Journal of Computer Languages 72 (2022) 101154

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Taming stateful computations in Rust with typestates ☆

José Duarte [a,b,*,1], António Ravara [a]

[a] NOVA LINCS & NOVA School of Science and Technology, Portugal
[b] Dystematic Ltd., England, United Kingdom

ARTICLE INFO

Keywords:
Rust
Typestate
Behavioral types
DSL
Meta-programming

ABSTRACT

As our lives become increasingly more reliant on software, the impact of its failures grows as well; these failures have diverse causes and their impact ranges from negligible to life-threatening; thus, it is our duty as developers to minimize their occurrence, just as other fields do.

To that end, we build abstractions, move complexity from component to component, and much more, just to stop the end-user from shooting themselves in the foot. However, building said abstractions still requires the original author to know where the pitfalls lie and how to avoid them, an implicit contract that does not constitute a guarantee that they will not shoot themselves and their users in the feet.

Rust aims to minimize the amount of handguns users have at their disposal, locking them behind special `unsafe` blocks and restricting the set of possible programs through static analysis; this analysis is performed by the compiler which ensures that the program does not contain memory related errors such as *use-after-free* bugs.

While Rust is able to succeed in the previous domain, other error classes persist, such as errors related to API misusage. Our work aims to tackle that domain, providing a tool which enables developers to write safer APIs using typestates.

We propose a macro which embeds a typestate description DSL in Rust which allows developers to specify typestates for their APIs; the typestate is checked at compile-time for common mistakes and to ensure the correct usage of the typestate, we leverage Rust's type system.

Our work only requires a Rust compiler, avoiding workflow bloat and keeping the development experience simple; it is open-source and available at https://github.com/rustype/typestate-rs.

1. Introduction

For years, languages such as C & C++ have dominated the systems programming landscape and for good reason — both languages provide fine-grain control over the machine, allowing developers to extract the most of their systems. However, with great power comes great responsibility; which, time and time again, we have proved to be unable to handle [1].

Rust is a programming language geared towards this problem space, bringing advances in programming languages into the mainstream to address problems such as memory management and concurrency. At the same time, Rust keeps performance and productivity at its core, ensuring developers make the most out of language.

While (safe) Rust covers a wide range of issues systems developers might face when using C/C++, it is unable to tackle issues such as protocol compliance (both at local and global levels).

Consider one of the simplest state machines there is — the light bulb; the developer might model the `Bulb` object with two methods, `turn_on` and `turn_off`, but what happens if we call `turn_off` twice in a row?

```
1  class Bulb:
2      def __init__(self):
3          self.is_on = False
4
5      def turn_off(self):
6          if not self.is_on:
7              raise ValueError("bulb is already
       off")
8          self.is_on = False
9
10     def turn_on(self):
11         if self.is_on:
13
```

☆ This article is an extended version of José Duarte and António Ravara (2021).

* Corresponding author at: NOVA LINCS & NOVA School of Science and Technology, Portugal.
E-mail addresses: jmg.duarte@campus.fct.unl.pt (J. Duarte), aravara@fct.unl.pt (A. Ravara).
[1] Affiliated to NOVA LINCS & NOVA School of Science and Technology at the time of writing.

https://doi.org/10.1016/j.cola.2022.101154
Received 21 February 2022; Received in revised form 22 August 2022; Accepted 7 September 2022
Available online 17 September 2022

```
12              raise ValueError("bulb is already on
    ")
14          self.is_on = True
```

Listing 1: A *very* naïve Python implementation of the light bulb automaton.

In the real world, by design, switches do not let you perform such sequence of actions; but in code, we might use `if` statements, exceptions or other kinds of control flow to replicate the same behavior but (see Listing 1 as an example); ideally we want to prevent our users from filing crash reports because they misused the APIs —

we want to forbid misuses by design and check for them during compile time.

Previously, Rust had a typestate system which would help handle this kind of issues, but unfortunately, it was removed [2,3] and while it still is possible to use typestates in Rust, no other first-class system took its place.

Recent ongoing research efforts have focused on Rust's memory model [4,5] or session types [6–9]; as far as we are aware, our work is the first to explore typestates as first-class members in modern Rust. We present a library which allows developers to declare statically-checked automata, providing a typestated API backed by Rust's advanced type system. We also verify that the developer is not "shooting themselves in the foot" with additional verifications over the automaton.

2. The `#[typestate]` macro

The `#[typestate]` macro's goals are twofold; aiming for correctness and safety while keeping the entry barrier for new users relatively low.

Our work is inspired by Fugue [10], a static checker for languages that compile to the Common Language Runtime. Fugue allows developers to specify, through the use of annotations, finite state machines and their respective properties (i.e. which states connect to which and more), then checked by Fugue for protocol compliance. Our `#[typestate]` macro takes this approach and Rust's powerful macro system to provide an embedded DSL, avoiding workflow bloat by relying solely on Rust's compiler. Furthermore, we attempt to reduce adoption hardships by keeping the DSL's syntax and semantics close to Rust's.

Like Fugue, our work allows users to define deterministic state machines However, we also support the definition of deterministic object automata [11]; the defined automata are verified by our macro while state transition enforcement is left up to the Rust compiler.

We proceed with an overview of the macro DSL, followed by a deeper discussion about its implementation.

2.1. Your first typestate

To get started using `#[typestate]`, the user needs only to have a state machine which they want to specify. Listing 2 defines a simple example: a light bulb. The bulb can only be turned on or off. Additionally it can be screwed into its socket or removed by unscrewing it from the socket, constructing or consuming the object, respectively.

The entrypoint for our typestate is the `#[typestate]` annotation, which can only be attached to modules. Our DSL lives inside said module, which, for our example, is defined in the second line of Listing 2.

The DSL defines three main specification elements (see Table 1):

The `#[automaton]` attribute (lines 3–4) defines the attached structure as the main one of our typestate; only a single definition is allowed per module. The typestate can have fields, which will be made available in all states.

The `#[state]` attribute (lines 6–7 & 14–15) defines the attached structure as a state; several states can be declared inside a module and each state can also contain data, which, however it will only be accessible to that specific state.

```
1  // The DSL's entrypoint
2  #[typestate]
3  mod light_bulb {
4      // Only one automaton is allowed per
       typestate specification
5      #[automaton]
6      pub struct LightBulb;
7
8      // State declaration, several declarations
       are allowed
9      #[state]
10     pub struct Off;
11
12     #[state]
13     pub struct On;
14
15     // Trait declaration, allowing the
       definition of transitions
16     // Traits share the name of the state they
       relate to
17     pub trait Off {
18         // Functions that do not consume `self`
       and return a valid
19         // state are considered to be "initial
       transitions"
20         fn screw() -> Off;
21
22         // Functions that consume `self` but do
       not return a valid
23         // state are considered to be "final
       transitions"
24         fn unscrew(self);
25
26         // Functions which consume `self` and
       return a valid
27         // state are considered to be
       deterministic transitions
28         fn turn_on(self) -> On;
29     }
30
31     pub trait On {
32         fn turn_off(self) -> Off;
33     }
34 }
```

Listing 2: A light bulb state machine, specified using the `#[typestate]` macro.

