Tesis de grado
Ingeniería en Computación
ALLIANCE sobre Torocó:
Arquitecturas cooperativas basadas en comportamientos

Mauro Mottini

Supervisor: Facundo Benavides

29 de julio de 2021

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

Índice general

Índice de figuras

Capítulo 1

Presentación

Uno de los factores que más potencia el desarrollo en la robótica es el beneficio que brinda tener robots realizando actividades que pueden ser peligrosas para un humano, ya sea porque la actividad en sí es de riesgo o porque el lugar en el cual deben ser realizadas es de riesgo para una persona. Algunos ejemplos pueden ser la limpieza de material tóxico o radioactivo, exploración espacial, misiones de rescate, tareas en fábricas que pueden resultar peligrosas o extremadamente repetitivas, entre otras. En la mayoría de los casos, se requiere de robots que puedan funcionar de forma totalmente autónoma para lograr sus metas, sin la necesidad de interacción humana.

Un posible enfoque para resolver estos problemas es la implementación de un robot único que pueda resolver todos los problemas que sean necesarios, y que logre adaptarse a los cambios en el entorno a medida que sea necesario. El problema de este enfoque es que se espera mucho de un único robot, haciendo que su complejidad crezca rápidamente a medida que se agregan nuevas funcionalidades y generando un punto de falla único para la misión. Además, existen tareas cuya naturaleza es paralelizable o requieren de mas de un robot para poder ser completadas.

Dadas estas razones, muchas veces se desea tener un conjunto de robots, posiblemente heterogéneo, que trabajen juntos para poder realizar una misión que sería imposible para uno solo. El problema con esto es que cuando se quiere manejar un conjunto de robots como el mencionado, surgen varias dificultades que se deben resolver.

- ¿Cómo dividimos las diferentes tareas a realizar entre los diferentes robots?
- ¿Cómo se maneja la comunicación e interacciones entre robots?
- ¿Cómo podemos asegurar que los robots actúen de forma coherente?
- ¿Cómo permitimos que los robots reconozcan y resuelvan posibles problemas que pueden surgir?

La arquitectura ALLIANCE, propuesta por Lynne E. Parker en 1994 [Parker, 1994], surge como una solución a los problemas planteados anteriormente y permite un manejo de los robots en forma cooperativa y tolerante a posibles fallos.

La implementación que se plantea como parte de este informe fue realizada sobre la biblioteca Torocó [Bettosini and Clavelli, 2015] desarrollada por Ignacio Bettosini y Agustín Clavelli en el año 2015 utilizando el lenguaje de programación Lua [d. F. Roberto Ierusalimschy and Celes, 1993]. Esta brinda una implementación de la arquitectura Subsumption propuesta por R. A. Brooks para la implementación de robots reactivos. Dado que la arquitectura Subsumption es una predecesora de ALLIANCE, y esta última toma los elementos de Subsumption y construye sobre ellos, se concluyó que construir sobre Torocó y agregar las funcionalidades necesarias para permitir la implementación de sistemas ALLIANCE era un buen camino para la implementación.

1.1. Objetivos

El objetivo principal de este proyecto es la correcta implementación de la arquitectura ALLIANCE a partir de la biblioteca Torocó, la cual ya brinda una implementación de la arquitectura Subsumption, pero a partir de este objetivo primario también surgen nuevos objetivos a tener en cuenta.

Por un lado se quieren mantener los objetivos tanto funcionales como no funcionales planteados por Torocó, como pueden ser el rendimiento de la arquitectura y la facilidad de uso para el usuario. Nuestra implementación deberá ser compatible con cualquier código que utilice Torocó actualmente, sin remover funcionalidades ni empeorar su rendimiento.

Por otro lado también debemos plantearnos nuevos objetivos correspondientes a las diferencias de la arquitectura ALLIANCE y la implementación propuesta. Se debe poder comprobar el correcto funcionamiento de la implementación mediante un análisis de casos de prueba. Además de esto, también sería de interés utilizar estos casos para destacar ciertos comportamientos interesantes de la arquitectura y de ser necesario alterar la implementación para introducir mejoras.

1.2. Organización del documento

En el capítulo 2 del informe se dará un marco teórico al problema enfrentado, haciendo una introducción a la robótica y a las arquitecturas utilizadas. En el capítulo 3 veremos los antecedentes que se consideraron pertinentes a este informe, donde se verá Torocó y otras implementaciones de la arquitectura ALLIANCE que resultaron de interés. En el capítulo 4 nos enfocaremos en la arquitectura implementada, planteando algunos objetivos, que decisiones surgen de estos y cómo esas decisiones llevaron a la implementación de la arquitectura. En el capítulo 5 se presentan diferentes experimentos que fueron realizados a modo de verificar el correcto funcionamiento del sistema junto con el entorno de simulación utilizado. A continuación, en el capítulo 6 se presenta una extensión a la arquitectura con algunas mejoras que surgen a partir de lo visto en los experimentos realizados. El capítulo 7 dará conclusión a todo lo visto en el informe, planteando además algunos posibles trabajos futuros.

Se agregó además al final del informe un anexo con la interfaz de las nuevas funcionalidades de la biblioteca y un glosario el cual incluye términos utilizados a lo largo del informe, los cuales generalmente provienen de las arquitecturas mismas.

Capítulo 2

Marco teórico

Contents

2.1. Robots

Un robot es una máquina que fue programada para automatizar ciertos trabajos, los cuales suelen ser muy tediosos o peligrosos para un humano. No existe un consenso sobre que define exactamente a un robot, dado que sus actividades y funcionamientos pueden variar totalmente entre un robot y otro, generando que diferentes fuentes y autores utilicen sus propias definiciones. Para nuestros propósitos utilizaremos la definición propuesta por Ronald C. Arkin en su libro «Behavior Based Robotics» [Arkin, 1998], la cual dice: «Un robot inteligente es una máquina capaz de extraer información de su entorno y utilizar conocimiento del mundo para moverse de manera significativa y con un propósito».

La idea de máquinas automatizadas ha existido a lo largo de la mayoría de la historia humana, siendo parte de la mitología y religión de muchas culturas.

Figura 2.1: Vasija griega del año 450 A.C. representado la muerte de Talos, un robot de la mitología griega que protegía la isla de Creta de piratas. (Imagen de: Wikimedia Commons / Forzaruvo9)

A pesar de lo antigua que resulta la idea, los robots completamente autónomos recién comienzan a existir en la segunda mitad del siglo XX. Hoy en día el uso de robots industriales resulta muy común y se encargan de tareas de forma más barata, precisa y consistente que un humano. Los robots también se han comenzado a popularizar en el uso doméstico donde pueden cumplir con tareas como aspirar, planchar o cortar el pasto, entre otras actividades.

2.2. Paradigma reactivo

A lo largo de la historia de la robótica múltiples técnicas diferentes han sido utilizadas para los sistemas de control. Ronald C. Arkin [Arkin, 1998] define un espectro para estas técnicas en el cual los métodos que utilizan razonamiento deliberativo se encuentran en un extremo y los que usan control reactivo en el otro.

Los robots que utilizan razonamiento deliberativo requieren control sobre el conocimiento del entorno y utilizan este conocimiento para predecir el resultado de sus acciones, optimizando su rendimiento. El razonamiento deliberativo generalmente requiere asumir ciertas características del entorno, entre las cuales suele estar asumir que el conocimiento sobre el cual estamos razonando es consistente, confiable y certero. Si la información

Figura 2.2: Primera generación del robot aspiradora Roomba fabricado por iRobot. (iRobot)

que se está utilizando es errada o cambia desde el momento que se recibe, es probable que el resultado del razonamiento sea erróneo. Por este motivo es que en este tipo de sistemas se suelen utilizar representaciones del entorno basadas tanto en conocimiento previo del entorno como de la información recolectada por los sensores. Los sistemas de razonamiento deliberativo suelen cumplir con las siguientes características:

- Tienen una estructura jerárquica con una subdivisiones claramente definidas.
- La comunicación y el control ocurren de manera predecible y predeterminada, fluyendo hacia arriba y hacia abajo en la jerarquía, con poco o ningún movimiento lateral.
- Los niveles superiores en la jerarquía proporcionan subobjetivos para los niveles subordinados inferiores.
- El alcance de la planificación, tanto espacial como temporal, cambia durante el descenso en la jerarquía. Los requisitos de tiempo son más cortos y las consideraciones espaciales son más locales en los niveles inferiores.
- Dependen en gran medida de la representación simbólica del modelo del entorno.

Por otro lado, el control reactivo es una técnica que acopla la percepción y acción, típicamente en el contexto de conductas motoras, para producir respuestas más rápidas en entornos dinámicos y no estructurados. Se utilizan los datos más recientes obtenidos por los sensores para determinar qué acción el robot deberá realizar en cada instante. Estos tipos de sistemas cuentan con las siguientes características:

- Los comportamientos sirven como los bloques principales para crear las acciones del robot. Un comportamiento consiste típicamente de un par entrada/salida, donde el comportamiento recibe la información necesaria de los sensores o de otros comportamientos, y envía una señal a los actuadores del robot o a otros comportamientos dentro de la misma arquitectura.
- Se evita la abstracción de las lecturas de los sensores para generar una respuesta. En un sistema puramente reactivo el robot debería de actuar acorde a lo que detecte directamente del mundo exterior, evitando procesar esta entrada para crear

una representación del estado actual del entorno. Esto también elimina la etapa de planeamiento, en la cual se utilizan los datos obtenidos para planificar acciones futuras. Al evitar el procesamiento el robot ahorra tiempo y recursos, permitiéndole reaccionar más rápidamente a cambios impredecibles y potencialmente problemáticos.

- El comportamiento animal suele ser utilizado como base para este tipo de sistemas. La idea general es que los animales suelen realizar un procesamiento mínimo de su entorno, permitiéndoles reflejos muy rápidos ante cualquier tipo de situación, pero a su vez esto no evita que logren sus objetivos y tengan comportamientos que pueden ser muy complejos y funcionales.
- Los sistemas en un paradigma reactivo están diseñados para ser modulares desde el punto de vista del software. Los comportamientos que debe tener un robot se separan en diferentes módulos conectados, y se empiezan a implementar desde los más simples hasta los más complejos. Esto permite mayor simplicidad a la hora de desarrollar, y da facilidad al momento de agregar nuevos comportamientos más complejos a un robot.

Los sistemas puramente reactivos brindan varios beneficios, tanto para el comportamiento del robot como al momento de desarrollarlos y actualizarlos, pero también tienen sus limitaciones que hacen que en la práctica se suelen utilizar en conjunto con otros sistemas para agregar funcionalidades y robustez. Un claro ejemplo de sus limitaciones proviene de la falta de planes; por un lado esto le brinda al sistema mayor agilidad y disminuye el tiempo de respuesta ante cambios en el ambiente, pero también genera que el sistema carezca de memoria.

Si por ejemplo el robot tuviera como objetivo la planificación de rutas en determinado terreno, sería preferible que el robot recordara el terreno y planeara la mejor ruta acorde. Pero también sería conveniente que logre evitar obstáculos que puedan aparecer espontáneamente en su camino, y la reacción rápida que brinda un sistema reactivo resulta esencial en estos casos.

2.3. Robótica de enjambres

La robótica de enjambres es un área de la robótica para sistemas multi-robots que toma como principal inspiración los comportamientos sociales de los animales. Animales como las hormigas, las abejas, los pájaros y los peces son claros ejemplos de cómo a pesar de que individualmente son animales simples, un grupo de ellos puede llegar a tener comportamientos muy complejos. La idea principal de los sistemas de robótica de enjambres es emular este comportamiento animal. Mediante reglas simples e interacciones locales se puede diseñar sistemas robustos, escalables y flexibles para el manejo de un gran número de robots. Vemos entonces primero como definimos estas 3 características para sistemas de este tipo [Brambilla et al., 2013].

Que el sistema sea robusto implica que puede manejar con la pérdida de robots individuales sin comprometer el objetivo común. En el caso de los animales, la robustez se obtiene mediante la redundancia y falta de un líder en el grupo.

La escalabilidad por otro lado es la habilidad de actuar correctamente con diferentes tamaños de grupos. La introducción o pérdida de un un robot no debería resultar en un

cambio drástico en los resultados. En los animales la escalabilidad se logra dado que las capacidad de sensar y de comunicación son siempre locales.

Por último la flexibilidad es la habilidad de poder manejar una variedad de diferentes entornos y tareas. En el mundo animal la flexibilidad se logra mediante la redundancia, la simplicidad de las tareas y mecanismos para alocar estas tareas.

Las principales características de un sistema de enjambres de robots entonces son:

- Los robots son autónomos
- Los robots son situados en el entorno y pueden modificarlo
- Las capacidad de sensar y de comunicación de los robots son locales
- Los robots no tienen acceso a un sistema centralizado de control ni a información global
- Los robots cooperan para completar una tarea

Figura 2.3: Grupo de robots Colias, utilizados para replicar el comportamiento de enjambres de abejas. (Universidad de Lincoln)

La figura 2.3 muestra una imagen del robot Colias [Arvin et al., 2014], el cual es un ejemplo del uso de la robótica de enjambres. Este robot suele ser utilizado en enjambres de diferentes tamaños los cuales sirven, entre otras cosas, para simular colonias de abejas. El robot cuenta con una plataforma circular de 4 cm de diámetro y una velocidad máxima de 35 cm/s, lo cual le permite moverse rápidamente y funcionar adecuadamente en diferentes escenarios. También cuenta con un módulo infrarrojo que le permite comunicarse con vecinos cercanos en un radio de entre 0.5cm y 2m. El robot tiene las siguientes especificaciones:

- Procesador de placa superior
- Segundo procesador para gestión de energía y movimiento
- Ruedas de 2.2 cm de diámetro
- Sensores de proximidad
- Transmisores infrarrojos
- Decodificadores de infrarrojo

2.4. Arquitectura Subsumption

La Arquitectura Subsumption es una arquitectura de control basada en el paradigma reactivo propuesta por Rodney Brooks en 1986 [Brooks, 1986]. Esta arquitectura busca resolver algunos de los puntos que se consideran esenciales cuando se trabaja con un sistema del tipo reactivo:

- Múltiples objetivos: Es usual que un robot tenga múltiples objetivos que deba cumplir, y que varios de ellos entren en conflicto entre sí. A modo de ejemplo el robot podría tener como objetivo llegar a un lugar determinado consumiendo la menor cantidad de energía, pero a su vez también deba evitar las colisiones. En estos casos la importancia de cada una de las metas se vuelve dependiente al contexto actual, por lo tanto el sistema debe poder cambiar entre diferentes comportamientos rápidamente.
- Múltiples sensores: Es muy probable que el robot tenga múltiples sensores distintos, los cuales tienen cierto grado de error y es posible que den datos inconsistentes entre sí. El robot deberá poder tomar decisiones a pesar de estas condiciones.
- Robustez: Cuando alguno de los sensores falle el robot deberá adaptarse correctamente y continuar con su funcionamiento utilizando los sensores que continúan funcionando correctamente. A su vez, cuando el contexto cambie drásticamente el robot deberá adaptarse al cambio y mantener un funcionamiento adecuado en lugar de detener su funcionamiento.
- Extensibilidad: El sistema debe poder extenderse con cierta facilidad para aumentar la funcionalidades del robot.

Para resolver estos puntos Brooks planteó que el sistema de control debe ser simple y su complejidad no provenga de su implementación, sino de sus interacciones con el entorno. La arquitectura también deberá ser modular, permitiendo desarrollar diferentes comportamientos de forma independiente al resto y sea fácil de agregar nuevos comportamientos de ser necesario.

Típicamente en la implementación de robots móviles se separa el problema en diferentes capas de competencia: percepción, modelado, planeación, ejecución y control de los motores. Esta descomposición suele hacerse de forma horizontal, donde las capas crean una especie de cadena y los datos fluyen de una capa a la siguiente, empezando por los sensores y terminando en los actuadores, como puede observarse en la figura 2.4. Con una implementación de este tipo primero se necesitan implementar todas las capas antes de tener una versión funcional del robot, y si se desea hacer alteraciones a una de las capas hay que mantener su interfaz o alterar las capas adyacentes de modo de mantener el correcto funcionamiento.

Subsumption en cambio decide descomponer el problema a resolver en forma vertical. El problema se divide en base a los distintos comportamientos que el robot deberá cumplir. Las capas superiores indican un comportamiento más complejo y específico, mientras que las capas inferiores cumplen comportamientos más básicos que las capas superiores ya darán como resuelto, como puede ser mantener el equilibrio del robot o evitar colisiones. De esta forma se puede construir el programa de forma iterativa, empezando por las capas inferiores las cuales no tienen conocimiento de las superiores, y

Figura 2.4: Arquitectura tradicional

agregando capas para comportamientos más avanzados (figura 2.5) Todas las capas ejecutan de forma concurrente y permanecen activas en todo momento. Esto se ve ilustrado por la figura 2.6.

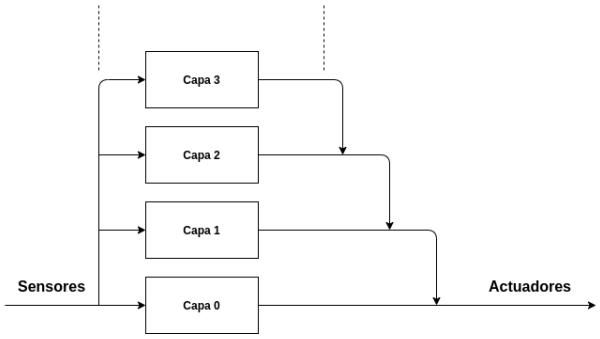


Figura 2.5: Arquitectura por capas

Cada capa de competencia se compone de múltiples módulos, los cuales funcionan como máquinas de estado independientes entre sí. Cada módulo tiene entradas que pueden provenir de los sensores o de otro módulo en la misma capas, y salidas que pueden ser utilizadas por otros módulos o actuadores.

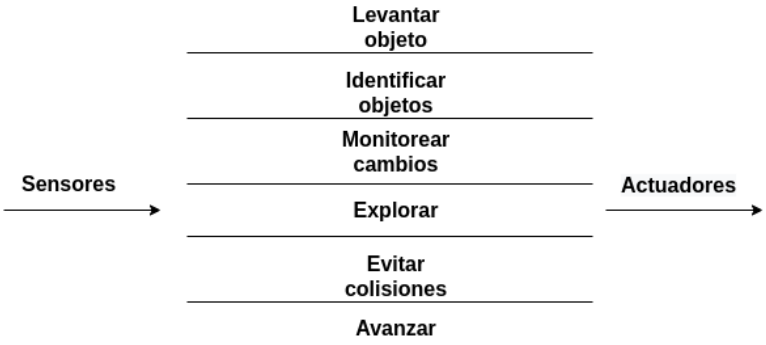


Figura 2.6: Arquitectura basada en comportamientos

La interacción entre capas se realiza mediante el intercambio de datos. Los módulos

de capas superiores pueden utilizar las salidas de los módulos de capas inferiores como entradas. Las capas superiores también pueden intervenir con la comunicación de los módulos de capas inferiores mediante la inhibición o supresión de los datos. Un módulo puede inhibir la salida de otro que pertenezca a una capa inferior, anulando su salida durante un periodo determinado de tiempo. La supresión en cambio le permite a un módulo sustituir la entrada de otro por determinado tiempo. También existe una tercera forma de interacción, en la cual se envía una señal de reinicio a otro módulo, volviendo a su estado inicial. La figura 2.7 permite visualizar como estas 3 operaciones interactúan sobre un módulo, donde las señales de supresión, inhibición y reinicio provienen de una capa superior.

