Backwards Compatible Automata

José Duarte

September 4, 2023

Theory-ish

We will start by defining one of the simplest state machines — a light bulb. Using the classical model, the light bulb automata can be defined as follows:

$$(\Sigma, S, s_0, \delta, F) = (\{click\}, \{\text{Off}, \text{On}\}, \text{Off}, S \times \Sigma \to S, \text{Off}) \tag{1}$$

However, for our purposes we do not need to care for the start nor final states, for the sake of brevity we will remove them from our automata definitions going forward, as well as the transition function (which is implicitly defined by stating that our state machines must be DFA). Hence, we can express our light bulb as follows:

$$(\Sigma, S) = (\{click\}, \{\text{Off}, \text{On}\}) \tag{2}$$

Which we can observe in Figure 1. The bulb has two states — *On* and *Off* — both of which transition on the application of the symbol click.

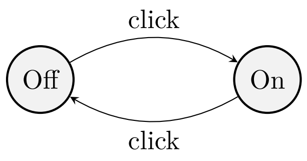

Figure 1: Light bulb FSM

Let there be light

Consider now that we are tasked with changing the state from carrying a boolean to a number, describing the current light bulb intensity. Maybe in the future, the manufacturer wants the system to work with potentiometers. To that end, we change the state machine definition from Equation 2 to the following (Equation 3 and Figure 2):

$$(\Sigma, S) = (\{click\}, \{0, 1\}) \tag{3}$$

At first sight, this is not a problem, in code we can just replace the old boolean for an integer, this would be fine if every node of the swarm could be required to update, but that is not the case. To cope with said problem, we

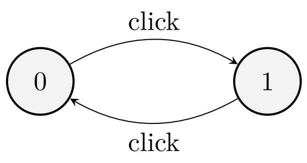

Figure 2: Light bulb FSM with Light intensity

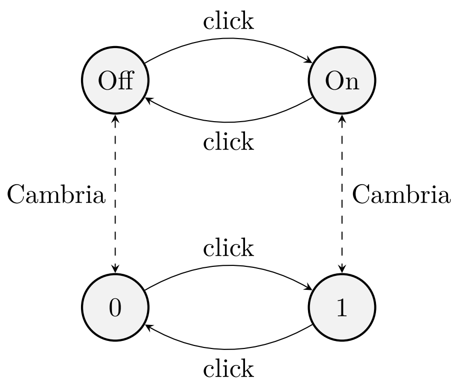

Figure 3: Light bulb FSM with Cambria state mapping

need to map the old states to the new ones, using Cambria [1] we can convert state information (see Figure 3).

Along with state conversion, we can extend Cambria to event labels (see Figure 4), which is arguably an even better match since the labels (or payloads for more involved cases) are what cross the network boundary.

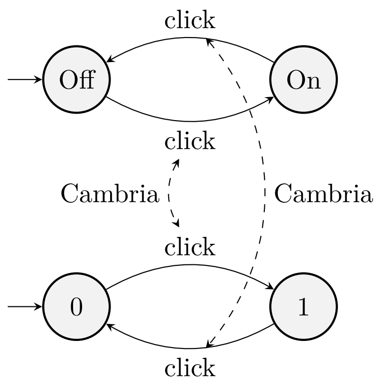

Figure 4: Light bulb FSM with Cambria event mapping

There is still an issue left to address: *How does the old state machine know about new transformations?* — to which the answer is simple — *It does not know. It cannot know.*

If the old machine were able to get the new transformation, it would mean that it could get the new state machine as well, so we need to assume it cannot

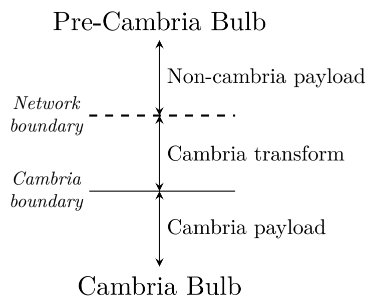

Figure 5: Asymmetric interaction between two bulbs, using Cambria to mediate data between them.

