Erlang/OTP

VOLUMEN II

Las bases de OTP

2ª edición

MANUEL ÁNGEL RUBIO JIMÉNEZ

Erlang/OTP

Volumen II: Las Bases de OTP

Manuel Angel Rubio Jimenez

Muestra gratuita

Adquiere el libro completo aquí:

https://altenwald.com/book/erlang-ii

Erlang/OTP
Volumen II: Las Bases de OTP

Manuel Angel Rubio Jimenez

Resumen

El desarrollo en Erlang se fundamenta en dos bases bien definidas por sus creadores. La primera son la potencia de los procesos que implementa la máquina virtual de Erlang. La segunda es la metodología de Programación Orientada a la Concurrencia que se facilita con el framework OTP.

El uso de Erlang queda incompleto si no se emplean ambas cosas. Su potencia no se descubre hasta no haber hecho uso de los procesos como fuente principal de programación y de los ***comportamientos*** que integra el framework OTP. Para la realización de un proyecto profesional en Erlang es indispensable el conocimiento y dominio de esta tecnología.

Este segundo volumen de Erlang/OTP cubre el conocimiento de este framework, la creación de proyectos profesionales con él y la Programación Orientada a la Concurrencia o Modelo Actor como se conoce más extensamente. Termina el recorrido necesario para conocer las potencias del lenguaje y su plataforma. Da al lector un recorrido por la teoría y la práctica, dándole aún más herramientas y más código útil para lanzarse a desarrollar.

Además en esta segunda edición cubrimos hasta la versión 25 de Erlang/OTP, agregamos un nuevo capítulo sobre logger y mejoramos el capítulo sobre distribución.

9 788412 452020

Erlang/OTP: Volumen II: Las Bases de OTP por Manuel Angel Rubio Jimenez[1] se encuentra bajo una Licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported[2].

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

Capítulo 5. Máquinas de Estados

Controlar la complejidad es la esencia de la programación.
— Brian Kernigan

El comportamiento de ***gen_statem*** se basa en la máquina de estados de Mealy. En versiones anteriores a OTP 20 existía otro comportamiento llamado ***gen_fsm*** que implementaba una máquina de estados de Moore. La implementación de la máquina de Mealy supuso una mejora en rendimiento para todas las librerías de OTP.

Este comportamiento está construido sobre el comportamiento del servidor (***gen_server***) y dispone de las mismas dinámicas vistas en el capítulo anterior agregando además la gestión de estados a nivel de comportamiento. La diferencia entre el servidor y la máquina de estados es que un evento recibido por un servidor es gestionado en una sola dimensión, el código implementado y los datos del único estado existente. En la máquina de estados el código que atiende cada petición, cada evento, depende del estado interno de la máquina de estados.

Es decir, si una máquina tiene dos estados (por ejemplo abierto y cerrado), podemos desarrollar en nuestro sistema dos versiones del servidor, uno identificando qué haríamos en el caso de estar en el primer estado (abierto) y sin tener en cuenta el otro estado, y después pasaríamos a desarrollar cómo debe comportarse el servidor en el segundo estado (cerrado) e igualmente sin tener en cuenta el estado anterior. Obviamente y gracias a la concordancia podemos aprovechar los eventos o peticiones comunes que sean independientes del estado.

Intentaremos explotar al máximo las características disponibles en este comportamiento a través de los ejemplos del semáforo, el ascensor y el sistema de pago. Veremos estos ejemplos a lo largo del capítulo.

1. Eventos, Estados y Diagramas de Estados

Una de las herramientas más importantes en la generación de una máquina de estados finitos son los diagramas de estados. Estos diagramas nos proporcionan en un vistazo todos los eventos que podemos recibir, todos los estados y todas las transiciones entre estados posibles.

Los **estados** en estos diagramas se representan con una circunferencia (o un círculo) con el nombre del estado escrito dentro. Idealmente todos tienen el mismo tamaño y el mismo color. Para facilitar la lectura en

diagramas con muchos estados podemos cambiar el color de algunos de ellos para formar grupos visuales.

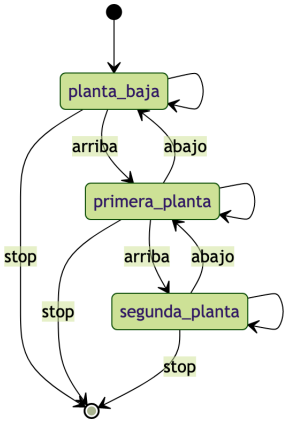
Los **iniciadores y terminadores** son líneas verticales u horizontales de cierto grosor. Indican el inicio y/o fin de la máquina de estados.

Los **eventos** son las líneas que permiten hacer el tránsito de un estado a otro. Estas flechas parten de un estado o iniciador y terminan en otro estado o finalizador. La punta de la flecha marca su dirección. A lo largo de su trayectoria se escribe el nombre del evento.

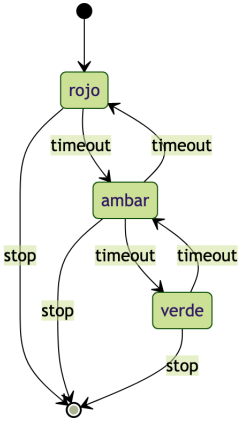
Los eventos también pueden hacer bucles, es decir iniciar y terminar en el mismo estado indicando que aunque se ha recibido el evento y se realiza una acción, no se modifica el estado.

Los eventos generalmente son acciones que se realizan por una entrada de datos sobre la máquina de estados. Esto puede significar la recepción de un mensaje o incluso el disparo de un evento de tiempo.

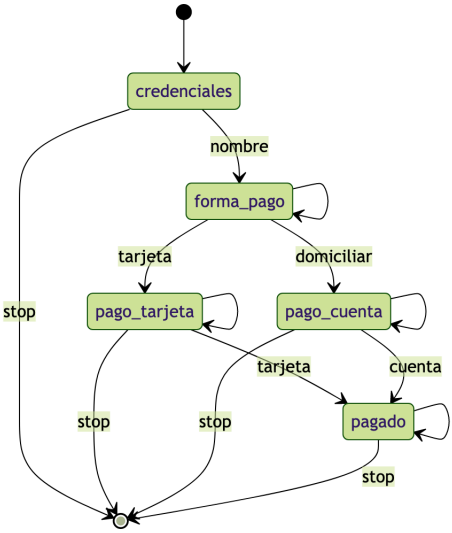
Para nuestros ejemplos vamos a dibujar sus diagramas de estados. El ejemplo del ascensor tendrá el siguiente diagrama:



El ejemplo del semáforo mostrará el siguiente diagrama:



Por último, el ejemplo de nuestro proceso de pago tendrá el siguiente diagrama:



Ahora veremos como implementarlos.