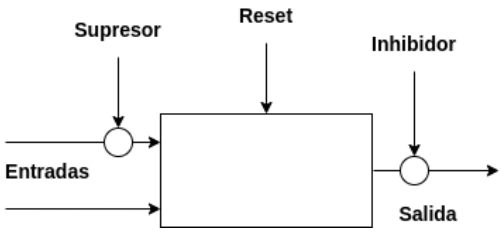


Figura 2.7: Supresión e inhibición de un módulo

Subsumption entonces logra cumplir con cada uno de los requisitos vistos anteriormente:

- Múltiples objetivos: Capas individuales pueden trabajar en objetivos individuales concurrentemente. El mecanismo de supresión de capas inferiores va a terminar decidiendo qué acciones son realizadas.
- Múltiples sensores: El problema generado por tener múltiples sensores mencionado anteriormente puede ser ignorado, dado que no todos los sensores van a ser utilizados por todos los módulos en cada momento. Cada módulo se encarga de utilizar las entradas de los sensores que desee de la forma que le parezca más conveniente.
- Robustez: Dado que las capas inferiores de la arquitectura, las cuales representan comportamientos más básicos, resultan las más probadas y no detienen su funcionamiento al agregar más capas. A su vez, como cada módulo trabaja de forma independiente la falla de un módulo no altera a los demás, especialmente si ocurre en capas superiores.
- Extensibilidad: La arquitectura en capas permite mucha flexibilidad al momento de la implementación, y agregar nuevas funcionalidades resulta tan simple como agregar una nueva capa al sistema. Todas las capas y módulos ya implementados y funcionando correctamente se pueden mantener sin necesidad de realizar ningún cambio.

La figura 2.8 muestra un ejemplo de la arquitectura con 3 capas. La capa de más abajo, «Evitar Obstáculos», se encarga de avanzar o detener al robot en caso que detecte que este va a colisionar con otro objeto. La capa de exploración permite que el robot se mueva en ausencia de obstáculos y logre moverse rápidamente para explorar su entorno.

Por último, la capa superior «Salir de situaciones difíciles», permite al robot invertir la dirección en espacios particularmente estrechos donde las capas inferiores no son lo suficientemente precisas como para evitar colisiones adecuadamente.

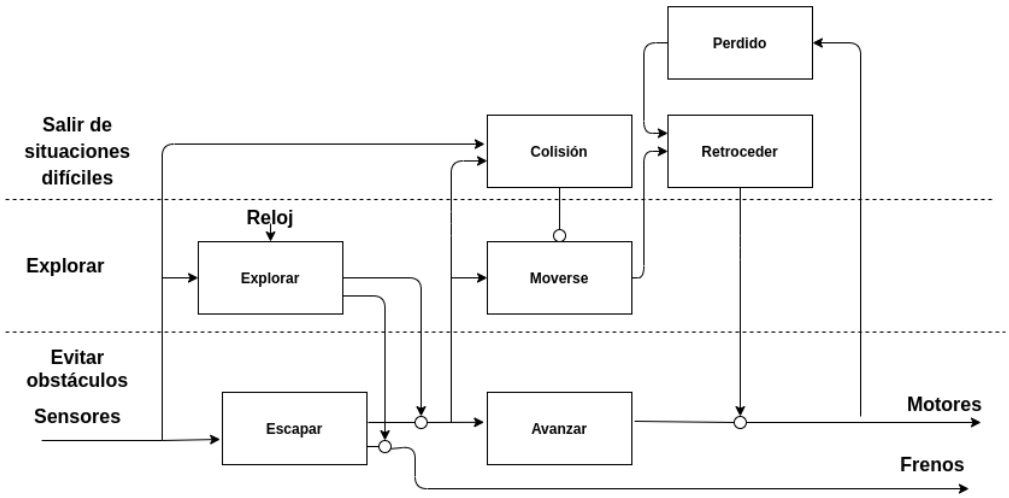


Figura 2.8: Ejemplo de una arquitectura basada en Subsumption

2.5. Arquitectura ALLIANCE

ALLIANCE es una arquitectura que facilita la implementación de sistemas de robots móviles que cooperan para lograr un conjunto de tareas generalmente independientes y en las cuales la tolerancia a posibles fallas resulta de gran importancia. ALLIANCE permite que dado un conjunto de robots, estos logren dividirse las tareas a realizar utilizando como datos los requerimientos de la misión, las condiciones ambientales actuales, la actividad que está realizando cada robot y el estado interno de cada robot. La arquitectura es a su vez totalmente distribuida, y por lo tanto cada robot cuenta con sus propias motivaciones para su comportamiento, y es responsable de adaptarse a medida que surjan cambios en el entorno.

Al momento de diseñar la arquitectura se plantearon algunas suposiciones que deben de cumplirse para su correcto funcionamiento. Las vemos a continuación:

- Los robots pueden detectar los efectos de sus propias acciones con una probabilidad mayor a 0.
- Los robots pueden detectar las acciones de los demás mediante algún medio de comunicación, como podría ser un broadcast.
- Los robots comparten un lenguaje común entre sí.
- Los robots no mienten ni tampoco tienen intenciones adversas a la misión, al menos no intencionalmente.

- El medio de comunicación entre robots no se encuentra necesariamente disponible en todo momento.
- Los sensores y actuadores de los robots no son perfectos y pueden dar fallas.
- Ante la falla de un robot, es posible que este no logre comunicar su falla al resto de los robots.
- No se cuenta con un sistema centralizado que pueda brindar información del entorno.

Se considera que las suposiciones planteadas por la arquitectura no son demasiado restrictivas y reflejan de forma bastante precisa la realidad de la mayoría de los ambientes en los cuales se podría querer utilizar un sistema con arquitectura ALLIANCE.

Cada robot en un arquitectura de tipo ALLIANCE se diseña basándose en los comportamientos que deberá cumplir. Dado esto, distintos conjuntos de comportamientos actúan de forma simultánea recibiendo las entradas de los sensores y enviando señales a los distintos actuadores. Los comportamientos de más bajo nivel se encargan de las acciones más primitivas, como pueden ser mantener el equilibrio o evitar colisiones, mientras que los de más alto nivel se encargan de comportamientos más complejos como pueden ser explorar el entorno o la recolección de objetos, de forma similar a lo ocurrido en Subsumption. Este enfoque funciona correctamente, pero tiene sus limitaciones. Cuando se tienen múltiples tareas que deben ser resueltas, estas compiten entre sí para decidir qué acciones toma el robot, dado que en general estas tareas no pueden ser resueltas simultáneamente. La figura 2.9 muestra la arquitectura general de ALLIANCE, utilizando un ejemplo con 2 *behavior sets*

Para resolver este problema ALLIANCE propone crear grupos de comportamientos, a los que llama *behavior sets*, y de los cuales solo uno puede estar activo a la vez. De esta forma se pueden tener varios *behavior sets*, cada uno definiendo un objetivo particular que el robot quiera cumplir.

2.5.1. Motivational behavior

Dado que un robot puede tener varios *behavior sets* distintos y estos pueden causar conflictos entre sí, se debe tener un mecanismo que le permita al robot elegir cual de estos *behavior sets* estará activo en cada momento y desactive los otros para evitar conflictos. Para esto, ALLIANCE define los *motivational behaviors*, los cuales se encargan de la activación de su *behavior set* correspondiente.

En todo momento cada *motivational behavio*r recibe un conjunto de entradas, las cuales incluyen información de los sensores, datos de la comunicación entre robots, inhibición de otros *motivational behaviors* y motivaciones internas llamadas impaciencia y consentimiento. La salida de un *motivational behavior* será entonces un valor numérico no negativo que tenga en cuenta todas estas entradas y representado el nivel de activación de su *behavior set* correspondiente. Esta salida es llamada la motivación de un *motivational behavior*. Cuando la motivación supera cierto nivel de activación entonces el *behavior set* se activa y el resto de los *behavior sets* son suprimidos.

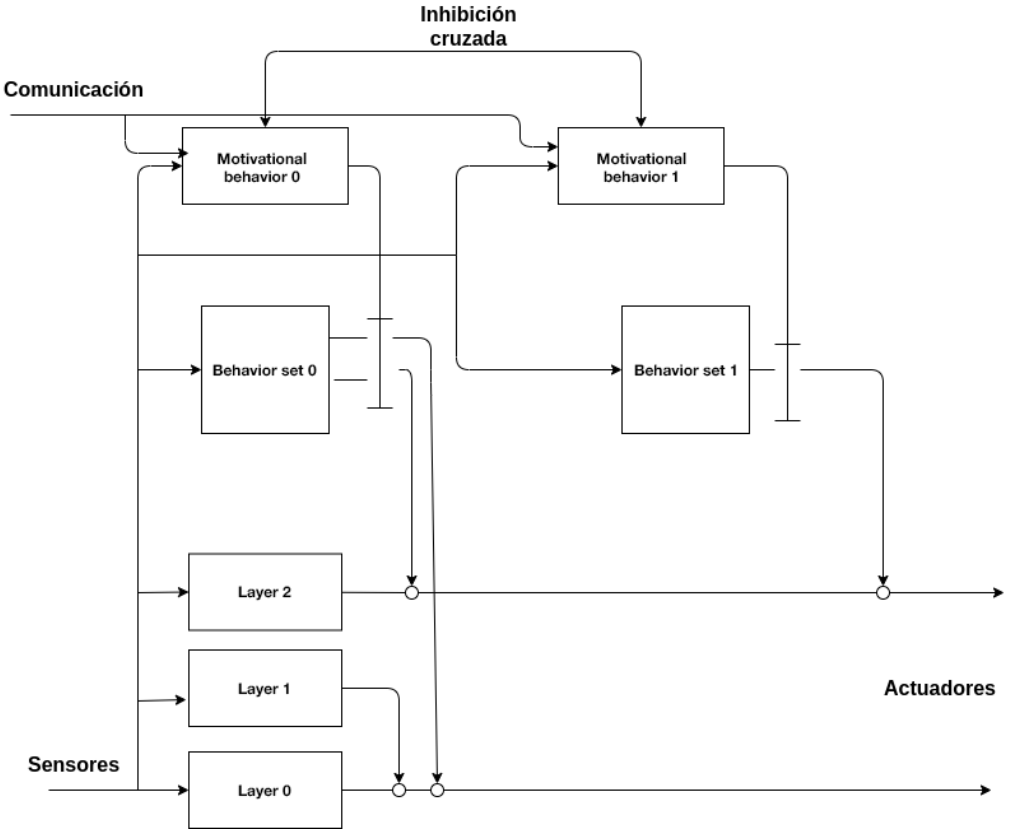


Figura 2.9: Arquitectura ALLIANCE

2.5.2. Formalización del modelo

Para la formalización del problema definiremos un conjunto $R = \{r_1, r_2, r_3...r_n\}$ de n robots heterogéneos que conforman el equipo, y el conjunto $T = \{task_1, task_2, ..., task_m\}$ de m diferentes tareas que conformarán la misión a cumplir. A su vez cada robot r_i puede disponer de varios *behavior sets* los cuales llamaremos $A_i = \{a_{i1}, a_{i2}, ...\}$. Además, como diferentes robots pueden tener diferentes maneras de resolver una misma tarea, debemos tener una forma de referenciar en qué tarea se encuentra trabajando un determinado robot. Para esto se definen un conjunto de funciones $\{h_1(a_{1k}), h_2(a_{2k}), ..., h_n(a_{nk})\}$ donde $h_i(a_{ik})$ retorna la tarea en la cual el robot r_i está trabajando al activarse el *behavior set* a_{ik}

Ahora que ya tenemos formalizado el modelo, veremos como se definen tanto el nivel de activación para un *motivational behavior* como los principales valores a ser utilizados para calcular la motivación.

2.5.2.1. Nivel de activación

El nivel de activación de un *behavior set* se configura con un parámetro θ y corresponde al nivel de motivación requerido para que un *behavior set* se active. Diferentes niveles de activación pueden ser configurados para diferentes behavior sets y diferentes robots, pero dado que con otras configuración se logra el mismo resultado, utilizar un mismo nivel para todos resulta igualmente válido.

2.5.2.2. Información sensorial

Los sensores le proveen al *motivational behavior* con información necesaria para determinar si la activación de un *behavior set* es necesaria o no. Con este propósito se propone la función $sensory\_feedback$:

$$sensory\_feedback_{ij}(t) = \begin{cases} 1 & \text{si la información sensorial del robot } r_i \text{ le indica en el} \\ & \text{momento } t \text{ que es válido activar el behavior set } a_{ij} \\ 0 & \text{en el caso contrario} \end{cases} \tag{2.1}$$

2.5.2.3. Comunicación entre robots

La comunicación entre robots se realiza por medio de broadcast y resulta imprescindible para que un robot pueda determinar el comportamiento actual de sus compañeros, además de permitirle reconocer la posible caída de otro robot o si algún otro robot se sumó al grupo.

Dos parámetros son utilizados para controlar la comunicación entre robots: ϖ_i y τ_i. El primer parámetro ϖ_i configura la frecuencia con la cual el robot r_i envía un mensaje notificando de su actividad actual. El segundo parámetro, τ_i provee cierta tolerancia a fallas, permitiendo configurar el tiempo que el robot r_i espera sin haber recibido mensajes de otro robot hasta que lo considera fuera de servicio. Para monitorear los mensajes recibidos se define la función *comm_received* de la siguiente manera:

$$comm\_received(i,k,j,t_1,t_2) = \begin{cases} 1 & \text{si el robot } r_i \text{ ha recibido un mensaje del robot} \\ & r_k \text{ referenciando la tarea } h_i(a_{ij}) \text{ en el intervalo} \\ & \text{de tiempo } (t_1,t_2) \text{ con } t_1 < t_2 \\ 0 & \text{en el caso contrario} \end{cases} \tag{2.2}$$

2.5.2.4. Supresión entre motivational behaviors

Es importante que un robot tenga solo un motivational behavior activado en cada momento para evitar conflictos entre sí. Para esto, cuando un motivational behavior se activa, este comienza a inhibir a los otros dentro del mismo robot para evitar que estos se activen. Eventualmente el robot completa su tarea o decide abandonarla, desactiva el behavior set correspondiente y dejando de inhibir a los otros motivational behaviors. Para modelar esta inhibición se utiliza la función *activity_supression*:

$$activity\_supression_{ij}(t) = \begin{cases} 0 & \text{si algún otro behavior set } a_{ik} \text{ se encuentra activo,} \\ & k \neq j, \text{ en el robot } r_i \text{ en el momento } t \\ 1 & \text{en el caso contrario} \end{cases} \tag{2.3}$$

2.5.2.5. Impaciencia

Para la implementación de la impaciencia se introducen tres nuevos parámetros: $\phi_{ij}(k,t)$, $\delta\_slow_{ij}(k,t)$ y $\delta\_fast_{ij}(t)$. El primer parámetro, $\phi_{ij}(k,t)$, representa el tiempo durante el cual el robot r_i va a tener en cuenta los mensajes del robot r_k al momento de que estos afecten el cálculo de la motivación para el behavior set a_{ij}.

Los siguientes parámetros, $\delta\_slow_{ij}(k,t)$ y $\delta\_fast_{ij}(t)$, configuran la velocidad con la cual va a crecer la motivación del behavior set a_{ij} en el robot r_i, tanto cuando otro robot r_k se encuentra realizando una tarea correspondiente al behavior set a_{ij} o cuando ningún robot lo está haciendo, respectivamente. Se asume que el valor de $\delta\_fast_{ij}(t)$ va a ser mayor y corresponder a una velocidad de crecimiento más rápido que la de $\delta\_slow_{ij}(k,t)$, dado que usualmente se le prefiere dar prioridad a las tareas que ningún otro robot se encuentra realizando. En el caso que se decida utilizar diferentes valores de δ_slow para diferentes robots, se opta por utilizar el mínimo. Dado estos parámetros, podemos definir el cálculo de la impaciencia como:

$$impatience_{ij}(t) = \begin{cases} min_k \delta\_slow_{ij}(k,t) & \text{si } comm\_received(i,k,j,t-\tau_i,t) = 1 \\ & \text{y } comm\_received(i,k,j,0,t-\phi_{ij}(k,t)) = 1 \\ \delta\_fast_{ij}(t) & \text{en el caso contrario} \end{cases} \tag{2.4}$$

Por último, también se desea que la motivación de un behavior set a_{ij} regrese a cero la primera vez que se recibe un mensaje de otro robot indicando que está realizando esta tarea, por lo tanto:

$$impatience\_reset_{ij}(t) = \begin{cases} 0 & \text{si } \exists k((comm\_received(i,k,j,t-\delta t,t) = 1) \text{ y} \\ & (comm\_received(i,k,j,t,t-\delta t) = 0)) \text{ donde } \delta t \text{ equi-} \\ & \text{vale al tiempo transcurrido desde el último chequeo} \\ & \text{de comunicación} \\ 1 & \text{en el caso contrario} \end{cases} \tag{2.5}$$

2.5.2.6. Consentimiento

ALLIANCE utiliza 2 parámetros para la configuración del consentimiento: $\psi_{ij}(t)$ y $\lambda_{ij}(t)$. El primero de estos parámetros, $\psi_{ij}(t)$, se encarga de configurar el tiempo que el robot r_i quiere mantener el behavior set a_{ij} antes de abandonarlo y dejárselo a otro robot. El segundo parámetro, $\lambda_{ij}(t)$, tiene un comportamiento similar, pero correspondiente al tiempo que espera el robot r_i antes de abandonarlo (a pesar de que no haya otro robot con ese comportamiento actualmente).

ALLIANCE entonces define *acquiescence* como:

$$acquiescence_{ij}(t) = \begin{cases} 0 & \text{si [el behavior set } a_{ij} \text{ del robot } r_i \text{ ha estado activo por} \\ & \text{más de } \psi_{ij}(t) \text{ unidades de tiempo en el momento t) y} \\ & \exists x.(comm\_received(i, x, j, t-\tau_i, t) = 1) \text{ o (el behavior} \\ & \text{set } a_{ij} \text{ del robot } r_i \text{ ha estado activo por más de } \lambda_{ij}(t) \\ & \text{unidades de tiempo en el momento t)]} \\ 1 & \text{en el caso contrario} \end{cases} \tag{2.6}$$

El consentimiento es entonces utilizado por ALLIANCE para que un robot pueda decidir por cuenta propia cuando abandonar una tarea, luego de haber intentado completarla durante un tiempo determinado. Esto evita que robots que sean incapaces o ineficientes realizando una tarea, eventualmente la abandonen para intentar otras que tal vez si puedan completar.

