# Máquinas de Estado Finito

## Contenido

# Introducción

Los sistemas embebidos en tiempo real son sistemas reactivos. Su propósito principal es responder o reaccionar a las señales de su entorno.

El diseño de estos sistemas es un proceso complejo, que requiere la integración de métodos de diseño tanto en hardware como en software para cumplir con los requerimientos funcionales y no funcionales.

Las máquinas de estado finito (FSM - Finite State Machine) son una poderosa herramienta matemática y gráfica para especificar el comportamiento de sistemas reactivos.
A lo largo de las décadas, se han convertido en uno de los diseños más populares.

Las máquinas de estado finito son una herramienta muy útil para especificar aspectos relacionados con tiempo real, dominios reactivos o autónomos, computación reactiva, protocolos, circuitos, arquitecturas de software, etc.

El modelo de FSM es un modelo que posee sintaxis y semántica formales y que sirve para representar aspectos dinámicos que no se expresan en otros diagramas.

## Conceptos básicos de la máquina de estado finito

Un FSM es un modelo de cálculo abstracto que se utiliza para diseñar tanto software y circuitos lógicos secuenciales.

En general, una máquina de estado es un dispositivo que almacena el contexto de un objeto en un momento dado y opera con eventos de entrada para cambiar el contexto y/o provocar una acción o una salida dado el cambio.
El contexto se llama estado, que captura los aspectos relevantes de la historia del objeto.

Un FSM puede estar en uno de un número finito de estados. El cambio de un estado a otro se llama transición.
Se produce una transición de estado debido a la ocurrencia de algunos eventos o condiciones desencadenantes.

El estado en el que se encuentra un objeto en un momento dado se denomina estado actual.

Un FSM se puede representar mediante un gráfico dirigido llamado diagrama de estados, en donde cada estado está representado por un nodo (círculo) y cada transición está representada por una arista o línea.

## Máquinas de estado finito determinísticas y no determinísticas

Existen dos tipos de FSM: automatización finita determinista (DFA) y automatización finita no determinista (NDFA).

En DFA, se puede determinar el estado al que se moverá la máquina para cada evento desencadenante.

Un DFA también se conoce como un aceptador finito determinista, una máquina que acepta y rechaza la secuencia finita de entradas y solo produce un cálculo único (o ejecución) del autómata para cada secuencia de entrada.

Por ejemplo, el DFA que se muestra en la figura siguiente solo aceptará una contraseña de "pw123." Todas las demás cadenas de entrada llevarán a la máquina al error y al estado muerto q6.

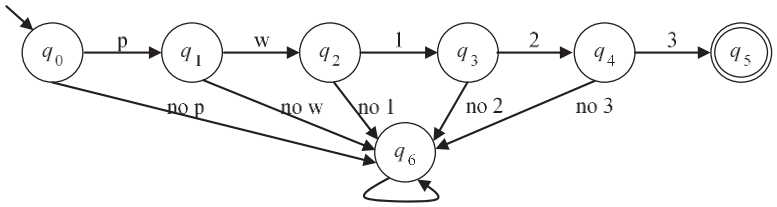
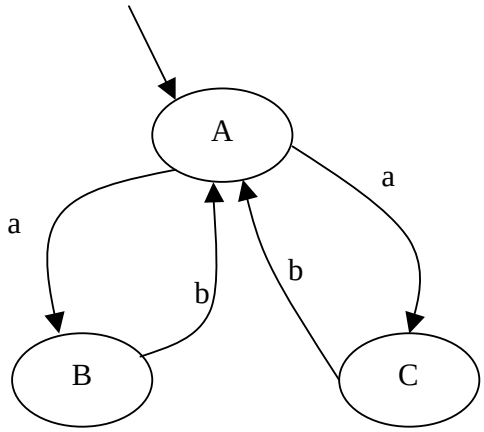



**Figura:** Un DFA para la contraseña pw123.

## No Determinismo (NDFA)

Si dado un estado y dado un evento, existe más de una transición válida (cuya condición, si existe, se satisface), hacia distintos estados, se asumirá un comportamiento no determinístico: es decir, cualquiera de los estados posibles podrá ser alcanzado, sin orden alguno.

