UNIVERSIDAD VERACRUZANA

FACULTAD DE FÍSICA E INTELIGENCIA ARTIFICIAL

DEPARTAMENTO DE INTELIGENCIA ARTIFICIAL

"Un Enfoque de Agentes y Artefactos para el Control de Componentes de un Robot Cognitivo"

TESIS

Que para obtener el grado de :

Maestro en Inteligencia Artificial

Presenta:

Esteban Antonio Castro Estrada

Director:

Dr. Alejandro Guerra Hernández

Co-director:

Dr. Héctor Gabriel Acosta Mesa

Xalapa, Veracruz. 31 de Enero de 2014

Índice general

CAPÍTULO

1

Introducción

1.1 Marco Teórico

Recientemente ha emergido un nuevo campo dentro de la Inteligencia Artificial (IA) llamado robótica cognitiva, el cual se enfoca en el desarrollo de robots capaces de generar inteligencia y comportamientos que apuntan a ser similares al del humano mediante la integración de percepciones, acciones, aprendizaje, toma de decisiones e interacción individual o social con el ambiente. La robótica cognitiva intenta incorporar conceptos de las ciencias cognitivas dentro de arquitecturas robóticas que simulen el procesamiento de información de la cognición humana. Aunque actualmente no exista alguna teoría que defina concretamente cuáles son los requerimientos de un robot para que éste muestre un proceso cognitivo real, especialmente como el del humano, la comunidad investigadora en este campo ha desarrollado arquitecturas robóticas que representan de algún modo componentes aislados de la cognición humana como la atención, memoria, la capacidad de toma de decisiones, resolución de problemas y aprendizaje [1].

Algunas investigaciones, entre las que destacan [2–6], han obtenido resultados positivos al utilizar lógica de primer orden para abordar algunos de los retos que presenta la robótica cognitiva, debido a que la semánticas de ésta, así como expresividad y motores de inferencias permiten representar y manipular conocimiento,

planes, metas y modelado del ambiente de una manera más simple y ordenada que con lenguajes procedurales [7].

Al mismo tiempo, otro campo de la IA en el cual también se ha recurrido al uso de la lógica de primer orden para modelar la cognición humana, es el campo de Sistemas Multi-Agentes (SMA). Éste se concentra en el estudio y desarrollo de entidades computacionales llamadas agentes.

Un agente es un sistema reactivo que presenta un grado de autonomía, el cual al situarlo dentro de un ambiente al que puede percibir y en el cual puede actuar, éste determina por sí mismo cómo llevar a cabo sus metas planteadas. Sin embargo, la parte más importante de un agente es cómo decidir qué hacer. Esta cuestión ha impulsado a filósofos e investigadores de inteligencia artificial para desarrollar arquitecturas de agentes inspiradas en el razonamiento practico humano, entre la que destaca la arquitectura de agente BDI (Belief-Desire-Intention) [8]. Los agentes BDI pueden ser vistos como programas computacionales provistos de estructuras de datos que representan analogías computacionales de estados mentales como creencias, deseos e intenciones. Estas analogías computacionales incorporadas dentro de un ciclo de razonamiento permiten a los agentes tomar decisiones, generar planes y actuar dentro de su ambiente para lograr sus metas de manera autónoma.

1.2 Planteamiento del problema

Actualmente existen una gran variedad de lenguajes de programación de agentes (LPA) que han sido diseñados para el desarrollo e implementación de agentes BDI. Ejemplos de éstos son 2APL [9], AgentSpeak(L) [10], Jason [11], GOAL [12], entre otros [13]. Sin embargo, a pesar de que la teoría de agentes ya se ha utilizado con anterioridad para el campo de la róbótica, ningún lenguaje orientado a agentes ha sido diseñado específicamente para la programación de robots. Más allá de ello, existen investigaciones recientes en las que se han desarrollado interfaces entre frameworks de sistemas multiagentes y plataformas robóticas para la programación de agentes que desempeñen el papel del control cognitivo dentro de una arquitectura robótica [14–16], e incluso se han planteado algunos de los requerimientos de LPA para la programación de robots cognitivos [17]. Estas investigaciones surgen con la intención de explotar las propiedades de los LPA que favorecen el desarrollo

de controles de robots cognitivos, como la representación simbólica del conocimiento y selección de acciones de alto nivel por medio de eventos.

Algunas de las arquitecturas recientes propuestas para robots cognitivos controlados por agentes comparten la base estructural inspirada en la clásica arquitectura robótica de tres capas [18], la cual destaca por su capacidad de integrar la reactividad y deliberación necesarias para los sistemas robóticos actuales. Este tipo de arquitecturas cuentan con tres capas principales: Una capa funcional encargada del control del hardware y proveer la capacidad de percepciones y acciones de bajo nivel; una capa deliberativa encargada de producir planes para lograr las metas del robot; finalmente, una capa ejecutiva que reside entre las otras dos que se encarga principalmente en la ejecución de planes. La forma en que ha sido integrada esta arquitectura en robots cognitivos ha sido mediante la sustitución de la capa deliberativa, compuesta generalmente por planificadores automáticos y los componentes que requerían mayor cantidad de recursos computacionales, por una capa cognitiva compuesta por un agente autónomo con la capacidad de soportar representaciones simbólicas del ambiente para facilitar la selección de acciones de alto nivel sin descuidar la capacidad reactiva, además de proporcionar abstracciones de procesos complejos que facilitan el desarrollo de sistemas robóticos.

Uno de los retos a los que se han enfrentado los investigadores para el desarrollo de robots cognitivos controlados por agentes es la adaptación entre el LPA con el framework o plataforma robótica en cuestión. Para lograrlo han desarrollado sofisticadas interfaces, compuestas por uno a varios módulos, que permiten tranformar la información proveniente de los sensores en representaciones simbólicas para adaptarlas como creencias del agente y facilitar la deliberación, así como interpretar las acciones del agente y transformalas en secuencias de instrucciones para los actuadores. Sin embargo, al ser un campo de investigación poco explorado, una característica común en ellas es la dificultad que presentan, para quienes deseen reutilizar dichas implementaciones, en la manipulación o modificación de los componentes de sus arqutecturas. Ya sea por que requieren de una gran cantidad de frameworks auxiliares para su composición, utilizando distintos lenguajes de programación, o por incorporar de una manera poco modular cada uno de sus componentes.

