TOROCÓ
SISTEMA DE CONTROL DE ROBOTS BASADO EN COMPORTAMIENTOS

INFORME FINAL

Ignacio Bettosini - Agustín Clavelli

Tutores
Facundo Benavides - Jorge Visca

Instituto de Computación
Facultad de Ingeniería - Universidad de la República
Montevideo, Uruguay

Febrero de 2015

Resumen

El sistema de control de un robot es el programa que controla el funcionamiento del robot, es decir que obtiene datos del entorno provenientes de los sensores, decide cómo responder ante la situación y envía comandos a los actuadores.

Para construir el sistema de control de un robot, se debe determinar su arquitectura, es decir la organización de las tareas que ejecuta el programa. Aparte de que cumpla su cometido, es deseable que el sistema de control tenga otras cualidades, tales como que sea fácil de usar, flexible, extensible y reutilizable.

En el informe se presenta una biblioteca para desarrollar sistemas de control de robots móviles que sigue los objetivos planteados. Utiliza la arquitectura Subsumption, en la que las tareas del robot se organizan en comportamientos reactivos y concurrentes. Subsumption además establece operaciones para que los comportamientos interactúen entre sí y con los sensores y actuadores.

La biblioteca ofrece un sistema de comunicación basado en eventos, lo que evita la necesidad de realizar polling. A su vez, se desarrolló en el lenguaje Lua, por lo que es portable a múltiples plataformas robóticas. Para comprobar el funcionamiento de la biblioteca, se realizó un prototipo de robot sobre la plataforma Butiá.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción al proyecto

Este capítulo presenta una introducción al proyecto de desarrollo de un sistema de control de robots basada en la arquitectura Subsumption. La sección 1 describe la motivación del proyecto. En la sección 2 se plantean los objetivos del proyecto, y en la sección 3 se describe la estructura del informe.

1.1. Motivación

Dentro de la robótica, una de las áreas de investigación son los sistemas de control de robots, es decir los programas que controlan el funcionamiento del robot. Este tema es particularmente interesante en el caso de los robots móviles autónomos, que son aquellos que pueden desplazarse libremente en el entorno según los datos que obtienen con sus sensores. Los robots móviles autónomos pueden utilizarse con fines de exploración, industriales, educativos y recreativos.

El sistema de control de un robot influye fuertemente en la forma de implementar su comportamiento y en el resultado obtenido. Al construir el software de control, además de elegir el lenguaje a utilizar, importa determinar qué arquitectura utilizará el sistema. Según la estructura topológica, las reglas y las restricciones que imponga la arquitectura, el comportamiento final del robot puede variar en forma considerable. [1] [2, p. 124]

Uno de los tipos de sistemas de control más estudiados son las arquitecturas basadas en comportamientos. En ellas, el sistema se divide en capas de competencia, cada una de las cuales cumple una tarea específica. Los comportamientos ejecutan de manera paralela, e interactúan entre sí mediante algún mecanismo de comunicación.

A mediados de la década de 1980, Brooks desarrolló una arquitectura para el control de robots móviles autónomos llamada Subsumption. [3] [2, p.130-141] Ésta sigue un enfoque reactivo basado en comportamientos, donde las capas de competencia ejecutan de forma concurrente y asíncrona en relación con las restantes, buscando cumplir sus propios objetivos.

Las capas de competencia se componen a su vez de un conjunto de módulos que intercambian datos entre sí. Los módulos de las capas superiores intervienen en la comunicación de los módulos de capas inferiores indirectamente, mediante operaciones tales como la inhibición de las salidas de datos y la supresión de las entradas.

1.2. Objetivos

El proyecto consiste en implementar una biblioteca para desarrollar sistemas de control de robots móviles autónomos que sigan la arquitectura Subsumption.

El sistema permitirá que los comportamientos se comuniquen con el hardware del robot para cumplir con su tarea, lo que incluye los mecanismos de inhibición y supresión. Los comportamientos ejecutarán concurrentemente, lo que implica que pueden recibir, procesar y enviar datos en todo momento.

Para comprobar el funcionamiento de la biblioteca, se desarrollará un prototipo de robot con dicho sistema de control. El prototipo de robot se implementará sobre la plataforma robótica Butiá, para demostrar las características más relevantes del sistema.

1.3. Estructura del documento

En el capítulo 2 del informe se presenta el estado del arte, introduciendo en el tema de la robótica basada en comportamientos. Los requisitos del sistema se detallan en el capítulo 3. En el capítulo 4 se muestra la arquitectura del sistema que se desarrolló. El capítulo 5 contiene la descripción y los resultados de las pruebas del sistema. El análisis y las conclusiones se encuentran en el capítulo 6 del informe.

Capítulo 2

Marco teórico

Este capítulo releva el marco teórico del área de control de robots, con énfasis en las arquitecturas basadas en comportamientos. La sección 2.1 introduce al concepto de control de robots, sus paradigmas y la arquitectura basada en comportamientos. La arquitectura Subsumption y su aplicación se describe en la sección 2.2. Luego se presentan casos de estudio de robots basados en comportamientos en la sección 2.3.

La sección 2.4 detalla las características de la plataforma robótica elegida para el prototipo, en particular aquellas vinculadas con el sistema de control. En la sección 2.5 se presentan consideraciones sobre características de interés de un sistema de control de robots.

2.1. Control de robots

2.1.1. Introducción

El control de robots es la especificación del comportamiento de un robot de manera que cumpla con sus objetivos en la forma deseada. [1] Se puede definir en base a las tres funciones primitivas: [4, p. 5]

- **Sensar**: Obtención de información a partir de los datos de entrada provenientes de los sensores.
- **Planificar**: Interpretación y evaluación de datos, toma de decisiones.
- **Actuar**: Determinación de los datos de salida hacia los actuadores.

Una arquitectura abarca la definición del conjunto de componentes, incluyendo su funcionalidad, topología e interfaz de interacción. [5] Es decir, una arquitectura describe el tipo de componentes de software que existen en el sistema, la interfaz de los componentes, la estructura en la que están organizados, y el mecanismo de interacción entre ellos.

La arquitectura de un sistema impone restricciones a las maneras posibles de resolver un problema dado. [6] En el caso de los sistemas de control de robots, algunas restricciones posibles son:

- **Tipos de datos**: Numéricos, texto, estructuras, discretos, etc.
- **Información almacenada**: Datos sensados, estado de comportamientos, modelo del mundo, plan de ejecución, comportamientos habilitados.
- **Tareas permitidas**: Enviar órdenes a actuadores, almacenar datos de sensores, procesar modelo del mundo, cambiar estado de comportamientos.
- **Topología de tareas**: Ordenada, en capas, libre.
- **Ejecución de tareas**: Paralela o secuencial.
- **Métodos de coordinación de tareas**: Cooperación, competencia, pizarrón.

Según las funciones del robot y las características del entorno, es preferible utilizar una arquitectura de control u otra. Esto se debe a que las restricciones de la arquitectura pueden facilitar o entorpecer la especificación del comportamiento del robot.

Por ejemplo, si se supone que un sistema debe realizar una única tarea a la vez, es preferible que la arquitectura sea secuencial, para asegurarse de que no haya múltiples tareas ejecutando a la vez. En cambio, si un robot posee varias tareas que se deben ejecutar a la vez, una arquitectura paralela permitiría implementar el sistema más fácilmente.

2.1.2. Paradigma reactivo

Uno de los paradigmas de control de robots es el reactivo, en el cual el sistema utiliza los datos de los sensores más recientes para determinar la acción a realizar en cada instante. [3] [2, p.130-141]. Un sistema de este tipo carece de memoria alguna, sino que se basa únicamente en la información más reciente.

Dicho de otra manera, la función primitiva de sensar genera una reacción inmediata en la función de actuar, como se muestra en la figura 2.1. El paradigma reactivo se inspira en los reflejos animales, que se realizan con un procesamiento mínimo. Un ejemplo es cuando una persona coloca la mano sobre una superficie excesivamente caliente, ella retira la mano inmediatamente, antes de ser consciente del dolor.

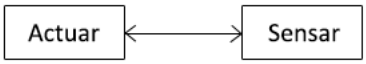

Figura 2.1 Esquema del paradigma reactivo

Al eliminar la etapa de planificación y acoplar las etapas de sensar y actuar, el paradigma reactivo pretende reducir el tiempo de respuesta del robot. No obstante, un sistema reactivo puede sufrir inconvenientes si el entorno es parcialmente observable, es decir si el robot no puede obtener información completa del entorno. Al carecer de información esencial, el sistema de control podría tomar decisiones menos buenas que si se dispusiera de dicha información.

Un ejemplo de este inconveniente del paradigma reactivo es la planificación de rutas, en cuyo caso conviene conocer el terreno completo para determinar el mejor camino. En un caso extremo, el robot puede quedar en deadlock, por ejemplo al recorrer una serie de callejones sin percatarse de que la salida se encuentra en una pared lateral.

2.1.3. Arquitecturas basadas en comportamientos

Las arquitecturas basadas en comportamientos son aquellas que se descomponen verticalmente en capas de competencia. Las capas definen una clase de comportamientos del robot, que puede ser más general (por ejemplo avanzar) o específica (avanzar en línea recta a cierta velocidad), como se muestra en el ejemplo de la figura 2.2.

Cada capa de competencia implementa las etapas de sensado y acción de manera independiente a las demás. Las capas ejecutan solamente las tareas de su interés, lo que reduce el tiempo de cómputo desde la entrada hasta la salida.

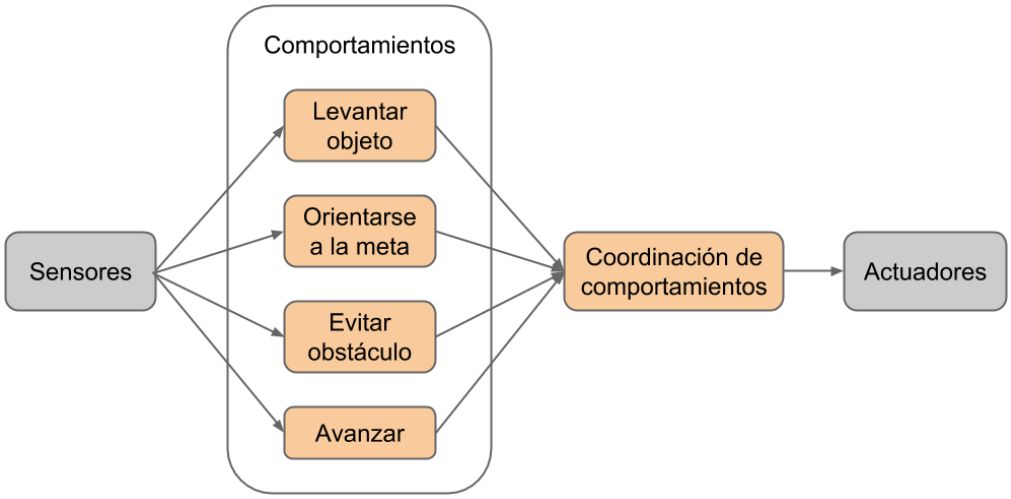

Figura 2.2 Ejemplo de arquitectura basada en comportamientos.

Las capas de competencia superiores extienden el comportamiento de las capas inferiores. Por ello, el sistema se desarrolla incrementalmente de abajo hacia arriba: cada capa nueva agrega restricciones al comportamiento de las capas inferiores.

Las arquitecturas basadas en comportamientos pueden ser paralelas o no. En el primer caso, todas las capas ejecutan al mismo tiempo. Esto requiere establecer un mecanismo de coordinación, para determinar cómo las distintas capas interactúan entre sí, en particular para utilizar los actuadores. En el segundo caso, solamente un comportamiento ejecuta a la vez. Así, se necesita un mecanismo para determinar qué capa debe estar activa en cada momento.

La descomposición en capas de competencia pretende facilitar el desarrollo y la verificación del sistema. Se puede probar el funcionamiento de cada capa de manera incremental de abajo hacia arriba. Al modificar una capa superior, no se necesita volver a estudiar el funcionamiento de las capas inferiores, sino apenas la interacción entre ellas.

2.1.4. Clasificación de arquitecturas

Un ejemplo de arquitectura basada en comportamientos es Subsumption, que se describe en la sección 3. En dicha arquitectura, cada comportamiento se ejecuta de manera reactiva. Arkin considera que las arquitecturas basadas en comportamientos en general son reactivas. [2, p. 173]

Sin embargo, Mataric [7] distingue las arquitecturas basadas en comportamientos de las

reactivas, definiendo a las primeras como aquellas cuyos comportamientos mantienen estados y representaciones de datos.

Como los comportamientos de la arquitectura Subsumption no manejan representaciones de datos [2] [3], no sería catalogada como basada en comportamientos bajo la clasificación de Mataric. Esto parece una clasificación forzada, dado que el propio Brooks impulsó las arquitecturas basadas en comportamientos.

Por otra parte, es válido plantear la existencia de arquitecturas basadas en comportamientos que cumplan las características listadas por Mataric. A su vez, dentro del paradigma híbrido es posible definir tanto arquitecturas basadas en comportamientos como sin ellos.

Por ello y para evitar confusiones, es preferible manejar la clasificación de arquitectura basada en comportamientos de manera independiente al paradigma reactivo.

2.2. Subsumption

Subsumption es una propuesta de arquitectura de control reactiva basada en comportamientos elaborada por Rodney Brooks. [3]

El autor se planteó como objetivo que la arquitectura permitiera cumplir varias metas a la vez, utilizar múltiples sensores, ser tolerante ante datos de entrada desconocidos o erróneos (robustez), y permitir extender el comportamiento del robot en forma incremental sin tener que rehacer todo el sistema (extensibilidad).

Por tanto, Brooks tomó como principio que el sistema de control debe ser simple. Por el contrario, la complejidad de la conducta deberá emerger de la interacción del robot con el entorno. A su vez, optó por una arquitectura modular, que permite diseñar, implementar y verificar los distintos comportamientos independientemente de los demás.

2.2.1. Arquitectura

En la arquitectura Subsumption, las capas de competencia se ejecutan concurrentemente. Cada una de ellas se compone de una serie de módulos. Los módulos son máquinas de estados que se ejecutan independientemente de los demás.

Cada módulo tiene entrada y salida de datos, que se conectan con otros módulos. También se comunican con el hardware: algunos módulos toman datos provenientes de sensores o envían datos a actuadores.

Del mismo modo, las capas interactúan entre sí mediante intercambio de datos. Así, las capas superiores toman como datos de entrada las salidas de módulos de capas inferiores.

En tanto, los módulos de las capas superiores intervienen en la comunicación de los módulos de capas inferiores indirectamente con tres métodos. El primero de ellos es mediante inhibición de los datos de salida, es decir que se anula la salida durante cierto período de tiempo. El segundo es supresión de datos de entrada, es decir que se sustituye la entrada durante cierto período de tiempo por otros datos provenientes del módulo superior. El tercero es una señal de reinicio del módulo.

La figura 2.3 muestra un ejemplo de sistema de control Subsumption. La capa inferior implementa el comportamiento de evitar obstáculos. El módulo “escapar” toma los datos de los sensores y determina si el robot debe evitar un obstáculo. En caso afirmativo, envía una señal al módulo “avanzar” para dejar de acelerar, y otra señal a los frenos para detener el robot.

