Erlang/OTP

VOLUME II

The OTP Basics

MANUEL ÁNGEL RUBIO JIMÉNEZ

Edited by **Ayanda Dube**
Translated by **Ana M. Rubio**

Erlang/OTP

Volume II: The OTP Basics

Manuel Angel Rubio Jimenez

Translated by

Ana Maria Rubio Jimenez

Edited by

Ayanda Dube

Erlang/OTP
Volume II: The OTP Basics

Manuel Angel Rubio Jimenez

Translated by
Ana Maria Rubio Jimenez

Edited by
Ayanda Dube

Resumen

The development in Erlang is based on two bases well defined by its creators. The first is the power of the processes that the Erlang virtual machine implements. The second is the Concurrency Oriented Programming methodology that is facilitated with the OTP framework.

The use of Erlang is incomplete if both are not used. Its power is not discovered until processes have been used as the main source of programming and of the ***behaviors*** that the OTP framework integrates. To carry out a professional project in Erlang, knowledge and mastery of this technology is essential.

This second volume of Erlang/OTP covers the knowledge of this framework, the creation of professional projects with it and the Concurrency Oriented Programming or Actor Model as it is more widely known. Finish the necessary tour to know the powers of language and its platform. It gives the reader a tour of theory and practice, giving you even more tools and more useful code to launch into development.

In addition we cover Erlang/OTP 25, the logger system and a chapter dedicated to distributed programming.

9 788494 552380

[1] http://altenwald.org/curriculum-vitae/manuel
[2] http://creativecommons.org/licenses/by-nc-sa/3.0/

Chapter 5. State Machines

Controlling complexity is the essence of computer programming.
— Brian Kernigan

The ***gen_statem*** behaviour is based on the Mealy state machine. In versions before OTP 20, another behaviour called ***gen_fsm*** implemented a Moore state machine. The Mealy machine implementation improved performance for all OTP libraries.

This behavior is built on the server behavior (***gen_server***) and has the same dynamics seen in the previous chapter, also adding state management at the behavior level. The difference between the server and the state machine is that an event received by a server is handled in only one dimension, the implemented code and the only existing state data. In the state machine, the code that handles each request, and each event, depends on the internal state of the state machine.

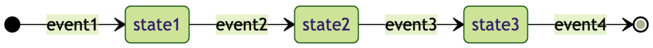
That is, if a machine has two states (for example, opened and closed), we can develop two versions of the server in our system, one identifying what we would do in the case of being in the first state (opened) and not taking into account the other state, and then we would go on to develop how the server should behave in the second state (closed) and equally regardless of the previous state. Obviously and thanks to the pattern matching we can take advantage of common events or requests that are independent of the state.

We will try to fully exploit the features available in this behavior through the examples of the traffic light, the elevator and the payment system. We will see these examples throughout the chapter.

1. Events, States, and State Diagrams

One of the most important tools in generating a finite state machine is state diagrams. These diagrams provide us at a glance with all the events that we can receive, all the states and all the possible transitions between states.

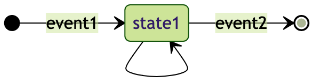
The **states** in these diagrams are represented by a circumference (or a circle) with the name of the state written inside. Ideally, they are all the same size and the same color. To facilitate reading in diagrams with many states we can change the color of some of them to form visual groups.



Starters and Terminators are vertical or horizontal lines of a certain thickness. They indicate the beginning and/or end of the state machine.

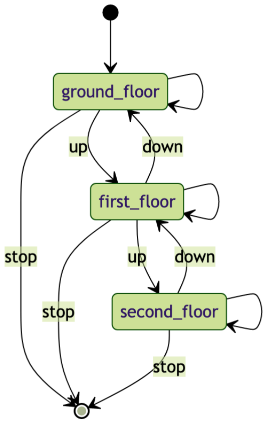
The **events** are the lines that allow transit from one state to another. These arrows start from one state or initiator and end in another state or finisher. The arrowhead marks your direction. Along its trajectory, the name of the event is written.

Events can also loop, that is, start and end in the same state, indicating that even though the event has been received and an action is performed, the state is not modified.

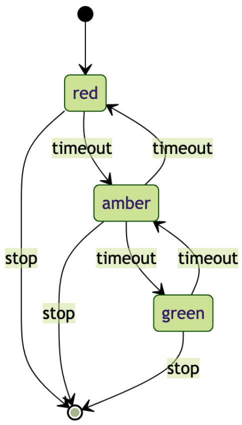


Events are generally actions that are performed by a data input on the state machine. This can mean receiving a message or even triggering a timing event.

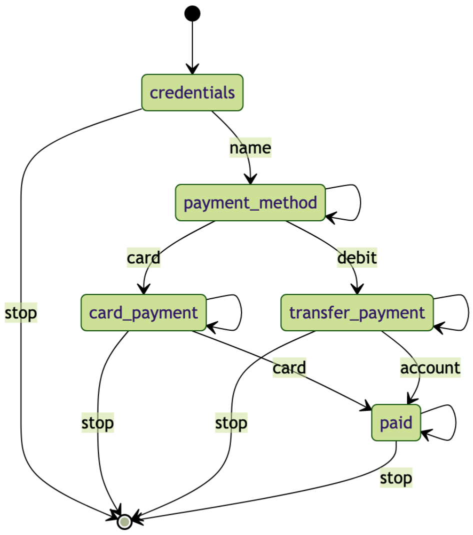
For our examples, we are going to draw their state diagrams. The elevator example will have the following diagram:



The traffic light example will show the following diagram:



Finally, the example of our payment process will have the following diagram:



Now we will see how to implement them.

2. Mealy State Machine

Theory tells us that any Moore state machine can be represented by a Mealy state machine, but not all Mealy state machines can be represented by a Moore state machine.

This is due to the special feature of Mealy machines of combining actions with state to obtain the result of the change of state for an incoming event. While the Moore state machine can be represented as:

```
State(S) x Event(E) -> State(S')
```

For each event processed in a given state, we obtain the same or another different state of departure.

The Mealy machine adds actions to be performed in the result:

```
State(S) x Event(E) -> Action(A), State(S')
```

In this way, we return the change of state and also one or several actions to be carried out.

Through the callbacks, the server can return not only a state change for the process but also the return to perform to be sent to the client. In the case of the state machine, the actions include not only the response to the user but also timers, or next events to be fired immediately similar to the effect of ***continue***, or postponing events, or changing the module in use.

In our traffic light example, we set timers. In the Mealy machine setting the timer only based on the current state and without affecting the newly received events is possible. We will see it later.

Our elevator example might have number buttons with the floor number. The actions would give us support to move to another floor and execute a check action to know if the elevator should stop or continue to the next stop. It can be an interesting exercise.