2.5.2.7. Motivación

Por último, ya habiendo visto cómo se calculan todos los datos a tener en cuenta, vemos el cálculo de la motivación:

$$\begin{aligned} m_{ij}(0) &= 0 \\ m_{ij}(t) &= [m_{ij}(t-1) + impatience_{ij}(t)] \\ &\times sensory\_feedback_{ij}(t) \\ &\times activity\_supression_{ij}(t) \\ &\times impatiences\_reset_{ij}(t) \\ &\times acquiescence_{ij}(t) \end{aligned} \tag{2.7}$$

En un principio la motivación del robot r_i para realizar el behavior set a_{ij} comienza en 0. Esta motivación incrementa al pasar el tiempo a no ser que ocurra una de las siguientes situaciones:

- La información sensorial indica que ese behavior set no es aplicable en este momento
- EL robot tiene otro behavior set activado
- Otro robot acaba de tomar esta misma tarea por primera vez
- El robot ha decidido abandonar la tarea

La motivación entonces crece hasta que supera el nivel de activación y el behavior set correspondiente toma control sobre el robot.

2.5.3. Ejemplo

La figura 2.10 muestra un ejemplo de la arquitectura de un sistema que implementa la arquitectura ALLIANCE, en el cual los robots se encargan de la limpieza de residuos tóxicos.

En la capa inferior se tiene el *behavior* «Evitar Colisiones» el cual no pertenece a ningún *behavior sets*, por lo tanto está activo en todo momento independientemente del *behavior set* que se encuentre activo. Por otro lado tenemos 3 *behavior sets* distintos con sus correspondientes *motivational behaviors*. El primero de ellos, «Explorar», se encarga de buscar áreas en las que exista residuo que el robot deba limpiar. Al encontrar estas áreas el $sensory\_feedback$ le deberá indicar al robot que este comportamiento ya no es necesario y desactivarlo para dar lugar a otros. «Limpiar» es el *behavior set* encargado de efectuar la limpieza y solo podrá ser activado cuando el robot se encuentra en un área que deba ser limpiada. Cuando el robot termine de limpiar un área el *behavior set* será desactivado por su *motivational behavior* y dará lugar a que se vuelva a activar «Explorar» para encontrar otra área de limpieza, o que se active «Reportar Progreso» para que el robot reporte el avance de la misión.

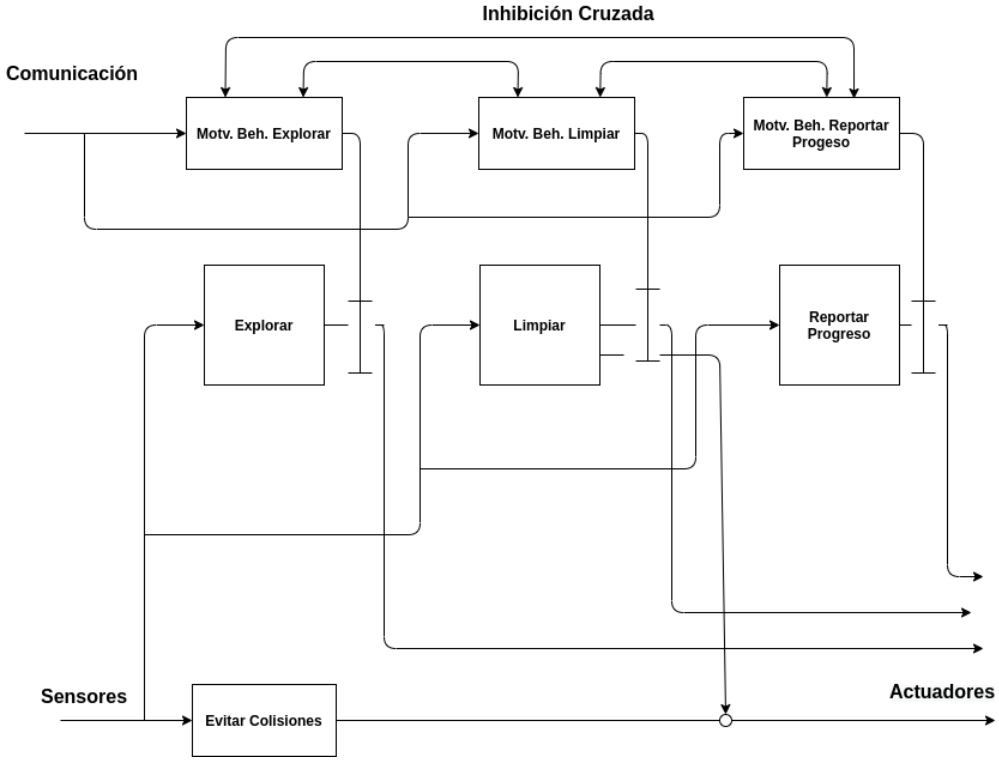


Figura 2.10: Ejemplo de la arquitectura ALLIANCE.

Capítulo 3

Trabajos previos

Contents

En esta sección se le dará un enfoque a los proyectos que preceden el trabajo realizado en este informe. Se verá la biblioteca Torocó, la cual forma la base de la implementación que se propone y sobre la cual expandimos para llegar a una versión de ALLIANCE. También se mencionan con diferentes grados de detalle otras implementaciones de la arquitectura ALLIANCE que se consideraron de interés.

3.1. Torocó

Como base para la implementación de Alliance propuesta en este informe se utilizó la biblioteca Torocó, creada por Ignacio Bettosini y Agustín Clavelli en el 2015 como proyecto de grado en la Facultad de Ingeniería [Bettosini and Clavelli, 2015]. Torocó brinda una implementación en el lenguaje Lua de la arquitectura Subsumtion, brindando las siguientes funcionalidades:

- Manejo de comportamientos.
- Comunicación con dispositivos de *hardward*, tanto con sensores como actuadores.
- Comunicación entre comportamientos.
- Comunicación con sistemas remotos mediante protocolos TCP o UDP.
- Supresión e inhibición entre comportamientos

El objetivo de Torocó es brindar una implementación de Subsumtion que permita buen rendimiento y que a su vez resulte sencilla de utilizar y flexible desde el punto de vista del usuario del sistema. A continuación veremos los requisitos funcionales y no funcionales planteados para Torocó, y cómo se construyó la arquitectura a modo de brindar las funcionalidades mencionadas anteriormente.

3.1.1. Objetivos

Torocó permite definir los comportamientos del robot, donde cada uno se encarga de una tarea determinada. Estos comportamientos pueden comunicarse con los sensores y actuadores del robot, para de esta forma poder completar la misión definida. Los comportamientos deben ejecutarse en forma concurrente, lo cual implica que deberán ser capaces de recibir, procesar y enviar datos en todo momento. Torocó se encarga de brindar el mecanismo para el manejo de la comunicación entre componentes

3.1.2. Requisitos funcionales

3.1.2.1. Comportamientos

La arquitectura del sistema de control está basada en comportamientos los cuales pueden almacenar y procesar datos, así como comunicarse con dispositivos de hardware u otros comportamientos. Los comportamientos del sistema se ejecutan concurrentemente.

3.1.2.2. 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.

3.1.2.3. Comunicación entre comportamientos

Los comportamientos podrán comunicarse entre sí para intercambiar datos.

3.1.2.4. Comunicación con sistemas remotos

El sistema podrá comunicarse con sistemas remotos utilizando conexiones TCP o UDP.

3.1.2.5. Comunicación por eventos

Los comportamientos tendrán comunicación por eventos, incluyendo los siguientes funcionamientos:

- Suscribirse a eventos para detectar cambios en la salida de otros comportamientos o sensores.
- Detener su ejecución esperando a que ocurra un evento determinado.
- Emitir señales ante eventos que ocurren internamente.

3.1.2.6. Supresión e inhibición

El sistema ofrecerá mecanismos de inhibición y supresión de comportamientos. 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.

3.1.3. Requisitos no funcionales

Torocó propone los siguientes requisitos no funcionales:

- Los comportamientos deberán ejecutarse en forma concurrente.
- El sistema deberá ejecutar sobre Linux.
- El sistema se implementará sobre el *framework* Toribio [Visca, 2012b] en el lenguaje Lua, versión 5.2.3.
- El sistema funcionará con los sensores de Usb4Butiá, que están basados en polling.
- El robot de prueba utilizará como placa de control una *single board* de propósito general, de al menos 64 MB de RAM.

3.1.4. Implementación

3.1.4.1. Tipo de interfaz

Torocó brinda una interfaz en forma de una biblioteca de Lua, la cual ofrece un conjunto de funcionalidades que el programador puede utilizar cuando desee en su programa. Se ofrece a Torocó en formato de biblioteca para que ofrezca mayor libertad al programador del sistema y sea necesario invertir el menor tiempo posible en el proceso de desarrollo.

3.1.4.2. Interacción del sistema

Para las interacciones con el entorno, los objetos, otros robots y sistemas remotos, el sistema debe responder 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.
- Sensores: 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.

3.1.4.3. Componentes

El sistema consiste en un conjunto de comportamientos, sensores y actuadores que se comunican por medio de eventos. A su vez, cada comportamiento contiene un conjunto de corrutinas ejecutando en paralelo. Torocó se encarga de manejar los eventos del sistema, teniendo en cuenta los mecanismos de inhibición y supresión.

La comunicación con el *hardward* es manejada por la biblioteca Toribio, mientras que el manejo de eventos se maneja a través de la biblioteca Lumen [Visca, 2012a], la cual es utilizada tanto por Torocó como por Toribio.

Para la implementación de un sistema utilizando Torocó, el usuario deberá implementar los comportamientos del robot en conjunto con un programa principal, el cual se encarga de configurar la comunicación entre componentes e iniciar el sistema.

Los comportamientos pueden ser implementados utilizando *triggers* y corrutinas. Para el caso de los *triggers*, una vez que el comportamiento recibe el evento de entrada, la función correspondiente al *handler* es ejecutada. En el caso de las corrutinas, estas se ejecutan al iniciar la ejecución del sistema.

Cada comportamiento puede conectar un evento a una sola entrada, siendo las entradas de todas las corrutinas del comportamiento iguales. Esto evita que un usuario suprima un evento para una corrutina del comportamiento y deje habilitado el mismo evento para otra. A su vez la función de emitir las salidas del comportamiento envía todas simultáneamente.

Cada entrada de un comportamiento se conecta a un conjunto de eventos de salida, ordenados por prioridad. Los eventos se conectan con módulos supresores en serie, sin permitir otras formas. 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.

3.1.4.4. 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 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.

Toda comunicación interna al sistema es realizada mediante eventos. Los comportamientos deben definir sus entradas y salidas, donde cada entrada puede registrarse a uno o más eventos de salida. Por su parte, los comportamientos también pueden emitir eventos de salida, los cuales pueden ser recibidos por otros comportamientos o por los mismos actuadores. El comportamiento es muy parecido para los actuadores y sensores, donde los actuadores definen eventos de entrada y los sensores eventos de salida. Torocó brinda dos formas de emitir los eventos de salida: emitir el evento una única vez o fijarlo de manera permanente.

Torocó usa los eventos y funciones de los *devices* de manera unificada, por lo que los comportamientos pueden usar ambos mecanismos de manera indistinta. Para cada función de los sensores el sistema genera una tarea que se encarga de hacer polling, y cuando la función devuelve un valor nuevo, se envía un evento con dicho valor a los receptores. Esto resulta menos eficiente que usar eventos, pero permite el funcionamiento con cualquier tipo de dispositivos y no solo los que se manejan por eventos. En el caso de los actuadores, Torocó genera una tarea que captura los eventos emitidos al actuador e invoca a la función con dichos valores como parámetros.

También se le permite al usuario fijar eventos de salida los cuales se manejan de forma independiente del tiempo de inhibición y supresión. 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.

3.1.4.5. Inhibición y supresión

Los eventos emiten salidas genéricas, y al definir la interconexión entre comportamientos se indica que las salidas de un comportamiento sirven para inhibir o suprimir a otro. El código del comportamiento no indica qué otros comportamientos debe inhibir o suprimir, sino que eso 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.

La inhibición consiste en anular un evento de salida de un emisor. Para cada salida de un comportamiento o *device*, se puede especificar un conjunto de eventos supresores, y cuando se emite alguno de estos eventos la salida queda suprimida. La supresión en cambio consiste en anular los datos de entrada de un receptor, y sustituirlos por otros provistos por el supresor.

3.2. Contribución al diseño de sistemas multi robots utilizando ALLIANCE

En noviembre del 2004 Gonzalo Daniel Tejera López publicó «Contribución al diseño de sistemas multi robots utilizando ALLIANCE» [López, 2004] como su tesis de maestría. En su tesis, Tejera implementó la arquitectura ALLIANCE en JAVA pero al hacerlo encontró algunas inconsistencias con las definiciones formales planteadas por Parker. Para solucionar esto plantea la arquitectura N-ALLIANCE, la cual propone brindar mayor flexibilidad que ALLIANCE, y que su sucesora L-ALLIANCE, manteniendo al mismo tiempo las principales virtudes de estas arquitecturas. En esta sección veremos algunas de las críticas que se realizan a la arquitectura ALLIANCE y las soluciones propuestas para la mejora de la arquitectura.

La figura 3.1 muestra una imagen del robot Khepera [K-Team, 1999], el cual fue utilizado en el entorno de simulador YAKS[Zaxmy, 2003] para probar la arquitectura N-ALLIANCE.

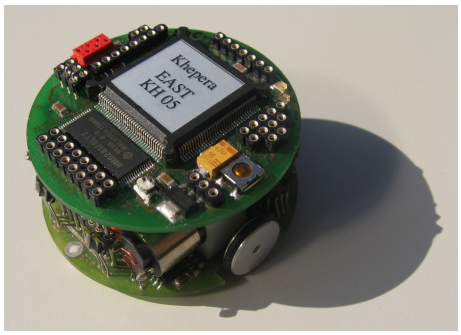


Figura 3.1: Primera generación del robot Khepera.

3.2.1. Ambigüedades y contradicciones

En la mayoría de las cosas, las ambigüedades que Tejera señala surgen de diferencias entre la descripción en lenguaje natural usada por Parker y la definición formal de las funciones propuestas para ALLIANCE. A pesar de que también se proponen mejoras a la arquitectura L-ALLIANCE, estas no serán mencionadas en este informe dado que el enfoque se da únicamente en la arquitectura ALLIANCE.

3.2.1.1. Comunicación entre robots

Una de las primeras contradicciones surge a partir de la definición dada para la función *comm_received*. Según propone Parker: «each motivational behavior a_{ij} of robot r_i must also note when a team memeber is pursuing task $h_i(a_{ij})$)», pero la definición formal de la función *comm_received* indica que esta devuelve 1 cuando robot r_i recibió un mensaje del robot r_k respecto a la tarea $h_i(a_{ij})$) en el intervalo de tiempo (t_1, t_2). Tejera entonces nos plantea la siguiente pregunta: «La función *comm_received* debe indicar si en algún momento el robot trabajó en la tarea o si el robot está trabajando en la tarea?». La

definición formal (ecuación 2.2) utiliza un rango de tiempo determinado, lo cual indica si el robot trabajó en la tarea, mientras que la explicación de Parker habla de la tarea que se está ejecutando actualmente.

Para que las dos velocidades de crecimiento de la motivación tengan sentido, se debe optar por la definición utilizada en el modelo formal, que utiliza un rango de tiempo dado.

Al optar por la interpretación de *comm_received* que maneja intervalos de tiempo, se presenta un conflicto con la definición de *impatience_reset* propuesta, dado que esta permite que la falla de un robot lleve a cero ilimitadas veces la motivación de otros robots, comprometiendo así la misión.

Para ilustrar este escenario se da el ejemplo de un robot r_i el cual activa la tarea w, envía el mensaje correspondiente a su activación y luego debido a una falla la desactiva inmediatamente. En este escenario todos los robots que recibieron ese mensaje bajan la motivación para esa tarea a 0 debido al *impatience_reset*. Si esto se repite por siempre se evita que otros robots del equipo puedan ejecutar la tarea w.

3.2.1.2. Reinicio de impaciencia

Para la función *impatience_reset* (2.5), Tejera propone agregar la condición de que si un robot recibe sus propias notificaciones, esto no reinicia la impaciencia. Para esto redefine la función de la siguiente manera:

$$impatience\_reset_{ij}(t) = \begin{cases} 0 & \text{si } \exists k(k \neq i)((comm\_received(i,k,j,t-\delta t,t) = 1) \text{ y} \\ & (comm\_received(i,k,j,t,t-\delta t) = 0)) \text{ donde } \delta t \text{ equi-} \\ & \text{vale al tiempo transcurrido desde el último chequeo} \\ & \text{de comunicación} \\ 1 & \text{en el caso contrario} \end{cases} \tag{3.1}$$

Además, en caso de que varios robots decidan activar el mismo *behavior set* en el mismo margen de tiempo, todos lo desactivaran inmediatamente dado que recibirán mensajes del resto, lo cual interfiere con el rendimiento del equipo de robots. Para resolver esto opta por reescribir la función de la siguiente manera:

$$impatience\_reset_{ij}(t) = \begin{cases} 0 & \text{si } \exists k(k \neq i)((k < i) \text{ o } ((k > i) \text{ y } (m_{ij}(t-1) + \\ & impatience_{ij}(t) < \theta))) \text{ y } ((comm\_received(i,k,j,t - \\ & \delta t,t) = 1) \text{ y } (comm\_received(i,k,j,t,t-\delta t) = 0)) \\ & \text{donde } \delta t \text{ equivale al tiempo transcurrido desde el últi-} \\ & \text{mo chequeo de comunicación} \\ 1 & \text{en el caso contrario} \end{cases} \tag{3.2}$$

Con este cambio, si sucediera que varios robots activan la misma tarea simultáneamente, se prioriza la activación de los comportamientos de alto nivel para los robots con menor identificador, evitando conflictos.

3.2.1.3. Consentimiento

La definición de la función de consentimiento (2.6) presenta un problema similar al visto anteriormente, donde los mensajes recibidos por el mismo robot pueden presentar inconvenientes. Para esto se redefine la función de consentimiento:

$$acquiescence_{ij}(t) = \begin{cases} 0 & \text{si [(el behavior set } a_{ij} \text{ del robot } r_i \text{ ha estado activo por más de } \psi_{ij}(t) \text{ unidades de tiempo en el momento t) y } \exists x.(x \neq i)(comm\_received(i, x, j, t - \tau_i, t) = 1) \text{ o (el behavior set } a_{ij} \text{ del robot } r_i \text{ ha estado activo por más de } \lambda_{ij}(t) \text{ unidades de tiempo en el momento t)]} \\ 1 & \text{en el caso contrario} \end{cases} \tag{3.3}$$

3.3. Implementing ALLIANCE in networked robots using mobile agents

En este estudio, Liam Cragg y Huosheng Hu [Cragg and Hu, 2004] buscan utilizar agentes móviles para extender las funcionalidades de ALLIANCE. Veremos una breve introducción a agentes móviles y cómo estos fueron utilizados para expandir la arquitectura.

