

# Informe Técnico: Sistema Multi-Agente de Limpieza Doméstica

---

## Introducción

Este documento analiza el comportamiento de un sistema multi-agente desarrollado en Jason (AgentSpeak) que simula un entorno doméstico. El sistema está compuesto por un robot de limpieza autónomo que debe mantener la casa limpia mientras interactúa con agentes humanos (propietarios) que se mueven libremente por el hogar.

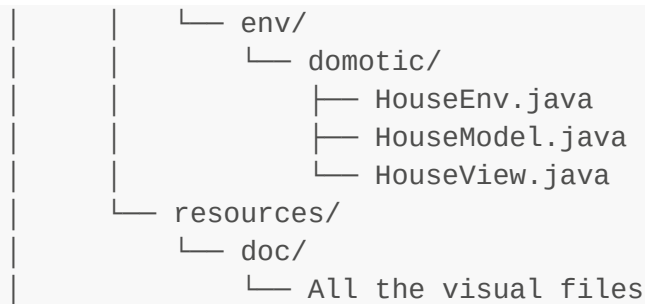
El problema que se aborda es el siguiente: un robot debe recorrer todas las habitaciones de una casa y limpiarlas completamente, mientras que los habitantes se desplazan de manera aparentemente aleatoria, sentándose, durmiéndose y realizando otras actividades cotidianas. El robot detecta la suciedad en las celdas que ocupa mediante la percepción `at(robot, dirty)` y debe garantizar que cada habitación quede completamente limpia al salir de ella.

## Arquitectura del Sistema

El sistema se compone de tres archivos principales que definen el comportamiento de los agentes:

- `movement.asl` es un módulo compartido que proporciona las funcionalidades básicas de navegación. Contiene la representación del entorno (cómo están conectadas las habitaciones), algoritmos de planificación de rutas y mecanismos para evitar que los agentes se queden atascados.
- `robot.asl` define el comportamiento del robot limpiador, incluyendo la estrategia de selección de habitaciones a limpiar, el algoritmo de barrido sistemático de cada habitación y la lógica para evitar molestar al propietario.
- `owner.asl` simula el comportamiento humano del dueño de la casa, que elige aleatoriamente entre diversas actividades como sentarse en diferentes muebles o acostarse en las camas. Para que funcionen estos agentes con su entorno, se necesita la siguiente configuración de carpetas:

```
domestic_robot/  
|  ├── docs/  
|  |   └── README.md  
|  ├── DomesticRobot.mas2j  
|  ├── lib/  
|  |   └── All the .jar dependencies  
|  └── src/  
|      ├── agt/  
|      |   ├── movement.asl  
|      |   ├── owner.asl  
|      |   └── robot.asl  
|      └── main/
```



## Módulo de Movimiento y Navegación

El módulo de movimiento actúa como biblioteca base para todos los agentes del sistema. La representación del entorno se basa en conexiones explícitas entre habitaciones mediante predicados del tipo `connect(habitacion1, habitacion2, puerta)`. Por ejemplo, `connect(kitchen, hall, doorKit1)` indica que la cocina y el hall están conectados a través de la puerta doorKit1.

### Planificación de Rutas

La planificación de rutas se realiza mediante dos predicados principales. El predicado `findPathRoom` implementa una búsqueda en profundidad limitada que explora recursivamente las conexiones entre habitaciones. Este predicado es especialmente interesante porque mantiene una lista de puertas visitadas para evitar ciclos:

```
findPathRoom(Current, Target, Visited, Path, MaxDepth)
:-
    connect(Current, NextRoom, Door)
    &
    minusOne(MaxDepth, N1)
    &
    N1 > 0
    &
    not .member(Door, Visited)
    &
    findPathRoom(NextRoom, Target, [Door|Visited], SubPath, N1)
    &
    Path = [Door|SubPath].
```

El predicado `shortestRoomPath` optimiza la búsqueda intentando primero con profundidades menores, lo que garantiza encontrar el camino más corto.

### Navegación entre Habitaciones

La navegación entre habitaciones se gestiona mediante el plan `goToRoom`, que calcula el camino óptimo y luego realiza un único movimiento hacia la primera puerta del camino. Este plan no es cíclico: se invoca repetidamente desde el bucle del agente que lo llama, pero cada invocación realiza solo un paso de movimiento.