2.1. Template of a state machine

The way to use ***gen_statem*** differs depending on the way to handle the events that we decide to use. We can choose between state functions or the event handler function.

The choice is made through the definition of the callback `callback_mode/0`. This callback must return one of the following values:

`handle_event_function`

> Defines the use of a single `handle_event/4` function as the entry point for all events received in any state.

`state_functions`

> It allows to use of as many different functions as states have been defined. In addition to having the name of the state, these functions have three other parameters.

The callback return can be any one of these two atoms or a list containing one of the two atoms and more configuration options. At the moment there is only one option called ***state_enter***. We will see later what ***state_enter*** does.

3. Template for state functions

This template uses functions named after the states. They need to be added depending on the names of our states. As an example, the template has defined two states ***turned_on*** and ***turned_off***.

```
-module(simplestate_state_functions).
-behaviour(gen_statem).

-export([callback_mode/0,
         init/1,
         code_change/4,
         terminate/3]).

%% definir funciones de estado. Ejemplo:
-export([turned_on/3,
         turned_off/3]).

callback_mode() -> state_functions.

-record(state_data, {}).

init([]) ->
    {ok, turned_off, #state_data{}}.

code_change(_Vsn, State, Data, _Extra) ->
    {ok, State, Data}.

terminate(_Reason, _State, _Data) ->
    ok.

turned_on(_EventType, _EventContent, Data) ->
    {next_state, turned_off, Data}.

turned_off(_EventType, _EventContent, Data) ->
    {next_state, turned_on, Data}.
```

3.1. Template for event management function

This template does not require defining state functions, all requests will go into `handle_event/4`:

```
-module(simplestate_handle_event).
-behaviour(gen_statem).

-export([callback_mode/0,
         init/1,
         code_change/4,
         terminate/3,
         handle_event/4]).

callback_mode() -> handle_event_function.

-record(state_data, {}).

init([]) ->
    {ok, turned_off, #state_data{}}.

code_change(_Vsn, State, Data, _Extra) ->
    {ok, State, Data}.

terminate(_Reason, _State, _Data) ->
    ok.

handle_event(_EventType, _EventContent, turned_on, Data) ->
    {next_state, turned_off, Data};

handle_event(_EventType, _EventContent, turned_off, Data) ->
    {next_state, turned_on, Data}.
```

4. An event life cycle

The life cycle of a state machine is defined by its transitions, and each transition happens when an event is received. Let's see in detail what happens in the state machine when receiving a step event. Regardless of the event, the steps are as follows:

1. We receive an event. The event is decomposed to indicate in the callback the type of request and the data. We add the state data and state name and execute the function.

2. We receive the return from the function. The callback executed and gave us what to do next:

   a. We keep (`keep_state`). There is no change of state.

   b. We change or repeat state (`next_state` or `repeat_state`). We change the state, it implies the following tasks:

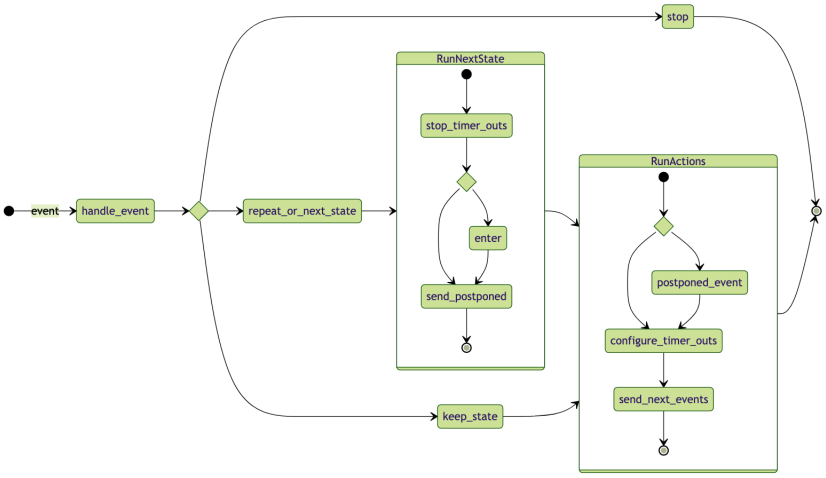
      - All state timers are stopped.

  - If ***state_enter*** was configured, the execution of an `enter` event with the event data of the previous state is added as an action.
  - Check if there are postponed events to repeat them.

 c. We stop the state machine (`stop`).

3. We execute the actions defined in the return of the callback:
   - Postpone. The current event enters the queue of postponed events to be relaunched when we change the state.
   - Timers. Configure the timers indicated in the actions.
   - Add the events (`next_event`) to be fired immediately.

Throughout the flow, we see how new events can occur from the processing of the state change or actions of the `next_event` type. These new events will be executed before going back to the cycle of listening for new incoming events.



5. Initiating the state machine

To start the state machine we only need to execute the `gen_statem:start_link/3-4`, `gen_statem:start/3-4`, or [gen_statem:start_monitor/3-4] function depending on the type of link to the process that will start the process. We can see the definition of the parameters for this function below:

```
-spec start_link(module(), args(), options()) -> result().
-spec start_link(servername(), module(), args(),
                 options()) -> result().

-type servername() :: {local,name()} |
                      {global,globalname()} |
                      {via,module(),vianame()}.
-type name() :: atom().
-type globalname() :: term().
-type vianame() :: term().
-type args() :: term().
```

```
-type options() :: [option()].
-type option() :: {debug,dbgs()} |
                  {timeout,time()} |
                  {spawn_opt,sopts()}.
-type dbgs() :: [dbg()].
-type dbg() :: trace | log | statistics |
               {log_to_file,filename()} |
               {install, {function(), functionstate()}}.
-type sopts() :: [term()].

-type result() :: {ok,pid()} | ignore | {error,error()}.
-type error() :: {already_started,pid()} | term().
```

Nothing new. The parameters to be used will change according to the example to be implemented. In our elevator example, we can start a state machine process as follows:

```
start_link() ->
    gen_statem:start_link({local, ?MODULE}, ?MODULE, [], []).
```

This execution allows us to spawn a process with the name of the module and access it through that name.