Ejemplo



En un NDFA, dado un estado y una entrada, puede haber más de un estado siguiente, o un estado puede pasar de un estado a otro sin ninguna entrada, o no hay ningún estado siguiente en absoluto para alguna entrada dada.

Los NDFA son buenos para especificar un comportamiento desconocido o no especificado del sistema. Para cualquier NDFA, siempre hay un DFA equivalente. Sin embargo, el modelo NDFA es más compacto y utiliza menos estados en comparación con el modelo DFA.

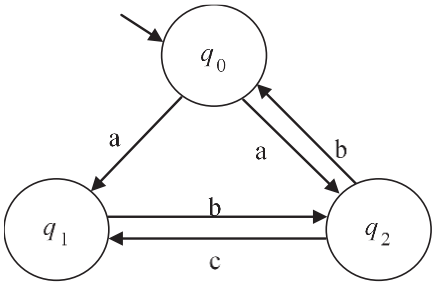



**Figura:** Un simple NDFA.

La figura muestra un NDFA, en el cual la entrada a en el estado q0 puede cambiar el sistema al estado q1 o q2. En otras palabras, el comportamiento del sistema en respuesta al evento de entrada a en el estado inicial no es determinista.

Las transiciones de estado también se pueden describir mediante una tabla de transición de estado como ejemplificamos a continuación:

**Cuadro** Transiciones de estado del FSM

| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|
| $q_0$ | $q_0$ | $q_0$ | $q_1$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ |
| $q_1$ | $q_2$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ |
| $q_2$ | $q_0$ | $q_3$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ |
| $q_3$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_0$ | $q_4$ | $q_0$ | $q_0$ |

Una máquina de estado puede tener salidas correspondientes a cada transición.

Hay dos tipos de FSM que generan salidas:

1. Máquinas Moore
2. Máquinas de Mealy

Una máquina Moore es un DFA cuya salida depende sólo del estado actual. Por el contrario, una máquina Mealy es un DFA cuya salida depende de la entrada actual, así como del estado actual. Cualquier máquina Moore se puede convertir en una máquina Mealy y viceversa.

# Modelo Matemático de un FSM

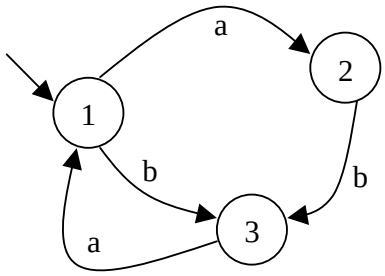
Las máquinas de estado finito se definen como una tupla

$(S, \Sigma, A \subseteq S \times \Sigma \times S, sk)$

donde:

- $S=\{S_1, S_2, \ldots, S_m\}$: es un conjunto finito de nodos.
- $\Sigma$: es un alfabeto finito de etiquetas.
- A: es un conjunto finito de aristas etiquetadas que unen nodos.
- $sk \in S$: es el estado inicial.

**Ejemplo**




- ( S = {1,2,3},
- Σ = {a,b},
- A = {(1, a, 2), (2, b,3), (3, a,1), (1, b, 3)},
- sk =1 )

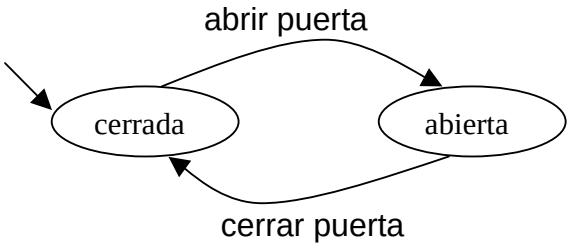
Vale la pena destacar que formalmente la máquina de estado es la tupla anterior y no el “gráfico”. Este es tan sólo una representación de la máquina de estado para tener una más sencilla y rápida visualización de su contenido.

## Semántica

Los nodos representan los posibles estados de aquello que se desea modelar. Las etiquetas representan eventos que provocan un cambio. Las aristas determinan de qué manera cada estado, dado un evento, deriva en otro estado.

Ejemplo

Supongamos que se quiere modelar el comportamiento de una puerta. La puerta, inicialmente cerrada, puede pasar a estar abierta tras el evento “abrir puerta”. Una vez abierta, puede pasar al estado cerrada, tras el evento “cerrar puerta”.

