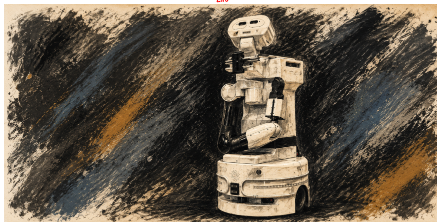
Behavior Trees

Francisco Martín Rico y Rodrigo Pérez Rodríguez



1. Parte I – Principios teóricos
2. Parte II – Aplicación

1 Parte I – Principios teóricos

2 Parte II – Aplicación

Intelligent Robotics Lab

Origen histórico: de los videojuegos a la robótica

- Los Behavior Trees no nacieron en robótica, sino en la industria de los videojuegos (años 2000).
- Problema: modelar comportamientos complejos de NPCs (Non-Player Characters) de forma escalable.
- Las FSM tradicionales resultaban inmanejables: docenas de enemigos $\times$ múltiples comportamientos $=$ explosión de estados y transiciones.
- Modificar un comportamiento requería rastrear transiciones dispersas por todo el grafo.

Ejemplo: Juegos AAA con enemigos que patrullan, persiguen, atacan, buscan cobertura, piden refuerzos, huyen...

Origen histórico: Halo 2 y la revolución de los BT

- **Halo 2 (Bungie Studios, 2004):** Damián Isla propuso una arquitectura jerárquica basada en árboles.
- Permitía componer comportamientos complejos a partir de bloques reutilizables.
- **Halo 3 (2007):** Isla formalizó completamente el concepto de BT y lo presentó en la *Game Developers Conference* (GDC) de 2005: *«Handling Complexity in the Halo 2 AI»*

Ventajas que explicó Isla:

- Expresar jerarquías de decisión de forma natural (estrategia → táctica → acción)
- Reutilizar subárboles completos entre diferentes tipos de enemigos
- Iterar rápidamente en el diseño sin refactorizar toda la arquitectura
- Visualizar el comportamiento de forma intuitiva para diseñadores no programadores

Adopción en la industria y transición a la robótica

- **Adopción rápida:** Spore (2008), Crysis (2007), Unreal Engine, Unity, CryEngine...
- Prácticamente todos los motores de videojuegos modernos incorporaron sistemas BT nativos.
- **Transición a robótica (2010-2012):** Investigadores en robótica de servicio y vehículos autónomos comenzaron a experimentar con BT.
- **Pioneros:** Michele Colledanchise y Petter Ögren (KTH Royal Institute of Technology, Suecia).
- **2014:** Primer trabajo académico riguroso: «*Behavior Trees in Robotics and AI: An Introduction*»
- Formalizó matemáticamente propiedades de los BT: reactividad, modularidad, determinismo.

Hoy: Los BT se han consolidado como alternativa arquitectónica a FSM en robótica autónoma, especialmente en ROS 2 (BehaviorTree.CPP).

Motivación de los Behavior Trees

- Los BT surgen como respuesta a limitaciones de escalabilidad y mantenibilidad de las FSM.
- A medida que los sistemas robóticos crecen en complejidad, se necesitan mecanismos jerárquicos y modulares.
- Un BT organiza el comportamiento como un árbol de nodos que se evalúan de manera recurrente.
- Facilita la representación explícita de prioridades, secuencias y condiciones de ejecución.

Ejemplo: Robot de servicio recoge objeto de una mesa

- FSM: estados explícitos para cada fase + transiciones + requisitos adicionales = explosión de complejidad
- BT: jerarquía con `Sequence` (coordina subtareas) + `Fallback` (gestiona reintentos)
- Añadir «verificar batería» = insertar nodo en jerarquía (sin modificar estructura existente)

Ventaja arquitectónica fundamental: Composición modular

Comparación con FSM

FSM y BT persiguen el mismo objetivo (modelar comportamiento), pero desde perspectivas distintas:

Aspecto	**FSM**	**Behavior Tree**
Unidad básica	Estado persistente	Nodo de ejecución
Flujo de control	Transiciones entre estados	Evaluación recursiva del árbol
Reactividad	Mediante eventos/condiciones	Reevaluación continua desde raíz
Composición	Ortogonalidad (estados paralelos)	Jerarquía de subárboles
Escalabilidad	Explosión de estados	Crecimiento logarítmico
Persistencia	Estado persiste hasta transición	Cada tick recalcula flujo
Modularidad	Media (estados acoplados)	Alta (subárboles independientes)

Comparación con FSM

¿Cuándo usar cada uno?

