

Typestates to Automata and back: a tool

André Trindade

NOVA School of Science and Technology
NOVA University Lisbon
Lisbon, Portugal
adt.trindade@campus.fct.unl.pt

João Mota

NOVA School of Science and Technology
NOVA University Lisbon
Lisbon, Portugal
jd.mota@campus.fct.unl.pt

António Ravara

NOVA School of Science and Technology
NOVA University Lisbon
Lisbon, Portugal
aravara@fct.unl.pt

Development of software is an iterative process. Graphical tools to represent the relevant entities and processes can be helpful. In particular, automata capture well the intended execution flow of applications, and are thus behind many formal approaches, namely behavioral types.

Typestate-oriented programming allow us to model and validate the intended protocol of applications, not only providing a top-down approach to the development of software, but also coping well with compositional development. Moreover, it provides important static guarantees like protocol fidelity and some forms of progress.

Mungo is a front-end tool for Java that associates a typestate describing the valid orders of method calls to each class, and statically checks that the code of all classes follows the prescribed order of method calls.

To assist programming with Mungo, as typestates are textual descriptions that are terms of an elaborate grammar, we developed a tool that bidirectionally converts typestates into an adequate form of automata, providing on one direction a visualization of the underlying protocol specified by the typestate, and on the reverse direction a way to get a syntactically correct typestate from the more intuitive automata representation.

1 Introduction

Detecting software errors and vulnerabilities is becoming increasingly important in a world where the demand for code development is soaring, often leading to incomplete specifications. Building tools to help achieve this goal is crucial as testing and manual revisions have proven to be insufficient to guarantee software correctness.

Indeed, a carefully designed test suite may detect the presence of many bugs, but it does not guarantee their absence [6]. A widespread practice is the use of programming languages with type systems [2]. These ensure that programs do not present errors for executing invalid operations, but the type of detected errors is quite limited. Day-to-day programmers have to deal with problems that arise from not checking the correct usage of an object. For example, a type system would prevent one from using an undefined method, but not from trying to write on a file prior to opening it.

Stateful objects are non-uniform [9], i.e., their methods' availability depends on their internal state. Behavioral types [8] are notions of types for programming languages representing the possible behavior of an entity, such as an automaton or a state machine. These notions allow us to declare behavioral specifications capturing the availability of methods. For example, a file should first be opened (once),

then it could be written (multiple times, as long as there is space), or read (provided it is not empty), and when its use is finished, it should be closed (once), and cannot be used until opened again.

Textual behavioral specifications, however, can be long and cumbersome. Furthermore, not only does the existing code not use the concept of behavioral types, but it is also quite difficult to define these notions for more elaborate programs. To deal with legacy code, automatic inference tools are needed. Our goal is to enhance the support in the definition of behavioral types in programming languages like Java. We chose the Mungo tool [4], which associates behavioral specifications – *Mungo protocols* or *typestates* – to Java classes and verifies if objects are used correctly.

Given how helpful automata are in abstracting system behavior and assisting developers in understanding the underlying relation between operations, and how useful Mungo typestates are in specifying object behavior and making sure Java code respects such specification, we developed a tool that assists programmers in the (naturally iterative) process of designing an application. The key idea is to support the conception and visualization of (Mungo) typestates as automata.

Specifically, our main contributions are the following.

1. **An automaton model equivalent to Mungo typestates:** we defined a specific automaton model to Mungo protocols; besides allowing a more detailed and accurate graphical representation of the behavioral specification, it simplifies the conversion between typestate and automaton.
2. **A grammar for Mungo typestates:** we defined the full Mungo typestates grammar, providing a formal basis for constructing Mungo protocols. This detailed definition also simplifies the task of describing terms generated by the grammar as automata, given that formal grammars generally have a corresponding abstract machine.
3. **Bi-directional translation between typestates and automata:** we have defined two algorithms that, respectively, translate Mungo protocols into automata by following the productions of the Mungo Typestate Grammar, and produce a Mungo protocol (a term of the grammar) by following all possible execution paths of an automaton.
4. **Implementation and web-based tool:** lastly, we developed a functional implementation of the algorithms, as well as an interactive web-based tool, which allows developers to obtain an automaton from their Mungo protocol, and, conversely, obtain a Mungo typestate from an automaton.

2 How to model an entity as a Mungo protocol?

Suppose we want to build a program describing the behavior of a drone that should be able to take off, move, and land. To describe such system, we have first to understand the ordering between these actions, as it is obvious that the drone must take off prior to moving somewhere, or that it can only land if it is not yet on the ground. We can easily represent this behavior through an automaton (Figure 1), where each state represents those of the drone, and each transition its operations.

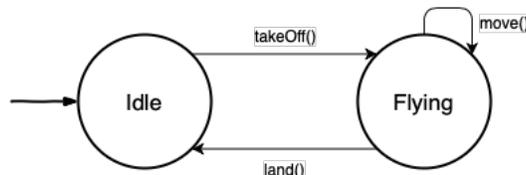


Figure 1: Automaton 1

We now realize the drone should be able to move to a specified destination, but the current specification does not allow this behavior. We change the automaton to include a `moveTo` method instead of `move`, where `x` and `y` specify the coordinates of the desired destination (Figure 2). Additionally, we also add a method `hasArrived` that allows us to check if the drone has arrived at the specified destination.

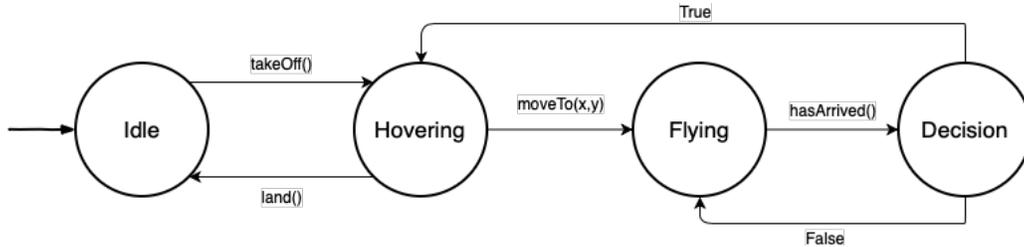


Figure 2: Automaton 2

Notice that state `Decision` is different from the other states: transitions from this node depend on the result of method `hasArrived`, whereas transitions from other nodes are performed by executing methods. On that account, we would like to define a new kind of automaton that correctly renders this idea of states offering *method-call transitions*, and states or *internal-choice states* offering *result-based transitions*. We call *Deterministic Object Automata* (DOA) to this new type of automata, which we will be formalizing in Sec. 3.1. We tweak the previous automaton in order to make these changes apparent, resulting in what you see in Figure 3.