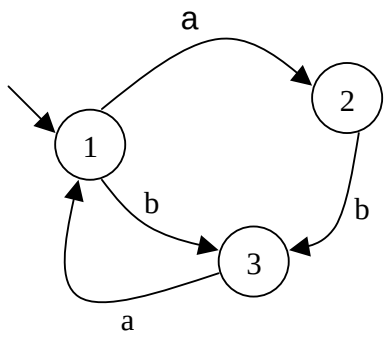


# Trazas

El conjunto de posibles trazas correspondientes a una máquina de estado finitos se puede definir en término de grafos, cómo el conjunto de todos los caminos (de ejes) alcanzables desde el estado inicial.

## Ejemplo

Dada la FSM de ejemplo:




Las trazas de esta FSM son:

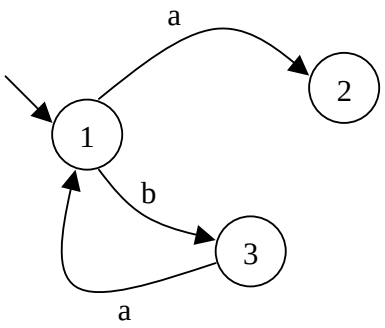
{a, b, a} correspondiente a 1, 2, 3, 1
{b, a} correspondiente a 1, 3, 1
{a, b, a, b, a} correspondiente a 1, 2, 3, 1, 3, 1
{b, a, a, b, a} correspondiente a 1, 3, 1, 2, 3, 1
{b, a, b, a, ..., b, a} 1, 3, 1, 3, …, 1, 3
Etc…

# Deadlock

Formalmente hablando, una FSM $\langle S, \Sigma, A \subseteq S \times \Sigma \times S, sk \rangle$, tiene deadlock, si existe algún nodo $s \in S$, tal que no existen un evento e y un nodo $t \in S$ tal que $(s,e,t) \in A$. En otras palabras, si existe algún nodo que no posea “salida” para ningún evento.

Ejemplo

El estado 2 no posee salida alguna.

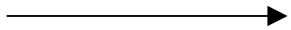


# Extensiones

Para aumentar el poder expresivo, existen extensiones al lenguaje de FSM visto anteriormente.

## Condiciones

Son expresiones booleanas que determinan si una transición entre estados puede realizarse. Se escriben entre corchetes.



[expresión booleana]

## Acciones

Inmediatamente después de ejecutar una transición y antes de arribar al siguiente estado, la máquina de estados ejecuta una acción. Básicamente, para nosotros, las acciones serán asignaciones de valores a variables. En algunos contextos, dentro de las acciones, también se lanzan eventos (para luego sincronizar otras máquinas). Se escriben entre llaves.

{acción; acción}

## Variables

Sus valores posibles deben pertenecer a un dominio finito. Pueden utilizarse estructuras como arreglos o tipos abstractos de datos en general (cómo por ejemplo conjuntos o pilas).

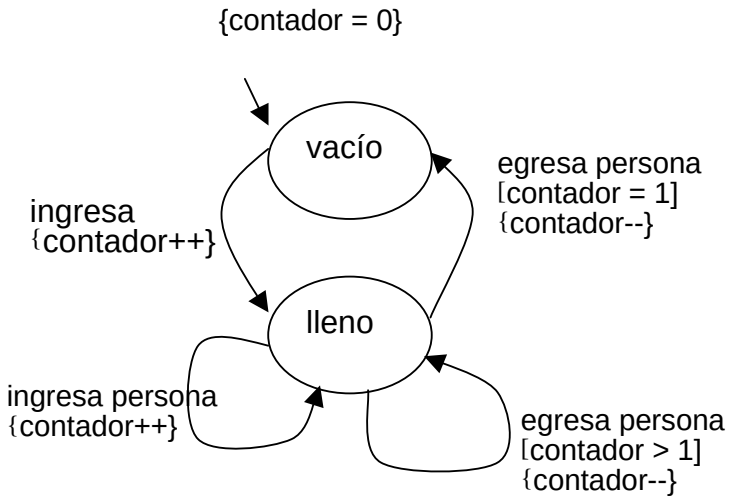
Los variables pueden usarse dentro de las expresiones booleanas de las condiciones y su valor puede cambiarse utilizando acciones, generalmente asignaciones. A su vez, salvo explicitado lo contrario, las variables son compartidas por todas las FSMs de la composición (es decir pueden verse como globales a todas las máquinas de estado).