Es bien sabido que una arquitectura limpia y ordenada puede tener ventajas significativas en la especificación, ejecución y validación de un sistema robótico. Por otro lado la separación de componentes en unidades modulares ayuda a incrementar la comprensión y la reutilización, además de facilitar la depuración y validación individual de cada unidad.

En base a los trabajos relacionados más recientes enfocados en el diseño de arquitecturas robóticas cognitivas controladas por agentes, el reto de diseñar e implementar un interfaz entre un agente y algún framework o plataforma robótica de una manera ordena de tal forma que ésta sea compresible y de fácil reutilización sigue siendo un tema abierto, este tema aun es muy nuevo como para que se compruebe cuáles son las mejores metodologías o herramientas para su impliementación, por lo que es necesario explorarlo desde diferentes enfoques.

1.3 Antecedentes

Cada trabajo relacionado a éste propone una forma particular de implementar la interfaz entre el LPA (usado dentro de la capa cognitiva dentro de su arquitectura) y la plataforma robótica (capa funcional) en cuestión. Éstas varían desde formas simples como la definición de acciones del agente que encapsulan funciones propias de la plataforma robótica, hasta arquitecturas más complejas en las que la interacción entre el agente y la plataforma robótica no es directa, sino que se ven involucradas una o más capas en las que se llevan a cabo procesos separados, con lo que se facilita el cómputo deliberativo.

El más simple de estos, el trabajo de Jensen [14], consiste en una interfaz entre un agente programado en el intérprete Jason y la plataforma robótica Lego Mindstorms NXT programada en una versión de Java para la misma plataforma llamada leJOS [19]. Al ser implementados en Java ambos framworks de programación, el acoplamiento entre éstos es directo, mediante la inclusión de bibliotecas de leJOS dentro de una clase extendida de la clase base de arquitectura de agente Jason. Esta nueva clase contiene acciones y percepciones de agente que encapsulan funciones de leJOS para el manejo de motores y adquisición de señales de los sensores.

Por otra parte, se encuentra el trabajo de Wei et al [16] que presenta una arquitectura para robots cognitivos basados en agentes. Ésta se constituye en capas

sobrepuestas en las que se dividen los objetivos de programación por su nivel de abstracción. La capa más alta, capa cognitiva, utiliza información simbólica de alto nivel de abstracción para la toma decisiones, ésta se compone por un agente programado en GOAL [12]; la capa más baja, encargada del control del hardware, utiliza el lenguaje de programación robótico URBI[1] con el que es posible programar diversas plataformas robóticas; la capa intermedia, la cual maneja información sub-simbólica, es utilizada para el control de comportamientos, ésta se encuentra dividida en módulos programados en C++ encargados principalmente en el procesamiento de datos obtenidos por sensores, y la ejecución de comportamientos y acciones. Adicionalmente a las tres capa principales se encuentra una capa de interfaz entre las dos capas superiores, programada en el software EIS [20] (utilizado para el enlace entre agentes programados en diversos LPA con ambientes), la cual es utilizada para la traducción de información sub-simbólica a representaciones simbólicas, para ello utiliza un esquema de codificación que indica cómo interpretar y mapear cada dato procesado por la capa intermedia de una forma subsimbólica a una forma simbólica que facilita la toma de decisiones de la capa cognitiva.

Otro trabajo relacionado es el de Mordenti [15], el cual se enfoca en la exploración del uso del intérprete Jason junto con el framework *CArtAgO* (descrito en la sección 2.5) para el control de robots simulados en Webots[2]. Este trabajo también presenta una arquitectura de tres capas compuesta por un agente programado en Jason en la capa cognitiva, un artefacto programado en *CArtAgO* en la capa ejecutiva, y un robot simulado en Webots como capa funcional. El artefacto utilizado en la capa ejecutiva representa un módulo compuesto por un conjunto de comportamientos ad-hoc de acuerdo con los requerimientos de las tareas del robot, los cuales son disparados por el agente de la capa cognitiva. *CArtAgO* utiliza el lenguaje de propósito general Java para la programación de sus artefactos, con lo cual las funciones de Webots son invocadas desde el artefacto de la capa ejecutiva.

[1] http://www.urbiforge.org/
[2] http://www.cyberbotics.com/

1.4 Hipótesis

Siguiendo la línea de investigación de los trabajos relacionados más recientes enfocados en el diseño de arquitecturas robóticas cognitivas controladas por agentes [14–17], la hipótesis sugerida para esta tesis es que se puede implementar una capa de control de componentes físicos de un robot, de manera modular mediante el empleo de artefactos, dentro de una arquitectura de robot cognitivo basado en la arquitectura de tres capas como una posible alternativa para facilitar la comprensión, mantenimiento y reutilización de frameworks para la implementación de robots cognitivos.

1.5 Justificación

Con el enfoque propuesto en esta tesis se espera ofrecer una noción de equivalencia entre los componentes físicos de un robot y módulos dentro de una capa de la arquitectura del control robótico, representándolos de manera individual con cada módulo. Con lo cual se espera proveer un sistema flexible y reconfigurable, con la propiedad de permitir al diseñador de un sistema robótico adaptar únicamente aquellos componentes físicos que sean de su interés. Además de proveer la capacidad de extensibilidad dentro de la misma capa, haciendo que los controles de componentes físicos no sean los únicos componentes robóticos incorporados dentro de ésta, sino que también pueda ser posible añadir procesos auxiliares, provistos por algún otro software existente, dentro de otros módulos usando artefactos, de tal forma que faciliten al robot llevar a cabo sus tareas.

1.6 Objetivos

El objetivo principal de este proyecto es desarrollar un capa de control de componentes físicos dentro de una arquitectura robótica cognitiva. La cual pueda ser empleada para la implementación de robots cognitivos utilizando un agente programado en Jason como parte cognitiva, artefactos del ambiente de agentes *CArtAgO* para el control de los componentes físicos y la platofma robótica Lego Mindstorms NXT programada en leJOS [19] como parte funcional. Se espera que ésta cuente

con la capacidad para ser utilizada en aplicaciones como el diseño de experimentos en ambientes físicos o bien en el campo de la robótica educativa. Para conseguir esto se han abordado los siguientes puntos:

1. Desarrollo de un conjunto de artefactos *CArtAgO* que representen los respectivos controles para cada componente físico de la plataforma robótica Lego Mindstorms NXT.

2. Desarrollo de artefacto auxiliares que sirvan para la comunicación con la plataforma robótica y manejo de los artefactos anteriormente mencionados.