Table 1
Overview of the DSL's annotations.

Annotation	Attaches to	Declares
`#[typestate]`	Module	API
`#[automaton]`	Structure	Automaton/Typestate
`#[state]`	Structure	State

Traits (lines 8–12 & 16–18) define transitions, in our DSL traits share their name with the respective state. We define three main transition types: *initial*, *final* and *state* transitions (the latter being further subdivided into *deterministic* and *non-deterministic* transitions) (see Table 2).

Initial transitions are constructors, in other terms, they do not take a `self` parameter and will return a valid state. In Listing 2, the `screw` function is an initial transition.

Final transitions are the opposite of initial transitions, they will take `self` and not return a valid state, thus consuming it according to Rust's borrowing rules. In Listing 2 (see Fig. 3), the `unscrew` function is a final transition.

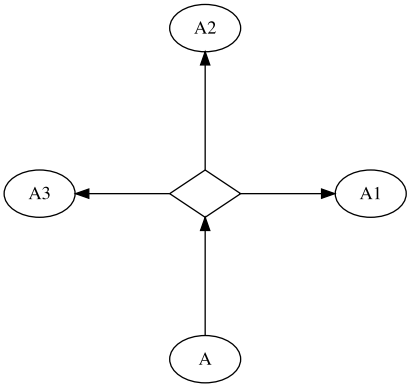


Fig. 1. Final graph example, without extra information the transition context is lost.

Table 2
Overview of the DSL's annotations.

Function signature	Consumes	Returns	Inferred
`fn (self, . . . ) -> State`	`State`	`State`	Transition
`fn (...) -> State`		`State`	Initial state
`fn (self, . . . ) -> ...`	`State`		Final state

State transitions are functions which take the current state (i.e. `self`) and return a new valid state, effectively, transitioning between state `A` and `B`. The functions `turn_on` and `turn_off` are examples of state transitions, as they perform the transition between the `On` and `Off` states.

Along with the already presented features, the user can also specify *self-transitions*, which we define as functions that take `&self` (mutable or not), thus, since these functions only borrow `self`, the ownership system disallows the consumption of the current state and consequently, stop the user from performing transitions.

2.2. Typestates and non-determinism

As we know, computations can fail, thus, considering points of failure is crucial for the correct functioning of any application. A place where one might find fallible states is at the boundary between the program and the outside world (i.e. IO), if the user is not careful, inputs that are unaccounted for may crash the program, or even worse, allow it to keep going, without the realization that the current state is invalid.

To consider such cases, where failures might happen, we take advantage of Rust's enumerations, effectively transforming our deterministic state machine into a deterministic object automaton [11].

Enumerations are considered a *non-deterministic state* as they are effectively the sum of several possible states; a neat side effect of enumerations is that the user is now forced to match against them and consider possible cases before proceeding with its computation.

Listing 4 is the extension of Listing 2, now including a state describing a bulb that has died (i.e. will not turn on again) and a state which considers such occurrence.

Additionally, users can add a `#[metadata(label="...")]` attribute to each enumeration arm, allowing them to add a label to each transition, without the attribute, the decision nodes' transitions (i.e. outgoing edges) will not have labels. Consider the code from Listing 3, a generic state `A`, a decision node and three possible outcomes, states `A1`, `A2` and `A3`; without the `metadata` attributes, the final graph is similar to Fig. 1, with the attributes, the edges contain extra information as shown in Fig. 2.

```
enum A {
    #[metadata(label="if x > 10")] A1,
    #[metadata(label="if x == 10")] A2,
    #[metadata(label="if x < 10")] A3,
}
```

Listing 3: `#[metadata]` usage example.

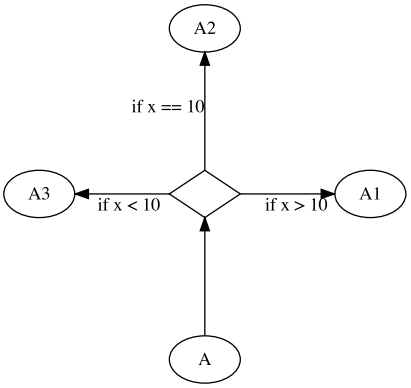


Fig. 2. Final graph example while using `#[metadata(label="...")]` to document each state's condition.

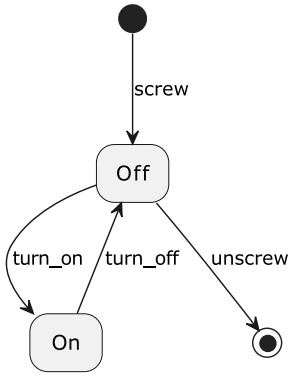


Fig. 3. Listing 2 automaton visualization.

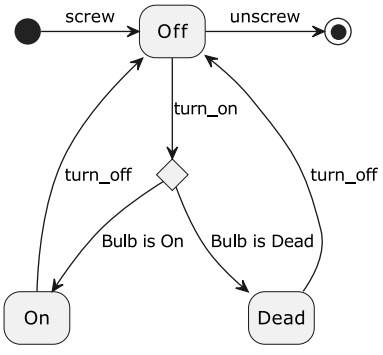


Fig. 4. Listing 4 automaton visualization.

```rust
#[typestate]
mod light_bulb {
    #[automaton]
    pub struct LightBulb;

    #[state]
    pub struct Off;

    #[state]
    pub struct On;

    #[state]
    pub struct Dead;

    // Enumerations will contain *only* existing states
    // Given that non-deterministic states are handled
    // outside the typestate they do not have traits associated to them
    pub enum AfterOn {
        On,   // The case in which the bulb was able to be turned on
        Dead  // The case in which the bulb is dead
    }

    pub trait Off {
        fn screw() -> Off;
        fn unscrew(self);

        // Now returns a possible dead bulb
        fn turn_on(self) -> AfterOn;
    }

    pub trait On {
        fn turn_off(self) -> Off;
    }

    // If the bulb is dead, we turn the switch off so we can unscrew
    pub trait Dead {
        fn turn_off(self) -> Off;
    }
}
```

Listing 4: The light buld state machine, now able to cope with the bulb's failure.

```rust
fn main() {
    let bulb = LightBulb::<Off>::screw();
    let bulb = bulb.turn_on();
    match bulb {
        On(state) => // ...
        Dead(state) => state.unscrew();
    }
}
```

Listing 5: Example usage of the state machine from Listing 4.