**Ejemplo**

Un local de ropa puede estar vacío o lleno según la cantidad de personas que hay en su interior. Inicialmente está vacío, pero luego de ingresar la primera persona está lleno. A partir de ahí, la gente puede ingresar o salir. Cuando sale el último, vuelve a estar vacío.

Todas las variables se deben definir de antemano:

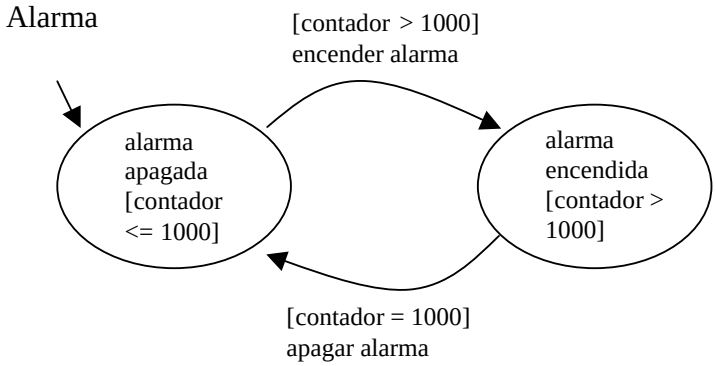
contador: [0, 99999999] (por simplicidad, permitiremos decir N. Sin embargo, debemos resaltar que esto no es del todo correcto: el dominio debe tener un número finito de valores).




## Condiciones de salida de un estado

Es posible colocar condiciones booleanas dentro de los estados (utilizando [expresión booleana] dentro de los nodos) para obligar a que se salga del mismo. Mientras la condición agregada a un estado sea verdadera es posible mantenerse en dicho estado o salir del mismo a través de una transición cualquiera. Sin embargo, apenas dicha condición se haga falsa la maquina estará obligada a dejar el estado por alguna de las transiciones de salida. Si hay muchas posibles se saldrá por cualquiera de estas de forma no determinística. Si en cambio no hay ninguna transición posible, se caerá indefectiblemente en un deadlock.

Supongamos que al ejemplo anterior del local de ropa se le agrega la condición de que si hay más de 1000 personas dentro, debe sonar una alarma. Una forma de representar dicha situación es componiendo la siguiente FSM a la ya hecha:




LocalSeguro = Local || Alarma

## FSMs Temporizadas

Muchas veces los disparadores de los eventos que provocan los cambios de estado de las máquinas son iniciados por el paso del tiempo. Por ej. el cambio de luz de los semáforos, un rociador automático de césped que se enciende y riega cada 2 horas por 10 minutos, o un sistema de dosificación de químicos en una cañería que cada determinada cantidad de tiempo inyecta distintas sustancias en un orden especial. En todos estos casos la decisión de que ocurran los eventos es disparada por el tiempo. Para poder representar este funcionamiento utilizando FSMs se hace necesario extenderlas agregando un componente extra denominado “timer”.

Este será declarado junto con el resto de las variables de la siguiente manera:

nombreTimer: timer. En -unidad de tiempo-. (por ej. segundos).

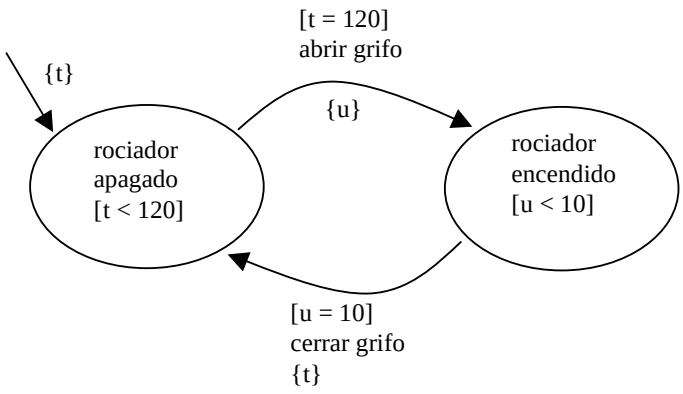
Las únicas operaciones permitidas por el timer son “resetearlo” (es decir colocar su cuenta en 0) y chequear su valor (es decir, ver el tiempo transcurrido desde su último “reset”). Una vez que el timer es “reseteado” empieza a contar el tiempo transcurrido hacia adelante, y eso es todo lo que sabe hacer (no se lo puede setear un valor determinado). Podremos chequear el tiempo transcurrido a través del uso de condiciones y podremos resetearlo utilizando acciones.
La sintaxis para resetear el timer es colocando al nombre del mismo dentro de llaves.