3. Desarrollo de un programa genérico en leJOS (ver sección 4.2), sobre la plataforma robótica en cuestión, que sirva como la capa funcional de la arquitectura implementada, permitiendo ser utilizado sobre cualquier robot bajo la misma arquitectura propuesta sin la necesidad de ser modificado, a no ser que sea con fines de ser extendido, mejorado o depurado.

4. Diseño e implementación de un conjunto de prácticas para la verificación de la funcionalidad de la arquitectura y principalmente de la capa de control de componentes propuesta.

1.7 Exposición y métodos

Este documento se encuentra dividido en dos partes:

- **Parte I**: En los capítulos 2-4 se presentan los conceptos básicos para comprender el contexto en el que se encuentra el presente trabajo de investigación. El capítulo 2 presenta una introducción a la programación orientada agentes, centrándose en los agentes BDI programados en Jason, así como el ambiente de agentes *CArtAgO*; en el capítulo 3 se presentan algunos antecedes de la robótica cognitiva, principalmente aquellos que están más relacionados con el presente trabajo, estos son aquellos que recurren al uso de lenguajes de programación de agentes BDI para la implementación de la parte cognitiva del robot; El capítulo 4 presenta una descripción de la plataforma robótica Lego Mindstorms NXT, utilizada para la implementación práctica de este proyecto de investigación.

- **Parte II**: En los capítulos 5-7 se expone la implmentación de la integración de la capa de control de componentes propuesta en este proyecto dentro de una arquitectura para el control de un robot cognitivo. En el capítulo 5 se describe la arquitectura, centrándose en la capa de control de componentes y su integración con las otras capas de la arquitectura; en el capítulo 6 se presentan los experimentos realizados para verificar el funcionamiento de la implementación; finalmente el capítulo 7 presenta una discusión sobre los resultados obtenidos en este trabajo, así como una comparativa con los trabajos relacionados.

Parte I

Antecedentes y Conceptos Básicos

CAPÍTULO

2

Agentes BDI y Artefactos

Antes de hablar sobre la capa de control de componentes para una arquitectura de robot cognitivo propuesta en este proyecto, es necesario introducir algunos conceptos básicos que permitan obtener una idea general del contexto en el que éste se encuentra. Empezamos por introducir a los agentes BDI y el ambiente de agentes basado en artefactos *CArtAgO*, ambos utilizados para el desarrollo de nuestra implementación. En este capítulo se brinda una breve reseña acerca de la programación orientada a agentes, y principalmente de los agentes BDI programados con el lenguaje de agentes *AgentSpeak(L)* y su intérprete Jason. Así mismo, se describe de manera general el ambiente de agentes basado en artefactos *CArtAgO*. Cabe mencionar que gran parte del contenido de este capítulo ha sido recopilado de [11, 21].

2.1 Programación orientada agentes

Para comprender qué es un *agente*, dentro del campo de *sistemas multi-agentes*, es útil considerar cómo se relaciona un agente con otro tipo de programas. Considerando un programa *funcional* que toma una entrada y tras procesarla produce alguna salida, también visto como una función $f : I \rightarrow O$ en donde un dominio I de posibles entradas puede producir algún rango O de posibles salidas. Este tipo

de programas posiblemente puedan parecer simples para el desarrollo de software, debido a que existen una gran variedad de técnicas para abordarlos. Desafortunádamente muchos otros tipos de programas no tienen este tipo de simple estructura operacional de entrada-salida. En particular nos referimos a aquellos programas que necesitan tener una conducta reactiva [22], es decir que tienen que mantener a largo plazo una interacción continua con su ambiente, éstos no simplemente computan alguna función de entrada y terminan.

Entre este tipo de programas se encuentran los sistemas operativos, sistemas de control, servidores web, entre otros. Por lo general este tipo de programas no pueden ser descritos adecuadamente desde el punto de vista *funcional*. Típicamente éstos son descritos y especificados en términos de sus comportamientos obtenidos sobre la marcha.

Por otra parte, como una subcategoría de los sistemas reactivos se encuentran los *agentes*. Éstos pueden ser considerados como sistemas reactivos situados dentro de algún ambiente. Dentro de éste los agentes son capaces de percibir su entorno, y de tener un repertorio de posibles acciones que puedan ejecutar con la finalidad de modificar el ambiente.

Si bien no existe una definición universal de lo que es un *agente* dentro del contexto de la Inteligencia Artificial, podemos tomar en cuenta dos de las definiciones más aceptadas. Por una parte Russel & Norving [23] proporcionan una definición general de agente:

Definición 2.1. *Un agente es cualquier cosa capaz de percibir su medio ambiente con la ayuda de sensores y actuar en ese medio utilizando actuadores.*

Mientras que Wooldridge [24] presenta una definición de agente en la que lo delimita al espacio computacional además de añadir el concepto de autonomía:

Definición 2.2. *Un agente es un sistema computacional que está situado en un ambiente y es capaz de actuar de manera autónoma para satisfacer sus objetivos.*

Puesto que las definiciones anteriormente mencionadas son demasiado generales, y una descripción más concreta está lejos de ser sencilla, nos apoyaremos para definir un agente en término de las propiedades clave que éstos deberían poseer, como lo describen Wooldridge y Jennings [25]:

Autonomía Los agentes funcionan sin la intervención directa de humanos u otras entidades, y tienen un tipo de control sobre sus acciones y estados internos.

Reactividad Los agentes perciben su ambiente (el cual puede ser el mundo físico, Internet, o algún otro tipo de ambiente computacional), y responden oportunamente a los cambios que ocurren en éste.

Iniciativa Los agentes no actúan simplemente en respuesta a su ambiente, sino que también son capaces de presentar comportamientos orientados por sus metas para satisfacer sus objetivos.

Habilidad social Los agentes interactúan con otros agentes para llevar a cabo actividades de cooperación o coordinación, de tal forma que las tareas pueden ser delegadas entre ellos.

Las propiedades descritas anteriormente tan solo son parte de una noción débil de agencia, una noción fuerte podría representar un mayor significado, sin embargo, para esta sección no es necesario profundizar en las nociones fuertes debido a que en ellas, además de involucrar la propiedades ya descritas, adicionalmente incluyen conceptualizaciones o implementaciones de nociones que son más aplicadas usualmente al humano. Por ejemplo, el enfoque utilizado en este trabajo considera la caracterización de un agente usando nociones mentales como creencias, deseos e intenciones [8], con esto nos referimos a los agentes BDI, descritos más detalladamente en la sección 2.2.

