Behavioural description of object-oriented programs

André Trindade

Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa,
Quinta da Torre, Campus Universitário, 2829-516 Caparica, Portugal
adt.trindade@campus.fct.unl.pt

Abstract. When building a program, testing is not enough. There is a need for tools that allow one to reason about the possible behaviour of an entity. At the same time, one needs to be certain those entities are used accordingly. Approaches like behavioural types already proved to be useful. Our contribution focus on the formal description of typestates and their representation. By using abstract machines such as automata to represent the behaviour of an entity, we make it easier to describe and visualize that same behaviour. Offering a method for translating typestates into abstract machines and vice versa, to help developers create new applications or further tune them.

Keywords: Object-oriented programming · Behavioural types · Functional programming · OCaml · Grammar · Automata.

1 Introduction

1.1 Context

Detecting software errors and vulnerabilities is becoming increasingly important in a highly connected and computationally developed world. Building tools to help achieve this goal is crucial as testing and manual revisions are insufficient to guarantee software correctness.

A widespread practice is the use of programming languages with type systems. These ensure that programs do not present errors for executing invalid operations, but the type of detected errors is quite limited.

Stateful objects are non-uniform [1], i.e., their methods' availability depends on their internal state. Behavioural types [2] are notions of types for programming languages representing the possible behaviour of an entity, for example an object, such as an automaton or a state machine. These notions allow us to declare behaviour protocols capturing the availability of methods: for instance, a file should first be opened (once), then it could be written (multiple times, as long as there is space) and/or read (provided it is not empty), and when its use is finished it should be closed (once) and can not be used until opened again.

1.2 Problem

Day-to-day programmers have to deal with problems that arise from not checking the correct usage of an object. As an example, we refer to an issue that arose (recently in the Jedis library) because a "close" method could be called even after a socket timeout[1].

Type systems are beginning to emerge for programming languages such as Java. However, these type systems provide limited help in developing and defining behavioural types. Textual representations, specially when long, can be cumbersome. Furthermore, not only does the existing code not use the concept of behavioural types, but it is also quite difficult to define these notions for more elaborate programs.

The idea is to help the programmer by automatically inferring behavioural types for the code and suggesting them, so that they might adjust the program accordingly. Moreover, it would be useful to provide support for automatically checking their syntactic correctness and transform them in automata, a more intuitive representation.

1.3 Objectives and Contributions

We aim at enhancing the support to the definition of behavioural types in programming languages like Java. We chose the Mungo tool [4, 5], which associates behavioural types to Java classes and verifies if objects are used correctly. It is necessary to analyze the behavioural types used by Mungo, as typestates [3], and, by studying the properties of formal grammars, formalize Mungo protocols and model a coherent abstract machine.

In this paper, we will be mainly focusing on defining the theoretical and formal basis necessary for the work developed simultaneously with João Mota. By analyzing the concept of session types and typestates using Scribble, StMungo and Mungo, we were able to:

1. Formalize Mungo typestates as a grammar, studying its properties as a context-free and LL(1) deterministic grammar;
2. Define an automata model to represent Mungo typestates;
3. Formalize an algorithm for translating Mungo typestates into automata, implementing it as functional code, making exhaustive examples and testing;
4. Formalize an algorithm for translating the defined automata model into Mungo typestates, implementing it as functional code, making examples and testing;

2 From Multiparty Session Types to Typestates

The study of the tools we present in this section falls in the interest of better understanding how, by making a structured description of a communication protocol - specifying the type, sequence and direction of messages - and integrating it with typestates, we are able to describe the behaviour of an object.

[1] `https://github.com/xetorthio/jedis/issues/1747`

2.1 Scribble

Scribble [6] is a language used to describe distributed applications through a multiparty protocol - *global protocol* - which describes the communication between participating parties. It is able to verify if the produced protocol is safe to be implemented, and generates local protocols - *endpoint projections* - for each role (participating party), which describe how the role interacts with other parties.