Una FSM es óptima cuando:

- El comportamiento tiene estados claramente diferenciados con persistencia temporal significativa
- Las transiciones están bien definidas y son relativamente escasas
- Se requiere control fino sobre cuándo y cómo ocurren las transiciones

Un BT es preferible cuando:

- El comportamiento se estructura naturalmente como una jerarquía de decisiones
- Se requiere alta reactividad ante cambios del entorno
- La reutilización y composición modular son prioritarias
- El sistema debe escalar a decenas o cientos de comportamientos distintos

Anatomía de un Behavior Tree

- Un BT se compone de nodos organizados jerárquicamente.
- Cada nodo representa una unidad de comportamiento.
- Al ejecutarse, devuelve uno de tres estados posibles: `SUCCESS`, `FAILURE` o `RUNNING`.
- Este modelo simple pero poderoso permite expresar comportamientos complejos mediante composición.

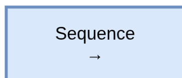
Estados de retorno:

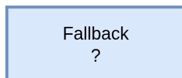
- **SUCCESS:** El nodo completó su tarea satisfactoriamente.
- **FAILURE:** El nodo no pudo completar su tarea.
- **RUNNING:** El nodo está en ejecución y requiere más tiempo.

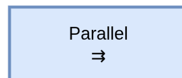
Clave: Protocolo asíncrono que permite acciones de larga duración sin bloquear la evaluación del árbol.

Tipos de nodos en un Behavior Tree

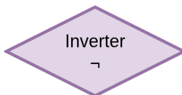
Nodos de Control

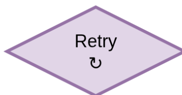


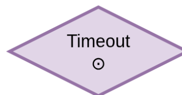




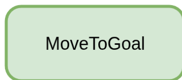
Decoradores







Acción



Condición

Tipos de nodos: resumen

Tipo	Hijos	Estados	Función principal
Nodos de Control	1:N	S, F, R	Orquestar ejecución de múltiples hijos según reglas de control de flujo
Decoradores	1	S, F, R	Modificar o transformar comportamiento/resultado del hijo
Nodos de Acción	0	S, F, R	Ejecutar operaciones concretas con efectos en el sistema
Nodos de Condición	0	S, F	Evaluar predicados sin efectos secundarios (instantáneo)

Nodos de control principales:

- `Sequence`: Ejecuta hijos en orden hasta que uno falla
- `Fallback` (o `Selector`): Ejecuta hijos hasta que uno tiene éxito

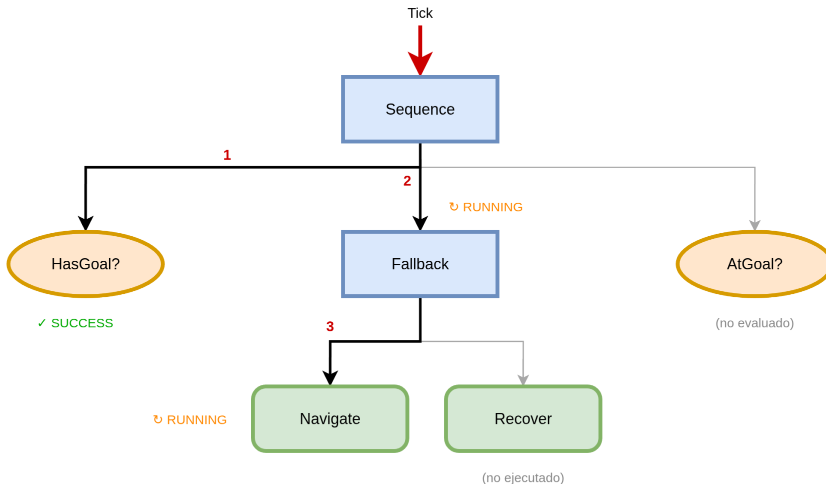
Ejecución: el concepto de tick

- El árbol se evalúa mediante **ticks** periódicos.
- En cada tick, se envía una señal desde la raíz que se propaga recursivamente por el árbol.
- Cada nodo de control decide qué hijos tickear y en qué orden según su lógica.
- Esta evaluación reactiva garantiza respuesta inmediata a cambios en el entorno.

