Dreamcatcher

Robert Gersak<robi@neyho.com>
Version v.1.0, 05.02.2019

ToC

1. What is dreamcatcher?

Dreamcatcher is naive implementation of basic state machine concepts. Concept itself can be alternative for currently available programing tools/techniques like polymorphism and abstraction as well as a tool that provides high level overview of program structure or application architecture.

2. Elements of dreamcatcher

There are two entities that are explored in this naive implementation. Model and model instance. Model that is collection consisted of state¬state transition/validator mapping. Model itself should be stateless as possible.

Next to state machine model there should be some instance of that model. Something we can run, test, play around with.

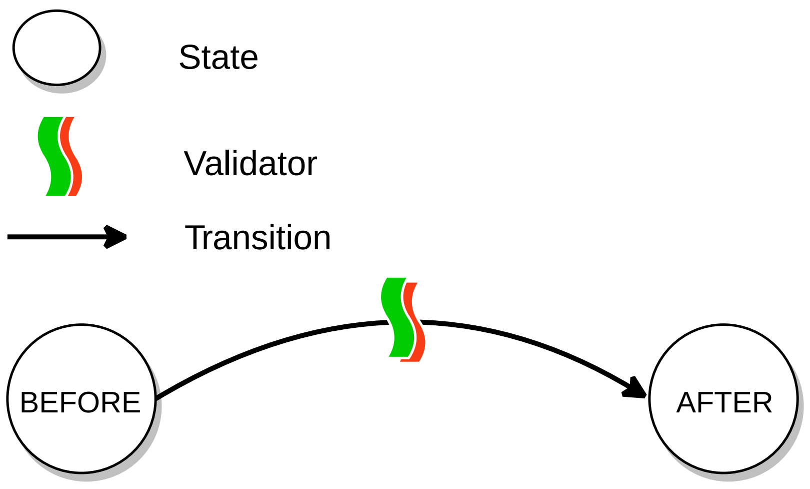
2.1. Model instance

Machine instance is a collection of *(model, data, context, state)* where:

- *model* is set of logic rules and transformations.
- *data* is some kind of data provided by runtime environment.
- *context* is data as well except this data is not supposed to be in relation with state/data and should be in relation with model.
- Current *state* positions instance data in model so that instance can move through model based on logic rules and transformations.

2.2. Model

Doesn’t do anything by itself. It is a clean representation of transformations that should happen if some conditions are met based on current position(state) in a model. Model has nothing to do with model instance per se.



2.2.1. State

State is **not** complected with instance data and certainly not mutable. State should be something that is unique, something that can uniquely position machine instance inside a model. This can be number, string, keyword as well as Object, map, record, or any other type of supported runtime data.

2.2.2. Transition

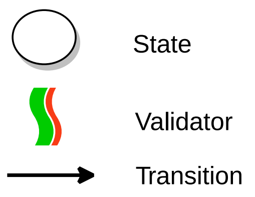
Transitions are functions of one argument. This functions or methods take machine instance as input value and transform it to new machine instance.

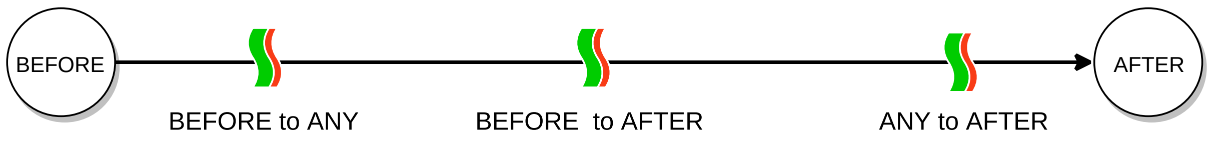
2.2.3. Validator

Similar to transitions, validators are functions that take machine instance as input and produce boolean value stating if transition can happen or if transition can’t happen.

3. State transition mechanics

Dreamcatcher considers one **special** reference to state that can be used to set model transformations and logic. **ANY** state is special in terms that machine instance can't ever be in ANY state. ANY state is considered only when applying transitions or processing validators. It is just a tool to simplify modeling since complexity groves with each new state.