2.3. Architecture and implementation

The macro's architecture is illustrated in Fig. 5. As previously stated, the DSL's entrypoint is the `#[typestate]` attribute; during compilation, Rust's macro system will process the attribute and attempt to expand it.

This section demystifies the expansion process of our work; in short the expansion process will perform two main tasks: expand the input module into typestate-oriented constructs, and validate that the typestate obeys a set of rules, namely — all states should be both productive and useful (for a state S to be considered *productive* the final state must be reachable from S; for a state S to be considered *useful*, it must be productive *and* be reachable from at least one initial state); if the typestate fails to check against the rules, the expansion is stopped and an error is issued.

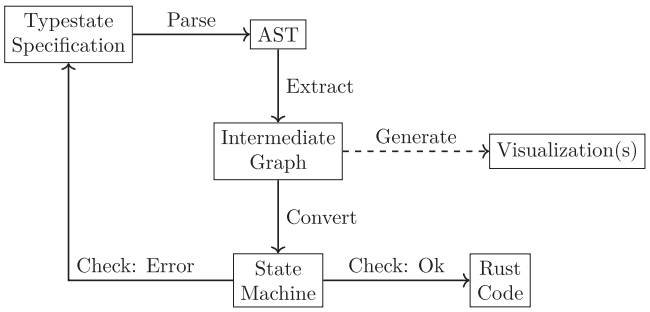


Fig. 5. From DSL specification to Rust code. First the DSL is parsed, an intermediate graph representing the automaton in more general terms is extracted from the AST, from the graph the macro will convert the user can generate visualizations (for debugging or documentation); this last step is optional.

2.3.1. From the AST to the typestate

The input for an attribute macro are two `TokenStreams`; one for the attribute itself, which can take many forms and thus, must be parsed; and the code's AST. However, we still need to extract meaning from the input, the `syn` crate grants us the ability to visit and modify the AST's nodes.

The `#[typestate]` macro performs three visits, targeting different nodes in the AST: it starts by visiting structures, to extract the automaton structure and its states; then, it visits all the enumerations, extracting the *decision* nodes and checking if there are states missing or not connected; finally, it will parse the traits, extracting the transitions, made possible because all states are known at this time of the process.

2.3.2. Validating the typestate

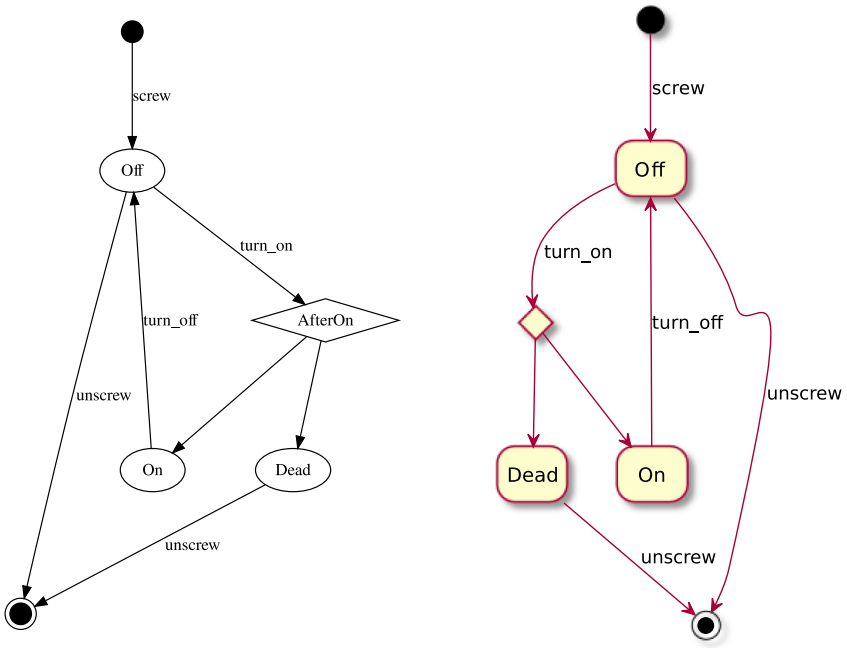
Internally, the typestate is represented as a directed graph with two kinds of nodes — deterministic and non-deterministic; to simplify the analysis process, the graph is converted into a standard directed graph with a single node kind. This is done by expanding the non-deterministic nodes into transitions between the start node up to all other destination nodes. To exemplify — consider deterministic states A, B, C and a single non-deterministic state X, and the following transitions $A \to X$ and $X \to \{B, C\}$; the expansion of X will yield transitions $A \to B$ and $A \to C$.

The validation is then done over the expanded graph, ensuring that all states are useful.

2.4. Visualizing your typestates

Since mentally visualizing data structures is not trivial, we provide the ability to export your typestate as a graph file, in the DOT, PlantUML or Mermaid formats; the latter can be used to render the state machine inside the documentation, enhancing communication between developers and preventing documentation rot, as the visualization is generated by the macro each time `cargo doc` is run.

All typestate visualizations in this paper were generated by our macro. The macro generates a file in the target language (e.g. DOT, PlantUML or Mermaid), it is then up to the user to "compile" the file using their preferred tool (e.g. `dot`, `neato`, etc.). In our case, we used different tools, hence the differences between Figs. 4, 6(a) and 6(b).



(a) Rendered with GraphViz's DOT. (b) Rendered with PlantUML.

Fig. 6. The light bulb state machine, visualized.

3. Case studies

We present three case studies, highlighting different features of our macro.

- The ring example, where we have N participants sitting in a circle, and each participant will receive a message from their left neighbor and replay it to their right neighbor.
- The PIN example, which has three principals, a card, which is read by the card reader, and the client which uses the reader.
- The auction example; which demonstrates how one can build a *typestated* API on top of an existing one, enhancing guarantees provided by the latter.

3.1. Ring

The ring example is taken from the Rumpsteak [9] repository.[2] We now present a direct comparison with the Rumpsteak library, which aims to ease the implementation of message-passing asynchronous programs in Rust. It makes use of multi-party session types [12] to verify the absence of communication errors and asynchronous subtyping [13] to verify message reordering and optimize communication.

3.1.1. Rumpsteak

The Rumpsteak library will first declare a series of types, the global type — `Roles`; each participant endpoint — `structs` `A`, `B` and `C`; the message being passed around — `Value`; finally, each session type — `structs` `RingA`, `RingB` and `RingC`. The functionality of each participant is then declared as an `async` function, notice that `ring_b` (lines 32–39 from Listing 6) and `ring_c` (lines 41–48 from Listing 6) are identical, except for the types used.