The following example shows a Scribble global protocol that mimics a travel agency where a client, C, intends to book a flight through an agent, A, which answers the client with the flight's price. Knowing the price, the client contacts the bank, B, to check if they are allowed to perform the transaction. In case of approval, the bank sends an approve code to the client and the agent, also sending the payment to the latter. The agent then sends the invoice and the ticket to the client. In case of refusal, a refuse code is sent to the client and the agent.

```
global protocol Agency(role C, role A, role B) {
    book(Flight) from C to A;
    fare(Price) from A to C;
    check(Price) from C to B;
    choice at B {
        approve(Code) from B to C,A;
        payment(Price) from B to A;
        invoice(Invoice) from A to C;
        ticket(Ticket) from A to C;
    } or {
        refuse(Code) from B to A,C;
    }
}
```

As previously explained, to validate the protocol definition and, for each role, derive its *endpoint projection*, the Scribble tool is used. Each projection describes only the messages involving a specific role.

The following code shows the derived local protocol for the client endpoint.

```
local protocol Agency at C(role C,role A,role B) {
    book(Flight) to A;
    fare(Price) from A;
    check(Price) to B;
    choice at B {
        approve(Code) from B;
        invoice(Invoice) from A;
        ticket(Ticket) from A;
    } or {
        refuse(Code) from B;
    }
}
```

Notice that the exchange of messages not involving the client, such as `payment` are not included.

Now that we have described our distributed application, through a Scribble communication protocol, we pretend to generate a typestate for each entity in our application, thus describing their behaviour.

2.2 StMungo

StMungo [4] is a tool used for translating Scribble to Java. More specifically, it translates Scribble local protocols into Mungo typestate specifications for each role, based on the protocol's message flow. StMungo is then able to abstract each role as a Java class by following its generated typestate protocol, as well as a Java main class corresponding to that same role, which complies with the Mungo protocol. For every `choice at` in a Scribble protocol, StMungo generates a Java class with an enum, where the values are the names of the first message in each branch of the choice.

Considering the previous example, StMungo converts the Scribble local protocol for the C endpoint into:

1. `CProtocol`, a Mungo protocol (typestate specification).
2. `CRole`, a Java class implementing the typestate specification, used as an API to implement the C endpoint.
3. `CMain`, a Java main class with the implementation of the C endpoint, making use of a `CRole` object to communicate with the other roles.

The following code shows the generated Mungo typestate specification, `CProtocol`*, defining the allowed sequences of method calls for the C endpoint.*

```
typestate CProtocol {
    State0 = { void send_bookFlightToA(String): State1 }
    State1 = { int receive_farePriceFromA(): State2 }
    State2 = { void send_checkPriceToB(int): State3 }
    State3 = { Choice1 receive_Choice1LabelFromB():
                    <APPROVE: State4, REFUSE: State7> }
    State4 = { String receive_approveCodeFromB(): State5 }
    State5 = { String receive_invoiceInvoiceFromA(): State6 }
    State6 = { String receive_ticketTicketFromA(): end }
    State7 = { String receive_refuseCodeFromB(): end }
}
```

Note. The generated Mungo protocols and Java code must be slightly adjusted, as they are almost as a template from where the programmer can tune the code to their liking and to what is expected from the application.

2.3 Mungo

Mungo [4, 5] is a tool used for associating typestate specifications with Java classes. These typestate specifications define the sequence of permitted method calls, which depend on the state of the object.

The Mungo protocols (typestate specifications) are defined through a Java-like syntax and then associated to a class. It is possible to check if the associated classes are used correctly, since Mungo has a typechecker [4]. If the typestate is violated, Mungo reports the errors, otherwise we can securely compile and run the Java code using the standard Java tools.

Following our example from the previous section, the typestate specification, `CProtocol`, ensures that a Java object representing the client, in the described multiparty application, respects the expected behaviour (e.g. sending a booking request to the agent before sending the price to the bank).