Algoritmo de tick (patrón recursivo):

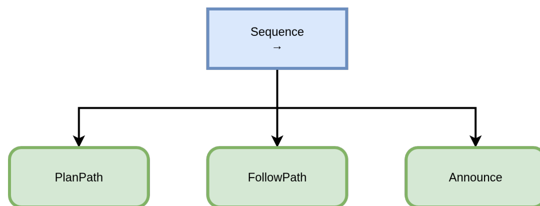
1. El nodo raíz recibe un tick
2. Según su tipo, decide qué hijos tickear y en qué orden
3. Cada hijo retorna `SUCCESS`, `FAILURE` o `RUNNING`
4. El nodo padre agrega estos resultados según su lógica
5. El resultado agregado se propaga hacia arriba

Propagación de tick y agregación de resultados



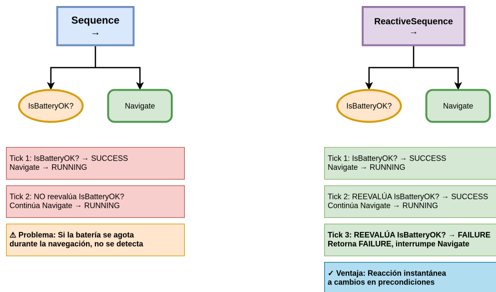
Nodo Sequence: versión con memoria

- Símbolo: $\rightarrow$
- Ejecuta sus hijos de izquierda a derecha.
- **Retorna:**
  - `SUCCESS` si **todos** los hijos retornan `SUCCESS`
  - `FAILURE` tan pronto como un hijo retorna `FAILURE`
  - `RUNNING` si el hijo actual retorna `RUNNING`
- Semántica: operador lógico AND
- Uso: requisitos en cadena «primero haz A, luego B, luego C»



ReactiveSequence: versión sin memoria

- El `ReactiveSequence` elimina la memoria de estado.
- En cada tick, reevalúa desde el primer hijo.
- Esta diferencia es fundamental para la reactividad del sistema.



Ventaja: Verifica precondiciones continuamente. Muy útil cuando las condiciones iniciales pueden invalidarse durante la ejecución de acciones largas.

Sequence vs ReactiveSequence: comportamiento comparado

Comportamiento con Sequence estándar:

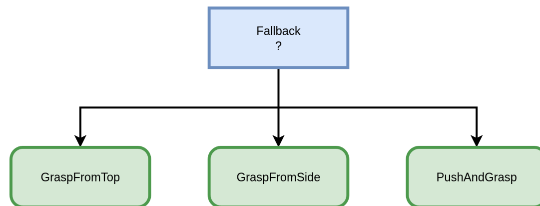
1. Tick 1: Evalúa `IsBatteryOK?` → `SUCCESS`. Continúa a `Navigate` → `RUNNING`
2. Tick 2: El `Sequence` recuerda que está en `Navigate`, **no reevalúa** `IsBatteryOK?`
3. Ticks 3-N: Mismo comportamiento (mientras `Navigate` devuelva `RUNNING`). `IsBatteryOK?` nunca se vuelve a verificar
4. Problema: Si la batería se agota durante navegación, el sistema no reacciona hasta que `Navigate` complete

Comportamiento con ReactiveSequence:

1. Tick 1: Evalúa `IsBatteryOK?` → `SUCCESS`. Continúa a `Navigate` → `RUNNING`
2. Tick 2: **Reevalúa** `IsBatteryOK?` → `SUCCESS`. Continúa con `Navigate` → `RUNNING`
3. Ticks 3-N: **Reevalúa** `IsBatteryOK?` → `FAILURE` (batería baja)
4. El `ReactiveSequence` retorna `FAILURE` inmediatamente, interrumpiendo `Navigate`
5. Ventaja: Reacción instantánea a cambios en las precondiciones