Given that Rumpsteak is oriented towards session types, the library is able to enforce communication patterns at the type level. In lines 19–21 of Listing 6 we can see through `RingA`'s type that `A` will forcibly send the value to `B` and will then receive the value from `C` (`#[typestate]` is unable to enforce this, as it is intended to control only each participant's protocol).

The launch routine (Listing 7) is standard, using the executor from the `future` crate, along with the `try_join` macro as a convenience to launch each one. Rumpsteak ends up being fairly verbose w.r.t. the used types, although the rest of development is straightforward. The library would greatly benefit from a visualization feature such as the one described in 2.4 or in https://typestate-editor.github.io/ (visited in 9/2/2022).

3.1.2. `#[typestate]`

At the type-level, using `#[typestate]` requires less background knowledge than Rumpsteak; a user that understands the DSL should be able to infer what happens *behind the scenes*.

In our ring implementation (Listings 8, 10, 12) the protocol is enforced by declaring the state machine of each participant; in this example, the ring can scale up to N participants — one for the participant that initiates communication (i.e. `RingA`) and another for the following participants (i.e. `RingB`).

Both participants receive two channels (respectively, lines 12 & 23–25 in Listing 8, and lines 18 & 38–40 in Listing 10), one from which they will receive the value and one to which they will send the value, these channels are received upon instantiation of the respective `Ring` object.

[2] https://github.com/zakcutner/rumpsteak/blob/a30ab689229e90e199b80819ed2fb52b2560f009/examples/ring.rs (visited in 24/1/2022).

```rust
#[derive(Roles)]
struct Roles(A, B, C);

#[derive(Role)]
#[message(Value)]
struct A(#[route(B)] Sender, #[route(C)] Receiver);

#[derive(Role)]
#[message(Value)]
struct B(#[route(A)] Receiver, #[route(C)] Sender);

#[derive(Role)]
#[message(Value)]
struct C(#[route(A)] Sender, #[route(B)] Receiver);

#[derive(Message)]
struct Value(i32);

#[session] type RingA = Send<B, Value, Receive<C, Value, End>>;
#[session] type RingB = Receive<A, Value, Send<C, Value, End>>;
#[session] type RingC = Receive<B, Value, Send<A, Value, End>>;

async fn ring_a(role: &mut A, input: i32) -> Result<i32> {
    let x = input;
    try_session(role, |s: RingA<'_, _>| async {
        let s = s.send(Value(x)).await?;
        let (Value(y), s) = s.receive().await?;
        Ok((x + y, s))
    }).await
}

async fn ring_b(role: &mut B, input: i32) -> Result<i32> {
    let x = input;
    try_session(role, |s: RingB<'_, _>| async {
        let (Value(y), s) = s.receive().await?;
        let s = s.send(Value(x)).await?;
        Ok((x + y, s))
    }).await
}

async fn ring_c(role: &mut C, input: i32) -> Result<i32> {
    let x = input;
    try_session(role, |s: RingC<'_, _>| async {
        let (Value(y), s) = s.receive().await?;
        let s = s.send(Value(x)).await?;
        Ok((x + y, s))
    }).await
}
```

Listing 6: The Ring implementation using the Rumpsteak library.

The process starts with `RingA` in the *sending* state (lines 10–16 of Listing 8, emphasizing line 12, which declares the initial state); in this state, the API allows the client to: *1* — inspect the value with `get_value` (lines 13 & 27); *2* — transition to the next state with `send` (lines 14 & 29–36); *3* — end the protocol with `end` (lines 15 & 38).

`RingB` (and any following participants) will first receive the value and then send it to the next participant, as expected. Its protocol (Listing 10) starts in a *receiving* state (lines 16–21, with emphasis in line 18, which declares the initial state).

```rust
fn main() {
    let Roles(mut a, mut b, mut c) = Roles::default();
    let input = (1, 2, 3);
    println!("input = {:?}", input);
    let output = executor::block_on(async {
        try_join!(
            ring_a(&mut a, input.0),
            ring_b(&mut b, input.1),
            ring_c(&mut c, input.2),
        )
        .unwrap()
    });
    println!("output = {:?}", output);
}
```

Listing 7: Rumpsteak's Ring `main` function.

```rust
#[typestate]
mod ring_a {
    use std::sync::mpsc::{Receiver, Sender};

    // The Ring automaton structure
    #[automaton] pub struct RingA {
        pub(crate) send: Sender<i32>,
        pub(crate) receiver: Receiver<i32>,
    }

    // The Send state for B
    // It is the starting and end state,
    // indicated by the constructor ‘new‘ and "destructor" ‘end‘
    #[state] pub struct SendA(pub i32);
    pub trait SendA {
        fn new(
            value: i32,
            send: Sender<i32>,
            receiver: Receiver<i32>
        ) -> SendA;
        fn get_value(&self) -> i32;
        fn send(self) -> RecvA;
        fn end(self);
    }

    // The Recv state for B
    // It can only receive the value, transitioning state
    #[state] pub struct RecvA;
    pub trait RecvA {
        fn recv(self) -> SendA;
    }
}
```

Listing 8: Ring participant A typestate specification.

While `#[typestate]` is unable to enforce communication patterns, it is able to reuse the participants more effectively, given that B and C perform the same operations, we can simply reuse `RingB` for both (lines 8–9 of Listing 12). We can further exploit this to design a macro that saves us the thread declaration boilerplate, as shown in Listing 13.

3.1.3. Comparison summary

As previously stated, Rumpsteak allows the developer to enforce communication patterns at compile-time (i.e. the "external" protocol),

```rust
// Implementation of the Send state
impl SendAState for RingA<SendA> {
    fn new(
        value: i32,
        send: Sender<i32>,
        receiver: Receiver<i32>
    ) -> Self {
        Self { send, receiver, state: SendA(value) }
    }

    fn get_value(&self) -> i32 {
        self.state.0
    }

    fn send(self) -> RingA<RecvA> {
        self.send.send(self.state.0).unwrap();
        RingA::<RecvA> {
            send: self.send,
            receiver: self.receiver,
            state: RecvA,
        }
    }

    fn end(self) {}
}

// Implementation of the Recv state
impl RecvAState for RingA<RecvA> {
    fn recv(self) -> RingA<SendA> {
        let value = self.receiver.recv().unwrap();
        RingA::<SendA> {
            send: self.send,
            receiver: self.receiver,
            state: SendA(value),
        }
    }
}
```

Listing 9: Ring participant A implementation

```rust
#[typestate]
mod ring_b {
    use std::sync::mpsc::{Receiver, Sender};

    // The Ring automaton structure
    #[automaton]
    pub struct RingB {
        pub(crate) send: Sender<i32>,
        pub(crate) receiver: Receiver<i32>,
    }

    // The Send state for B
    // It allows the value to be read and sent
    #[state] pub struct SendB(pub i32);
    pub trait SendB {
        fn get_value(&self) -> i32;
        fn send(self) -> RecvB;
    }

    // The Recv state for B
    // It is the starting and end state,
    // indicated by the constructor ‘new‘ and "destructor" ‘end‘
    #[state] pub struct RecvB;
    pub trait RecvB {
        fn new(send: Sender<i32>, receiver: Receiver<i32>) -> RecvB;
        fn recv(self) -> SendB;
        fn end(self);
    }
}
```

Listing 10: Ring participant B typestate specification.

while `#[typestate]` does not. However, the latter is able to enforce behavior patterns at the participant level (i.e. the “internal” protocol). The `#[typestate]` macro is more flexible, being suited for both synchronous and concurrent programming as well as being a better fit for more domains (namely, embedded development, where typestates are already used[3]), while Rumpsteak is solely usable in `async/await` contexts.

