach presented in this paper aims to provide a bridge between typestates and modern Rust, allowing developers to harness the power of typestates and develop software with guarantees beyond memory safety.

Contributions. The main contribution of this paper consists on a novel approach to typestates in Rust, which exploits the Rust type system for typestate checking and relies on its advanced macro system to generate the necessary boilerplate. Our macro:

[1]https://tinyurl.com/Okane2019
[2]https://tinyurl.com/Okane2020
[3]https://tinyurl.com/DCampbell2020

[4]https://tinyurl.com/Mageit2020

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SBLP'21, September 27-October 1, 2021, Joinville, Brazil
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9062-0/21/09.
https://doi.org/10.1145/3475061.3475082

- *Enables the developer to describe typestates in pure Rust.*
- *Provides state machine related guarantees.*
- *Emits helpful error messages regarding the guarantees.*

This work falls in the "*correct by construction*" approach to software; it allows Rust developers to harness the power of typestates, reduce the number of possible bugs and cut down the time spent debugging by side-stepping state related bugs.

Structure. We start by demonstrating how one can write typestates in pure Rust "*by hand*" and discuss such approach flaws in Section 2. In Section 3 and Section 4 we give a tour into our tool in a top-down fashion, from the architecture and syntax down to the code generation process. In Section 5 we demonstrate our tool in action through an implemented case study. Section 6 discusses related work.

2 TYPESTATES, THE HARD WAY

Rust is no stranger to typestates, before the 0.4 release, Rust had contract-style typestates; currently, *The Rust Embedded Book* provides an introduction to typestates in Rust[5]. Along with several blogs on the subject[6,7], one can safely say that Rust is in fact able to reason about state at the type level.

2.1 Ingredients

Aliasing control. One of the main obstacles between typestates and "mainstream" languages is the lack of aliasing control, which are required to provide state coherence when sharing objects [1]. Consider a multi-threaded environment where a given object is in a certain state and is shared among several threads; if one of the threads modifies the state of the object, all other threads will have an invalid view of the object state. Rust's ownership system solves this problem by providing strict aliasing control of objects.

Consider the simple example where a thread is spawned to print all the vector's elements:

```rust
fn main() {
  let v = vec![1, 2, 3];
  let handle = std::thread::spawn(|| {
    println!("Here's a vector: {:?}", v);
  });
  handle.join().unwrap();
}
```

The previous code does not compile since the thread's closure may outlive the `main` function (owner of `v`), since the thread borrows `v`, it could happen while the thread is running, `v`'s owner dies and invalidates `v`. Rust compiler not only provides the previous explanation, it also provides a solution, which is to prepend the closure with `move`, forcing the thread to take ownership of `v`.

Meta-programming mechanism. Rust provides programmers with the capability to express typestates, by taking advantage of its meta-programming system (i.e. macros) we can reduce errors through automation and leverage information embedded in the code. In Rust, macros come under two flavors, declarative and procedural macros, we focus on the more powerful procedural macros, which allow for arbitrary code to be run, thus granting us the possibility of extracting information from the code and process it all before compilation. Languages providing equally powerful meta-programming systems may be able to retrofit typestates into them, following this approach, provided the rest of the requirements are met.

2.2 The manual approach

We now describe how the manual approach works — consider a light bulb which has two states: `On` and `Off`. Our bulb's typestate requirements are simple: methods should not be called out-of-order and state extension should be forbidden to downstream users. Modelling its typestate in Rust yields[8]:

```rust
// The light bulb structure
struct LightBulb<State> { state: PhantomData<State> }
// Possible states
struct On;
struct Off;
```

We now have a light bulb, however, it does nothing yet, and while it can be constructed "*by hand*", we will also write a constructor[9].

```rust
// Functions available to all states
impl<State> LightBulb<State> {
  fn new() -> LightBulb<Off> {
    LightBulb::<Off> { state: PhantomData }
  }
}
```

The attentive reader might notice that it makes no sense for the constructor function, which returns a concrete state, to be implemented using a generic `impl`. A better alternative would enforce the type calling `new` to be of the same type as the returned value and take advantage of `Self`.

```rust
impl LightBulb<Off> {
  fn new() -> Self {
    Self { state: PhantomData }
  }
}
```

This constructor creates a new light bulb, starting in the `Off` state. We now need to make the bulb transition between states; to do so, we define functions just like our constructor — using a concrete `impl`.

```rust
// Functions available in the On state
impl LightBulb<On> {
  fn turn_off(self) -> LightBulb<Off> {
    LightBulb::<Off> { state: PhantomData }
  }
}
// Functions available in the Off state
impl LightBulb<Off> {
  fn turn_on(self) -> LightBulb<On> {
    LightBulb::<On> { state: PhantomData }
  }
}
```