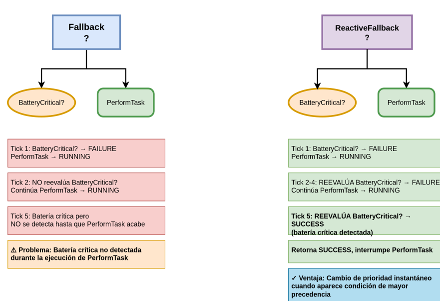
Nodo Fallback: versión con memoria

- También llamado `Selector`. Símbolo: **?**
- Ejecuta sus hijos de izquierda a derecha buscando uno que tenga éxito.
- **Retorna:**
  - `SUCCESS` tan pronto como un hijo retorna `SUCCESS`
  - `FAILURE` si **todos** los hijos retornan `FAILURE`
  - `RUNNING` si el hijo actual retorna `RUNNING`
- Semántica: operador lógico OR
- Uso: alternativas con prioridad «intenta A; si falla, intenta B; si falla, intenta C»



ReactiveFallback: versión sin memoria

- El `ReactiveFallback` no mantiene memoria.
- En cada tick, reevalúa desde el primer hijo.
- Permite cambios de prioridad instantáneos cuando aparecen condiciones de mayor precedencia.



Ventaja: Monitoriza prioridades continuamente. Muy útil en sistemas donde pueden aparecer condiciones de alta prioridad que deben prevalecer sobre el comportamiento en curso.

Fallback vs ReactiveFallback: comportamiento comparado

Comportamiento con Fallback estándar:

1. Tick 1: Evalúa `BatteryCritical?` → `FAILURE` (batería OK). Continúa a `PerformTask` → `RUNNING`
2. Tick 2: El `Fallback` recuerda que está en `PerformTask`, **no reevalúa** `BatteryCritical?`
3. Tick 5: La batería se vuelve crítica, pero `BatteryCritical?` no se reevalúa. La tarea continúa hasta completar
4. Problema: La condición crítica no se detecta hasta que `PerformTask` termina, arriesgando apagado inesperado

Comportamiento con ReactiveFallback:

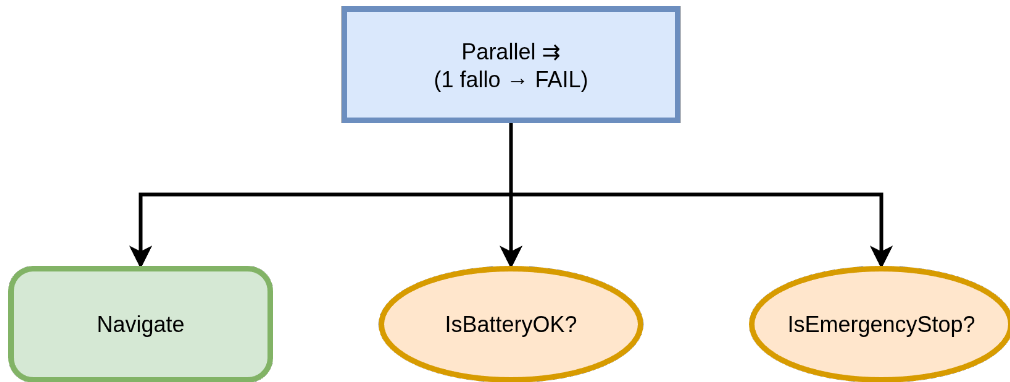
1. Tick 1: Evalúa `BatteryCritical?` → `FAILURE` (batería OK). Continúa a `PerformTask` → `RUNNING`
2. Tick 2-4: **Reevalúa** `BatteryCritical?` → `FAILURE`. Continúa con `PerformTask` → `RUNNING`
3. Tick 5: **Reevalúa** `BatteryCritical?` → `SUCCESS` (batería crítica detectada)
4. El `ReactiveFallback` retorna `SUCCESS` inmediatamente, interrumpiendo `PerformTask`
5. Ventaja: El robot puede reaccionar inmediatamente para ir a cargar, evitando apagado inesperado