Un aspecto crítico de la navegación es la detección y resolución de atascos. Si el agente se encuentra en una puerta durante dos ciclos consecutivos (lo que se detecta mediante la variable temporal `wasAtDoor`), se considera que está atascado y ejecuta movimientos aleatorios hasta liberarse.

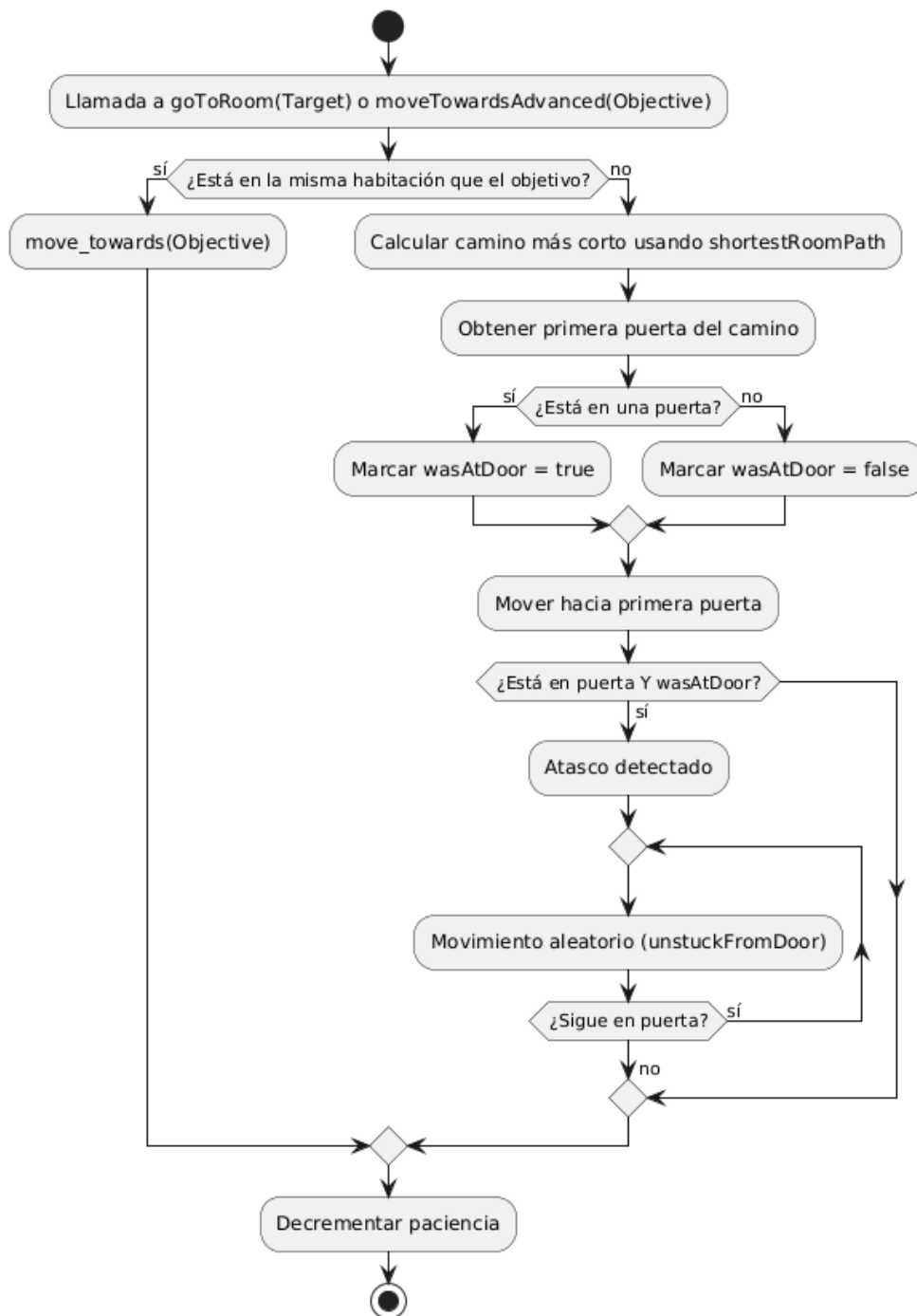
## Mecanismo de Paciencia

Un elemento clave para la robustez del sistema es el mecanismo de paciencia. Cada agente mantiene un contador (`patience`) que se decrementa con cada movimiento. Cuando este contador llega a cero, se activa automáticamente un plan reactivo (mediante `+patience(0)`) que ejecuta un movimiento aleatorio y reinicia el contador. Este mecanismo funciona de forma similar a como se activan otros planes reactivos como `+at(robot, dirty)`.