While very different, there is nothing stopping a user from using both libraries in conjunction, using Rumpsteak to control communication patterns and `#[typestate]` to control behavioral patterns.

3.2. PIN

As stated, the PIN example has two main components, a card and its reader, the latter is used by a final client (API or human). The client that uses the reader is detailed in Listing 14, which in turn interacts with the card being read, detailed in Listing 15. Using the card requires the reader to run the `check_for_card` function (line 10 of Listing 14); from here, the card is either present, allowing the transition to the `CardPresent` state (lines 13–18 of Listing 15), or not, resulting in a transition to the `Error` state (lines 28–33 of Listing 14).

[3] https://docs.rust-embedded.org/book/static-guarantees/typestate-programming.html (visited 9/2/2022).

Along with the previous check, the `CardPresent` state also requires the `Card` itself to be in the `Start` state, this feature allows users to enforce properties *across* typestates, relating them with each other (shown in Fig. 7). From the `CardPresent` state, the reader can issue an authentication operation, done by checking the PIN with the one stored in the card. In case the authentication succeeds, both the reader and the card states transition to their `Authenticated` state.

Any client of the reader API will be required to check all steps before moving forward. The state-embedding mechanism also helps ensure that all the `Reader` is a well-behaved client of `Card`.

3.3. Auction

The goal of the auction API is to stop the user from performing non-optimal bids (i.e. higher than the existing ones); if the user is outbid by another, the client is then required to withdraw its bid before submitting another.

The API (listed in Listing 16, visualized in Fig. 8) allows the client to bid on an auction, while emulating the closing of the auction (by randomly generating a boolean), if the auction closes, the bidding follows and is closed too.

The client is required to check if the auction is running before performing any actions; this is done by calling the `has_ended` function (line 13 of Listing 17), which returns the `AuctionState` state (line 46 of Listing 17), an enumeration representing each possible outcome, from here the developer is forced to handle the outcomes to proceed (usually by using `match`), in this case, that will mean either end the session, in case the auction is closed, or proceed to the `NoBids` state, as the user did not submit any bids. From this state, the user can perform a bid; in case that bid is not the highest, the user is forced to withdraw their bid, if their bid is the highest, the user can check if the auction has ended or if their bid is still the highest. If their bid is no longer the highest, they are forced to withdraw; if the auction ended the user can

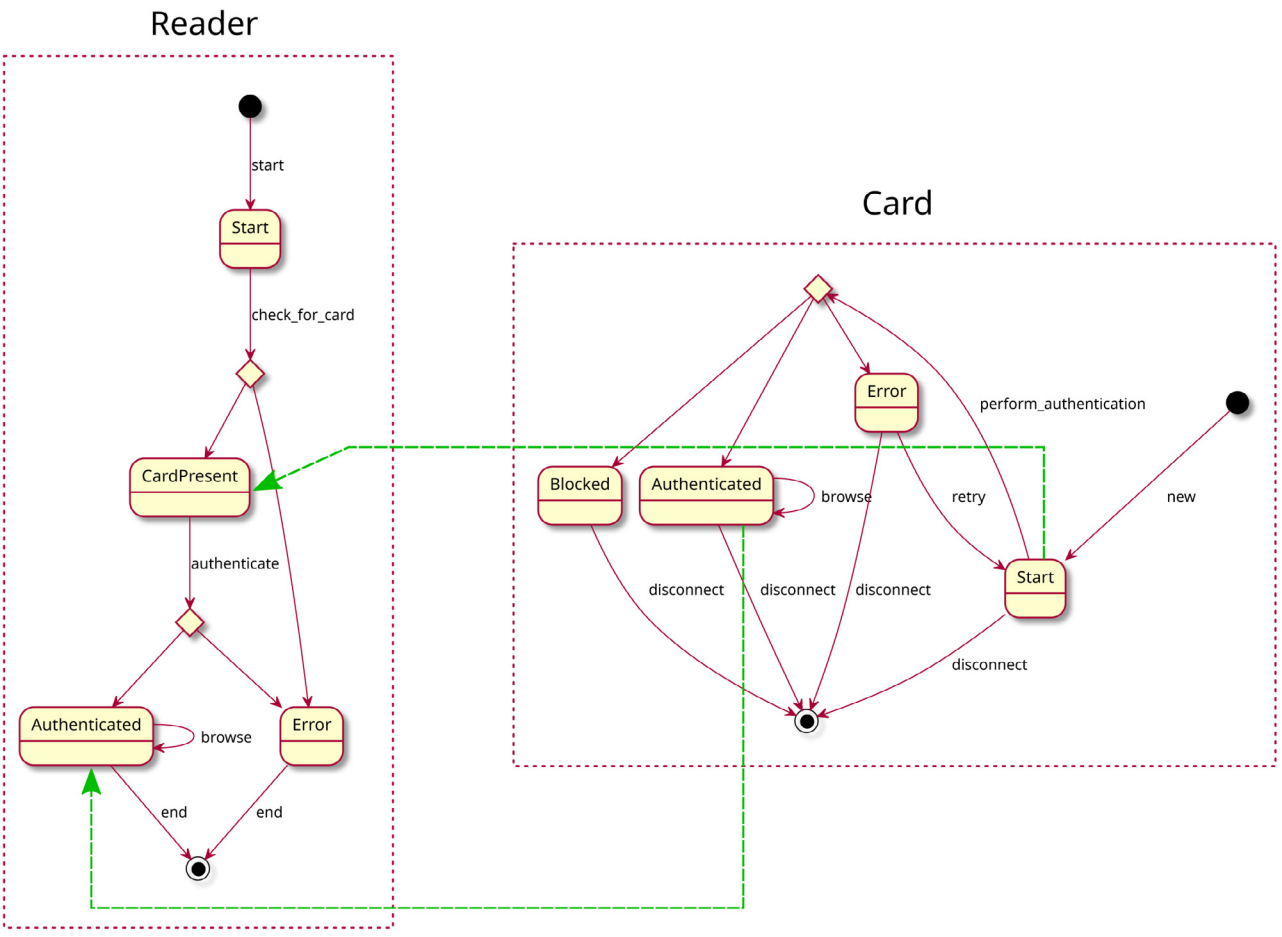


Fig. 7. The reader and card typestates. The green line shows the relationship across states of different typestates, implying that the card's `Start` state required the reader to be in the `CardPresent` state, the same applies for the `Authenticated` states. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

check if they won, taking the corresponding action afterwards before closing the session.