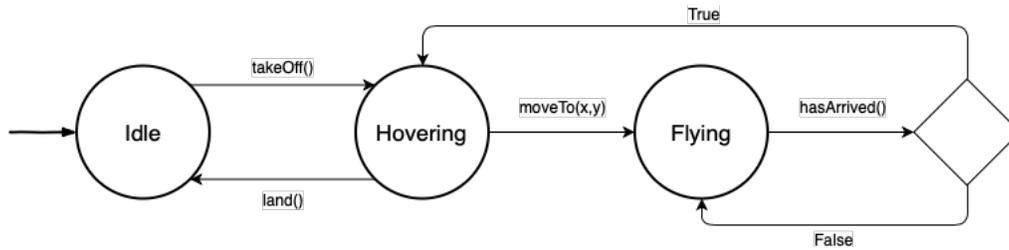


Figure 3: Automaton 3

However easy and intuitive may the representation of system behavior through state machines be, when it comes to implementation, we would be interested in having protocols that verify if, indeed, our code respects the behavioral model we have just designed as an automaton. To do so, one may resort to Mungo [4], a tool used for associating automata-like specifications – *typestates* – with Java classes. These *typestate specifications* abstract the available operations on an object, by defining the sequence of permitted method-calls, which depend on the state of the object [10, 1].

Mungo protocols are defined through a grammar generating DOAs, which can then be associated to a class. It is possible to check if the associated classes are used correctly (according to the *typestate*), since Mungo has a typechecker. If the *typestate* is violated, Mungo reports the errors, otherwise we can securely compile and run the Java code using the standard Java tools.

In Listing 1, we present a Mungo protocol in an attempt to define our automaton representation as a *typestate* that we can use to check if our code respects the intended behavior of the drone.

```

1 typestate DroneProtocol {
2   Idle = { void takeOff(): Hovering }
3   Hovering = { void land(): Idle,
4               void moveTo(double, double): Flying }
5   Flying = { Boolean hasArrived(): <True: Hovering, False: Flying> }
6 }

```

Listing 1: Mungo typestate specifying the drone’s behavior.

Similarly to the automaton we have previously defined, since `Idle` is the first state in our Mungo typestate, a new drone object starts in this state. From here, the only method available is `takeOff`. Calling this method changes the object’s state to `Hovering`, where we can invoke method `land` and go back to state `Idle`, or invoke method `moveTo` and change to state `Flying`. In that state, executing method `hasArrived` has two possible outcomes: if the result is `True`, we change to state `Hovering`; otherwise, we stay in the same state. This mimics the idea of internal-choice states we have previously mentioned. The execution of `hasArrived` takes us to a state that evaluates the result of this method and consequently changes states according to that result.

After drafting the typestate and developing the code for the drone (see appendix A), we now realize that this specification is quite restrictive, as it does not allow changing the drone’s destination nor stop its movement mid-flight. We made the appropriate changes, rewriting the Mungo typestate as in Listing 2.

```

1 typestate DroneProtocol {
2   Idle = { void takeOff(): Hovering }
3   Hovering = { void land(): Idle,
4               void moveTo(double, double): Flying }
5   Flying = { void moveTo(double, double): Flying,
6             void stop(): Hovering,
7             Boolean hasArrived(): <True: Hovering, False: Flying> }
8 }

```

Listing 2: Mungo typestate after changes.

To better understand these changes, it would be helpful to visualize the resulting automaton. By following each step of the Mungo protocol, we are able to draw it (Figure 4). Naturally, it could also be obtained from the previous automaton by adding the corresponding transitions. The code with these modifications can be seen in appendix B.

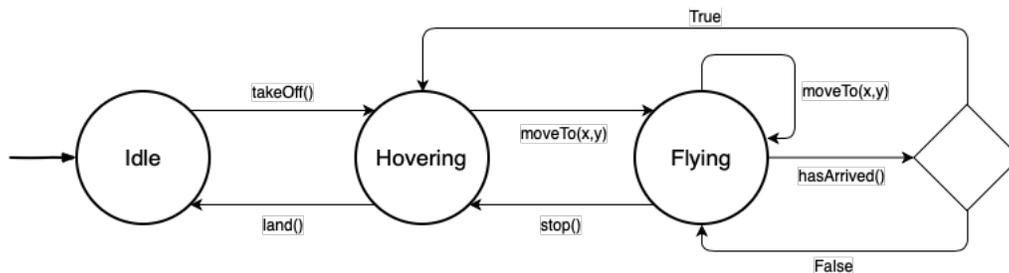


Figure 4: Automaton 4

The process of describing an automaton model as a Mungo typestate, and *vice versa*, can be hard and error-prone, specially when describing complex systems. In the following sections we present the work leading to the development of a tool does this transformation automatically and further allows one to directly change either the automaton or the typestate, and obtain the converse representation, by formalizing the translation between typestate and automaton.

3 Bi-directional conversion between Mungo Tpestates and Deterministic Object Automata

One way of looking at the problem of obtaining a Deterministic Object Automaton from a Mungo typestate (and *vice versa*) is to understand similarities with typical properties regarding Formal Languages and Automata, specifically, translating a finite state automaton from a regular expression (and *vice versa*). Generally, formal grammars have a corresponding state machine, for example, we can abstract regular grammars as finite automata or unrestricted grammars as Turing machines [3, 7]. Therefore, formalizing a grammar for Mungo tpestates naturally simplifies the task of describing these tpestates as abstract machines, helping us better understand the possible behavior of an object.

Having this in mind, the first steps towards defining a conversion algorithm are the formalization of DOA, the automaton model to be equivalent to Mungo tpestates, which we have highlighted in the previous section, and the formalization of a grammar for Mungo tpestates. Finally, we take these two elements and, by studying their structure, define two algorithms that allow for the bi-directional translation of between both grammar and automata.

Since the end goal was to develop a tool that automates the process of translation between tpestate and DOA, one of our concerns while designing such algorithms was to test that they produce the intended results ¹. For this reason, their definition was accompanied by the implementation of equivalent OCaml code ², a natural choice given how simple it is to represent mathematical definitions of recursive functions as functional code.

3.1 Deterministic Object Automata

Communicating automata have been used in the past to model binary session types [5, 11]. However, the model does not fit Mungo usages perfectly, namely it does not distinguish external from internal-choice states, as clearly as DOA do. To have an automata model adequate to Mungo tpestates, namely leading to an equivalence result, we defined the novel approach introduced in Sec. 2. The definition of a new model allows us to have a finer control over its righteous representation of a Mungo tpestate, which in the end is far easier than adapting existing models.

Definition 3.1 (Deterministic Object Automata). A DOA is an octuple $\langle S, T, M, L, s, F, D, E \rangle$ where:

- S is a set of (external choice) states;
- T is a set of (internal choice) states (disjoint from S);
- M is a set of method identifiers;
- L is a set of label identifiers;
- $s \in S$ is the initial state;
- $F \subseteq S$ are the final states;
- $D \subseteq S \times M \times (S \cup T)$ contains the method-call transitions (external);
- $E \subseteq T \times L \times S$ contains all the result transitions (internal).