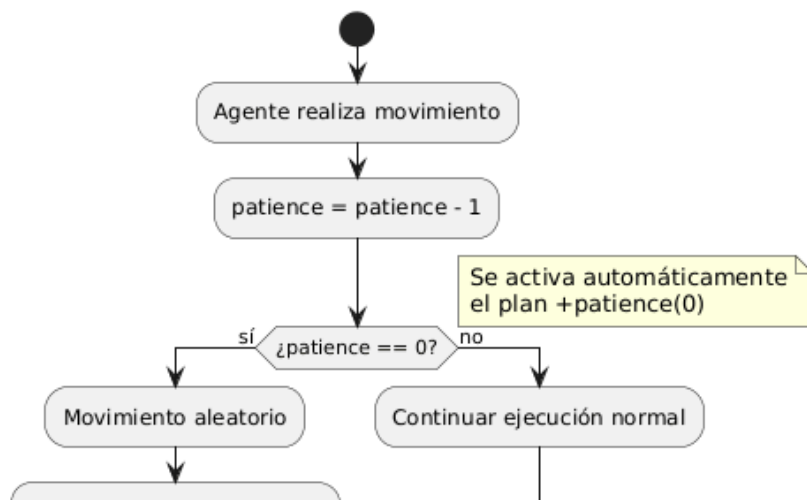
El contador de paciencia se reinicia en dos situaciones: automáticamente cuando llega a cero tras el movimiento aleatorio, o manualmente cuando el robot o el owner alcanzan sus objetivos y llaman a `!resetPatience`.

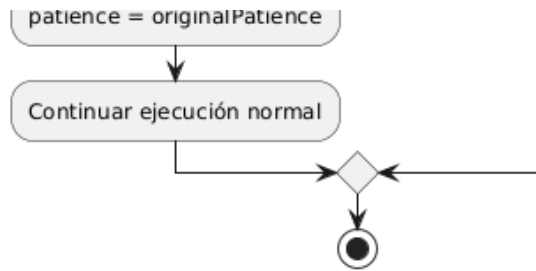
## Movimientos Especiales para el Barrido

Para el robot, que debe barrer habitaciones sin salir de ellas, existen variantes especiales de los movimientos básicos: `moveUpNoExit`, `moveDownNoExit`, `moveLeftNoExit` y `moveRightNoExit`. Estas versiones verifican después de cada movimiento si el agente ha salido de la habitación o está en una puerta, y en ese caso deshacen el movimiento. Además, actualizan los contadores de posición (`height` y `width`) que el robot usa para controlar el barrido sistemático.



### Mecanismo de Paciencia





## Agente Robot

El robot opera mediante un bucle principal que ejecuta continuamente dos objetivos: **cleaningLoop** para la limpieza y **evadeOwner** para evitar al propietario. Este ciclo se repite indefinidamente, permitiendo que el robot responda de manera reactiva a los cambios en el entorno.

La estrategia de limpieza del robot es sofisticada y se divide en varias fases. Primero, el robot debe elegir qué habitación limpiar. Para ello, obtiene la lista de todas las habitaciones sucias y calcula el camino más corto a cada una de ellas. La selección no es trivial: el robot prioriza habitaciones que no sean el pasillo (a menos que sea la única opción), ya que el pasillo se limpia naturalmente al transitar por él. El código que implementa esta lógica itera sobre las habitaciones barajadas y mantiene la mejor opción encontrada:

```

for ( .member(Room, ShuffledRooms) ) {
  ?shortestRoomPath(CurrentRoom, Room, Path, MaxDepth);
  .length(Path, PathLength);
  ?bestRoom(CurrentBest, BestLen);

  if ((PathLength < BestLen | CurrentBest = hallway) &
      (not Room = hallway | CurrentBest = nil)) {
    -bestRoom(_, _);
    +bestRoom(Room, PathLength);
  };
}