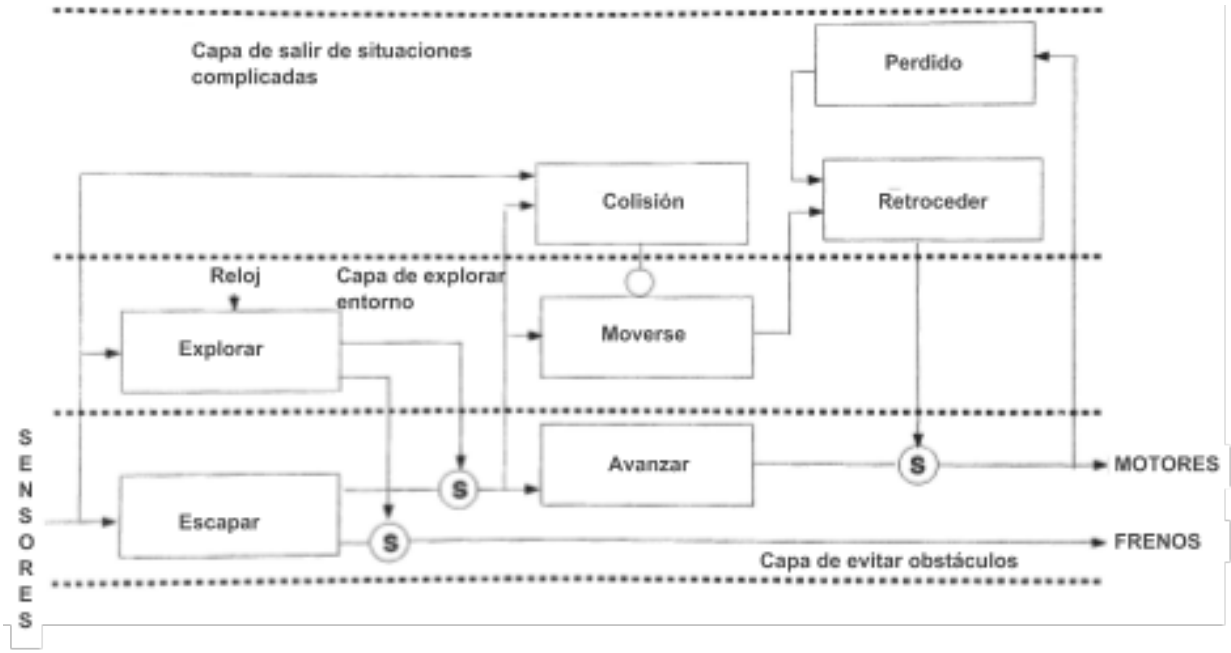

Figura 2.3 Ejemplo de sistema de control con arquitectura Subsumption [2, p. 134].
Las flechas son datos enviados de un módulo a otro.
La supresión de datos de entrada se representa con círculos "S".

La segunda capa del sistema implementa el comportamiento de explorar el entorno. El módulo “explorar” elige si realizar la exploración. En dicha situación el módulo envía señales para suprimir los canales de comunicación del módulo “escapar”, lo cual genera que el módulo “avanzar” reciba un valor positivo y los frenos se desactivan. Los módulos restantes funcionan de manera análoga.

2.2.2. Características

La arquitectura Subsumption es paralela, dado que cada módulo puede ejecutar en un procesador independiente. Si un módulo queda bloqueado procesando gran cantidad de datos, los demás módulos siguen funcionando con normalidad. Asimismo, se puede aumentar el poder de cómputo del sistema instalando más procesadores. Debido al paralelismo, el tiempo de respuesta es menor que en el caso secuencial. [8]

También es una arquitectura robusta, dado que si un módulo falla, el resto del sistema puede funcionar, en particular si la falla ocurre en capas superiores.

No obstante, según Rosenblatt y Payton [8] la arquitectura tiene como inconveniente que, para poder implementar el comportamiento deseado en una capa de competencia, en la práctica se requiere modificar los módulos de las capas inferiores. Por tanto, según los autores no se cumple el principio de desarrollo de abajo hacia arriba.

2.2.3. Aplicación a robots móviles autónomos

Un agente autónomo consiste en una entidad que puede interactuar con el entorno en base a la información obtenida de sus sensores. [9] Por el contrario, otros robots son teleoperados, es decir que un operario humano controla sus acciones con mayor o nivel de abstracción. [10]

El nivel de autonomía de un robot es mayor si posee funcionalidades de inteligencia artificial, tales como razonamiento cognitivo y aprendizaje. Esto sirve para que el robot tenga un mejor desempeño, en ocasiones superior al de un humano.

Un robot móvil es aquél que se puede desplazar libremente en el entorno, por ejemplo con ruedas u orugas motorizadas. Otros robots están fijos en un sitio, por ejemplo un brazo articulado industrial, que puede realizar movimientos pero está unido a una base fija. [11]

Un robot móvil autónomo es aquél que puede navegar entornos desconocidos, para lo cual procesa la información obtenida de sus sensores. Algunas funcionalidades existentes en robots móviles autónomos son identificar objetos y otras características del entorno, planificar rutas que eviten obstáculos, y actuar sobre objetos de interés. Para ello, un robot cuenta con sensores tales como cámaras, sensores de distancia, brújulas, acelerómetros, GPS y sensores de contacto.

Los entornos dinámicos son aquellos que cambian aún sin intervención del agente. Por su parte, un robot situacional es aquél que pueden reconocer y reaccionar rápidamente ante dichos cambios en el entorno. [12] La arquitectura Subsumption encaja con esta definición, dado que los módulos que la componen son reactivos.

2.3. Casos de estudio

En este capitulo se muestran ejemplos de implementaciones de arquitecturas basadas en comportamientos.

2.3.1. LeJOS NXJ

Lego Mindstorms NXT es una plataforma robótica lanzada en 2006, que se utiliza en enseñanza de informática. [13] Cuenta con motores y sensores de luz, color y distancia, como se muestra en la figura 2.4, y ofrece múltiples frameworks de programación.

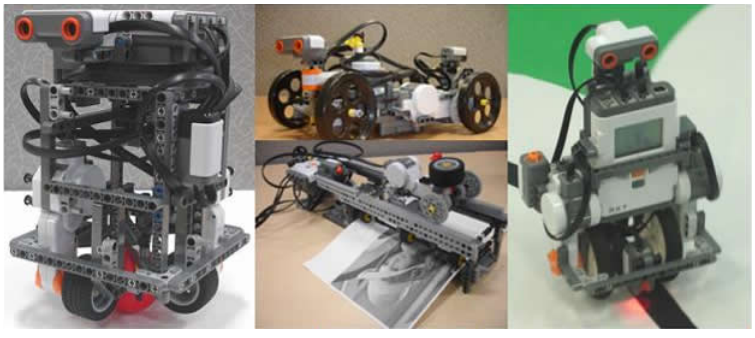

Figura 2.4 Robots Lego Mindstorms NXT

LeJOS NXJ es una máquina virtual Java para Lego Mindstorms NXT, cuya biblioteca *lejos.robotics.subsumption* ofrece una implementación de arquitectura Subsumption. [14]

A diferencia de la propuesta de Brooks, solamente se ejecuta un comportamiento a la vez. Los comportamientos están ordenadas por prioridad, por lo que un comportamiento puede tomar el control del sistema solamente si es de mayor prioridad que el comportamiento activo.

Cada comportamiento debe implementar tres procedimientos de la interfaz de aplicación:

takeControl() Devuelve verdadero si el comportamiento desea tomar el control, y falso si no lo desea. Dicho procedimiento puede utilizar todos los sensores y variables que dispone para determinar el resultado.

action() Comienza a ejecutar cuando el comportamiento toma el control del sistema. El procedimiento puede utilizar los dispositivos para que el robot actúe. Si el procedimiento finaliza su ejecución, el sistema decide qué comportamiento toma el control (puede ser él mismo).

suppress() Se ejecuta cuando el sistema determina que otro comportamiento tomará el control. Típicamente, el procedimiento cambia el valor de una bandera para que el procedimiento *action()* finalice su ejecución rápidamente, salteando bloques de código o saliendo de un bucle.

Mientras se ejecuta un comportamiento, el control del sistema ejecuta periódicamente los procedimientos *takeControl()* de los comportamientos de mayor prioridad, para determinar si alguno de los comportamientos solicita tomar el control.

Si ello ocurre, se ejecuta el procedimiento *suppress()* del comportamiento activo, para indicarle que debe ceder el control. Cuando el procedimiento *action()* del comportamiento activo finaliza su ejecución, el nuevo comportamiento pasa a ser el activo, y se inicia la ejecución de su *action()*.

Si un comportamiento finaliza su ejecución voluntariamente, el sistema ejecuta los *takeControl()* de todos los comportamientos para determinar cuál debe tomar el control.

2.3.2. Robot de Flanagan

Flanagan propuso una implementación de arquitectura Subsumption, que sustituye las capas de competencia de Brooks por un conjunto de procesos que ejecutan tareas. [15] El control

del robot lo realiza un planificador, que toma en cuenta las prioridades y urgencias de todos los procesos.

Los procesos se comunican entre sí leyendo y escribiendo datos en un pizarrón. La estructura ofrece información de interés a los demás procesos, y se gestiona de manera centralizada.

2.3.3. eButiá

eButiá [16] es un entorno de desarrollo de sistemas de control para la plataforma Butiá [17] [18], basado en el programa eToys. Se desarrolló en la Facultad de Ingeniería entre los años 2011 y 2012.

Su arquitectura se basa fuertemente en la propuesta Subsumption. Cada comportamiento puede suprimir o inhibir algún recurso para los comportamientos inferiores por un intervalo de tiempo establecido.

Para ello, cada componente de hardware tiene asociado un módulo de control. Los comportamientos invocan métodos del módulo para usar el componente. El módulo se encarga de ejecutar las acciones de inhibición y supresión.

2.3.4. Yatay

Yatay es un entorno de desarrollo de sistemas de control para la plataforma robótica Butiá [17] [18] desarrollado como proyecto de grado en la Facultad de Ingeniería entre los años 2013 y 2014 [19]. El sistema se utiliza mediante un lenguaje visual de bloques basado en Blockly [20], que sustituye al texto. Esto lo hace especialmente apto para las primeras etapas de enseñanza de informática a niños y adolescentes.

El entorno está basado en el lenguaje HTML5 y ejecuta desde un navegador web. Por tanto, funciona en múltiples plataformas de desarrollo, en particular equipos móviles Android y FirefoxOS.

Cuenta con una arquitectura basada en comportamientos semejante a la de LeJOS NXJ. Por ello, el sistema posee un único comportamiento activo a la vez, y los comportamientos están ordenados por prioridad.

Cada comportamiento de Yatay posee dos procedimientos:

compete_for_active() Evalúa si la capa desea tomar el control del sistema. Si ocurre y su prioridad es mayor que la del comportamiento activo, se asigna como activo.

run() Comienza a ejecutar cuando el comportamiento toma el control del sistema.

Cuando un comportamiento de mayor prioridad se quiere activar, el administrador de comportamientos termina la ejecución del comportamiento activo, e inicia la ejecución del nuevo comportamiento.

2.4. Plataforma de implementación

El sistema de control a ser desarrollado debe tener en cuenta las características de la plataforma sobre la cual se va a construir. Sus restricciones pueden ayudar a lograr las características deseadas de la arquitectura, o por el contrario oponerse a ellas, lo que requeriría de mecanismos para evitar los obstáculos o directamente descartar dichas características. Algunos aspectos importantes a considerar son si utiliza consulta de sensores por polling o por eventos, si soporta tareas concurrentes y la forma de coordinación entre ellas.

El sistema se realizará sobre Toribio y el prototipo sobre la plataforma Butiá. En las siguientes secciones se describen sus características principales.

2.4.1. Butiá

Butiá es una plataforma robótica desarrollada por la Facultad de Ingeniería para que los alumnos de escuelas y liceos públicos de Uruguay aprendan conocimientos básicos de informática, a través de la programación del comportamiento de robots. [17] [18] El proyecto Butiá aprovecha las computadoras XO del proyecto OLPC que se han distribuido en el Plan Ceibal, y pretende fomentar la interdisciplinariedad y la conjugación de teoría y práctica educativa, así como desarrollar la creatividad y la exploración en los niños y adolescentes.

La plataforma comenzó a desarrollarse en setiembre de 2009, y en setiembre de 2010 comenzó a distribuirse en centros educativos. La plataforma es genérica, lo que permite adaptarla a múltiples usos, y tanto el software como el hardware son abiertos, lo que facilita su producción, difusión y modificación, además de fomentar la apropiación de la tecnología y el conocimiento.

La plataforma Butiá carece de un sistema de control propio. A tales efectos, se puede usar sistemas de propósito general, tales como la laptop XO (ver figura 2.5) y las *single board* Raspberry Pi y Foxboard.

Los componentes del hardware son motores, LED y sensores de luz, contacto y distancia. Además cuenta con diversas piezas de construcción (tales como base, ruedas y conectores), que permiten una gran variedad de diseños de robots.

Figura 2.5 Robot Butiá 2.0

2.4.2. Toribio

Toribio [21] es una biblioteca para desarrollar aplicaciones robóticas mediante rutinas, señales y callbacks. [22] Provee mecanismos que facilitan el acceso al hardware y está orientada para sistemas embebidos tales como computadoras *single board*.

Al estar basada en el planificador cooperativo Lumen [23], Toribio permite multitasking basado en corutinas. Esto sirve para que los comportamientos ejecuten concurrentemente, como plantea la propuesta de Brooks. Además Toribió se desarrolló en el lenguaje Lua, por lo que se puede utilizar en múltiples plataformas robóticas.

La arquitectura de Toribio se compone de tres conceptos básicos: los dispositivos, las tareas y la configuración (ver figura 2.6).

Tareas

Las tareas de usuario son las que implementan la lógica de negocio, son compatibles con tareas de Lumen y se comunican entre sí usando el planificador. El planificador es cooperati-

vo, lo que implica que no expropia el procesador. Por tanto, las tareas deben ceder el control del sistema periódicamente, sino las demás tareas no tienen la posibilidad de retomar el control del procesador.

Las tareas tienen acceso a los dispositivos, además pueden emitir señales y quedarse bloqueadas esperando por ellas.

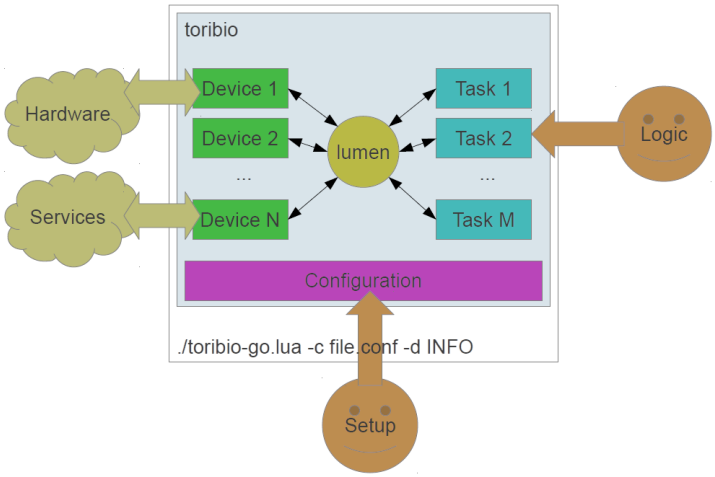

Figura 2.6 Arquitectura de Toribio

Dispositivos y cargadores

Los dispositivos son objetos que representan el hardware o servicios, por ejemplo sensores y motores. Estos proveen una API, y pueden emitir señales.

Los cargadores de dispositivos, o deviceloaders, son tareas de Lumen cuyo propósito es instanciar dispositivos. Toribio tiene incorporados varios cargadores por defecto, en tanto que el programador puede definir sus propios cargadores.

Configuración

La configuración de una instalación Toribio se realiza en el archivo *toribio.conf*. Es un repositorio central y controla el funcionamiento del sistema. En ella se definen las tareas a ser iniciadas, los dispositivos y parámetros de configuración.

2.5. Consideraciones

2.5.1. Arquitectura Subsumption

Brooks establece que los módulos son máquinas de estados sin estructuras de datos, y que la comunicación hacia capas inferiores se realiza mediante los mecanismos de supresión, inhibición y reinicio. Para un usuario principiante, puede que dichas características se alejen de su visión del funcionamiento del robot.

Como consecuencia, es preferible que el sistema permita implementar las funcionalidades de la arquitectura Subsumption, pero que sea más sencilla de entender y más flexible desde la visión del usuario del sistema. Por ejemplo, podría permitirse que cuando los módulos reciben una entrada, se ejecute código definido por el usuario.

2.5.2. Eficiencia de cómputo

Un robot móvil autónomo debe tener un tiempo de respuesta rápido, para poder desempeñarse correctamente en entornos dinámicos. Esto se ve afectado por las restricciones que tiene un robot móvil en cuanto a su capacidad de procesamiento, por razones de costos y espacio. Lo mismo ocurre con el alcance (duración de la fuente de energía); ambos factores obligan a reducir el consumo de energía lo más posible.