We can initiate the payment process anonymously:

```
start_link() ->
    gen_statem:start_link(?MODULE, [], []).
```

In this way, we can generate as many as we need. Finally, we will start the traffic light process by passing it parameters:

```
start_link(GreenTime, RedTime) ->
    gen_statem:start_link({local, ?MODULE}, ?MODULE,
                          [GreenTime, RedTime], []).
```

The state machine `init/1` function has one parameter. The parameters are filled with the information passed in the function `gen_statem:start_link/3-4`. The return will include the state data and also the name of the state:

```
-spec init([term()]) -> result().

-type result() :: {ok, statename(), statedata()} |
                  {ok, statename(), statedata(), actions()} |
                  {stop, reason()} |
                  ignore.
-type actions() :: [action()].
-type reason() :: term().
-type statename() :: atom().
-type statedata() :: term().
```

The server-related options can be seen in the Section 2, “Starting the server”. The contribution of ***gen_statem*** is the addition of ***statename()***

and ***actions()***. The state name allows us to configure the initial state, the entry point for the first event in our system. Actions define a series of activities to perform before attending the next event. We will see later all the types of actions available.

The implementation for our elevator example is as follows:

```
init([]) ->
    {ok, ground_floor, #state{}}.
```

In our traffic light, the initialization will get two parameters. We will first define a register for the state and function as follows:

```
-record(state, {
    green_time :: pos_integer(),
    red_time :: pos_integer(),
    next :: green | red
}).

init([GreenTime, RedTime]) ->
    {ok, red, #state{
        green_time = GreenTime,
        red_time = RedTime
    }, [{state_timeout, RedTime, timeout}]}.
```

We start the red light and configure the waiting times for each of the colors. Also, we added an action to create a state timer (***state_timeout***) to raise a timed event (***timeout***) when the timeout expires. We will see this timer action and the rest of the timer actions in the Section 8, "Timers".

Finally, the implementation of the `init/1` function for our payment system. Data collection from the payment system consists of several steps. That information should be stored in the state:

```
-record(state, {
    name :: string(),
    email :: string(),
    price :: float(),
    card :: string(),
    address :: string()
}).

init([]) ->
    {ok, identify, #state{}}.
```

We start at ***identify*** and wait for the arrival of the events.

6. Events and State change

Once we have our state machine started and in the first state we are prepared to attend events. The events will be received in the `handle_event/4` callback or the state-named functions for event handling. It depends on the type of callback chosen. For the examples, I am going to use only the type of ***state_functions***.

Note

In my experience, the ***state_functions*** mode is simpler to understand and works well as a way of learning how the state machine works. However, when implementing state machines, the ***handle_event_function*** mode tends to be more powerful because it is similar to the ***gen_server*** callbacks (`handle_call/3` and `handle_cast/2`) and allows for matching of events and states in combination.

Remember that functions will have three parameters. The first parameter is in charge of the type of event, the second will be the content of the event and the third parameter is the data of the state of the process. We are going to list the types of events that we can receive in these functions, that is, the content that can be received in the first parameter:

`{call, from()}`

Synchronous call is sent to the state machine from the aggregate process as the second parameter of the tuple. It is sent using the function `gen_statem:call/2` or `gen_statem:send_request/2`.

`cast`

Asynchronous call sent to the state machine. It is sent using the `gen_statem:cast/2` function.

`info`

Information received in the process without a specific format. It is sent using the `erlang:send/2` function or the specific syntax for sending messages using the exclamation mark (!).

`timeout | {timeout, term()} | state_timeout`

This event is received by a timer generated through one of the state machine's timeout actions.

`internal`

This event can only be generated from the actions section in the return tuple of the status functions or the event handler function. We will see later how to generate this event.

As events we receive not only information from timers or messages but also structured information such as ***call*** (synchronous calls) and ***cast*** (asynchronous calls) as we saw in ***gen_server***.

In our elevator example, we can see the implementation of the status functions using ***cast*** to receive events from the functions `button_up/0` and `button_down/0` as follows:

```
ground_floor(cast, up, StateData) ->
    io:format("going up to the first floor~n", []),
    {next_state, first_floor, StateData};

ground_floor(cast, down, StateData) ->
    io:format("beeeep! cannot go down~n", []),
    {next_state, ground_floor, StateData}.

first_floor(cast, up, StateData) ->
    io:format("going up to the second floor~n", []),
    {next_state, second_floor, StateData};

first_floor(cast, down, StateData) ->
    io:format("going down to the ground floor~n", []),
    {next_state, ground_floor, StateData}.

second_floor(cast, up, StateData) ->
    io:format("beeeep! cannot do up~n", []),
    {next_state, second_floor, StateData};

second_floor(cast, down, StateData) ->
    io:format("going down to the firt floor~n", []),
    {next_state, first_floor, StateData}.
```

We can always change state through the return of a callback when it is executed. The combinations that we have are the following:

`{next_state, state_name(), state_data(), actions()}`

> It will change the state of our state machine to a new one. We must add the new state as the second element of the tuple. The status can be a new one or the same one. Continuing in the same state is not considered a change to ***next_state*** and it is preferable to use one of the following options. The actions are optional and can be omitted thus using a tuple of only three elements.

`{keep_state, state_data(), actions()}`

> We maintain the current state of the state machine, being able to change only the state data. The actions are optional and can be omitted thus using a tuple of only two elements.

`keep_state_and_data | {keep_state_and_data, actions()}`

> We maintain current status and current status information. It allows us to make returns more compact and more readable. The actions are optional and if they are not indicated we can return the atom ***keep_state_and_data*** only.

`{repeat_state, state_data(), actions()}`

> We repeat the current state. Unlike maintaining our previous state with this option, the system supposes a state exit and re-entry. If we had added the ***state_enter*** option in the `callback_mode/0` function it will generate the execution of the state again with the ***enter*** event. Actions are optional and can be omitted.

`{repeat_state_and_data, actions()}`

> We repeat the current state and the state data. As in the previous case, it allows us to exit and re-enter the state, generating an ***enter*** event if so. Actions are optional and can be omitted by using only the ***repeat_state_and_data*** atom instead.

`stop | {stop, reason(), state_data()}`

> Stops execution of the state machine and passes execution to the `terminate/3` callback. Specifying state information is optional.

`{stop_and_reply, reason(), reply(), state_data()}`

> Stops execution of the state machine and passes execution to the `terminate/3` callback. Before it sends the response to the calling process. Specifying state information is optional.

Although in the definition of the Mealy machine, the return of each event is a new state and actions we can maintain the state and we can ignore the actions if we do not have actions to execute, as we saw in the elevator example above.