```

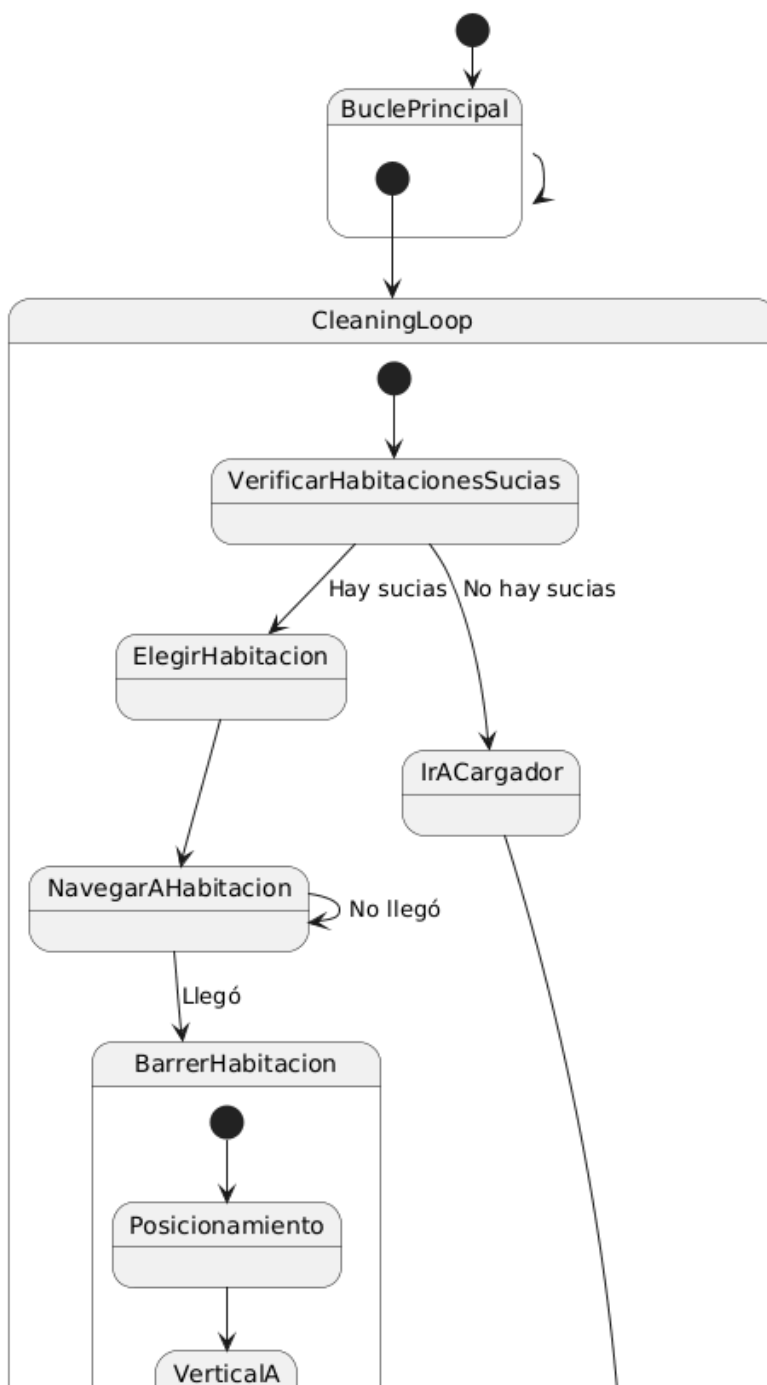
Una vez que el robot llega a la habitación elegida, comienza el proceso de barrido sistemático. Este es quizás el aspecto más técnico del comportamiento del robot. El barrido se realiza en cuatro fases distintas que garantizan que cada celda de la habitación sea visitada al menos una vez después de bajar a la parte inferior izquierda:

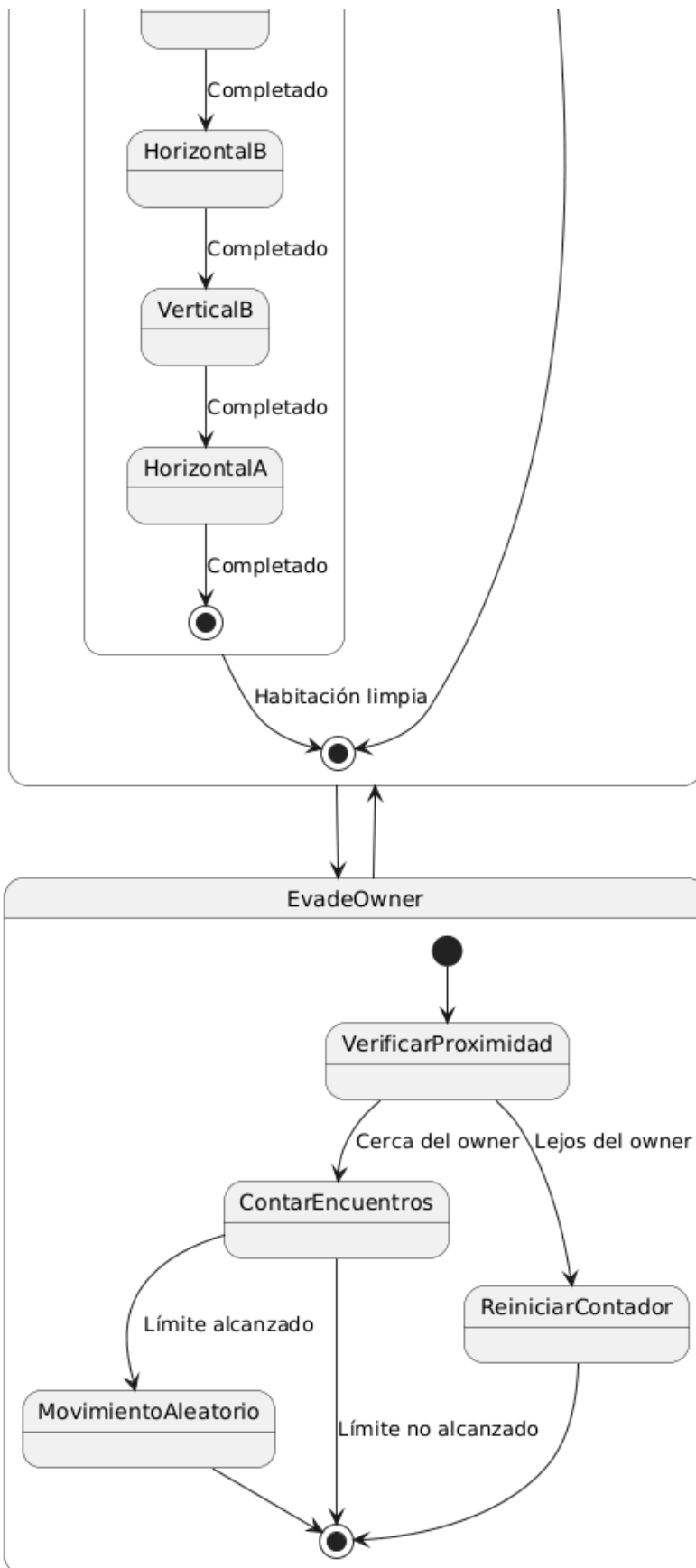
- La primera fase (Vertical A) consiste en subir/bajar por las columnas de la habitación hacia la derecha.
- La segunda fase (Horizontal B) continúa desde donde terminó la anterior, moviéndose horizontalmente y limpiando fila por fila.
- La tercera fase (Vertical B) invierte el patrón, columna por columna mientras se mueve hacia la izquierda.
- Finalmente, la fase Horizontal A completa el barrido moviéndose hacia la izquierda fila por fila.

El robot mantiene varias variables de estado para controlar este proceso: `height(X)` y `width(Y)` rastrean la posición relativa en el barrido, `bottomReached` y `leftReached` marcan el fin de las fases iniciales de posicionamiento, y las variables `movingUp`, `movingDown`, etc., indican la dirección actual del movimiento. Los flags `verticalSweepA`, `horizontalSweepB`, etc., determinan qué fase del barrido está activa.