Since `State0` is the first state in the Mungo protocol, a new client object starts in this state. From here, the only method available is `send_bookFlightToA`. Calling this method, changes the object's state to `State1`, where it can invoke `receive_farePriceFromA` and change to `State2`. But notice `State3`. By calling `receive_Choice1LabelFromB`, we have two possible outcomes: if the bank authorizes the transaction the object transitions to state `State4`; otherwise, the client transitions to state `State7`; naturally influencing the course of the protocol. Other instances worth noting are `State6` and `State7`. By calling their allowed methods, they both reach an `end` state. This means no further interactions with the object are allowed.

The given example only has one accessible method per state. However, Mungo typestates allow states to have more than one method available, depending on what is expected from the object. Nevertheless, only one method may be called at each state, since by computing a method we are also computing a state transition.

We make available the complete code of the examples shown throughout this section[2], including all generated Scribble and Mungo protocols and Java classes, as well as a more detailed view on these examples[3].

3 Mungo Typestate Grammar

With the study of Mungo typestates, specifically, their structure, along with the study of the properties of formal grammars, namely, Regular, Context-free, Context-sensitive and Unrestricted grammars [7–9], we were able to define a grammar for the Mungo typestates.

By formalizing a grammar, we give a formal basis for constructing Mungo protocols (typestates), which may help the user get acquainted with Mungo's syntax, but also a norm to follow when building a typestate.

[2] `https://github.com/draexlar/behaviour-types-research/tree/distributed_agency/src`

[3] `https://goo.gl/dcwWCA`

Formal grammars, generally have a corresponding abstract machine, for example, we can abstract Regular grammars as finite automata or Unrestricted grammars as Turing machines. Therefore, formalizing a grammar for Mungo typestates, also simplifies the task of describing these typestates as abstract machines, helping us better understand the possible behaviour of an entity.

Definition 1 (Mungo Typestate Grammar). *A Mungo Typestate Grammar, G_{Mungo}, is a quadruple (V, Σ, P, T) 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 $\}$

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

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

3.1 Properties

You can see 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, we characterize the Mungo Typestate Grammar as being a context-free grammar.

One of our concerns while defining the grammar we present, was that it 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. For example, a grammar containing a production rule such as $S \to a \mid aB$, cannot be considered LL(1) deterministic.

Shall the reader not be fully acquainted with the concepts of the formal grammars referenced in this section, we make available a document where we explain them and discuss their properties[4].

4 Translating Mungo Typestates into Deterministic Object Automata

Describing systems or entities as state machines can be helpful. It allows one to characterize their condition through states and model their behaviour as a sequence of actions.

Now that we formalized a grammar for Mungo protocols, we intend to describe them as abstract machines. Allowing us to better understand and visualize the behaviour of an object, but also giving a meaningful tool for representing that behaviour.

4.1 Deterministic Object Automata

In order to rightly capture the behaviour of an object, described by a Mungo typestate, we will be using Deterministic Object Automata (DOA), originally defined by A. Ravara.

Definition 2 (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).*

Definition 3 (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:*

[4] `https://goo.gl/rQQfzn`

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

4.2 Compile Function

With the purpose of translating Mungo Typestates into Deterministic Object Automata, we defined a recursive function, *Compile*, which, given a typestate, infers its states and transitions by following the production rules of the Mungo Typestate Grammar, returning a DOA.

Notice that, in our DOA definition, there is a concept of external choice states and internal choice states (decision states), as well as corresponding transitions. Before working on a method for computing the translation, we needed to make sure Mungo typestates also have this level of abstraction.

By looking at the example given in Sect. 2.2, you can see that Mungo expresses a similar behaviour. In `State3`, when calling `receive_Choice1LabelFromB`, we transit to a state where a *decision* takes place, but when in any other state, by calling the method available, we immediately change to a defined state.

For the following definition, please recall the sets defined in Sect. 3: K, a set of all possible state names; R, a set of all possible data types; Y, a set of all possible method names; and Z a set of all possible label names.

