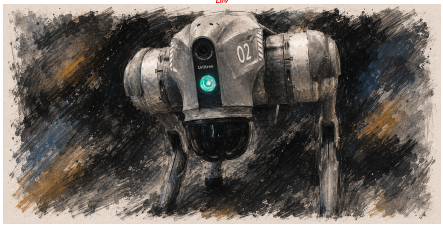
Máquinas de estados finitos

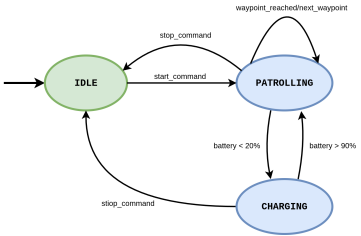
Francisco Martín Rico y Rodrigo Pérez Rodríguez

Intelligent Robotics Lab



La máquina de estados como arquitectura de decisión

- Las FSM (*Finite State Machines*) modelan el comportamiento mediante estados explícitos y transiciones definidas.
- A diferencia de enfoques puramente reactivos, las FSM representan secuencias de comportamiento y modos de operación persistentes.
- Cada estado encapsula una lógica específica y las transiciones definen cuándo y cómo cambia el comportamiento.
- La arquitectura es declarativa: el diagrama comunica el comportamiento completo del sistema sin necesidad de código.



La máquina de estados como arquitectura de decisión

Elementos fundamentales

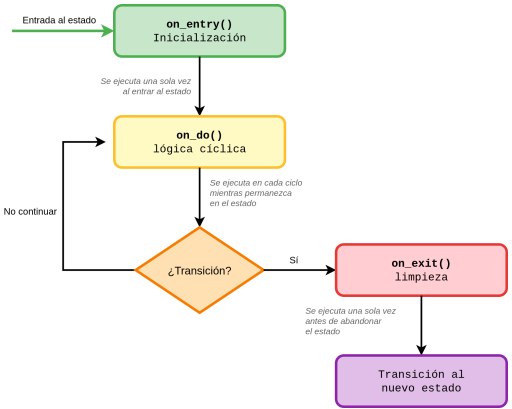
- **Estados bien definidos:** Representan modos de operación del sistema (ej. IDLE, PATROLLING, CHARGING).
- **Transiciones explícitas:** Flechas que conectan estados, muestran los cambios posibles.
- **Eventos disparadores:** Etiquetas en las transiciones, indican qué provoca el cambio (ej. start_command).
- **Guardas condicionales:** Expresiones booleanas que condicionan la transición (ej. battery $< 20\,\%$).
- La FSM permite coordinar acciones a lo largo del tiempo y reaccionar de forma controlada ante eventos del entorno.

Anatomía de un estado en robótica

- En robótica, un estado controla recursos físicos (motores, sensores) que requieren gestión segura.
- Cada estado se estructura en tres fases diferenciadas:
  - **on_entry:** Ejecutada una vez al entrar; inicializa temporizadores, sensores, parámetros.
  - **on_do / on_run:** Ejecutada cíclicamente; implementa la lógica continua del estado.
  - **on_exit:** Ejecutada una vez al salir; detiene motores, libera recursos, garantiza seguridad.
- Esta estructura garantiza que el robot deje recursos en estado conocido y seguro, independientemente de cómo se abandone el estado.

Anatomía de un estado en robótica

Ciclo de vida de un estado



Anatomía de un estado en robótica

Secuencia de ejecución

1. **on_entry:** Se ejecuta automáticamente al entrar, inicializando recursos y garantizando configuración conocida.
2. **on_do:** Se ejecuta repetidamente en bucle cíclico, evaluando condiciones, calculando control y verificando transiciones.
3. **Decisión de transición:** En cada ciclo se evalúa si se cumplen condiciones para cambiar de estado.
4. **on_exit:** Se ejecuta antes de transitar, deteniendo motores, apagando sensores y liberando recursos de forma segura.
5. **Transición:** Solo después de completar on_exit se procede al nuevo estado.

Esta secuencia estricta (entrada → ejecución cíclica → salida → transición) evita condiciones de carrera y estados inconsistentes en los actuadores del robot.