Finally, we can start using our bulb. We show an example of a well-behaved bulb usage:

[5]https://docs.rust-embedded.org/book/static-guarantees/typestate-programming.html
[6]https://yoric.github.io/post/rust-typestate/
[7]http://cliffle.com/blog/rust-typestate/

[8]`PhantomData` is a type which pretends that owns a `T`. See https://doc.rust-lang.org/std/marker/struct.PhantomData.html. When discussing typestates, the Rust community will usually use `PhantomData` for *field-less* states, being the most common approach and even the suggested one in *The Rust Embedded Book.*
[9]In Rust, one defines the implementation of a structure inside `impl` blocks. See https://doc.rust-lang.org/std/keyword.impl.html for a more detailed description.

```rust
fn main() {
  let bulb = LightBulb::<Off>::new();
  let bulb = bulb.turn_on();
  let _ = bulb.turn_off();
}
```

If instead of using the bulb as normal, turning it on and off, we try and turn it on twice, the Rust compiler will issue an error and let us know that `turn_on` is not available in the `On` state.

```
let _ = bulb.turn_off();
|             ^^^^^^^^ method not found in
|             `light_bulb::LightBulb<light_bulb::Off>`
```

2.3 Restricting possible states

One detail the reader might have noticed is that this API can be freely extended by the client, that is, nothing stops the API client from adding another state and extending the bulb's functionality in unpredictable ways.

We are required to implement the *sealed trait pattern*[10,11], which instead of requiring types to implement a single trait, requires types to implement two traits: a private one, unexposed to the client and a public one which requires the private one to be implemented. This approach effectively disables the client from implementing the public trait since it requires a private one to which the user does not have access.

```rust
mod private {
  use super::{On, Off};
  pub trait Private {}
  impl Private for On {}
  impl Private for Off {}
}
trait State: private::Private {}
impl State for On {}
impl State for Off {}
struct LightBulb<S: State> { state: PhantomData<S> }
```

2.4 Approach Flaws

This approach has two major disadvantages — its verbosity and lack of guarantees regarding the typestate's state machine.

Verbosity. Writing a small state machine requires a lot of attention to detail, the developer cannot forget the sealed traits and their implementation for every state, along with implementing each transition to the right state and possible extra functionality.

Guarantees. Unless the user checks the state machine properties (e.g. if all states are productive) ahead of time by designing the state machine in a tool built for such purpose, this approach is unable to provide any kind of guarantees regarding the typestate's state machine. This adds overhead to development, since one is required to design *and* verify the state machine before writing any code.

3 DESIGNING A PURE RUST TYPESTATE DSL

Our decision to design a DSL is inspired by Fugue [3], which allows to specify finite automata whilst providing some of the usual guarantees. Its design is based around annotations from which

[10]Traits define abstract interfaces — for a more detailed description of traits in Rust, please see https://doc.rust-lang.org/book/ch10-02-traits.html, https://doc.rust-lang.org/book/ch19-03-advanced-traits.html or https://doc.rust-lang.org/reference/items/traits.html.

[11]https://rust-lang.github.io/api-guidelines/future-proofing.html

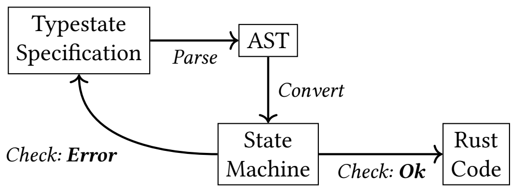

Figure 1: From DSL specification to Rust code.

properties are inferred and then checked against. Before diving into the implementation details, we start by discussing our macro choice, followed by an overview over the DSL architecture.

3.1 Requirements & Macros

We review our DSL's requirements and explain the reasoning behind our macro pick:

Clear & concise The DSL should allow the user to describe the typestate in a clear and concise manner.

Useful error messages Pinpointing the offending item and the reasoning behind the offense.

Simple The DSL's syntax should be as close to Rust as possible, avoiding requiring the developer to learn a new language.

Attribute macros. Attachable to most items, attribute macros allow for the replacement of the input; effectively rewriting the input source code. As any item in Rust can only be valid Rust syntax, its input is strictly valid Rust. Taking this into account we can attach this macro to a module. Since code inside a module is treated similarly to code written in the top-level of file, we have most of Rust's syntax to our disposal to manipulate, effectively allowing a DSL to be developed using *only* pure Rust, and used in a limited scope (the module's scope).