In our traffic light implementation, we have the state timers and synchronous call (***call***) events along with returns where we can see the state change (***next_state***) and how to keep the state and state data (***keep_state_and_data***) in addition to the response actions (***reply***) and state timer (***state_timeout***):

```
handle_event(state_timeout, {change, green}, red, State) ->
    {next_state, amber, State,
     [{state_timeout, ?AMBER_TIME, {change, red}}]};

handle_event(state_timeout, {change, green}, amber, State) ->
    #state{green_time = GreenTime} = State,
    {next_state, green, State,
     [{state_timeout, GreenTime, {change, red}}]};

handle_event(state_timeout, {change, red}, amber, State) ->
    #state{red_time = RedTime} = State,
    {next_state, red, State,
     [{state_timeout, RedTime, {change, green}}]};

handle_event(state_timeout, {change, red}, green, State) ->
    {next_state, amber, State,
     [{state_timeout, ?AMBER_TIME, {change, green}}]};

handle_event({call, From}, watch_light, StateName, _StateData)
 ->
    {keep_state_and_data,
     [{reply, From, StateName}]}.
```

In the payment example we only use synchronous calls (***call***) like the ***watch_light*** call seen above. You can see its implementation at the end of this chapter in Section 16, "Example of payment".

7. Actions

We have been able to see in the return of the callbacks the possibility of indicating some actions. In this section we will see the types of actions that we can add in this return.

We can organize the actions in groups. These groups are not exclusive. We can add each action to none or several groups. We will also see actions not belonging to any group. The groups are:

Input actions

These actions include hibernation, timeout, and response actions.

Timing actions

These actions include event, generic and state timers.

Transition actions

These actions include postponing an event, hibernation, event timer, generic timer and state timer.

Note

These groups have been defined through types in the source code of the ***gen_statem*** module. In the Erlang/OTP documentation[1] for this module we can clearly see these types and what they include.

The actions are accumulated in a list. This list could contain duplicate or incompatible actions with each other. As a general rule, the transition actions overwrite each other and only the last one that we specify is taken into account.

Other actions like sending an event (***next_event***) can be added as many as needed.

We are going to list all the possible actions that we have:

[1] http://erlang.org/doc/man/gen_statem.html

`postpone | {postpone, boolean()}`

It allows us to postpone the treatment of an event. This action places postponed messages in a queue to be processed only when the status changes. Very useful if we find ourselves in a state where a type of event cannot yet be processed, for example in a disconnected state.

`{next_event, event_type(), event_content()}`

It allows us to add a new event in the event queue to be processed by the state machine. Of this type we can add as many actions as we need. If we need to differentiate between events sent by external elements via ***call***, ***cast***, ***info*** or ***timeout*** we can use ***internal***. In this way we know the unequivocal origin of the event.

`hibernate | {hibernate, boolean()}`

Sends the process to hibernation. This state lasts until receiving a new event. If we have many state machine processes and event handling takes a long time for a certain state, we can add this action, thus compressing memory usage and reducing processor consumption[2].

`timeout() | {timeout, timeout(), event_content(), options()}`

Create a nameless or event timer. We will look at the timers later along with their options. The ***timeout()*** data type can indicate a positive integer or the atom ***infinity***.

`{{timeout, name()}, timeout(), event_content(), options()}`

Create a generic or named timer. We will look at the timers later along with their options. The options in this specification are optional.

`{state_timeout, timeout(), event_content(), options()}`

Create a state timer. We will look at the timers later along with their options. The options in this specification are optional.

`{reply, from(), term()}`

Sends a response to the calling process. We must have the data ***From***. Normally this type of action is only used in functions where an event of type ***call*** has been received.

[2]However, if there are a large number of Constantly received events hibernation could be a performance penalty. Use this option carefully.

{change_callback_module, module()}

Change the module for the execution of the following callbacks. This function is useful if our state machine implementation is very large and we want to write the specific functions of each state in a different module.

{push_callback_module, module()}

It acts similarly to the previous case but maintains a stack with the modules that we have been changing. This allows the designing of multilevel state machines where the module acts as another dimension within the states and allows to define of the states by groups. We will see this concept later.

pop_callback_module

It allows us to return to the previous group of states. It is the complementary operation to `push_callback_module`.

Within our examples, as we saw in the previous section, we used the state timeout actions (***state_timeout***) as well as the response action (***reply***).

We will develop the next version of the traffic light a bit later where we will use more actions like ***hibernate***.

We can also see two versions of some actions where they appear in the form of a tuple. Remember that actions are a property list (see ***proplists*** module). The property list allows the duplication of its elements and depending on the function to obtain the data, we can obtain all the values under the same key, or the first value of the list. So, if we need to avoid postponing an event or going into hibernation, we can add to the action list:

```
NewActions =
    [{hibernate, false}, {postpone, false} | Actions],
```

8. Timers

Unlike the other behaviors in the state machine, we can find different types of timers. Each type acts differently and is implemented differently.

The types that we can find are the following:

event timer

This timer is similar to what we can find in any other behavior. It is only triggered if, after specifying it, no other event occurs in that period of time. That is, it is an ideal waiting time to measure inactivity in the process.

state timer

> Act on the state. As long as the status does not change, it will continue to count the time until it expires and generates the ***status_timeout*** event. Useful to limit permanence in a state, for example, of an unauthenticated user in protocols such as IMAP or XMPP.

generic timer

> This timer requires the specification of a name. If before generating the call ***timeout, name()}*** another generic timer with the same name is generated, the counter is reset. It can be useful to react at specific times to ***ping*** systems where a user action is required or the communication channel is closed.

In the case of using ***infinity*** as timeout instead of configuring the timer, it is ignored or eliminated. If the value passed is zero (0) a timer is not generated either, instead the event is sent directly.

The timing actions will not only allow us to configure the type of timer but also the information to be launched when the timeout event occurs. Therefore, the events received as event type and event content will be:

timeout, event_content()

> For actions fired as ***{timeout, 100, tick}*** will be received as the ***timeout*** event type and as the ***tick*** event content.

{timeout, name()}, event_content()

> For actions fired as ***{{timeout, clock}, 100, tick}*** will be received as event type ***{timeout, clock}*** and as content of the ***tick*** event.

state_timeout, event_content()

> For actions fired as ***{state_timeout, 100, tick}*** will be received as ***state_timeout*** event type and as ***tick*** event content.