4. Related work

4.1. A session type provider

This work by Neykova et al. leverages F#'s type providers to facilitate practical session types along with refinements. The refinements allow developers to constrain the protocol's messages, described using Scribble; the protocol and the respective refinements are validated before code generation by the .NET platform.

4.2. `session-types`

The crate introduced in [6] was the first to bring session types [14] into Rust, providing an interpretation of session types and a base for the work which followed.

While the library provides the required abstractions for session types, they are built on top of `unsafe` blocks, which may deter some users and be completely forbidden in some projects.

The provided model for session types normalizes long `Offer/Choice` (i.e. session type constructs) chains, leading to complex and verbose types when the compiler reports errors. While crate offers abstractions to deal with the previous problems, they fail when using loops.

4.3. `sesh`

In contrast to `session-types`, the `sesh` crate [7], 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 does not use `unsafe` blocks, it does require a *nightly* compiler (due to the usage of the `!` or "never" type, a feature in development since 2016 [15]); not recommended for usage in production. Furthermore, at the time of writing the crate's last update was at the 17th of February 2021.

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.

4.4. Multiparty session types for rust

Akin to previous work, the work by Lagaillardie et al. [8] leverages the Scribble toolchain to improve upon `sesh` essentially extending it to multiparty session types. Unlike other Rust libraries, this work requires the usage of external libraries, adding complexity to the development process.

4.5. Rumpsteak

Rumpsteak [9] is a Rust library targeting asynchronous applications, it offers clean session types and its approach relies heavily on macros. The crate works by defining a global type which is projected into

```rust
// Implementation of the Send state
impl SendBState for RingB<SendB> {
    fn get_value(&self) -> i32 {
        self.state.0
    }

    fn send(self) -> RingB<RecvB> {
        self.send.send(self.state.0).unwrap();
        RingB::<RecvB> {
            send: self.send,
            receiver: self.receiver,
            state: RecvB,
        }
    }
}

// Implementation of the Recv state
impl RecvBState for RingB<RecvB> {
    fn new(send: Sender<i32>, receiver: Receiver<i32>) -> Self {
        Self { send, receiver, state: RecvB }
    }

    fn recv(self) -> RingB<SendB> {
        let value = self.receiver.recv().unwrap();
        RingB::<SendB> {
            send: self.send,
            receiver: self.receiver,
            state: SendB(value),
        }
    }

    fn end(self) {}
}
```

Listing 11: Ring participant B's implementation.

roles, this is done by Scribble which in turn requires an external toolchain; from each role an endpoint finite state machine is extracted and optimized, in the end an API is generated which can then be used to build each process. To enforce linear resource usage when build the processes the library makes use of the type system to enforce the consumption of each "state"; enforcing protocol completion is done through a closure, which takes the initial session type and returns a terminal `End` type, a session is then required to be run until completion otherwise the return type will not be respected and the type checker will raise an error.

4.6. Plaid

Part of the *typestate-oriented* paradigm, the Plaid language [16] is an extension of the object paradigm. 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, similar to Rust's affine 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;

```rust
fn main() {
    // Setup the channels
    let (a_sender, b_receiver) = channel::<i32>();
    let (b_sender, c_receiver) = channel::<i32>();
    let (c_sender, a_receiver) = channel::<i32>();
    // Setup the ring participants in their staring states.
    let a = RingA::<SendA>::new(0, a_sender, a_receiver);
    let b = RingB::<RecvB>::new(b_sender, b_receiver);
    let c = RingB::<RecvB>::new(c_sender, c_receiver);
    // Spawn a thread for each participant
    vec![
        thread::spawn(move || {
            // RingA starts by sending the value
            println!("a: {}", a.get_value());
            let a = a.send();
            let a = a.recv();
            a.end();
        }),
        thread::spawn(move || {
            // RingB starts by receiving the value from RingA
            let b = b.recv();
            println!("b: {}", b.get_value());
            let b = b.send();
            b.end();
        }),
        thread::spawn(move || {
            // RingC works just like RingB
            let c = c.recv();
            println!("c: {}", c.get_value());
            let c = c.send();
            c.end();
        }),
    ]
    .into_iter()
    .map(|handle| handle.join())
    .collect::<Result<_, _>>()
    .unwrap()
}
```

Listing 12: Typestate's Ring `main` function.

- 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.

4.7. Mungo & StMungo

Mungo is a project that treads the waters of both typestates (being core project) and session types an extended version (StMungo), built on top of Scribble and Mungo.

4.7.1. Mungo

The core project, named solely Mungo, can be described as a framework, composed of a Java annotation, a typestate description language and its checker, the project [17] allows developers to make use of typestates in languages such as Java.

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`

```
macro_rules! ring_recv {
    ($ident:ident) => {
        thread::spawn(move || {
            let $ident = $ident.recv();
            println!("{}: {}", stringify!($ident
    ), $ident.get_value());
            let $ident = $ident.send();
            $ident.end();
        })
    }
}

fn main() {
    // Setup the channels
    let (a_sender, b_receiver) = channel::<i32
    >();
    let (b_sender, c_receiver) = channel::<i32
    >();
    let (c_sender, a_receiver) = channel::<i32
    >();
    // Setup the ring participants in their
    staring states.
    let a = RingA::<SendA>::new(0, a_sender,
    a_receiver);
    let b = RingB::<RecvB>::new(b_sender,
    b_receiver);
    let c = RingB::<RecvB>::new(c_sender,
    c_receiver);
    // Spawn a thread for each participant
    vec![
        thread::spawn(move || {
            // RingA starts by sending the value
            println!("a: {}", a.get_value());
            let a = a.send();
            let a = a.recv();
            a.end();
        }),
        // Using the ring_recv! macro we can
    reduce code duplication
        ring_recv!(b), ring_recv!(c),
    ]
    .into_iter()
    .map(|handle| handle.join())
    .collect::<Result<_, _>>()
    .unwrap()
}
```

Listing 13: Typestate's Ring `main` function, using macros to reduce code duplication.

```
#[typestate]
pub mod reader_api {
    use crate::card::card_api;

    #[automaton] pub struct Reader;

    #[state] pub struct Start;
    pub trait Start {
        fn start() -> Start;
        fn check_for_card(self) ->
    CheckCardResult;
    }

    #[state] pub struct CardPresent {
        pub card: card_api::Card<card_api::Start
    >,
    }
    pub trait CardPresent {
        fn authenticate(self, pin: [u8; 4]) ->
    AuthResult;
    }

    #[state] pub struct Authenticated {
        pub card: card_api::Card<card_api::
    Authenticated>,
    }
    pub trait Authenticated {
        fn browse(&self);
        fn end(self);
    }

    #[state] pub struct Error {
        pub message: String,
    }
    pub trait Error {
        fn end(self);
    }

    pub enum CheckCardResult { CardPresent,
    Error }
    pub enum AuthResult { Authenticated, Error }
}
```

Listing 14: The `Reader` typestate specification.