3.2 Architecture

The architecture of our macro is illustrated in Figure 1; the process starts with the typestate specification, which is annotated Rust code. As macros are effectively arbitrary code ran by the compiler, we are able to process the input token stream and extract a state machine from the specification. In the case the extracted state machine passes all checks, the final Rust code is successfully expanded and the user is able to continue development.

3.3 Syntax & State Machine Extraction

We describe the DSL's syntax and how it relates to the state machine, starting with the simpler and more obvious aspects of the DSL.

A quick primer on the DSL's main elements is presented in Figure 2; the primer covers — the DSL's entry point, the declaration of the automaton, its states and transitions, and the declaration of initial and final states.

To demonstrate the DSL we start by rebuilding the previous light bulb example (Figure 3) and then extend upon it to demonstrate the advanced features (Subsection 3.4).

Figure 3 declares the `LightBulb` structure as the automaton/-*typestated* structure through the usage of the `automata` attribute (line 2); the automaton's possible states are then declared using Rust's structures and the `state` attribute — `On` & `Off` (lines 3-4); finally, their functions are declared using the `trait` keyword followed by their respective names; each of the presented functions

```
 1 // The entry point to the DSL
 2 #[typestate]
 3 mod typestate_dsl {
 4   // Only one automaton per typestate specification
 5   #[automata] struct Automaton;
 6   // N-states are possible
 7   #[state] struct StateA;
 8   #[state] struct StateB;
 9   // Functions are defined inside traits
10   // Traits share their name with an existing structure
11   trait StateA {
12     // Transitions are functions that consume `self`
13     // and return an existing state
14     fn transition(self) -> StateB;
15     // Functions can declare states as initial/final
16     // Initial state declarations do not take `self`
17     fn new() -> StateA;
18     // Final state declarations consume `self` and
19     // do not return an existing state
20     fn end(self);
21   }
22 }
```

Figure 2: The main elements for the #[typestate] DSL.

```
 1 #[typestate] mod light_bulb {
 2   #[automata] struct LightBulb;
 3   #[state] struct On;
 4   #[state] struct Off;
 5   trait On { fn turn_off(self) -> Off; }
 6   trait Off {
 7     fn screw() -> Off;      // initial state declaration
 8     fn unscrew(self);       // final state declaration
 9     fn turn_on(self) -> On; // Off => On transition
10   }
11 }
```

Figure 3: The light bulb's typestate specification.

either declares a transition (lines 5 and 9) or the current state as being either initial or final (lines 7 and 8).

Transitions, as well as the initial and final states are inferred from the declared functions, the rules are as follows:

- *Transitions* are inferred from functions that consume `self` and return a state (Figure 2, lines 12-14).
- *Initial states* are inferred from functions that *do not* consume `self` and return a state (Figure 2, lines 15-17).
- *Final states* are inferred from functions that consume `self` and *do not* return a state (Figure 2, lines 18-20)

During the specification process, several errors may appear:

- Missing automata declaration.
- Missing initial and/or final states.
- A given state is not productive or useful.

3.4 Advanced Features

We now present the more advanced features; consider now that we wish to model a smart light bulb. The main difference to our previous bulb is the ability to change color.

One way of modelling builds on the previous bulb example. We add the color variable to `LightBulb`, as it needs to "*remember*" the last color the user set. This makes the variable available in all states, as it is part of the automaton and not a specific state.

```
#[automata] struct SmartBulb { rgb_color: (u8, u8, u8) }
```

However, while it is available in all states, it should not be used in every state, thus we define a "*getter*" and a "*setter*" for it, only in the On state.

```
trait On {
  fn turn_off(self) -> Off;
  fn get_color(&self) -> (u8, u8, u8);
  fn set_color(&mut self, color: (u8, u8, u8));
}
```

Notice how the new `color` functions make use of references instead of taking ownership; allowing the state to be read and modified, respectively, without performing an actual state transition.

Finally, consider that our smart light bulb is "not that smart" and sometimes when we try to change the color, it can fail and turn itself off. This bifurcates the state transition to either `On` or `Off`. To model this behavior we need to replace the definition of `set_color` with the following:

```
fn set_color(self, color: (u8, u8, u8)) -> Unknown;
```

Notice that we now consume the state, and we return Unknown[12] instead of a concrete state. The compiler will throw an error since Unknown is undefined, so let's define it.

```
enum Unknown { On, Off }
```

We use the enumeration to represent the non-deterministic nature of the transition [12], as well as force the user to match against both cases. This enumeration has a few rules; all variants must be of `Unit` type[13], and they must also share their identifier with an existing state.