Semántica de una transición

- Una transición se define formalmente mediante la tupla:

$$\text{Transición} = \text{Evento} + [\text{Guarda}]/\text{Acción}$$

- **Evento:** Suceso (externo o interno) que despierta la posibilidad de cambio.
  - En ROS 2: llegada de mensaje, timeout, resultado de Acción.
- **Guarda:** Expresión lógica que debe ser verdadera para que ocurra la transición dado el evento.
  - Ejemplo: recibir mensaje láser (evento) + distancia $<$ umbral (guarda).
- **Acción:** Operación ejecutada al ocurrir la transición (opcional, depende de Moore/Mealy).

Distinguir evento y guarda es crucial para depuración: el robot puede no cambiar porque el evento no llega (fallo comunicación) o porque la guarda no se cumple (fallo lógica).

Topologías: Moore vs Mealy

- La diferencia tiene impacto directo en la **latencia** (tiempo de reacción).
- La clave está en determinar *en qué ciclo de control* se envía el comando a los motores.

Máquinas de Moore

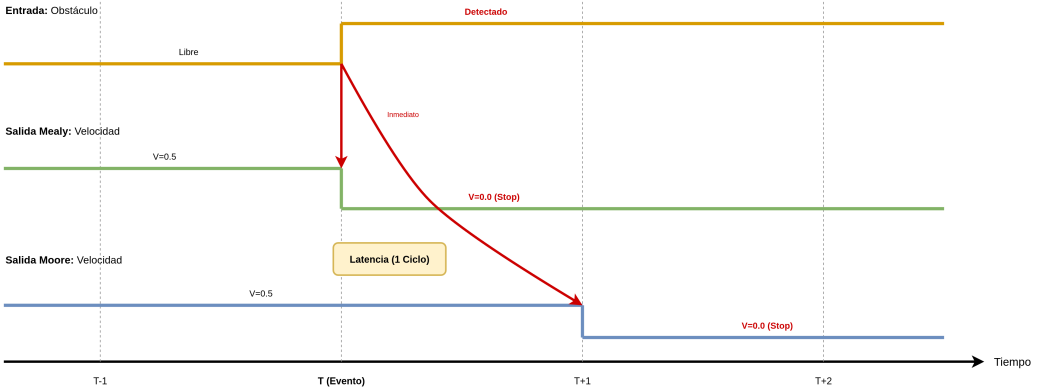
- Las acciones de control se ejecutan *dentro del código del estado*.
- **Latencia:** 1 ciclo completo.
- Más ordenado y seguro.
- Código más modular.

Máquinas de Mealy

- Las acciones se ejecutan *en la transición*, antes de cambiar de estado.
- **Latencia:** Reacción instantánea (mismo ciclo).
- Más rápido en respuesta.
- Preferible para acciones críticas.

Topologías: Moore vs Mealy

Ejemplo: frenada de emergencia (ciclo 100ms)



Topologías: Moore vs Mealy

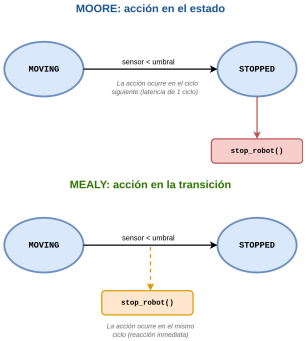
Análisis temporal

- En el ciclo T, el sensor detecta obstáculo (señal naranja cambia de «Libre» a «Detectado»).
- **Salida Mealy:** La velocidad cae a cero *inmediatamente* en el ciclo T. La acción forma parte de la transición.
- **Salida Moore:** La velocidad se mantiene durante todo el ciclo T, y solo en $T + 1$ (100ms después) envía el comando de parada. La acción está dentro del estado destino.
- Consecuencia: A 0.5 m/s, 100ms = 5 cm adicionales de desplazamiento antes de detenerse.

En aplicaciones críticas (frenado de emergencia, detección de bordes), Mealy es preferible. Para tareas donde el orden y claridad del código son prioritarios, Moore ofrece mejor diseño.