We can see an example case in the traffic light code:

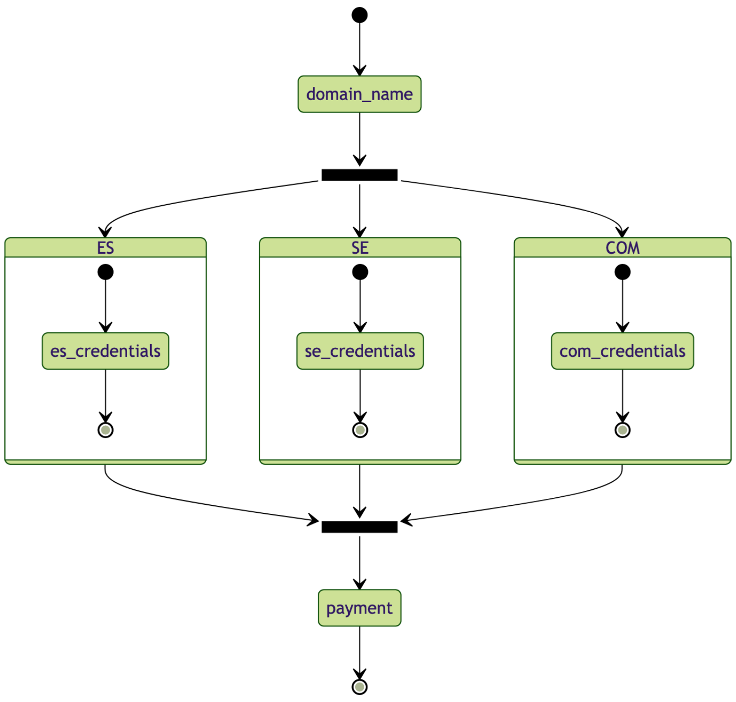
```
handle_event(state_timeout, {change, green}, red, State) ->
    {next_state, amber, State,
     [{state_timeout, ?AMBER_TIME, {change, red}}]}.
```

We receive a stateful timer event (***{change, green}***) and generate an action with another stateful timer. This time change to red. In this way, we maintain the flow of change between traffic light colors and waiting times.

9. States groups

We are in the process of purchasing a domain. Domains have specific characteristics when we try to buy domains from a specific country. For example, to buy a .es domain, the system requires us to add our Spanish identification number (DNI or CIF), for a .se domain we need to have a Swedish ID and the same for other domains. Each domain has its specific data with its validations and peculiarities and yet there is a common data collection and only a different part of the process.

Although we could solve this problem in many different ways, let's create a state machine for shopping like this:



We see the groups formed by the specific domains in this example. However, the groups can indicate a higher partitioning in each of the groups, making a process with dozens of states can be partitioned and better-managed thanks to its division and encapsulation.

At the same time, groups of states can be reused at different locations throughout the workflow allowing the use of ***push*** and ***pop***. An example could be the lexical analysis of a text because the rules are associated with syntactic trees and can jump from one state to another at each level, going very deep. A stack in this case would allow the states to be stacked and then returned in the reverse way.

10. Terminating the state machine

Termination of the state machine is accomplished by returning ***stop*** on a callback or by calling the `gen_statem:stop/1` or `gen_statem:stop/3` function.

In the implementation of our examples, we have used only the first option considering the completion as ***normal***. The implementation in the example of the elevator and the traffic light is:

```
stop() ->
    gen_statem:stop(?MODULE).
```

The completion for the payment example differs because it is started anonymously. The implementation is:

```
stop(Name) ->
    gen_statem:stop(Name).
```

Executing these functions to stop the execution of the state machines executes the `terminate/3` function. The parameters of this function are the termination reason. In our three examples, it will always be ***normal***. The only way to receive a different reason would be to end the process through the `exit/2` function to specify a different reason.

The specification of the `terminate/3` callback is as follows:

```
-spec terminate(reason(), state_name(), state_data()) ->
 no_return().

-type reason() :: atom().
-type state_name() :: atom().
-type state_data() :: term().
```

Note

This callback is optional. By default ***gen_statem*** does not require this function, if it is not defined it proceeds directly to terminate the process.

For our payment example, we have the implementation of this function as follows:

```
terminate(normal, paid, #state{payment_method = card} = State)
 ->
    io:format("~p Name: ~s~nCard: ~s~nPaid.~n",
              [self(), State#state.name, State#state.card]),
    ok;

terminate(normal, paid, #state{payment_method = debit} =
 State) ->
    io:format("~p Name: ~s~nCard: ~s~nPaid.~n",
              [self(), State#state.name,
 State#state.account]),
    ok;

terminate(normal, _StateName, _StateData) ->
    io:format("~p No paid.~n", [self()]),
    ok.
```

We offer information about the completion of the payment process depending on the information stored in the process.

11. Hot swapping code

For hot swapping, we have developed two versions of the lift. In principle, the first elevator code is an elevator with two buttons, one to go up to a higher floor and another to go down to a lower floor, validating whether it is possible to do so or not.

We have developed a second version that actually acts as an elevator. We keep the `button_up/0` and `button_down/0` function for backward compatibility. We've implemented the `button_num/1` function for the actual functionality of pressing a number button.

The new code keeps the elevator moving by going up and down depending on which buttons are pressed. If there are no more buttons pressed, it remains on the floor until a new order is received. Also, the number of floors is increased to 10 since the states have been modified to be: ***stopped*** (stopped), ***going_up*** (up) and ***going_down*** (down).

As we have done with other hot code changes, we have added a version to the headers of each code example to make it easier to change from one version to the next:

```
-module(elevator).
-author('manuel@altenwald.com').
-vsn(1).
```

In the second version file, we have to add the same thing but add a number to the version:

```
-module(elevator).
-author('manuel@altenwald.com').
```

```
-vsn(2).
```

The code changes are quite extensive. In principle, we change not only the states but also the information stored within the state. In the second version, our state is as follows:

```
-record(state, {
    pressed = sets:new() :: sets:sets(),
    current_floor = 0 :: pos_integer()
}).
```

Using a set in the button-pressed list ensures that duplicate items are removed and we don't have to worry about them. Also, when changing the state diagram from the floor where the elevator is located to the actual state of the motor (stopped, going up or down) we must store the current floor in the state data.

The code corresponding to the update for the second file must be implemented in the `code_change/4` function. This function has the following definition:

```
-spec code_change(old_vsn(), state_name(), state_data(),
 extra()) -> {ok, state_name(), state_data()} | {error,
 reason()}.

-type old_vsn() :: vsn() | {down, vsn()}.
-type vsn() :: term().
-type state_name() :: atom().
-type state_data() :: term().
-type reason() :: term().
-type extra() :: term().
```

In the first version of the code, the implementation is empty. The code in the first versions is not used since it will not migrate or change information from any previous version. The second version of the code is as follows:

```
code_change(1, ground_floor, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 0}};

code_change(1, first_floor, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 1}};

code_change(1, second_floor, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 2}};

code_change({down, 1}, _, StateData, _Extra) ->
    NewState = case StateData#state.current_floor of
        0 -> ground_floor;
        1 -> first_floor;
        _ -> second_floor
    end,
    {ok, NewState, {state}}.
```

We cover upgrading from version 1 based on the state it is in and also the supposed rollback if you roll back to version 1.

Let's test our code from a console. We assume that we have the code for the first version in the `elevator_v1` directory and the second version in the `elevator_v2` directory to perform our test.

