Backwards Compatible Automata

José Duarte Roland Kuhn

September 12, 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 fig. 1. The bulb has two states — *On* and *Off* — both of which transition on the application of the symbol click.

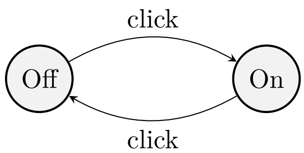

Figure 1: Light bulb FSM

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 eq. (2) to the following (eq. (3) and fig. 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

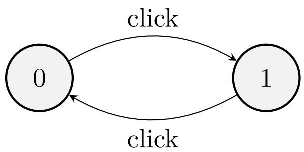

Figure 2: Light bulb FSM with Light intensity

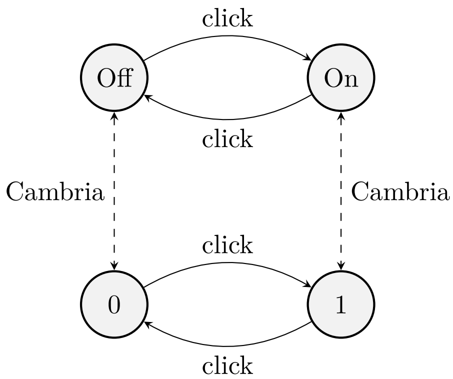

Figure 3: Light bulb FSM with Cambria state mapping

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

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

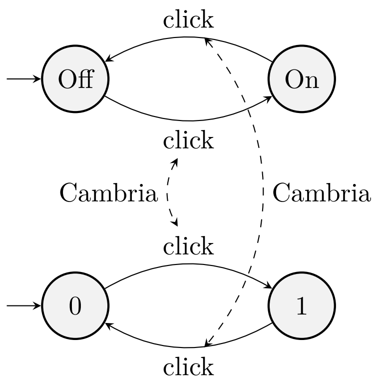

Figure 4: Light bulb FSM with Cambria event mapping

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

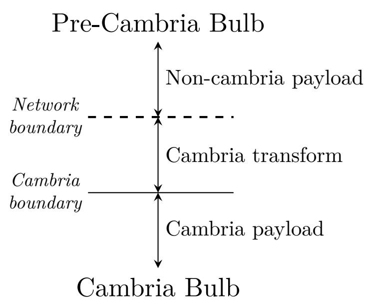

Figure 5: Asymmetric interaction between two bulbs, using Cambria to mediate data between them.

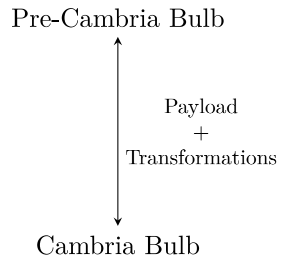

Figure 6: Asymmetric interaction between two bulbs, using Cambria to mediate data between them.

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 fig. 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 fig. 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 eq. (4)).

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

Usually, we would first design the new state machine (see fig. 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.

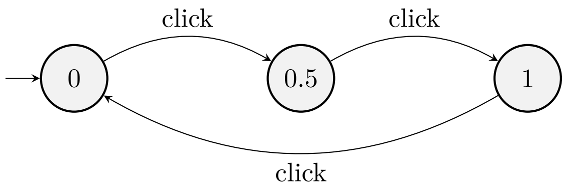

Figure 7: Light bulb with support for multiple intensities (as defined in eq. (4)).

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 eq. (2)) as:

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

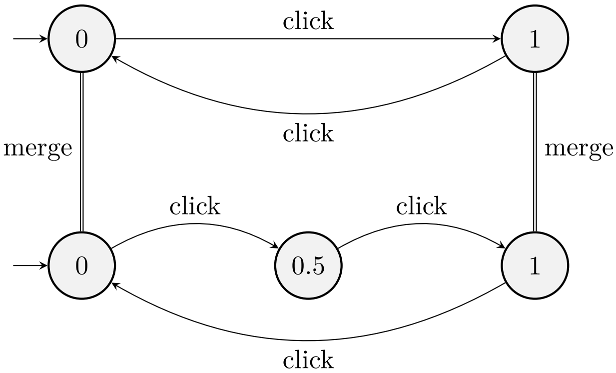

Figure 8: Visual approximation of $merge(M_1, M_2)$, where M_1 is the state machine from fig. 1 and M_2 is the state machine from fig. 2.