¹(Mechanized) Formal proofs are work in progress.

²https://github.com/draexlar/compile_doa

Definition 3.2 (Transition function). Let $\Delta = D \cup E$ be the set of transitions, with D ranged over by δ and E ranged over by τ . Consider $x, y \in (S \cup T)$, $a \in (M \cup L)$ and $w \in (M \cup L)^*$.

The function $\Delta^* \subseteq (S \cup T) \times (M \cup L)^* \times (S \cup T)$ is inductively defined by the rules:

- $\Delta^*(x, \varepsilon) = x$;
- $\Delta^*(x, aw) = \Delta^*(\delta(x, a), w)$, if $a \in M$;
- $\Delta^*(x, aw) = \Delta^*(\tau(x, a), w)$, if $a \in L$.

3.2 Mungo Typestate Grammar

Definition 3.3 (Mungo Typestate Grammar). A Mungo Typestate Grammar, G_{Mungo} , is a quadruple $\langle V, \Sigma, P, T \rangle$, where:

- V is a finite set of non-terminal symbols;
- Σ is a finite set of terminal symbols (alphabet), disjoint from V;
- P is a finite set of production rules;
- $T \in V$ is the start symbol.

Let N be a set of strings one can use to name a Mungo protocol, K be a set of all possible state names, R be a set of all possible data types, Y be a set of all possible method names, and Z be a set of all possible label names. Moreover, let $name \in N$ be the name of a typestate, $state \in K$ be any valid Java identifier, $type \in R$ be any Java identifier that points to a Java data type, $method \in Y$ be any valid Java identifier for a method name, and $label \in Z$ be any valid Java identifier.

$$V = \{ T, TB, SDN, SD, S, SN, M, A, AN, W, O, ON, L, LT \}$$

$$\Sigma = \{ \text{typestate, end, } <, >, (,), :, \{, \}, , , = \} \cup \{name\} \cup K \cup R \cup Y \cup Z$$

$$P = \left\{ \begin{array}{lll} T \rightarrow \text{typestate } name \{TB\}; & SN \rightarrow \varepsilon \mid , M SN; & O \rightarrow L ON; \\ TB \rightarrow \varepsilon \mid SDN TB; & M \rightarrow \text{type } method (A) : W; & ON \rightarrow \varepsilon \mid , O; \\ SDN \rightarrow state = SD; & A \rightarrow \varepsilon \mid \text{type } AN; & L \rightarrow \text{label} : LT; \\ SD \rightarrow \{S\}; & AN \rightarrow \varepsilon \mid , \text{type } AN; & LT \rightarrow \text{end} \mid state \mid SD \}; \\ S \rightarrow \varepsilon \mid M SN; & W \rightarrow \text{end} \mid SD \mid < O > \mid state; \end{array} \right.$$

Properties Notice that the grammar's production rules are neither left-regular nor right-regular, since their right-hand side accepts an arbitrary sequence of terminal and non-terminal symbols – $P \subseteq V \times (V \cup \Sigma)^*$ – as seen in the first rule of our grammar. Therefore, this is a context-free grammar.

One of our concerns while defining the grammar we present, was to ensure it to be LL(1) deterministic. This means, it is not ambiguous (produces one leftmost derivation), scans the input from left to right, and is not left-recursive. Furthermore, at each step of a derivation, there is only one applicable rule. It is simple to verify the result was achieved.

3.3 Translating Mungo Typestates into Deterministic Object Automata

Now that we have formalized a grammar for Mungo protocols, we can move on to the task of defining an algorithm for translating Mungo Typestates into Deterministic Object Automata. To this purpose, we have defined a recursive function, *Compile*, which, given a typestate specification, infers the automaton's states and transitions by following the production rules of the Mungo Typestate Grammar, returning a DOA (as defined in Sec. 3.1).

Compile Function. Please recall the sets defined in Sect. 3.2: K , the set of all possible state names; R , the set of all possible data types; Y , the set of all possible method names; and Z , the set of all possible label names. Let $start, next \in K$, $type \in R$, $m \in Y$, $label \in Z$, and let G be the set of available states defined in the tpestate³.

As previously noted, the *Compile* function parses a given tpestate by following the production rules of the defined Mungo Tpestate Grammar. To avoid ambiguity, in cases where there is a recursive production rule, we use an apostrophe (') to distinguish symbols. For example, we denote $TB \rightarrow SDN TB$ as $TB = SDN TB'$.

We begin with the start symbol in our grammar, T , which represents a tpestate. When parsing the tpestate, some symbols may be ignored since they are not relevant for constructing a DOA. For example, even though T contains more information than TB ($T \rightarrow \text{tpestate } name \{TB\}$), we can ignore the symbols *tpestate*, *name*, $\{$, and $\}$, because we are only interested in states and their respective transitions to build an automaton.

$$Compile_G(T) = Compile_G(TB)$$

$$Compile_G(TB) = \begin{cases} \langle \{end\}, \{\}, \{\}, \{\}, end, \{end\}, \{\}, \{\} \rangle & \text{if } TB = \varepsilon \\ Union(Compile_G(SDN), Compile_G(TB')) & \text{if } TB = SDN TB' \end{cases}$$

If the tpestate's body is empty – no states are defined – the object is idle, and therefore the resulting automaton is characterized by having just one initial and final state: *end*, which is defined by default in any Mungo protocol. Otherwise, there is at least one defined state, thus the resulting DOA is the union (which we will define later in this section) of the automaton generated by the state *SDN* and the remainder of the tpestate (TB').

As an example, consider Listing 3 where we define a Mungo tpestate with just two states, one initial and the final state *end*, and with only one method-call transition.

```
1 tpestate basic {
2     begin = { void terminate(): end }
3 }
```

Listing 3: Example tpestate.

According to the algorithm, since the tpestate's body is not empty, we would now have to compute $Union(Compile_G(\text{begin}=\{\text{void } terminate():\text{end}\}), Compile_G(\emptyset))$, with $G = \{\text{begin}\}$. Since the tpestate's body only has one state definition, the remainder of the tpestate is empty. For this reason, we already know that the final union will be with an automaton with just one initial and final state, *end*: $\langle \{end\}, \{\}, \{\}, \{\}, end, \{end\}, \{\}, \{\} \rangle$, resulting from the computation of $Compile_G(\emptyset)$.

Moving on with the definition of *Compile*, when translating a state, if the state's body is empty ($S = \varepsilon$) – no transitions to other states are defined – the resulting DOA only has one state, which is both initial and final. Otherwise, it is necessary to translate the state's transitions, as defined below.

$$Compile_G(SDN) = Compile_G(start, SD) = Compile_G(start, S)$$

$$Compile_G(start, S) = \begin{cases} \langle \{start\}, \{\}, \{\}, \{\}, start, \{start\}, \{\}, \{\} \rangle & \text{if } S = \varepsilon \\ Compile_G(start, M, SN) & \text{if } S = M SN \end{cases}$$

³For the example given in Sect. 2, $G = \{\text{Idle}, \text{Hovering}, \text{Flying}\}$.