Loops & Recursion. Our work supports both recursion and loops; to disambiguate, consider, once more, the light bulb example: one can start in the `Off` state, transition to the `On` state and back to the `Off` state. This process can occur indefinitely and scales to arbitrarily large state chains.

```
let bulb = LightBulb::screw();
let bulb = bulb.turn_on(); let bulb = bulb.turn_off();
let bulb = bulb.turn_on(); let bulb = bulb.turn_off();
// this pattern might be very long
```

Loops, do not have such a simple solution, as Rust's type system does not allow a variable to change type; for each iteration of the loop we need to keep the same variable type.

```
let bulb: LightBulb<Off> = LightBulb::screw();
loop {
  // since `turn_on` returns `LightBulb<On>`
  // this will not type check
  bulb = bulb.turn_on();
}
```

This constraint makes our state machines very limited; and so, a standard solution is to declare an enumeration containing all the possible states. Given that the type is always the same (i.e. the enumeration's type), the type checker will be happy, allowing us to replace the variable in each iteration and when needed, match against it to take its inner state. Another example can be seen in Figure 7 and Figure 8.

```
enum Bulb {
  Off(LightBulb<Off>),
  On(LightBulb<On>)
}
```

[12]The identifier Unknown is not special and arbitrary names can be used.
[13]`Unit` type variants do not contain information other than their name.

```
let bulb: Bulb = LightBulb::screw().into();
loop {
  bulb = match bulb {
    Bulb::Off(b) => { b.turn_on().into() }
    Bulb::On(b) => { b.turn_off().into() }
  }
}
```

3.5 Utilities

Along with the already present features, our work also provides extra features which aim to make developer's lives easier.

3.5.1 State constructors. This feature generates default constructors for each state with variables; the name of the constructors is customizable. To use this feature the developer can simply pass an argument to our macro's entrypoint.

```
#[typestate(state_constructors)]         // default name
#[typestate(state_constructors = "new")] // set to "new"
```

Generated for state structures that have fields, the default identifier for the constructor function is `new_state`, if a value is attributed to the `state_constructors` macro argument, that is used instead. The function's parameters are named after each field, by the order they were written. For the following structure:

```
struct S { f1: u8, f2: Vec<String> }
```

The following constructor would be generated:

```
impl S {
  fn new_state(f1: u8, f2: Vec<String>) -> Self {
    Self { f1, f2 }
  }
}
```

3.5.2 State enumeration. This feature generates an enumeration containing all states, such is useful when dealing with loops or data structures, where one cannot use multiple types; using an enumeration allows us to "*unify*" all states under one type. Similarly to the previous feature, the developer passes an argument to our macro's entrypoint, in this case, the argument represents a prefix rather than an identifier replacement.

```
#[typestate(enumerate)]          // default prefix
#[typestate(enumerate = "Enum")] // set to "Enum"
```

Generated from automaton the states, its default naming behavior is to prepend an `E` to the automaton identifier; so an automaton `Automaton` would generate an enumeration `EAutomaton`. In the case the user attributes a value to `enumerate`, that is used instead. Along with the enumeration, `From` traits are implemented for each type to ease conversion from the typestate to the enumeration.

Revisiting our `LightBulb` example (Figure 3), with states `On` and `Off`, the following enumeration would be generated:

```
enum ELightBulb {On(LightBulb<On>), Off(LightBulb<Off>)}
```

3.5.3 Typestate Visualization. While the previous features aim to make writing code easier, this one aims to allow for state machine inspection; it takes the form of a library feature[14] instead of a macro argument, allowing users to easily toggle it on and off.

This feature comes under two flavors, exporting DOT files or PlantUML state diagrams; the files can then be rendered into any

[14]Library features are declared in the "*dependency manifest*" and can be toggled on and off by library clients.

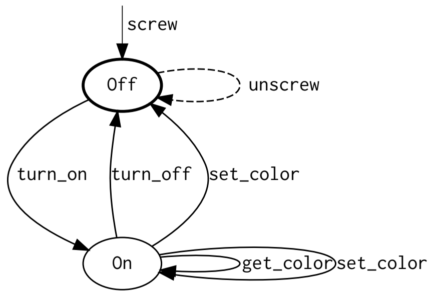

Figure 4: The smart light bulb example rendered from the exported DOT file.

format supported by the respective tool. A DOT diagram is illustrated in Figure 4, and PlantUML diagrams are illustrated in Figure 6 and Figure 10.

When using DOT, the rendered representation is different to the automata representation.

- Transitions declaring initial states are represented as labeled directed edges without an origin node (e.g. `screw`).
- Transitions declaring final states are represented as labeled loops with a dashed lined (e.g. `unscrew`).
- Final states have a heavier stroke (e.g. `Off`).
- Functions using references, as described in Subsection 3.4, are represented as labeled loops (e.g. `get_color`).