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

Equation (6) shows the result of merging the state machines from eqs. (3) and (4), however, while the result is accurate according to the rules we established so far, it creates a problem that is not obvious when the resulting state machine is represented in text. If we display the result in a diagram (see fig. 9) it becomes apparent — the state machine is now non-deterministic (notice the two outgoing edges from state *0* with the same *click* label).

We can to make it deterministic, which will result in a single state with a self loop. This happens because our state machine's semantics are different from traditional state machines (or acceptors), our state machine expresses behavior and acceptors express a string matching mechanism. Hence, we cannot use a traditional determinization algorithm and must rather, define our own approach.

The proposed solution is to label edges based on their version, thus removing collisions when merging. By attributing versions to labels we are able to preserve the original states and keep the state machine deterministic as shown in fig. 10. The versioning mechanism is left up to the state machine designer, for the current work, we have opted to prefix a simple version (v1, v2, etc) to the label.

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

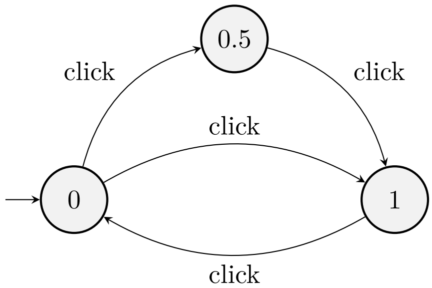

Figure 9: $merge(M_1, M_2)$ results in an NFA (eq. (5)).

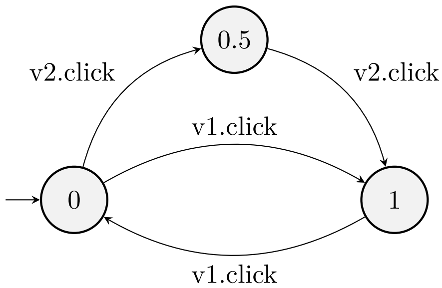

Figure 10: $merge(M_1, M_2)$ with the new labels preserves the DFA (eq. (7)).

Having solved the determinism issue, consider now a living room where the lights are controlled by two distinct buttons, the first one is only able to send *v1.click*s and the second one is only able to send *v2.click*s. Consider the trace from eqs. (8a) and (8b); in eq. (8a) someone toggled the lights to half intensity by using the second button, followed by eq. (8b) where someone clicked the button, but no action is taken. Looking back to fig. 10, we can see that state *0.5* does not have any way of handling *v1.click*. This is problematic because it means that one of the buttons is useless until the state machine goes back to a state that supports it, breaking backwards compatibility.

$$\delta(0, v2.click) \rightarrow 0.5 \tag{8a}$$

$$\delta(0.5, v1.click) \rightarrow \ ? \tag{8b}$$

To fix this, we require that each new version be complete regarding previous versions' symbols, that is, each new version state must handle *all* state labels from previous versions.

More formally, consider two state machines M_n and M_{n+1} (where M_n represents all previous versions, or $M_n = M_1 \cup M_2 \cup \cdots \cup M_n$), we require that all new states have transitions accounting for all states of previous versions:

$$\forall l \in M_n.\Sigma, s \in M_{n+1}.S, \exists s' \in (M_n.S \wedge M_{n+1}.S) : s \times l \to s' \tag{9}$$

With that in mind, M_2 defined in eq. (4) needs to be redefined (see eq. (10)); resulting in the state machine displayed in fig. 11.

$$(\Sigma, S) = (\{v1.click, v2.click\}, \{0, 0.5, 1\}) \tag{10}$$

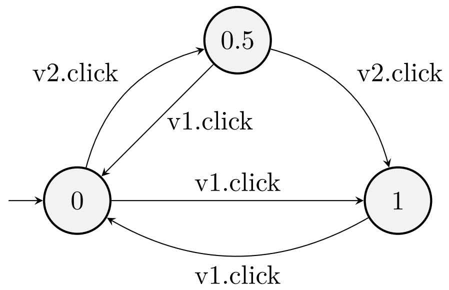

Figure 11: The result of $merge(M_1, M_2')$, where M_2' is defined in eq. (10).