Notice the use of *start* instead of the name *state* used in the grammar definition. Further on this document, we will see why this change is helpful but, since $start \in K$, it plays the same role in this instance, being *start* the name of the current state.

Since the state definition from our example is not empty, the next step in the computation of our example would be to compute the result of $Compile_G(\text{begin}=\{\text{void terminate():end}\})$, thus, according to the new step of the algorithm, we now have

$$Union(Compile_G(\text{begin}, \text{void terminate}(), \text{end}), \langle \{\text{end}\}, \{\}, \{\}, \{\}, \text{end}, \{\text{end}\}, \{\}, \{\} \rangle).$$

Continuing where we left off, in the instance of a non-empty state definition, we consider the event of only one transition being allowed – only one method is defined in the current state ($SN = \varepsilon$) – and the event of multiple transitions being allowed ($SN = , M' SN'$). In the latter, the resulting DOA is the union of the automaton generated by the first method transition (M) of the current state, and the one generated by the remainder method transitions ($M' SN'$).

$$Compile_G(\text{start}, M, SN) = \begin{cases} Compile_G(\text{start}, M) & \text{if } SN = \varepsilon \\ Union(Compile_D(\text{start}, M), \\ \quad Compile_G(\text{start}, M', SN')) & \text{if } SN = , M' SN' \end{cases}$$

To compile a method transition, we need only focus on the current state, the method allowing the transition, and the resulting state, ignoring all other symbols:

$$Compile_G(\text{start}, M) = Compile_G(\text{start}, \text{type } m(A), W)$$

$$Compile_G(\text{start}, \text{type } m(A), W) =$$

$$\left\{ \begin{array}{l} \langle \{\text{start}, \text{end}\}, \{\}, \{\text{type } m(A)\}, \{\}, \text{start}, \{\text{end}\}, \{\delta(\text{start}, \text{type } m(A)) = \text{end}\}, \{\} \rangle \\ \quad \text{if } W = \text{end or } W = \{\} \\ \langle \{\text{start}, \text{next}\}, \{\}, \{\text{type } m(A)\}, \{\}, \text{start}, \{\}, \{\delta(\text{start}, \text{type } m(A)) = \text{next}\}, \{\} \rangle \\ \quad \text{if } W = \text{next} \\ Union(Compile_{G \cup \{\text{inner}\}}(\text{start}, \text{type } m(A), \text{inner}), Compile_{G \cup \{\text{inner}\}}(\text{inner}, SD)) \\ \quad \text{if } W = SD, \text{inner} \notin G, \text{inner} \in K \\ Compile_G(\text{start}, \text{type } m(A), O) \\ \quad \text{if } W = \langle O \rangle \end{array} \right.$$

Once again, notice how we use *start* instead of *state*, but also *m* instead of *method* and *next* instead of *state* which, respectively, stand for the current state, the method allowing the transition, and the resulting state. The use of *state* would force us to work with $state = \{\text{type } method(A) : state\}$, which is ambiguous, hence, the use of *start* and *next*, both elements of K . Bear in mind that we are not concerned with which or how many arguments a method accepts, and thus we assume A is well defined.

When the resulting state is the end state – which can be denoted by *end* or $\{\}$ in Mungo protocols – the resulting automaton has: two external-choice states, *start* and *end*; one method, *m*; an initial state, *start*; a final state, *end*; and a (*method-call*) transition from *start* to *end*, through *m*. As you can see, we follow the same reasoning when the resulting state is *next*.

Mungo protocols also allow states to be defined inside other states, similar to an inlining. These “inner” states do not have names, so we need to assign them one. The assigned name must be unique, and thus, before assigning it, one must first check if the name is already in use, by checking if it is in G ⁴.

⁴Recall that G is the set of defined states in the typestate.

We will be referring to the assigned name as *inner* and we may see it as an element of K . Now, we need to update G to include the newly defined inner state, so that, in the future, it will not be possible to create a state with that same name. Consequently, the generated automaton of a transition to an inner state ($W = SD$), is the union of “compiling” the current state with a resulting state *inner*, along with the resulting automaton of compiling the inner state.

Lastly, Mungo protocols allow a state to transition to an *internal-choice state* with a set of options ($W = \langle O \rangle$), through a method m .

According to the last few steps, we can understand that, from where we left off our example, since our state has only one defined transition, and this transition leads to state end, our computation results in solving:

$$\text{Union} \left(\langle \{\text{begin, end}\}, \{\}, \{\text{void terminate()}\}, \{\}, \text{begin}, \{\text{end}\}, \{\delta(\text{begin}, \text{void terminate()}) = \text{end}\}, \{\}, \langle \{\text{end}\}, \{\}, \{\}, \{\}, \text{end}, \{\text{end}\}, \{\}, \{\} \rangle \right).$$

Continuing with the definition of *Compile*, we have still to define the translation of transitions to internal-choice states.

$$\begin{aligned} \text{Compile}_G(\text{start}, \text{type } m(A), O) = \\ \text{Union} \left(\langle \{\text{start}\}, \{\text{choice}\}, \{\text{type } m(A)\}, \{\}, \text{start}, \{\}, \{\delta(\text{start}, \text{type } m(A)) = \text{choice}\}, \{\} \rangle, \right. \\ \left. \text{Compile}_{G \cup \{\text{choice}\}}(\text{choice}, O) \right), \text{ where } \text{choice} \notin G, \text{ choice} \in K \end{aligned}$$

To translate a transition to an internal-choice state, we must first assign a name to that state, which we will be referring to as *choice*, and we may see it as an element of K . Although S (the set of external-choice states) and T (the set of internal-choice states) are disjoint, *choice* should be unique to avoid ambiguity. So, similarly to what was done for the inner states, we check if *choice* is in G , and then update this set. The resulting DOA is the union of an automaton with: one external-choice state, *start*; one internal-choice state, *choice*; a method, m ; an initial state; and an (method-call) transition from *start* to *choice*; along with the resulting automaton of translating the set of options (O) offered by *choice*.

$$\text{Compile}_G(\text{choice}, O) = \text{Compile}_G(\text{choice}, L, ON)$$

$$\text{Compile}_G(\text{choice}, L, ON) = \begin{cases} \text{Compile}_G(\text{choice}, L) & \text{if } ON = \varepsilon \\ \text{Union}(\text{Compile}_G(\text{choice}, L), \text{Compile}_G(\text{choice}, O)) & \text{if } ON = , O \end{cases}$$

Notice that, our grammar does not accept an internal-choice state – a decision state – without options. Hence, we consider the instances of only one option being defined in the current decision state ($ON = \varepsilon$), and the instance of having multiple options ($ON = , O$). We then “compile” the decision state with its first option (or only option). By performing its union with the automaton generated by the the remainder options in *choice*, we satisfy the latter instance.