2. Máquina de Estados de Mealy

La teoría nos dice que cualquier máquina de estados Moore puede ser representada por una máquina de estados Mealy, pero no todas las máquinas de estados de Mealy pueden ser representadas por una máquina de estados Moore.

Esto se debe a la particularidad especial de las máquinas de Mealy de combinar acciones con el estado para obtener el resultado del cambio de estado para un evento entrante. Mientras la máquina de estados de Moore puede representarse como:

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

Para cada evento procesado en un estado determinado obtenemos el mismo u otro estado diferente de salida.

La máquina de Mealy agrega acciones a ser realizadas en el resultado:

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

De esta forma no se retorna solo el cambio de estado sino también una o varias acciones a realizar.

A través de las retro-llamadas el servidor puede retornar no solo un cambio de estado para el proceso sino también el retorno a realizar para ser enviado al cliente. En el caso de la máquina de estados las acciones no solo incluyen la respuesta al usuario sino también temporizadores, siguientes eventos a ser disparados inmediatamente similar al efecto de ***continue***, postponer eventos o cambiar el módulo en uso.

En nuestro ejemplo del semáforo establecemos temporizadores. En la máquina Mealy establecer el temporizador solo basado en el estado actual y sin afectar los nuevos eventos recibidos es posible. Lo veremos más adelante.

Nuestro ejemplo del ascensor podría tener botones numéricos con el número de planta. Las acciones nos darían soporte para movernos a otra planta y ejecutar una acción de comprobación para saber si debe parar o continuar el ascensor hasta la siguiente parada. Puede ser un interesante ejercicio.

2.1. Plantilla de una Máquina de Estados

La forma de usar ***gen_statem*** difiere según el modo de manejar los eventos que decidamos emplear. Podemos elegir entre funciones de estado o función manejadora de eventos.

La elección se realiza a través de la definición de la retro-llamada `callback_mode/0`. Esta retro-llamada debe retornar uno de los siguientes valores:

`handle_event_function`

Define el uso de una única función `handle_event/4` como punto de entrada para todos los eventos recibidos en cualquier estado.

`state_functions`

Permite usar tantas funciones diferentes como estados haya definidos. Además de tener el nombre del estado estas funciones disponen de otros tres parámetros.

El retorno de la retro-llamada puede ser cualquiera de estos dos átomos o una lista conteniendo alguno de los dos átomos y otras opciones más de configuración. De momento solo existe una opción llamada ***state_enter***. Veremos más adelante qué hace ***state_enter***.

3. Plantilla para Funciones de Estado

Esta plantilla emplea funciones con el nombre de los estados. Deben agregarse dependiendo de los nombres de nuestros estados. Como ejemplo en la plantilla se han definido dos estados ***encendido*** y ***apagado***:

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

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

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

callback_mode() -> state_functions.

-record(state_data, {}).

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

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

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

encendido(EventType, EventContent, Data) ->
    {next_state, apagado, Data}.

apagado(EventType, EventContent, Data) ->
```

```
    {next_state, encendido, Data}.
```

3.1. Plantilla para Función Manejadora de Eventos

Esta plantilla no requiere definir funciones de estados todas las peticiones entrarán a `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, apagado, #state_data{}}.

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

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

handle_event(EventType, EventContent, encendido, Data) ->
    {next_state, apagado, Data}.

handle_event(EventType, EventContent, apagado, Data) ->
    {next_state, encendido, Data}.
```

4. Ciclo de vida de un evento

El ciclo de vida de una máquina de estados se define por sus transiciones y cada transición sucede cuando se recibe un evento. Veamos con detalle qué sucede en la máquina de estados al recibir un evento por pasos. Independientemente del evento, los pasos son los siguientes:

1. Recibimos un evento. El evento se descompone para indicar en la retro-llamada el tipo de petición y los datos. Agregamos los datos del estado y el nombre del estado y ejecutamos la función.

2. Recibimos retorno de la función. La retro-llamada se ejecutó y nos proporcionó qué hacer a continuación:

   a. Nos mantenemos (`keep_state`). No hay cambio de estado.

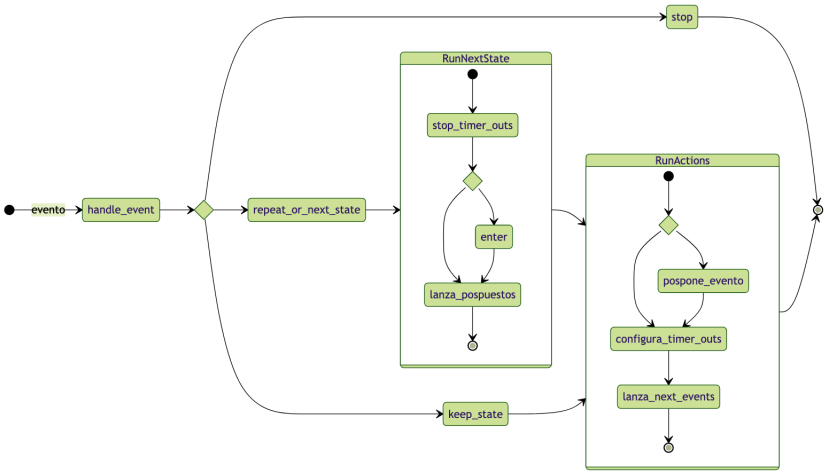
   b. Cambiamos o repetimos estado (`next_state` o `repeat_state`). Cambiamos el estado, implica las siguientes tareas:

- Se detienen todos los temporizadores de estado.
- Si se configuró ***state_enter*** se agrega como acción la ejecución de un evento `enter` con el dato de evento del estado anterior.
- Comprueba si hay eventos pospuestos para repetirlos.

c. Detenemos la máquina de estados (`stop`).

3. Ejecutamos las acciones definidas en el retorno de la retro-llamada:
   - Pospone. El evento actual entra en la cola de eventos pospuestos para ser relanzados cuando cambiemos de estado.
   - Temporizadores. Configura los temporizadores indicados en las acciones.
   - Agrega los eventos (`next_event`) para ser lanzados inmediatamente.

A lo largo del flujo vemos cómo pueden sucederse nuevos eventos del procesamiento del cambio de estado o las acciones de tipo `next_event`. Estos nuevos eventos se ejecutarán antes de volver al ciclo de escucha de nuevos eventos entrantes.