Plasmemos el ejemplo del rociador automático en una FSM para ver cómo se puede utilizar esta extensión para representar dicha situación:

t: timer. En minutos.

u: timer. En minutos.

Rociador



# Ejercicio Resuelto

Cierta estancia cuenta con tres parcelas en las cuales realiza sus cultivos. Con el fin de optimizar su productividad, se ha definido una política con respecto a cómo trabajar la tierra.

La primera etapa consiste en preparar la tierra en todas las parcelas. Para ello se utiliza un tractor. La estancia posee un único tractor, el cual puede ser usado de a una parcela a la vez (y luego liberado). No hay un orden predeterminado para estas actividades, es más, todas las parcelas “se pelean” entre sí para el uso del tractor. De esta manera, mientras que en una parcela se está realizando el preparado de la tierra, las demás parcelas deben esperar a que termine.

Recién cuando la tierra de todas las parcelas ha sido preparada se da inicio a la segunda etapa.

Durante la segunda etapa se siembran simultáneamente todas las parcelas (no se utiliza el tractor para tal tarea). La manera de hacer el sembrado no es relevante (por lo general se utilizan aviones especialmente equipados).

La tercera etapa es comprendida por la cosecha de cada parcela, nuevamente utilizando el único tractor (con las mismas condiciones en cuanto a su uso con respecto al preparado de la tierra). Luego de realizar la cosecha de todas las parcelas, se está en condiciones de dar inició, nuevamente, a la primera etapa.

Modele la política implementada por la estancia, utilizando FSM.
¿Cómo extendería el modelo para contemplar n parcelas?

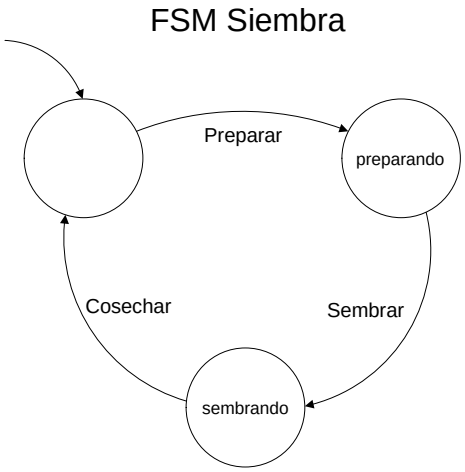
## Respuesta sugerida

Vamos a resolver el problema para n parcelas directamente. En este caso N sería 3.

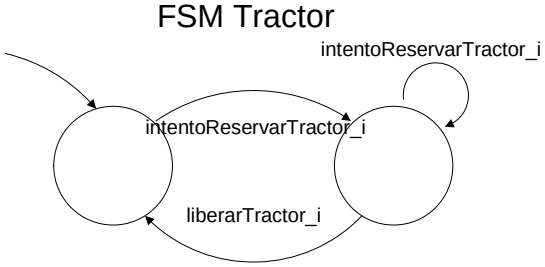
La respuesta es la composición paralela de las máquinas que describimos a continuación, esto se simboliza:

Siembra || Tractor || Parcela1 || .. || ParcelaN

Modelamos con una máquina la secuencia de operaciones que se realizarán en las parcelas:



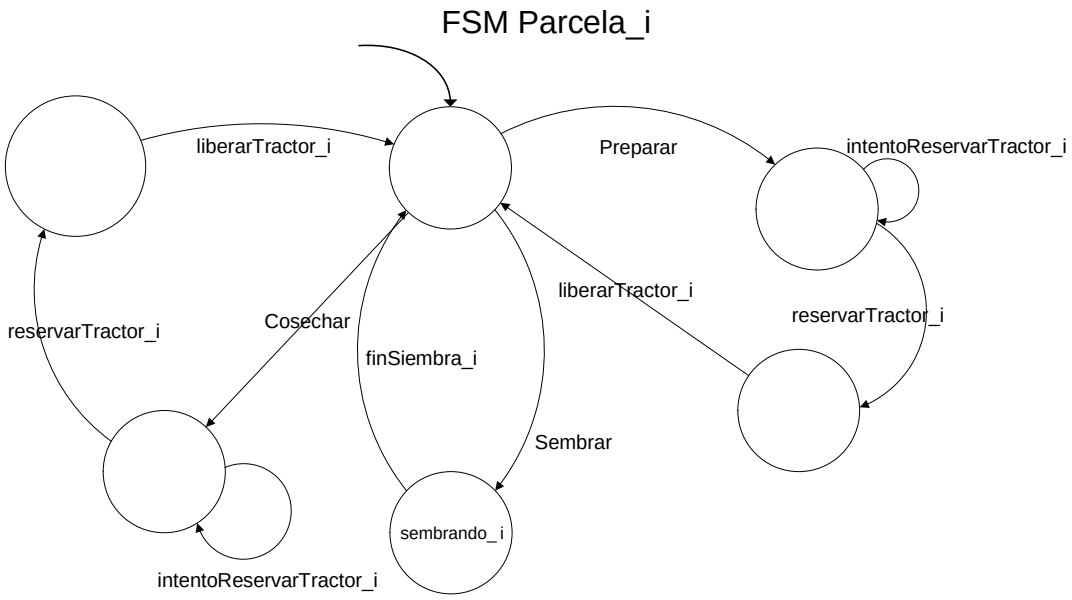

Luego modelamos el uso del tractor con otra máquina:




NOTA: Por cada evento hay n transiciones, una por cada parcela.

La inclusión de la transición intentoReservarTractor_i se hace para modelar que en realidad al no poder obtener el tractor cada parcela sigue intentando obtenerlo.

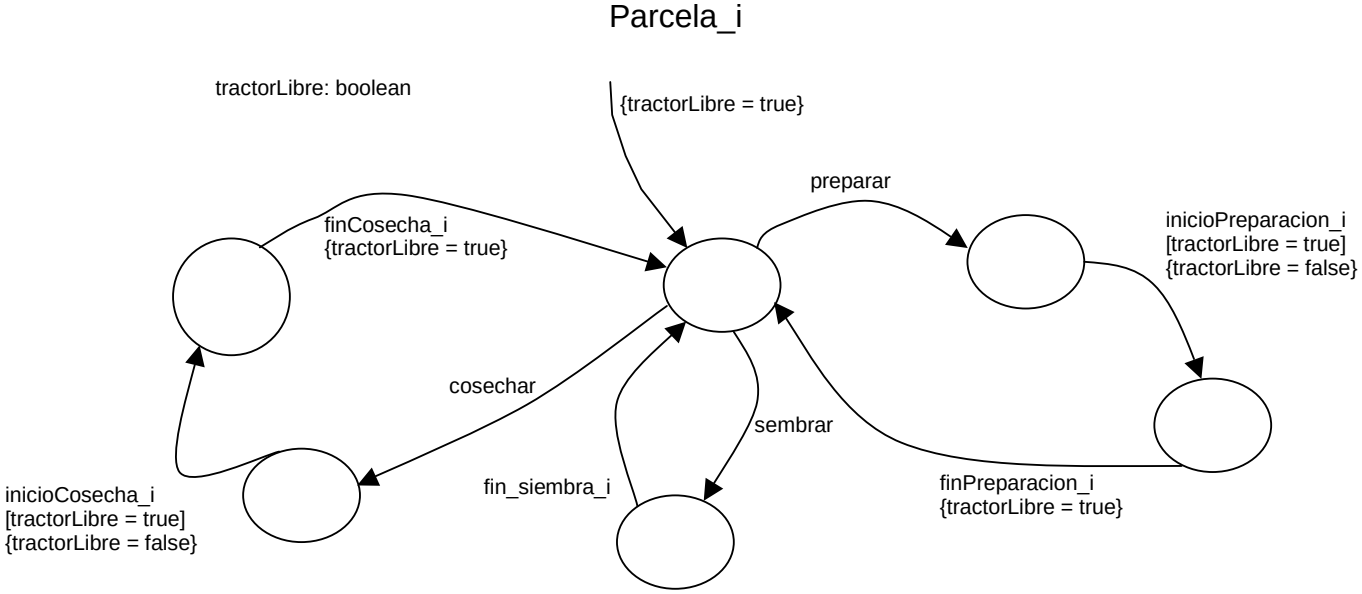
Finalmente tenemos la máquina que modela cada parcela:




## Otra solución podría ser la siguiente

Siembra || Parcela_1 || …. || Parcela_N

En este caso no modelamos al uso del tractor con una máquina, sino que utilizamos una variable que nos dice si ya está reservado o no.



# Programación de máquinas de estado finito

Como uno de los patrones de diseño más populares, los FSM se implementan en muchas aplicaciones integradas en tiempo real. Hay dos enfoques comunes para implementar un FSM. Uno es usando instrucciones condicionales. En este problema simple, cada estructura interna del caso del switch se puede reemplazar con una sentencia if–then.

```
int get_input();

void fsm( )
{
  enum states {STATE0, STATE1, STATE2, STATE3, STATE4} current_state;
  lock_safe();
  while(true)
  {
    input = get_input();
    switch(current_state)
    {
      case STATE0:
        switch(input)
        {
          case 2:
            current_state = STATE1;
          break;
          default:
            current_state = STATE0;
        }
      case STATE1:
        switch(input)
        {
          case 0:
            current_state = STATE2;
            break;
          default:
            current_state = STATE0;
        }
      case STATE2:
        switch(input)
        {
          case 1:
            current_state = STATE3;
            break;
          default:
            current_state = STATE0;
        }
      case STATE3:
        switch(input)
        {
          case 7:
            current_state = STATE4;
          break;
          default:
          current_state = STATE0;
        }
    }
  }
}
```

El enfoque de instrucciones condicionales es muy sencillo y fácil de entender. Sin embargo, cuando el número de estados y eventos de entrada crecen, el código puede volverse fácilmente difícil de manejar. Cuando el código de máquina de estado se ejecuta en varias páginas de pantalla, la depuración y el mantenimiento se volverán difíciles, por no mencionar la legibilidad del código.

Otro enfoque para implementar máquinas de estado se basa en tablas. Para que la implementación sea escalable para máquinas con un gran número de estados y eventos de entrada, este enfoque utiliza una tabla bidimensional, con una dimensión para los estados y la otra para los eventos, para almacenar funciones de transición. La tabla se puede implementar en C mediante una matriz bidimensional de punteros de función.

```
enum states {OFF, SEATED, READY, BELTED, TIMING, BUZZER} current_state;
enum events {SEAT, UNSEAT, BELT, UNBELT, KEY, TIMER_EXPIRES} new_event;

#define MAX_STATES 6
#define MAX_EVENTS 6

typedef void (*transition)();

transition state_table[MAX_STATES][MAX_EVENTS] =
{
  {seat, error  , error  , error    ,      error,        error}  , // state OFF
  {error, unseat, belt_s , error    ,    key_s,       error}  , // state SEATED
  {error, error , error  , unbelt_r , key_r,          error}  , // state READY
  {error, error , error  , unbelt_b , key_b,          error}  , // state BELTED
  {error, error , belt_t , error    ,      key_t,       timer}  , // state TIMING
  {error, error , belt_b , error    ,      key_z,       error}    // state BUZZER
};



/* Function belt_t
*
*  Input event    : BELT Output: timer_off
*  Current state  : TIMING Next state: BELTED
*
*/
void belt_t()
{
  turn_timer_off();

  current_state = BELTED; //variable global
}


while(true)
{
  new_event = get_new_event();

  if( (new_event >= 0) && (new_event < MAX_EVENTS) && (current_state >= 0) && (current_state < MAX_STATES) )
  {
    state_table[current_state][new_event]();
  }
  else
  {
    /* TODO: logueamos el error ... */
  }
}
```