El concepto de *programación orientada a agentes* (POA) fue introducido por Yoav Shoham [26] con la finalidad de describir un paradigma de programación basado en el cómputo desde un punto de vista cognitivo y social, particularmente en términos de nociones mentales(como creencias, deceso e intenciones) que han sido desarrolladas para la representación de las propiedades de los agentes.

El papel que toma el programa de una agentes es controlar la evolución de los estados mentales del agente. Para poder considerar un sistema como POA completo, Shoham plantea que éste debe incluir los siguientes tres componentes:

- Un *lenguaje formal restringido* con sintaxis y semántica claras para poder describir los estados mentales del agente.

- Un *lenguaje de programación interpretado* en el cual los agentes puedan ser programados. La semántica del lenguaje de programación dependerá en parte de la semántica del estado mental del agente.

- Un *"agentificador"* que convierta entidades neutras en agentes programables.

El lenguaje formal utilizado para este trabajo de investigación es *AgentSpeak(L)* 2.3, así como su intérprete *Jason* 2.4 descritos más adelante dentro de este capítulo.

2.2 Agentes BDI

El modelo de agencia BDI (siglas en inglés de deseos, creencias e intenciones) está inspirado y basado en teorías del razonamiento práctico humano desarolladas por Michale Bratman [27], así como la postura intencional de Dennett [28] y la comunicación de actos de habla de Searle [29].

La primera idea necesaria para comprender el modelo BDI es tratar los programas de computadora como si tuvieran estados mentales. Así, al referirnos a sistemas con deseos, creencias e intenciones, se habla de programas de computadoras con analogías computacionales de deseos, creencias e intenciones. Por lo general una arquitectura construida con base en el modelo de agentes BDI incluye las siguientes estructuras de datos:

Creencias Es la información que el agente tiene sobre su ambiente. Las creencias son actualizadas a través de la percepción del agente y la ejecución de sus acciones.

Deseos Estos son todos las tareas asignadas al agente. Tener un deseo no implica que un agente lo llevará a cabo: se debe ver como una influencia potencial en el comportamiento del agente. Éstos pueden ser vistos como opciones de un agente.

Intenciones Representan los cursos de acción que el agente se ha comprometido a cumplir. Las intenciones pueden ser metas que el agente debe cumplir o pueden ser resultado de la consideración de opciones: se puede pensar en

un agente que visualiza sus opciones y escoge entre ellas. Las intenciones representan la parte deliberativa del agente.

Eventos Las percepciones del agente se mapean a eventos discretos almacenados temporalmente en una colección de eventos. Los eventos incluyen la adquisición o eliminación de creencias, la percepción de mensajes en el caso de sistemas multi-agentes, y la adquisición de una nueva meta.

Planes Los planes de un agente constituyen los posibles cursos de de acción que el agente puede ejecutar como una respuesta a los eventos ocurridos.

El modelo práctico de toma de decisiones que subyace bajo el modelo BDI es conocido como *razonamiento práctico*. Éste consiste en un proceso compuesto por la siguientes actividades:

- *Deliberación* que consiste en la adopción de intenciones.
- *Razonamiento medios-fines* que consiste en la determinación de los medios para llevar a cabo las intenciones.

Las intenciones juegan un papel clave dentro del modelo BDI. En primer lugar, porque las intenciones proporcionan una *iniciativa*, motivando al cumplimiento de metas, de esta manera las intenciones son controladoras de conducta. En segundo lugar, las intenciones son persistentes proporcionando una inercia, es decir que una vez que son adoptadas éstas se resisten a ser abandonadas. Sin embargo, si la razón por la cual se creó la intención desaparece, entonces es posible abandonar dicha intención. En tercer lugar, las intenciones previenen posibles conflictos provocados por la incompatibilidad entre la intención adoptada y otras intenciones, haciendo que el agente no considere aquellas intenciones que son incompatibles con la adoptada.

Por otra parte, el *razonamiento medios-fines* es el proceso de decisión del cómo lograr un fin (como una intención adoptada) por medio de los medios disponibles para el agente (como las acciones que puede ejecutar dentro de su ambiente). El *razonamiento medios-fines* de manera general puede entenderse dentro de la comunidad de IA como un tipo de *planificador*. Un *planificador* es un sistema que toma como entrada la representación de:

- Una *meta*, o *intención*: algo que el agente quiera lograr.
- Las *creencias* presentes en el agente acerca del estado del ambiente.
- Las *acciones* disponibles para el agente.

Como salida, un algoritmo planificador genera un *plan*, o en términos simples, una receta". Uno de los primeros enfoques para la generación de planes era mediante el ensamblaje de un curso de acción completo (esencialmente un programa) cuyos componentes atómicos son acciones disponibles para el agente. El mayor problema de este enfoque es su alto costo computacional. No obstante, han ocurrido avances significativos en esta área de investigación con el desarrollo de algoritmos eficientes para la implementación de planificadores.

Uno de los enfoques de planificación que han obtenido un buen resultado en la practica para la implementación del *razonamiento medios-fines* dentro de lenguajes de agentes se puede describir de la siguiente manera: el programador elabora una colección de planes parciales para un agente al momento de diseñarlo, y la tarea del agente es entonces ensamblar estos planes parciales para su ejecución al vuelo, y no offline como se solía hacer en los primeros enfoques de planificación. Con este enfoque, la habilidad de un agente para ajustarse a las circunstancias depende del código de los planes parciales que el programador escribe para el agente.

Existen diferentes algoritmos para la implementación del razonamiento práctico de un agente BDI. Uno de éstos es presentado por Bordini a manera de ejemplo ([11], pag. 21), cuyo pseudocódigo se muestra en el de la figura 2.1. En donde las variable B, D e I representan creencias, deseos e intenciones respectivamente. En este algoritmo pueden observarse dos ciclos principales, uno interno y otro externo, dentro del ciclo externo (líneas 4-24), el agente observa su ambiente para obtener la siguiente percepción. En la línea 6 actualiza sus creencias a través de la *función de revisión de creencias* (*frc*), en la cual toma las creencias actuales del agente y la nueva percepción (ρ), y regresa las nuevas creencias del agente (las cuales son resultado de actualizar B con ρ).