Our PlantUML representation resembles *Deterministic Object Automata* [12], using diamonds to represent the possible outcomes of an operation.

4 PEEKING BEHIND THE VEIL

As previously stated, our objective is to automate the boilerplate and provide extra safety guarantees. In this section we describe the approach taken as to achieve our goals.

4.1 Parsing & code generation

Our macro heavily relies on `syn`[15], a parsing library for Rust's `TokenStream`[16]; the crate (i.e. library) provides several utilities such as item parsers and visitor traits[17,18].

Parsing is divided into three visitors, each of which can be seen as "*compilation phases*"; these visitors implement the `syn::VisitMut` trait, allowing us to *mutate* the AST as we process it, sparing the work of re-generating the whole tree with modifications.

4.1.1 State visitor. The first pass is performed by the state visitor, it visits all structures checking for the ones annotated with either `#[automata]` or `#[state]`.

The `#[automata]` annotated structure[19] is rewritten adding a generic `State` type parameter which is bounded by the sealed trait. Along the type parameter, a new field is added to the structure fields; this field has type `State` instead of `PhantomData`, enabling each state to carry data.

[15]https://docs.rs/syn/1.0.72/syn/
[16]https://doc.rust-lang.org/stable/proc_macro/struct.TokenStream.html
[17]https://docs.rs/syn/1.0.72/syn/visit/index.html
[18]https://docs.rs/syn/1.0.72/syn/visit_mut/index.html
[19]For each module annotated with `#[typestate]`, only one `#[automata]` annotated structure can be defined; otherwise an error is issued.

Structures annotated with `state` are not rewritten, but their identifiers are stored as the next passes are dependent on them.

4.1.2 Non-deterministic transition visitor. This stage checks existing enumerations; it ensures that each field is of type `Unit` and a valid state (i.e. a declared state), if these conditions are met it rewrites the variant to be of the `Unnamed` type[20], containing the type of the automaton along with the respective state (e.g. `Automaton<State>`).

4.1.3 Transition visitor. Finally, the transition visitor is responsible for visiting each trait, ensuring its identifier matches a valid state and extracting transition information from each function (described in Subsection 3.3 and Subsection 3.4).

A suffix is added to the trait's name as to avoid name clashes since structures and traits cannot share the same name in Rust.

Along with transition extraction, the `#[must_use]` annotation is added to functions that represent regular transitions (i.e. consume `self` and return a valid state), while the annotation alone does not enforce usage, it provides a warning to the user, which in turn represents that the protocol has not been completed. The extracted information is used to generate a graph representing the automata, see Subsection 4.2 for a detailed explanation of the graph verification.

4.1.4 Code generation. Along with the described tree mutations, the macro also performs code generation. The generated code is composed of helper code (constructors and enumerations, respectively described in subsubsection 3.5.1 and subsubsection 3.5.2) and the sealed pattern, described in Subsection 2.3.

Sealed Pattern. Implemented according to *Rust's API Guidelines*[21]. Once more, using the `LightBulb` example (Figure 3), the generated[22] code would be as follows:

```
mod private { pub trait LightBulbState {} }
pub trait LightBulbState: private::LightBulbState {}
impl private::LightBulbState for On {}
impl private::LightBulbState for Off {}
```

The private trait implementations are provided outside the module to cope with the fact that we do not know which states will be sealed when generating the `private` module.

4.2 Typestate, automaton & graph verification

Our tool extracts information from the user's code, which is then represented as a directed graph; this graph closely resembles the one illustrated in Figure 4. The graph structure has been adapted from the FAdo project [10], the adaptation adds extra fields and ports the original Python code to Rust.

The first check to be done is the presence of declared initial and final states, since without either of them, productiveness and usefulness checks will always fail; in case such states are not present, compilation errors are issued.

Afterwards, states are checked for productiveness that is, if all states have a path to the final state. This is implemented as a breadth-first search from all final states, as states not visited (i.e. reachable) when starting from a final state, will not be productive. Similarly, useful states will be productive states that are reachable from an initial state, and so it is also implemented as a breadth-first search starting from the set of initial states. The resulting set of states is then intersected with the set of productive states.

Any state found to not be productive or useful will trigger a compilation error, pointing towards that state. Correcting such error is left to the developer, as allowing the macro to delete them would not provide a transparent development experience. Similarly, minimization is not made for the same reasons.

Errors. Along with verification, comes the necessity of communicating to the user *why* its typestates are invalid (e.g. lacking an initial state); Rust's macro system allows us to provide custom errors to the user and point them to the most relevant token(s). As an example, if a structure is *non-useful*, its identifier is underlined and the error message issued by the compiler (Figure 5).

```
error: Non-useful state. For a state to be useful
it must first be productive and a path from initial
state to the state is required to exist.
--> examples/light_bulb.rs:15:21
  |
15 |     #[state] struct On;
  |                     ^^
```

Figure 5: Macro error example.