We open a console and compile the first code. We launch the process:

```
> c("elevator_v1/elevator.erl").
{ok,elevator}
> elevator:start_link().
starting on the ground floor
{ok,<0.64.0>}
> elevator:button_up().
going up to the first floor
ok
> elevator:button_up().
going up to the second floor
ok
```

We have the process running and our elevator on the second floor. Now we will change the code to get the functionality of a real elevator. For this, we execute:

```
> sys:suspend(elevator).
ok
> c("elevator_v2/elevator.erl").
{ok,elevator}
> sys:change_code(elevator, elevator, 1, []).
ok
> sys:resume(elevator).
ok
> elevator:button_num(5).
* closing doors and going up
ok
< passing floor 3
< passing floor 4
> stopping on floor 5, opening doors
```

The code has been executed and has changed the internal state of the process to add the new state where the current floor is the second. The elevator goes into stopped mode (***stopped***) and when executing the new command to move it to the fifth floor, it goes up from the second to the fifth without a problem.

If we needed to go back to the previous version of the elevator code we could execute the following code:

```
> sys:suspend(elevator).
ok
> sys:change_code(elevator, elevator, {down, 1}, []).
ok
> c("elevator_v1/elevator.erl").
```

```
{ok,elevator}
> sys:resume(elevator).
ok
> sys:get_state(elevator).
{second_floor,{state}}
> elevator:button_down().
going down to the first floor
ok
```

This time we must execute the back action (`sys:code_change/4`) before compiling and loading the previous code. We see that the state of the function changes correctly according to what has been developed and once we recover the execution of the process, the operation is correct.

12. Obtaining information from the state machine

In Section 11, "Obtaining server information" we talk about how to display the information of a server. At the end of the day, a state machine is a server. However, the internal information changes subtly when integrating the name of the state.

The specification of the function `format_status/2` is the same as long as we see two parameters but the way to call this function concerning the second parameter differs. The function specification is as follows:

```
-spec format_status(status()) -> status().

-type status() :: #{
    state => term(),
    message => term(),
    reason => term(),
    log => [sys:system_event()]
}.
```

Note

This callback is optional. By default ***gen_statem*** implements a mechanism to return the state of the process if this function is not defined.

Important

This function is available as of OTP 25. In earlier versions, there is an earlier version that receives two `format_status/2` parameters. The difference lies in the organization of the information, the new function uses a map and therefore it is easier to access the data, even matching the parameters of the function.

This callback will be used only when we use the `sys:get_state/1` function. By default, we will obtain a tuple with the elements of the state name and the state data:

```
> sys:get_state(elevator).
{first_floor,{state}}
```

We can see an example of the default dump of state machine information below using the function `sys:get_status/1`:

```
> sys:get_status(elevator).
{status,<0.64.0>,
        {module,gen_statem},
        [[{'$initial_call',{elevator,init,1}},
          {'$ancestors',[<0.57.0>]}],
         running,<0.57.0>,[],
         [{header,"Status for state machine elevator"},
          {data,[{"Status",running},
                 {"Parent",<0.57.0>},
                 {"Modules",[elevator]},
                 {"Time-outs",{0,[]}},
                 {"Logged Events",[]},
                 {"Postponed",[]}]},
          {data,[{"State",{first_floor,{state}}}]}]]}
```

We can see that there is a section with the key ***data*** (first element of the tuple) at the end that tells us its ***State*** through a tuple of two elements whose first element is the name of the state and the second element is the data of the state.

In the previous ***data*** entry we can see other data such as ***Postponed*** where the events are postponed through the specific action seen in the actions section or ***Time-outs*** where we can see the active timers will be stored. If we run the traffic light example we can see it:

```
> {ok, PID} = traffic_light:start_link(30_000, 30_000).
{ok,<0.117.0>}
> sys:get_status(PID).
{status,<0.117.0>,
        {module,gen_statem},
        [[{'$initial_call',{traffic_light,init,1}},
          {'$ancestors',[<0.85.0>]}],
         running,<0.85.0>,[],
         [{header,"Status for state machine traffic_light"},
          {data,[{"Status",running},
                 {"Parent",<0.85.0>},
                 {"Modules",[traffic_light]},
                 {"Time-outs",{1,[{state_timeout,
{change,green}}]}},
                 {"Logged Events",[]},
                 {"Postponed",[]}]},
          {data,[{"State",{red,{state,30000,30000}}}]}]]}
```

You can see the information about the number of ***time-outs*** defined and the information of each one of them. In this case we only have a state timer.

13. Tracing a state machine

The traces help us understand what the state machine is doing when it receives an event. The state machine can perform actions, change state, and receive different types of timers. Traces help us see all this information. For example, in the case of the traffic light, if we activate the traces we can see:

```
> {ok, PID} = traffic_light:start_link(30_000, 30_000).
{ok,<0.117.0>}
* change to amber
* change to green
> sys:trace(PID, true).
ok
*DBG* traffic_light receive state_timeout {change,red} in
 state green
* change to amber
*DBG* traffic_light consume state_timeout {change,red} in
 state green => amber
*DBG* traffic_light start_timer {state_timeout,1000,
{change,red},[]} in state amber
*DBG* traffic_light receive state_timeout {change,red} in
 state amber
* change to red
*DBG* traffic_light consume state_timeout {change,red} in
 state amber => red
*DBG* traffic_light start_timer {state_timeout,30000,
{change,green},[]} in state red
```

We can see the information about the ***state_timeout*** event received with the `{change, red}` information while it is green and how it does the change by going through amber first.

You can see more about the traces in the Section 12, “Tracing the server”.

14. Example of the elevator

Throughout the chapter we have seen the implementation of the elevator in parts. Now we are going to see the complete code of this state machine and we will test its execution. As this code has two versions, the one that we will present here will be the second most complete version.