Forward compatibility

So far, we have addressed backwards compatibility, that is, a state machine that is further ahead is able to support messages from previous versions without conflicts. However, for the whole system to work, old machines need to support messages from new versions without breaking. The main challenge now lies in ensuring that while allowing state machines with different version to diverge they still converge to the same state.

Consider now the state machines from figs. 2 and 10, if we run a trace composed of $[v2.click, v2.click]$, the state machines diverge, not being able to converge on the previously existing state (as shown in table 1). Furthermore, if we consider the trace $[v2.click, v1.click]$ we can observe a similar issue (see table 2), the state machines are not able to converge due to a clear incompatibility in their transitions.

	V_1	V_2
$\emptyset$	0	0
v2.click	0	0.5
v2.click	0	1

Table 1: We can see both state machines do not converge and thus generate a protocol breakage.

Fixing this requires replacing the 0.5 state's outgoing edges with a single v1.click edge, as shown in fig. 12. If we now revisit tables 1 and 2, the first table

	V_1	V_2
∅	0	0
v2.click	0	0.5
v1.click	1	0

Table 2: We can see both state machines do not converge and thus generate a protocol breakage.

is now ”incomplete” since it does not end in a state common to both versions, thus, the event sequence needs to changed to $[v2.click, v1.click]$. When running the new event sequence against our revisited version of the automata both state machines converge to the same state, as shown in table 3.

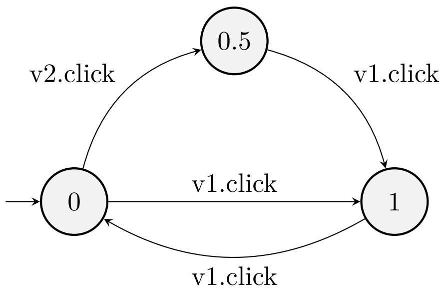

Figure 12: Reworked automata with convergence in mind.

	V_1	V_2
∅	0	0
v2.click	0	0.5
v1.click	1	1

Table 3: After changing the versioned automata, they now both converge to the same common state.

This works because both semantically and practically, the original path is kept intact, to the older version, there’s still a path to the state 1, to the new version, there are paths to states 0.5 and 1, with the former converging to 1.

Extending the state machine can be done in two ways, extending an existing path (just as we have been doing), or creating a completely new path. One might be tempted to generalize the approach we used previously, but that approach quickly fails when creating a completely new path as demonstrated in table 4 and fig. 13.

Hence, when adding completely new paths, the same rules do not apply and the proper fix here is to use $v2.event$ instead of $v1.event$.

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.

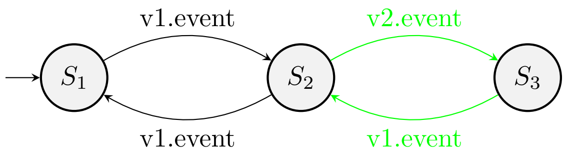

Figure 13: The green edges are the most recent additions to the state machine, extending S_2.

	V_1	V_2
$\emptyset$	S_1	S_1
v1.event	S_2	S_2
v2.event	S_2	S_3
v1.event	S_1	S_2

Table 4: Once more, the state machines do not converge.

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.

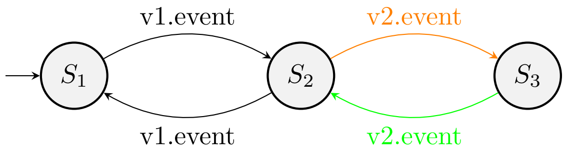

Figure 14: The colored edges are the most recent additions to the state machine, extending S_2.

	V_1	V_2
$\emptyset$	S_1	S_1
v1.event	S_2	S_2
v2.event	S_2	S_3
v2.event	S_2	S_2

Table 5: The state machines now converge.