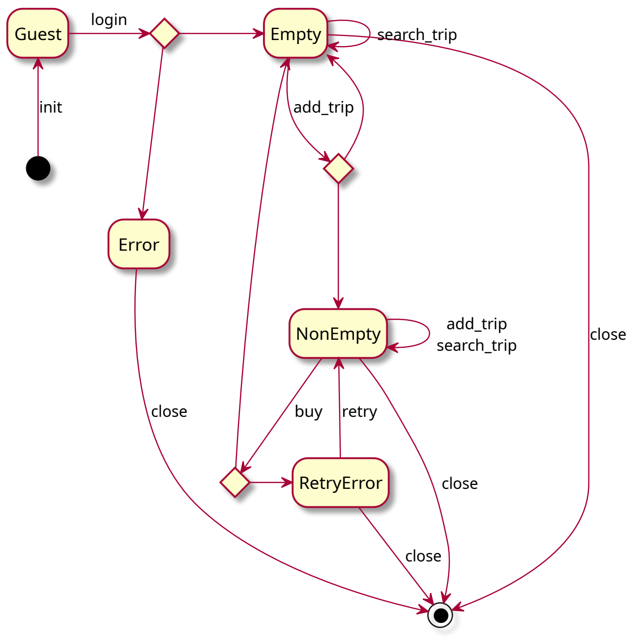

Figure 6: The travel agency's `Session` typestate.

5 CASE STUDY: TRAVEL AGENCY

To demonstrate `#[typestate]` in action we have developed a case study in which we typestate two protocols with communicating parties. Our example is based on the classic travel agency example (Figure 6) where a customer buys a ticket from a travel agency, and the agency sends a transaction to a bank to be processed.

[20] Along with their name, Unnamed type variants contain a single type, without a field identifier (e.g. `EnumVariant(UnnamedFieldType)`).
[21] https://rust-lang.github.io/api-guidelines/future-proofing.html
[22] The shown code omits some implementation details.

[22] The full code for this example — https://github.com/rustype/travel-agency

```
1  fn main() -> Result<()> {
2    let mut input_buffer = String::new();
3    // our session enumeration
4    let mut session = TSession::new(); // Figure 3
5    loop {
6      // show prompt and read command
7      prompt(&mut input_buffer, &session)?;
8      // parse command
9      let split_input: Vec<_> =
10       input_buffer.trim().split(" ").collect();
11     if let Some(&cmd) = split_input.first() {
12       // command matching
13       session = match session { /* Listing 5 */ }
14     }
15   }
16   return Ok(());
17 }
```

Figure 7: The travel agency's client `main` function.

```
1  session = match session {
2    TSession::Empty(mut s) => match cmd {
3      SEARCH => {
4        if split_input.len() != 2 {
5          println!("invalid search command.");
6        } else {
7          let trips = s.search_trip(split_input[1]);
8          for (i, trip) in trips.iter().enumerate() {
9            println!("{}: {:?}", i, trip);
10         }
11       }; s.into() // return the next state
12     },
13     // the other commands
14 }
```

Figure 8: The CLI matching procedure.

The client has at its disposal an interactive command line application, which uses the agency's API; before the client is able to do anything, it is required to log in; afterwards the client is free to browse the agency's database and add items to the cart, the client is unable to perform purchases before adding an item to the cart.

In Figure 7, notice how the session is marked as mutable and is an enumeration; given that the application runs in a loop, and we cannot change a variable's type after declaration, we are required to use Rust's `enum`s to unify the automata's types (line 4).

In Figure 8, the match handling logic first handles each state (line 1) and then possible commands for that state (line 2). Each `match` arm returns the next session state, we take advantage of the generated `From` trait to simply call `into` and perform the conversion from the state to the enumeration.

Diving deeper into the agency's typestate we can see how transitions are performed. Figure 9 demonstrates how the `add_trip` operation is implemented. This operation can fail since the added trip may not be valid, in such case we use the non-deterministic state transition enumeration and return the possible "*intermediate*" states (i.e. `Selection`'s `NonEmpty` and `Empty`).