To compile an option, we focus on the current internal-choice state, *choice*, the option’s label identi-

fier, *label*, and the option's resulting (external-choice) state for the label in consideration, LT:

$$\text{Compile}_G(\text{choice}, L) = \text{Compile}_G(\text{choice}, \text{label}, \text{LT})$$

$$\text{Compile}_G(\text{choice}, \text{label}, \text{LT}) =$$

$$\left\{ \begin{array}{ll} \langle \{\text{end}\}, \{\text{choice}\}, \{\}, \{\text{label}\}, \varepsilon, \{\text{end}\}, \{\}, \{\tau(\text{choice}, \text{label}) = \text{end}\} \rangle & \text{if } \text{LT} = \text{end or } \text{LT} = \{\} \\ \langle \{\text{next}\}, \{\text{choice}\}, \{\}, \{\text{label}\}, \varepsilon, \{\}, \{\}, \{\tau(\text{choice}, \text{label}) = \text{next}\} \rangle & \text{if } \text{LT} = \text{next} \\ \text{Union}(\text{Compile}_{G \cup \{\text{inner}\}}(\text{choice}, \text{label}, \text{inner}), \text{Compile}_{G \cup \{\text{inner}\}}(\text{inner}, \text{SD})) & \text{if } \text{LT} = \text{SD}, \text{inner} \notin G, \text{inner} \in K \end{array} \right.$$

Similarly to what was done for translating method-call transitions, when our resulting state is the end state, the resulting automaton has: one external-choice state, *end*; one internal-choice state, *choice*; one label; a final state; and a (*result-based*) transition from *choice* to *end*, through *label*. Otherwise, when the resulting state is *next*, the resulting automaton has: one external-choice state, *next*; one internal-choice state, *choice*; one label; and a (*result-based*) transition from *choice* to *next*, through *label*.

Mungo typestates also allow states to be defined inside internal-choice states (LT = SD). Therefore, the resulting automaton is the union of translating the current internal-choice state with a resulting state *inner* ($\text{inner} \in K$ and $\text{inner} \notin G$) through *label*, along with the resulting automaton of compiling the inner state. Recall that we then need to update *G* to include the new state *inner*, so that, in the future, no other state can have the same name.

Union of Deterministic Object Automata. At many steps in our *Compile* function we made use of a *Union* function. This is because simply performing the union (\cup) of two Deterministic Object Automata is not right. Although it would be valid for almost every set in our octuple, notice that, in the definition of DOA, there is only one initial state, which is assumed to be the first state defined in a Mungo typestate. Addressing this issue, a function $\text{Union}(a, b)$ is defined.

Definition 3.4 (Union Function). Let *a* and *b* be two DOA, where *a* is the automaton with the initial state considered to be the start:

$$\text{Union}(a, b) = \langle S_a \cup S_b, T_a \cup T_b, M_a \cup M_b, L_a \cup L_b, s_a, F_a \cup F_b, D_a \cup D_b, E_a \cup E_b \rangle$$

Now that the *Union* function is defined, we can now complete the last computation from our example, which we recall bellow.

$$\text{Union} \left(\langle \{\text{begin}, \text{end}\}, \{\}, \{\text{void terminate}()\}, \{\}, \text{begin}, \{\text{end}\}, \{\delta(\text{begin}, \text{void terminate}()) = \text{end}\}, \{\}\rangle, \langle \{\text{end}\}, \{\}, \{\}, \{\}, \text{end}, \{\text{end}\}, \{\}, \{\}\rangle \right)$$

Notice that, in this particular case, the first automaton already has a transition to *end*, therefore the resulting sets will be equal to those of this DOA. Furthermore, by definition, the initial state of the first automaton is the initial state of the resulting automaton, thus we obtain the DOA: $\langle \{\text{begin}, \text{end}\}, \{\}, \{\text{void terminate}()\}, \{\}, \text{begin}, \{\text{end}\}, \{\delta(\text{begin}, \text{void terminate}()) = \text{end}\}, \{\}\rangle$.

3.4 Translating Deterministic Object Automata into Mungo Typestates

We now define the converse translation. By describing the behavior of an entity as a state machine, one should be able to infer its corresponding Mungo protocol. This would be helpful since describing an automaton can be easier and more intuitive than writing a typestate specification.

In this section, we propose a method for translating Deterministic Object Automata into Mungo typestates by defining a function, *Decompile*, that, given a DOA, infers a corresponding Mungo typestate by following the automaton's transitions.

Decompile function. Let *name* be the name one wants to give the resulting Mungo typestate, and $doa = \langle S, T, M, L, s, F, D, E \rangle$ be a Deterministic Object Automaton. And let G be the set of already defined states in the typestate. Initially, $G = \{\text{end}\}$ given that the end state is predefined for every Mungo typestate. The *Decompile* function returns a string corresponding to the Mungo protocol described by *doa*.

We start by giving the typestate's name and the automaton *doa* representing the typestate as arguments of our *Decompile* function, in order to obtain the protocol's header, followed by the definition of its body.

$$Decompile(name, doa) = \text{typestate } name \{ Decompile_G(doa) \}$$

$$Decompile_G(doa) = \begin{cases} \varepsilon & \text{if } S = \emptyset \text{ or } S = \{\text{end}\} \\ s = \{ Decompile_{G \cup \{s\}}(s, doa, A) \} \\ \quad Decompile_{G \cup \{s\}}(\langle S \setminus \{s\}, T, M, L, n, F, D \setminus A, E \rangle), \\ \quad \text{with } A = \{ \delta(x, y) = z \in D \mid x = s \} \text{ and } n \in S \setminus \{s, \text{end}\} & \text{otherwise} \end{cases}$$

Looking at the Mungo Typestate Grammar definition, we see that the typestate's body can be empty ($TB \rightarrow \varepsilon \mid SDN \ TB$). Therefore, if the set of external-choice states, S , of the given DOA is empty or its only element is *end*, there are no (more) states to be defined in the typestate, and this instance of *Decompile* returns the empty string. Otherwise, we define state s in the typestate.

Recall that the first state defined in a Mungo protocol is the initial state of the typestate and, naturally, the initial state of the *doa*, s . For this instance, we also need to define the remaining states of the given automaton. Thus, we make the recursive call of *Decompile* with a DOA where: the set of external-choice states, S , no longer includes s ; its initial state can be any state in $S \setminus \{s, \text{end}\}$; and the set of method-call transitions, D , does not include any transition where the initial state is s . Lastly, it is important that we update the set of defined states in the protocol, G , to include the newly defined state s .