3.3.1. Agentes Móviles

Los agentes móviles existen dentro de un entorno de ejecución que provee de múltiples servidores de agentes, los cuales proveen áreas en las cuales los agentes móviles pueden residir y una interfaz para que accedan a diferentes funcionalidades. El agente móvil contiene el conocimiento y resultados de sus tareas anteriores, pero no los recursos necesarios para realizar una nueva tarea, los cuales necesita obtener del servidor. En vez de transferir información entre servidores, el agente móvil se mueve el mismo entre servidores para poder acceder a sus recursos. Cuando el agente termina con su tarea puede permanecer en el mismo servidor o moverse a otro, llevando consigo el conocimiento que ya tiene.

La figure 3.2 representa un sistema con agentes móviles, en donde se pueden observar 3 diferentes hosts ejecutando servidores entre los cuales los agentes móviles pueden moverse.

3.3.2. Arquitectura

En esta arquitectura se plantea entonces ejecutar los motivational behaviors, behavior sets, la comunicación entre robots y la funcionalidad base del agente móvil en hilos separados. El hilo de motivational behaviors se encarga del cálculo de la motivación cada 100 ms y el hilo de los behavior sets contiene el código para la ejecución de el behavior set seleccionado.

El hilo de comunicación interactúa con una nueva forma de agente móvil la cual es llamada ALLIANCE Communication Agent (AComm). En cada ciclo el hilo de comunicación de cada robot crea un único AComm. Este nuevo agente es creado con una

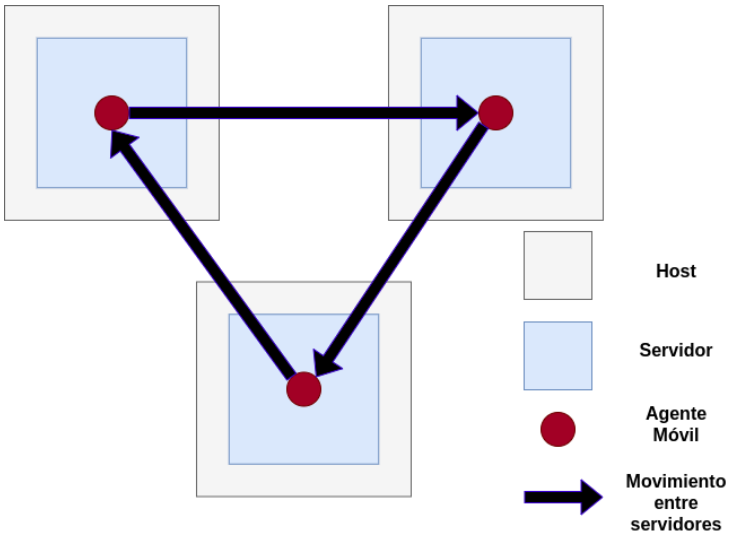


Figura 3.2: Representación de un sistema con agentes móviles.

lista de direcciones con las áreas de ejecución de todos los robots conocidos y el ID y *behavior set* activado del robot que lo creó. Una vez creado, el agente automáticamente crea una copia de sí mismo en todos los otros robots de la lista y se remueve del robot que lo creó.

Luego de creado el AComm, el hilo de comunicación usa el servidor para localizar e interactuar localmente con cualquier AComm que hayan sido copiados al robot, obteniéndose así información sobre la actividad del resto del equipo. Luego de proveer de esta información, los AComms se remueven a sí mismos del robot.

3.3.3. Resultados obtenidos

Tras realizar experimentos comparando la arquitectura propuesta con una implementación más tradicional de ALLIANCE, se concluye que hay una ventaja al utilizar agentes móviles, principalmente al momento de actualizar a los robots de manera más rápida. De esta forma el desarrollo de un sistema puede ser realizado más rápidamente, permitiendo descargar y probar nuevos algoritmos en un grupo de robots en tiempo de ejecución.

3.4. Implementing and Simulating an ALLIANCE-based Multi-robot Task Allocation Architecture Using ROS

En este trabajo [dos Reis and Bastos, 2016] Wallace Pereira Neves dos Reis y Guilherme Sousa Bastos proponen una implementación de la arquitectura ALLIANCE sobre ROS (Robot Operating System) [Open Robotics, 2007].

ROS es un *framework* open-source para el desarrollo de robots que apunta a resolver ciertos problemas que surgen al desarrollar sistemas robóticos a gran escala. Dentro de las premisas del *framework*, se destacan sus características multi-lenguaje y un enfoque en el uso de herramientas específicas para cada aplicación.

Nodos, mensajes, tópicos y servicios son los conceptos fundamentales de la implementación de ROS. Un sistema ROS es comúnmente compuesto de muchos nodos, los cuales se comunican entre sí mediante mensajes. Los nodos publican mensajes en los tópicos para luego ser recibidos por los otros nodos que estén suscritos a este. Por otro lado, también están los servicios, que funcionan de forma similar a un servidor web, con mensajes de consulta y mensaje de respuesta.

En la arquitectura que se plantea, los elementos y funcionalidades de ALLIANCE se mantienen tal cual son, pero se utilizan algunas de las herramientas de ROS para facilitar su implementación, principalmente en términos de la comunicación entre *behaviors* y entre robots.

La comunicación entre *behaviors* se maneja mediante tópicos y mensajes de ROS, de esta forma para configurar su salida un *behavior* solo deberá enviar un mensaje a un tópico de ROS, y de esta forma todos los otros *behaviors* que se que suscribieron podrán recibir el mensaje. El manejo de los sensores se maneja de forma análoga, donde los sensores utilizan un tópico para enviar la información que reciben. La comunicación entre robots también es otro punto de la arquitectura en el cual se utilizan los tópicos y mensajes brindados por ROS. La arquitectura define el tópico *TaskAllocation* el cual todos los robots usarán para enviar mensajes incluyendo un identificador del robot y el *behavior set* que tenga activo, y a su vez al cual también deberán estar suscritos para recibir los mensajes de los otros robots.

La figura 3.3 muestra el robot Amigobot [Adept MobileRobots, 2003], el cual fue utilizado en las pruebas de este trabajo debido a su compatibilidad con ROS.

Figura 3.3: Robot Amigobot, diseñado por Adept MobileRobots.

Según los autores, una de las ventajas que brinda este enfoque es que facilita la implementación de la comunicación entre robots, la cual es manejada utilizando la interfaz que brinda ROS. Esto también permite poder escalar fácilmente el equipo agregando más robots sin la necesidad de realizar cambios en el código de los robots.

3.5. Conclusiones

En este capítulo vimos algunos trabajos previos relacionados a lo visto en este informe, los cuales se consideraron de interés tanto para dar contexto a lo que se verá más adelante, como para ver otras implementaciones de ALLIANCE y la variedad de usos de la arquitectura.

El análisis de Torocó (Sec. 3.1) sirve no solo para dar contexto histórico al trabajo que presentaremos en el capítulo 4, sino también para entender las decisiones que fueron tomadas al momento de su implementación. Este conocimiento sirve para educar las decisiones de la implementación de ALLIANCE y así lograr mantener los beneficios que brinda Torocó.

A pesar de que las mejoras propuestas por Tejera (Sec. 3.2) no fueron implementadas en este trabajo, las ideas planteadas y sus críticas a la arquitectura ALLIANCE ayudan a reconsiderar algunas de las definiciones planteadas por Parker y destacar algunos de los puntos débiles de la arquitectura.

Por último los trabajos de ALLIANCE sobre ROS (Sec. 3.4) y el uso de ALLIANCE con agentes móviles (Sec. 3.3) abren la puerta a pensar diferentes usos de la arquitectura ALLIANCE y como esta puede ser utilizada en una variedad de sistemas distintos.

Capítulo 4

Alliance sobre Torocó

Contents

En este capítulo se presenta una nueva implementación de la arquitectura ALLIANCE. Con esta implementación se propone mantener los beneficios brindados por Torocó, analizados en la Sec. 3.1, agregando las funcionalidades de la arquitectura ALLIANCE presentadas en la Sec. 2.5 para el manejo de diferentes objetivos y la división de tareas en sistemas de múltiples robots.

Se verán las decisiones tomadas y su justificación, así como las funcionalidades agregadas a Torocó para poder llegar a una implementación satisfactoria de ALLIANCE. A modo de hacer la implementación pública y accesible al mundo se hace disponible un repositorio público [1] mediante la plataforma Gitlab [Software Freedom Conservancy, 2014].

4.1. Decisiones

Al momento de tomar decisiones se trató de tener presente en todo momento los objetivos de Torocó, para así poder mantenerlos y construir sobre ellos. Torocó se enfoca en brindar una implementación que sea tanto buena en términos de rendimiento como amigable con el programador, y fueron en estos puntos en los que se tuvo énfasis al momento de agregar nuevas funcionalidades.

Se buscó además un balance entre brindar una mayor flexibilidad al programador y que la biblioteca realice el trabajo que resulta más tedioso. En la mayoría de los casos cuando fue necesario elegir entre ellas, se prefirió facilitarle el trabajo al programador, brindando más funcionalidades con la menor cantidad de configuración y trabajo de su lado. A continuación veremos las decisiones tomadas, junto a los argumentos empleados para tomarlas.

4.1.1. Inhibición cruzada

Como se vio anteriormente, un robot en la arquitectura ALLIANCE solo puede tener un behavior set encendido al mismo tiempo, y todos los behaviors pertenecientes a otros behavior sets deben ser completamente suprimidos hasta el momento que se active su behavior set. Siendo estrictos con el planteamiento de la arquitectura los motivational behaviors deberían comunicarse entre sí con un sistema de inhibición cruzada, donde el motivational behavior activo se encarga de inhibir al resto hasta que termina su tarea actual, decida rendirse o ceder la tarea a otro robot.

Dado que la inhibición cruzada entre *motivational behaviors* es algo propio de la arquitectura misma y no debería variar entre implementaciones se decidió que la biblioteca debería encargarse de esto sin la necesidad de ser implementado por el programador. Esto se puede ver como una limitación dado que saca cierto control al programador, pero cambiar el funcionamiento de la inhibición cruzada entre *motivational behaviors* sería un cambio importante respecto a las ideas originales de la arquitectura ALLIANCE.

4.1.2. Comunicación entre robots

Para la comunicación entre robots se consideró adecuado que la biblioteca se encargara de la mayoría del trabajo, dejando a cargo del programador solo lo que se considera

[1] `https://gitlab.fing.edu.uy/mauro.mottini/alliance`.

completamente necesario. Con este propósito se considera que la biblioteca debería encargarse de los siguientes funcionamientos:

- Decidir cuándo enviar un mensaje.
- Guardar y manejar la información de los mensajes recibidos (de qué robot, que *motivational behavior* tiene activo, cuando llegó el mensaje, etc.)

El programador entonces solo deberá encargarse de brindar una forma para enviar y recibir los mensajes, y de cómo procesar estos mensajes para poder codificar y decodificar la información contenida en cada uno.

También se consideró que la biblioteca se encargará de manejar el formato de los mensajes, dejando en manos del programador solamente la elección del método de comunicación, sin tener que preocuparse por el contenido de los mensajes. Sin embargo, a pesar que agregaría comodidad, se terminó decidiendo en contra de esto porque podría limitar posibilidades.

Por un lado se podría querer aprovechar el sistema de comunicación para enviar más información que la utilizada por la biblioteca en un mismo mensaje. De obligar un formato específico esto podría generar la necesidad de enviar otros mensajes para la información no específica a la biblioteca, posiblemente creando cierta congestión en el medio de comunicación. Además, dado que a priori no se tiene ninguna información de qué medio de comunicación va a ser utilizado, forzar un formato específico podría causar problemas. Se podría por ejemplo estar limitado por un máximo tamaño de mensaje y que los mensajes generados por la biblioteca no puedan ser enviados, o se podría querer que estos se encuentren encriptados por temas de seguridad.

4.1.3. Motivational behaviors

Como vimos anteriormente la inhibición cruzada de los motivational behaviors ya viene incorporada dentro de las funcionalidades de la biblioteca, por lo tanto cada motivational behavior solo debe encargarse de calcular su motivación. Dado que la comunicación entre robots también se encuentra en su mayoría manejada por la biblioteca, se analizó entonces cada una de las componentes que son utilizadas para calcular la motivación para decidir cuáles de ellas deberían ser manejadas por la biblioteca y de cuales deberá encargarse el programador.

Para el caso de la impaciencia (ver Ec. 2.4) está solo utiliza la comunicación entre robots, de la cual se encarga la biblioteca, para decidir si crecer utilizando la velocidad rápida o lenta. En este caso se consideró que lo más conveniente sería pasarle estas velocidades a la biblioteca y que ella maneje el cálculo de la impaciencia utilizando estos valores.

Algo similar ocurre con el consentimiento, como la biblioteca maneja la comunicación y el cambio de behavior set, tiene todos los datos necesarios para poder calcularlo correctamente. El programador solo debería proveer los valores para el tiempo que un robot quiere mantener una determinada tarea ($\psi_{ij}(t)$) y el tiempo que deberá esperar antes de cederle la tarea otro robot ($\lambda_{ij}(t)$).

Por último se decidió configurar el nivel de activación con el valor 100, pero darle la flexibilidad al programador para poder cambiarlo. Se optó por tener un solo nivel de activación para todos los *motivational behaviors* porque resulta más simple pero

a la vez no remueve funcionalidades, dado que como indica Parker [Parker, 1994], se puede controlar cómo crece la impaciencia en cada motivational behavior, obteniéndose el mismo resultado que con diferentes niveles de activación.

Teniendo esto en cuenta, al momento de crear un *motivational behavior* solo se debe proporcionar:

- Su *behavior set* correspondiente
- Su velocidad de impaciencia rápida y lenta
- El tiempo durante el cual deja que la comunicación con otros robots lo afecte
- El tiempo por el cual quiere mantener la tarea actual
- El tiempo máximo que puede invertir en una tarea antes de cederle la tarea a otro robot
- Una función para calcular el *sensory feedback*

Con todos estos valores proporcionados la biblioteca se encarga del cálculo de la motivación y de la activación de los *behavior sets*.

4.2. Implementación

Veremos ahora cómo se llevaron a cabo las decisiones vistas anteriormente en términos de la implementación de la arquitectura.

El mayor cambio en términos de la arquitectura es la introducción de un nuevo elemento al cual llamamos *coordinator* y es implementado como una corrutina que por defecto se ejecuta una vez por segundo. Este valor fue elegido porque se considera que es un margen de tiempo suficiente como para que la mayoría de los hardwares puedan calcular la motivación y enviar un mensaje al resto de robots en caso de ser necesario. Al momento de cambiar este valor se debe tener en cuenta que el límite inferior lo define nuestro hardware y el tiempo que le tome a este realizar el cálculo de las motivaciones. Si se configura un valor demasiado chico, el tiempo entre iteraciones podría volverse inconsistente si nuestro hardware no es capaz de completar una iteración completa en ese tiempo. El programador también deberá tener en cuenta que de cambiarlo, el resto de las configuraciones también deberán ser modificadas acordemente si se desea mantener el mismo crecimiento de la motivación en el tiempo y el mismo nivel de activación.

En cada iteración el *coodinator* utiliza toda la información que el robot ha recolectado por medio de la comunicación con otros robots, el tiempo que pasó y el *sensory feedback* proporcionado por cada motivational behavior y utilizará esta información para calcular la motivación de cada behavior set. Luego de calculada la motivación, el *coordinator* decide en base a lo calculado y al nivel de activación configurado si un cambio de *behavior set* es pertinente, y de ser así se encarga de desactivar el *behavior set* que se encuentra ejecutando actualmente y de activar el nuevo. Al final de cada iteración el *coordinator* también se encarga de notificar al resto de los robots que *behavior set* el robot tiene activado en ese instante.

La figura 4.1 muestra un ejemplo de esta arquitectura. Similar al ejemplo de ALLIANCE visto previamente (fig. 2.9), se tienen 3 capas y 2 *behavior sets*, con sus correspondientes *motivational behaviors*. A su vez, también se observa el nuevo elemento de la arquitectura, el *coordinator*, el cual recibe como entrada la información de los mensajes recibidos de otros robots y el valor de los *sensory feedback* calculados por los *motivational behaviors*. Como se ve en la figura, los *motivational behaviors* ya no utilizan inhibición cruzada ni se encargan de inhibir a su *behavior set* correspondiente, sino que de todo esto se encarga ahora el *coordinator*.

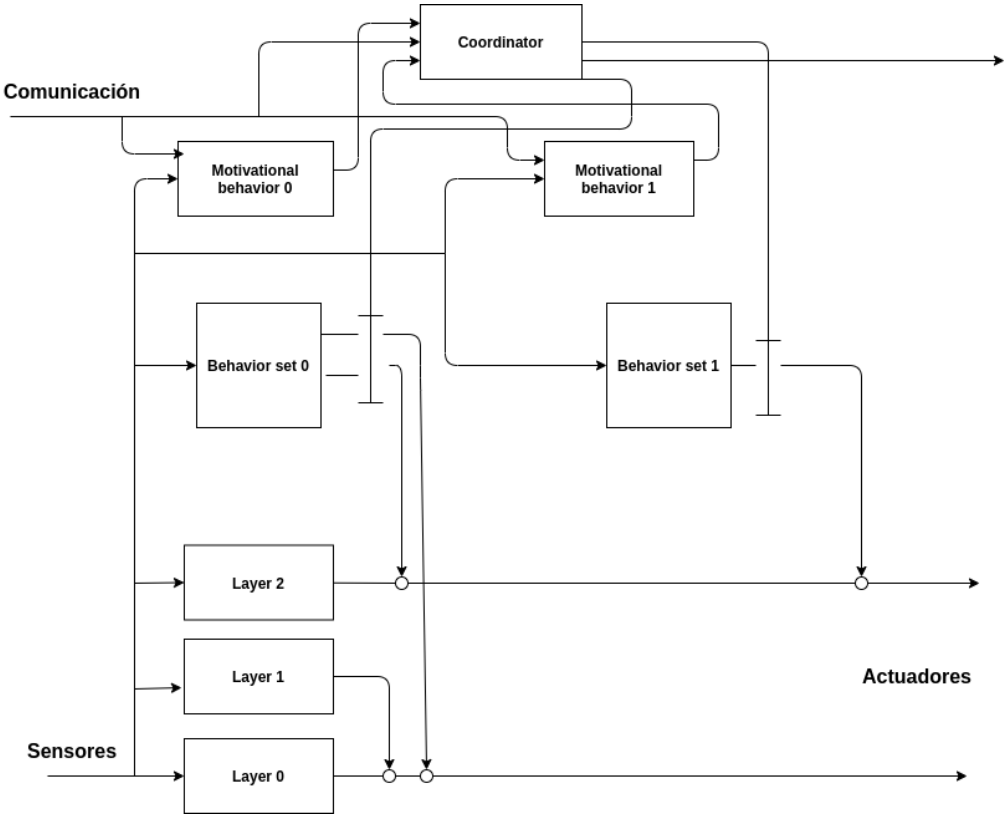


Figura 4.1: Arquitectura del sistema.