5. Iniciando la Máquina de Estados

Para iniciar la máquina de estados solo necesitamos ejecutar la función `gen_statem:start_link/3-4`, `gen_statem:start/3-4` o [gen_statem:start_monitor/3-4] dependiendo del tipo de enlace con el proceso que iniciará el proceso. Podemos ver la definición de los parámetros para esta función a continuación:

```
-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().
```

Nada nuevo. Los parámetros a emplear cambiarán según el ejemplo a implementar. En nuestro ejemplo del ascensor podemos lanzar un proceso de máquina de estados de la siguiente forma:

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

Esta ejecución nos permite generar un proceso con el nombre del módulo y acceder a él a través de ese nombre.

El proceso de pago podemos iniciarlo de forma anónima:

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

De esta forma podemos generar tantos como necesitemos. Por último, el proceso del semáforo lo iniciaremos pasándole parámetros:

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

La función de inicio de la máquina de estados `init/1` tiene un parámetro. Los parámetros son rellenados con la información pasada en la función `gen_statem:start_link/3-4`. El retorno incluirá los datos de estado y además el nombre del estado:

```
-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().
```

Las opciones relacionadas al servidor pueden verse en la Sección 2, "Iniciando el servidor". La aportación de ***gen_statem*** es la agregación

de ***statename()*** y ***actions()***. El nombre del estado nos permite configurar el estado inicial, el punto de entrada para el primer evento de nuestro sistema. Las acciones definen una serie de actividades a realizar antes de atender al siguiente evento. Veremos más adelante todos los tipos de acciones disponibles.

La implementación para nuestro ejemplo del ascensor es la siguiente:

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

En nuestro semáforo la inicialización obtendrá dos parámetros. Definiremos antes un registro para el estado y la función de la siguiente forma:

```
-record(state, {
    tiempo_verde :: pos_integer(),
    tiempo_rojo :: pos_integer(),
    siguiente :: verde | rojo
}).

init([TiempoVerde, TiempoRojo]) ->
    {ok, rojo, #state{
        tiempo_verde = TiempoVerde,
        tiempo_rojo = TiempoRojo
    }, [{state_timeout, TiempoRojo, timeout}]}.
```

Iniciamos el semáforo en rojo y configuramos los tiempos de espera para cada uno de los colores. Además, agregamos una acción para crear un temporizador de estado (***state_timeout***) para provocar un evento de tiempo (***timeout***) al acabar el tiempo de espera. Veremos esta acción de temporizador y el resto de acciones de temporizadores en la Sección8, "Temporizadores".

Por último, la implementación de la función `init/1` para nuestro sistema de pago. La toma de datos del sistema de pago consiste en varios pasos. Esa información debe almacenarse en el estado:

```
-record(state, {
    nombre :: string(),
    email :: string(),
    precio :: float(),
    tarjeta :: string(),
    direccion :: string()
}).

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

Comenzamos en ***identifica*** y esperamos la llegada de los eventos.

6. Eventos y Cambio de Estado

Una vez tenemos nuestra máquina de estados iniciada y en el primer estado estamos preparados para atender eventos. Los eventos serán recibidos en la retro-llamada `handle_event/4` o las funciones con nombre de estado para la gestión de los eventos. Depende del tipo de retro-llamada elegido. Para los ejemplos voy a emplear únicamente el tipo de funciones con nombre del estado (***state_functions***).

Nota

En mi experiencia el modo ***state_functions*** es más simple de entender y funciona bien a modo de aprender cómo funciona la máquina de estados. Sin embargo, cuando implementemos máquinas de estados, el modo ***handle_event_function*** tiende a ser más potente porque guarda similitud con las retro-llamadas de ***gen_server*** (`handle_call/3` y `handle_cast/2`) y permite realizar concordancias de eventos y estados en combinación.

Recordemos que las funciones tendrán tres parámetros. El primer parámetro se encarga del tipo de evento, el segundo será el contenido del evento y el tercer parámetro son los datos del estado del proceso. Vamos a listar los tipos de eventos que podemos recibir en estas funciones, es decir, el contenido posible de recibir en el primer parámetro:

`{call, from()}`

Llamada síncrona enviada a la máquina de estados desde el proceso agregado como segundo parámetro de la tupla. Es enviada usando la función `gen_statem:call/2` o `gen_statem:send_request/2`.

`cast`

Llamada asíncrona enviada a la máquina de estados. Es enviada usando la función `gen_statem:cast/2`.

`info`

Información recibida en el proceso sin formato específico. Es enviada usando `erlang:send/2` o la sintaxis específica para envío de mensajes usando el signo de exclamación (!).

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

Tiempo de espera agotado. Este evento es recibido por un temporizador generado a través de alguna de las acciones de tiempo de espera de la máquina de estados.

`internal`

> Este evento es posible generarlo únicamente desde la sección de acciones en la tupla de retorno de las funciones de estados o la función manejadora de eventos. Veremos más adelante cómo generar este evento.

Como eventos recibimos no solo información de temporizadores o mensajes sino también información estructurada como ***call*** (llamadas síncronas) y ***cast*** (llamadas asíncronas) tal y como las vimos en ***gen_server***.

En nuestro ejemplo del ascensor podemos ver la implementación de las funciones de estado empleando ***cast*** para la recepción de eventos de las funciones `boton_arriba/0` y `boton_abajo/0` de la siguiente forma:

```
planta_baja(cast, up, StateData) ->
    io:format("subiendo al primer piso~n", []),
    {next_state, primera_planta, StateData};

planta_baja(cast, down, StateData) ->
    io:format("beeeep! no se puede bajar~n", []),
    {next_state, planta_baja, StateData}.

primera_planta(cast, up, StateData) ->
    io:format("subiendo al segundo piso~n", []),
    {next_state, segunda_planta, StateData};

primera_planta(cast, down, StateData) ->
    io:format("bajando a planta baja~n", []),
    {next_state, planta_baja, StateData}.

segunda_planta(cast, up, StateData) ->
    io:format("beeeep! no se puede subir~n", []),
    {next_segunda, segunda_planta, StateData};