(`"ProtocolName"`), extracts the method call behavior and checks the extracted information against the typestate.

The Mungo toolchain is available online[4] and a new implementation (JATYC [18]) is available on GitHub.[5]

4.7.2. StMungo

StMungo [17,19] is a transpiler from Scribble to Java based on session types and typestates. The transpilation process takes Scribble local protocols as input, generating Mungo typestate specifications and a Java code "skeleton" for implementation; the output is checked by Mungo. The process is based on a formal translation from session types into typestates for channel objects.

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

4.8. Fugue

> 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. – [10]

Similar to our work, Fugue performs static checks to produce a list of errors and warnings, without requiring any intervention during runtime; although Fugue is a closed source project, targeted towards C#.

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 achieved through the ownership system, not requiring any addition to the language; 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*" [10].

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.

```rust
#[typestate]
pub mod card_api {
    #[automaton] pub struct Card {
        pub valid_pin: [u8; 4],
        pub attempts_left: u8,
    }

    #[state] pub struct Start;
    pub trait Start {
        fn new() -> Start;
        fn perform_authentication(self, pin: [u8; 4]) -> AuthResult;
        fn disconnect(self);
    }

    #[state] pub struct Authenticated;
    pub trait Authenticated {
        fn browse(&self);
        fn disconnect(self);
    }

    #[state] pub struct Error;
    pub trait Error {
        fn retry(self) -> Start;
        fn disconnect(self);
    }

    #[state] pub struct Blocked;
    pub trait Blocked {
        fn disconnect(self);
    }

    pub enum AuthResult { Authenticated, Blocked, Error }
}
```

Listing 15: The `Card` typestate specification.

While Fugue requires machinery not included in the language to enforce method call ordering, our work leverages the type system avoiding the need for extra components, furthermore, our work also allows typestates to rely on other typestates as demonstrated by Section 3.2 example.

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 modeled 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.

4.9. Summary

While all the presented projects present positive results they also present severe deficiencies. The Plaid language has been abandoned and its 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.

```rust
pub struct Auction {
    owner: String,
    bids: HashMap<String, u64>,
    highest_bid: u64,
    ended: bool,
}

impl Auction {
    pub fn new(owner: String) -> Self {
        Self {
            owner,
            bids: HashMap::new(),
            highest_bid: 0,
            ended: false,
        }
    }

    pub fn bid(&mut self, client: String, bid: u64) -> Option<()> {
        if self.owner != client && !self.has_ended() {
            self.ended = rand::random(); // simulate uncertainty
            self.bids.insert(client, bid);
            if self.highest_bid < bid {
                self.highest_bid = bid;
            }
            Some(())
        } else {
            None
        }
    }

    pub fn is_highest_bid(&self, client: &String) -> bool {
        match self.bids.get(client).map(|bid| self.highest_bid == *bid) {
            Some(is_highest) => is_highest,
            None => false,
        }
    }

    pub fn get_bid(&self, client: String) -> Option<&u64> {
        self.bids.get(&client)
    }

    pub fn has_ended(&self) -> bool { self.ended }
}
```

Listing 16: The auction's *non-typestated* API.

Regarding Fugue, while its approach using annotations is clever and elegant, it leaves much to be desired regarding code readability as it is very verbose when declaring states and their transitions. Our approach avoids this problem by using language constructs to cleanly express the typestate, instead of relying solely on annotations.

5. Conclusion

In this paper we presented a novel approach to typestates in Rust, while nothing stops users from writing typestates themselves, by hand, the process is cumbersome and error-prone; our work leverages Rust type and macro systems to provide a simple yet powerful DSL, able to specify and verify typestates whilst simplifying their development.

While the end-product is a single DSL, the project as a whole includes a verification system and visualization tool for the developed

```rust
#[typestate]
mod auction_client_api {
    use crate::auction::Auction;

    #[automaton] pub struct Client {
        pub(crate) name: String,
        pub(crate) auction: Auction,
    }

    #[state] pub struct AuctionRunning;
    pub trait AuctionRunning {
        fn start(name: String, auction: Auction)
     -> AuctionRunning;
        fn has_ended(self) -> AuctionState;
    }

    #[state] pub struct NoBids;
    pub trait NoBids {
        fn bid(self, bid: u64) -> BidStatus;
    }

    #[state] pub struct HasBidded;
    pub trait HasBidded {
        fn check_bid(self) -> BidStatus;
        fn has_ended(self) -> AuctionEnded;
    }

    #[state] pub struct CheckWinner;
    pub trait CheckWinner { fn is_highest_bid(
    self) -> WinnerStatus; }

    #[state] pub struct Withdraw;
    pub trait Withdraw { fn withdraw(self) ->
    AuctionRunning; }

    #[state] pub struct Lost;
    pub trait Lost { fn withdraw(self) -> End; }

    #[state] pub struct Winner;
    pub trait Winner { fn win(self) -> End; }

    #[state] pub struct End;
    pub trait End { fn end(self); }

    pub enum WinnerStatus { Lost, Winner }

    pub enum AuctionEnded { HasBidded,
    CheckWinner }

    pub enum AuctionState { NoBids, End }

    pub enum BidStatus { HasBidded, Withdraw }
}
```

Listing 17: The auction's typestated client specification.