Nodo Parallel: ejecución concurrente

- Símbolo: $\rightrightarrows$
- Ejecuta todos sus hijos simultáneamente en cada tick.
- Su política de terminación se configura mediante umbrales M y N:
  - Retorna `SUCCESS` cuando al menos M hijos retornan `SUCCESS`
  - Retorna `FAILURE` cuando al menos N hijos retornan `FAILURE`
  - Retorna `RUNNING` mientras no se alcancen los umbrales
- Casos comunes:
  - $M = \text{todos}, N = 1$: Todos deben tener éxito, falla al primer fallo (AND paralelo)
  - $M = 1, N = \text{todos}$: Basta un éxito, falla solo si todos fallan (OR paralelo)

Importante: `Parallel` no implica verdadero paralelismo a nivel de hilos. La «concurrencia» es lógica: el árbol no espera a que un hijo `RUNNING` complete antes de evaluar los demás.

Nodo Parallel: ejemplo de monitorización



Caso de uso típico: Ejecutar una acción principal mientras se monitorizan condiciones de seguridad. Si una condición falla, se aborta la acción principal inmediatamente (ejecución vigilada).

Resumen comparativo de nodos de control

Nodo	Retorna SUCCESS	Retorna FAILURE	Retorna RUNNING
`Sequence`	**Todos** los hijos `SUCCESS`	Tan pronto como **un hijo** `FAILURE`	El hijo actual `RUNNING`
`Reactive Sequence`	**Todos** los hijos `SUCCESS`	Tan pronto como **un hijo** `FAILURE`	El hijo actual `RUNNING`
`Fallback`	Tan pronto como **un hijo** `SUCCESS`	**Todos** los hijos `FAILURE`	El hijo actual `RUNNING`
`Reactive Fallback`	Tan pronto como **un hijo** `SUCCESS`	**Todos** los hijos `FAILURE`	El hijo actual `RUNNING`
`Parallel`	Al menos M hijos `SUCCESS`	Al menos N hijos `FAILURE`	No se alcanzan umbrales

Nodos estándar vs reactivos: comparación arquitectónica

Aspecto	**Sequence/Fallback estándar**	**ReactiveSequence/Fallback**
Memoria de estado	Recuerda qué hijo está `RUNNING`	No mantiene memoria, reevalúa desde el inicio
Reevaluación	Hijos anteriores **no** se reevalúan mientras hay uno `RUNNING`	**Todos** los hijos se reevalúan en cada tick
Reactividad	Baja: cambios en hijos anteriores ignorados	Alta: respuesta inmediata a cambios
Eficiencia	Alta: evalúa solo el hijo activo	Baja: reevalúa todos los hijos
Uso típico	Secuencias/alternativas sin cambios de contexto	Monitorización continua de condiciones

Cuándo usar nodos estándar vs reactivos

Los nodos reactivos son apropiados cuando:

- Se trabaja con precondiciones que pueden invalidarse durante la ejecución (batería, conectividad, visibilidad)
- Se deben monitorizar condiciones de seguridad (obstáculos, paradas de emergencia)
- El sistema tiene prioridades dinámicas donde eventos de alta prioridad deben interrumpir tareas en curso

Los nodos estándar son preferibles cuando:

- Las condiciones iniciales se mantienen válidas durante toda la ejecución
- La optimización de rendimiento es crucial (evitan reevaluaciones innecesarias)
- La persistencia hasta completar la tarea es deseable

Combinación efectiva: Nodos reactivos con `Parallel` proporcionan monitorización continua sin la sobrecarga de reevaluar toda la jerarquía.

Decoradores: transformadores funcionales del flujo

- Los decoradores envuelven un único hijo y modifican su comportamiento o resultado.
- Actúan como transformadores funcionales del flujo de control.
- El conjunto de decoradores es extensible y depende de la implementación del motor BT.