segunda_planta(cast, down, StateData) ->
    io:format("bajando al primer piso~n", []),
    {next_state, primera_planta, StateData}.
```

Podemos cambiar de estado siempre a través del retorno de una retrollamada cuando sea ejecutada. Las combinaciones de que disponemos son las siguientes:

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

> Cambiará el estado de nuestra máquina de estados a uno nuevo. El nuevo estado debemos agregarlo como segundo elemento de la tupla. El estado puede ser uno nuevo o el mismo. Continuar en el mismo estado no se considera un cambio para ***next_state*** y es preferible emplear alguna de las siguientes opciones. Las acciones son opcionales y pueden ser omitidas usando así una tupla de tan solo tres elementos.

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

Mantenemos el estado actual de la máquina de estados pudiendo cambiar tan solo los datos del estado. Las acciones son opcionales y pueden ser omitidas usando así una tupla de tan solo dos elementos.

`keep_state_and_data | {keep_state_and_data, actions()}`

Mantenemos el estado actual y la información del estado actual. Nos permite hacer retornos más compactos y más legibles. Las acciones son opcionales y en caso de no indicarlas podemos retornar el átomo ***keep_state_and_data*** únicamente.

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

Repetimos el estado actual. A diferencia de mantener nuestro estado anterior con esta opción el sistema supone una salida del estado y vuelta a entrar. Si habíamos agregado la opción ***state_enter*** en la función `callback_mode/0` generará la ejecución del estado de nuevo con el evento ***enter***. Las acciones son opcionales y pueden ser omitidas.

`{repeat_state_and_data, actions()}`

Repetimos el estado actual y los datos del estado. Al Igual que el caso anterior nos permite salir y volver a entrar del estado generando un evento ***enter*** en caso afirmativo. Las acciones son opcionales y pueden ser omitidas empleando en su lugar únicamente el átomo ***repeat_state_and_data***.

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

Detiene la ejecución de la máquina de estados y pasa la ejecución a la retro-llamada `terminate/3`. Especificar la información de estado es opcional.

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

Detiene la ejecución de la máquina de estados y pasa la ejecución a la retro-llamada `terminate/3`. Antes envía la respuesta al proceso llamante. Especificar la información de estado es opcional.

Aunque en la definición de la máquina de Mealy el retorno de cada evento es un nuevo estado y acciones en realidad el estado podemos mantenerlo y las acciones podemos obviarlas si no tenemos acciones que ejecutar tal y como vimos en el ejemplo del ascensor más arriba.

En la implementación de nuestro semáforo tenemos los eventos de temporizadores de estado y llamada síncrona (***call***) junto con retornos

donde podemos ver el cambio de estado (***next_state***) y la forma de mantener el estado y los datos del estado (***keep_state_and_data***) además de las acciones de respuesta (***reply***) y temporizador de estado (***state_timeout***):

```
handle_event(state_timeout, {cambia, verde}, rojo, State) ->
    {next_state, ambar, State,
     [{state_timeout, ?TIEMPO_EN_AMBAR, {cambia, rojo}}]};

handle_event(state_timeout, {cambia, verde}, ambar, State) ->
    #state{tiempo_verde = TiempoVerde} = State,
    {next_state, verde, State,
     [{state_timeout, TiempoVerde, {cambia, rojo}}]};

handle_event(state_timeout, {cambia, rojo}, ambar, State) ->
    #state{tiempo_rojo = TiempoRojo} = State,
    {next_state, rojo, State,
     [{state_timeout, TiempoRojo, {cambia, verde}}]};

handle_event(state_timeout, {cambia, rojo}, verde, State) ->
    {next_state, ambar, State,
     [{state_timeout, ?TIEMPO_EN_AMBAR, {cambia, verde}}]};

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

En el ejemplo de pago solo empleamos llamadas síncronas (***call***) al igual que la llamada ***ver_semaforo*** vista más arriba. Puedes ver su implementación al final de este capítulo en la Sección16, "Ejemplo de Pago".

7. Acciones

Hemos podido ver en el retorno de las retro-llamadas la posibilidad de indicar unas acciones. En esta sección veremos los tipos de acciones que podemos agregar en este retorno.

Podemos organizar las acciones en grupos. Estos grupos no son excluyentes. Podemos agregar cada acción a ninguno o varios grupos. También veremos acciones no pertenecientes a ningún grupo. Los grupos son:

Acciones de entrada

Estas acciones incluyen las acciones de hibernación, de temporización y de respuesta.

Acciones de temporización

Estas acciones incluyen los temporizadores de evento, genéricos y de estado.

Acciones de transición

Estas acciones incluyen posponer un evento, hibernación, temporizador de evento, temporizador genérico y temporizador de estado.

Nota

Estos grupos han sido definidos a través de tipos en el código fuente del módulo ***gen_statem***. En la documentación de Erlang/OTP[1] para este módulo podemos ver claramente estos tipos y qué incluyen.

Las acciones se acumulan en una lista. Esta lista podría contener acciones duplicadas o incompatibles entre sí. Como norma general las acciones de transición se sobrescriben entre sí y solo es tenida en cuenta la última que especifiquemos.

Otras acciones como el envío de un evento (***next_event***) pueden agregarse tantas como se necesiten.

Vamos a listar todas las posibles acciones de que disponemos:

`postpone | {postpone, boolean()}`

Nos permite posponer el tratamiento de un evento. Esta acción sitúa en una cola de mensajes pospuestos a procesar únicamente cuando se cambie de estado. Muy útil si nos encontramos en un estado donde un tipo de evento aún no puede ser procesado, por ejemplo en un estado de desconexión.

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

Nos permite agregar un nuevo evento en la cola de eventos para ser procesado por la máquina de estados. De este tipo podemos agregar tantas acciones como necesitemos. Si necesitamos diferenciar entre los eventos enviados por elementos externos vía ***call***, ***cast***, ***info*** o ***timeout*** podemos emplear ***internal***. De esta forma conocemos la procedencia inequívocamente del evento.

`hibernate | {hibernate, boolean()}`

Envía el proceso a hibernación. Este estado dura hasta recibir un nuevo evento. Si disponemos de muchos procesos de máquina de estados y la atención de eventos se dilata mucho en el tiempo para cierto estado podemos agregar esta acción, comprimir así el uso de memoria y reducir el consumo de procesador[2].

[1] http://erlang.org/doc/man/gen_statem.html
[2] No obstante si hay un gran número de eventos recibidos de forma constante la hibernación podría ser una penalización al rendimiento. Usa esta opción con cuidado.

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

Crea un temporizador de evento o sin nombre. Veremos los temporizadores más adelante junto con sus opciones. El tipo de dato ***timeout()*** puede indicar un número entero positivo o el átomo ***infinity***.

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

Crea un temporizador genérico o con nombre. Veremos los temporizadores más adelante junto con sus opciones. Las opciones en esta especificación son opcionales.

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

Crea un temporizador de estado. Veremos los temporizadores más adelante junto con sus opciones. Las opciones en esta especificación son opcionales.

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

Envía una respuesta al proceso llamante. Debemos tener el dato ***From***. Normalmente este tipo de acción se usa únicamente en funciones donde se ha recibido un evento de tipo ***call***.

`{change_callback_module, module()}`

Cambia el módulo para la ejecución de las siguientes retro-llamadas. Esta función es útil si la implementación de nuestra máquina de estados es muy grande y queremos escribir las funciones específicas de cada estado en un módulo diferente.

`{push_callback_module, module()}`

Realiza una acción similar al caso anterior pero mantiene una pila con los módulos donde hemos ido cambiando. Esto permite diseñar máquinas de estados multinivel donde el módulo actúa como otra dimensión dentro de los estados y permite definir los estados por grupos. Veremos este concepto más adelante.