En este aspecto, la implementación diverge un poco de la arquitectura planteada por Parker, pero nos permite concentrarnos en lo que se consideró más importante que es la facilidad para el usuario al momento de utilizar la biblioteca. El hecho de que la responsabilidad del cálculo de la motivación fuera transferido desde los *motivational behaviors* al *coordinator* nos permite liberar de esta responsabilidad tanto al programador como al *motivational behavior*, permitiendo que este solo se encargue del *sensory feedback*.

4.2.1. Comunicación

Como se estableció anteriormente, el programador deberá proporcionarle a la biblioteca herramientas para poder enviar y recibir mensajes, y la biblioteca usará esto para encargarse de enviar mensajes cuando considere necesario y de procesar los mensajes recibidos por otros robots. Con este propósito se construyeron dos funciones, $configure\_notifier$ y $message\_received$.

La primera de estas funciones se usa para configurar el envió de mensajes, y toma como parámetro una función la cual a su vez, deberá tomar un parámetro *string*, llamaremos a esta función $notifier\_function$ para evitar confusiones. La biblioteca se encargará entonces de llamar a $notifier\_function$ en cada iteración del *coordinator* pasándole como argumento el nombre del motivational behavior que se encuentra actualmente activado. De esta forma el programador no necesita encargarse de saber que motivational behavior se encuentra activado ni de cuándo deberá mandar el mensaje, pero si del método de envio.

De forma similar, la función $message\_received$ permite avisarle a la biblioteca de los mensajes recibidos para que esta pueda tomarlos en cuenta al momento de calcular la motivación. La función toma como parámetros el *id* del robot que envió el mensaje y el *motivational behavior* que este tiene activado. El programador deberá encargarse de recibir y procesar los mensajes de otros robots para saber de qué robot provienen y su *motivational behavior*, pero luego la biblioteca se encarga de manejar el contenido del mensaje para analizar cuánto tiempo pasó desde el último mensaje de ese robot y si este cambio de actividad. Esta información pasa a ser accesible por el *coordinator* para utilizar al momento de calcular la motivación.

4.2.2. Funcionalidades

Ya habiendo visto las decisiones que fueron tomadas y cómo estas repercuten en la implementación del arquitecto, pasamos ahora a ver qué funcionalidades se consideraron importantes para agregar a la biblioteca. Este conjunto de funcionalidades deberá ser suficiente para que un programador cuente con todas las herramientas necesarias para programar un robot siguiendo la arquitectura ALLIANCE.

4.2.2.1. Crear behavior set

Dado que Torocó ya nos brinda una forma de crear *behaviors* (A.2.1), nuestra nueva implementación debe construir sobre esto y permitir agruparlos en *behavior sets*. Para esto se debe poder seleccionar un nombre que lo identifique, el conjunto de *behaviors* que le corresponden y además debe poder ser vinculado con un *motivational behavior*.

4.2.2.2. Crear motivational behavior

Luego de haber creado un *behavior set*, el siguiente paso sería asignarle un motivational behavior. Con este propósito la biblioteca nos permite crear un *motivational behavior* y vincularlo con su *behavior set* correspondiente (A.2.2). Además, se pueden configurar todos los elementos necesarios para calcular la motivación de ese *behavior set*, entre los cuales tenemos los parámetros de impaciencia y de consentimiento.

Además, de forma similar a un *behavior*, los *motivational behaviors* consisten en un conjunto de corrutinas las cuales pueden utilizar información de los sensores para darle un valor a su *sensory feedback*, el cual será utilizado al momento de calcular la motivación.

4.2.2.3. Configurar entradas de motivational behaviors

Torocó nos provee con la función *set_inputs* la cual es utilizada para configurar las entradas de los behaviors y actuadores del robot. Nuestra implementación de ALLIANCE la expande permitiendo además utilizarla para configurar las entradas de los motivational behaviors (A.1).

4.2.2.4. Activación e inhibición de behavior sets

La biblioteca deberá manejar la activación e inhibición de los *behavior sets* cuando esto corresponda, basándose en la motivación calculada para cada *motivational behavior*.

4.2.2.5. Comunicación entre robots

La biblioteca también deberá encargarse de la comunicación entre robots. Para esto la biblioteca nos brinda 2 funciones *message_received* (A.4.1) y *configure_notifier* (A.4.2).

La primera de estas funciones se brinda como forma de notificar a la biblioteca de que se recibió un nuevo mensaje, y es llamada pasándole como parámetros el robot que envió el mensaje y el *behavior set* que este tiene activado. La biblioteca luego utiliza esta información para el cálculo de la motivación.

La segunda función cumple el propósito de pasarle a la biblioteca una forma de notificar a los otros robots que *behavior set* tiene activado el robot. Esta función será luego llamada en cada iteración el programa principal para notificar a los otros robots.

4.2.2.6. Configurar nivel de activación

Se provee también una función *set_motivation_threshold* (A.3.1) la cual se encarga de configurar el nivel de activación para todos los motivational behaviors del robot. Este valor viene por defecto configurado en 100 a no ser sea cambiado utilizando la función mencionada anteriormente.

Capítulo 5

Experimentos

Contents

En esta sección veremos los diferentes experimentos que fueron realizados para probar y demostrar la correcta implementación de la arquitectura Alliance, además del entorno en el cual se realizaron estos experimentos.

5.1. Entorno de simulación

Dada la complejidad que surge a partir de este tipo de sistemas y de la necesidad de múltiples robots para poder generar pruebas se tomó la decisión de utilizar MORSE [Echeverria et al., 2012] como entorno de simulación para realizar pruebas. MORSE es un simulador de pruebas robóticas con propósitos académicos implementado en Python [Python Software Foundation, 2021] que permite realizar simulaciones relativamente realistas en un entorno 3D. Como motor para renderizar las simulaciones MORSE utiliza Blender [Blender Foundation, 2010].

Una de las principales ventajas que tiene MORSE en comparación con otros simuladores similares es que tiene una biblioteca estándar de sensores (cámaras, GPS, colisiones, etc.), actuadores (control de velocidad, motores, etc.), robots (quadrotors, ATRV, Pioneerd3DX, etc.) y objetos (pelotas, sillas, paredes, árboles, etc.) que nos permite generar distintos entornos y programar pruebas con relativa facilidad.

A su vez MORSE resulta flexible, permitiendo utilizar Blender para crear todo lo que se considere necesario y no venga por defecto con MORSE. Esto nos permite crear nuevos entornos, robots, objetos, etc. con Blender que luego podemos integrar a nuestra simulación. Además, utilizando Blender podemos editar ciertos elementos de la realidad que sería muy difícil simular con situaciones reales, como puede ser la gravedad o la fricción entre diferentes superficies.

Otra de las ventajas que presenta MORSE es que resulta relativamente liviano, permitiendo generar pruebas más complejas sin tener que preocuparse por su rendimiento. Una simulación puede ser separada en varios nodos, cada uno ejecutando su propia versión de la simulación y encargándose de un conjunto de los robots simulados. Los nodos se comunican con un manejador central el cual se encarga de sincronizar las simulaciones de los nodos.

La ejecución en múltiples nodos permite dividir la carga de la ejecución de la simulación y facilita la comunicación con elementos externos, dado que cada robot puede ejecutar en un nodo distinto y tener sus propios puertos de comunicación, los cuales son accedidos por los programas de ejecución de cada robot.

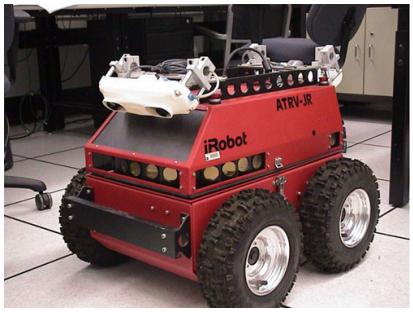
A pesar de todas sus ventajas MORSE tiene algunas limitaciones que pueden hacer que no se adapte a cualquier proyecto. Por un lado no tiene una interfaz gráfica, lo cual hace que su uso sea limitado para usuarios menos técnicos. A diferencia de otros simuladores, MORSE tampoco cuenta con una biblioteca de algoritmos que puedan utilizarse y asignar fácilmente a un robot, lo cual hace que siempre tengamos que programar manualmente el comportamiento de todos los robots que tengamos en nuestra simulación. A pesar de que funciona bien para pruebas de relativa complejidad, MORSE no se considera un simulador con físicas completamente reales, haciendo que no se adecue bien a simulaciones que tengan una componente física muy fuerte, y para este tipo de simulaciones sus creadores recomiendan utilizar otro simulador, nombrando OpenGrasp como ejemplo [León et al., 2010].

Dadas todas las ventajas vistas anteriormente y que ninguna de las limitaciones de

MORSE resultan problemáticas para el proyecto y las simulaciones que serán realizadas, se consideró que MORSE se adecua correctamente al proyecto y permite realizar pruebas tan complejas como sea necesario.

5.2. Robot utilizado

Para las simulaciones se decidió utilizar el robot ATRV [iRobot, 2012], creado y anteriormente distribuido por la compañía iRobot, el cual podemos visualizar en la imagen 5.1a. Este es un robot móvil que cuenta con 4 ruedas todo terreno y conexión ethernet inalámbrica, lo cual lo hace ideal para las pruebas a realizar.



(a) Robot ATRV-JR real

(b) Robot ATRV en simulación de morse

Figura 5.1: Robot ATRV

Dentro de la simulación, el robot utilizado cuenta con las mismas características (tamaño, peso, forma, etc.) que en la realidad, y una apariencia física considerablemente similar (fig. 5.1b). MORSE además permite agregar nuevos sensores al robot para aumentar sus capacidades. El simulador provee de algunos sensores ya programados como acelerómetros, cámaras, reloj, sensor de coaliciones, etc., pero también nos permite crear nuestros propios sensores de ser necesario, brindando mayor flexibilidad y permitiendo llegar a una simulación más similar a la realidad.

5.2.1. Actuadores

Para su movimiento MORSE nos provee con una interfaz para controlar el robot definiendo su velocidad angular y lineal. De esta forma se puede de manera muy simple controlar la velocidad de sus 4 ruedas y girar sus ruedas delanteras para controlar la dirección en la que avanza.

A su vez también se programó un actuador encargado de hacer broadcast de los mensajes del robot. Para esto se utilizó la biblioteca luasocket [Nehab, 2004] de Lua que nos permite abrir sockets UDP, los cuales son utilizados para enviar mensajes a los puertos correspondientes de los otros robots dentro de la misma simulación. Se destaca además que se optó por el uso de sockets UDP en vez de TCP dado que se adapta más a un entorno real en donde no se mantienen conexiones abiertas entre los robots. A pesar de que al utilizar UDP no se tiene una garantía de que los mensajes efectivamente lleguen al

destino, esto asemeja mejor las características de un entorno real en el cual dependiendo el método de comunicación pueden no tenerse tales garantías. UDP también tiene un mejor rendimiento que TCP, evitando los controles que permiten garantizar el orden y estado de los mensajes, de esta forma se tiene un mejor rendimiento, particularmente en situaciones donde los recursos pueden resultar limitados.

5.2.2. Sensores

Para los sensores se utilizaron un conjunto de diferentes sensores brindados por el simulador para permitirle al robot ubicar las pelotas que deberá empujar de la plataforma y a su vez darse cuenta cuando ya todas las pelotas de un conjunto fueron removidas.

Para poder ver y saber hacia donde moverse para poder empujar las pelotas se situaron 2 cámaras en el robot, una en su lado frontal derecho y otra en su lado frontal izquierdo. Al tener 2 cámaras distintas a ambos lados del robot resulta muy fácil saber donde están situadas las pelotas en torno al robot.

El hecho de que el robot no pueda ver algunas de las pelotas de un conjunto no quiere decir que ya se haya cumplido el objetivo de removerlas, el robot podría simplemente estar mirando para otro lado sin saber que existen más pelotas detrás de él. Esto genera la necesidad de agregar otro sensor que nos permite detectar cuando el objetivo fue cumplido. Con este propósito se agregó un sensor de proximidad, el cual le permite al robot detectar la presencia de objetos dentro de un determinado rango de distancia. Dado que al remover las pelotas de la plataforma estas caen al vacío, eventualmente salen del rango detectable por el sensor de proximidad haciendo que el robot ya no las detecte. De esta forma cuando el robot ya no detecta más pelotas del mismo conjunto en su proximidad puede asumir que se finalizó la tarea.

Por último, el robot también utiliza el socket UDP mencionado anteriormente para recibir los mensajes de otros robots.

5.3. Casos de prueba

Como base para todos los experimentos se decidió utilizar el mismo caso de prueba, realizando pequeñas modificaciones de ser necesario para ajustarse a lo que se quiere comprobar en cada experimento. Esto por un lado nos permite poder generar distintos experimentos rápidamente dado que la mayoría de los elementos se mantienen entre experimentos, pero además también se consideró que saca el foco en el caso de prueba particular y se lo da a los elementos y configuración que hacen a ese experimento distinto de los demás.

Tomando esto en cuenta se genera un caso de prueba en el cual el conjunto de robots deberá limpiar una plataforma cuadrada de diferentes conjuntos de pelotas situadas sobre ella. El caso base consiste de dos grupos diferentes, las pelotas azules y las pelotas rojas, donde cada conjunto va a corresponder a un *behavior set* distinto, los cuales llamaremos *collect* n, siendo n el conjunto de pelotas correspondientes.

Para este caso particular con 2 conjuntos el robot tendrá el behavior set *collect blue* el cual corresponde a tirar las pelotas azules y el behavior set *collect red* correspondiente a las rojas. La figura 5.2 muestra una imagen del comienzo de la simulación, donde se

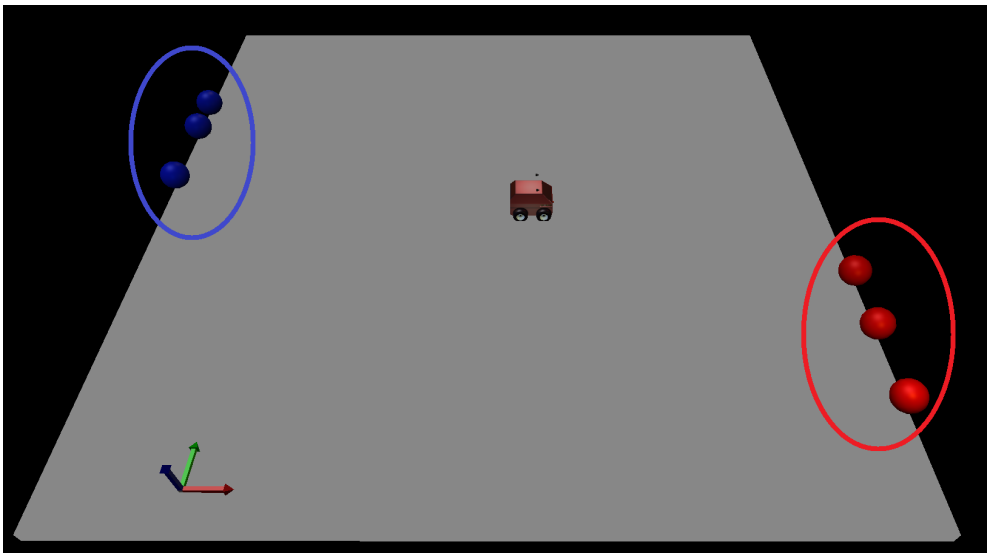


Figura 5.2: Caso de prueba donde se remarcaron los conjuntos de pelotas utilizados

puede observar el robot en el centro situado sobre la plataforma, y los 2 conjuntos de pelotas remarcados con sus colores correspondientes.

5.3.1. Funcionamiento de behavior sets

La primera prueba consiste en el caso base mencionado anteriormente, donde se tiene un solo robot con dos behavior sets distintos correspondiente a cada conjunto de pelotas. Para este caso se configuró para que el behavior set correspondiente al segundo conjunto tuviera una motivación que creciera más rápido que la otra, para así garantizar de que este se activará primero. Por otro lado también se configuró un rango de tiempo de 50 segundos antes de que el robot se rinda y baje su motivación a 0. Esta configuración puede verse en la tabla 5.1.

Con esta prueba se busca validar el funcionamiento básico de la arquitectura y observar su comportamiento con un solo robot.

	collect blue	collect red
slow_rate	1	2
fast_rate	2	3
yield_time	40	40
give_up_time	50	50
nivel de activación	20	

Cuadro 5.1: Configuración - Funcionamiento de behavior sets

Como se observa en la gráfica 5.3 tenemos que en principio ambas motivación crecen con el tiempo con una pendiente correspondiente a su $fast\_rate$. Esto hace que el moti-

vational behavior correspondiente al *behavior set collect red* llegue al nivel de activación necesario primero y por lo tanto se active.

Esta activación causa que la motivación para activar el *behavior set collect blue* baje a 0. Este comportamiento se mantiene hasta que el robot completa con su objetivo, lo cual genera que el $sensory\_feedback$ correspondiente a ese *motivational behavior* cambie a 0, y por lo tanto también su motivación. Esto también causa que la motivación del otro *motivational behavior* vuelva a crecer, dado que ya no lo suprime el *activity_supression*.

Como ahora solo una de las motivaciones crece, dado que la otra se mantiene en 0 porque se completo ese objetivo, la motivación de *collect blue* logra crecer hasta llegar al nivel de activación, en el cual se mantiene por un periodo de 50 segundos. Dado el *give_up_time* que fue configurado, pasados los 50 segundos la motivación baja a 0 de nuevo pero vuelve a crecer dado que el $sensory\_feedback$ indica que el objetivo todavía no fue completado. Esto causa que se llegue al nivel de activación nuevamente y se active el *behavior set*. Eventualmente el objetivo es completado y ambas motivaciones se mantienen en 0.

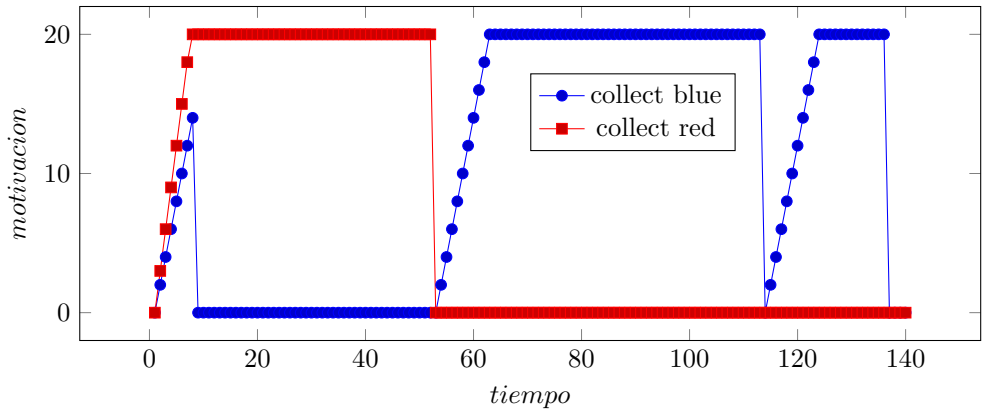


Figura 5.3: Funcionamiento de behavior sets

5.3.2. Cálculo de la motivación con comunicación

Para la segunda prueba se buscó hacer un énfasis en el crecimiento de la motivación para un *motivational behavior* particular. Para poder dar foco a esto sin que diferentes variables externas dificulten la visualización de lo que se quiere analizar se utilizó un solo robot con un solo *behavior set* y se enviaron manualmente señales al robot para simular la recepción de mensajes de otros robots. Para esto a los ocho segundos de haber iniciado la simulación se envió mediante un socket UDP un único mensaje al robot indicando que existía otro robot realizando la tarea. Luego de enviado este mensaje se dejó a la simulación continuar de forma normal. La tabla 5.2 muestra la configuración utilizada en esta prueba.