Además de la limpieza, el robot reacciona a percepciones del entorno mediante planes activados por la adición de creencias. Cuando detecta `at(Me, dirty)`, limpia inmediatamente esa celda. Si detecta `at(Me, intruder)`, emite una alerta y envía un mensaje al propietario.

La evasión del propietario es un comportamiento interesante que añade "cortesía" al robot. El sistema cuenta cuántas veces consecutivas el robot coincide con el dueño en la misma celda. Tras 5 coincidencias, el robot se disculpa y realiza un movimiento aleatorio para salir del camino. Este contador se reinicia cuando el robot ya no está cerca del propietario.





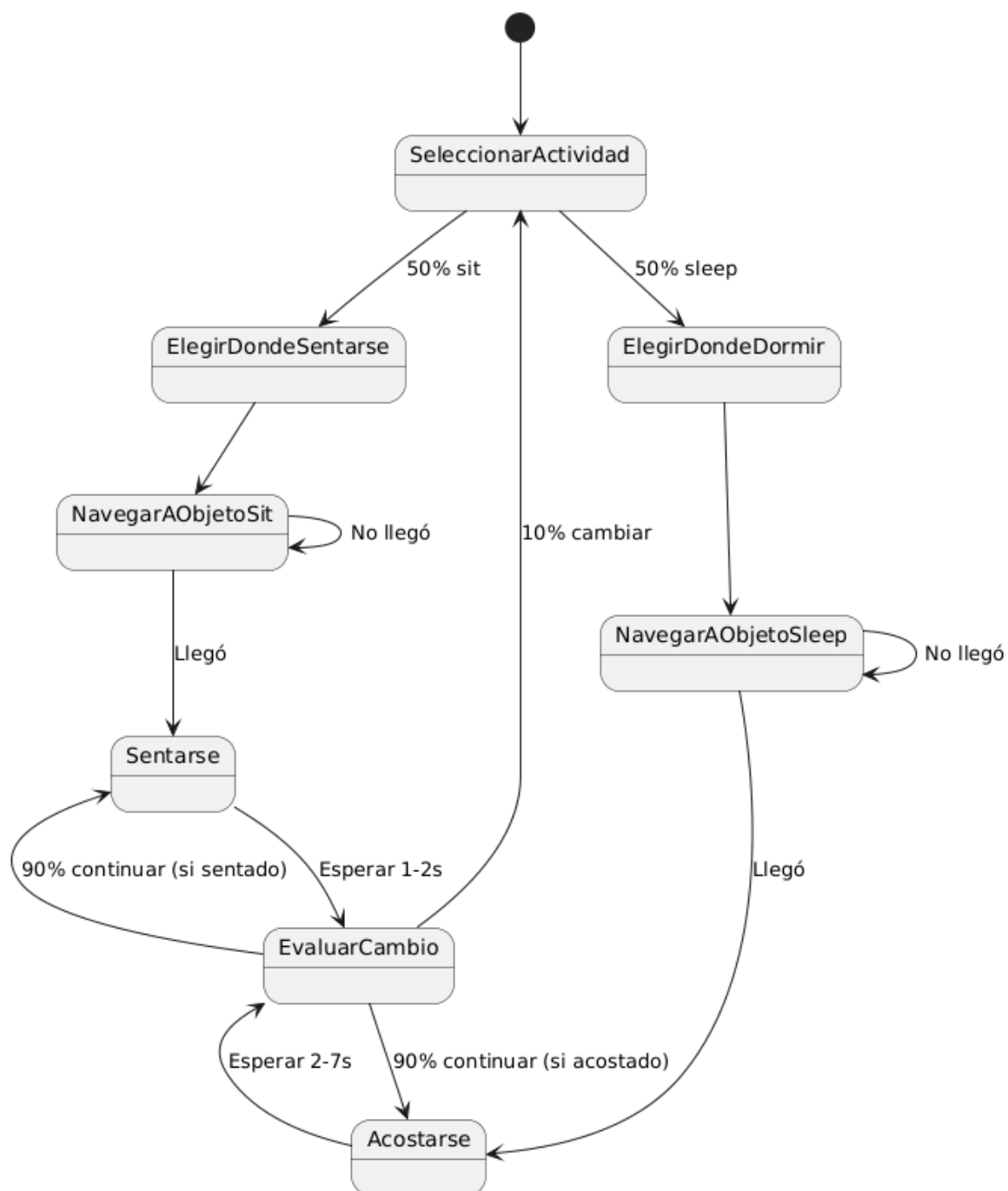
## Agente Owner

El propietario tiene un comportamiento mucho más simple pero igualmente interesante. Su ciclo principal consiste en elegir una actividad, ejecutarla y eventualmente cambiar a otra actividad.

El proceso comienza con la selección aleatoria entre dos tipos de objetivos: sentarse (50% de probabilidad) o dormir (50% de probabilidad). Una vez elegido el tipo, el agente selecciona aleatoriamente un objeto específico de las listas predefinidas. Por ejemplo, para sentarse puede elegir entre el sofá o varias sillas, mientras que para dormir elige entre las camas disponibles.

Una vez elegido el objetivo, el propietario navega hacia él utilizando el plan **moveTowardsAdvanced** del módulo de movimiento. Al llegar, ejecuta la acción correspondiente (sentarse o acostarse) durante un tiempo aleatorio: entre 1 y 2 segundos para sentarse, o entre 2 y 7 segundos para dormir. Tras este periodo, hay un 10% de probabilidad de que decida cambiar de actividad, reiniciando el ciclo.

El owner también puede recibir comunicación del robot. Específicamente, si el robot detecta un intruso, envía un mensaje que el propietario recibe y procesa, aunque en esta implementación solo emite una alerta.



## Características Destacadas del Diseño

El sistema presenta varias características que merecen destacarse. En términos de robustez, los mecanismos de paciencia y detección de atascos garantizan que los agentes nunca se bloqueen indefinidamente. La validación de movimientos durante el barrido asegura que el robot no abandone prematuramente una habitación.