Para mitigar dichos problemas, se puede buscar una mayor eficiencia de cómputo. Toda ganancia en dicha área ayuda al buen desempeño del sistema. Una buena arquitectura de control puede aportar a la eficiencia de cómputo.

Los sistemas LeJOS y Yatay utilizan el mecanismo de polling para determinar si cambiar el comportamiento activo. En cada iteración del bucle infinito, se ejecutan los procedimientos que determinan si cada comportamiento desea tomar control del sistema. El código del bucle se ejecuta con gran frecuencia, lo que genera un uso de procesador alto, afectando el tiempo de respuesta y la duración de la fuente de energía.

Una solución a esto es cambiar el mecanismo de consulta por eventos. En vez de que el módulo de un comportamiento ofrezca una función que el árbitro consulte mediante polling, la tarea puede registrarse a un evento provisto por un sensor o a una señal Toribio proveniente de otra tarea.

2.5.3. Concurrencia e interacción entre comportamientos

Los sistemas LeJOS y Yatay ejecutan un único comportamiento a la vez, y el sistema distingue entre ellos por orden de prioridad. Esta característica difiere notablemente de la propuesta Subsumption de Brooks, donde todos los comportamientos ejecutan concurrentemente.

Adoptar la concurrencia de comportamientos requiere determinar los mecanismos de interacción entre ellos. Brooks propone que los comportamientos interactúen mediante mecanismos de supresión, inhibición y reinicio de módulos, como se describió en la sección 3.

Existe una arquitectura alternativa, planteada por Rosenblatt y Payton [8, p. 6], en la que los módulos del sistema devuelvan como resultado un valor real entre -1 y 1. Así, el mecanismo de coordinación de los módulos es la integración de sus valores de salida mediante una suma ponderada de los valores de entrada.

2.5.4. Correctitud y completitud

La correctitud de un sistema es el aseguramiento de que éste responde ante cada situación estudiada según lo especificado. En tanto, la completitud de un sistema es el aseguramiento de que el comportamiento de éste está definido para cada situación posible.

La posibilidad de demostrar la correctitud y la completitud de un sistema es muy valiosa. En particular, es imprescindible para contextos críticos donde el sistema debe actuar efectivamente en todo tipo de situaciones. Es el caso de situaciones con peligro de vida, y de sistemas altamente costosos.

Una línea de estudio posible es analizar si se puede realizar una arquitectura de control que permita demostrar la correctitud y completitud de un sistema, o bien que cuente con características que ayuden a lograr dichas cualidades.

2.5.5. Entornos multiagente

Los robots autónomos operan a menudo en entornos multiagente, es decir en entornos donde hay múltiples agentes que intentan cumplir sus objetivos cada uno, sean compartidos, opuestos o simplemente distintos.

La interacción entre agentes se puede realizar de distintas maneras.

- **Percepción** El agente utiliza sus sensores para percibir el comportamiento de los demás agentes. Un ejemplo de esto es el sumo robótico.
- **A través del entorno**: El agente coloca rastros en el entorno (marcas, signos, objetos) para que sean percibidos por los demás agentes.
- **Comunicación**: Los agentes emiten información directamente a los demás agentes, por ejemplo a través de una red inalámbrica.

La arquitectura de control se ve involucrada especialmente en la comunicación entre agentes. El diseño del sistema podría orientarse a dicho enfoque, para facilitar el desarrollo de robots para entornos multiagente.

2.5.6. Tiempo real

Según el problema que se quiere resolver, puede ser necesario establecer restricciones a los tiempos de respuestas del sistema. Un sistema de este tipo se lo conoce como de tiempo real. [24, p. 28] Estos se aplican en situaciones donde al ocurrir demoras mayores a las previstas se generan consecuencias indeseables, o bien la utilidad de la respuesta se degrada rápidamente con el paso del tiempo. Por ejemplo, una demora excesiva en un sistema crítico puede tener consecuencias catastróficas, poniendo en riesgo la vida de personas o bienes materiales.

Comúnmente las restricciones sobre el tiempo de respuesta se dividen en dos tipos, blandas o duras. Según Liu [24, p. 28], un sistema de tiempo real es duro si garantiza que el tiempo de respuesta cumple la restricción en toda circunstancia, y blando si el tiempo de respuesta se garantiza de manera probabilística para un cierto nivel de confianza. La decisión sobre si elegir un sistema duro o blando depende de las consecuencias de un eventual incumplimiento.

Un sistema de tiempo real se implementa a través de la administración de los recursos del sistema. Para lograrlo se requiere que el hardware y software con el que se lo construye ofrezcan las funcionalidades necesarias. Por ejemplo, un sistema operativo expropiador permite detener tareas anticipadamente, lo que puede ser útil. El planificador de tareas puede considerar el tiempo de ejecución estimado de cada tarea, para realizar un plan que cumpla con las restricciones de tiempo.

Para que un robot móvil autónomo se desempeñe adecuadamente en entornos dinámicos, es importante que pueda responder rápidamente. Por ejemplo, si un robot no detecta un objeto con suficiente antelación, puede colisionar con él y averiarse, lo que puede afectar gravemente la tarea que debía cumplir. Al desarrollar un sistema de control de tiempo real, se permitiría garantizar el tiempo de respuesta del robot.

Sin embargo, las herramientas sobre las que se basa la biblioteca Toribio -tales como Lumen, Lua y Linux- no ofrecen funcionalidades que permitan realizar un sistema de tiempo real. Por ello, es poco factible intentar lograr dicho requisito en el producto.

2.5.7. Facilidad de uso

Es deseable que la interfaz de usuario del sistema sea fácil de aprender, entender y usar, y que su estética y uso sean agradables. Esto colabora para que los usuarios puedan usar el sistema con efectividad, enfocándose en la tarea de programación y no en la interfaz en sí. A su vez, facilita la tarea de corregir errores de programación.

La plataforma Butiá se utiliza en centros educativos por parte de niños y adolescentes, que por lo general recién empiezan a aprender informática. Del mismo modo, puede que buena parte de los profesores que enseñan a usar el robot sean poco expertos en programación. Por ello, para que el sistema se use con éxito en el marco de un proyecto educativo, es esencial contar con una buena interfaz de usuario.

Capítulo 3

Requisitos del sistema

Este capítulo presenta una visión general de la especificación de requerimientos del sistema de control Torocó. La sección 3.1 del documento presenta un panorama general de los requisitos del sistema de control, describiendo sus funcionalidades, usuarios e interfaces.

En la sección 3.2 se profundiza la especificación de los requisitos funcionales, tales como arquitectura basada en comportamientos, comunicación entre comportamientos y comunicación con hardware. Los requisitos no funcionales del sistema de control se detallan en la sección 3.3, abarcando aspectos como el paralelismo de comportamientos, sistema operativo, lenguaje de programación y hardware soportado.

3.1. Descripción general

3.1.1. Objetivos

El sistema de control de robots Torocó permitirá que el usuario defina los comportamientos que ejecutará el robot. Cada comportamiento se encargará de realizar una tarea, como puede ser avanzar hacia adelante, orientarse hacia un sitio, evitar obstáculos, levantar objetos, etc. Los comportamientos podrán comunicarse con los sensores y actuadores del robot para cumplir con su tarea.

Los comportamientos ejecutarán concurrentemente, lo que implica que podrán recibir, procesar y enviar datos en todo momento. Los comportamientos estarán ordenados verticalmente en capas.

El sistema brindará mecanismos de comunicación entre los comportamientos para que el robot cumpla con su función. Además, los comportamientos podrán acceder a sistemas remotos para intercambiar datos.

3.1.2. Usuarios

El usuario del sistema será un programador que pretende desarrollar robots para resolver problemas de su interés. Tendrá conocimientos de programación estructurada, programación concurrente y robótica basada en comportamientos.

3.1.3. Interfaces

El sistema proveerá al usuario una interfaz de aplicación (API), de modo que se utilizará como una biblioteca. El desarrollo de un entorno de desarrollo queda fuera del alcance del proyecto.

3.2. Requerimientos funcionales

3.2.1. Comportamientos

La arquitectura del sistema de control es basada en comportamientos. Un comportamiento es una entidad que puede almacenar y procesar datos, así como comunicarse con dispositivos de hardware u otros comportamientos. Los comportamientos del sistema ejecutarán concurrentemente.

3.2.2. Comunicación con dispositivos

Los comportamientos podrán comunicarse con los dispositivos de hardware, para obtener datos de los sensores y operar en el entorno mediante los actuadores.

3.2.3. Comunicación entre comportamientos

Los comportamientos podrán comunicarse entre sí para intercambiar datos.

3.2.4. Comunicación con sistemas remotos

El sistema podrá comunicarse con sistemas remotos mediante los protocolos de red TCP o UDP. Los comportamientos podrán utilizar dicho canal de comunicación para enviar y recibir datos.

3.2.5. Comunicación por eventos

Los comportamientos podrán registrarse a eventos, por ejemplo cambios de los valores de un sensor. Para ello, el comportamiento debería indicar el nombre del evento y la función de callback. Cuando el emisor envía una señal de evento, los comportamientos suscritos al evento capturan la señal y reaccionan ejecutando la función de callback.

Además, los comportamientos podrán detener su ejecución para esperar a que ocurra un evento. Para ello, el comportamiento deberá indicar el nombre del evento. Cuando el comportamiento recibe el evento, se reanuda su ejecución.

Por su parte, los comportamientos podrán emitir señales ante eventos que ocurran dentro de él. Para ello, deberá indicar el nombre del evento y los datos adjuntos.

La comunicación por eventos permite evitar las consultas por polling, donde se consulta una función insistentemente hasta obtener el resultado deseado. El mecanismo de polling es menos eficiente y requiere más recursos computacionales que la comunicación por eventos.

La comunicación por eventos también sirve para preprocesar información proveniente de un sensor. Un ejemplo de utilización sería un sensor que emite valores cada 5 milisegundos. El usuario podría definir un comportamiento que capture todas las señales del sensor y emita cada 100 milisegundos una señal representativa, por ejemplo un promedio de los valores obtenidos.

Así, los demás comportamientos reaccionarían ante el evento del intermediario, que tiene una tasa de refresco 20 veces menor que la del sensor, lo que reduciría la carga computacional del sistema.

3.2.6. Supresión e inhibición

El sistema ofrecerá mecanismos de inhibición y supresión de comportamientos, en base a la arquitectura Subsumption de Brooks. Esto significa que un comportamiento de una capa superior puede intervenir en los canales de comunicación de capas inferiores.

La inhibición consiste en anular un evento de salida de un comportamiento, es decir que los receptores no recibirán el evento. En tanto, la supresión consiste en anular los datos de entrada de un comportamiento, y sustituirlos por otros datos provistos por el comportamiento supresor.

Los mecanismos de inhibición y supresión se podrán activar durante cierto período de tiempo establecido por el comportamiento invocador, o bien hasta que éste envíe una señal de rehabilitación. Para que un comportamiento pueda suprimir o inhibir a otro, el invocador deberá tener mayor jerarquía que el otro.

3.3. Requerimientos no funcionales

3.3.1. Concurrencia

Los comportamientos deberán ejecutar en forma concurrente. Esto se implementará mediante tareas Toribio, las cuales incorporan la concurrencia cooperativa de Lumen.

3.3.2. Sistema operativo

El sistema deberá ejecutar sobre Linux.

3.3.3. Entorno

El sistema se implementará sobre el framework Toribio en el lenguaje Lua, versión 5.2.3.

3.3.4. Soporte para Usb4Butiá

El sistema funcionará con los sensores de Usb4Butiá, que están basados en polling.

3.3.5. Hardware mínimo

El robot de prueba utilizará como placa de control una *single board* de propósito general, de al menos 64 MB de RAM.

Capítulo 4

Arquitectura del sistema

Este capítulo especifica la arquitectura del sistema de control Torocó. En la sección 4.1 se explican las decisiones de arquitectura. La sección 4.2 describe la arquitectura del sistema, incluyendo los comportamientos, sensores, actuadores y eventos.

En la sección 4.3 se presentan las funcionalidades que ofrece la biblioteca. La sección 4.4 describe la interfaz de aplicación. En la sección 4.5 se profundiza en el diseño de los componentes del sistema.

4.1. Decisiones

4.1.1. Tipo de interfaz

Existen varios tipos de interfaz de sistemas informáticos. Una de ellas es la biblioteca, que ofrece un conjunto de funcionalidades que el programador puede utilizar cuando desee en su programa. Para ello, debe invocar la función deseada de la biblioteca, ingresando parámetros de entrada según la especificación de la interfaz. Es decir que el usuario se mantiene en control del flujo de ejecución.

Otro tipo de interfaz es el framework, en el que se ofrecen funcionalidades genéricas, que el programador debe completar agregando estructuras. Así, el framework invoca a dichas estructuras durante la ejecución del sistema; a esto se lo conoce como inversión del control. El usuario puede extender el framework solamente mediante la interfaz que se ofrece.

Se decidió implementar Torocó como una biblioteca, de manera de ofrecer la mayor libertad al programador del sistema. El objetivo fue interferir lo menos posible en el proceso de desarrollo del programador.

Además, para usar Torocó no se necesita invocar a las bibliotecas Toribio ni Lumen, lo que simplifica el uso. A su vez, el usuario puede crear tareas Toribio para que ejecuten en paralelo a los comportamientos de Torocó.

4.1.2. Comunicación por eventos

Torocó ofrece un sistema de comunicación de tipo push. En ella, cada emisor envía sus eventos de salida cuando lo desea, tras lo cual los receptores que estén registrados reciben el evento y procesan los datos adjuntos. Un receptor puede detenerse a esperar a que ocurra un evento, pero no lo contrario: el emisor nunca se entera de quiénes son los receptores ni si el evento se recibió o se perdió.

Esto difiere de la comunicación de tipo pull, en la que el emisor dispone de los datos a emitir, y los envía cada vez que un receptor los solicita.

Se optó por la comunicación de tipo push porque se asemeja a la arquitectura Subsumption, en la que los módulos funcionan independientemente entre sí. Cada uno recibe los datos que llegan a la entrada, los procesa y envía los datos de salida, desconociendo quién les envía datos de entrada y quién recibe los datos de salida. Del mismo modo, si ningún modulo recibe una señal emitida, el sistema descarta el dato y continúa funcionando.

Además, en un robot móvil autónomo los datos de los sensores generalmente cambian a una tasa menor a la que los comportamientos pueden procesarlos. Por ello, la comunicación de tipo pull tiene desventajas en este contexto. Cuando un comportamiento desea recibir un dato, tiene que realizar solicitudes constantemente hasta recibir un dato. Esto se conoce como polling, y genera un alto consumo de procesador.

En cambio, la comunicación de tipo push permite que los receptores dejen de ejecutar mientras esperan a recibir un dato. De este modo, cuando se envía el evento, cada receptor reanuda su ejecución inmediatamente. Esto es notoriamente más eficiente que el mecanismo de polling.

4.1.3. Entradas y salidas múltiples

En la arquitectura Subsumption, un módulo puede tener varios eventos de entrada, y combinar sus datos para emitir la salida. Sin embargo, cada módulo debe realizar una única tarea, por lo que es deseable que el sistema evite que el usuario utilice un módulo para ejecutar varias tareas independientes entre sí.

Por tanto, se decidió que en cada comportamiento se puede conectar un evento a una sola entrada, y las entradas son idénticas para todas las corutinas del comportamiento. Esto impide que un usuario intente suprimir un evento para una corutina del comportamiento y dejar habilitado el mismo evento para otra, lo cual violaría la unicidad del comportamiento (ver figura 4.1).