Como observamos en la gráfica 5.4, en principio la motivación crece con un valor constante correspondiente a su $fast\_rate$. Luego, en el momento que el robot recibe el mensaje que se le envió su motivación baja a 0 dado el *motivation_reset* y continúa

	collect blue
slow_rate	2
fast_rate	4
yield_time	80
give_up_time	100
nivel de activación	100

Cuadro 5.2: Configuración - Cálculo de la motivación con comunicación

creciendo con un rate más lento, correspondiente a su *slow_rate*. Eventualmente pasa suficiente tiempo sin que el robot reciba mensajes de otro robot, y dado que considera que ningún robot se encuentra trabajando en la tarea, la motivación vuelve a crecer con su $fast\_rate$. Eventualmente la motivación llega al nivel de activaciones que se configuró en 100 y el robot activa el *behavior set* correspondiente.

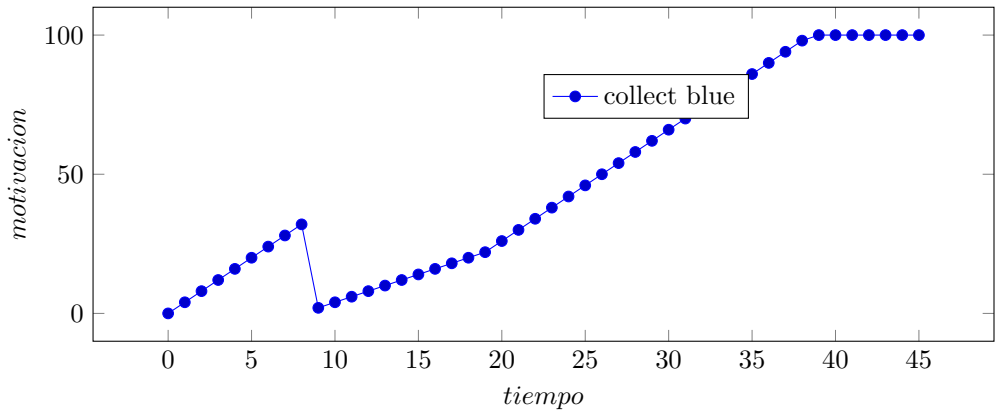


Figura 5.4: Cálculo de la motivación con comunicación

5.3.3. División de tareas

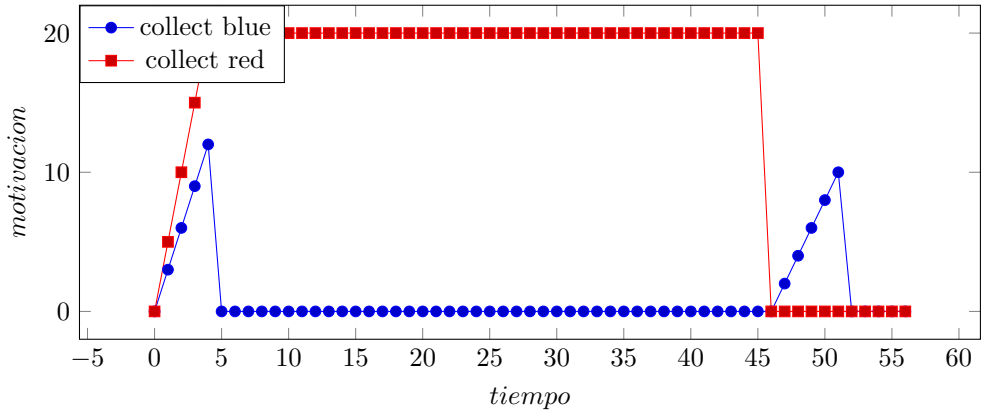
Para la tercera prueba se tiene un escenario muy similar al primero pero esta vez con 2 robots, ambos idénticos tanto en su programación como en la configuración de sus parámetros. En este caso se desea observar el correcto funcionamiento de la arquitectura cuando se tiene más de un robot, enfocándose principalmente en la comunicación entre ellos y el efecto que esto tiene en su elección de tareas. La única configuración que fue alterada con respecto a la prueba anterior es el *give_up_time* para evitar que un robot abandone una tarea antes de terminarla. La tabla 5.3 muestra las configuraciones utilizadas.

Comenzamos enfocándonos en el comportamiento del primer robo, correspondiente a la gráfica 5.5a. En este se ve un comportamiento muy similar al de la primera prueba, donde la motivación de un comportamiento crece más rápido y toma control sobre el robot hasta que eventualmente cumple con su objetivo y da lugar al otro comportamiento. La diferencia más notoria que se observa con la prueba anterior es que en este caso el

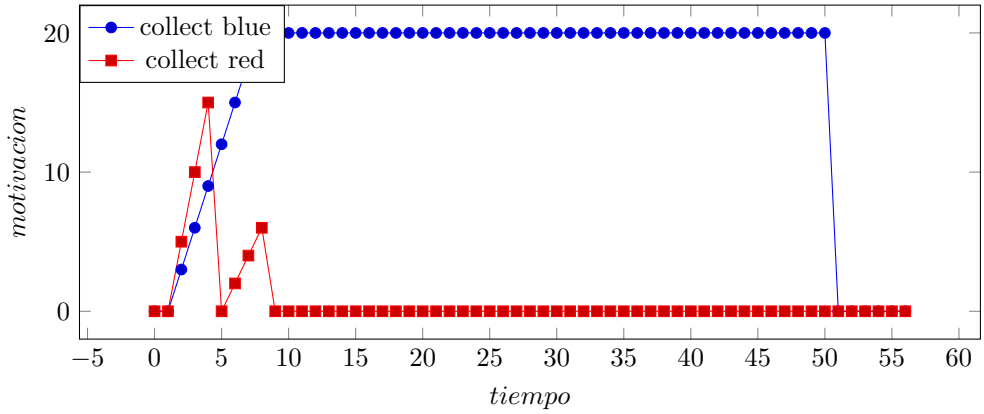
	collect blue	collect red
slow_rate	2	2
fast_rate	3	5
yield_time	80	80
give_up_time	100	100
nivel de activación	20	

Cuadro 5.3: Configuración - División de tareas

behavior set collect blue nunca llega a ser activado por este robot dado que el otro robot termina el objetivo antes de que la motivación para activarlo llegue al nivel de activación. Cuando esto sucede las motivación para ambos objetivos quedan en 0 dado que ya fueron cumplidos y se finaliza la simulación.



(a)



(b)

Figura 5.5: División de tareas

Observando el comportamiento del segundo robot en la gráfica 5.5b, podemos obser-

var algunas diferencias con lo que venimos viendo hasta ahora, donde siempre se venía activando el *motivational behavior* con mayor $fast\_rate$. En este caso al principio la motivación para *collect red* crece más rápido que para *collect blue*, pero dado que los robots no comenzaron su ejecución en exactamente el mismo instante, el primer robot activa este *behavior set* antes y se lo comunica al segundo robot. Esto causa que la motivación para hacer *collect red* baje a 0 debido al *impatience_reset* y luego crezca con una velocidad más lenta correspondiente al *slow_rate* para este comportamiento. Eventualmente el robot cumple con su objetivo y su motivación baja a 0, y como además el otro robot ya completó la tarea *collect red* la motivación para realizarla tampoco crece.

5.3.4. Comportamiento de grupos de mayor tamaño

En esta prueba se busca observar el correcto funcionamiento de los robots con un grupo más grande. Dado que la complejidad de la prueba crece rápidamente al aumentar la cantidad de robot, tanto en terminamos de analizar los datos obtenidos y verificar que efectivamente cada robot se comportó como debería, como también en términos de exponer los resueltos y los requerimientos de hardware para correr la simulación correctamente, se decidió que realizar una prueba con 3 robots nos permitiría mantener la complejidad baja pero también contar con suficientes robots como para alcanzar el objetivo deseado. En la tabla 5.4 podemos ver los parámetros utilizados en esta prueba.

	collect blue	collect red
slow_rate	1	2
fast_rate	2	3
yield_time	80	80
give_up_time	100	100
nivel de activación	50	

Cuadro 5.4: Configuración - Comportamiento de grupos de mayor tamaño

En este caso se agregó un robot más al experimento anterior manteniéndose los mismos comportamientos. Para la configuración se bajaron un poco los rates de la motivación y se aumentó el nivel de activación para que resultara más sencillo analizar el comportamiento de los robots en un entorno más lento.

En principio en los primeros dos robots se observa el mismo comportamiento que en el experimento anterior, donde el *motivation_reset* genera que ambos robots se dividan las tareas. El primer robot (fig.5.6a) que activa el *behavior set* con la motivación que crece más rápido, y el segundo (fig.5.6b) tomando el otro, para el cual no se ha bajado a 0 la motivación.

El comportamiento que resulta más interesante es el observado en el tercer robot (fig.5.6c). En principio dado que los otros dos robots logran llegar a los niveles de activación para sus *behavior sets* correspondientes antes que él, la motivación para realizar ambos baja a 0 antes de que pueda activar cualquiera de ellos. Además, debido a que ya hay un robot realizando cada una de las tareas, la motivación para realizar cada una utiliza el *slow_rate* correspondiente, haciendo que el robot demore más en poder activar algún *behavior set*. Esto causa que los otros robots terminen sus tareas antes de que el

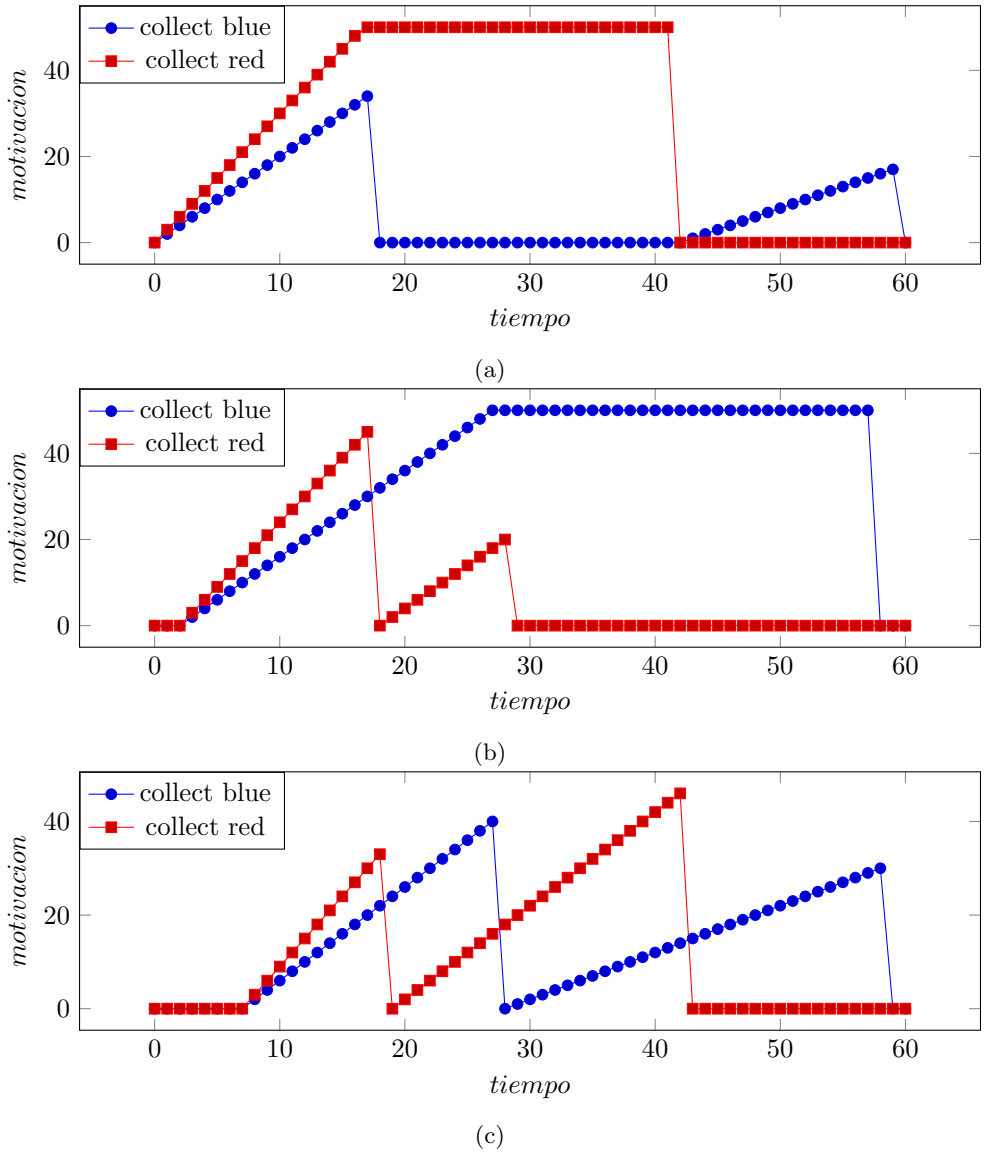


Figura 5.6: Comportamiento de grupos de mayor tamaño

tercer robot pueda comenzar a trabajar en cualquiera de ellas, y por lo tanto no logra ser de ayuda en la misión.

En principio esto puede parecer una falla del lado de la arquitectura, dado que uno de los robots no aportó a la misión y se quedó quieto mientras los otros trabajaban. En realidad lo que este experimento demuestra es que se utilizaron demasiados robots para el trabajo a ser realizado. Si contáramos con más objetivos a cumplir hubiéramos visto un comportamiento similar a la prueba anterior, donde los robots hubieran dividido el

trabajo, cada uno realizando una tarea. Si por otro lado se mantuvieran las 2 tareas pero estas requieren más esfuerzo de parte de los robots y demoran más tiempo en terminarse, entonces el robot hubiera podido activar uno de estos comportamientos antes de que los otros robots los terminaran.

5.3.5. Comportamiento ante una tarea imposible

La quinta prueba que se realizó es una variación de la primera, en la cual se quiere probar cómo reacciona un robot ante la situación de no poder completar el objetivo que tiene prioridad, dado que su $fast\_rate$ es más rápido como se puede ver en la tabla 5.5.

	collect blue	collect red
slow_rate	1	2
fast_rate	2	3
yield_time	20	20
give_up_time	40	40
nivel de activación	20	

Cuadro 5.5: Configuración - Comportamiento ante una tarea imposible

A modo de simular no poder completar una tarea se utilizó la misma simulación que en la primera prueba, pero aumentó el peso de las pelotas correspondientes al *behavior set collect red* y su fricción con el piso en la simulación. Estas alteraciones causan que el robot no tenga la suficiente fuerza como para mover las pelotas de lugar, causando que no las pueda tirar de la plataforma y por lo tanto tampoco completar su objetivo.

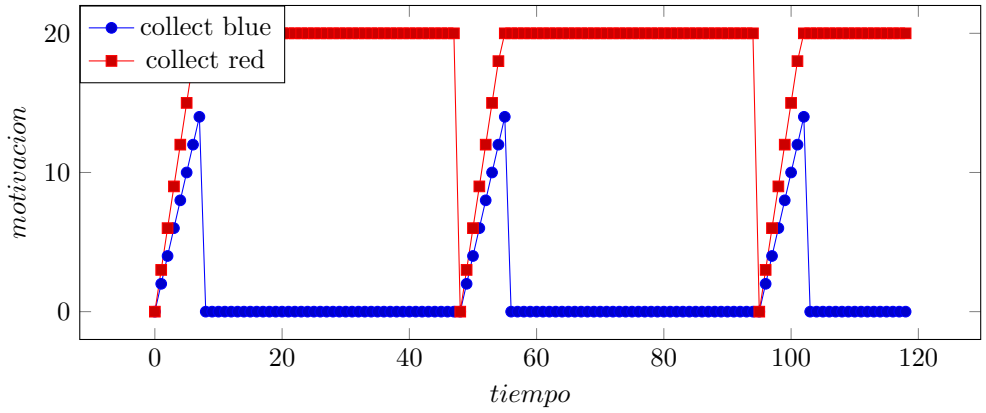


Figura 5.7: Comportamiento ante una tarea imposible

Lo que se observa en este caso (fig.5.7), es que el robot activa el *behavior set collect red* primero dado que es el que crece más rápido, y lo mantiene activo hasta que eventualmente se llega al *give_up_time*, causando que lo desactive y baje su motivación a 0. A continuación, dado que ambos objetivos siguen sin cumplirse y que la velocidad con la cual crece la motivación de *collect red* sigue siendo superior a la de *collect blue*,

este *behavior set* se activa de nuevo, por mas que el robot sea incapaz de completar su objetivo y que el otro *behavior set* permanezca sin cumplirse. Este comportamiento continúa hasta que se fuerza la terminación de la simulación.

Este caso destaca un punto débil de la arquitectura, dado que a pesar de que hubiera una tarea sin terminal, el robot continúa intentando con la tarea que tiene prioridad, a pesar de ser incapaz de completarla. Este caso resulta bastante particular dado que solo ocurre cuando se tiene un solo robot, de tener un conjunto de dos robots o más se vería un comportamiento similar al visto en la prueba 2 en donde los robots se dividen las tareas. Vemos a continuación otra prueba para validar estas hipótesis.

5.3.6. Comportamiento ante una tarea imposible con múltiples robots

Con el objetivo de comprobar que el comportamiento observado en la prueba anterior es un caso particular que sucede debido a tener un solo robot, se optó por realizar la misma prueba pero agregando un robot más. Las configuraciones en este caso son las mismas como se observa en las tablas 5.5 y 5.6. Las gráficas presentes en la figura en 5.8 muestran el comportamiento de cada robot.