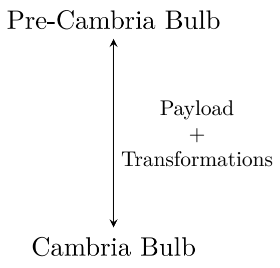

Figure 6: Asymmetric interaction between two bulbs, using Cambria to mediate data between them.

get the new Cambria transformation.

Thus, we are dealing with an asymmetric scenario where to keep the system running, the most up-to-date system needs to pick up the slack from the older participants. To do so, the up-to-date system can either pre-process all information going in and out (as shown in Figure 5) or require that all participants are at least capable of running arbitrary Cambria transformations and exchange data with the information of which transformations to apply along with the data (as shown in Figure 6).

Proper transforms now

Once again, consider the light bulb using the intensity as states, imagine that a new change was requested — instead of a toggle, we are now using a button that can switch intensity from 0, to 0.5, to 1 and back to 0 (see Equation 4).

$$(\Sigma, S) = (\{click\}, \{0, 0.5, 1\}) \tag{4}$$

Usually, we would first design the new state machine (see Figure 7) code it and ship it off as an update to the existing software, replacing the previous state machine. However, as previously discussed, our system cannot afford the luxury of keeping everyone's version in sync, so, if we want to support the previous version participants, we need to keep processing their messages and moving the state machine along.

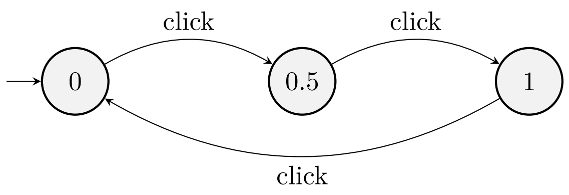

Figure 7: Light bulb with support for multiple intensities (as defined in Equation 4).

To support the previous state machine, we start by merging both state machines, in a more formal way, we can define $merge(M_1, M_2)$ (where M_1 and M_2 are automata as defined in Equation 2) as:

$$merge(M_1, M_2) = (M_1.\Sigma \cup M_2.\Sigma, M_1.S \cup M_2.S) \tag{5}$$

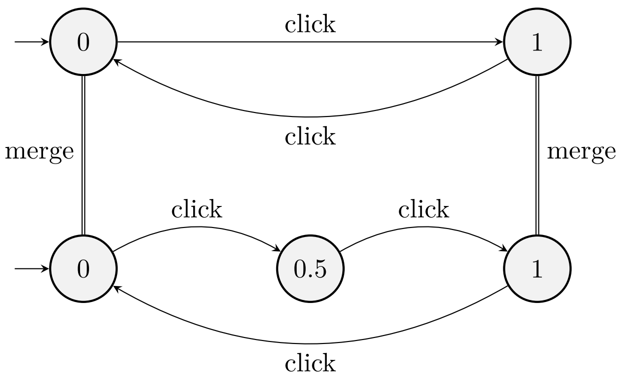

Figure 8: Visual approximation of $merge(M_1, M_2)$, where M_1 is the state machine from Figure 1 and M_2 is the state machine from Figure 2.

If we execute *merge* to the state machines from Equation 3 and Equation 4 we get:

$$\begin{aligned} merge(M_1, M_2) &= (\{click\} \cup \{click\}, \{0, 1\} \cup \{0, 0.5, 1\}) \\ &= (\{click\}, \{0, 0.5, 1\}) \end{aligned} \tag{6}$$

Which raises a problem that is not made apparent by the textual representation, but when displayed visually (see Figure 9) it becomes much more apparent — notice two outgoing edges from state *0* with the same *click* label; the state machine is now non-deterministic.

We can try to make it deterministic, however since we have a single event label and our states' meaning is different from string matching, determinization will yield a single state with a single loop labeled *click* (considering we do not have an accepting state).