En la línea 7, el agente determina sus deseos, o posibles opciones, con base en sus creencias actuales y sus intenciones. Ésto se lleva a cabo usando la función *opciones(...)*. El agente entonces escoge entre estas opciones, seleccionado algunas

para convertirlas en intenciones (línea 8) y genera un plan para lograr sus intenciones a través de una función *plan()*(línea 9).

```
1: procedure AGENTE − BDI(planes, creencias, deseos)
2:     B ← B_0;                                   ▷ B_0 son las creencias iniciales
3:     I ← I_0;                                   ▷ I_0 son las intenciones iniciales
4:     while true do
5:         se obtiene la siguiente percepción ρ a través de los sensores;
6:         B ← frc(B, ρ);
7:         D ← opciones(B, I);
8:         I ← filtro(B, D, I);
9:         π ← plan(B, I, Ac);                    ▷ Donde Ac es un conjunto de acciones
10:        while not (vacio(π) or (logrado(I, B) or imposible(I, B)) do
11:            α ← primer elemento de π;
12:            ejecuta(α)
13:            π ← cola de π
14:            Observar ambiente para obtener siguiente percepción ρ;
15:            B ← frc(B, ρ);
16:            if reconsidera(I, B) then
17:                D ← opciones(B, I);
18:                I ← filtro(B, D, I);
19:            end if
20:            if not valido(π, I, B) then
21:                π ← plan(B, I, Ac);
22:            end if
23:        end while
24:    end while
25: end procedure
```

Figura 2.1: Ciclo de control para el razonamiento práctico de un agente BDI

El ciclo interno (líneas 10-23) captura la ejecución de un plan para realizar las intenciones del agente. Si todo funciona bien, entonces el agente simplemente toma cada acción en turno de su plan y la ejecuta, hasta que el plan π se encuentre vacío (*vacio(...)*), es decir, que todas sus acciones hayan sido ejecutadas. Sin embargo, después de ejecutar una acción del plan (línea 12), el agente hace hace una pausa para observar su ambiente, actualizando sus creencias de nuevo. Entonces, se pregunta a sí mismo si vale la pena reconsiderar sus intenciones a través de la función *reconsidera(...)*.

Finalmente, independientemente de haber decidido reconsiderado o no, el agente se pregunta a sí mismo si el plan actual es válido (línea 20) con respecto a sus intenciones (qué quiere lograr con ese plan) asi como con sus creencias (cuál cree

que es el estado del ambiente). Si cree que el plan ya no es válido, lo replantea (línea 21).

2.3 AgentSpeak(L)

AgentSpeak(L)[10] es un lenguaje de programación de agentes BDI. Se basa en una lógica restringida de primer orden con eventos y acciones. Para *AgentSpeak(L)*, el estado actual del agente (el cuál es un modelo de sí mismo), su ambiente, y otros agentes (para el caso de sistemas multi-agentes), pueden ser vistos como las *creencias* presentes en el agente. Los estados que el agente quiere lograr con base en sus estímulos internos y externos, forman parte de sus *deseos*, y la adopción de programas para satisfacer estos deseos, constituyen sus *intenciones*.

Un agente programado en AgentSpeak(L) se compone básicamente por:

Base de creencias Se constituye de una colección de literales de la lógica de primer orden. Éstas representan las creencias que tiene el agente acerca de su entorno, de otros agentes o de sí mismo.

Biblioteca de planes Se constituye por una colección de planes que el agente tiene como recetas o procedimientos.

Adicionalmente, el lenguaje *AgentSpeak(L)* se basa en los siguientes constructores:

Creencias Son representadas como literales de la lógica de primer orden. Los átomos de creencia toman formas como $adyacente(X, Y)$ ó $pos(Robot, X)$, etc. Las creencias base del agente son casos sin variables libres de estas creencias atómicas, por ejemplo: $pos(r1, c(1, 3))$, etc., donde $c(i, j)$ denota una posición discreta (i, j) mientras que r_j al objeto j en el ambiente. Una cadena de caracteres que inicia con mayúscula es una *variable* mientras que en caso contrario se trata de una *constante*.

Metas Los agentes en AgentSpeak(L), consideran dos tipos de meta: las *metas alcanzables* (*achieve goals*) que tienen la forma $!g(t)$ y representan fórmulas que el agente desea sean satisfechas; las *metas verificables* (*test goals*) tienen

la forma $?g(t)$ y son estados que el agente desea probar si son verdaderos. Las meta verificables se asemejan a las *consultas* (*Queries*) de la Programación Lógica.

Eventos disparadores Los cambios en el entorno del agente y en su estado interno generan *eventos disparadores* (*trigger events*). Estos eventos incluyen agregar $(+)$ y borrar $(-)$ metas o creencias del estado del agente. Por ejemplo, un robot que evade obstáculos al detectar un obstáculo en la posición $c1$, además de agregar la creencia correspondiente, produce el evento disparador $+pos(obstaculo, c1)$.

Acciones El agente debe ejecutar acciones para lograr el cumplimiento de sus metas. Las *acciones* pueden verse como procedimientos a ejecutar. Por ejemplo, una acción como $ir(X, Y)$ debería tener como resultado que el agente se sitúe en la posición Y y deje de estar en X.

Planes Cada plan se constituye por un evento disparador que especifica cuándo debe ejecutarse dicho plan, un contexto que determina si el plan puede ejecutarse y un cuerpo que determina los posibles cursos de acción a ejecutar. Los planes de un agente en AgentSpeak(L) toman la forma $evento{-}disparador : contexto \leftarrow cuerpo$. Donde el *evento-disparador* y el *contexto* forman parte de la *cabeza del plan*. El *contexto* se compone de un conjunto de literales que deben cumplirse como verdaderas dentro de la base de creencias del agente para que el plan sea aplicable. Mientras que el *cuerpo* del plan es una secuencia de fórmulas, representadas por metas o acciones. Por conveniencia un contexto o cuerpo vacío se escribe $true$. Por ello un plan con un contexto vacío siempre es ejecutable.

Para ejemplificar los componentes anteriormente mencionados puede utilizarse como caso de estudio el escenario que se muestra en la figura 2.2.