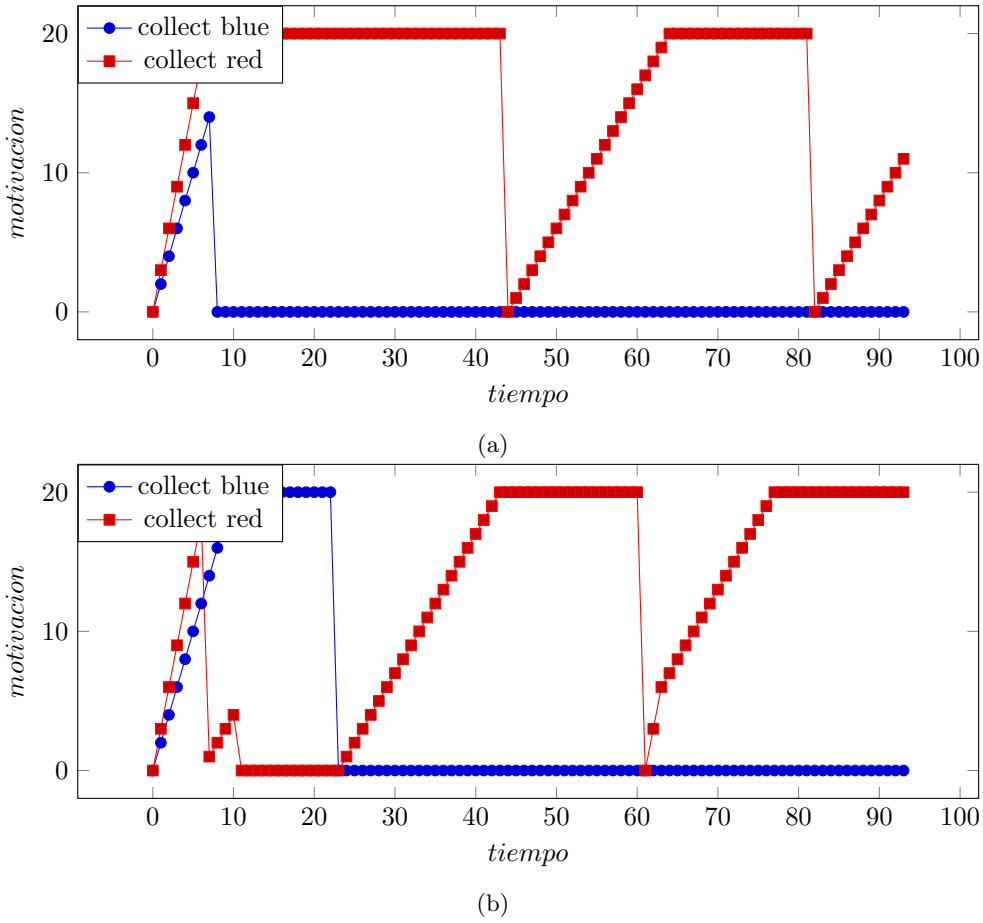
	collect blue	collect red
slow_rate	1	2
fast_rate	2	3
yield_time	20	20
give_up_time	40	40
nivel de activación	20	

Cuadro 5.6: Configuración - Comportamiento ante una tarea imposible con múltiples robots

Todos los parámetros se mantuvieron iguales a la prueba anterior, siendo la única diferencia el nuevo robot que se sumó al equipo.

Al observar el comportamiento de ambos robots se comprueba el resultado esperado. En principio los robots se dividieron las tareas, con un robot (5.8a) tomando la tarea que no pueden terminar y el otro (5.8b) tomando la otra. Lo que sucede entonces es que uno de los robots termina su tarea y luego pasa a tratar de ayudar al otro, dejando a ambos tratando de terminar esta tarea indefinidamente, dado que ninguno de los dos puede lograr el objetivo.

Se aprovecha también para destacar que la prueba también sirve para validar el correcto funcionamiento de los parámetros *yield_time* y *give_up_time*. Como podemos observar al principio al primer robot le toma 40 segundos abandonar su tarea, correspondiente al *give_up_time* configurado, pero luego dado que ya hay otro robot realizando la misma tarea ambos abandonan la tarea a los 20 segundos, equivalentes al *yield_time*.



(a)

(b)

Figura 5.8: Comportamiento ante una tarea imposible con múltiples robots

5.4. Conclusiones

De las pruebas realizadas anteriormente se puede observar el correcto funcionamiento de la implementación construida para la arquitectura ALLIANCE.

Las primeras pruebas ayudan a ver el comportamiento de una misión en la cual solo participa un único robot. Esto nos permitió analizar el funcionamiento del robot con más detalle, particularmente las funcionalidades de la separación de tareas por *behavior sets*, la activación y supresión de estos, y como distintos eventos afectan la motivación de cada *motivational behavior*.

Las siguientes pruebas se concentraron en validar el correcto funcionamiento de la arquitectura frente a un grupo con múltiples robots. Estas pruebas consisten en conjuntos de 2 y 3 robots, lo cual permitió mantener las pruebas fáciles de analizar y livianas al momento de ejecutarlas, pero también siendo conjuntos suficientemente grandes como para que las pruebas sean válidas. Al analizar estas pruebas se vio que el conjunto de robots logra separar las tareas correctamente mediante los mensajes enviados entre sí,

sin necesidad de ninguna intervención externa.

Por último se realizaron pruebas para analizar el funcionamiento de los robots ante una tarea que tiene prioridad dado su $fast\_rate$, pero que les resulta imposible de realizar. La primera de estas pruebas se realizó con un solo robot, el cual a pesar de tener 2 tareas para realizar se queda atascado indefinidamente tratando de completar la que tiene prioridad, nunca intentando realizar la otra tarea. Al realizar una prueba con las mismas características pero agregando otro robot en la simulación, se observó un comportamiento similar, pero en este caso los robots comienzan separándose las tareas, por lo tanto el segundo robot logra terminar su tarea y luego procede a tratar de ayudar al otro, eventualmente quedando ambos trancados sin poder terminar la tarea. Tras analizar estas últimas pruebas se concluyó que a pesar de que el comportamiento no es necesariamente el deseado y es un punto débil de la arquitectura, el funcionamiento es de todas formas consistente con la arquitectura ALLIANCE.

Dado que todas las pruebas realizadas cumplieron con las expectativas de funcionamiento de los robots en base a la arquitectura ALLIANCE, se concluye que la implementación propuesta es correcta y brinda las herramientas para construir equipos de robots que funcionan correctamente en conjunto. También, dadas las últimas pruebas realizadas, se detectó un punto débil de la arquitectura que genera que un equipo con un solo robot ignore tareas que todavía no ha intentado y permanezca tratando de completar una tarea con prioridad que no puede realizar. Para resolver este último punto se decidió realizar una extensión a la arquitectura ALLIANCE que permita resolver este caso manteniendo los componentes básicos de la arquitectura.

Capítulo 6

Extensión de la arquitectura

Contents

Con el objetivo de solucionar los casos en los cuales se noto ciertas fallas en la arquitectura y se vio lugar para mejoras, se decidió extender la arquitectura ALLIANCE para de esta forma lograr mejores resultados. A modo de analizar diferentes opciones y posibles soluciones a los problemas mencionados anteriormente, se dará una introducción a la arquitectura L-ALLIANCE [Parker, 1998] la cual propone una variedad de mejoras a la arquitectura ALLIANCE base. Luego se verá qué cambios fueron realizados a nuestra arquitectura y el efecto que estos tienen con nuevas pruebas al sistema.

6.1. L-ALLIANCE

6.1.1. Introducción

Parker destaca algunas áreas de la arquitectura ALLIANCE sobre las cuales se pueden incluir mejoras para mejorar su rendimiento. Entre ellas se encuentran:

- Nada asegura que los robots trabajen en las tareas para las cuales están mejor capacitados.
- El rendimiento no mejora con el tiempo, por más que cada vez se debería tener más información.
- Una falla al realizar una tarea puede llevar a una falla total del robot.
- El tiempo que los robots permanecen ociosos puede resultar bastante alto.

Se definen además dos hipótesis sobre las cuales se construye la arquitectura:

- El promedio de rendimiento ejecutando una tarea específica en los últimos intentos de un robot son un estimador razonable de su rendimiento en el futuro.
- Si el robot r_i monitorea la condición C para determinar el rendimiento de otro robot r_k, y la condición C cambia, entonces este cambio se le atribuye al robot r_k

Dadas algunas de las faltas en la arquitectura ALLIANCE, Parker creó una extensión de la arquitectura llamada L-ALLIANCE, en la cual propone mecanismos para ajustar los parámetros de la arquitectura de forma automática,

A continuación veremos los elementos que utiliza L-ALLIANCE para poder aumentar el rendimiento en ALLIANCE en los puntos mencionados anteriormente.

6.1.2. Monitores de rendimiento

Para extender la arquitectura ALLIANCE se incorporan monitores de rendimiento para cada motivational behavior dentro del robot. Cada uno de estos monitores se encarga de observar, evaluar y catalogar el rendimiento de los robots de equipo (incluido el mismo) cuando estos realizan una tarea correspondiente al *behavior set* relacionado a ese monitor. Formalmente, el robot r_i que tiene los *behavior sets* $A_i = a_{i1}, a_{i2}..., a_{ib}$ y los monitores $MON_i = mon_{i1}, mon_{i2}, mon_{i3}, ..., mon_{ib}$, tal que el monitor mon_{ij} observa el rendimiento de todos los robots realizando la tarea $h_i(a_{ij})$ y registra métricas correspondientes a su rendimiento, generalmente utilizando el tiempo que le toma completar la

tarea. El monitor mon_{ij} utiliza esta información para actualizar los parámetros correspondientes al *behavior set* a_{ij}. Es importante destacar que el robot r_i no tiene monitores para los *behavior sets* que él mismo no tiene, lo cual le permite a L-ALLIANCE escalar a medida que la cantidad de tareas en la misión crece.

6.1.3. Fases de control

En L-ALLIANCE se hace distinción entre dos tipos de misiones, las misiones de entrenamiento y las misiones reales.

El propósito de las misiones de entrenamiento es que los robots se puedan familiarizar con ellos mismos y con los otros robots del equipo. Los robots podrán entonces explorar sus capacidades sin poner en riesgo el resultado de una misión real.

En las misiones reales se debe garantizar que la misión será realizada tan eficientemente como sea posible. De cualquier manera, incluso cuando los robots se encuentran en una misión real, también se deberá aprovechar esta oportunidad para recolectar información y aprender.

L-ALLIANCE entonces define dos fases de control las cuales pueden ser utilizadas dependiendo el tipo de misión en la que se encuentre el equipo. Durante las misiones de entrenamiento, los robots se encuentran en la fase de aprendizaje activo, mientras que en las misiones reales se encuentran en la fase de aprendizaje adaptativo.

En la fase de aprendizaje activo cada robot elige de forma aleatoria una tarea del conjunto de las tareas que determina están incompletas y que ningún otro robot está ejecutando actualmente. Mientras se realizan estas tareas, los robots tienen sus valores de paciencia maximizados y los de consentimiento minimizados, lo cual hace que el robot no trate de tomar una tarea que otro robot está realizando ni le deje su tarea actual a otro robot. Durante la fase de aprendizaje activo, cada monitor mon_{ij} de cada robot r_i se encarga tanto de monitorear y catalogar el rendimiento de los otros robots r_k que están realizando la tarea $h_i(a_{ij})$ como de actualizar los parámetros de control apropiados.

Cuando un robot está en una misión real, este se encuentra en la fase de aprendizaje adaptativo, en la cual las configuración de consentimiento e impaciencia se configuran acorde a lo aprendido en la fase de aprendizaje activo y a la estrategia de actualización utilizada, las cuales veremos más adelante. Durante esta fase el robot sigue recolectando información de la misma forma que hacía en la fase de aprendizaje activo, también utilizando esta nueva información durante la misión.

6.1.4. Estrategias de actualización de parámetros

L-ALLIANCE define tres estrategias para la actualización de la impaciencia y el consentimiento, y tres estrategias para el ordenado de las tareas. La tabla 6.1 muestra un resumen de estas estrategias, las cuales serán definidas más en detalle a continuación.

6.1.4.1. Estrategias de actualización de la impaciencia y el consentimiento

La primera estrategia (I) que veremos será llamada «Desconfiar del conocimiento de rendimiento del equipo». Esta estrategia toma un enfoque minimalista al problema, requiriendo que el robot solo use el conocimiento de su propio rendimiento. Bajo esta estrategia un robot r_i se vuelve impaciente con cualquier otro robot r_k que no complete la

Estrategia	impaciencia	Consentimiento
I	Tiempo propio	Tiempo propio
II	Tiempo propio	Tiempo mínimo del equipo
III	Tiempo que el robot ejecuta la tarea	Tiempo propio

Cuadro 6.1: Estrategias definidas por L-ALLIANCE

tarea $h_i(a_{ij})$ en el mismo tiempo que le requiere a r_i completarla. Esta estrategia resulta útil cuando se tiene poca información del equipo o la información no es suficientemente confiable como para ser utilizada.

La segunda (II) estrategia llamada «Que el mejor robot gane» establece que cada robot compare su rendimiento con el de el mejor robot del equipo. Si el robot r_i aprende que el menor tiempo en el que un robot del equipo puede terminar la tarea $h_i(a_{ij})$ es t, entonces el robot le cederá la tarea a otro robot si ha estado intentando realizarla por más tiempo que t. Además, el robot r_i solo se volverá impaciente con otro robot r_k cuando este último haya estado intentando realizar una tarea por más tiempo que el que le tomaría a r_i completarla.

La última estrategia (III), la cual llamaremos «Dar una oportunidad de trabajar» hace que el equipo de robots juzgue el rendimiento de los robots en base al rendimiento esperado de cada robot. Bajo esta estrategia el robot r_i solo se volverá impaciente con el robot r_k si está teniendo un peor rendimiento que el que tiene normalmente. Adicionalmente cada robot se niega a dejar su tarea a otro robot hasta que considere que intentó por suficiente tiempo, basándose en información sobre su rendimiento pasado.

6.1.4.2. Tres estrategias para ordenar tareas

Veremos entonces las tres estrategias planteadas por Parker para que un robot pueda elegir entre las tareas incompletas que ningún otro robot se encuentra realizando. En ALLIANCE esto lo define el $fast\_rate$ dado que al ser lo que define el crecimiento de la motivación, la tarea con mayor $fast\_rate$ va a ser la seleccionada. Al realizar modificaciones a este parámetro se llegan a las diferentes estrategias que veremos.

La primera estrategia es la de «Tarea más larga primero», y tal y como indica el nombre, define que los robots deberán elegir entre la tarea que les tome más tiempo terminar primero. Para lograr esto el $fast\_rate$ se ajusta proporcionalmente a la duración esperada de la tarea. Esta estrategia tiene un mal desempeño cuando los robots son heterogéneos, dado que los robots empezaron realizando las tareas para las cuales están menos capacitados primero.

La siguiente estrategia es una variación de elegir la tarea más corta y la llamaremos «Tarea más corta primero modificada». Bajo esta estrategia se clasifican las tareas en dos categorías: las que el robot sabe realizar mejor que cualquier otro en el equipo y el resto. Cada robot elige primero de las tareas en la primera categoría, y al terminarlas empieza a trabajar en las tareas de la segunda categoría. Para elegir la tarea más corta dentro de la categoría 1 se ajusta el $fast\_rate$ inversamente proporcional a la duración esperada de la tarea.

Por último se propone la estrategia de «Tarea aleatoria modificada», la cual es una variación de la estrategia anterior. Las tareas se dividen en las mismas 2 categorías que

en la estrategia anterior, pero ahora en vez de elegir la tarea más corta dentro de cada categoría se elige una aleatoriamente. La idea detrás de esta estrategia es tener una base para poder comparar las otras estrategias.

6.2. Extensión

A modo de evitar que un robot se quede atascado tratando de realizar una tarea que le resulta imposible, mientras todavía existen otras tareas que podría estar haciendo, se tomó inspiración de las estrategias propuestas por Parker en L-ALLIANCE. Los cambios que veremos a continuación surgen a partir de una combinación de las estrategias «Desconfiar del conocimiento de rendimiento del equipo» y «Tarea más corta primero modificada». La idea general sería que el robot pueda aprender basándose únicamente en el conocimiento que obtiene de su propio rendimiento y utilice esto para separar las tareas en categorías, las cuales utilizará para elegir su siguiente tarea. Se destaca además que por más que los cambios realizados son inspirados en L-ALLIANCE, se quiso mantener el funcionamiento base de ALLIANCE intacto mientras fuera posible.

Se crearon entonces dos conjuntos de behavior sets los cuales llamaremos primarios y secundarios. El conjunto primario corresponde a los *behavior sets* que el robot tendrá en cuenta al momento de elegir su siguiente tarea. El conjunto secundario en cambio corresponde a los *behavior sets* que el robot ha tratado de completar pero no ha podido, y por lo tanto no tendrá en cuenta hasta haber completado, o al menos intentando completar, el resto de los *behavior sets*. Veremos a continuación cómo funciona esto en la práctica y cómo se formaliza en la arquitectura.

Al comenzar la ejecución del programa todos los *behavior sets* del robot comienzan en el conjunto primario. El robot entonces funcionará en principio de la misma manera a la vista anteriormente, eligiendo el *behavior set* que primero llegue al nivel de activación configurado. Eventualmente la motivación que tiene el robot para continuar haciendo la tarea elegida bajará a cero, lo cual podría suceder porque terminó la tarea o porque decidió darse por vencido o ceder a otro robot.

Lo que sucede a continuación es que el robot realiza todas las tareas del conjunto primario, logrando completarlas o pasándolas al conjunto secundario si no logra completarlas, hasta que eventualmente todas las tareas fueron completadas o pertenecen al conjunto secundario. Llegado este momento el robot limpia su conjunto secundario y pasa todos los *behavior sets* al conjunto primario nuevamente, intentando nuevamente y repitiendo el proceso.

Lo que se logra con este cambio es evitar que el robot quede atascado intentando tareas que nunca va a poder terminar, y en cambio realiza una selección similar a un «Round Robín», manteniéndose la selección por el rate más rápido que le da prioridad a cada *behavior set*.

6.3. Formalización

Comenzamos definiendo una nueva función llamada *category*, la cual utilizaremos para saber en qué categoría se encuentra cada *behavior set*. Definimos entonces a *category* de la siguiente mantera:

$$category_{ij}(t) = \begin{cases} 0 & \text{si el robot } r_i \text{ tiene al behavior set } a_{ij} \text{ en su categoría} \\ & \text{secundaria en momento } t \\ 1 & \text{en el caso contrario} \end{cases} \tag{6.1}$$

Habiendo definido *category*, la nueva fórmula para calcular la motivación pasa a ser:

$$\begin{aligned} m_{ij}(0) &= 0 \\ m_{ij}(t) &= [m_{ij}(t-1) + impatience_{ij}(t)] \\ &\times sensory\_feedback_{ij}(t) \\ &\times activity\_supression_{ij}(t) \\ &\times impatiences\_reset_{ij}(t) \\ &\times acquiescence_{ij}(t) \\ &\times category_{ij}(t) \end{aligned} \tag{6.2}$$

6.4. Implementación

Para la implementación de los cambios se crea un set en el cual el robot guarda los *behavior sets* que pertenecen al grupo secundario, donde se sabe que todo *behavior set* que no pertenezca a este conjunto forma parte del conjunto primario.

