Elixir code generation from TLA$^+$ specifications

Gabriela Mafra

June 29, 2021

/ 1 / Abstract

Specifying systems before writing code is a good planning practice because it allows early validation and modifications. Using a formal specification method to do so requires extra work, but provides the benefit of automatically verifying the specification against desired properties, helping to prevent unpredicted behaviors. TLA$^+$ (Temporal Logic of Actions$^+$) is a formal specification language focused on concurrent systems, where behaviors can be hard to predict. However, a TLA$^+$ specification alone cannot produce its own implementation, since there is currently no tool to translate it into a usable artifact. This work presents a code generation tool to create executable Elixir code from a TLA$^+$ specification and demonstrates its usage.

/ 2 / Overview

Generating code from a TLA$^+$ specification can reduce the gap between design and implementation. For companies where the value of a formal specification doesn't seem enough to justify the time spent on writing it, adding an executable artifact that can be used as a proof of concept to the list of assets produced by the spec can be enough to account for the time invested. The code generation method presented here aims to produce readable code that a programmer can find correspondence to the TLA$^+$ spec without issue.

Taking into account that many TLA$^+$ specs are written in a high abstraction level and would not produce a useful code alone, the generated code should be very pluggable in a way where unspecified details can be resolved by different modules. Interfacing between deterministic and non-deterministic behaviors is done by a mechanism I called oracle.

Considering a simple TLA$^+$ spec for traffic semaphores as

MODULE TrafficSemaphore

VARIABLE color

$\text{TurnRed} \triangleq \text{color} = \text{“yellow”} \land \text{color}' = \text{“red”}$
$\text{TurnYellow} \triangleq \text{color} = \text{“green”} \land \text{color}' = \text{“yellow”}$
$\text{TurnGreen} \triangleq \text{color} = \text{“red”} \land \text{color}' = \text{“green”}$

$\text{Init} \triangleq \text{color} \in \{\text{“red”}, \text{“yellow”}, \text{“green”}\}$

$\text{Next} \triangleq \text{TurnRed} \lor \text{TurnYellow} \lor \text{TurnGreen}$

Figure 1: TLA$^+$ Specification for the traffic semaphore

an action such as TurnRed can be translated to two elixir functions: one representing the condition to enable that action (color in the current state is yellow) and other representing the transition to the next state (color will be red).

```
def turn_red_condition(variables) do
  variables[:color] == "yellow"
end

def turn_red(variables) do
  %{ color: "red" }
end
```

The next state action is translated to a main function that, for each form of disjunction, will find enabled actions. If no action is enabled, there is a deadlock and the process stops. If exactly one action is enabled, it obtains the next state from that action's transition. And if more than one action is found, it lists all enabled actions and sends them to an oracle - which is another process on them BEAM (Erlang's virtual machine) - waits for a response with a choice and executes the chosen action. This oracle can be any implementation that respects the messaging interface, and some examples are: an I/O oracle which reads choices from console input, useful for debugging; and a random oracle which chooses randomly, like a simulation.

In a paper submitted for this year's SBLP (Brazilian Symposium on Programming Languages), pending review, the translation rules are further explained and applied to a real-world use-case by generating code for a pump-station specification and running it with an MQTT oracle that reads data from (simulated) sensors to decide which actions to take. The submitted version of this paper can be found in a different attachment.