Decoradores fundamentales:

- **Inverter** ($\neg$): Invierte el resultado (`SUCCESS` $\leftrightarrow$ `FAILURE`)
- **Retry** (⟳): Reintenta el hijo N veces si retorna `FAILURE`
- **ForceSuccess/ForceFailure**: Reemplaza el resultado del hijo por un valor fijo
- **Timeout** ($\odot$): Aborta el hijo retornando `FAILURE` si no completa en tiempo límite

Decoradores añadiendo robustez a un árbol

Sequence

Inv ¬

T:300s ⊙

Retry:3 ↻

ForceOK

IsBatteryLow?

NavigateToPose

GraspObject

LogEvent

El `Sequence` combina cuatro operaciones protegidas: verifica que la batería *no* esté baja (invertida), navega con límite de 5 minutos, agarra con hasta 3 reintentos, y registra el evento (sin bloquear si falla).

Modelado de misiones y tareas

- Recordemos el modelo arquitectónico **misión–tarea–capacidad**.
- Los BT son versátiles: operan tanto a nivel de tarea como de misión.
- **A nivel de tarea:** Un BT encapsula la lógica de coordinación de capacidades para un objetivo concreto.
- **A nivel de misión:** Los BT actúan como orquestadores que coordinan múltiples tareas.
- Si cada tarea es un BT independiente $\rightarrow$ **behavior forest** (colección de BT orquestados por BT maestro)

Esquema híbrido efectivo: FSM en misión + BT en tarea

- Misiones: fases de larga duración con transiciones bien definidas (dominio de FSM)
- Tareas: requieren reactividad continua, monitorización múltiple, coordinación jerárquica (dominio de BT)
- FSM de misión selecciona qué fase ejecutar, cada estado activa un BT que implementa las tareas con reactividad

Clave arquitectónica: Separación de responsabilidades. Un BT define el *qué* y el *cuándo*, los niveles inferiores definen el *cómo*.

Reutilización de comportamientos

- Una de las principales ventajas de los BT es la facilidad para reutilizar comportamientos.
- Subárboles completos pueden encapsular patrones de comportamiento comunes.
- Se pueden reutilizar en distintas misiones o robots.
- La reutilización no se limita al código, sino también al diseño del comportamiento.
- Los BT permiten construir bibliotecas de comportamientos que pueden combinarse y adaptarse a nuevos contextos.

Ventajas arquitectónicas:

- Favorece la escalabilidad del sistema
- Reduce la duplicación de lógica
- Se alinea con objetivos de modularidad y mantenibilidad

1 Parte I – Principios teóricos

2 Parte II – Aplicación

Behavior Trees en ROS 2: BehaviorTree.CPP

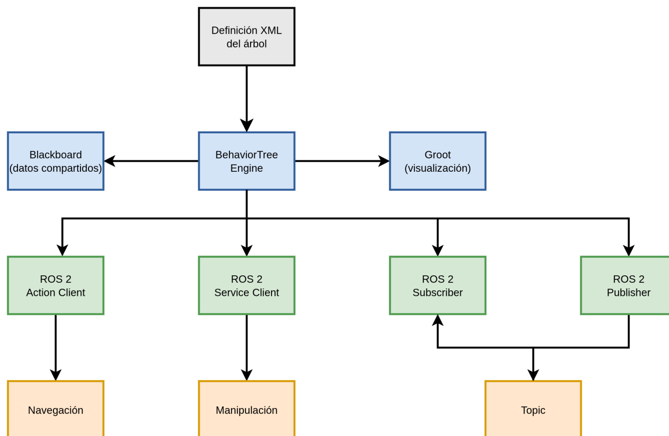
- En ROS 2, los BT se implementan mediante librerías específicas.
- Librería de referencia: **BehaviorTree.CPP** (desarrollada por Davide Faconti).
- Infraestructura robusta y extensible para implementar BT en C++.

Características principales:

- Motor de ejecución eficiente con soporte para árboles grandes
- Mecanismo de puertos (*blackboard*) para compartir datos entre nodos
- Carga de árboles desde archivos XML para facilitar iteración de diseño
- Sistema extensible para implementar nodos personalizados
- Herramientas de visualización (`Groot`) para depuración en tiempo real

Integración con ROS 2: BehaviorTree.CPP es agnóstico al framework. La integración se realiza mediante implementación de nodos personalizados que acceden a interfaces ROS 2.