Therefore multiple input transitions to state can share common ANY to state transition or validator. Vice versa multiple output transitions can share common state to ANY transition and validator.

3.1. Order of execution:

1. Validates instance and if instance is valid for transition BEFORE to ANY then apply **outgoing** state transition to instance otherwise return BEFORE instance
2. Validates instance from previous step (1) with direct validator BEFORE to AFTER and then apply **direct** state transition BEFORE to AFTER. Otherwise return BEFORE instance
3. Validates instance from previous step (2) with incoming validator ANY to AFTER and then apply **incoming** state transition ANY to AFTER and return final result. Otherwise return BEFORE instance

3.2. Instruments

To get more information about what is going on inside state machine, beside possibility to implement logging or some other tool into transitions, dreamcatcher offers optional **ANY** to **ANY** hook that will accept function or method of two arguments. First argument is *instance_before_transition* and second argument is *instance_after_transition*. This hook will activate only upon successful transition, so if one of phases of transition didn't happen and result is input instance, then ANY to ANY won't be called.

NOTE Result of ANY to ANY function is not affecting transition. Transition already happened, so this is place only for informing outside services about what happened.

4. Example STM

What can we do with elements described so far. Let us try and create not that complex state machine that will make beverages. This state machine should be capable to make following beverages:

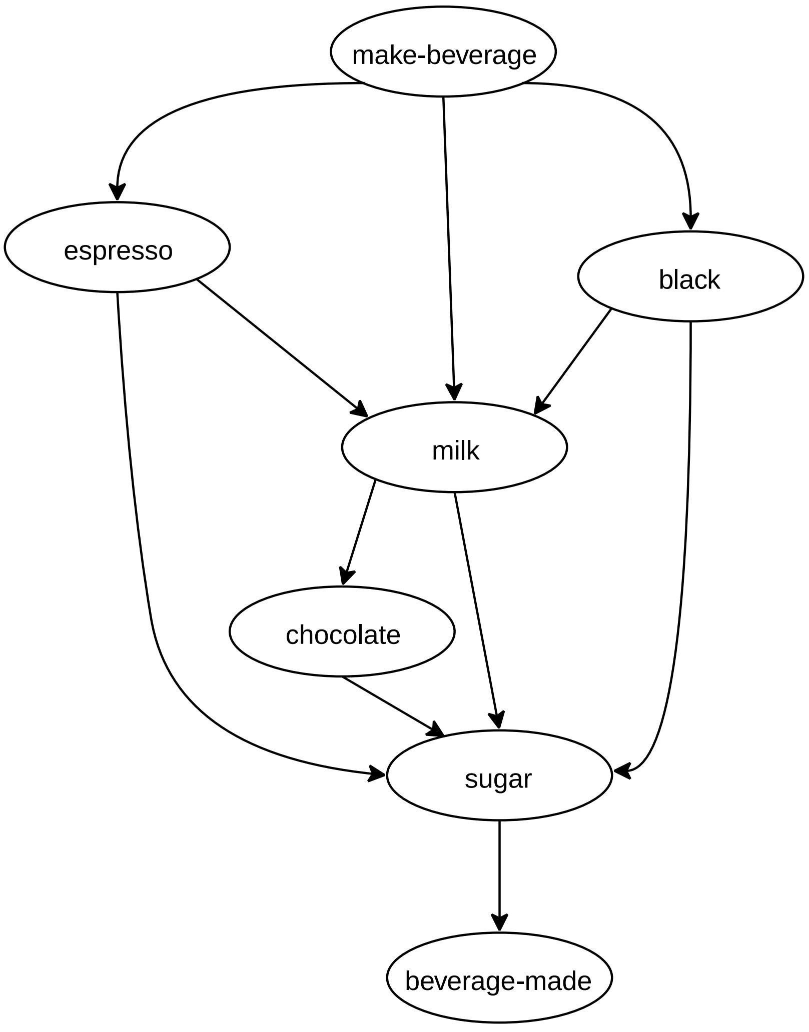
- Black Coffee
- Latte
- Espresso
- Espressino
- Cacao
- Milk

Machine should have milk, black coffee, coffee for espresso, cacao, water and sugar available for making beverages. We won't go into detail about how much of each does machine currently have. These resources are simply always available.