El cuadro 2.1 muestra un *base de creencias* (líneas 1-6) y una *biblioteca de planes* (líneas 8-20) para el agente de nuestro caso de estudio. Cuando se encuentre una basura en alguno de los carriles del ambiente, el *evento disparador* $pos(basura, X)$ es generado y el plan $p0$ será aplicable cuando el robot se encuentre en el mismo

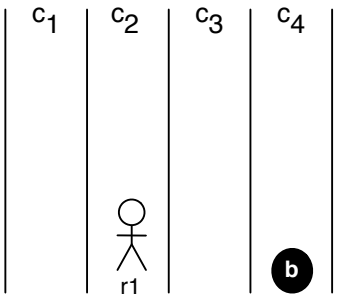

Figura 2.2: Escenario utilizado como caso de estudio: Una autopista de cuatro carriles $c_1 \ldots c_4$, donde $r1$ recoge la basura b.

carril que el de la basura encontrada y no es el mismo donde se encuentra el depósito (*contexto* del plan $p0$). Al ser el plan $p0$ aplicable, el robot intentará dicho plan mediante la ejecución del *cuerpo del plan* (líneas 10-12). Primero, el robot ejecutará la *acción* de levantar la basura (línea 10), y se planteará la *meta* alcanzable de ir al carril donde está el depósito (línea 11), para finalmente ejecutar la *acción* de tirar ahí la basura(línea 12). El comportamiento del agente se completa con los planes para ir de su carril a algún otro (planes $p1$ y $p2$).

2.4 Jason

Jason [11] es un intérprete del lenguaje de programación de agentes *AgentSpeak(L)*. Una característica que hace destacar a *Jason* es su versatilidad debido a que está implementado en el lenguaje de propósito general Java, lo que le atribuye propiedades tales como la ejecución en diferentes plataformas además de permitir la facilidad de ser extendido, debido a que además está distribuido bajo licencia libre GNU LGPL. Algunas otras características de Jason se mencionan a continuación:

- *Anotaciones sobre literales*, las cuales tienen usos preestablecidos, como la identificación de la fuente de una creencia; o definidos por el usuario, como en las funciones de selección configurables desde Java.

- *Anotaciones sobre los planes*, las cuales pueden ser usadas para diseñar funciones de selección basadas en Teoría de Decisión [30]. Estas funciones de

```
1  adyacente(c1,c2).
2  adyacente(c2,c3).
3  adyacente(c3,c4).
4  pos(r1,c1).
5  pos(basura,c2).
6  pos(deposito,c4).
7
8  [p0]
9  +pos(basura,X) : pos(r1,X) & pos(deposito,Y) <-
10   levantar(basura);
11   !pos(r1, Y);
12   tirar(basura).
13
14 [p1]
15 +!pos(r1,X): pos(r1,X) <- true.
16
17 [p2]
18 +!pos(r1,X): pos(r1,Y ) & not(X=Y) & adyacente(Y,Z) <-
19   ir(Y, Z);
20   !pos(r1,X).
```

Cuadro 2.1: Biblioteca de planes del agente en $AgentSpeak(L)$ del caso de estudio.

selección son totalmente configurables desde Java.

- *Acciones internas* programables en Java, las cuales afectan exclusivamente el estado de un agente, sin afectar su medio ambiente. Esto provee de extensibilidad al lenguaje.

- Una noción de *ambiente* programable en Java, incluyendo *acciones externas* definidas por el usuario y diferentes tipos de ambientes predefinidos. Una extensión más reciente es la adopción del modelo de Agentes y Artefactos [31] para trabajar con la definición e implementación de ambientes concebidos como una colección de herramientas, los artefactos, que un agente puede usar en su medio ambiente. Este tema se discutirá más detalladamente en la sección 2.5.

- Comunicación entre agentes basada en *actos de habla* [29]. En particular, se implementa el protocolo conocido como *Knowledge Query and Manipulation Language* (KQML)[32] y se consideran anotaciones en las creencias con información sobre las fuentes de los mensajes.

En el cuadro 2.2 se define la gramática simplificada del intérprete $Jason$. En ésta, el símbolo $\top$ denota elementos vacíos en el lenguaje, como una lista vacía o una secuencia vacía.

$$
\begin{array}{lcll}
ag & ::= & bs \quad ps & \\
bs & ::= & b_1 \ldots b_n & (n \geq 0) \\
ps & ::= & p_1 \ldots p_n & (n \geq 1) \\
p & ::= & te : ct \leftarrow h & \\
te & ::= & +at \mid -at \mid +g \mid -g & \\
ct & ::= & ct_1 \mid \top & \\
ct_1 & ::= & at \mid \neg at \mid ct_1 \wedge ct_1 & \\
h & ::= & h_1; \top \mid \top & \\
h_1 & ::= & a \mid g \mid u \mid h_1; h_1 & \\
at & ::= & P(t_1, \ldots, t_n) & (n \geq 0) \\
 & \mid & P(t_1, \ldots, t_n)[s_1, \ldots, s_m] & (n \geq 0, m > 0) \\
s & ::= & \texttt{percept} \mid \texttt{self} \mid id & \\
a & ::= & A(t_1, \ldots, t_n) & (n \geq 0) \\
g & ::= & !at \mid ?at & \\
u & ::= & +b \mid -at &
\end{array}
$$

Cuadro 2.2: Sintaxis de $Jason$ definida mediante una gramática BNF. Adaptada de Bordini et al. [11]

Un agente ag está definido por un conjunto de *creencias* bs y *planes* ps. Cada *creencia* $b \in bs$ toma la forma de un átomo de base (sin variables libres) de primer orden. Se denota como at a las fórmulas atómicas que no son de base (con variables). Debe observarse que los átomos vienen en dos modalidades, con y sin etiquetas. Las *etiquetas* s se usan para identificar la fuente de la fórmula atómica, a saber: el ambiente ($percept$), el agente mismo ($self$) u otro agente (id). Cada *plan* $p \in ps$ tiene la forma $te : ct \leftarrow h$, donde:

- te denota el *evento disparador* del plan que define el suceso ó evento para el cual el plan en cuestión es *relevante*, es decir, para qué evento sirve el plan. Los eventos a considerar son básicamente el agregar ($+$) ó eliminar ($-$) una creencia; ó una meta. Las *metas* (g)son de dos tipos: conseguir (!) que un átomo sea verdadero; y verificar (?) si lo es. Las primeras están asociadas al

razonamiento práctico del agente (¿Qué hago?); las segundas al epistémico (¿Qué sé?). El evento disparador de un plan no puede ser vacío. Debe observarse que se utilizan fórmulas atómicas at y no creencias en esta definición; esto es, los eventos disparadores pueden incluir variables en sus argumentos.