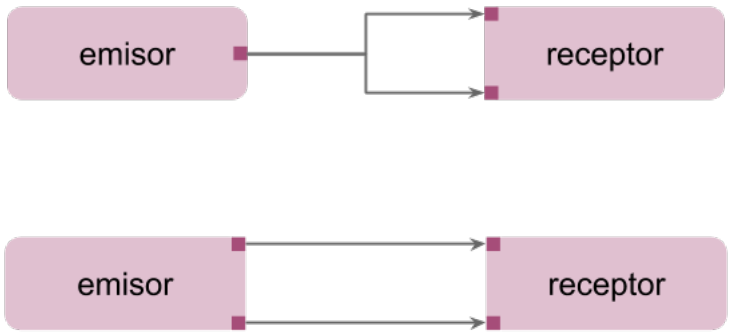

Figura 4.1 Ejemplo de entradas múltiples a un comportamiento.
En el diagrama superior, un mismo evento está conectado a dos entradas de un comportamiento. El sistema no permite suprimir una entrada y dejar habilitada la otra.
En el diagrama inferior, dos eventos están conectados a dos entradas de un comportamiento. Ambas entradas se inhiben independientemente entre sí.

Del mismo modo, la función de emitir las salidas del comportamiento envía todas simultáneamente, sin dar la posibilidad de emitir un subconjunto de ellas. Se puede liberar las inhibiciones y supresiones de ciertos eventos de manera individual, pero esto puede desencadenar otras acciones, por ejemplo que otros comportamientos reenvíen sus eventos.

4.1.4. Inhibición y supresión

La arquitectura Subsumption permite que los comportamientos inhiban salidas de datos y supriman entradas de otros comportamientos. Esto se podría ofrecer en funciones de inhibir y suprimir, para que las invoquen los comportamientos. Sin embargo, el prototipo implementado por Brooks utiliza otro mecanismo: los eventos emiten salidas genéricas, y al definir la interconexión entre comportamientos se indica que las salidas de un comportamiento sirve para inhibir o suprimir a otro.

Se optó por esta segunda opción, lo que implica que el código de comportamiento no indica qué otros comportamientos debe inhibir o suprimir, sino que ello se determina en el programa principal al definir la interconexión entre comportamientos. Esto permite modificar qué comportamientos se inhiben y suprimen sin tener que reprogramarlos.

4.1.5. Interconexión entre comportamientos

La interfaz de interconexión entre comportamientos puede llegar a ser muy compleja, si se intenta permitir todo tipo de conexiones. Se prefirió tomar decisiones que tiendan a simplificar el sistema, reduciendo la variedad de conexiones posibles.

Cada entrada de un comportamiento se conecta a un conjunto de eventos de entrada, ordenados por prioridad. Es decir, los eventos se conectan con módulos supresores en serie, sin permitir otras formas (ver figura 4.2). Por su parte, en cada salida de un comportamiento, las inhibiciones también se conectan en serie, por lo que una salida queda inhibida si ocurrió alguno de los eventos inhibidores (ver figura 4.3).

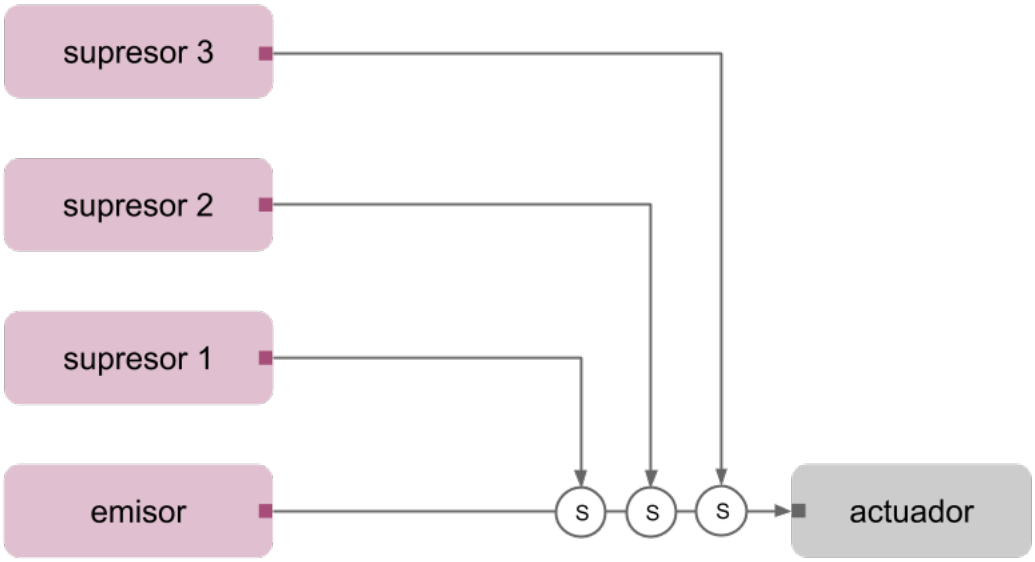

Figura 4.2 Ejemplo de supresión de eventos en serie.

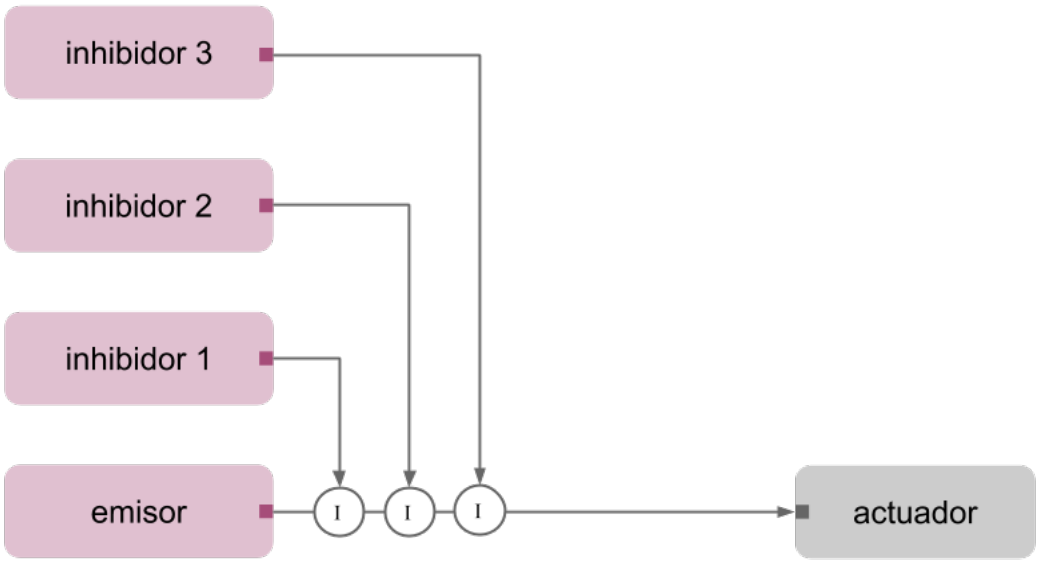

Figura 4.3 Ejemplo de inhibición de eventos en serie.

4.1.6. Eventos fijos

Al desarrollar el robot de prueba, hubo dificultades para que los comportamientos fueran simples y efectivos. En concreto, cuando un comportamiento debe dejar de indicarle a los

motores que giren hacia un lado, el sistema debía reaccionar haciendo que el robot volviera a avanzar hacia adelante.

Para lograrlo se estudiaron varias opciones: que el comportamiento de avanzar emitiera su salida periódicamente; que los comportamientos de girar emitieran una salida de avanzar al desactivarse; que los comportamientos de girar emitieran una salida hacia el comportamiento de avanzar para indicarle que vuelva emitir su salida, entre otras. En todos los casos, se consideró que la opción era poco clara o efectiva.

Por tanto, se decidió que Torocó permitiera al usuario fijar eventos de salida. Así, el seguidor de líneas tiene un comportamiento de avanzar con una corutina que fija su salida hacia los motores para que el robot avance. Los comportamientos de girar suprimen dicho evento cuando sus sensores detectan negro, y dejan de suprimirlo cuando detectan blanco. Como el comportamiento de avanzar tiene salida fija, cuando deja de estar suprimida, Torocó reenvía el evento de avanzar a los motores.

El manejo de eventos fijos es independiente del tiempo de inhibición y supresión. En otras palabras, al emitir un evento hacia un receptor, los eventos de menor prioridad de dicha entrada quedan suprimidos durante el tiempo especificado, aún cuando dicho evento no es fijo. Del mismo modo, al emitir un evento que inhibe la salida de otro emisor, dicha inhibición tiene la duración especificada. Las inhibiciones y supresiones se levantan al emitir el evento de liberar, o al ejecutar la función de liberar evento fijo.

4.1.7. Triggers y corutinas

En la solución realizada, los comportamientos se pueden implementar mediante triggers y corutinas. En el primer caso, cada vez que el comportamiento recibe la entrada, se ejecuta la función handler del trigger. En el segundo caso, al iniciar la ejecución del sistema se invoca a una función.

Los triggers se pueden implementar también con corutinas, aunque el resultado es ligeramente distinto. Se considera que dicha herramienta sirve para facilitarle el trabajo al usuario del sistema. Por otra parte, se entendió que ofrecer solamente triggers quitaría libertad al programador, y restringiría la funcionalidad del sistema.

Para que las corutinas puedan utilizar las entradas, se ofrece la función *wait_for_input()*, que detiene la ejecución de la corutina para esperar a que el comportamiento reciba una entrada.

La función también se puede usar en triggers, lo cual nuevamente aumenta la flexibilidad del sistema.

4.2. Arquitectura del sistema

Como se detalla en el capítulo 2, la arquitectura Subsumption consiste en una serie de comportamientos que ejecutan en paralelo. Cada comportamiento se encarga de realizar una tarea, y puede comunicarse con los sensores y actuadores del robot para cumplirla.

Torocó permite definir los comportamientos del sistema, y permite intercambiar eventos entre los comportamientos, sensores y actuadores. Las funcionalidades de la biblioteca Torocó serán invocadas por los comportamientos que defina el desarrollador del robot.

4.2.1. Interacción del sistema

El usuario del sistema de control es el desarrollador del robot. Sin embargo, el sistema de control no interactúa con el desarrollador, sino que el robot se desempeña de manera autónoma en un entorno, interactuando con los objetos, otros robots y sistemas remotos. Por tanto, el sistema responde a los siguientes agentes externos:

- **Reloj**: El sistema de control puede responder a un evento lanzado por un reloj, sea de manera periódica o individual.
- **Sensor**: El sistema de control puede reaccionar ante un dato proveniente de un sensor, como puede ser un sensor de distancia, acelerómetro o cámara.
- **Sistema remoto**: El sistema de control puede recibir una comunicación proveniente de un sistema remoto, a través de un socket TCP o UDP.

4.2.2. Componentes

El sistema de control consiste en un conjunto de comportamientos, sensores y actuadores que se comunican entre sí intercambiando eventos, como se muestra en la figura 4.4. Cada comportamiento contiene un conjunto de corutinas que ejecutan en paralelo.

Torocó se encarga de gestionar la interconexión de eventos y los mecanismos de inhibición y supresión. En tanto, Toribio ofrece un conjunto de funcionalidades que permiten que el usuario se comunique con el hardware. Por otra parte, Toribio y Torocó usan la biblioteca Lumen, que coordina la ejecución de tareas e implementa la comunicación por eventos.

Para realizar un sistema de control mediante Torocó, el usuario debe implementar los comportamientos del robot, y un programa principal que se encargue de configurar la comunicación entre los componentes e iniciar la ejecución del sistema. Los comportamientos del robot se realizan mediante las funciones que ofrece la biblioteca Torocó, sin necesidad de acceder directamente a Toribio ni Lumen.

El usuario debe definir además el archivo de configuración de Toribio (por defecto es *toribio.conf*), donde se indican los devices que utilizará el sistema.

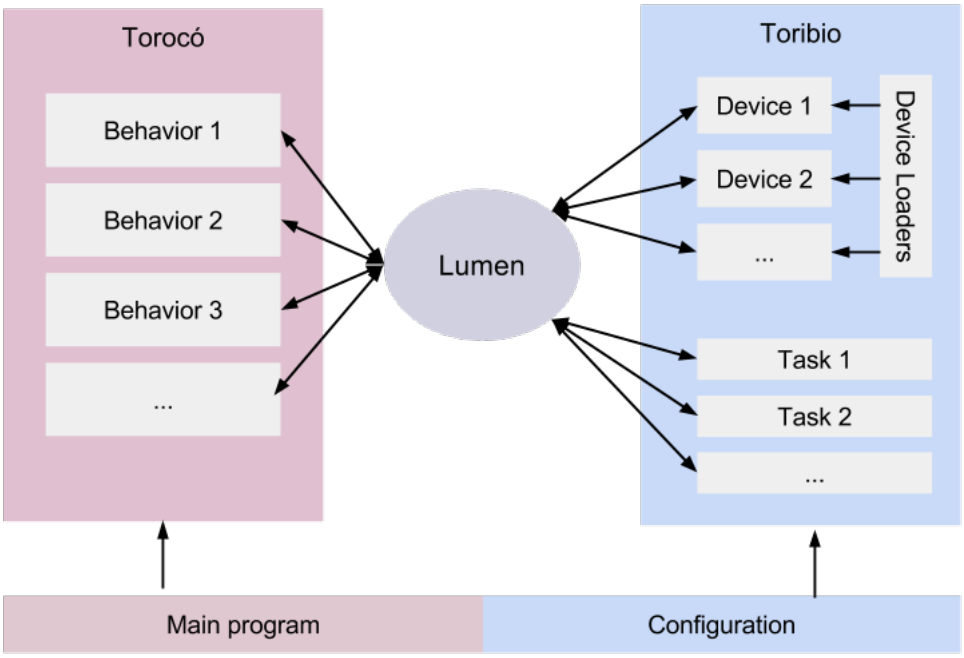

Figura 4.4 Diagrama de componentes del sistema de control

4.2.3. Comunicación por eventos

La comunicación entre comportamientos, sensores y actuadores se realiza mediante eventos. La figura 4.5 muestra un ejemplo de un sistema de control de robot implementado mediante Torocó.

Cada comportamiento define una serie de entradas y salidas. Cada entrada puede registrarse a uno o más eventos de salida de otros comportamientos o sensores. Al recibir una entrada, el comportamiento obtiene además los datos adjuntos al evento.

Por su parte, los comportamientos pueden emitir sus eventos de salida, para que los reciban los receptores que se hayan registrado. Existen dos formas de emitir los eventos de salida: emitir el evento una única vez o fijarlo de manera permanente. En el segundo caso, si dicha salida es suprimida o inhibida y luego se levanta la acción, los receptores vuelven a recibir dicho evento.

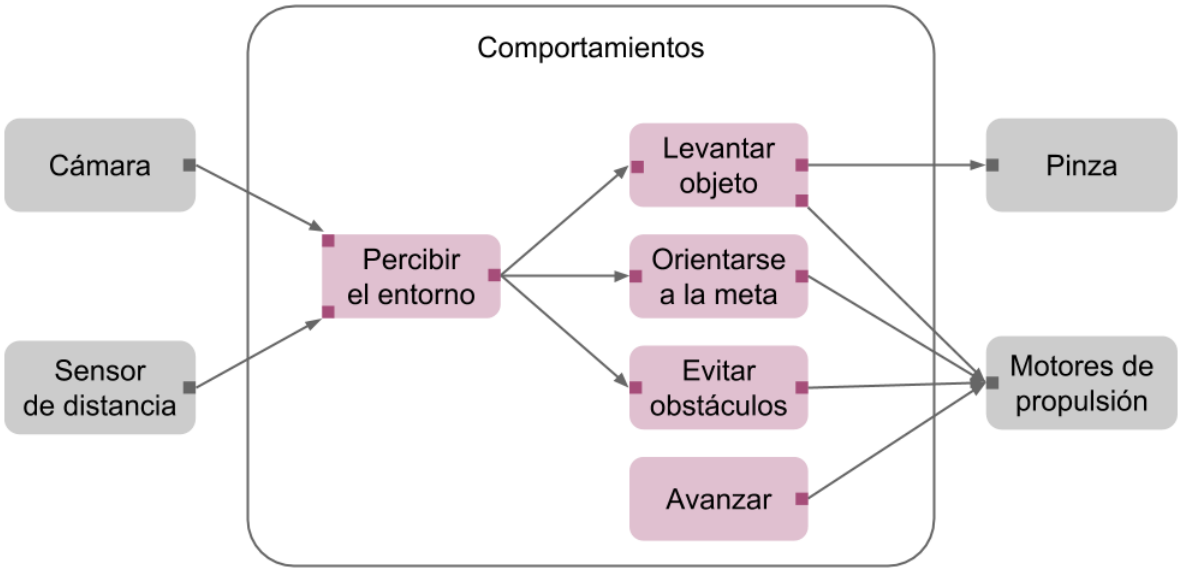

Figura 4.5 Ejemplo de diagrama de comportamientos de un robot recolector.
Las flechas representan eventos y los cuadrados representan entradas.
Véase que la entrada del motor recibe varios eventos; Levantar objeto tiene varias salidas; y Percibir el entorno tiene varias entradas y una salida que emite a varios receptores.