Arquitectura de BehaviorTree.CPP en ROS 2



El motor BehaviorTree.CPP carga la definición del árbol (XML), proporciona blackboard para compartir datos, y ofrece integración con Groot. Los nodos del árbol interactúan con capacidades

Blackboard y puertos: comunicación sin acoplamiento

- El **blackboard** es un espacio de memoria compartida (diccionario clave-valor tipado).
- Permite comunicación de datos entre nodos del árbol sin acoplarlos directamente.
- Los nodos acceden al blackboard mediante **puertos**.
- Los puertos son interfaces declarativas que especifican qué datos necesita leer un nodo (entrada) y qué datos puede escribir (salida).

Problema que resuelve: ¿Cómo comparten datos nodos en diferentes partes del árbol sin crear dependencias explícitas?

Solución: Indirección. Los nodos no se comunican directamente, sino a través del blackboard.

Arquitectura del blackboard y sistema de puertos

DetectObject

OutputPort "detected_pose"

Blackboard

object_position: Pose
battery_level: double
goal_reached: bool

clave: "object_position"

NavigateToObject

InputPort "goal"

InputPort "target"

GraspObject

Los nodos declaran puertos (`InputPort`, `OutputPort`) que se conectan al blackboard mediante claves. Un nodo escribe en el blackboard, otros leen, permitiendo comunicación sin acoplamiento directo.

Intelligent Robotics Lab

Blackboard: ventajas arquitectónicas

Desacoplamiento estructural:

- Los nodos no necesitan referencias directas entre sí, solo declaran puertos de entrada/salida

Facilita la reutilización:

- Un nodo con puertos claros puede emplearse en diferentes contextos sin modificación

Composición flexible:

- Los árboles se reconfiguran cambiando conexiones en XML sin recompilar código

Observación y depuración:

- El blackboard proporciona un punto único para observar datos compartidos
- Herramientas como Groot inspeccionan el estado en tiempo real

Validación temprana:

- Los puertos se declaran estáticamente: el motor verifica conexiones y tipos antes de ejecutar

Ejemplo: bump-and-go con recuperación

- Comportamiento clásico que ilustra las ventajas arquitectónicas de los BT frente a FSM.
- El robot debe avanzar hacia un objetivo mientras monitoriza su sensor láser.
- Si detecta obstáculo cercano (< 0.5m), ejecuta recuperación: retroceder y girar.

Estructura: `Fallback` raíz establece política: intenta gestionar obstáculos (prioridad seguridad), luego navegación (objetivo principal), finalmente abortar (último recurso).

Ejemplo: bump-and-go con recuperación

1. **Datos globales de misión:**
   - Programa principal establece `goal_x`, `goal_y`, `goal_theta` en blackboard global
   - Nodo `MoveTowardsGoal` lee estos valores mediante puertos de entrada
2. **Conexión directa entre nodos:**
   - `IsObstacleNear` detecta distancia mínima y escribe en puerto de salida `obstacle_distance`
   - Esta información fluye hacia `BackUp` y `Spin`, que la usan para parametrizar su recuperación
   - Si obstáculo muy cerca (< 0.3m), `Spin` gira 180°; si más lejos, solo 90°
   - `BackUp` retrocede proporcionalmente a la distancia detectada
3. **Parámetros con valores por defecto:**
   - Nodos declaran parámetros configurables con valores razonables
   - Permite ajustar el comportamiento desde XML sin modificar código

Comparación con FSM equivalente: Necesitaríamos estados explícitos para cada fase + transiciones complejas + variables globales. El BT es mucho más compacto y legible.