Definition 4 (Compile Function). *Let* $start, next \in K$, $type \in R$, $m \in Y$ *and* $label \in Z$. *And let* D *be the set of available states defined in the typestate*[5].

As previously noted, the *Compile* function parses a given typestate by following the production rules of the defined Mungo Typestate 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 → SDN TB as TB = SDN TB'.

We begin with the start symbol in our grammar, T, which represents a typestate.

$$Compile_D(\text{T}) = Compile_D(\text{TB})$$

When parsing the typestate, some symbols may be ignored since they are not relevant for constructing an automaton. For example, although T contains more information than TB (T → typestate *name* { TB }), we can ignore the symbols "typestate", "*name*", "{" and "}", because we are only interested in states and their respective transitions to build a DOA.

$$Compile_D(\text{TB}) = \begin{cases} \varnothing & \textit{if } \text{TB} = \varepsilon \\ Union\big(Compile_D(\text{SDN}),\ Compile_D(\text{TB'})\big) & \textit{if } \text{TB} = \text{SDN TB'} \end{cases}$$

[5] For the example given in Sect. 2.2, D = {`State0`, `State1`, ... , `State7`}

If the typestate's body is empty (TB = ε) - no states are defined - it is not possible to build an automaton. Otherwise - there is at least one state defined (TB = SDN TB') - the resulting DOA is be the union of the automaton generated by the state SDN and the remainder of the typestate (TB').

$$Compile_D(\text{SDN}) = Compile_D(start = \text{SD}) = Compile_D(start = \{\text{S}\})$$

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

Above, is the definition of *Compile* when translating a state of the typestate.

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 $start, state \in K$, they play the same role in this instance, being *start* the name of the current state.

If the state's body is empty (S = ε) - 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.

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

In this instance, we consider the event of only one transition being allowed - only one method is defined in the current state (SN = ε) - 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').

$$Compile_D(start = \{\text{M}\}) = Compile_D(start = \{type\ m(\text{A}) : \text{W}\})$$

$$Compile_D(start = \{type\ m(\text{A}) : \text{W}\}) = \begin{cases} \big\langle \{start, \text{end}\}, \{\}, \{type\ m(\text{A})\}, \{\}, \\ \quad start, \{\text{end}\}, \{\delta(start,\ type\ m(\text{A})) = \text{end}\}, \{\} \big\rangle & \textit{if } \text{W} = \text{end } \textit{or } \text{W} = \{\} \\ \big\langle \{start, next\}, \{\}, \{type\ m(\text{A})\}, \{\}, \\ \quad start, \{\}, \{\delta(start,\ type\ m(\text{A})) = next\}, \{\} \big\rangle & \textit{if } \text{W} = next \\ Union\big(\ Compile_D(start = \{type\ m(\text{A}) : inner\}), \\ \qquad Compile_D(inner = \text{SD})\ \big) & \textit{if } \text{W} = \text{SD},\ inner \notin D, \\ & \quad inner \in K, \\ & \quad D = \{inner\} \cup D \\ Compile_D(start = \{type\ m(\text{A}) : < \text{O} > \}) & \textit{if } \text{W} = < \text{O} > \end{cases}$$

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

Once again, 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 $state = \{type\ method(\mathrm{A}) : state\}$, 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 as "end" or "{}", in the 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 an (external) transition from *start* to end, through m.

Similarly, when the resulting state is *next* ($\mathrm{W} = next$), the resulting automaton has: two external choice states, *start* and *next*; one method, m; an initial state, *start*; and an (external) transition from *start* to *next*, through m.

Mungo protocols also allow states to be defined inside other states. 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 D. 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 D 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 ($\mathrm{W} = \mathrm{SD}$), is the union of "compiling" the current state with a resulting state *inner*, along with the resulting automaton of compiling the inner state. An example of inner states in Mungo protocols can be seen in Appendix A.