Topologías: Moore vs Mealy

Diferencia arquitectónica



Moore: acción `stop_robot()` en el estado destino STOPPED (ciclo siguiente).
Mealy: acción `stop_robot()` en la transición misma (ciclo actual).

El problema de la concurrencia

- Las FSM clásicas son **secuenciales**: el sistema solo puede estar en un único estado en un instante dado.
- Problema: el robot debe realizar múltiples tareas independientes simultáneamente.
  - Ejemplo: navegar hacia un punto **Y** monitorizar nivel de batería.
- Con FSM clásica: explosión combinatoria de estados (*producto de estados*).
- Navegación (3 estados) + Batería (2 estados) $= 3 \times 2 = 6$ estados combinados.
- Si añadimos luces (2 estados): $6 \times 2 = 12$ estados. Con 4 dimensiones: 24 estados.

Consecuencia: Cada nueva funcionalidad independiente *multiplica* el número de estados en lugar de sumarlos, violando modularidad.

El problema de la concurrencia

La pesadilla combinatoria

```
1  // Si queremos navegar (3 estados) y vigilar la bateria
2  // (2 estados) necesitamos 3x2 = 6 estados explicitos
3  enum State {
4      IDLE_BAT_OK,
5      IDLE_BAT_LOW,
6      MOVING_BAT_OK,
7      MOVING_BAT_LOW,
8      RECOVERING_BAT_OK,
9      RECOVERING_BAT_LOW
10 };
11 // Cada vez que anadimos una funcionalidad,
12 // los estados se multiplican.
```

Enfoque completamente insostenible para sistemas complejos: imposible de mantener y depurar.

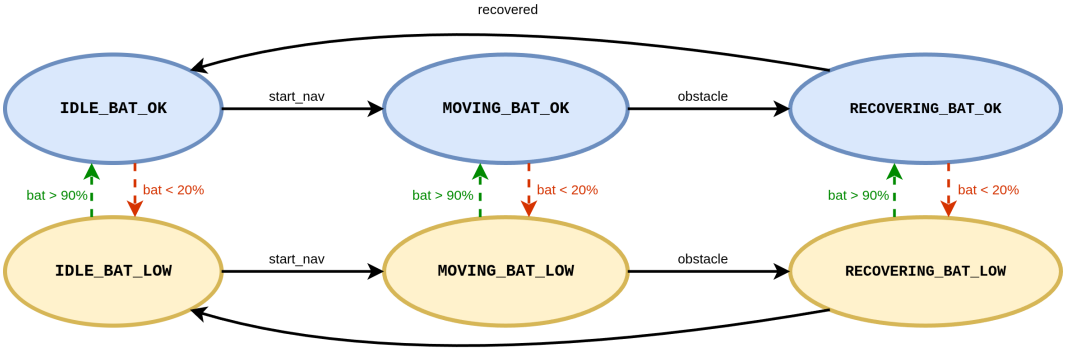
El problema de la concurrencia

Solución: Statecharts y regiones ortogonales

- David Harel (1987) propuso **statecharts**: extensión de FSM con **ortogonalidad** (estados AND).
- El robot está *simultáneamente* en un estado de región A **Y** en un estado de región B.
- En código: descomponer el sistema en múltiples FSM más pequeñas que se ejecutan secuencialmente en el mismo ciclo.
- Ventaja matemática: complejidad crece linealmente, no exponencialmente.
- Navegación (3) + Batería (2) $= 3 + 2 = 5$ estructuras, no $3 \times 2 = 6$ estados combinados.