Los sensores pueden definir eventos de salida y funciones, para que las entradas de los comportamientos se conecten a ellos. Del mismo modo, los actuadores pueden definir eventos de entrada y funciones, para que las salidas de los comportamientos se conecten a ellos.

Torocó usa los eventos y funciones de los devices de manera unificada, por lo que los comportamientos pueden usar ambos mecanismos de manera indistinta. Más precisamente, para cada función de los sensores, el sistema genera una tarea que se encarga de hacer polling. De este modo, cuando la función devuelve un valor nuevo, se envía un evento con dicho valor a los receptores. Esto es menos eficiente que usar eventos, pero permite compatibilidad

con devices que no funcionan por eventos. En tanto, para cada función de los actuadores, el sistema genera una tarea que captura los eventos emitidos al actuador e invoca a la función con dichos valores como parámetros.

El diagrama 4.6 muestra la propagación de eventos en el sistema. El despachador se encarga de controlar la comunicación por eventos. Cuando un comportamiento emite su salida, el despachador procesa cada evento para enviarlo a todos los receptores que están registrados, aplicando los mecanismos de inhibición y supresión.

Por cada evento de salida que se emite, el despachador verifica si el emisor está habilitado para enviarlo. Luego se recorre la lista de receptores registrados a cada evento habilitado. Por cada receptor, el despachador verifica si el receptor está habilitado para recibir el evento, y en caso afirmativo le reenvía el evento.

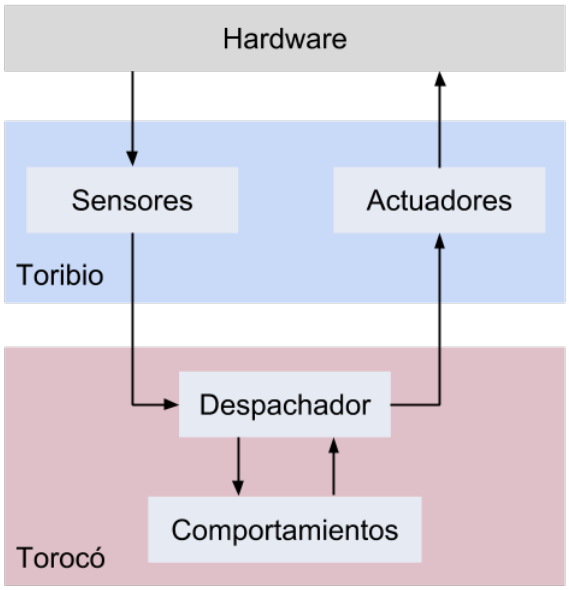

Figura 4.6 Diagrama de flujo de eventos del sistema

4.2.4. Inhibición y supresión

Los comportamientos pueden intervenir la comunicación de capas inferiores, en base a la arquitectura Subsumption de Brooks. De este modo, un comportamiento puede inhibir eventos de salida o suprimir eventos de entrada de capas inferiores.

La inhibición consiste en anular un evento de salida de un emisor, es decir que los receptores no recibirán el evento. Para cada salida de un comportamiento o device, se puede especificar

un conjunto de eventos supresores. Cuando se emite alguno de dichos eventos, la salida queda suprimida.

En tanto, la supresión consiste en anular los datos de entrada de un receptor, y sustituirlos por otros datos provistos por el comportamiento supresor. Como se dijo anteriormente, una entrada de un receptor puede registrarse a varios eventos. Cuando el receptor recibe un evento para dicha entrada, los eventos de menor prioridad de la entrada quedan suprimidos.

Los mecanismos de inhibición y supresión se podrán activar durante cierto período de tiempo establecido por el comportamiento invocador, o bien hasta que éste envíe un evento de liberar.

En la sección 4.4.2. se describe el diseño del despachador.

4.3. Funcionalidades

En la figura 4.7 se muestran las funcionalidades que ofrece la biblioteca Torocó.

Figura 4.7 Funcionalidades del sistema

4.3.1. Iniciar sistema

El programa principal del sistema de control debe indicar los comportamientos a ejecutar, así como la interconexión entre entradas y salidas de comportamientos, sensores y actuadores.

Luego, el programa debe iniciar la ejecución del sistema de control.

4.3.2. Crear comportamiento

El sistema crea un comportamiento nuevo, sea desde archivo o manualmente. El comportamiento consiste en un conjunto de corutinas.

Los triggers son un tipo de corutina que asocia una entrada con una función handler. De este modo, cada vez que el comportamiento recibe la entrada, se ejecuta la función handler.

4.3.3. Enviar eventos

Las corutinas y triggers pueden emitir la salida del comportamiento, y el sistema los reenvía aplicando los mecanismos de inhibición y supresión.

El despachador verifica si el emisor está habilitado para enviar los eventos. Por cada evento habilitado, el sistema recorre la lista de comportamientos registrados a dichos eventos. Para cada uno, verifica si el receptor está habilitado para recibir el evento, y en caso afirmativo le reenvía el evento.

4.3.4. Esperar por un evento

Las corutinas y triggers pueden esperar por una entrada del comportamiento. Cuando el comportamiento recibe la entrada, la corutina recibe los datos del evento y se reanuda la ejecución.

4.3.5. Inhibir evento de un emisor

Un comportamiento le indica al sistema que se inhiba un evento de salida de otro comportamiento o device. Esto significa que si el emisor intenta enviar el evento, el sistema no lo reenviará a los receptores.

El comportamiento debe indicar el evento a inhibir y un período de tiempo de inhibición. Opcionalmente se puede indicar que la inhibición ocurra por tiempo indefinido.

El sistema comprueba que el comportamiento que solicita la inhibición sea de mayor jerarquía que el emisor. En caso negativo, devuelve un resultado de error.

4.3.6. Suprimir evento para un receptor

Un comportamiento le indica al sistema que se suprima un evento de entrada de otro comportamiento o device. Esto significa que si se envía el evento, el sistema no lo reenviará a dicho receptor.

El comportamiento debe indicar el receptor a suprimir, el evento a suprimir, y un período de tiempo de la supresión. Opcionalmente se puede indicar que la supresión ocurra por tiempo indefinido.

El comportamiento también podrá indicar un dato adjunto, para que el sistema envíe el evento al receptor indicado con dicho dato. Esto permitirá implementar la supresión de la arquitectura Subsumption.

El sistema comprueba que el comportamiento que solicita la supresión sea de mayor jerarquía que los receptores indicados. En caso negativo, devuelve un resultado de error.

4.3.7. Detener inhibición

Un comportamiento le indica al sistema que se detenga la inhibición de un evento. Esto significa que si todos los comportamientos detienen dicha inhibición, el sistema volverá a reenviar el evento cuando éste ocurra.

Si el comportamiento no es de mayor jerarquía que el emisor, el sistema devuelve un resultado de error.

4.3.8. Detener supresión

Un comportamiento le indica al sistema que se detenga la supresión de un receptor para un evento. Esto significa que si todos los comportamientos detienen dicha supresión, el sistema volverá a reenviar el evento a dicho receptor cuando éste ocurra.

Si el comportamiento no es de mayor jerarquía que los receptores indicados, el sistema devuelve un resultado de error.

4.4. Diseño

4.4.1. Comportamientos

Las corutinas pueden emitir los eventos de salida del comportamiento, para que los reciban las corutinas de otros comportamientos y los actuadores (ver figura 4.8). La tabla *events* contiene a los eventos de salida de cada comportamiento.

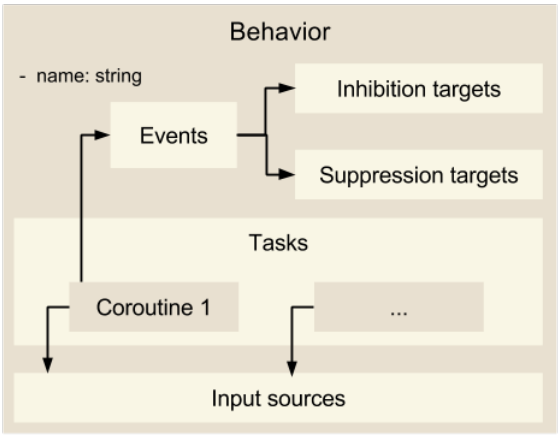

Figura 4.8 Diagrama de flujo de un comportamiento

Cuando se emite un evento de salida, se realizan las inhibiciones y supresiones que correspondan según las tablas *inhibition_targets* y *suppression_targets*.

Además, las corutinas de los comportamientos pueden detener su ejecución y quedar esperando por una entrada, es decir eventos provenientes de otros comportamientos o sensores. La tabla *input_sources* especifica cuáles son los eventos a los que están conectadas cada una de las entradas del comportamiento.

Eventos activos

Cuando un comportamiento emite sus eventos de salida usando *set_output*, estos quedan marcados como activos en la tabla *active_events*.

Por otro lado, cada evento tiene a su vez asociado un evento de liberar, que se emite cuando éste deja de ser inhibido o suprimido. En el caso de la supresión se indica también quién es el receptor que debe volver a recibir el evento; en el caso de la inhibición se dirige a todos los receptores.

Cuando un comportamiento registra sus eventos de entrada, se registra a los eventos de liberar asociados a éstos. Por tanto, cuando alguien emite el evento de liberar, si el evento está activo y está dirigido al comportamiento en cuestión, se vuelve a emitir el evento.

4.4.2. Despachador

El despachador se encarga de controlar la comunicación por eventos. Cada vez que se emite un evento de salida, el despachador se ejecuta para enviar el evento a todos los receptores que están registrados, aplicando los mecanismos de inhibición y supresión.

El despachador está implementado en la función *dispatch_signal*. Todos los eventos de salida del sistema se registran a la función del despachador. Por tanto, cuando un comportamiento emita sus eventos de salida, el despachador se ejecutará para cada uno de los eventos.

Cada entrada de un receptor tiene asociado un evento alias, cuyo funcionamiento se muestra en la figura 4.9. Al conectar un evento de otro comportamiento o sensor a dicha entrada, se registra el evento alias al evento emisor. Cuando se emita el evento alias, el receptor lo recibirá en dicha entrada.

Por tanto, cuando un comportamiento emite su salida, el despachador procesa cada uno de los eventos. Por cada evento emitido, el despachador recorre cada receptor registrado al evento. Si el receptor no está inhibido, el despachador emite el evento alias de dicha entrada.

Durante su ejecución, el despachador primero actualiza las inhibiciones para el evento. Luego, si el evento no está inhibido, el despachador procesa cada receptor registrado al evento. Primero actualiza sus supresiones y si luego comprueba se está suprimido. Si no lo está, el despachador encola el evento alias del receptor para ser emitida al finalizar la ejecución del despachador. Los receptores, es decir las funciones que están bloqueadas esperando por el evento alias, se despiertan al recibir el evento alias, y reanudan su ejecución.

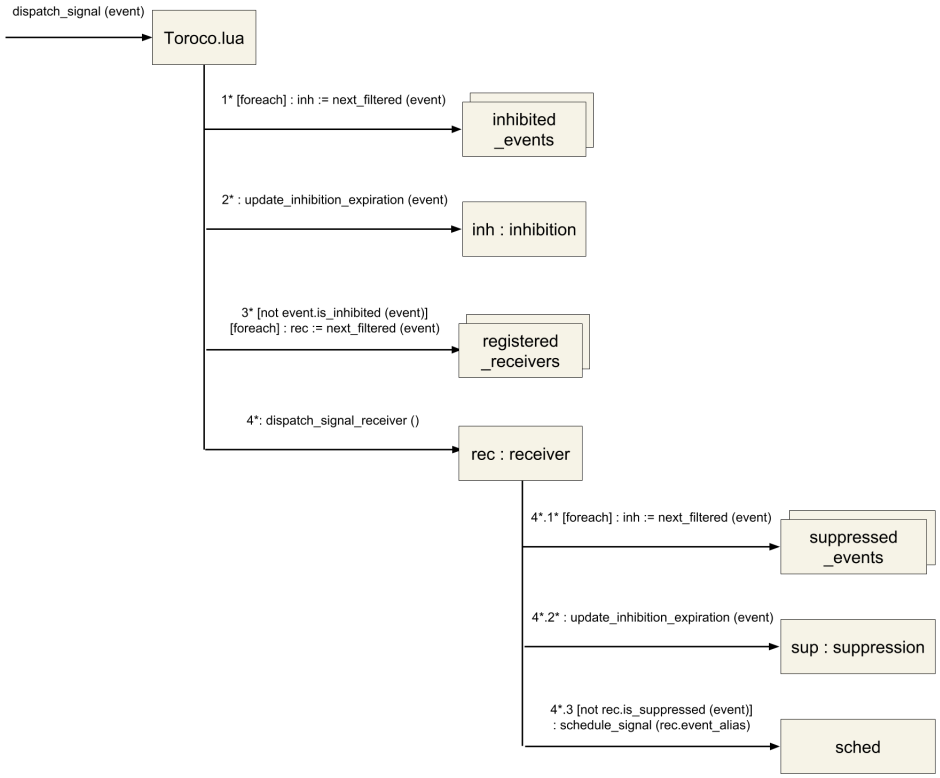

Figura 4.9 Diagrama de comunicación de dispatch_signal()

Capítulo 5

Pruebas del sistema

Este capítulo presenta las pruebas realizadas con el sistema de control, en las que se intenta mostrar sus características y verificar su funcionamiento.

En la sección 5.1 del capítulo se detalla el hardware del robot de prueba. La sección 5.2 describe el Torobot, que es una demostración de robot móvil autónomo. En la sección 5.3 se muestran los resultados de las pruebas de verificación del sistema de control. Finalmente, las pruebas de desempeño del sistema se presentan en la sección 5.4.

5.1. Hardware

Todas las pruebas fueron realizadas con el mismo hardware de base. Las especificaciones de la placa de control donde ejecuta el sistema de control se muestran en el cuadro 5.1.

Modelo	Beaglebone Black revisión A6A
Procesador	Sitara AM3359AZCZ100, 1 GHz, 2000 MIPS
Memoria RAM	512 MB DDR3L
Memoria interna	eMMC 2 GB
Memoria externa	Micro SD 8 GB

Cuadro 5.1 Especificaciones de la placa de control

Como se muestra en la figura 5.1, el robot posee dos ruedas traseras de tracción y una rueda loca en la parte delantera. Los dos motores de propulsión son servos Springrc SM-S3317SR.

El robot tiene una cámara web Logitech apuntando hacia el frente. Debido a que el campo de visión es reducido, se colocó la cámara en la posición lo más retrasada posible. También cuenta con dos sensores de distancia infrarrojos marca Sharp de rango corto, colocados al frente del robot apuntando ligeramente hacia los lados.

Para el Torobot se utilizó un router TP-Link en el robot, para que la computadora remota pueda conectarse a él. Por último, el robot tiene un botón de encendido de motores.