Lastly, Mungo protocols allow a state to transition to an internal choice state with a set of options ($\mathrm{W} = < \mathrm{O} >$), through a method m.

$$
\begin{aligned}
&Compile_D(start = \{type\ m(\mathrm{A}) : < \mathrm{O} > \}) = \\
&\quad Union\Big(\big\langle \{start\}, \{choice\}, \{type\ m(\mathrm{A})\}, \{\}, start, \{\}, \\
&\qquad \{\delta(start,\ type\ m(\mathrm{A})) = choice\}, \{\}\big\rangle,\ Compile_D(choice,\ \mathrm{O})\Big), \\
&\qquad\qquad where\ \ choice \notin D,\ choice \in K,\ D = \{choice\} \cup D
\end{aligned}
$$

To translate a transition to an internal choice state, we 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 D, 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 (external) transition from *start* to *choice*; along with the resulting automaton of translating the set of options (O) offered by *choice*.

$$Compile_D(choice,\ \mathrm{O}) = Compile_D(choice,\ \mathrm{L}\ \ \mathrm{ON})$$

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

Notice that, our grammar does not accept an internal choice state (decision state) with no options. Hence, we consider the instances of only one option being defined in the current decision state ($\mathrm{ON} = \varepsilon$), and the instance of having multiple options ($\mathrm{ON} = \ , \mathrm{O}$). We then "compile" the decision state with its first option (or only option). By making its union with the automaton generated by the the remainder options in *choice*, we satisfy the latter instance.

$$Compile_D(choice,\ \mathrm{L}) = Compile_D(choice,\ label:\ \mathrm{LT})$$

$$Compile_D(choice,\ label:\ \mathrm{LT}) = \begin{cases} \langle \{\mathrm{end}\}, \{choice\}, \{\}, \{label\}, \epsilon, \{\mathrm{end}\}, \{\}, \{\tau(choice, label) = \mathrm{end}\} \rangle \\ \qquad \text{if } \mathrm{LT} = \mathrm{end} \text{ or } \mathrm{LT} = \{\} \\ \langle \{next\}, \{choice\}, \{\}, \{label\}, \epsilon, \{\}, \{\}, \{\tau(choice, label) = next\} \rangle \\ \qquad \text{if } \mathrm{LT} = next \\ Union\big(Compile_D(choice,\ label : inner),\ Compile_D(inner = \mathrm{SD})\big) \\ \qquad \text{if } \mathrm{LT} = \mathrm{SD},\ inner \notin D,\ inner \in K,\ D = \{inner\} \cup D \end{cases}$$

To compile an option, we focus on the current internal choice state (*choice*), the option's label identifier (*label*) and the option's resulting (external choice) state for the label in consideration (LT).

Similarly to what was done for translating method transitions, when our resulting state is the end state ($\mathrm{LT} = \mathrm{end}$ or $\mathrm{LT} = \{\}$), the resulting automaton has: one external choice state, end; one internal choice state, *choice*; one label; a final state; and a result transition from *choice* to *end*, through *label*.

Otherwise, when the resulting state is *next* ($\mathrm{LT} = next$), the resulting automaton has: one external choice state, *next*; one internal choice state, *choice*; one label; and a result transition from *choice* to *next*, through *label*.

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

4.3 Automata Union

In the previous section, we made use of a $Union$ function to define the $Compile$ function. This is because, simply making 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 Deterministic Object Automata, there is only one initial state, which is assumed to be the first state defined in a Mungo protocol. Addressing this issue, a function $Union(a, b)$ is defined.

Definition 5 (Union Function). *Let a and b be two DOA, where a is the automaton with the initial state equal to the first state of a typestate:*

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

The union with {end} is necessary since the "end" state is predefined for every Mungo typestate, even if there is no transition to it from the defined states.

4.4 Implementation

In order to see the *Compile* function in action and verify that it works as intended, we made an OCaml implementation. The choice for this programming language is obvious, allowing us to directly translate the *Compile* function into ML code. We make available the complete functional code[6], including the examples used for testing, along with an in depth explanation of its stages and considered errors[7].