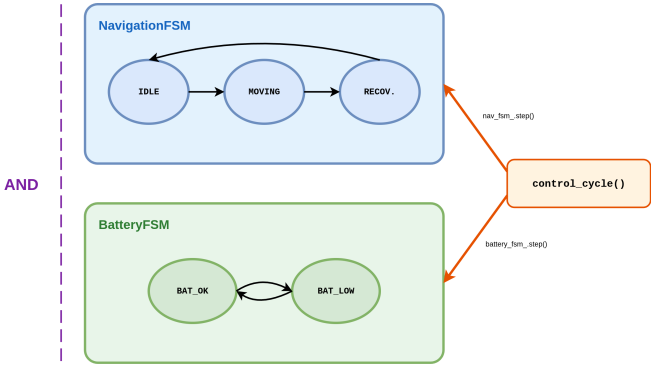
El problema de la concurrencia

Enfoque tradicional: explosión combinatoria



El problema de la concurrencia

Descomposición ortogonal



El robot está simultáneamente en un estado de navegación Y en un estado de batería

El problema de la concurrencia

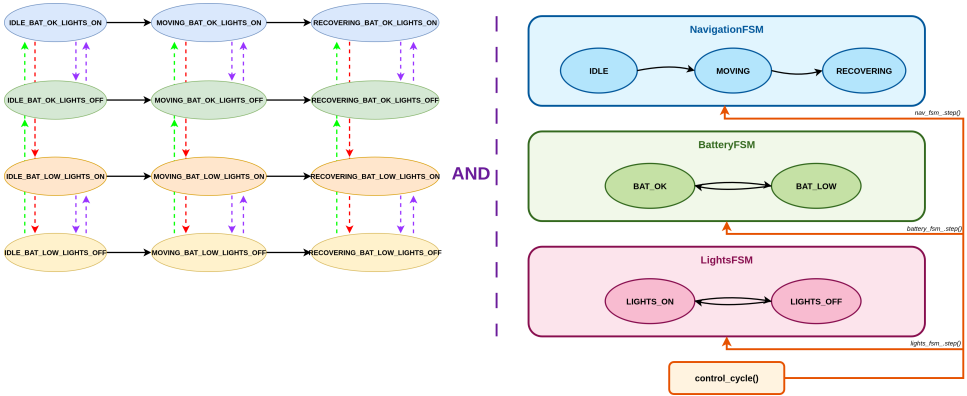
Comparación: tradicional vs ortogonal

- **Enfoque tradicional:** Producto cartesiano de $3 \times 2 = 6$ estados combinados (explosión combinatoria).
- **Descomposición ortogonal:** Dos FSM independientes con $3 + 2 = 5$ estructuras de estado.
- `NavigationFSM` gestiona navegación (3 estados) y `BatteryFSM` monitoriza batería (2 estados).
- En cada iteración del bucle de control (`control_cycle()`), se invoca el método `step()` de ambas FSM.
- Aunque las llamadas son secuenciales en código, conceptualmente ambas están activas *simultáneamente*.
- El robot está, en todo momento, en un estado de navegación **Y** en un estado de batería.

Si añadimos el sistema de luces (2 estados): enfoque tradicional = 12 estados combinados vs ortogonal = 7 componentes.

El problema de la concurrencia

Escalando a 3 dimensiones: añadiendo control de luces



Izquierda: Enfoque tradicional con $3 \times 2 \times 2 = 12$ estados combinados. **Derecha:** Descomposición ortogonal con $3 + 2 + 2 = 7$ FSM independientes.

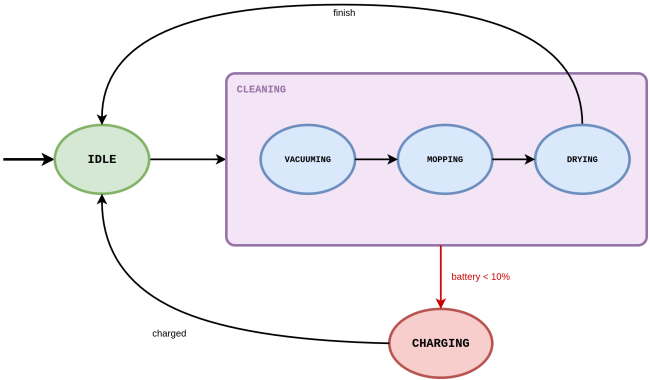
Máquinas de estados jerárquicas

- Harel también formalizó la **jerarquía** (estados OR): un estado puede contener otra FSM completa.
- Tres ventajas arquitectónicas:
  1. **Abstracción:** Un superestado visto desde fuera es una caja negra.
  2. **Reutilización:** Una sub-máquina bien diseñada puede encapsularse y reutilizarse.
  3. **Factorización de transiciones:** Transiciones desde el superestado aplican a todos sus subestados, simplificando el diagrama.