`pop_callback_module`

Nos permite volver al grupo de estados anterior. Es la operación complementaria a `push_callback_module`.

Dentro de nuestros ejemplos tal y como vimos en la sección anterior empleábamos las acciones de temporización de estado (***state_timeout***) así como la acción de respuesta (***reply***).

Desarrollaremos un poco más adelante la siguiente versión del semáforo donde emplearemos más acciones como ***hibernate***.

También podemos ver dos versiones de algunas acciones donde aparecen en forma de tupla. Recordemos que las acciones son una lista de propiedades (ver módulo ***proplists***). La lista de propiedades permite la duplicidad de sus elementos y dependiendo de la función para obtener los datos, podemos obtener todos los valores bajo una misma clave, o el primer valor de la lista. Por lo que, si necesitamos evitar posponer un evento o entrar en hibernación, podemos agregar a la lista de acciones:

```
NewAcciones =
    [{hibernate, false}, {postpone, false} | Acciones],
```

8. Temporizadores

A diferencia de los otros comportamientos en la máquina de estados podemos encontrar diferentes tipos de temporizadores. Cada tipo actúa de forma diferente y está implementado de forma diferente.

Los tipos que podemos encontrar son los siguientes:

temporizador de evento

Este temporizador es similar al que podemos encontrar en cualquier otro comportamiento. Solo se dispara si después de especificarlo no se sucede ningún otro evento en ese espacio de tiempo. Es decir, es un tiempo de espera ideal para medir inactividad en el proceso.

temporizador de estado

Actúa sobre el estado. Mientras el estado no cambie seguirá contando el tiempo hasta expirar y generar el evento ***status_timeout***. Útil para limitar la permanencia en un estado por ejemplo de usuario no autenticado en protocolos como IMAP o XMPP.

temporizador genérico

Este temporizador requiere la especificación de un nombre. Si antes de generar la llamada ***{timeout, name()}*** se genera otro temporizador genérico con el mismo nombre, el contador es reiniciado. Puede ser útil para reaccionar en tiempos específicos a sistemas de ***ping*** donde se requiere una acción del usuario o se cierra el canal de comunicación.

En caso de emplear ***infinity*** como tiempo de espera en lugar de configurar el temporizador este es ignorado o eliminado. Si el valor

pasado es cero (0) tampoco se genera un temporizador en su lugar se genera directamente el evento.

Las acciones de temporización no solo nos permitirán configurar el tipo de temporizador sino también la información a lanzar cuando se produzca el evento de tiempo de espera agotado. Por tanto los eventos recibidos como tipo de evento y contenido de evento serán:

timeout, event_content()

> Para acciones lanzadas como ***{timeout, 100, tick}*** se recibirá como tipo de evento ***timeout*** y como contenido del evento ***tick***.

{timeout, name()}, event_content()

> Para acciones lanzadas como ***{{timeout, clock}, 100, tick}*** se recibirá como tipo de evento ***{timeout, clock}*** y como contenido del evento ***tick***.

state_timeout, event_content()

> Para acciones lanzadas como ***{state_timeout, 100, tick}*** se recibirá como tipo de evento ***state_timeout*** y como contenido del evento ***tick***.

Podemos ver un caso de ejemplo en el código del semáforo:

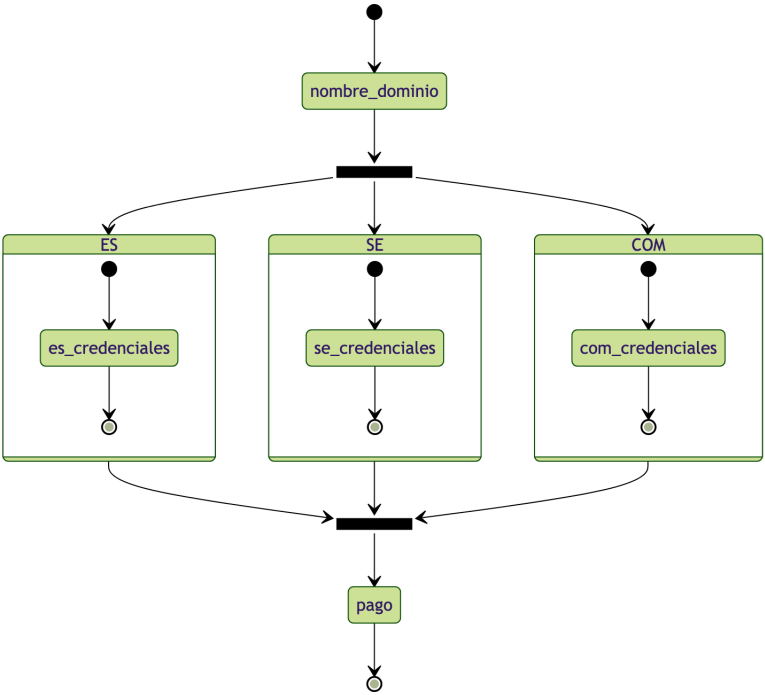
```
handle_event(state_timeout, {cambia, verde}, rojo, State) ->
    {next_state, ambar, State,
     [{state_timeout, ?TIEMPO_EN_AMBAR, {cambia, rojo}}]};
```

Recibimos un evento de temporizador de estado (***{cambia, verde}***) y generamos una acción con otro temporizador de estado. Esta vez para cambiar a rojo. De esta forma mantenemos el flujo de cambio entre los colores del semáforo y los tiempos de espera.

9. Grupos de Estados

Estamos en el proceso de compra de un dominio. Los dominios tienen características específicas cuando tratamos comprar dominios de algún país concreto. Por ejemplo, para comprar un dominio .es el sistema nos requiere agregar nuestro número de identificación español (DNI o CIF), para un dominio .se requerimos tener un ID sueco e igual para otros dominios. Cada dominio tiene sus datos específicos con sus validaciones y peculiaridades y sin embargo hay una toma de datos común y solo una parte del proceso diferente.

Aunque podríamos resolver este problema de muchas formas diferentes, vamos a crear una máquina de estados para la compra de esta forma:



Vemos los grupos formados por los dominios específicos en este ejemplo. No obstante, los grupos pueden indicar un fraccionamiento superior en cada uno de los grupos haciendo que un proceso con decenas de estados pueda fraccionarse y gestionarse mejor gracias a su división y encapsulación.

Al mismo tiempo, grupos de estados pueden reutilizarse en diferentes localizaciones a lo largo del flujo de trabajo permitiendo el uso de ***push*** y ***pop***. Un ejemplo podría ser el análisis léxico de un texto porque las reglas se asocian como árboles sintácticos y pueden saltar de un estado a otro en cada nivel llegando a profundizar mucho. Una pila en este caso permitiría apilar los estados para después regresar en el camino inverso.