5 Translating Deterministic Object Automata into Mungo Typestates

By describing the behaviour of an entity as a state machine, one should be able to infer its corresponding Mungo typestate. This would be helpful since describing an automaton can be easier and more intuitive than defining a typestate.

In this section, we propound a method for translating Deterministic Object Automata into Mungo typestates, so that, by modeling an automaton of this kind, one can easily obtain a Mungo protocol.

5.1 Decompile Function

For the following definition, please recall the definition of Deterministic Object Automata introduced in Sect. 4.1.

Definition 6 (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.*

The Decompile function returns a string corresponding to the Mungo typestate described by the given DOA.

[6] `https://github.com/draexlar/compile_doa`
[7] `https://goo.gl/NwD11q`

We start by giving the typestate's name and the DOA representing the typestate, as arguments of our *Decompile* function.

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

Now that we already have the "header" of our typestate, we need to define the typestate's states - the typestate body.

$$Decompile(doa, G) =$$
$$\begin{cases} \varepsilon \quad \textit{if } S = \emptyset \textit{ or } S = \{\text{end}\} \\ s = \{\ Decompile(s, doa, A, G')\ \} \\ \qquad Decompile\big(\langle S \setminus \{s\}, T, M, L, n, F, D \setminus A, E\rangle, G'\big) \\ \quad \textit{if } S \neq \emptyset,\ n \in S \setminus \{s, \text{end}\},\ A = \{\delta(x, y) = z \in D \mid x = s\},\ G' = G \cup \{s\} \end{cases}$$

Before further explanation, notice that we receive a set G as argument. This is the set of already defined states in the Mungo protocol. Recall that the "end" state is predefined for every Mungo typestate so we are not concerned with its definition, hence $G = \{\text{end}\}$, initially.

Looking at the Mungo Typestate Grammar definition, we see that the typestate's body can be empty (TB $\rightarrow \varepsilon$ | 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 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. Therefore, we make the recursive call of *Decompile* with a DOA where: the set of external choice states no longer includes s; its initial state can be any state in $S \setminus \{s, \text{end}\}$; and the set of method-call transitions 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, to include the newly defined state s.

The body of an (external choice) state is the state's method-call transitions. For this reason, besides the current state, c, and the DOA, we also receive the set of all the current state's method-call transitions, A, and the set of already defined states in the protocol, G, as argument.

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

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 state (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 just defined.

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

$$Decompile(n, doa, G) =$$
$$\begin{cases} n & \text{if } n \in S \text{ or } n \in G \\ < Decompile(n, doa, \{\tau(x, y) = z \in E \mid x = n\}, G) > & \text{if } n \in T \end{cases}$$

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.

As previously stated, an internal choice state is described as a series of choices, in a Mungo protocol. To help us achieve this result, we call *Decompile* with the state we are transitioning to, n, the DOA we are translating, the set of all result transitions starting in n, and the set of defined states.

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

In the Mungo Typestate Grammar definition, you can see that 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 protocol, we define a result transition by writing a label, l, followed by colon (:) and the resulting external choice state, n, of the transition allowed by that label. Notice that, since n is an external choice state, it must be an element of S, the set of external choice states in *doa*, or an element of G, the set of defined states in the typestate.

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 just defined.

5.2 Implementation

Similarly to what was done for the *Compile* function, we have implemented an OCaml program based in our definition of the *Decompile* function. This allows us to verify if the function has the intended behaviour. We make available the complete code[8], as well as an in depth look at its stages and possible errors[9].

6 Conclusions

Nowadays, programming languages rely on type systems to avoid errors such as the execution of invalid operations but, as programmers, in our day-to-day life, when building a program or application, we many times come across other types of errors such as trying to access a none existing position of an array or trying to read from a file that is not yet opened. The concept of behavioural types came to light with the intention to fill this gap, defining typestates that are able to describe the possible and expected behaviour of an entity.