The complete code is the following:

```
-module(elevator).
-author('manuel@altenwald.com').
-vsn(2).

-behaviour(gen_statem).
```

```
-export([
    start_link/0,
    stop/0,
    button_num/1,
    button_up/0,
    button_down/0
]).

-export([
    callback_mode/0,
    init/1,
    handle_event/4,
    terminate/3,
    code_change/4
]).

-define(MAX_FLOORS, 10).
-define(TOP_BORDER_FLOOR, ?MAX_FLOORS + 1).
-define(BOTTOM_BORDER_FLOOR, -1).
-define(TIME_TO_FLOOR, 1000).

-record(state, {
    pressed = sets:new() :: sets:sets(),
    current_floor = 0 :: pos_integer()
}).

start_link() ->
    gen_statem:start_link({local, ?MODULE}, ?MODULE, [], []).

stop() ->
    gen_statem:stop(?MODULE).

button_num(Num) ->
    gen_statem:cast(?MODULE, {pressed, Num}).

button_up() ->
    gen_statem:cast(?MODULE, {pressed, up}).

button_down() ->
    gen_statem:cast(?MODULE, {pressed, down}).

callback_mode() ->
    handle_event_function.

init([]) ->
    io:format("* initiating elevator, floor 0~n", []),
    {ok, stopped, #state{}, [hibernate]}.

add_num(Num, #state{pressed = Pressed} = StateData) ->
    NewPressed = sets:add_element(Num, Pressed),
    StateData#state{pressed = NewPressed}.

del_num(Num, #state{pressed = Pressed} = StateData) ->
    NewPressed = sets:del_element(Num, Pressed),
    StateData#state{pressed = NewPressed}.

current(StateData, Num) ->
    StateData#state{current_floor = Num}.

handle_event(cast, {pressed, up}, _Name, #state{current_floor
 = Current}) ->
```

```
    {keep_state_and_data, [{next_event, cast, {pressed,
 Current + 1}}]};
handle_event(cast, {pressed, down}, _Name,
 #state{current_floor = Current}) ->
    {keep_state_and_data, [{next_event, cast, {pressed,
 Current - 1}}]};
handle_event(cast, {pressed, Num}, _Name, _Data)
        when Num =< ?BOTTOM_BORDER_FLOOR orelse
             Num >= ?TOP_BORDER_FLOOR ->
    io:format("x received incorrect floor: ~p~n", [Num]),
    keep_state_and_data;
handle_event(cast, {pressed, Num}, stopped, StateData) ->
    case StateData#state.current_floor of
        Current when Current < Num ->
            NewState = add_num(Num, StateData),
            Next = NewState#state.current_floor + 1,
            Actions = [{state_timeout, ?TIME_TO_FLOOR, {stop,
 Next}}],
            io:format("* closing doors and going up~n", []),
            {next_state, going_up, NewState, Actions};
        Current when Current > Num ->
            NewState = add_num(Num, StateData),
            Next = NewState#state.current_floor - 1,
            Actions = [{state_timeout, ?TIME_TO_FLOOR, {stop,
 Next}}],
            io:format("* closing doors and going down~n", []),
            {next_state, going_down, NewState, Actions};
        Current when Current =:= Num ->
            io:format("> current floor, doors opened~n", []),
            keep_state_and_data
    end;
handle_event(cast, {pressed, Num}, _StateName, StateData) ->
    NewState = add_num(Num, StateData),
    Nums = lists:sort(sets:to_list(NewState#state.pressed)),
    io:format("+ adding floor ~p to stop (~p)~n", [Num,
 Nums]),
    {keep_state, NewState};

handle_event(state_timeout, {stop, Current}, StateName,
 StateData) ->
    case sets:is_element(Current, StateData#state.pressed) of
        true ->
            io:format("> stopping on floor ~p, opening
 doors~n", [Current]),
            NewState = del_num(Current, current(StateData,
 Current)),
            {Up, Down} = lists:partition(fun(E) -> E > Current
 end,

 sets:to_list(NewState#state.pressed)),
            case {length(Up), length(Down), StateName} of
                {0, 0, _} ->
                    {next_state, stopped, NewState};
                {UpL, _, going_up} when UpL > 0 ->
                    Next = Current + 1,
                    Actions = [{state_timeout, ?TIME_TO_FLOOR,
 {stop, Next}}],
                    {keep_state, NewState, Actions};
                {0, _, going_up} ->
                    Next = Current - 1,
                    Actions = [{state_timeout, ?TIME_TO_FLOOR,
 {stop, Next}}],
                    {next_state, going_down, NewState,
 Actions};
                {_, DownL, going_down} when DownL > 0 ->
                    Next = Current - 1,
                    Actions = [{state_timeout, ?TIME_TO_FLOOR,
 {stop, Next}}],
                    {keep_state, NewState, Actions};
                {_, 0, going_down} ->
                    Next = Current + 1,
                    Actions = [{state_timeout, ?TIME_TO_FLOOR,
 {stop, Next}}],
                    {next_state, going_up, NewState, Actions}
            end;
        false ->
            io:format("< passing floor ~p~n", [Current]),
            Next = case StateName of
                going_up -> Current + 1;
                going_down -> Current - 1
            end,
            Actions = [{state_timeout, ?TIME_TO_FLOOR, {stop,
 Next}}],
            {keep_state, current(StateData, Current), Actions}
    end.

terminate(_Reason, _StateName, _StateData) ->
    ok.

code_change(1, ground_floor, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 0}};
code_change(1, first_floor, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 1}};
code_change(1, second_floor, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 2}};
code_change({down, 1}, _, StateData, _Extra) ->
    NewState = case StateData#state.current_floor of
        0 -> ground_floor;
        1 -> first_floor;
        _ -> second_floor
    end,
    {ok, NewState, {state}}.
```

We open a shell and try it as follows:

```
> c(elevator).
{ok,elevator}

> elevator:start_link().
* initiating elevator, floor 0
{ok,<0.116.0>}

> elevator:button_num(2).
* closing doors and going up
ok
< passing floor 1
> stopping on floor 2, opening doors

> elevator:button_abajo().
* closing doors and going down
ok
> stopping on floor 1, opening doors
> lists:foreach(fun elevator:button_num/1, [8,5,3,9,0]).
* closing doors and going up
ok
+ adding floor 5 to stop ([5,8])
+ adding floor 3 to stop ([3,5,8])
+ adding floor 9 to stop ([3,5,8,9])
+ adding floor 0 to stop ([0,3,5,8,9])
< passing floor 2
> stopping on floor 3, opening doors
< passing floor 4
> stopping on floor 5, opening doors
< passing floor 6
< passing floor 7
> stopping on floor 8, opening doors
> stopping on floor 9, opening doors
< passing floor 8
< passing floor 7
< passing floor 6
< passing floor 5
< passing floor 4
< passing floor 3
< passing floor 2
< passing floor 1
> stopping on floor 0, opening doors

> elevator:stop().
ok
```

The elevator is started as a process and we can make it go up and down using the corresponding functions. As the last step, we can stop it. The implementation of our elevator works correctly.