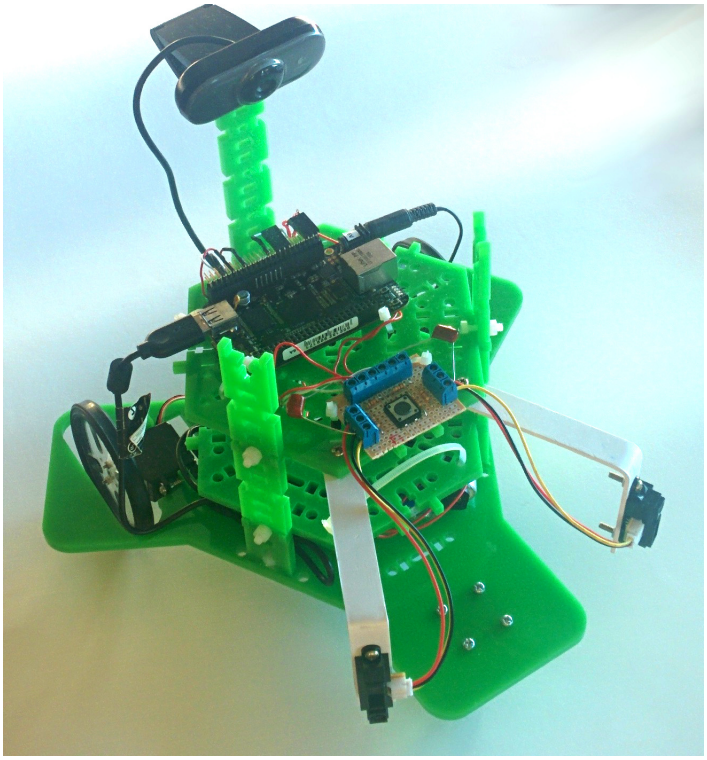

Figura 5.1 Robot de prueba

5.1.1. Comunicación con sensores

Se presentaron tres alternativas para comunicar los sensores con el sistema de control.

La primera opción era utilizar la Usb4Butiá [25], que es una placa de interfaz entrada/salida desarrollada en la Facultad de Ingeniería para la plataforma Butiá. La Usb4Butiá ya cuenta con soporte para los sensores de distancia Sharp, sin embargo la interfaz no es por eventos sino que está basada en polling. El diseño de la placa y el software de la Usb4Butiá son abiertos, por cual una posibilidad era extender el firmware, para que ésta se encargue de realizar el polling de los sensores y genere eventos cuando corresponda.

Como segunda opción se consideró utilizar el bus de control i2c. La Beaglebone Black cuenta con soporte para este bus de comunicación y se disponía de dos sensores para este bus. Si bien existe la posibilidad de utilizar el i2c junto con interrupciones, esto funciona si los sensores envían interrupciones, que no era el caso de los sensores disponibles. Por tanto, habría sido necesario leer los datos de los sensores mediante polling.

La tercera opción era leer los sensores utilizando las entradas analógicas de la Beaglebone Black. Esto en principio también implicaría la necesidad de hacer polling, pero la Beaglebone Black cuenta con dos coprocesadores adicionales llamados PRU (Programmable Real-time Unit), capaces de ejecutar su propio programa. Esto permite implementar una rutina que lea los sensores mediante polling y procese los datos sin ocupar el procesador principal, por ejemplo aplicando un umbral con histéresis, y que genere los eventos por interrupciones.

Esta ultima fue la opción elegida, ya que además de ser por eventos no requiere de recursos de hardware adicionales como la Usb4Butiá y un hub USB que permita conectar la cámara web junto con dicha placa.

Detalles de implementación

La placa de control Beaglebone Black cuenta con 7 entradas analógicas y un conversor analógico digital (ADC) de 12 bits con un rango de tensión de 0 - 1.8V. Sin embargo, los sensores de distancia Sharp según su hoja de datos llegan a un máximo de 3V. Por tanto, se construyó una placa con dos divisores de tensión, uno por cada sensor, usando resistencias de 10kΩ. Esto permite reducir a la mitad el voltaje de la salida del sensor, ubicándolo dentro del rango del ADC. Dicha placa posee además un botón con una resistencia pull-up.

El software de los sensores se divide en tres componentes: el programa PRU_Polling, la biblioteca LuaPRU, y un driver de Toribio que permite generar eventos cuando el sensor reporta datos.

PRU_Polling permite utilizar los sensores en dos modos: analógico simple o analógico con umbral. Los sensores simples devuelven el valor crudo. En tanto, los sensores con umbral devuelven un valor binario (0-1), según si valor sensado supera o no un umbral. En caso de tener umbral, debe configurarse dos parámetros: el nivel para el cual el valor pasa a 1 y el valor en cual pasa a ser 0. Para cada sensor debe especificarse además el canal según la entrada analógica que sea.

La rutina PRU_Polling ejecuta en la PRU y se encarga de consultar el valor del sensor mediante polling. Luego de inicializar el ADC, entra en un bucle que lee los canales analógicos y aplica el umbral si corresponde. Si el resultado cambió desde el ciclo anterior, lo guarda en memoria compartida y levanta la interrupción para enviarle el dato a la biblioteca LuaPRU.

En tanto, la biblioteca LuaPRU está programada en el lenguaje C y actúa como middleware entre el entorno Lua y los sensores. Permite cargar y ejecutar el programa PRU_Polling en la PRU. Por otra parte, la biblioteca guarda la configuración de los sensores a ser leídos en memoria compartida con la PRU. Por último, se registra a las interrupciones de la PRU para leer sus datos.

Resulta útil que la biblioteca permita especificar la configuración de los sensores, ya que evita tener que recompilar el programa de la PRU cada vez que se la desea modificar.

Por último, el deviceloader de Toribio conecta la biblioteca LuaPRU con el sistema de control. Primero levanta la configuración de los sensores del archivo de configuración *toribio.conf*, y luego un device con una salida *update* por cada sensor.

A continuación, el deviceloader necesita llamar a LuaPRU para quedar esperando por dato. Torocó esta basado en el sistema de multitarea Lumen, que es de hilo único, por lo que si este queda bloqueado esperando por la interrupción, el sistema se detiene. Para evitar esto y que el sistema siga ejecutando, se hace un fork para crear un hilo hijo y se crea un pipe que comunica los dos threads.

El hilo hijo queda bloqueado hasta que LuaPRU le envía un dato de un sensor. Cuando despierta, emite al hilo principal el número del sensor y el valor leído, escribiendo los datos en el pipe. Por otro lado, el hilo principal registra una función handler que recibe los datos del pipe, y se encarga de enviar el evento a los comportamientos, a través de la salida del device de Toribio.

5.2. Prueba de concepto

Para comprobar que el sistema de control permite implementar fácilmente un robot de conducta compleja, se desarrolló un robot móvil autónomo que sirve como prueba de concepto.

El Torobot se comporta como un toro bravo, es decir que si detecta objetos rojos, los persigue hasta embestirlos, y si no los detecta, se mueve cautelosamente sin rumbo. Por su parte, evita los objetos azules a menos que detecte además uno rojo. Además evita otro tipo de obstáculos, por ejemplo las paredes del recinto.

5.2.1. Arquitectura

Como se muestra en la figura 5.2, el sistema de control posee cinco comportamientos principales: uno de avanzar, dos de evitar obstáculos, uno de ignorar de objetos azules y uno de seguir objetos rojos.

El comportamiento *forward* hace que el robot avance en línea recta. Para ello, emite un evento fijo a los motores.

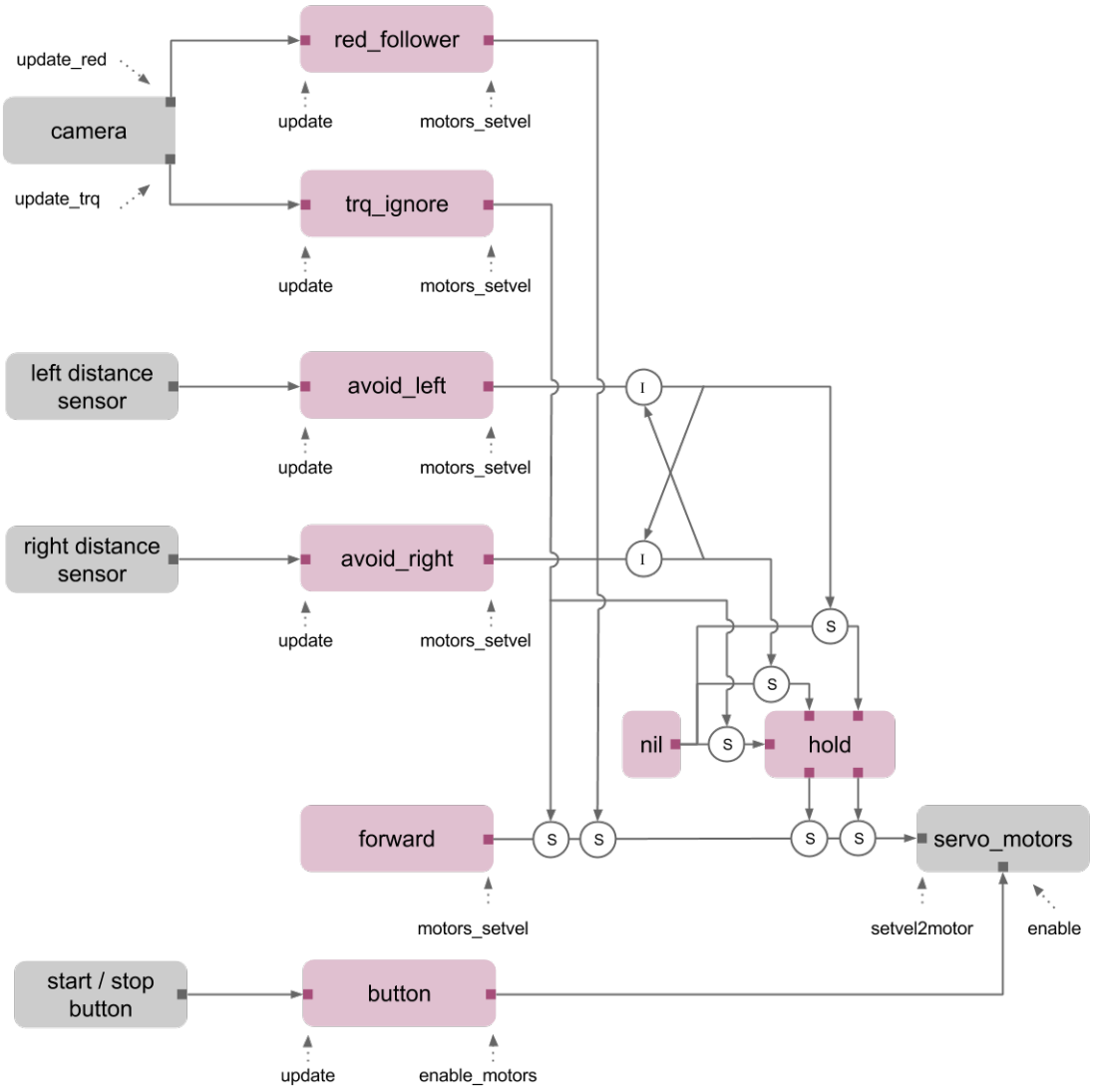

Figura 5.2 Arquitectura de control del Torobot

Por su parte, el *red_follower* tiene un trigger que recibe la posición del objeto rojo en la cámara. Si la posición en el eje x está cerca del centro, avanza en línea recta. Si está hacia

un lado de la cámara, avanza girando hacia ese lado. Si no hay objeto rojo en la cámara, el comportamiento libera su salida.

El *trq_ignore* es análogo a *red_follower*, pero funciona de modo inverso: si detecta un objeto azul a un lado de la cámara, avanza hacia el otro lado. Si no hay objeto azul en la cámara, el comportamiento libera su salida.

Los comportamientos *avoid_left* y *avoid_right* reciben el evento de los sensores de distancia. Si el valor es de cercanía, emite una salida hacia los motores para girar y evitar el obstáculo. Si el valor es de lejanía, se libera la salida.

Todos los comportamientos tienen conectados sus eventos de salida a la entrada del device del motor. El *red_follower* tiene mayor prioridad y suprime a los demás. El *trq_ignore* tiene prioridad media alta y suprime a los *avoid*. Por último, *forward* tiene prioridad mínima.

Cuando alguno de los comportamientos suprime el *forward* y luego se levantan dichas supresiones, es necesario que los motores reciban nuevamente el evento de avanzar hacia adelante. Lo mismo ocurre con otras parejas de comportamientos. Si el comportamiento que dejó de estar suprimido tuviera que encargarse de reenviar el evento, sería necesario notificarle que la supresión se levantó, lo que hubiera arruinado la modularidad del sistema.

En cambio, se optó por modificar la biblioteca Torocó para ofrecer un mecanismo de eventos fijos. Esto significa que al levantar la supresión de un evento, éste se reenvía al receptor automáticamente. De esta manera, el *forward* simplemente emite un evento fijo a los motores y deja de ejecutar. Su evento fijo queda activo aún después de dejar de ejecutar, y Torocó lo reenvía a los motores cuando corresponde.

5.2.2. Evasión de obstáculos

El robot se diseño para que evita golpear las paredes del recinto, y para que no derribe los objetos azules. En ambos casos, el robot responde girando hacia el lado contrario al obstáculo.

Al probar la primera versión del sistema de evasión, se detectó que cuando el robot se acercaba a una pared de manera semioblicua, uno de los sensores emitía un evento de cercanía, e instantes después lo hacía el otro sensor. Como uno de los sensores debía suprimir al otro en la entrada del motor, el robot impactaba contra el muro si se acercaba del lado incorrecto.

Para resolverlo, se definió que las salidas de los *avoid* se inhibieran entre sí. De este modo,

cuando el primer sensor detecta la pared, inhibe al otro sensor y el robot evita la pared adecuadamente.

A su vez, se daba una situación similar entre los sensores de distancia y el detector de objetos azules: si ambos daban verdadero, el robot impactaba contra la pared o el objeto azul. Para resolverlo, se modificó el diseño del sistema de control, de modo que si se detectan ambos obstáculos, el robot sigue girando en el sentido indicado por el *avoid*, hasta que se evitan ambos obstáculos.

Esto se logró mediante un módulo *hold*, que toma como entradas la salida de los comportamientos *trq_ignore* y *avoid*. Mientras *trq_ignore* sea verdadero, el módulo emite el último valor de *avoid*, aún si esta entrada se limpia. Cuando *trq_ignore* pasa a ser falso, se emite el valor de *avoid* tal cual la entrada.

Debido a la implementación de Torocó, el módulo *hold* no se entera cuando se limpia una entrada. Para que el módulo pueda detectar la acción, se conectó un evento fijo de menor prioridad a ambas entradas. Aquí se generó un nuevo problema: al conectar un mismo evento a dos entradas de un módulo, la supresión del evento afecta a ambas entradas, tal cual se había especificado el sistema (ver sección 4.1).

Esto se resolvió generando dos instancias del módulo que emite el evento fijo. Es una solución poco elegante pero que funciona adecuadamente. De todos modos, sería deseable que una futura versión de Torocó permitiera utilizar un mismo evento para múltiples entradas sin restricciones.

5.2.3. Detección de objetos

Por fuera del sistema de control Torocó, el robot ejecuta un programa de detección de objetos. El programa captura la filmación de la cámara y devuelve la ubicación del objeto rojo más grande; si no lo hay devuelve un valor nulo.

El detector de objetos está realizado en el lenguaje C++. Está basado en el programa ColorDetect [26], y utiliza la biblioteca de computación gráfica OpenCV [27]. Mediante las operaciones *inRange()*, *dilate()*, *erode()* y *findContours()*, el programa reconoce la región más grande de la imagen que coincide con el rango de colores indicado. El programa ejecuta dos funciones de manera alternada, para detectar rojo y azul simultáneamente.

Para mejorar la precisión en la detección de rojo, el programa opera con los valores de *hue*, *lightness* y *saturation*. Para cada píxel, se aplica una serie de filtros para eliminar los colores

distintos al rojo vivo, lo cual consiste en asignar dicho píxel como gris.

El resultado de los filtros se despliega en la pantalla de la computadora desde la que se controla al robot. Éste muestra los píxeles reconocidos de la imagen, así como la región reconocida más grande y el centro. Esto permite verificar si el reconocedor funciona correctamente.

El programa envía las coordenadas (x, y) del centro del objeto en un paquete UDP, junto con un indicador de azul o rojo. Por su parte, el sistema de control Torocó tiene un sensor que recibe los paquetes UDP y emite el evento *update_red* o *update_trq* según el color, con los valores (x, y) como datos adjuntos. Los comportamientos *red_follower* y *trq_ignore* capturan los respectivos eventos y los procesan.