Además, el *coordinator* también tiene la responsabilidad de manejar el nuevo conjunto de *behavior sets*. Cada vez que se deshabilite un *behavior set* el *coordinator* agrega ese *behavior set* al conjunto, para de esta forma mantener su motivación en cero hasta que salga de él. Luego se verifica si todos los *behavior sets* del robot que correspondan a tareas sin terminar, o sea las que tengan $sensory\_feedback = 1$, pertenecen al conjunto, y de ser así lo vacía, lo cual corresponde a pasar todos los behavior sets al grupo primario.

6.5. Nuevos casos de prueba

Para comprobar que los cambios realizados a la arquitectura efectivamente tuvieron los resultados esperados se repitieron las mismas pruebas descriptas en 5.3. En todos los casos, menos en la prueba equivalente al caso 5.3.5 visto anteriormente, los resultados fueron los mismos que los vistos anteriormente, por lo cual no serán vistos nuevamente y se dará enfoque a la prueba en la cual sí se vieron cambios.

6.5.1. Comportamiento ante una tarea imposible - Actualizado

En esta prueba se utiliza exactamente la misma configuración que en la prueba 5.3.5 vista en la sección anterior, siendo la única diferencia el cambio en la arquitectura. La configuración utilizada en esta prueba puede verse en la tabla 6.2.

Lo que se espera comprobar con esta prueba es que el robot ya no queda trancado indefinidamente tratando de realizar la misma tarea, la cual nunca va a terminar.

Lo que efectivamente sucede, como se puede observar en la figura 6.1, es que el robot primero intenta con la tarea *collect red*, la cual no logra terminar y termina rindiéndose.

	collect blue	collect red
slow_rate	1	2
fast_rate	2	3
yield_time	30	30
give_up_time	40	40
nivel de activación	20	

Cuadro 6.2: Configuración - Comportamiento ante una tarea imposible - Actualizado

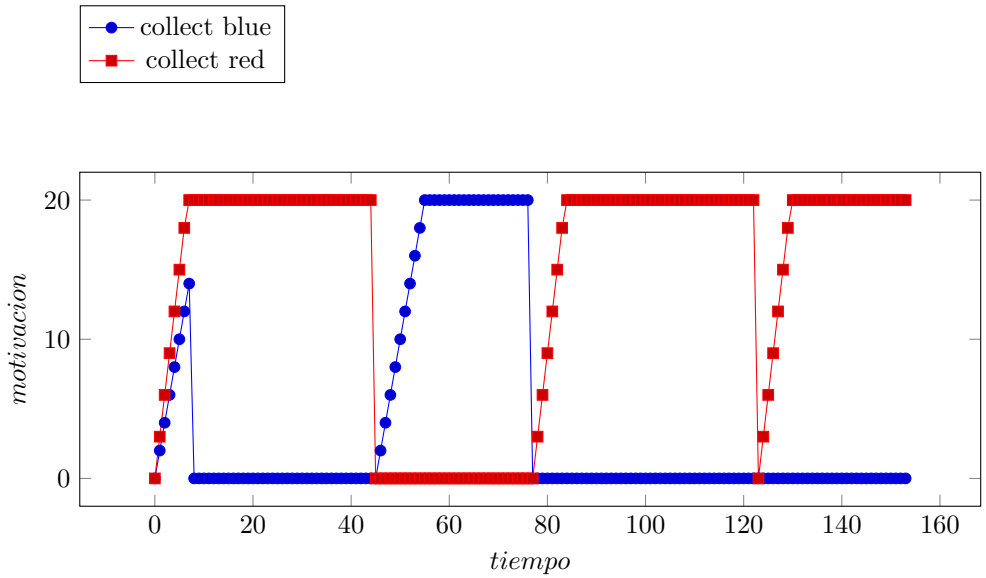


Figura 6.1: Comportamiento ante una tarea imposible - Actualizado

Como el *behavior set* corresponde a *collect red* ahora forma parte del conjunto secundario de *behavior sets* su motivación permanece en 0 hasta salir de él, lo que causa que a continuación se active el *behavior set* correspondiente a *collect blue*, a pesar de que su $fast\_rate$ sea más lento. Cuando el robot completa esta tarea todos los *behavior sets* para los cuales su $sensory\_feedback$ indica que no fueron terminados pertenecen al conjunto secundario, por lo tanto el robot lo vació y la motivación de *collect red* vuelve a crecer. Cuando el robot vuelve a fallar con esta tarea, el *behavior set* vuelve al conjunto secundario, pero dado que sigue siendo la única tarea que falta por terminar sale de él al instante y el robot se queda intentando terminar la tarea indefinidamente.

6.6. Conclusiones

Como se observó anteriormente, la extensión de la arquitectura realizada permitió no solo mantener el correcto funcionamiento visto en las pruebas de la sección anterior, sino además resolver el caso en el cual un robot intenta indefinidamente la misma tarea

dejando otras sin realizar. Se destaca además que esta implementación surge a partir de la arquitectura L-ALLIANCE planteada por Parker como extensión de la arquitectura ALLIANCE, por lo cual mantiene los mismos ideales y elementos clave de la arquitectura ALLIANCE.

Capítulo 7

Conclusiones y Trabajo Futuro

Presentaremos a continuación las conclusiones resultantes de este trabajo junto con algunas posibles ideas de trabajos futuros que se consideran interesantes para ser realizados a partir del trabajo realizado.

7.1. Conclusiones

En este trabajo se desarrolló una implementación de la arquitectura ALLIANCE, la cual fue construida sobre la biblioteca Torocó. Para eso fue necesario realizar un análisis tanto de la arquitectura ALLIANCE y de su antecesora Subsumption, como de la biblioteca Torocó sobre la cual se construyó y de otros trabajos previos relacionados a estas arquitecturas. La implementación propuesta plantea mantener las características y funcionamiento de Torocó, pero agregando nuevas funcionalidades para la implementación de sistemas de control basados en ALLIANCE.

Para probar el correcto funcionamiento de la biblioteca se realizaron una variedad de pruebas en un entorno simulado mediante el uso del simulador MORSE, en el cual un conjunto de robots debían remover diferentes conjuntos de pelotas de una plataforma.

Tras analizar las pruebas realizadas se pudo comprobar que la implementación de la arquitectura era correcta y funcionaba correctamente, pero también se detectaron algunos puntos débiles de la arquitectura los cuales fueron analizados para buscar posibles soluciones. En particular se destaca el caso en el cual cuando se tiene un equipo de un solo robot y múltiples tarea, algunas teniendo prioridad ante otras, el robot siempre intentará realizar las tareas con prioridad, a pesar de ser incapaz de terminarlas. Esto causa que el robot se quede atascado intentando siempre realizar las mismas tareas e ignorando las tareas con menor prioridad, cuando tal vez él si fuera capaz de completarlas.

A modo de solucionar estos puntos débiles se decidió realizar algunos cambios a la arquitectura, los cuales fueron inspirados por la arquitectura L-ALLIANCE, también propuesta por Parker como una mejora a su arquitectura. De allí se tomó la idea de dividir las tareas en dos conjuntos, el conjunto primario del cual el robot elegirá su siguiente tarea a realizar, y el conjunto secundario en el cual el robot coloca las tareas que ya intentó pero no logró terminar. De esta forma el robot intenta todas las tareas posibles al menos una vez, comenzando por las que tienen más prioridad, y eventualmente

todas las tareas están o terminadas, o pertenecen el grupo secundario del cual el robot no elige tareas. Cuando sucede el primer caso la misión se da por terminada, y cuando sucede el segundo caso el robot pasará todas las tareas de nuevo al conjunto primario para intentarlas de nuevo.

Luego de haber realizado los nuevos cambios a la arquitectura, todas las pruebas en el simulador fueron realizadas de nuevo, tanto para comprobar que el comportamiento de las pruebas que si fueron exitosas se mantuvo igual, como para verificar que los casos que fueron destacados como punto débiles fueron resueltos exitosamente, y en ambos casos se obtuvieron los resultados esperados.

En conclusión, se realizó un análisis de la arquitectura ALLIANCE y se desarrolló una implementación sobre la cual, luego de algunas pruebas, se decidió agregar algunos elementos de la arquitectura L-ALLIANCE para solucionar algunos puntos débiles de la arquitectura. Se agrega además que, a pesar de haber encontrado algunos puntos a mejorar, la arquitectura ALLIANCE resultó práctica para coordinar grupos de robots de diferentes tamaños, brindando una arquitectura escalable y amigable para el desarrollador.

7.2. Trabajos Futuros

A pesar de que la implementación actual ya toma cierta inspiración de las mejoras propuestas en L-ALLIANCE, sería de interés agregar una implementación completa de la arquitectura que pueda manejar las diferentes estrategias vistas en la sección 6.1.4 y la actualización dinámica de los parámetros. La arquitectura N-ALLIANCE también provee varias mejoras tanto sobre ALLIANCE como L-ALLIANCE que resultan interesantes para agregar a la implementación actual.

Por otro lado, actualmente la implementación de ALLIANCE sobre Torocó solo fue probada en entornos virtuales utilizando el simulador Morse, lo cual hace que sea deseable realizar pruebas utilizando robots reales y de ser posible con grupos de robots más grandes.

También sería interesante probar la arquitectura con grupos de robots heterogéneos para observar su comportamiento cuando los robots tienen diferentes capacidades, configuraciones y objetivos.

Bibliografía

[Adept MobileRobots, 2003] Adept MobileRobots (2003). Amigobot. `https://www.generationrobots.com/en/402391-amigobot-wireless-ethernet.html`. Visitado: 29/07/2021.

[Arkin, 1998] Arkin, R. C. (1998). *Behavior Based Robotics*. MIT Press, Cambridge, MA, USA.

[Arvin et al., 2014] Arvin, F., Murray, J., Zhan, C. Z., and Yue, S. (2014). Colias: An autonomous micro robot for swarm robotic applications. *International Journal of Advanced Robotic Systems*, 11(7).

[Bettosini and Clavelli, 2015] Bettosini, I. and Clavelli, A. (2015). Torocó sistema de control de robots basado en comportamientos. Technical report, Universidad de la República.

[Blender Foundation, 2010] Blender Foundation (2010). Rblender. `https://www.blender.org/`. Visitado: 29/07/2021.

[Brambilla et al., 2013] Brambilla, M., Ferrante, E., Birattari, M., and Dorigo, M. (2013). Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41.

[Brooks, 1986] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1).

[Cragg and Hu, 2004] Cragg, L. and Hu, H. (2004). Implementing alliance in networked robots using mobile agents. *IFAC Proceedings Volumes*, 37(8):376–381.

[d. F. Roberto Ierusalimschy and Celes, 1993] d. F. Roberto Ierusalimschy, L. H. and Celes, W. (1993). Lua. `http://www.lua.org/about.html`. Visitado: 29/07/2021.

[dos Reis and Bastos, 2016] dos Reis, W. P. N. and Bastos, G. S. (2016). Implementing and simulating an alliance-based multi-robot task allocation architecture using ros. In Santos Osório, F. and Sales Gonçalves, R., editors, *Robotics*, pages 210–227, Cham. Springer International Publishing.

[Echeverria et al., 2012] Echeverria, G., Lemaignan, S., Degroote, A., Lacroix, S., Karg, M., Koch, P., Lesire, C., and Stinckwich, S. (2012). Simulating complex robotic scenarios with morse. In *SIMPAR*, pages 197–208.

[iRobot, 2012] iRobot (2012). irobot atrv platform. `https://www.openrobots.org/morse/doc/latest/user/robots/atrv.html`. Visitado: 29/07/2021.

[K-Team, 1999] K-Team (1999). Khepera. `https://www.k-team.com/khepera-iv`. Visitado: 29/07/2021.

[León et al., 2010] León, S., Ulbrich, R., Diankov, G., Puche, M., Przybylski, A., Morales, T., Asfour, S., Moisio, J., Bohg, J., R., K., and Dillmann (2010). Opengrasp. `http://opengrasp.sourceforge.net/`. Visitado: 29/07/2021.

[López, 2004] López, G. D. T. (2004). Contribución al diseño de sistemas multi robots utilizando alliance. Master's thesis, PEDECIBA Informática.

[Nehab, 2004] Nehab, D. (2004). Luasocket. `http://w3.impa.br/~diego/software/luasocket/home.html`. Visitado: 29/07/2021.

[Open Robotics, 2007] Open Robotics (2007). Robot operating system. `"https://www.ros.org/`. Visitado: 29/07/2021.

[Parker, 1994] Parker, L. E. (1994). Alliance: an architecture for fault tolerance multi-robot cooperation. *IEEE Transactions on Robotics and Automation*.

[Parker, 1998] Parker, L. E. (1998). L-alliance: Task-oriented multi-robot learning in behavior based systems. *Journal of Advanced Robotics*.

[Python Software Foundation, 2021] Python Software Foundation (2021). Python language reference. `http://www.python.org`. Visitado: 29/07/2021.

[Software Freedom Conservancy, 2014] Software Freedom Conservancy (2014). Gitlab. `https://gitlab.com/`. Visitado: 29/07/2021.

[Visca, 2012a] Visca, J. (2012a). Lumen. `https://github.com/xopxe/Lumen`. Visitado: 29/07/2021.

[Visca, 2012b] Visca, J. (2012b). Toribio. `https://github.com/xopxe/Toribio`. Visitado: 29/07/2021.

[Zaxmy, 2003] Zaxmy, J. (2003). Yaks - yet another khepera simulator. `http://freshmeat.sourceforge.net/projects/yaks/`. Visitado: 29/07/2021.

Apéndice A

Interfaz

Veremos en esta sección qué funciones fueron agregadas o actualizadas sobre las proporcionadas por Torocó para poder utilizar las funcionalidades de ALLIANCE. Las funciones que no son mencionadas se mantuvieron igual a su implementación en Torocó o son de uso interno de la biblioteca y por lo tanto no forman parte de la interfaz utilizada por los usuarios. En todos los casos se proporcionan ejemplos de código demostrando el uso de la funcionalidad.

A.1. Asignar entradas

La función *set_inputs*(*receiver_desc*, *inputs*) fue actualizada para también poder ser utilizada para definir las entradas de un *motivational behavior*. El parámetro *receiver_desc* ahora puede ser el descriptor de un comportamiento, actuador o de un *motivational behavior*.

```
...
toroco.set_inputs(
  motivational_behavior.motivational_behavior_name_1,
  {
    input_name = input
  }
)
...
```

A.2. Crear nuevos elementos de la arquitectura

A.2.1. Crear behavior set

Se implementó la nueva función *load_behavior_set*(*behavior_set_desc*, *behaviors*) la cual permite agrupar varios *behaviors* en un *behavior set*. Para esto toma como parámetros el descriptor de un *behavior set* y una tabla con los descriptores de los *behaviors* que lo van a formar.

```
...
toroco.load_behavior_set (
    behavior_set.behavior_set_name_1,
    {
        behavior.behavior_name_1,
        behavior.behavior_name_2,
    }
)
...
```

A.2.2. Crear motivational behavior

La función *load_motivational_behavior*(*motivational_behavior_desc*, *pathname*, *behavior_set*, *params*) permite al usuario crear un nuevo *motivational behavior*. Para esto toma como parámetros el descriptor del *motivational behavior*, la ruta del archivo en el cual se encuentra, el descriptor del *behavior set* que le corresponde, y una tabla con los parámetros necesarios para configurarlo.

```
...
toroco.load_motivational_behavior (
    motivational_behavior.motivational_behavior_name_1,
    'motivational_behaviors/motivational_behavior_file_1',
    behavior_set.behavior_set_name_1,
    {
        impatience = {
            slow_rate = 1,
            fast_rate = 3,
            affect_time = 100
        },
        acquiescence = {
            yield_time = 20,
            give_up_time = 40
        }
    }
)
...
```

A.3. Configuración

A.3.1. Configurar el nivel de activación

Para configurar el nivel de activación global y cambiarlo de su valor por defecto de 100, se provee la función *set_motivation_threshold*(*threshold*). Esta función toma como parámetro el nuevo valor que será utilizado.

```
...
toroco.set_motivation_threshold(new_value)
```

```
...
```

A.3.2. Configurar iteraciones

A modo de configurar el tiempo entre iteraciones del *coordinator*, el cual es de 1 segundo por defecto, la biblioteca expone la función *set_iteration_timeout*(*timeout*) la cual permite cambiarlo. Esta función toma un solo valor el cual deberá ser el tiempo en segundos entre iteraciones.

```
...
toroco.set_iteration_timeout(new_timeout)
...
```

A.4. Comunicación

A.4.1. Mensaje recibido

Para notificar a la biblioteca de que el robot recibió un nuevo mensaje se provee de la función *message_received*(*robot_id*, *behavior_desc*). Esta función toma como parámetros el identificador del robot que envió el mensaje y el nombre del *behavior set* correspondiente al mensaje recibido.

```
...
toroco.message_received(robot_id, behavior_name)
...
```

A.4.2. Configurar envío de mensajes

Para que el usuario pueda brindarle a la biblioteca de una forma para enviar mensajes se proporciona la función $configure\_notifier(notifier\_function)$. Al llamar a la función se le deberá pasar como argumento una función, la cual la librería se encargará de llamar cada vez que esta envíe un mensaje.

```
...
toroco.configure_notifier (notifier_function)
...
```

Glosario

Actuador Herramienta utilizada por un robot para obtener interactuar con su entorno.

Agente movil Programa autónomo con la capacidad de moverse atravez de diferentes nodos en una red.

ALLIANCE Arquitectura de software para enjambres de robots móviles.

Arquitectura de control Especificación de la estructura de un sistema de control.

ATRV Robot creado por la compañía iRobot y utilizada en el simulador MORSE.

Colias Robot utilizado en robótica de enjambres para simulador el comportamiento de enjambres de abejas.

Comportamiento Módulo que permite procesar los datos de eventos de entrada y emitir eventos de salida.

Devise Modulo que permite la comunicación con un dispositivo de hardware o simulación de uno.

Driver Controlador, rutina o programa que enlaza un dispositivo periférico al sistema operativo.

Emisor Sensor o comportamiento que envía eventos.

Framework Marco o esquema de trabajo utilizado en el desarrollo de software.

Host Computadora o dispositivo conectado a una red que provee o utiliza recursos de ella.

L-ALLIANCE Arquitectura de software para enjambres de robots móviles la cual construye sobre definido en ALLIANCE.

Lua Lenguaje de programación.

MORSE Simulador utilizado para pruebas de robots en entornos 3D.

N-ALLIANCE Arquitectura de control basada en las ideas principales de ALLIANCE.

Paradigma de control Caracterización general de una familia de sistemas de control.

Python Lenguaje de programación.

Receptor Actuador o componente que recibe eventos..

ROS Sigla para Robot Operating System, framework open-source para el desarrollo de robots.

Sensor Herramienta utilizada por un robot para obtener información de su entorno.

Sistema de control Programa que permite definir la conducta de un robot.

Subsumption Arquitectura de control basada en comportamientos.

Supresión Metodo de interaccion entre comportamientos, por el cual el supresor anula un evento de entrada de un receptor.

Torocó Biblioteca para la implementación de sistemas de control con arquitectura Subsumption.

YAKS Simulador de robots Khepera (Yet Another Khepera Simulator).