Recipes:

1. Black Coffee = Water + Coffee
2. Latte = Water + Coffee + Milk
3. Espresso = Water + Espresso Coffee
4. Espressino = Water + Espresso Coffee + Milk
5. Cacao = Milk + Chocolate
6. Milk = Milk

We'll call this machine *Beverage Maker* and it would look like something like this.



Picture above doesn't provide any implementation details of what each transition should do. How will water be heated, where can we find black/espresso coffee? What dosage is supposed to be used etc. Only abstract model is provided and in that abstract model one information is important for the task at hand. This information is what beverage do we want to brew/make. *make-beverage* is going to be start state of machine instance. Machine instance will behave according to rules and transitions from model above. Data for this instance will be a *Map* data structure with key **"beverage/type"** holding a value of selected beverage.

Let’s say that we defined function:

```
(defn is-beverage
  "Function expects set of beverages as input. Return value is
  function that accepts machine instance and returns \"true\" if
  machine instance data has :beverage/type that is contained in
  input set. Otherwise false"
  [beverages]
  {:pre [(set? beverages)
         (not-empty beverages)]}
  (fn [instance]
    (let [{beverage :beverage/type} (dreamcatcher/data instance)]
      (contains? beverages beverage))))
```

Validators:

```
[:make-beverage :black         (is-beverage #{"Black Coffee"})
 :make-beverage :espresso      (is-beverage #{"Espresso" "Espressino"})
 :make-beverage :milk          (is-beverage #{"Cacao" "Latte" "Milk"})
 :milk          :chocolate     (is-beverage #{"Cacao"})
 :milk          :sugar         (complement (is-beverage #{"Cacao"}))
 :black         :milk          (is-beverage #{"Latte"})
 :black         :sugar         (is-beverage #{"Black Coffee"})
 :espresso      :milk          (is-beverage #{"Espressino"})
 :espresso      :sugar         (is-beverage #{"Espresso"})]
```

Code above is proposed way of structuring validators. First "column" is source state, second one is destination state and third column represents validator function that returns true if "beverage/type" of instance is contained in second argument. Actually, function *is-beverage* returns function that does that.