Ejemplo: bump-and-go con recuperación

Fallback
(MainLoop)

Sequence
(BatteryManagement)

ReactiveFallback
(BumpAndGoLoop)

IsBatteryLow?
threshold=0.10

ReturnToCharger
{charger_x}, {charger_y}

Sequence
(ObstacleRecovery)

MoveForward
speed=0.3

IsObstacleNear?
threshold=0.5
→ obstacle_distance

Sequence
(RecoveryManeuver)

obstacle_distance

BackUp
{obstacle_distance}
base_distance=0.3

Spin
{obstacle_distance}
base_angle=1.57

Intelligent Robotics Lab

Integración con capacidades externas

- En sistemas reales, las capacidades se implementan como acciones o servicios ROS 2.
- Ejemplo: `MoveTowardsGoal` puede conectarse con Nav2 (`/navigate_to_pose`).
- Nav2 proporciona navegación global robusta con planificación A*/Theta*, control DWB, capas de costmap.

Patrón arquitectónico:

- Los nodos BT actúan como clientes de acciones/servicios
- Delegan la ejecución mientras supervisan progreso
- Deciden cuándo activar, cancelar o reintentar

Arquitectura asíncrona:

- `onStart()`: envía objetivo al servidor de acción y retorna `RUNNING`
- `onRunning()`: consulta periódicamente si ha completado (polling)
- `onHalted()`: cancela la acción si el BT requiere interrumpirla

Arquitectura de integración con ROS 2

Separación clara de responsabilidades:

- **Nodo BT:** Decide *cuándo* invocar la capacidad y *qué hacer* con el resultado
- **Servidor de acción:** Decide *cómo* ejecutar la tarea
- El nodo actúa como adaptador entre árbol BT (evaluación síncrona) y acciones ROS 2 (ejecución asíncrona)

Roles diferenciados:

- **BT:** Proporciona supervisión, recuperación ante fallos, orquestación de múltiples capacidades
- **Capacidades:** Encapsulan su implementación específica

El BT orquesta,
las capacidades ejecutan

Intelligent Robotics Lab

Gestión de preemption y supervisión

Gestión de preemption (interrupción anticipada):

- *Preemption:* Capacidad de cancelar una acción en curso antes de que complete
- El BT puede interrumpir la ejecución si una condición de mayor prioridad lo requiere
- Ejemplo: batería baja → interrumpir navegación → ir a estación de carga
- Mecanismo genérico aplicable a cualquier servidor de acción ROS 2

Supervisión continua:

- Los nodos reactivos permiten monitorizar condiciones mientras se ejecutan acciones largas
- El `Parallel` permite ejecutar capacidades mientras se vigilan condiciones de seguridad
- Si una condición crítica falla, el BT puede interrumpir inmediatamente la acción en curso

Ventaja clave:

- Las capacidades pueden diseñarse sin conocimiento de las políticas de supervisión
- El BT añade la lógica de supervisión y recuperación sin modificar las capacidades existentes

Conclusiones: ventajas de los Behavior Trees

Ventajas arquitectónicas y funcionales:

- **Expresión jerárquica:** Prioridades y decisiones se modelan naturalmente como árbol
- **Composición modular:** Comportamientos reutilizables que se combinan fácilmente
- **Alta reactividad:** Respuesta inmediata ante cambios del entorno
- **Recuperación automática:** Nodos `Fallback` gestionan fallos con estrategias de respaldo

Ventajas de implementación:

- **Integración natural:** Conexión directa con acciones y servicios ROS 2
- **Escalabilidad:** Sin explosión de estados (problema típico de FSM grandes)
- **Visualización intuitiva:** Estructura gráfica clara del comportamiento
- **Mantenibilidad:** Fácil modificación y extensión de comportamientos

BT vs FSM: guía de decisión

Preferir Behavior Trees cuando:

- El comportamiento se estructura como jerarquía de decisiones
- La reactividad a cambios del entorno es prioritaria
- La reutilización y composición modular son fundamentales
- El sistema debe escalar a decenas o cientos de comportamientos
- Se requiere supervisión continua de múltiples condiciones

Preferir FSM cuando:

- El comportamiento tiene fases bien definidas y de larga duración
- Las transiciones entre estados son explícitas y claras
- El número de estados es manejable (< 10-15)
- La persistencia de estado es más importante que la reactividad

Esquema híbrido efectivo: FSM a nivel de misión (fases de larga duración) + BT a nivel de tarea (reactividad continua).

¿Preguntas?

Intelligent Robotics Lab