As an example, consider a DOA with just two states, one initial and one final with only one method-call transition:

$$d = \langle \{begin\}, \emptyset, \{\text{void } terminate()\}, \emptyset, begin, \{\text{end}\}, \{(begin, \text{void } terminate(), \text{end})\}, \emptyset \rangle.$$

According to the first steps of *Decompile*, if we called $Decompile(\text{basic}, d)$, we would have $s = begin$, $A = \{(begin, \text{void } terminate(), \text{end})\}$, and $G = \{\text{end}, begin\}$, given that *begin* is the initial state and the only transition beginning in *begin* is that of calling method *terminate*.

Continuing with the formalization of *Decompile*, the body of an external-choice state is the state's method-call transitions. For this reason, besides the current state, c , and *doa*, we also receive the set of all the current state's method-call transitions, A , as follows.

$$Decompile_G(c, doa, A) = \begin{cases} \varepsilon & \text{if } A = \emptyset \\ m : Decompile_G(n, doa) & \text{if } A = \{\delta(c, m) = n\}, m \in M \\ m : Decompile_G(n, doa), \\ \quad Decompile_G(c, doa, A \setminus \{\delta(c, m) = n\}) & \text{if } \#A \neq 1, \{\delta(c, m) = n\} \subset A, m \in M \end{cases}$$

If the set of method-call transitions from c is empty, there are no (more) transitions to be defined for the current state. If the set of method-call transitions from c only has one element, then we need to define the transition relative to that element. Taking a look at our grammar definition, we define a transition by writing the method allowing it, followed by colon ($:$) and the resulting (external-choice) state or a series of choices (internal-choice state).

Otherwise, the set of method-call transitions has multiple elements. We start by choosing one of its elements and defining it, similarly to what was explained in the prior instance, followed by a comma ($,$) and the definition of the remaining transitions allowed in the current state, c . This is done by recursively calling *Decompile* with a set of method-call transitions excluding the one we have just defined.

To finish defining a transition, we need to decide whether the state we are transitioning to, n , is external or internal.

$$Decompile_G(n, doa) = \begin{cases} n & \text{if } n \in S \cup G \\ \langle Decompile_G(n, doa, \{\tau(x, y) = z \in E \mid x = n\}) \rangle & \text{if } n \in T \end{cases}$$

To do so, we need only check if n is in S , the set of external-choice states, or in G , the set of already defined states; or otherwise check if n is in T , the set of internal-choice states. In which case, we start by writing \langle , which identifies the choice; call *Decompile* with the state we are transitioning to, n , the automaton doa , and the set of all result transitions starting in n , which we will be calling B ; followed by closing the choice with \rangle .

$$Decompile_G(c, doa, B) = \begin{cases} l : n & \text{if } B = \{\tau(c, l) = n\}, l \in L, n \in S \cup G \\ l : n, Decompile_G(c, doa, B \setminus \{\tau(c, l) = n\}) & \text{if } \#B > 1, \{\tau(c, l) = n\} \subset B, l \in L, n \in S \cup G \end{cases}$$

In the Mungo Typestate Grammar definition, you can see there must be at least one choice in an internal-choice state ($O \rightarrow L ON$). Therefore, we consider the instance of B (the set of result transitions starting in c) having only one element, and the instance of B having multiple elements.

Taking a look at the grammar definition, in a Mungo typestate, we define a result transition by writing a label, l , followed by colon ($:$), and the resulting external-choice state, n . Since n is an external-choice state, it must be an element of S or an element of G . When having multiple result transitions for c , we start by choosing one of its elements and defining it, as explained above, followed by a comma ($,$), and the definition of the remaining result transitions allowed in the current state. This is done by recursively calling *Decompile* with a set of result transitions excluding the one we have just defined.

From these last few steps of *Decompile* we can see that, given that set A is the singleton $A = \{(begin, \text{void } terminate(), \text{end})\}$ and that, indeed, $\text{end} \in S \cup G$, the resulting typestate from our previous example is the following.

```

1 typestate basic {
2     begin = { void terminate(): end }
3 }

```

Listing 4: Typestate obtained from *d*.

4 The Web Tool

The work presented in the previous section gave us the formal basis to what we sought out to do from the beginning: build a tool ⁵ that helps developers describe behavioral specifications by automating the process of converting Mungo typestates into automata and *vice versa*. This tool allows any user to write a Mungo protocol and preview a visual representation of the corresponding automaton.

The implementation was done using TypeScript ⁶, a static type checker for JavaScript. Although we tried to stick to the functional approach, discussed in the previous section, as to follow the *Compile* and *Decompile* function definitions, some parts of the implementation are imperative in nature, since they work with mutable structures for performance sake. Lastly, to help us with handling the data and the graphical representation of typestates as automata, we used the `vis.js` ⁷ library.

Figure 5 presents the tool’s interface when converting a Mungo protocol into an Deterministic Object Automata. On the left, one can type the typestate specification and click the “Do” button to produce a corresponding visual representation – a DOA – which can be observed on the right. Each element of the image has a meaning: the gray arrow points to the initial state; blue arrows represent transitions; external-choice states are represented as circles; and internal-choice states are represented as diamonds. Finally, one can also download the automaton as a PNG file or copy the automaton in JSON to the clipboard.

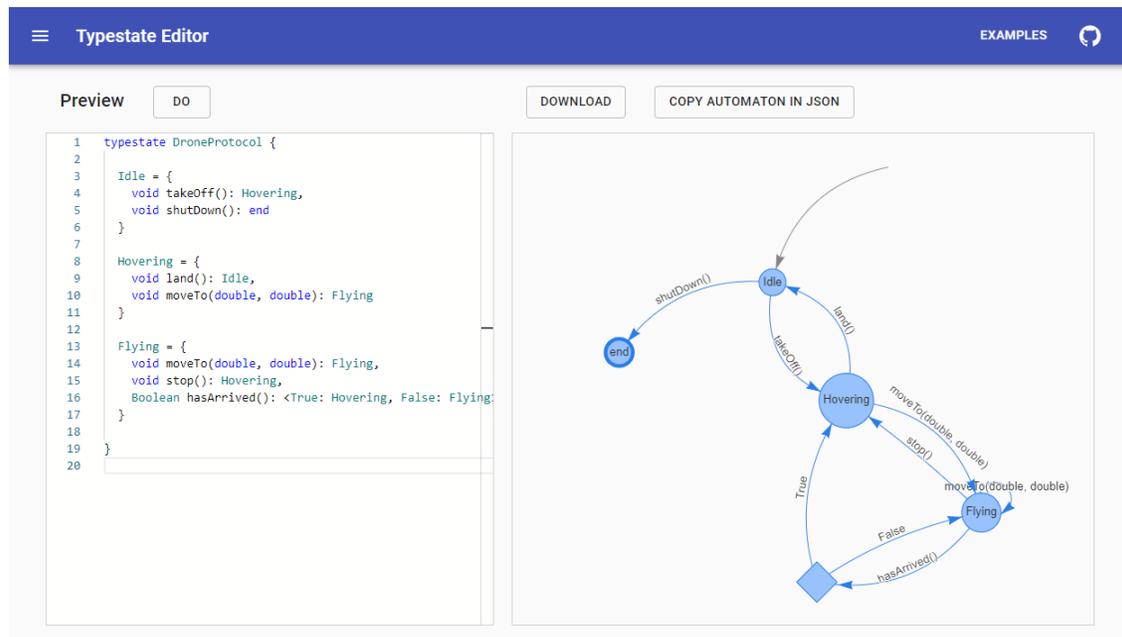


Figure 5: Web tool interface: Mungo typestate to DOA.