5.2.4. Resultados

La mayor dificultad para realizar la detección de objetos fue la utilización del OpenCV en la placa del robot. Para instalarlo se requirieron numerosos pasos para resolver dependencias de bibliotecas, lo que llevó un tiempo extenso. Por otra parte, durante el desarrollo del robot se descubrió que la biblioteca gstreamer tenía un desempeño notoriamente superior en comparación con la que utilizaba el OpenCV original al capturar la imagen. Se descubrió que OpenCV era capaz de utilizar gstreamer, por lo que se modificó la biblioteca para que la utilice. Esto tomó su tiempo pero significó una mejora notable en el desempeño del detector de objetos.

El trabajo restante para lograr la detección de objetos se desarrolló sin mayores inconvenientes. Por ejemplo, la comunicación entre el detector y el sistema de control consistió en levantar un socket UDP desde un device Toribio, lo cual no presentó dificultades. Aún así, el detector de objetos requiere ser calibrado frecuentemente para adaptarlo a los cambios de luz en el entorno. Esto se debe a que el rango de detección es fijo, y no depende del contexto como puede ser el brillo y contraste detectado.

El detector de objetos fue modificado para evitar que reconociera colores similares al rojo, en particular marrón y rosado, pero dichos filtros se dejaron de lado. La documentación del OpenCV fue insuficiente, por lo que llevó cierto tiempo entender su sistema de colores. De todos modos, una vez lograda dicha tarea fue fácil definir filtros simples para reconocer objetos. La cámara web poseía escasos rangos de contraste y luz dinámica, pero no generó problemas mayores.

Realizar los comportamientos fue la parte más sencilla de la prueba. La lógica de los comportamientos resultó ser sencilla, lo que mostró que la arquitectura Subsumption lleva hacia soluciones donde los comportamientos funcionan de manera reactiva. Además, con poco esfuerzo se logró que los comportamientos tuvieran la conducta deseada, es decir que Torocó mostró ser fácil de utilizar.

En cambio, la parte más dificultosa de la especificación del sistema de control fue determinar la interacción entre los comportamientos. Resolver la evasión de obstáculos requirió de varias iteraciones y un esfuerzo considerable para hallar una solución adecuada. Aún así, la esfuerzo se centró en determinar la arquitectura de control, y no en comprender cómo implementar la arquitectura propuesta utilizando Torocó. Además, la iteraciones se realizaron de abajo hacia arriba, es decir agregando funcionalidades al sistema sin tener que modificar las existentes, como afirma la propuesta de Brooks.

5.3. Pruebas de verificación

A lo largo del desarrollo, se realizaron una serie de pruebas para verificar que el funcionamiento sea el especificado. Las pruebas representan las distintas formas de utilizar la interfaz de usuario.

Se probó ejecutar los seis ejemplos tanto en una computadora como en la placa del robot. Todos los ejemplos tienen como entrada el ratón, y como salida la consola.

Comunicación por eventos

El programa *sensor_signalling* muestra cómo comunicar eventos desde un sensor a un comportamiento, entre comportamientos, y desde un comportamiento a un actuador.

El comportamiento emisor captura los eventos de clic izquierdo y derecho del ratón, y emite un evento repetidor con el mensaje *trigger_left* o *trigger_right* respectivamente, así como el valor verdadero al presionar el botón y falso al levantar el botón. En tanto, el comportamiento receptor recibe dicho evento y lo reenvía al actuador.

Al ejecutarse, el programa imprime los mensajes correspondientes cada vez que se presiona o levanta el clic izquierdo o derecho.

Inhibición y supresión de eventos

El programa *inhibition_suppression* muestra cómo utilizar los mecanismos de inhibición y supresión de eventos.

El comportamiento de nivel 1 captura los eventos de clic izquierdo del ratón, y envía un evento al actuador. En tanto, el comportamiento de nivel 2 captura los eventos de clic derecho, y emite un evento de inhibición o supresión, de modo que el comportamiento de nivel 1 no recibe los eventos del ratón durante 2,5 segundos.

Al ejecutar el programa, se imprimen los mensajes correspondientes al clic izquierdo cuando el botón derecho está levantado, y no se imprimen durante 2,5 segundos al presionar el botón derecho.

Esperar por eventos en trigger

El programa *wait_for_event_trigger* muestra cómo pausar la ejecución de un trigger, esperar por un evento y reanudar la ejecución.

El comportamiento de nivel 1 tiene un trigger que espera a que ocurran dos eventos del clic izquierdo, y emite un evento de salida. En tanto, el actuador captura ese evento e imprime los datos adjuntos.

Por su parte, el comportamiento de nivel 2 tiene recibe los eventos de clic derecho. Al presionar el botón, envía un evento que inhibe el clic izquierdo por 2,5 segundos, y al levantar el botón desinhibe el clic izquierdo inmediatamente.

Al iniciar el programa, se imprimen los datos luego de presionar y levantar el clic izquierdo del ratón. Sin embargo, al mediante clic derecho se inhibe el clic izquierdo durante 2,5 segundos, por lo que al presionarlo y levantarlo el programa no responde. Esto implica que si durante la inhibición se cambia el estado del clic izquierdo, el funcionamiento del clic izquierdo queda invertido, es decir que se imprimen los datos luego de levantar y presionar el clic izquierdo del ratón.

Esperar por eventos en corutina

El programa *wait_for_event_coroutine* muestra cómo pausar la ejecución de una corutina, esperar por un evento y reanudar la ejecución.

Es similar a *wait_for_event_trigger*, pero el comportamiento de nivel 1 tiene una corutina que ejecuta un bucle infinito. En cada ciclo se espera por dos eventos del clic izquierdo, y emite un evento de salida. En tanto, el actuador captura ese evento e imprime los datos adjuntos.

Suspender y reanudar comportamientos

El programa *beh_suspend_resume* muestra cómo suspender y reanudar la ejecución de comportamientos.

El comportamiento de nivel 1 imprime un mensaje al cliquear el botón izquierdo. Por su parte, el comportamiento de nivel 2 lo suspende al cliquear el botón derecho, y lo reanuda al levantarlo.

Agregar y quitar comportamientos

El programa *beh_add_remove* muestra cómo agregar y quitar comportamientos durante la ejecución del sistema de control.

Al iniciar el programa, el comportamiento de nivel 1 imprime un mensaje en pantalla al cliquear el botón. Por su parte, el comportamiento de nivel 2 espera a que ocurra el clic derecho.. Cuando ocurre, el handler del trigger quita el propio comportamiento, y luego agrega un comportamiento nuevo, que se encarga de inhibir el clic izquierdo de manera similar al ejemplo *inhibition_suppression*.

5.4. Pruebas de desempeño

Se realizaron dos pruebas de desempeño del sistema, más precisamente del tiempo de respuesta. En la prueba externa se simula un sensor y un actuador que ejecutan en un computadora diferente a la del sistema, por lo que se mide el tiempo desde que un sistema remoto envía un dato hasta que otro sistema remoto lo recibe.

La prueba interna es similar, pero la generación de eventos la realiza un comportamiento, por lo que se mide el tiempo desde que un comportamiento emite el dato hasta que un actuador lo recibe.

5.4.1. Prueba externa

En la primera prueba para evaluar la respuesta del sistema Torocó, un sensor recibe una serie de paquetes UDP y emite eventos hacia un comportamiento. El comportamiento los recibe y reenvía hacia un actuador. El actuador envía un paquete UDP por cada evento recibido.

Como se muestra en la figura 5.3, se realizó un programa emisor que envía paquetes UDP hacia el sensor. De manera análoga, un programa receptor se encarga de recibir los paquetes UDP del actuador. El programa emisor y el receptor están escritos en el lenguaje C++ y ejecutan en una computadora remota.

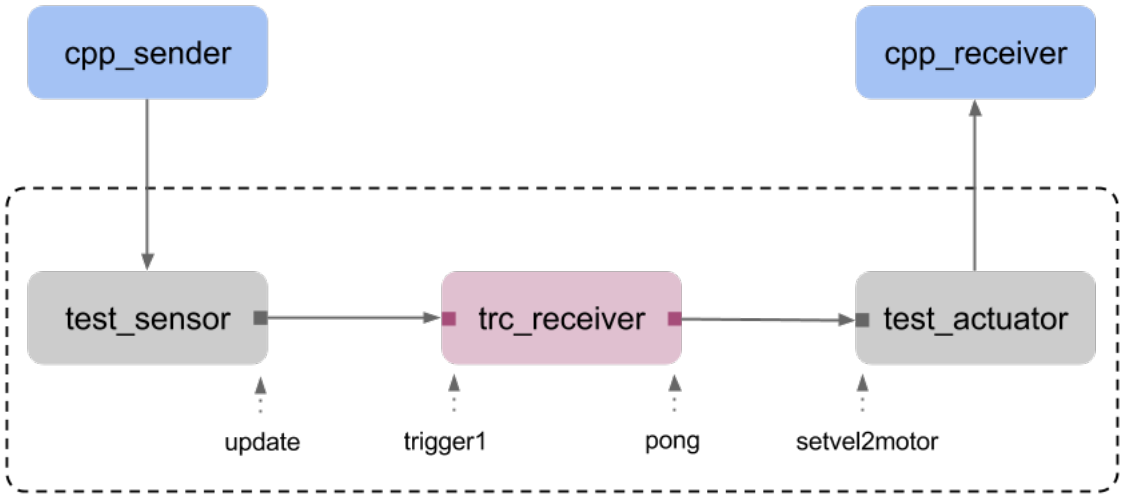

Figura 5.3 Arquitectura de la prueba externa de desempeño del sistema

Los paquetes contienen la hora del sistema a la que se emiten. El programa receptor compara la hora de recepción con la del paquete. Esto permite determinar el tiempo que transcurre entre la emisión y la recepción de los paquetes. Este dato sirve para estimar la latencia del sistema de control.

A su vez, al comparar el tiempo entre dos paquetes consecutivos, se puede determinar la tasa de transferencia externa del sistema.

Se utilizó el adaptador de red USB de la Beaglebone Black que permite la comunicación Ethernet entre la computadora remota y la placa del robot usando el cable USB. El intervalo entre paquetes enviados es de 10 milisegundos, el más corto que no genera paquetes perdidos.

5.4.2. Prueba interna

En la segunda prueba, se eliminaron los programas emisor y receptor de C++. En cambio, se agregó un comportamiento emisor, que ejecuta un bucle que emite eventos lo más rápido posible. Por su parte, el actuador recibe los paquetes del comportamiento receptor, y compara la hora de recepción con la del paquete (ver figura 5.4).

A su vez, al comparar el tiempo entre dos paquetes recibidos, se puede determinar la tasa de transferencia interna del sistema.

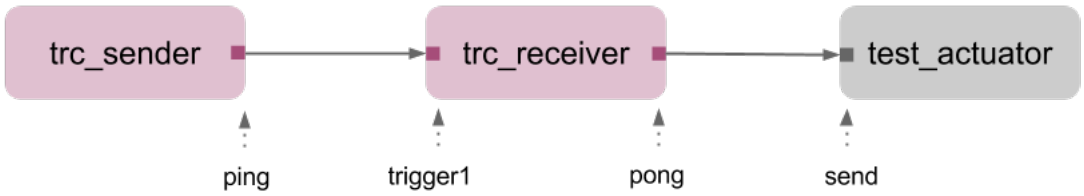

Figura 5.4 Arquitectura de la prueba interna de desempeño del sistema

5.4.3. Resultados

Los cuadros 5.2 y 5.3 muestran los resultados que se obtuvieron en la prueba externa y la prueba interna. En todos los casos se emitieron 10.000 eventos.

La penúltima columna de los datos de la prueba permite calcular la tasa de refresco efectiva. Para una tasa de 309 Hz, la carga del procesador en la prueba interna es del 75 %. Para una tasa de 132 Hz se requiere un 45 % de carga, y para 80 Hz se necesita un 28 %.

La demora promedio del ciclo de paquetes ronda los 1,5 milisegundos. Es un tiempo de respuesta bajo, perfectamente adecuado para un robot móvil. Por ejemplo, si el robot se desplaza a 60 km/h, el tiempo de reacción equivaldría a un recorrido de 25 milímetros.

Intervalo de emisión (ms)	Demora máxima (ms)	Demora promedio (ms)	Demora mínima (ms)	Carga máxima del procesador (%)
10,0	9,248	2,533	2,085	30

Cuadro 5.2 Resultados de la prueba externa de desempeño del sistema

Intervalo de emisión (ms)	Demora máxima (ms)	Demora promedio (ms)	Demora mínima (ms)	Intervalo promedio (ms)	Carga máxima del procesador (%)
0,0	9,169	0,422	0,335	0,738	99
2,0	9,421	0,725	0,395	3,237	75
5,0	7,947	1,522	1,273	7,598	45
10,0	7,895	1,534	1,273	12,551	28

Cuadro 5.3 Resultados de la prueba interna de desempeño del sistema

Al comparar las dos pruebas con intervalo de emisión de 10 milisegundos, la prueba externa genera tiempos ligeramente más altos que la prueba interna, al obtenerse una demora promedio de 2,5 milisegundos. Utilizando el mismo cálculo que para la prueba interna, para un robot que se desplaza a 60 km/h la demora equivale a un desplazamiento de 40 milímetros, un valor que sigue siendo muy bajo.

Capítulo 6

Análisis y conclusiones

En el capítulo final se realiza una serie de análisis y conclusiones del producto y el proyecto. La sección 6.1 presenta una evaluación de las distintas características del sistema. La sección 6.2 contiene propuestas de trabajos futuros. El capítulo cierra en la sección 6.3 con las conclusiones finales del proyecto.

6.1. Evaluación del sistema

6.1.1. Facilidad de uso

Los sistemas de control son radicalmente distintos a otros tipos de programas, como pueden ser los sistema de información o videojuegos. Del mismo modo, las distintas arquitecturas de control de robots difieren notablemente entre sí.

Torocó tiene varias características particulares, tales como la inhibición y supresión de datos, la ejecución concurrente y la comunicación basada en eventos. Entender la arquitectura y las demás características del sistema pueden requerir un esfuerzo considerable, en particular si el usuario no está familiarizado con los sistemas de control ni la concurrencia.

Otro punto a mencionar es que Lumen es un planificador cooperativo, lo que significa que éste no quita el control del sistema a los comportamientos. Ellos deben ceder el control voluntariamente, o en caso contrario el funcionamiento del sistema se degrada. Para que funcione, el programador debe realizar correctamente los comportamientos.

Para mejorar la facilidad de uso, se buscó reducir la cantidad de código redundante o repetido (*boilerplate code*) que debe escribir el usuario, y lograr una interfaz de aplicación lo más simple posible. Se buscó que los ejemplos funcionales más simples tuvieran un código reducido y simple. También se intentó evitar que el sistema introduzca complejidad, es decir que dado un problema, la única dificultad sea definir el programa que lo resuelva.

Otro aspecto del sistema es que fue pensado para que haya un mapeo lo más directo posible entre el código y una representación gráfica. Esto se muestra en el programa principal del sistema de control, que utiliza una función para declarar los comportamientos a utilizar y otra para definir las interconexiones. Esto facilitará un eventual proyecto de entorno de desarrollo gráfico, que permita definir la interconexión entre los comportamientos y dispositivos dibujando líneas.

6.1.2. Desempeño

Las pruebas de verificación fueron exitosas, y por tanto no se conocen errores de programación en Torocó. En cuanto a la completitud, las pruebas de validación permitieron realizar sistemas de control complejos, y la biblioteca ofrece una gran variedad de funcionalidades. Por tanto, no hay indicios de que el sistema sea incompleto.

Torocó no fue implementado en un sistema en tiempo real, por lo tanto no se tienen garantías para el tiempo de respuesta del sistema. No obstante, las pruebas de desempeño muestran que Torocó tiene un tiempo de respuesta bajo, y en las pruebas de validación el robot funcionó adecuadamente para la tarea planteada.

El sistema ofrece la posibilidad de que toda la comunicación del sistema sea por eventos. Se incluye una opción para fijar eventos, lo que evita la necesidad de tener que reenviarlos cuando se levanta una inhibición o supresión. Ambas cualidades reducen drásticamente la necesidad de realizar polling, lo que simplifica considerablemente el programa resultante.

6.1.3. Manejo de errores

Torocó es capaz de detectar errores de usuario, ante los cuales el sistema deja de ejecutar. En dicho caso se devuelve un mensaje descriptivo, indicando la función que falla, el motivo del error y las estructuras de datos que se ingresaron.