Respecto a la eficiencia, la planificación de rutas encuentra siempre el camino más corto entre habitaciones, y el algoritmo de barrido garantiza una cobertura completa sin redundancia. La priorización inteligente de habitaciones evita trabajo innecesario en el pasillo.

La coordinación entre agentes es emergente más que explícita. El robot evita proactivamente al propietario tras múltiples encuentros, pero no hay negociación ni planificación conjunta. La comunicación se limita a alertas asíncronas mediante mensajes. Cada agente persigue sus objetivos de manera independiente, y el sistema funciona por la suma de estos comportamientos individuales.

## Mejoras al entorno

Nos gustaría proponer algunas mejoras al entorno:

### Charger

Ahora mismo el `move_towards` no funciona con el cargador aunque el robot se encuentre en la misma habitación.

### Detectar obstáculos

Ahora mismo el robot limpia muy lentamente porque no sabe cuando no puede continuar. Esto se podría solucionar de varias formas, por ejemplo:

- Añadir una percepción que se choque con un obstáculo.
- Añadir unas percepciones que dicen si hay obstáculos en las celdas adyacentes (arriba, abajo, izquierda, derecha).
- Añadir unas percepciones que dicen la distancia al obstáculo más cercano en cada dirección.

## Conclusiones

El sistema demuestra cómo la arquitectura BDI (Beliefs-Desires-Intentions) de Jason permite implementar comportamientos complejos de manera declarativa. El diseño modular, con el módulo de movimiento compartido, facilita la reutilización de código y simplifica el mantenimiento.

El robot implementa un comportamiento determinista y complejo para cumplir su objetivo de limpieza exhaustiva, mientras que el propietario exhibe un comportamiento estocástico que añade imprevisibilidad al entorno. Esta combinación crea un sistema dinámico donde el robot debe adaptarse constantemente a las condiciones cambiantes.

La robustez del sistema ante situaciones imprevistas (atascos, colisiones) demuestra que los mecanismos de recuperación implementados son efectivos. El sistema puede funcionar

indefinidamente sin intervención externa, cumpliendo con el objetivo de mantener la casa limpia mientras respeta el espacio del propietario.