⁵<http://typestate-editor.github.io/>

⁶<https://www.typescriptlang.org/>

⁷<https://visjs.org/>

Additionally, the user may view the internal representations and intermediate transformations to understand how the tool works. Namely, the user may transform the typestate into an abstract syntax tree (and *vice versa*), and transform an abstract syntax tree into an automaton (and *vice versa*), both represented in JSON. Figure 6 shows an example of a transformation from an automaton to the abstract syntax tree, which can then be used to obtain the corresponding typestate, as illustrated in Figure 7. Notice how the inverse transformation produced the same typestate initially presented in Figure 5.

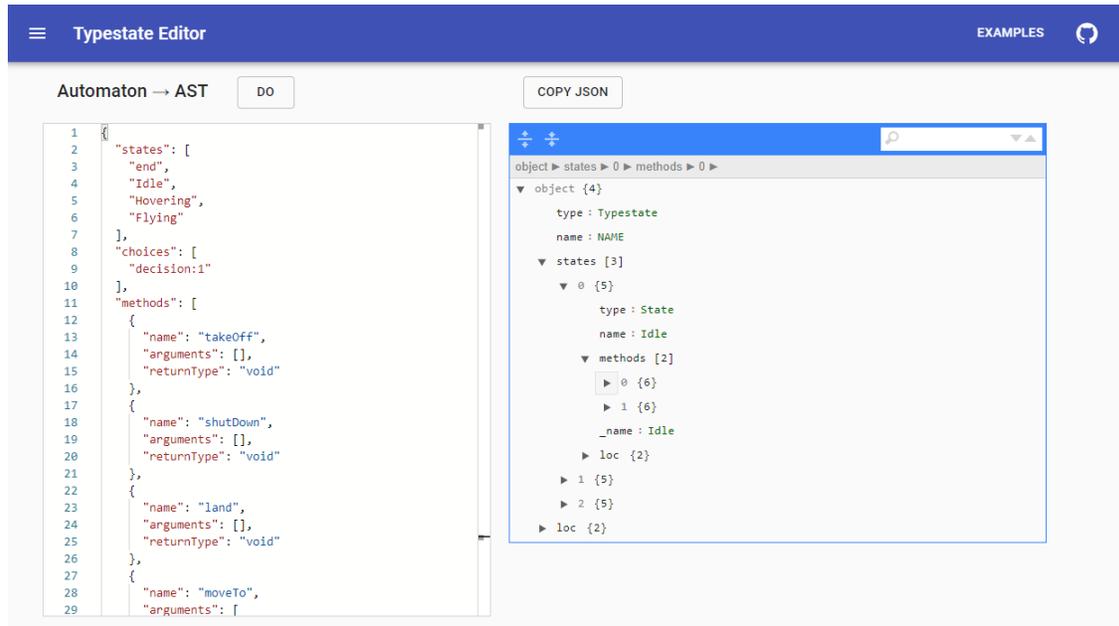


Figure 6: Web tool interface: DOA to AST.

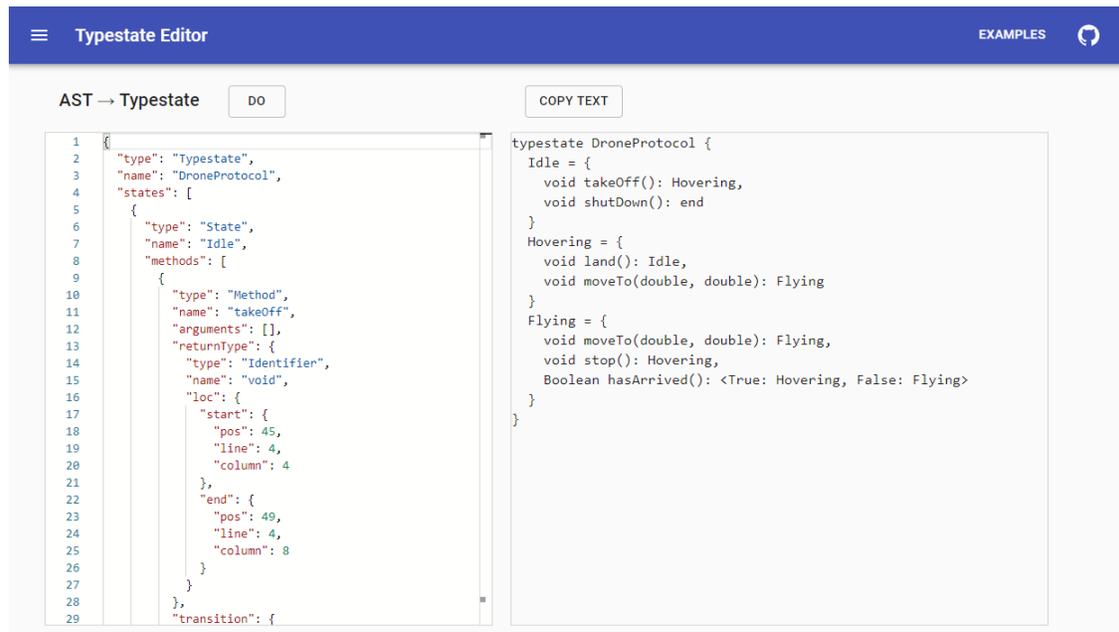


Figure 7: Web tool interface: AST to Mungo typestate.

Lastly, we would like to add that the tool also checks for errors. In the written tpestates, it detects syntax errors and errors related to the structure of the tpestate, such as: states with the reserved name `end`; references to states that are not defined; duplicate states, transitions (for the same state) or labels (in internal-choice states); and internal-choice states without options. When performing the converse translation, we look for errors regarding transitions to undefined states, internal-choice states without result transitions, or even transitions from internal-choice states to internal-choice states.

5 Conclusions and Further work

Nowadays, programming languages rely heavily on type systems to avoid non-trivial errors such as the execution of operations on invalid states, as trying to access a non-existent position of an array or trying to read from a file that is not yet opened. Behavioral types are useful methods to prevent errors resulting from applying operations in wrong order.