Ejemplo: Robot de limpieza con superestado CLEANING que contiene VACUUMING, MOPPING, DRYING. La transición de emergencia (batería baja) se define una vez desde el superestado.

Máquinas de estados jerárquicas

Ejemplo: Robot de limpieza



Máquinas de estados jerárquicas

Factorización de transiciones

- La flecha roja gruesa sale del rectángulo morado (superestado CLEANING), no de cada círculo individual.
- Significa: *independientemente de si el robot está aspirando, fregando o secando*, si batería $<$ 10 %, transita a CHARGING.
- Si el superestado tuviera 10 subestados en lugar de 3, seguiríamos necesitando una única transición de emergencia.
- Esta factorización es especialmente valiosa en sistemas complejos y facilita el mantenimiento.

La jerarquía permite escalar el diseño sin aumentar proporcionalmente la complejidad del diagrama.

Ventajas y problemas de escalabilidad

Ventajas

- Simplicidad conceptual
- Facilidad de implementación
- Transparencia del comportamiento
- Diseño declarativo (diagrama = documentación)
- Adecuadas para número limitado de estados

Problemas

- A medida que el número de estados crece, difíciles de mantener
- Explosión de estados y transiciones (*state explosion*)
- Diseños complejos y poco legibles
- Difíciles de extender sin reestructuración

Las extensiones (ortogonalidad y jerarquía) mitigan estos problemas, permitiendo diseños más escalables y mantenibles.

Modelado de misiones y tareas

- En arquitecturas deliberativas modernas, la FSM puede ocupar la capa de **misión**: modela la macrosecuencia del sistema, define las fases principales y los criterios de transición.
- Cada estado de la FSM de misión representa una *fase* o *modo* operativo relevante (ej. `PLANNING`, `EXECUTING`, `REPORTING`).
- Al entrar en un nuevo estado de misión, se activa la tarea correspondiente (ej. `INSPECT_AREA`, `TRANSPORT_OBJECT`).
- Las tareas son unidades ejecutables concretas, que pueden implementarse mediante otros modelos de control: Behavior Trees (BT), sub-FSM, etc.
- En sistemas sencillos, una tarea puede ser una FSM; en arquitecturas escalables, es preferible reservar la FSM para la misión y emplear otros mecanismos para las tareas.

La FSM de misión orquesta el flujo global y la persistencia de estado; las tareas orquestan capacidades y gestionan la ejecución concreta.

Capacidades: el 'cómo' ejecutable

- Las **capacidades** son servicios gobernables y reutilizables que implementan el "cómo" de cada acción elemental (navegar, manipular, percibir, etc.).
- Tienen contratos claros de precondiciones, feedback, resultado y cancelación.
- Las tareas consumen capacidades y consolidan su feedback, reportando progreso y causas de fallo hacia la misión.
- Así, la misión no necesita espiar telemetría de bajo nivel, sino que recibe información semántica relevante para decidir si replanificar, cambiar de objetivo o modificar la política.

Separar misión, tarea y capacidad mantiene un diseño modular, escalable y trazable.

Diseño compositivo e híbrido

- Este diseño habilita arquitecturas híbridas: la FSM de misión gestiona el flujo global, mientras que las tareas orquestan capacidades y gestionan la ejecución concreta.
- Las tareas pueden emplear BT, sub-FSM (FSM anidada dentro de una tarea) u otros modelos según convenga.
- Ejemplo: la fase `EXPLORE_AREAS` de la misión puede activar una tarea implementada como BT que coordina navegación, inspección visual y monitorización de batería de forma reactiva.
- La FSM de misión permanece en ese estado hasta que la tarea reporte completitud o fallo.

La contribución arquitectónica fundamental de la FSM es definir el *cuándo* y el *en qué fase* se encuentra el sistema; el *cómo* reside en tareas y capacidades.