10. Finalizando la Máquina de Estados

La finalización de la máquina de estados se realiza a través de retornar ***stop*** en una retro-llamada o llamando a la función `gen_statem:stop/1` o `gen_statem:stop/3`.

En la implementación de nuestros ejemplos hemos empleado tan solo la primera opción considerando la finalización como ***normal***. La implementación en el ejemplo del ascensor y el semáforo es:

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

La finalización para el ejemplo del pago difiere por iniciarse de forma anónima. La implementación es:

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

Al ejecutar estas funciones para salir de la ejecución de las máquinas de estados se ejecuta la función `terminate/3`. Los parámetros de esta función son la razón de terminación. En nuestros tres ejemplos será siempre ***normal***. La única forma de recibir otra razón diferente sería finalizando el proceso a través de la función `exit/2` para especificar otra razón diferente.

La especificación de la retro-llamada `terminate/3` es la siguiente:

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

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

Nota

Esta retro-llamada es opcional. Por defecto ***gen_statem*** no requiere esta función, si no es definida se procede directamente a terminar el proceso.

Para nuestro ejemplo de pago tenemos la implementación de esta función como sigue:

```
terminate(normal, pagado, #state{forma_pago = tarjeta} =
 State) ->
    io:format("~p Nombre: ~s~nTarjeta: ~s~nPagado.~n",
              [self(), State#state.nombre,
 State#state.tarjeta]),
    ok;

terminate(normal, pagado, #state{forma_pago = domiciliar} =
 State) ->
    io:format("~p Nombre: ~s~nCuenta: ~s~nPagado.~n",
              [self(), State#state.nombre,
 State#state.cuenta]),
    ok;
```

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

Ofrecemos información sobre la finalización del proceso de pago dependiendo de la información almacenada en el proceso.

11. Cambio de código en caliente

Para el cambio en caliente hemos desarrollado dos versiones del ascensor. En principio, el primer código del ascensor realmente es un elevador con dos botones, uno para subir a una planta superior y otro para bajar a una planta inferior validando si es posible hacerlo o no.

Hemos desarrollado una segunda versión que realmente actúa como un ascensor. Mantenemos las funciones `boton_arriba/0` y `boton_abajo/0` para mantener compatibilidad hacia atrás. Implementamos la función `boton_num/1` para la funcionalidad real de presionar un botón numérico.

El nuevo código mantiene el ascensor en movimiento subiendo y bajando dependiendo de los botones presionados. Si no hay más botones presionados se mantiene en la planta hasta recibir una nueva orden. Además, el número de plantas se incrementa a 10 puesto que los estados se han modificado para ser: ***stopped*** (parado), ***going_up*** (hacia arriba) y ***going_down*** (hacia abajo).

Como hemos hecho con otros cambios de código en caliente hemos agregado en las cabeceras de cada ejemplo de código una versión para facilitar el cambio de una versión a la siguiente:

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

En el fichero de la segunda versión tenemos que agregar lo mismo pero sumando un número a la versión:

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

Los cambios del código son bastante extensos. En principio cambiamos no solo los estados sino también la información guardada dentro del estado. En la segunda versión nuestro estado es como sigue:

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

```
}).
```

El uso de un conjunto en la lista de botones presionados nos garantiza la eliminación de los elementos duplicados y no tener que preocuparnos por ellos. Además, al cambiar el diagrama de estados de la planta en la que se encuentra el ascensor al estado en sí del motor (parado, subiendo o bajando) debemos almacenar la planta actual en los datos de estado.

El código correspondiente a la actualización para el segundo fichero debemos implementarlo en la función `code_change/4`. Esta función tiene la siguiente definición:

```
-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().
```

En la primera versión del código la implementación está vacía. El código en las primeras versiones no se usa puesto no va a migrar ni cambiar información de ninguna versión anterior. En la segunda versión del código es como sigue:

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

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

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

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

Cubrimos la actualización de la versión `1` según el estado en que se encuentre y también el supuesto cambio hacia atrás en caso de realizar una marcha atrás para volver a la versión `1`.

Vamos a probar nuestro código desde una consola. Damos por supuesto que tenemos el código de la primera versión en el directorio `ascensor_v1` y la segunda versión en el directorio `ascensor_v2` para poder realizar nuestra prueba.

Abrimos una consola y compilamos el primer código. Lanzamos el proceso:

```
> c("ascensor_v1/ascensor.erl").
{ok,ascensor}
> ascensor:start_link().
iniciamos en el primer piso
{ok,<0.64.0>}
> ascensor:boton_arriba().
subiendo al primer piso
ok
> ascensor:boton_arriba().
subiendo al segundo piso
ok
```

Tenemos el proceso en ejecución y nuestro elevador en el segundo piso. Ahora cambiaremos el código para obtener la funcionalidad de un ascensor real. Para ello ejecutamos:

```
> sys:suspend(ascensor).
ok
> c("ascensor_v2/ascensor.erl").
{ok,ascensor}
> sys:change_code(ascensor, ascensor, 1, []).
ok
> sys:resume(ascensor).
ok
> ascensor:boton_num(5).
* cerrando puertas y subiendo
ok
< pasando por planta 3
< pasando por planta 4
> parando en planta 5, abriendo puertas
```

El código se ha ejecutado y ha cambiado el estado interno del proceso para agregar el nuevo estado donde la planta actual es la segunda. El ascensor entra en modo detenido (***stopped***) y al ejecutar el nuevo comando para desplazarlo a la quinta planta sube desde la segunda hasta la quinta sin problema.

Si necesitásemos volver al código del elevador podríamos ejecutar el siguiente código:

```
> sys:suspend(ascensor).
ok
> sys:change_code(ascensor, ascensor, {down, 1}, []).
ok
> c("ascensor_v1/ascensor.erl").
{ok,ascensor}
> sys:resume(ascensor).
ok
> sys:get_state(ascensor).
{segunda_planta,{state}}
> ascensor:boton_abajo().
bajando al primer piso
```

```
ok
```

En esta ocasión debemos ejecutar la acción de vuelta atrás (`sys:code_change/4`) antes de la compilación y carga del código anterior. Vemos que el estado de la función cambia correctamente acorde a lo desarrollado y una vez volvemos a recuperar la ejecución del proceso el funcionamiento es correcto.

12. Obteniendo información de la Máquina de Estados

En Sección 11, "Obteniendo información del servidor" hablamos de cómo mostrar la información de un servidor. A todos los efectos una máquina de estados es un servidor. No obstante la información interna cambia sutilmente al integrar el nombre del estado.