In this paper we focus on understanding the philosophy behind behavioural types, and how tools such as Mungo allow us to describe the behaviour of entities through its typestates. We also explore this concept allied with session types in communication-based applications, showing that behavioural types have relevance in today's reality where communication dominates modern computing.

By presenting a formal grammar description of Mungo protocols, we define a norm for building Mungo typestates. We also present a new kind of automata that can rightly portray the behaviour of an entity, thus giving a concrete way of visualizing the described entity's behaviour. We formalize a function for translating Mungo typestates into Deterministic Object Automata, as well as a function that computes the reverse translation, hence giving mathematical grounds for developing tools that work upon this subject. Lastly, we have implemented these functions in OCaml code, proving their behaviour and intended use.

The work presented herein, specifically, the translation between automata and typestates, is incredibly helpful in describing the behaviour of a system or entity. It allows users to concretely visualize all the stages of the described object, possibly further tuning them to their liking, and obtaining a typestate as a result.

[8] `https://github.com/draexlar/compile_doa`
[9] `https://goo.gl/MFkUkF`

7 Acknowledgements

This research was done during the course of APDC under supervision of Professor António Ravara.

References

1. Nierstrasz, O.: Regular Types for Active Objects. SIGPLAN Not. vol. 28, no. 10, pp.1–15. ACM, New York, NY, USA (1993) https://doi.org/10.1145/167962.167976
2. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., and Zavattaro, G.: Foundations of Session Types and Behavioural Contracts. ACM Comput. Surv. vol. 49, **1**(3), pp.1–36. ACM, New York, NY, USA (2016) https://doi.org/10.1145/2873052
3. Strom, R.E., and Yemini, S.: Grid Typestate: A programming language concept for enhancing software reliability. In: IEEE Transactions on Software Engineering, vol. SE-12, no. 1, pp.157–171. (1986) https://doi.org/10.1109/TSE.1986.6312929
4. Dardha, O., Gay, S.J., Kouzapas, D., Perera, P., Voinea, A.L., Florian, W.: Mungo and StMungo: Tools for Typechecking Protocols in Java. In: Simon, G., Ravara, A. (eds.) Behavioural Types: from Theory to Tools, pp. 309–328. River Publishers, Denmark (2017).
5. Mungo Homepage, `http://www.dcs.gla.ac.uk/research/mungo/index.html`. Last accessed 19 Jun 2018
6. Scribble Homepage, `http://www.scribble.org/index.html`. Last accessed 19 Jun 2018
7. Serenadas, C.: Introdução à Teoria da Computação. (Portuguese) [Introduction to the Theory of Computation]. 2nd edn. Editorial Presença, Lisbon (1992)
8. Hopcroft, J.E., Rajeev, M., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. 3rd edn. Publisher, Addison-Wesley (1979)
9. Chomsky, N.: Three models for the description of language. IRE Transactions on Information Theory **2**(3), 113–124 (1956) https://doi.org/10.1109/TIT.1956.1056813

A Mungo protocol with inner state

In the code below, when calling the `connect()` method, the state `Init` transitions to an external choice state, but notice that the resulting state is defined inside `Init`.

```
typestate AliceProtocol {
    Init = {
        void connect(): {
            String recvStringFromBob(): ReceiveChoice
        }
    }
    ReceiveChoice = {
        BobChoice choiceFromBob() :
            <TIME: SendTime, GREET: HowAreYou>
    }
    SendTime = {
        void sendTimeToBob(int) : EndProtocol
    }
    HowAreYou = {
        void sendGreetToBob(String) : EndProtocol
    }
    EndProtocol = {
        void endCommunication() : end
    }
}
```

(*Example adapted from* `https://bitbucket.org/abcd-glasgow/mungo/src/e83ac35cc4be635ca095cff11e8e1d6f98fca77c/examples/TwoParties/AliceProtocol.protocol`)