No obstante, no se logró una detección total de errores de usuario. En algunos casos, se intentó detectar un error pero también se los detectaba en programas correctos, por lo que se decidió ignorar dicho error.

En otros casos, ocurrió que el error no se puede detectar en el momento correcto, debido a que el programa no posee la información completa. Esto ocurrió por ejemplo porque los archivos de comportamientos no devuelven una lista de eventos de entrada. Por tanto, si se intenta conectar un evento a una entrada inexistente, el programa no puede detectar el error. Esto se debe a que los eventos de salidas del comportamiento se desconocen hasta que una corutina emite la salida.

6.2. Conclusiones

Se desarrolló un sistema de control de robots móviles autónomos basado en comportamientos con arquitectura Subsumption. Dicho sistema consiste en una serie de comportamientos que ejecutan concurrentemente, y pueden intercambiar datos entre sí y con el hardware del robot mediante comunicación por eventos. Los comportamientos pueden inhibir los eventos de salida de otros comportamientos y suprimir los eventos de entrada.

El sistema de control se probó en un robot móvil autónomo sobre la plataforma Butiá, que posee cámara y sensores de distancia. Dicho robot tiene un comportamiento que sigue objetos rojos, otro que ignora objetos azules, y otro que evita obstáculos.

Esto permitió mostrar que el sistema de control es lo suficientemente flexible para lograr una conducta compleja, con poco esfuerzo por parte del programador del robot. En tanto, el sistema de control mostró un desempeño adecuado en las diversas pruebas realizadas con el robot, al responder rápidamente ante los cambios en el entorno, y no presentó errores en las pruebas realizadas.

La interfaz de aplicación de Torocó sigue fielmente la propuesta de Brooks de interconexión de módulos, en la que los módulos y la interconexión se definen por separado. De esta manera, los comportamientos resultan simples y reutilizables, a la vez que la complejidad del sistema se encuentra en las interconexiones. Además, se desarrolló un novedoso sistema de eventos fijos, que facilita la implementación de los comportamientos.

La biblioteca se desarrolló en el lenguaje Lua mediante la biblioteca Toribio, por lo que se podría portar a múltiples plataformas robóticas. Además, el sistema podría integrarse a otras herramientas de desarrollo, tales como el entorno de programación gráfico Yatay.

En síntesis, se logró desarrollar una implementación de la arquitectura de control Subsumption, fiel a la propuesta de Brooks y que mostró ser efectiva en un robot de prueba.

6.3. Trabajo futuro

6.3.1. Entorno de desarrollo gráfico

Subsumption se caracteriza por su diseño modular, en el que una parte importante del desarrollo del programa consiste en interconectar bloques. Esta tarea sería más fácil de realizar de manera gráfica que escrita. Por tanto, se podría desarrollar una herramienta gráfica para realizar dicha tarea, en la que las interconexiones se definan dibujando líneas entre los bloques.

Cada instancia de comportamiento se representaría mediante un rectángulo que tendría sus entradas a la izquierda y sus salidas a la derecha. Para definir la interconexión, el usuario trazaría líneas entre la salida del emisor y la entrada del receptor.

Esto permitirá que el usuario comprendiera mejor el funcionamiento del sistema de control, a la ves que reduciría el esfuerzo de desarrollo y mantenimiento. Además reduce la curva de aprendizaje del sistema, lo que permitiría acceder a un espectro más amplio de usuarios, en especial los liceales.

6.3.2. Integración con Yatay

El proyecto Yatay es un sistema de control de robot basado en Toribio, cuya arquitectura es basada en comportamientos pero es secuencial, es decir que ejecuta un solo comportamiento a la vez.

Yatay incorpora un entorno de programación gráfico, orientado a niños y adolescentes. En vez de programar escribiendo texto, el usuario define el código de los comportamientos enganchando elementos gráficos en una lista vertical. Sería interesante combinar el entorno gráfico de Yatay con el sistema de control Torocó. Esto permitiría que los niños y adolescentes puedan aprender a desarrollar programas de ejecución paralela mediante una interfaz amigable.

Por otra parte, Yatay ofrece un servicio de repositorio de proyectos, que permite que varios usuarios colaboren para desarrollar un mismo sistema de control. Agregar dicha funcionalidad a Torocó haría más fácil que un grupo de personas realice un sistema de control en colaboración.

6.3.3. Biblioteca de comportamientos

La arquitectura Subsumption pretende aumentar la reutilización de comportamientos, y el sistema Torocó siguió dicha filosofía. Por tanto, sería interesante ofrecer una biblioteca de comportamientos, es decir un conjunto de comportamientos básicos que se puedan combinar para implementar sistemas de control.

Además de ofrecer la biblioteca de comportamientos en sí, otra línea de trabajo posible sería desarrollar un sistema de distribución de comportamientos, que permite que los usuarios exporten sus comportamientos a un repositorio web, y a su vez descargar comportamientos realizados por otras personas.

Sumando esto al repositorio de proyecto, se podría crear una comunidad de desarrolladores de sistemas de control de robots. Más aún, sería interesante que el sistema permitiera saber qué proyectos utilizan cada uno de los comportamientos.

Bibliografía

[1] M. J. Mataric, *The Robotics Primer*. MIT Press, 2007.

[2] R. C. Arkin, *Behavior-Based Robotics*. MIT Press, 1998.

[3] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation Vol. RA-2*, no. 1, 1986.

[4] R. R. Murphy, *Introduction to AI Robotics*. MIT Press, 2000.

[5] B. Hayes-Roth, "An architecture for adaptative intelligent systems," *Artificial Intelligence*, no. Vol 72, No 1-2, January, 1995.

[6] M. J. Mataric, "Behavior-based control: Main properties and implications," *International Conference on Robotics and Automation*, 1992.

[7] M. J. Mataric, "Behavior-based control: Examples from navigation, learning, and group behavior," *Journal of Experimental and Theoretical Artificial Intelligence*, 1997.

[8] J. Rosenblatt and D. Payton, "A fine-grained alternative to the subsumption architecture for mobile robot control," *International Joint Conference on Neural Networks*, 1999.

[9] J. Brustoloni, "Autonomous agents: Characterization and requirements," carnegie mellon technical report cmu-cs-91-204, Pittsburgh: Carnegie Mellon University, 1991.

[10] F. Osorio, D. Wolf, E. Simoes, G. Pessin, L. Fernandes, A. Hata, P. Shinazto, M. Dias, and L. Couto, "Tele-operated and autonomous mobile robots," 2012.

[11] R. Siegwart and I. R. Nourbakhsh, "Introduction to autonomous," *MIT Press*, 2011.

[12] M. J. Mataric, "Learning in behavior-based multi-robot systems: Policies, models, and other agents," *Cognitive Systems Research*, 2001.

[13] "History of lego robotics." http://www.lego.com/en-us/mindstorms/gettingstarted/historypage/. Visitado: 22/05/2014.

[14] "Lejos tutorial - behavior programming." http://www.lejos.org/nxt/nxj/tutorial/Behaviors/BehaviorProgramming.htm. Visitado: 05/02/2015.

[15] C. Flanagan, D. Toal, and B. Strunz, "Subsumption architecture for the control of robots," *University of Limerick*, 1996.

[16] A. Achkar and A. Margale, "Ide de programación orientado al desarrollo de arquitecturas robóticas basadas en comportamientos," *Facultad de Ingeniería*, 2013.

[17] "Sitio oficial del proyecto butiá." http://www.fing.edu.uy/inco/proyectos/butia/. Visitado: 05/02/2015.

[18] "Wiki del proyecto butiá." http://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/Wiki_Butiá. Visitado: 05/02/2015.

[19] A. Nebel and R. Rozza, "Proyecto yatay." http://www.fing.edu.uy/~pgbutiablet, 2014. Visitado: 22/05/2014.

[20] "Sitio oficial de blockly." http://developers.google.com/blockly/. Visitado: 05/02/2015.

[21] J. Visca, "Toribio." https://github.com/xopxe/Toribio, 2012. Visitado: 22/05/2014.

[22] L. H. d. F. Roberto Ierusalimschy, Waldemar Celes, "Lua." http://www.lua.org/about.html. Visitado: 22/05/2014.

[23] J. Visca, "Lumen." https://github.com/xopxe/Lumen, 2012. Visitado: 22/05/2014.

[24] J. Liu, *Real-Time Systems*. Prentice-Hall, 2000.

[25] "Usb4butiá." http://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/USB4butiá. Visitado: 22/05/2014.

[26] D. Oleszczuk, "Colordetect documentation." https://gitorious.org/colordetect/colordetect. Visitado: 29/10/2014.

[27] "Opencv documentation." http://docs.opencv.org/. Visitado: 29/10/2014.

Anexo A

Glosario

- **Arquitectura de control**: Especificación de la estructura de un sistema de control.
- **Capa de competencia**: Conjunto de módulos que funcionan en conjunto para realizar una acción de control del robot.
- **Comportamiento**: Módulo que permite procesar los datos de eventos de entrada y emitir eventos de salida.
- **Comunicación por eventos**: Método de comunicación por el cual un emisor envía eventos, opcionalmente con datos adjuntos. Los comportamientos pueden registrarse a los eventos de otros emisores.
- **Conducta**: Acciones que realiza el robot. Abarca tanto las acciones visibles como el procesamiento interno.
- **Device**: Módulo que permite la comunicación con un dispositivo de hardware.
- **Emisor**: Sensor o comportamiento que envía eventos.
- **Entorno parcialmente observable**: Entorno que del robot no es capaz de obtener información completa.
- **Módulo**: Componente de software que posee canales de comunicación de entrada y salida. Término genérico que abarca a comportamientos, sensores y actuadores.
- **Inhibición**: Método de interacción entre comportamientos, por el cual el inhibidor anula un evento de salida de un emisor de una capa inferior.

- **Paradigma de control**: Caracterización general de una familia de sistemas de control.
- **Receptor**: Actuador o comportamiento que recibe eventos.
- **Robot móvil**: Robot que puede desplazarse libremente en el entorno, por ejemplo con ruedas u orugas motorizadas.
- **Robot móvil autónomo**: Robot que puede navegar entornos desconocidos, procesando la información obtenida de sus sensores.
- **Sistema de control**: Programa que permite definir la conducta de un robot.
- **Subsumption**: Arquitectura de control de robots basado en comportamientos que consiste en una serie de módulos que ejecutan concurrentemente e intercambian datos entre sí.
- **Supresión**: Método de interacción entre comportamientos, por el cual el supresor anula un evento de entrada de un receptor de una capa inferior.
- **Torocó**: Biblioteca que permite implementar un sistema de control de robots con arquitectura Subsumption.

Anexo B

Interfaz de aplicación

La biblioteca Torocó ofrece al usuario las siguientes funciones para definir los comportamientos y la comunicación entre comportamientos, sensores y actuadores:

B.1. Descriptores

Los descriptores se utilizan para hacer referencia a los elementos del sistema, son ayudantes que generan un tabla para que sea mas fácil de leer al escribir el código. Se usan como parámetros de entrada a las funciones del sistema.

B.1.1. Device

El código *device.device1* devuelve un descriptor del device llamado *device1*.

B.1.2. Evento de device

El código *device.device1.event1* devuelve un descriptor del evento de salida *event1* del device llamado *device1*.

bsubsectionComportamiento

El código *behavior.beh1* devuelve un descriptor del comportamiento llamado *beh1*.

B.1.3. Evento de comportamiento

El código *behavior.beh1.event1* devuelve un descriptor del evento de salida *event1* del comportamiento llamado *beh1*.

B.1.4. Entrada

El código *input.input1* devuelve un descriptor de la entrada llamada *input1*. Se utiliza en *trigger()*.

B.2. Entrada de eventos

B.2.1. Esperar por un evento individual

wait_for_input (input_desc, timeout)

La función pausa una corutina o handler de trigger de un comportamiento, y espera por una entrada del comportamiento. Cuando ocurre la entrada deseada, la corutina recibe los datos del evento y se reanuda la ejecución.

El parámetro de entrada es el descriptor de la entrada, y se devuelven los datos de la entrada recibida. Opcionalmente se puede establecer un tiempo de espera. De este modo, si luego de ese tiempo no se recibió la entrada, la función devuelve como resultado un valor nulo y la corutina reanuda su ejecución.

B.2.2. Registrarse a un evento permanentemente

trigger (input_desc, handler)

La función devuelve una corutina que implementa el trigger. De este modo, cada vez que el comportamiento recibe la entrada, se ejecuta la función handler. La corutina se debe incluir en la lista de valores retornados del comportamiento.

Los parámetros de entrada son el descriptor de la entrada y la función handler del trigger.

B.3. Salida de eventos

B.3.1. Enviar salida individual

send_output (output_values)

La función envía todos los eventos de salida del comportamiento por una única vez. Para cada evento, se debe especificar una tabla con los datos a enviar (puede ser una tabla vacía), así como el tiempo de duración de la inhibición/supresión.

Si los eventos de salida del comportamiento son entradas de otros receptores, se aplica la supresión a los eventos de menor prioridad para dichos receptores durante el tiempo especificado.

Si los eventos de salida del comportamiento son inhibidores de salidas de otros emisores, se aplica la inhibición para dichos emisores durante el tiempo especificado.

B.3.2. Fijar salida persistente

set_output (output_values)

La función fija los eventos de salida del comportamiento de manera persiste. Es equivalente a *send_output()*, con la diferencia de que si uno de los eventos de salida es inhibido o suprimido y luego se levanta la acción, se vuelve a enviar el evento a los receptores.

B.3.3. Limpiar salida

unset_output ()

La función limpia todos los eventos de salida del comportamiento. De este modo, si las salidas son entradas prioritarias de otros receptores o inhibidoras de salidas de otros emisores, se reenvían los eventos asignados que correspondan.

B.4. Configuración de comportamientos

B.4.1. Cargar comportamiento desde archivo

load_behavior (behavior_desc, pathname, params)

La función carga un comportamiento desde un archivo. Los parámetros de entrada son el descriptor del comportamiento, la ruta del archivo y opcionalmente una tabla de parámetros. Típicamente se utiliza en el programa principal.

El archivo del comportamiento debe devolver como resultado la lista de corutinas y triggers.

B.4.2. Agregar comportamiento manualmente

add_behavior (behavior_desc, coroutines, params)

La función permite agregar un comportamiento de forma manual. Los parámetros de entrada son el descriptor del comportamiento, una tabla de corutinas y opcionalmente una tabla de parámetros.

B.4.3. Asignar entradas a un comportamiento

set_inputs (receiver_desc, inputs)

La función define las entradas del receptor. El parámetro de entrada *receiver_desc* es el descriptor de un comportamiento o actuador, y el parámetro *inputs* es una tabla con los descriptores de las entradas, que pueden ser eventos de salida o bien funciones de sensores.

B.4.4. Asignar inhibidores de un comportamiento

set_inhibitors (emitter_desc, outputs_inhibitors)

La función define los eventos inhibidores de cada salida del emisor. El parámetro de entrada *emitter_desc* es el descriptor de un comportamiento o sensor, y el parámetro *outputs_inhibitors* es una tabla con los descriptores de los inhibidores de cada salida.

B.5. Ejecución de comportamientos

B.5.1. Iniciar ejecución de los comportamientos

run (toribio_conf_file)

La función inicia la ejecución de los comportamientos. Opcionalmente se puede especificar la ruta del archivo de configuración de Toribio (por defecto se usa *toribio.conf*).

Suspender ejecución de un comportamiento

suspend_behavior (behavior_desc)

La función suspende la ejecución de un comportamiento. Esto significa que sus corutinas y handlers dejan de ejecutarse. El parámetro de entrada es el descriptor del comportamiento.

B.5.2. Reanudar ejecución de un comportamiento

resume_behavior (behavior_desc)

La función reanuda la ejecución de un comportamiento. Esto significa que sus corutinas y handlers vuelven a ejecutarse. El parámetro entrada es el descriptor del comportamiento.

B.5.3. Eliminar comportamiento

remove_behavior (behavior_desc)

La función elimina un comportamiento de manera permanente. El parámetro de entrada es el descriptor del comportamiento.