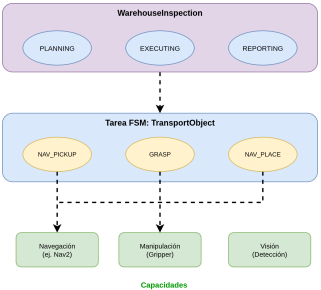
Diseño compositivo e híbrido

- Este diseño habilita arquitecturas híbridas: la FSM de misión gestiona el flujo global, mientras que las tareas orquestan capacidades y gestionan la ejecución concreta.
- Las tareas pueden emplear BT, sub-FSM (FSM anidada dentro de una tarea) u otros modelos según convenga.
- Ejemplo: la fase `EXPLORE_AREAS` de la misión puede activar una tarea implementada como BT que coordina navegación, inspección visual y monitorización de batería de forma reactiva.
- La FSM de misión permanece en ese estado hasta que la tarea reporte completitud o fallo.

La contribución arquitectónica fundamental de la FSM es definir el *cuándo* y el *en qué fase* se encuentra el sistema; el *cómo* reside en tareas y capacidades.

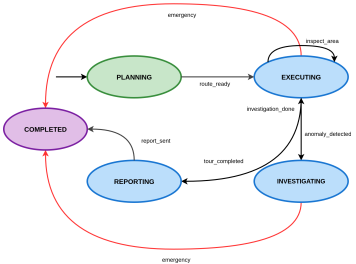
Ejemplo visual: composición misión–tarea–capacidad

- Una misión contiene una FSM de alto nivel cuyos estados activan tareas.
- Cada tarea tiene su propia lógica (BT, FSM, etc.) y puede invocar capacidades cuando sea necesario.
- Las capacidades son módulos reutilizables y desacoplados.



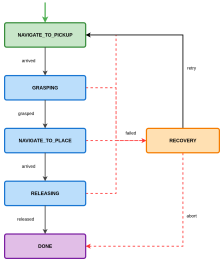
FSM de misión: ejemplo `WarehouseInspection`

- Una FSM de misión coordina tareas de larga duración para alcanzar objetivos globales (ej. inspección de almacén).
- Cada estado representa una fase: planificación, ejecución, investigación de anomalías, generación de informe.
- Las transiciones pueden ser nominales (flujo normal) o especiales (anomalías, emergencias).
- Cada estado puede invocar tareas complejas, que a su vez tienen su propia FSM o BT.



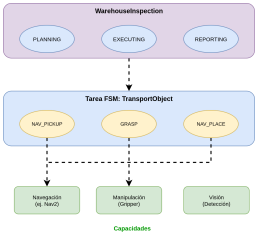
FSM de tarea: ejemplo `TransportObject`

- Una FSM de tarea coordina la secuencia de acciones para lograr un objetivo concreto, como transportar un objeto.
- Cada estado representa una fase: navegar al objeto, agarrar, navegar al destino, soltar, recuperación ante fallo.
- Los estados invocan capacidades del robot (navegación, manipulación) pero no implementan su lógica interna.
- La FSM decide la estrategia y gestiona la recuperación; las capacidades ejecutan las operaciones.



Composición misión–tarea–capacidad

- En sistemas reales, la arquitectura es jerárquica: la misión activa tareas, y las tareas invocan capacidades.
- Cada nivel tiene su FSM (o BT), y las capacidades son módulos reutilizables y desacoplados.
- Esta composición permite modularidad, escalabilidad y trazabilidad del comportamiento robótico.



Conclusiones

- Las FSM son una herramienta fundamental en ingeniería de software robótico.
- Valor arquitectónico: hacen explícito el comportamiento mediante estados bien definidos y transiciones claras.
- Las FSM constituyen la base sobre la cual se construyen sistemas más sofisticados (Behavior Trees, grafos jerárquicos).
- Para tareas de complejidad media, una FSM bien diseñada proporciona equilibrio óptimo entre expresividad, eficiencia y facilidad de depuración.
- Las extensiones (ortogonalidad, jerarquía) mitigan problemas de escalabilidad.
- Herramienta indispensable en el arsenal de robótica autónoma.

Preguntas