On the bank's side, where the agency acts as its client, the typestate represents a single transaction; the first step is to validate the information passed by the agency, this operation can fail, and thus it uses our enumeration approach; in case the information is valid, the transaction is attempted, failing in case the client does not have enough funds; in both cases, if the operation fails, a transition to the

```
impl EmptyState for Session<Empty> {
  fn add_trip(self, idx: usize) -> Selection {
    println!("{:?}", self.state.last_search);
    if idx < self.state.last_search.len() {
      Selection::NonEmpty(Session::<NonEmpty> {
        state: NonEmpty {
          selected:
            vec![self.state.last_search[idx].clone()],
          last_search: self.state.last_search,
        }, })
    } else {
      Selection::Empty(self)
    }
  }
}
```

Figure 9: The `EmptyState` implementation, showcasing the usage of an enumeration as return value.

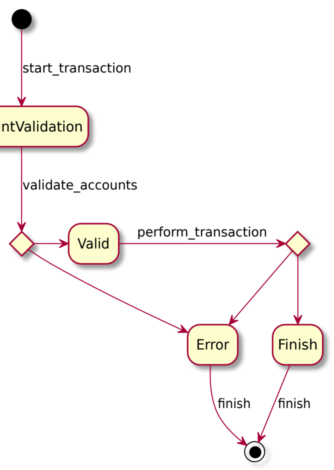

Figure 10: The bank's `Transaction` typestate.

`Error` state occurs where the API closes the transaction by calling `finish`. The transaction's typestate is illustrated in Figure 10.

6 RELATED WORK

The `session-types` *crate [6],* the first to bring session types [4] into Rust, provides an interpretation of session types and a base for the work which followed.

The library provides the required abstractions for session types, but these are built on top of `unsafe` blocks, which may deter some users; furthermore, having long `Offer`/`Choice` (i.e. session type constructs) chains becomes normal, leading to complex and verbose types when the compiler reports errors. The crate offers abstractions to deal with the previous problems that work when programming "linearly", but fail to work when using loops.

The `sesh` *crate [7],* in contrast to `session-types`, builds on a different theory, provides much cleaner types and embraces Rust's affine type system, instead of actively trying to make it linear. While the crate requires no use of `unsafe`, it does require a *nightly* compiler, unsuited for production environments.

Like `session-types`, the `sesh` crate provides the required abstractions to use session types in Rust, however, the crate's documentation is limited, possibly being "inaccessible" to less experienced users w.r.t. Rust and session types.

The Plaid language, part of the *typestate-oriented* paradigm, is an extension of the object paradigm [1]. In typestate-oriented programming objects are modeled in terms of classes and changing states, much like our Rust-based approach.

Besides the new paradigm, one of the distinguishing features of Plaid is its permission-based type system; this system describes how an object is shared. Plaid has three keywords to control an object's aliasing.

- The `unique` indicates that an object does not have aliases, similar to Rust's mutability constraint, which only allows the object's owner to mutate it;
- The `immutable` allows for unlimited aliasing, but, as the name dictates, it does not allow any mutation, being similar to Rust's immutable references;
- The `shared` keyword allows mutation through the existing alias, in contrast with the previous ones. For example, an object in Java can be mutated through all aliases, at any time, by anyone.

The Mungo project [8, 13], comprises several pieces which work towards a common and bigger goal, allowing developers to make use of typestates in languages such as Java.

The toolchain consists of a Java annotation and a typestate description language along with its checker. The compilation process starts by typechecking the code according to regular Java and then using the Mungo toolchain, which reads the typestate protocol declared by the annotation (`@Typestate("ProtocolName")`), extracts the method call behavior and checks the extracted information against the typestate [13].

The Mungo toolchain is available online[23] and a newer version, named JATYC is available on GitHub[24] [9].

"Fugue is a software checker that allows interface protocols to be specified as annotations in a library's source code or in Fugue's specification repository" [3]. Similar to our work, Fugue performs static checks to produce a list of errors and warnings, without requiring any intervention during runtime.

Fugue allows the developer to specify two types of protocols, *resource* and *state machine* protocols; resource protocols concern the allocation and release of resources, in Rust this is done by the ownership system; state machine protocols constrain the order in which an object's method can be called: "*Given a class with a state machine protocol, Fugue guarantees that, for all paths in every analyzed method, the string of method calls made on an instance of that class is in the language that the finite state machine accepts. This is called the method order guarantee*" [3]. In our work, method call ordering is enforced through the type system.

Going further, Fugue also allows developers to relate states from different state machines. Consider state machines A and B, by relating certain states from A to B, Fugue can ensure A is a well-behaved client of B.

Along with method call ordering, Fugue also provides more complex state machine protocols through the use of *custom state*, enabling an object's state to be modelled through another object; as well as domain-specific checks through predicates. Both features assign methods to pre and post conditions, these methods are then invoked during checking to perform state checks and transitions.