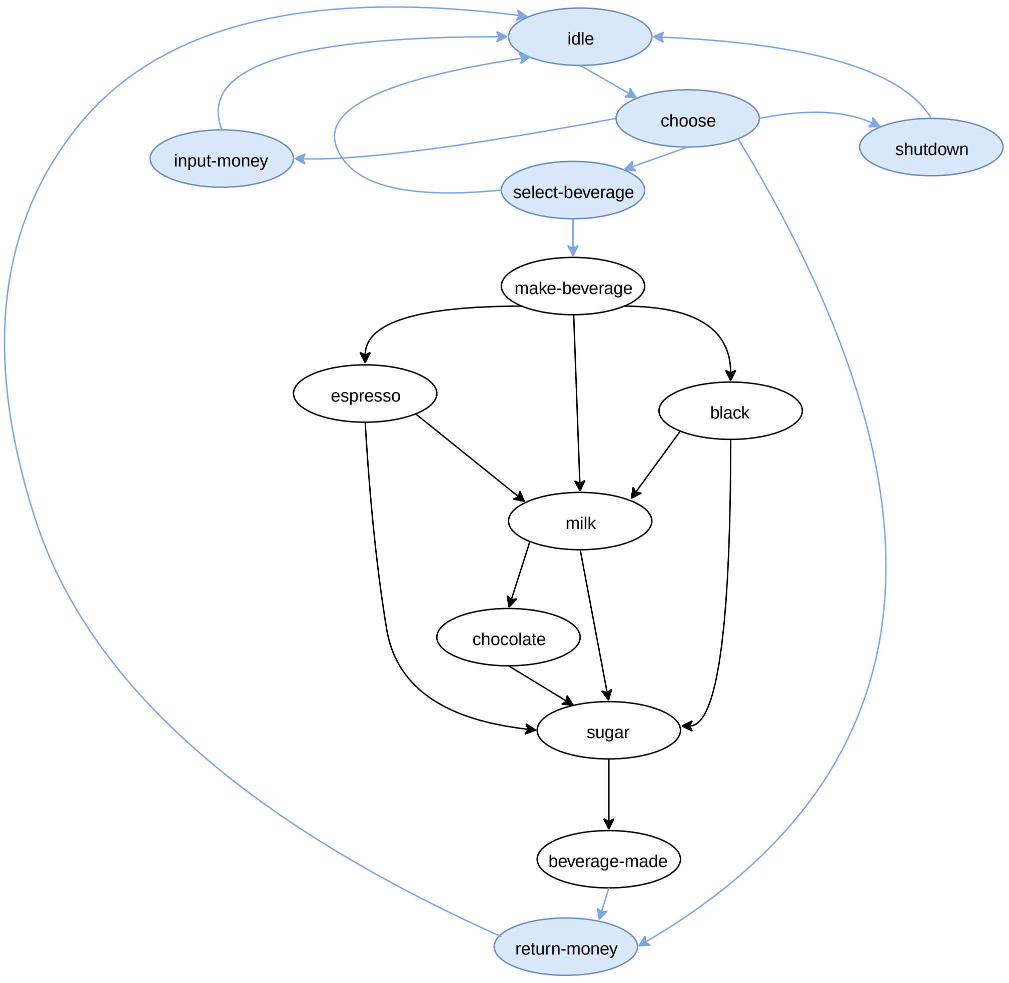
Basically, this is all the logic we need to move from state *make-beverage* to state *beverage-made*. Implementation of transitions doesn’t influence traversing this graph except if transition is tempering with the value of "beverage/type" key.

5. STM Composition

There are no obstacles to extend *Beverage Maker* and create for example *Vending Machine*. To extend *Beverage Maker* to *Vending Machine* new states, transitions and validators are required. *Beverage Maker* model will remain as is. It doesn't require change, since its function is to make beverage based only on "beverage/type".

What does *Vending machine* require?

- Money input - for end users to insert money and change "money/balance"
- Money return - either on user beverage selection or on explicit return money action machine should return "money/balance" to end user
- Beverage selection - End user should somehow select beverage, and if enough money was inserted vending machine should provide user with beverage, as well as change
- Shutdown - state that marks end of model traversing



From picture above, we can see that when user makes choice it will move machine instance to different state. We can define which state by adding value to key "vending-machine/selected" in machine instance data. After storing information in machine instance data, machine instance can resolve next step based on available transitions that are valid to complete.

For that purpose, let’s define function *is-selected* that will be used to check if user has selected specific choice.

In addition, another function is created that will check if there is enough money inserted into vending machine to allow transition to *make-beverage* state of *Beverage Maker* model.

```
(defn is-selected
  "Function returns function that accepts machine instance, extracts machine
instance
  data and compares value of "vending-machine/selected" key to input choice"
  [choice]
  (fn [instance]
    (let [{choice' :vending-machine/selected} (dreamcatcher/data instance)]
      (= choice choice'))))

(defn enough-money?
  "Function compares current money balance with beverage price. Returns true
  if money balance is greater or equal to beverage price"
  [instance]
  (let [{price :beverage/price
         balance :money/balance
         :or {balance 0}} (dreamcatcher/data instance)]
    (>= balance price)))

(def not-enough-money? (complement enough-money?))
```

In the end traversal of model transitions is limited by model validators for given machine instance.

Validators:

```
[d/any-state      :insert-money    (is-selected "Insert Money")
 d/any-state      :select-beverage (is-selected "Choose Beverage")
 :choose          :return-money    (is-selected "Return Money")
 d/any-state      :shutdown        (is-selected "Shutdown")
 :select-beverage :make-beverage   enough-money?
 :select-beverage :end             not-enough-money?]
```