La especificación de la función `format_status/2` igual en tanto vemos dos parámetros pero la forma de llamar a esta función con respecto al segundo parámetro difiere. La especificación de la función es como sigue:

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

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

Nota

Esta retro-llamada es opcional. Por defecto ***gen_statem*** implementa un mecanismo para retornar el estado del proceso si esta función no está definida.

Importante

Esta función está disponible a partir de OTP 25. En versiones anteriores existe una versión anterior que recibe dos parámetros `format_status/2`. La diferencia radica en la organización de la información, la nueva función emplea un mapa y por tanto es más fácil acceder a los datos realizando incluso concordancia en los parámetros de la función.

Esta retro-llamada será empleada únicamente cuando empleemos la función `sys:get_state/1`. Por defecto obtendremos una tupla con los elementos del nombre de estado y los datos del estado:

```
> sys:get_state(ascensor).
{primera_planta,{state}}
```

Podemos ver un ejemplo del volcado por defecto de la información de una máquina de estados a continuación usando la función `sys:get_status/1`:

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

Podemos ver que hay una sección con la clave ***data*** (primer elemento de la tupla) al final que nos dice su ***State*** a través de una tupla de dos elementos cuyo primer elemento es el nombre del estado y el segundo elemento los datos del estado.

En la entrada ***data*** anterior podemos ver otros datos como ***Postponed*** donde se almacenarán los eventos pospuestos a través de la acción específica vista en la sección de acciones o ***Time-outs*** donde podemos ver los temporizadores activos. Si ejecutamos el ejemplo del semáforo podemos verlo:

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

Puedes ver la información sobre la cantidad de ***time-outs*** definidos y la información de cada uno de ellos. En este caso solo tenemos un temporizador de estado.

13. Trazando la Máquina de Estados

Las trazas nos ayudan a entender qué está haciendo la máquina de estados cuando recibe un evento. La máquina de estados puede realizar acciones, cambiar de estado y recibir diferentes tipos de temporizadores. Las trazas nos ayudan a ver toda esta información. Por ejemplo, en el caso del semáforo, si activamos las trazas podemos ver:

```
> {ok, PID} = semaforo:start_link(30_000, 30_000).
{ok,<0.117.0>}
* cambia a ambar
* cambia a verde
> sys:trace(PID, true).
ok
*DBG* semaforo receive state_timeout {cambia,rojo} in state
 verde
* cambia a ambar
*DBG* semaforo consume state_timeout {cambia,rojo} in state
 verde => ambar
*DBG* semaforo start_timer {state_timeout,1000,{cambia,rojo},
[]} in state ambar
*DBG* semaforo receive state_timeout {cambia,rojo} in state
 ambar
* cambia a rojo
*DBG* semaforo consume state_timeout {cambia,rojo} in state
 ambar => rojo
*DBG* semaforo start_timer {state_timeout,30000,
{cambia,verde},[]} in state rojo
```

Podemos ver la información sobre el evento ***state_timeout*** recibido con la información `{cambia, rojo}` mientras está en verde y cómo realiza el cambio pasando antes por el ámbar.

Puedes ver más sobre las trazas en la Sección12, "Trazando el Servidor" .

14. Ejemplo del Ascensor

A lo largo del capítulo hemos visto la implementación del ascensor por partes. Ahora vamos a ver el código completo de esta máquina de estados y probaremos su ejecución. Como este código dispone de dos versiones la que presentaremos aquí será la segunda versión más completa.

El código completo es el siguiente:

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

-behaviour(gen_statem).

-export([
    start_link/0,
    stop/0,
    boton_num/1,
```

```
    boton_arriba/0,
    boton_abajo/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).

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

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

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

callback_mode() ->
    handle_event_function.

init([]) ->
    io:format("* iniciando ascensor, planta 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 recibida planta incorrecta: ~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("* cerrando puertas y subiendo~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("* cerrando puertas y bajando~n", []),
            {next_state, going_down, NewState, Actions};
        Current when Current =:= Num ->
            io:format("> planta actual, puertas abiertas~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("+ agregada planta ~p para parar (~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("> parando en planta ~p, abriendo
 puertas~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("< pasando por planta ~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, planta_baja, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 0}};
code_change(1, primera_planta, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 1}};
code_change(1, segunda_planta, _OldData, _Extra) ->
    {ok, stopped, #state{current_floor = 2}};
code_change({down, 1}, _, StateData, _Extra) ->
    NewState = case StateData#state.current_floor of
        0 -> planta_baja;
        1 -> primera_planta;
        _ -> segunda_planta
    end,
    {ok, NewState, {state}}.
```

Entramos en una consola y lo probamos de la siguiente forma:

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

> ascensor:start_link().
* iniciando ascensor, planta 0
{ok,<0.116.0>}

> ascensor:boton_num(2).
* cerrando puertas y subiendo
ok
< pasando por planta 1
> parando en planta 2, abriendo puertas

> ascensor:boton_abajo().
* cerrando puertas y bajando
ok
> parando en planta 1, abriendo puertas

> lists:foreach(fun ascensor:boton_num/1, [8,5,3,9,0]).
* cerrando puertas y subiendo
ok
```

```
+ agregada planta 5 para parar ([5,8])
+ agregada planta 3 para parar ([3,5,8])
+ agregada planta 9 para parar ([3,5,8,9])
+ agregada planta 0 para parar ([0,3,5,8,9])
< pasando por planta 2
> parando en planta 3, abriendo puertas
< pasando por planta 4
> parando en planta 5, abriendo puertas
< pasando por planta 6
< pasando por planta 7
> parando en planta 8, abriendo puertas
> parando en planta 9, abriendo puertas
< pasando por planta 8
< pasando por planta 7
< pasando por planta 6
< pasando por planta 5
< pasando por planta 4
< pasando por planta 3
< pasando por planta 2
< pasando por planta 1
> parando en planta 0, abriendo puertas

> ascensor:stop().
ok
```

El ascensor se inicia como proceso y podemos hacer que suba y baje usando las funciones correspondientes. Como último paso podemos detenerlo. La implementación de nuestro ascensor funciona correctamente.

15. Ejemplo del Semáforo

El código del semáforo como máquina de estados es tal y como sigue:

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

-behaviour(gen_statem).