- ct denota el *contexto* del plan que define las condiciones que hacen que éste sea efectivamente *ejecutable*, es decir, cuándo se puede ejecutar el plan. Tales condiciones se expresan como literales (átomos o su negación) de primer orden o una conjunción de ellas. El contexto vacío está asociado a planes que son ejecutables en cualquier circunstancia del agente.

- h denota el *cuerpo* del plan. Un cuerpo no vacío es una secuencia finita (h_1) de *acciones* (a), metas (g) y actualizaciones de creencias (u). Las acciones pueden tener argumentos. Las actualizaciones incluyen agregar una creencia ($+b$) o eliminar una o un conjunto de ellas ($-at$). Cabe resaltar que agregar una creencia solo admite átomos de base como argumento.

El intérprete *Jason* utiliza un *ciclo de razonamiento* que puede ser representado de una manera simplificada como se muestra en la figura 2.3 (en esta representación se omiten los procesos relacionados con la comunicación). En la figura, los conjuntos (de creencias, eventos, planes e intenciones) son representados como rectángulos, los rombos representan funciones de selección (de uno o más elementos de un conjunto), mientras que los círculos representan alguno de los procesos involucrados dentro de la interpretación de los programas $AgentSpeak(L)$.

En cada ciclo de razonamiento de un programa de agente se actualiza una lista de eventos, la cual puede ser generada a partir de las percepciones del ambiente, o desde la ejecución de intenciones (cuando se ha especificado una meta dentro del cuerpo de un plan). Las creencias también son actualizadas desde las percepciones o desde operaciones controladas por un plan. Cada vez que exista un cambio en las creencias del agente se ve reflejado en la inserción de un evento en el conjunto de eventos. La actualización, tanto de la base de creencias como de la lista de eventos, se lleva a cabo medio de la *función de revisión de creencias*.

Durante un solo ciclo pueden presentarse varios eventos pendientes a la espera de ser procesados, por ejemplo cuando varios aspectos del ambiente han cambiado

recientemente, sin embargo, solo uno es abordado por cada ciclo de razonamiento. Es por ello que es necesario seleccionar un evento a manejar por medio de una *fucnión de selección de eventos* (S_E), la cual puede ser personalizada dependiendo de la aplicación, aunque por defecto ésta selecciona los eventos en el mismo orden en que se presentan.

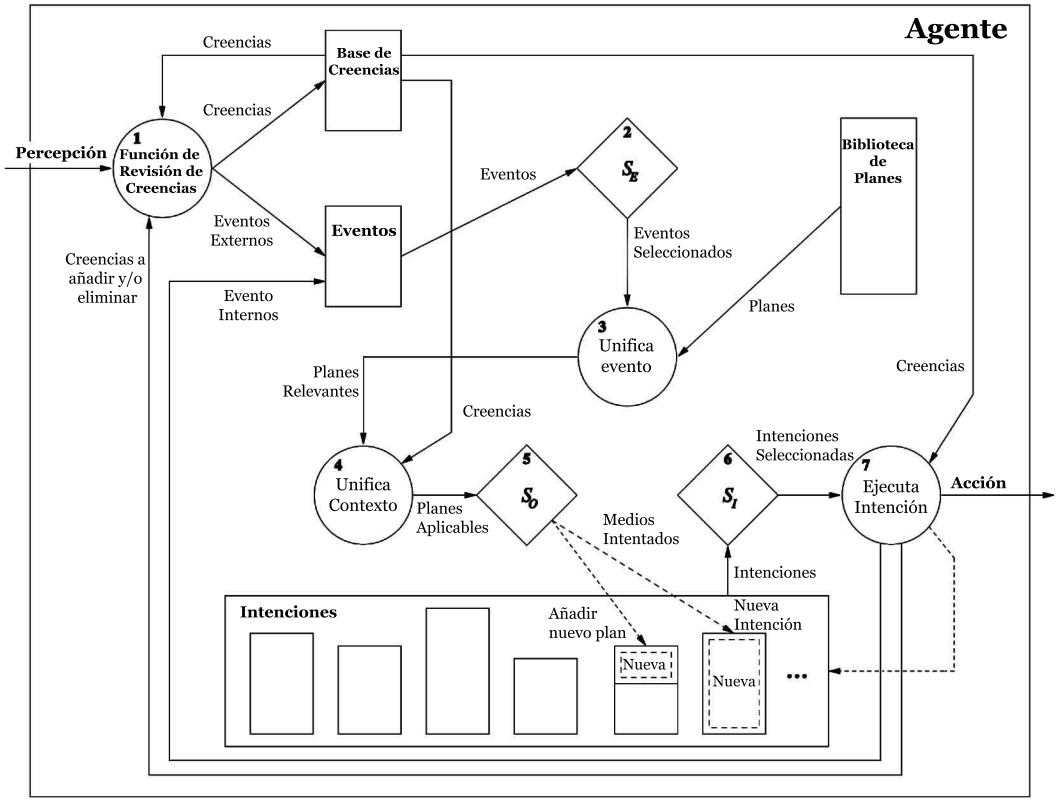

Figura 2.3: Versión simplificada del ciclo de razonamiento de un agente en Jason

Una vez que se ha seleccionado un evento pendiente a procesar, el intérprete tiene que encontrar un plan que le permita al agente actuar a fin de manejar dicho evento. Para ésto, primero se encuentra dentro de la *biblioteca de planes* todos los *planes relevantes* para el evento dado. Ésto se lleva a cabo mediante la devolución de todos los planes de la *biblioteca de planes* cuyo evento disparador en su *cabeza* unifique con el evento seleccionado. Posteriormente se revisa si el contexto de cada plan relevante se cumple como verdadero para las creencias, es decir que su contexto sea *consecuencia lógica* de la base de creencias del agente, si lo es, el plan es

considerado como *plan aplicable*.

Dado que pueden presentarse varios planes aplicables para un mismo evento, es necesario seleccionar solo uno para añadirlo al conjunto de intenciones. Este proceso se lleva a cabo con la *función de selección del plan aplicable* (S_O). El plan aplicable seleccionado es llamado *medios intentados*, esto es debido a que el curso de acción definido por el plan establece los medios para que el agente intente usar para manejar el evento pendiente a tratar.