[23]http://www.dcs.gla.ac.uk/research/mungo/
[24]https://github.com/jdmota/java-typestate-checker

In summary, while all the presented projects present positive results they also present severe deficiencies. The Plaid language has been abandoned and the authors moved on to other projects, furthermore, the produced typestates are scattered around the code and mentally visualizing the state machine from the code quickly becomes unfeasible; our approach keeps states and transitions close together, furthermore we are able to export the specified typestate allowing the user to visualize the state machine.

One of Mungo's strengths is also one of its weaknesses, while being an external tool provides flexibility it comes at the cost of complicating the compilation process; its feature set and imposition of linearity over the user also complicate the development of more complex systems. Our approach is embedded in its target language, reducing usage complexity; however, like Mungo, we do not support subtyping and generics.

Regarding Fugue, while its approach using annotations is clever and elegant, it leaves much to be desired regarding code readability. Fugue is unavailable (at least for the public). Our approach avoids this problem by using language constructs to cleanly express the typestate, instead of relying solely on annotations.

7 CONCLUSION

While Rust provides several improvements over the previous state of the art regarding memory safety, that is not enough to avoid runtime errors and thus tools addressing higher-level concerns are equally necessary to improve software quality.

Our work explores how we can use Rust's type and macro systems to provide safer APIs through DSLs. We provide a DSL which reduces boilerplate when dealing with Rust's typestates, as well as bring extra guarantees to the table, which previously, were only possible if the state machine was processed by an external tool.

Still, this kind of tool is partially limited by the ecosystem; for instance, `rust-analyzer` partially flags valid code as errors (as of writing this paper), which is annoying for a developer using our tool. Until these problems are addressed, the tool's adoption can be impacted due to factors outside our control.

This paper has been written using `typestate`'s version 0.6 (as of 14/07/2021 the most recent version is 0.8) along with `rustc` version 1.51 and 1.53 nightly.

Future Work. The issues to address are: (i) evaluate the present work based on a larger, real-life use case and implement improvements taking into account the evaluation results; (ii) include generics and subtyping, both in automata and states; (iii) formally define the checking process, state and prove the key properties envisaged.

Acknowledgements. We thank Bernardo Toninho, Daniel Henry-Mantilla, João Mota, Mathias Jakobsen, Ornela Dardha, Philip Munksgaard, Simon Fowler, Tomás Alagoa and Wen Kokke for their comments on previous versions of this paper, and the reviewers'. This work is partially supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT.IP and by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 (BehAPI).

REFERENCES

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *OOPSLA'09*. ACM Press, 1015. https://doi.org/10.1145/1639950.1640073

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral types in programming languages. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

[3] Rob DeLine and Manuel Fahndrich. 2004. *The Fugue Protocol Checker: Is Your Software Baroque?* Technical Report MSR-TR-2004-07. https://www.microsoft.com/en-us/research/publication/the-fugue-protocol-checker-is-your-software-baroque/

[4] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

[5] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Lúis Caires, Mmarco Carbone, Pierre Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of session types and behavioural contracts. *Comput. Surveys* 49, 1 (2016), 1–36. https://doi.org/10.1145/2873052

[6] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for rust. In *WGP'15, co-located with ICFP*. 13–22. https://doi.org/10.1145/2808098.2808100

[7] Wen Kokke. 2019. Rusty Variation Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science, EPTCS* 304 (2019). https://doi.org/10.4204/EPTCS.304.4

[8] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Science of Computer Programming* 155 (2018), 52–75. https://doi.org/10.1016/j.scico.2017.10.006

[9] João Mota. 2021. *Coping with the reality: adding crucial features to a typestate-oriented language*. Master's thesis. NOVA School of Science and Technology. https://github.com/jdmota/java-typestate-checker/blob/master/docs/msc-thesis.pdf

[10] Rogério Reis and Nelma Moreira. 2002. *FAdo: tools for finite automata and regular expressions manipulation*. Technical Report DCC-2002-02. Universidade do Porto.

[11] Robert E. Strom. 1983. Mechanisms for compile-time enforcement of security. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '83*. ACM Press, New York, New York, USA, 276–284. https://doi.org/10.1145/567067.567093

[12] André Trindade, João Mota, and Antonio Ravara. 2020. Typestates to automata and back: A tool. *Electronic Proceedings in Theoretical Computer Science, EPTCS* 324 (2020). https://doi.org/10.4204/EPTCS.324.4

[13] A. Laura Voinea, Ornela Dardha, and Simon J. Gay. 2020. Typechecking Java Protocols with [St]Mungo. In *FORTE'20 (Lecture Notes in Computer Science, Vol. 12136)*. Springer, 208–224. https://doi.org/10.1007/978-3-030-50086-3_12