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

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

-define(TIEMPO_EN_AMBAR, 1000). %% 1 segundo

-record(state, {
    tiempo_verde :: pos_integer(),
    tiempo_rojo :: pos_integer()
}).
```

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

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

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

callback_mode() ->
    handle_event_function.

init([TiempoVerde, TiempoRojo]) ->
    {ok, rojo, #state{
        tiempo_verde = TiempoVerde,
        tiempo_rojo = TiempoRojo
    }, [{state_timeout, TiempoRojo, {cambia, verde}}]}.

handle_event(state_timeout, Event, rojo, State) ->
    io:format("* cambia a ambar~n", []),
    {next_state, ambar, State,
     [{state_timeout, ?TIEMPO_EN_AMBAR, Event}]};

handle_event(state_timeout, {cambia, verde}, ambar, State) ->
    io:format("* cambia a verde~n", []),
    {next_state, verde, State,
     [{state_timeout, State#state.tiempo_verde, {cambia,
 rojo}}]};

handle_event(state_timeout, {cambia, rojo}, ambar, State) ->
    io:format("* cambia a rojo~n", []),
    {next_state, rojo, State,
     [{state_timeout, State#state.tiempo_rojo, {cambia,
 verde}}]};

handle_event(state_timeout, Event, verde, State) ->
    io:format("* cambia a ambar~n", []),
    {next_state, ambar, State,
     [{state_timeout, ?TIEMPO_EN_AMBAR, Event}]};

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

Entramos en una consola y probamos una ejecución:

```
12> c(semaforo).
{ok,semaforo}
13> semaforo:start_link(2000, 2000).
{ok,<0.103.0>}
* cambia a ambar
* cambia a verde
* cambia a ambar
* cambia a rojo
14> semaforo:ver_semaforo().
rojo
* cambia a ambar
* cambia a verde
* cambia a ambar
```

```
* cambia a rojo
15> semaforo:stop().
ok
```

Los cambios se suceden correctamente y obtenemos información cuando el semáforo cambia de color cada vez.

16. Ejemplo de Pago

En las secciones anteriores hemos visto cómo progresaba nuestro código para implementar el diagrama de estados de pago. Este ejemplo es un poco más largo que los anteriores porque dispone de más estados y una bifurcación en uno de los estados.

No obstante este sistema es lineal. No vuelve a estados anteriores y finaliza tras realizar todas las transiciones programadas. Esto lo convierte en el más fácil de seguir.

Su código completo puede verse a continuación:

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

-behaviour(gen_statem).

-export([
    start_link/0,
    stop/1,
    da_nombre/2,
    da_forma_pago/2,
    da_tarjeta/2,
    da_cuenta/2,
    obtiene_info/1
]).

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

-type forma_pago() :: tarjeta | domiciliar.

-record(state, {
        nombre :: string(),
        forma_pago :: forma_pago(),
        tarjeta :: string(),
        cuenta :: string()
}).

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

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

```
da_nombre(PID, Nombre) ->
    gen_statem:call(PID, {nombre, Nombre}).

da_forma_pago(PID, FormaPago) ->
    gen_statem:call(PID, {forma_pago, FormaPago}).

da_tarjeta(PID, Tarjeta) ->
    gen_statem:call(PID, {tarjeta, Tarjeta}).

da_cuenta(PID, Cuenta) ->
    gen_statem:call(PID, {cuenta, Cuenta}).

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

callback_mode() ->
    handle_event_function.

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

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

handle_event({call, From}, {nombre, Nombre}, credenciales,
 State) ->
    {next_state, forma_pago, State#state{nombre=Nombre},
 [{reply, From, ok}]};
handle_event({call, From}, _Event, credenciales, _State) ->
    {keep_state_and_data,
     [{reply, From, {error, "necesitamos nombre!"}}]};

handle_event({call, From}, {forma_pago, tarjeta}, forma_pago,
 State) ->
    {next_state, pago_tarjeta,
 State#state{forma_pago=tarjeta},
     [{reply, From, ok}]};
handle_event({call, From}, {forma_pago, domiciliar},
 forma_pago, State) ->
    {next_state, pago_cuenta,
 State#state{forma_pago=domiciliar},
     [{reply, From, ok}]};
handle_event({call, From}, _Event, forma_pago, _State) ->
    {keep_state_and_data,
     [{reply, From, {error, "forma de pago: tarjeta o
 domiciliar"}}]};

handle_event({call, From}, {tarjeta, Tarjeta}, pago_tarjeta,
 State) ->
    {next_state, pagado, State#state{tarjeta=Tarjeta},
     [{reply, From, {ok,pagado}}, hibernate]};
handle_event({call, From}, _Event, pago_tarjeta, _State) ->
    {keep_state_and_data,
     [{reply, From, {error, "necesitamos tarjeta"}}]};

handle_event({call, From}, {cuenta, Cuenta}, pago_cuenta,
 State) ->
    {next_state, pagado, State#state{cuenta=Cuenta},
     [{reply, From, {ok,pagado}}, hibernate]};
handle_event({call, From}, _Event, pago_cuenta, _State) ->
```

```
    {keep_state_and_data,
     [{reply, From, {error, "necesitamos cuenta"}}]};

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

terminate(normal, pagado, #state{forma_pago = tarjeta} =
 State) ->
    io:format("~p Nombre: ~s~nTarjeta: ~s~nPagado.~n",
              [self(), State#state.nombre,
 State#state.tarjeta]),
    ok;
terminate(normal, pagado, #state{forma_pago = domiciliar} =
 State) ->
    io:format("~p Nombre: ~s~nCuenta: ~s~nPagado.~n",
              [self(), State#state.nombre,
 State#state.cuenta]),
    ok;
terminate(normal, _StateName, _StateData) ->
    io:format("~p No pagado.~n", [self()]),
    ok.

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

Para ejecutarlo vamos a una consola y seguimos estos pasos:

```
> c(pago).
{ok,pago}
> {ok, PID} = pago:start_link().
{ok,<0.133.0>}
> pago:obtiene_info(PID).
{credenciales,{state,undefined,undefined,undefined,
                     undefined}}
> pago:da_nombre(PID, "Manuel").
ok
> pago:obtiene_info(PID).
{forma_pago,{state,"Manuel",undefined,undefined,undefined}}
> pago:da_forma_pago(PID, cuenta).
{error,"forma de pago: tarjeta o domiciliar"}
> pago:da_forma_pago(PID, domiciliar).
ok
> pago:obtiene_info(PID).
{pago_cuenta,{state,"Manuel",domiciliar,undefined,
                    undefined}}
> pago:da_tarjeta(PID, "1234").
{error,"necesitamos cuenta"}
> pago:da_cuenta(PID, "1234").
{ok,pagado}
> pago:obtiene_info(PID).
{pagado,{state,"Manuel",domiciliar,undefined,"1234"}}
> pago:stop(PID).
<0.133.0> Nombre: Manuel
Cuenta: 1234
Pagado.
ok
```

Podemos ver la progresión de las peticiones de pago hasta llegar a la parada del proceso y obtener toda la información. Podemos probar a

detener antes el proceso y ver el estado en el que queda la operación. Puedes realizar todas las pruebas y cambios que desees.