In this paper we focus on how tools such as Mungo allow us to describe the behavior of entities through its tpestates. By presenting a formal grammar description of Mungo protocols, we define a norm for building Mungo tpestates, and by designing a new automata model that can soundly portray a Mungo tpestate, we give a concrete way of visualizing the described object's behavior. The formalization of a function for translating Mungo tpestates into Deterministic Object Automata, as well as a function that computes the reverse translation, grants mathematical grounds for developing tools to represent and manipulate behavioral descriptions of computational entities. Such is the web-based tool we have developed, which, through an easy to use graphical interface, allows any developer to obtain a Mungo tpestate from an automaton, or even get a better sense of how their Mungo tpestate is structured, by displaying an equivalent automaton graph. Allied with the Mungo tool, one can associate the protocols designed or obtained through our tool with corresponding Java class implementations, and Mungo's typechecker will verify if the code follows the revised protocols.

Nonetheless, as many have said, *a work of art is never finished*, and so is true for the work herein presented. A common theme throughout this paper, and the motivation driving the use of behavioral types is, in its core, the same as for using other formal methods: the necessity of systems that ensure correct behavior. Therefore, the next step in our work is to ensure the correct specification of our algorithms, by using deductive program verification to prove that the behavior and properties we expect from them are correct. Concretely, the formal proof is based on the typical properties regarding Formal Languages and Automata: the conversion between grammar and automaton preserve the language. This is, to prove the correction of the algorithm that converts Mungo tpestates into DOA is to prove that the *language* of the (resulting) DOA is the same as the *language* of the Mungo tpestate. Similarly, to prove the correction of the translation from DOA to Mungo tpestate is to prove that the *language* of the (resulting) Mungo tpestate is the same as the *language* of the DOA. The result of this work will contribute to the development of a mechanically verified version of the tool that developers can use with the assurance of correct results. Following this step, we intend on improving the usability of the tool, by allowing the users to construct and manipulate the graphical representation of the automaton, rather than having to work with the formal, more complex, textual representation.

References

- [1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini & Zachary Sparks (2009): *Typestate-Oriented Programming*. In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming*

- Systems Languages and Applications*, OOPSLA'09, ACM, p. 1015–1022, doi:10.1145/1639950.1640073.
- [2] Luca Cardelli (1996): *Type systems*. *ACM Computing Surveys (CSUR)* 28(1), pp. 263–264, doi:10.1145/234313.234418.
 - [3] Noam Chomsky (1956): *Three models for the description of language*. *IRE Transactions on Information Theory* 2, pp. 113–124, doi:10.1109/TIT.1956.1056813. Available at <http://www.chomsky.info/articles/195609--.pdf>.
 - [4] Ornela Dardha, Simon J. Gay, Dimitrios Kouzapas, Roly Perera, A. Laura Voinea & Florian Weber (2017): *Mungo and StMungo: tools for typechecking protocols in Java*. In Simon Gay & Antonio Ravara, editors: *Behavioural Types: from Theory to Tools*, River Publishers Series in Automation, Control and Robotics, River Publishers, pp. 309–328, doi:10.1016/j.scico.2017.10.006.
 - [5] Pierre-Malo Denielou & Nobuko Yoshida (2013): *Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types*. In: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, 7966, Springer, p. 174, doi:10.1007/978-3-642-39212-2_18. International Colloquium on Automata, Languages, and Programming (ICALP'13) ; Conference date: 08-07-2013 Through 12-07-2013.
 - [6] Edsger W Dijkstra (1972): *The humble programmer*. *Communications of the ACM* 15(10), pp. 859–866, doi:10.1145/355604.361591.
 - [7] John E. Hopcroft, Rajeev Motwani & Jeffrey D. Ullman (2006): *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., doi:10.5555/1177300.
 - [8] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Denielou, Dimitris Mostros, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira & Gianluigi Zavattaro (2016): *Foundations of Session Types and Behavioural Contracts*. *ACM Comput. Surv.* 49(1), pp. 3:1–3:36, doi:10.1145/2873052.
 - [9] Oscar Nierstrasz (1993): *Regular Types for Active Objects*. *SIGPLAN Not.* 28(10), pp. 1–15, doi:10.1145/167962.167976.
 - [10] Robert Strom & Shaula Yemini (1986): *Typestate: A programming language concept for enhancing software reliability*. *IEEE Trans. Softw. Eng.* 12, pp. 157–171, doi:10.1109/TSE.1986.6312929.
 - [11] Jules Villard (2011): *Heaps and Hops*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France. Available at <http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/villard-phd.pdf>.

A Drone with ability to check its arrival

```

1 tpestate DroneProtocol {
2     Idle = {
3         void takeOff(): Hovering,
4         void shutDown(): end
5     }
6     Hovering = {
7         void land(): Idle,
8         void moveTo(double, double): Flying
9     }
10    Flying = {
11        Boolean hasArrived(): <True: Hovering, False: Flying>
12    }
13 }

```

```

1 import mungo.lib.Typestate;
2 @Typestate("DroneProtocol")
3 public class Drone {
4     public Drone() {}
5     void takeOff() {}
6     void land() {}
7     void moveTo(double x, double y) {}
8     void shutDown() {}
9     Boolean hasArrived() {
10        return Boolean.False;
11    }
12 }

```

We noticed that the previous tpestate was too restricted so, we extended it so that we can check if the drone has arrived to its destination. In the following example, after telling the drone to move, we wait in a loop until the drone arrives. We use a do-while loop, break/continue statements and a switch statement so that Mungo understands the flow of execution. To simplify the code, we introduced a shutDown method so that we reach the end state, making Mungo accept our code – without the need to create another loop around it, similar to the first example.

```

1 public class Main {
2     public static void main(String[] args) {
3         Drone drone = new Drone();
4         drone.takeOff();
5         drone.moveTo(20.0, 10.0);
6         loop: do {
7             switch(drone.hasArrived()) {
8                 case True:
9                     break loop;
10                case False:
11                    continue loop;
12            }
13        } while(true);
14        drone.land();
15        drone.shutDown();
16    }
17 }

```

B Drone with ability to change course while flying

```

1 typestate DroneProtocol {
2     Idle = {
3         void takeOff(): Hovering,
4         void shutDown(): end
5     }
6     Hovering = {
7         void land(): Idle,
8         void moveTo(double, double): Flying
9     }
10    Flying = {
11        void moveTo(double, double): Flying,
12        void stop(): Hovering,
13        Boolean hasArrived(): <True: Hovering, False: Flying>
14    }
15 }

```

This final example is very similar to the previous one. We just introduced the possibility of changing the drone's course by allowing the method `moveTo` to be called in the `Flying` state. This code is also accepted by Mungo.

```

1 public class Main {
2     public static void main(String[] args) {
3         Drone drone = new Drone();
4         drone.takeOff();
5         drone.moveTo(20.0, 10.0);
6         drone.moveTo(10.0, 20.0);
7         loop: do {
8             switch(drone.hasArrived()) {
9                 case True:
10                    break loop;
11                case False:
12                    continue loop;
13            }
14        } while(true);
15        drone.land();
16        drone.shutDown();
17    }
18 }

```