15. Example of the traffic light

The traffic light code as a state machine is as follows:

```
-module(traffic_light).
-author('manuel@altenwald.com').

-behaviour(gen_statem).

-export([
    start_link/2,
    stop/0,
    watch_light/0
]).

-export([
    callback_mode/0,
    init/1,
    handle_event/4
]).

-define(AMBER_TIME, 1000). %% 1 second

-record(state, {
    green_time :: pos_integer(),
    red_time :: pos_integer()
}).
```

```
start_link(GreenTime, RedTime) ->
    gen_statem:start_link({local, ?MODULE}, ?MODULE,
                          [GreenTime, RedTime], []).

stop() ->
    gen_statem:stop(?MODULE).

watch_light() ->
    gen_statem:call(?MODULE, watch_light).

callback_mode() ->
    handle_event_function.

init([GreenTime, RedTime]) ->
    {ok, red, #state{
        green_time = GreenTime,
        red_time = RedTime
    }, [{state_timeout, RedTime, {change, green}}]}.

handle_event(state_timeout, Event, red, State) ->
    io:format("* change to amber~n", []),
    {next_state, amber, State,
     [{state_timeout, ?AMBER_TIME, Event}]};

handle_event(state_timeout, {change, green}, amber, State) ->
    io:format("* change to green~n", []),
    {next_state, green, State,
     [{state_timeout, State#state.green_time, {change,
 red}}]};

handle_event(state_timeout, {change, red}, amber, State) ->
    io:format("* change to red~n", []),
    {next_state, red, State,
     [{state_timeout, State#state.red_time, {change,
 green}}]};

handle_event(state_timeout, Event, green, State) ->
    io:format("* change to amber~n", []),
    {next_state, amber, State,
     [{state_timeout, ?AMBER_TIME, Event}]};

handle_event({call, From}, watch_light, StateName, _StateData)
 ->
    {keep_state_and_data,
     [{reply, From, StateName}]}.
```

We open a shell and prove an execution:

```
12> c(traffic_light).
{ok,traffic_light}
13> traffic_light:start_link(2000, 2000).
{ok,<0.103.0>}
* change to amber
* change to green
* change to amber
* change to red
14> traffic_light:watch_light().
red
* change to amber
* change to green
* change to amber
* change to red
15> traffic_light:stop().
ok
```

The changes follow each other correctly and obtain information when the traffic light changes color each time.

16. Example of payment

In the previous sections, we have seen how our code to implement the payment state diagram progressed. This example is a bit longer than the previous ones because it has more states and a branch in one of the states.

However, this system is linear. It does not return to previous states and ends after performing all the programmed transitions. This makes it the easiest to follow.

Its full code can be seen below:

```
-module(payment).
-author('manuel@altenwald.com').

-behaviour(gen_statem).

-export([
    start_link/0,
    stop/1,
    give_name/2,
    give_payment_method/2,
    give_card/2,
    give_account/2,
    get_info/1
]).

-export([
    callback_mode/0,
    init/1,
    handle_event/4,
    terminate/3,
    code_change/4
]).

-type payment_method() :: card | debit.

-record(state, {
        name :: string(),
        payment_method :: payment_method(),
        card :: string(),
        account :: string()
}).

start_link() ->
    gen_statem:start_link(?MODULE, [], []).

stop(Name) ->
```

```
    gen_statem:stop(Name).

give_name(PID, Name) ->
    gen_statem:call(PID, {name, Name}).

give_payment_method(PID, PaymentMethod) ->
    gen_statem:call(PID, {payment_method, PaymentMethod}).

give_card(PID, Card) ->
    gen_statem:call(PID, {card, Card}).

give_account(PID, Account) ->
    gen_statem:call(PID, {account, Account}).

get_info(PID) ->
    gen_statem:call(PID, info).

callback_mode() ->
    handle_event_function.

init([]) ->
    {ok, credentials, #state{}}.

handle_event({call, From}, info, StateName, StateData) ->
    {keep_state_and_data, [{reply, From, {StateName,
 StateData}}]};

handle_event({call, From}, {name, Name}, credentials, State) -
>
    {next_state, payment_method, State#state{name = Name},
 [{reply, From, ok}]};
handle_event({call, From}, _Event, credentials, _State) ->
    {keep_state_and_data, [{reply, From, {error, "name
 required!"}}]};

handle_event({call, From}, {payment_method, card},
 payment_method, State) ->
    {next_state, card_payment, State#state{payment_method =
 card},
     [{reply, From, ok}]};
handle_event({call, From}, {payment_method, debit},
 payment_method, State) ->
    {next_state, debit_payment, State#state{payment_method =
 debit},
     [{reply, From, ok}]};
handle_event({call, From}, _Event, payment_method, _State) ->
    {keep_state_and_data,
     [{reply, From, {error, "payment method: card or
 debit"}}]};

handle_event({call, From}, {card, Card}, card_payment, State)
 ->
    {next_state, paid, State#state{card = Card},
     [{reply, From, {ok, paid}}, hibernate]};
handle_event({call, From}, _Event, card_payment, _State) ->
    {keep_state_and_data,
     [{reply, From, {error, "card required"}}]};

handle_event({call, From}, {account, Account}, debit_payment,
 State) ->
    {next_state, paid, State#state{account = Account},
     [{reply, From, {ok, paid}}, hibernate]};
```

```
handle_event({call, From}, _Event, debit_payment, _State) ->
    {keep_state_and_data,
     [{reply, From, {error, "account required"}}]};

handle_event({call, From}, _Event, paid, _State) ->
    {keep_state_and_data, [{reply, From, {error,
 already_paid}}, hibernate]}.

terminate(normal, paid, #state{payment_method = card} = State)
 ->
    io:format("~p Name: ~s~nCard: ~s~nPaid.~n",
              [self(), State#state.name, State#state.card]),
    ok;
terminate(normal, paid, #state{payment_method = debit} =
 State) ->
    io:format("~p Name: ~s~nAccount: ~s~nPaid.~n",
              [self(), State#state.name,
 State#state.account]),
    ok;
terminate(normal, _StateName, _StateData) ->
    io:format("~p No paid.~n", [self()]),
    ok.

code_change(_OldVsn, StateName, StateData, _Extra) ->
    {ok, StateName, StateData}.
```

To run it we open a shell and follow these steps:

```
> c(payment).
{ok,payment}
> {ok, PID} = payment:start_link().
{ok,<0.133.0>}
> payment:get_info(PID).
{credentials,{state,undefined,undefined,undefined,
                    undefined}}
> payment:give_name(PID, "Manuel").
ok
> payment:get_info(PID).
{payment_method,
{state,"Manuel",undefined,undefined,undefined}}
> pago:give_payment_method(PID, account).
{error,"payment method: card or debit"}
> pago:give_payment_method(PID, debit).
ok
> pago:get_info(PID).
{debit_payment,{state,"Manuel",debit,undefined,
                      undefined}}
> pago:give_card(PID, "1234").
{error,"account required"}
> pago:give_account(PID, "1234").
{ok,paid}
> pago:get_info(PID).
{paid,{state,"Manuel",debit,undefined,"1234"}}
> pago:stop(PID).
<0.133.0> Name: Manuel
Account: 1234
Paid.
ok
```

We can see the progression of the payment requests until reaching the stoppage of the process and obtain all the information. We can try to stop the process beforehand and see the state in which the operation remains. You can make all the tests and changes you want.