The proposed solution is to label edges based on their version, thus removing the "collision" when merging. If we prefix (or suffix, writers choice) symbols with their version we are able to preserve the original states and keep the state machine deterministic as shown in Figure 10.

$$\begin{aligned} merge(M_1, M_2) &= (\{v1.click\} \cup \{v2.click\}, \{0,1\} \cup \{0, 0.5, 1\}) \\ &= (\{v1.click, v2.click\}, \{0, 0.5, 1\}) \end{aligned} \tag{7}$$

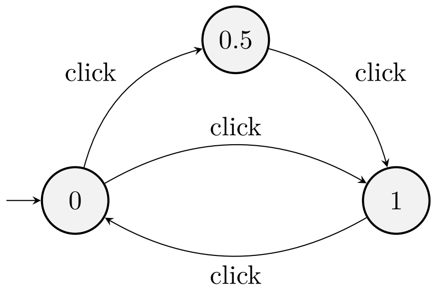

Figure 9: $merge(M_1, M_2)$ results in an NFA (Equation 5).

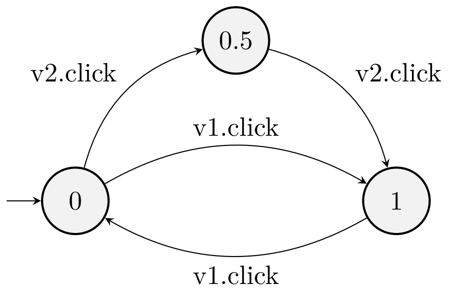

Figure 10: $merge(M_1, M_2)$ with the new labels preserves the DFA (Equation 7).

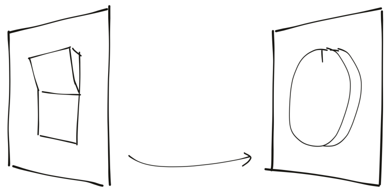

Figure 11: The interface to the bulb may change, requiring new capabilities.

In a regular situation, we would ship this off as an update, however, some systems cannot guarantee that everyone has updated, thus we are looking to keep backwards compatibility.

To do that, we need to once more, consider the old payloads and states. We could defer any old payload to the old state machine and create mechanisms to map between both, however, the new machine did not appear out of nowhere, but rather, it is an evolution of the original one, so we should treat it as such.

We start by merging the common states and transitions between them.

This way, we have a machine that can handle both cases; almost.

If we execute a trace: 0 v2 0.5 v1 ? We don't know the final state because it is undefined. To achieve backwards compatibility we need to address this since the client needs to be able to send actions to the system.

Theorem: When merging state machines, all new states must have outgoing edges for all existing transition labels.

In other words, states from version vn+1 are required to handle all states from vn or less.

States cannot be sinks, meaning that they always need to have at least one outgoing edge (unless they're final states of course).

Merging is defined by the union of states and transition labels from both the old and new state machine.

If we now want to be able to go back and forth arbitrarily, that is from 0 to 0.5 and back to 0. Or from 0 to 0.5 to 1 to 0.5, we need to add more events.

To keep backwards compatibility we cannot delete existing parts, but we can create new paths our previous rule still applies and as such, every new state will be required to handle all previous events.

Unsure: if starting at the initial state, a path is not reachable with older labels, we can effectively create ”feature gates” that stop a user from reaching a weird state. Feature gates can be done by ignoring previous states (not truly ideal but a valid fallback).

Runtime-ish

The state machine must be executed in some type of runtime, which is required to support a plugin mechanism, be it web assembly modules or OS DLLs.

The way new capabilities are added is additive, in the sense that the state machines are defined for a single version and merged later (in a pseudo compilation phase that ensures that the current version is compatible with the previous one and obeys the rules).

When creating a new version, the user shouldn't care to merge automatically, but should care about the backwards compatible transitions. The new state machine is described and an automatic checker should validate if both are compatible, only then is can be ”compiled”.

Modules should either be additive, meaning that a new module carries only the delta of changes; or they should be compiled with all previous versions and shipped in a single module.

References

[1] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. Project cambria, Oct 2020.