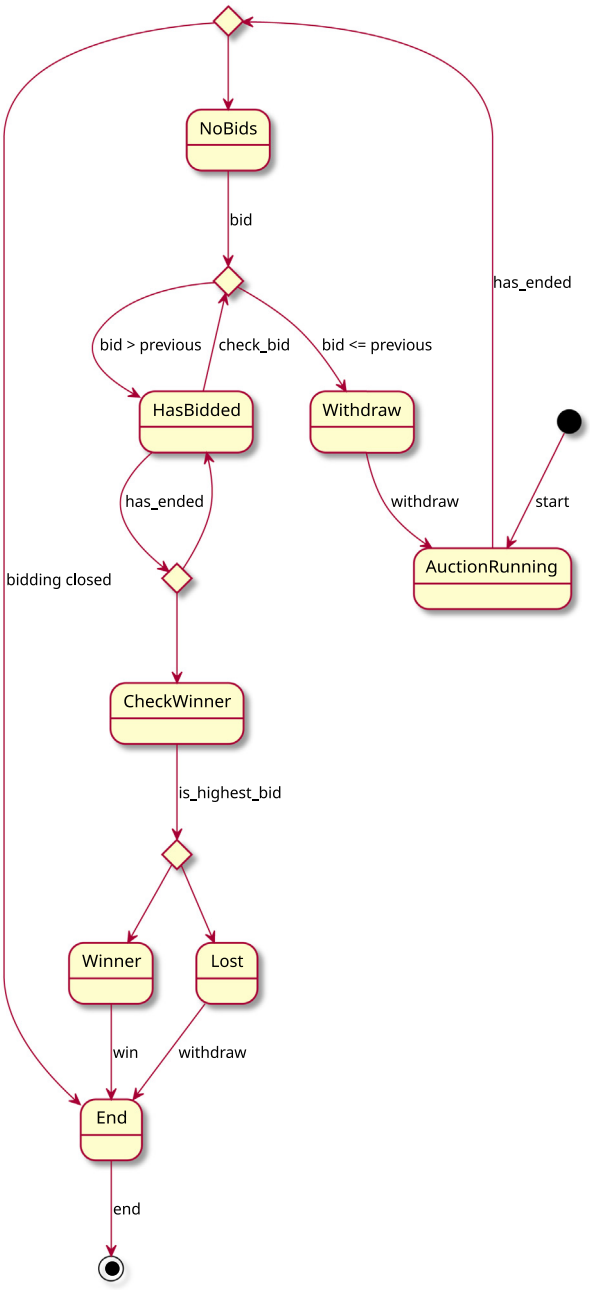


Fig. 8. The auction client API typestate.

typestates; all bundled in a single library which does not require any kind of external tooling (a common practice in other works). We hope to see more research surrounding typestates, moving towards safer programming practices, and hope (maybe naively) that in the future, mechanisms like typestates become as common as type systems are at the time of writing.

6. Future work

At the time of writing, the present work has several possible points of improvements, being some of these non-trivial and thus relegated as possible standalone projects. The issues to address are: (*i*) add support for generics and lifetimes; while a basic version has been implemented (available in version `0.9.0-rc2`), it does not cover much more than basic cases; (*ii*) review and improve the existing syntax; the aim is to further mix the specification and the implementation of the typestate, allowing them to coexist inside the module; (*iii*) formally define the checking process, state and prove the key properties envisaged; (*iv*) further invest in learning materials; currently an online book is made available at https://rustype.github.io/typestate-rs/ and in the project's repository there are two documents available [20,21], but it still is in its infancy, being closer to a collection of starting points rather than a complete guide to `#[typestate]`.

CRediT authorship contribution statement

José Duarte: Conceptualisation, Software, Validation, Visualisation, Writing. **António Ravara:** Resources, Writing – review, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Financial support was provided by NOVA LINCS (UIDB/04516/2020) backed by FCT.IP and by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233.

References

[1] Matt Miller, Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, 2019.
[2] Brian Anderson, GitHub issues - rust-lang/rust – remove typestate, etc., 2012.
[3] Brian Anderson, GitHub issues - rust-lang/rust – remove typestate from the language, 2012.
[4] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, Derek Dreyer, Rustbelt meets relaxed memory, Proc. ACM Program. Lang. 4 (POPL) (2019).
[5] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, Derek Dreyer, Stacked borrows: An aliasing model for rust, Proc. ACM Program. Lang. 4 (POPL) (2019).
[6] Thomas Bracht Laumann Jespersen, Philip Munksgaard, Ken Friis Larsen, Session types for rust, in: WGP'15, Co-Located with ICFP, 2015, pp. 13–22.
[7] Wen Kokke, Rusty variation deadlock-free sessions with failure in rust, Electron. Proc. Theor. Comput. Sci. EPTCS 304 (2019).
[8] Nicolas Lagaillardie, Rumyana Neykova, Nobuko Yoshida, Implementing multiparty session types in rust, in: Simon Bliudze, Laura Bocchi (Eds.), Coordination Models and Languages, Springer International Publishing, Cham, 2020, pp. 127–136.
[9] Zak Cutner, Nobuko Yoshida, Martin Vassor, Deadlock-free asynchronous message reordering in rust with multiparty session types, 2021.
[10] Rob DeLine, Manuel Fahndrich, The Fugue Protocol Checker: Is Your Software Baroque?, Technical Report MSR-TR-2004-07, Microsoft Research, 2004.
[11] André Trindade, João Mota, Antonio Ravara, Typestates to automata and back: A tool, Electron. Proc. Theor. Comput. Sci. EPTCS 324 (2020).
[12] Nobuko Yoshida, Lorenzo Gheri, A very gentle introduction to multiparty session types, in: 16th International Conference on Distributed Computing and Internet Technology, in: LNCS, 11969, Springer, 2020, pp. 73–93.
[13] Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, Nobuko Yoshida, Precise subtyping for asynchronous multiparty sessions, Proc. ACM Program. Lang. 5 (POPL) (2021).
[14] Kohei Honda, Types for dyadic interaction, in: Eike Best (Ed.), CONCUR'93, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 509–523.
[15] Niko Matsakis, Github issues - rust-lang/rust – tracking issue for promoting ! to a type (RFC 1216), 2016.
[16] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, Zachary Sparks, Typestate-oriented programming, in: OOPSLA'09, ACM Press, 2009, p. 1015.
[17] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, Simon J. Gay, Typechecking protocols with Mungo and StMungo: A session type toolchain for Java, Sci. Comput. Program. 155 (2018) 52–75.
[18] João Daniel da Luz Mota, Coping with the Reality: Adding Crucial Features to a Typestate-Oriented Language (Master's thesis), NOVA School of Science and Technology, 2021.
[19] A. Laura Voinea, Ornela Dardha, Simon J. Gay, Typechecking java protocols with [St]Mungo, in: FORTE'20, in: Lecture Notes in Computer Science, 12136, Springer, 2020, pp. 208–224.
[20] José Miguel Duarte Duarte, Retrofitting Typestates into Rust (Master's thesis), NOVA School of Science and Technology, 2021.
[21] José Duarte, António Ravara, Retrofitting typestates into rust, in: 25th Brazilian Symposium on Programming Languages, SBLP '21, 2021, pp. 83–91.