En la figura 2.3 puede observarse que hay dos líneas punteadas desde la función de selección S_O hacia el conjunto de intenciones. Esto debido a que existen dos maneras diferentes de actualizar las intenciones, dependiendo del origen del evento. Si se presenta un cambio en las metas del agente se considera como un *evento interno* y cuando se presenta la percepción de un cambio en el ambiente se considera como un *evento externo*. Si los *medios intentados* son seleccionados a causa de un *evento externo*, entonces éste creara una nueva intención para el agente. Mientras que los *medios intentados* seleccionados a causa de un *evento interno* son colocados sobre una intención existente en forma de pila. Cada pila de intenciones del conjunto de intenciones representa un foco de atención diferente para el agente.

Comúnmente, un agente tiene más de una intención en su conjunto de intenciones, cada una representando un posible foco de atención, éstas compiten por se atendidas por el agente. Sin embargo, como máximo solo es posible ejecutar una fórmula de una sola intención durante un ciclo de razonamiento. Por tal motivo, antes de que el agente realice alguna acción, es necesario seleccionar una intención particular de entre las que se encuentren listas a ser ejecutadas, para ello es utilizada la *función de selección de intenciones* (S_I). Una vez seleccionada la intención, se ejecuta la fórmula por la que comienza el cuerpo del plan superior de la pila. Esto puede implicar tanto una acción del agente sobre el ambiente, así como la generación de un evento interno (cuando la fórmula seleccionada representa una meta alcanzables o una meta verificable) con lo cual se genera un nuevo ciclo de razonamiento. Finalmente, tras la ejecución de una fórmula y antes de empezar un nuevo ciclo, dicha fórmula es eliminada de la intención seleccionada. Cuando todas las fórmulas han sido eliminadas del cuerpo de un plan, el plan completo es removido de la pila de intenciones que se encuentre como foco de atención.

2.5 Ambiente de agentes CArtAgO

Los agentes programados en Jason cuentan con la posibilidad de ser modelados e implementados con base en una paradigma llamado Agentes y Artefactos (*A&A*) utilizando el framework *CArtAgO* [31]. Éste propone la noción de ambiente de trabajo, compuesto por un conjunto dinámico de entidades llamadas *artefactos*, los cuales son utilizados por los agentes para llevar a cabo sus objetivos. Los *artefactos* son *abstracciones* de primera clase, que permiten a los agentes interactuar con su ambiente desde una capa de abstracción de alto nivel.

Según el enfoque del modelo *A&A*, es posible diseñar y programar un ambiente en términos de un conjunto dinámico de *artefactos*, agrupados en localidades llamadas *espacios de trabajo*, posiblemente distribuidas en diferentes nodos de una red. Los *artefactos* representan recursos o herramientas que los agentes pueden dinámicamente *instanciar*, *compartir* y utilizar para soportar sus actividades tanto individuales como colectivas. Mientras que los *espacios de trabajo* representan zonas del ambiente relacionadas a una o múltiples actividades donde se encuentran determinados agentes y artefactos.

Por otro lado, la propiedad de abstracción de primera clase de los artefactos permite a los programadores definir tipos de artefactos que pueden ser *instanciados* dentro de algún *espacio de trabajo* específico, así como definir sus estructuras y comportamientos computacionales. También, debido a que los artefactos se consideran entidades de primera clase dentro un ambiente de agentes, los agentes pueden percibirlos, usarlos, complementarlo y manipularlos.

Cada artefacto esta provisto por un conjunto de *operaciones* y un conjunto de *propiedades observables*. Las operaciones representan procesos computacionales ejecutados internamente en el artefacto, los cuales pueden ser disparados por los agentes o por otros artefactos. Las *propiedades observables* representan estados variables cuyo valor puede ser percibido por los agentes. El valor de una *propiedad observable* puede ser cambiado dinámicamente como resultado de la ejecución de una *operación*. La ejecución de una *operación* también puede generar *señales*, que los agentes pueden percibir también, sin embargo, a diferencia de las *propiedades*

observables, las *señales* son útiles para representar eventos observables no persistentes ocurridos dentro del artefacto, acarreando algún tipo de información. De una manera abstracta, un artefacto puede verse como se muestra en la figura 2.4

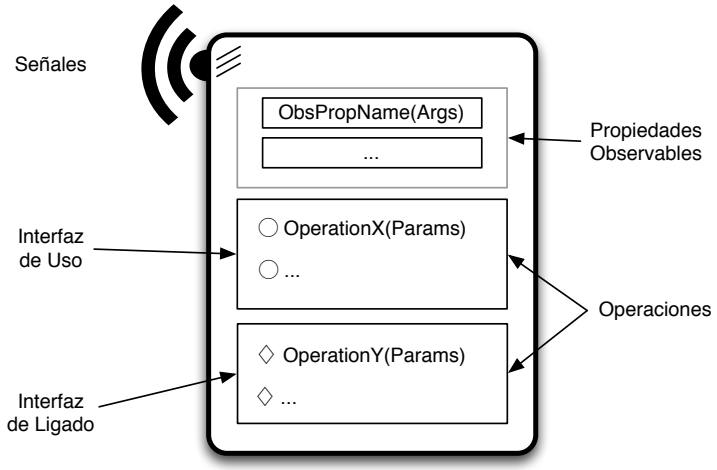

Figura 2.4: Representación abstracta de un artefacto. Adaptado por Ricci et al [31]

Desde el punto de vista de un agente, las *operaciones* de artefactos representan acciones externas provistas por el agente para el ambiente. De tal forma que, en el ambiente basado en artefactos, el repertorio de acciones de disponibles para el agentes está definido por el conjunto de artefactos que se encuentren en el ambiente. Esto implica que el repertorio de acciones puede ser dinámico. A su vez, las *propiedades observables* de los artefactos y *eventos* representan las percepciones del agente. En la figura 2.5 se observa una visión general de los principales conceptos que caracterizan el meta modelo A&A.

Otra propiedad de los artefactos es que éstos pueden *ligarse* entre si, de forma que un artefacto pueda disparar la ejecución de operaciones de otro artefacto. Con este propósito, los artefactos exhiben una *interfaz de ligado*, que análogamente a la interfaz de uso para los agentes, incluye el conjunto de operaciones que puede ser ejecutada por otros artefactos, una vez que dichos artefactos han sido ligados por un agente.

Por otra parte, un artefacto puede tener un *manual* en un formato legible por la computadora para ser consultado por los agentes. El manual contendría una des-

cripción de las funcionalidad provista por el artefacto y la manera de explotar dicha funcionalidad. Esta característica ha sido concebida pensando en los Sistemas Abiertos compuestos por agentes inteligentes que dinámicamente deciden que artefactos usar de acuerdo a sus metas.

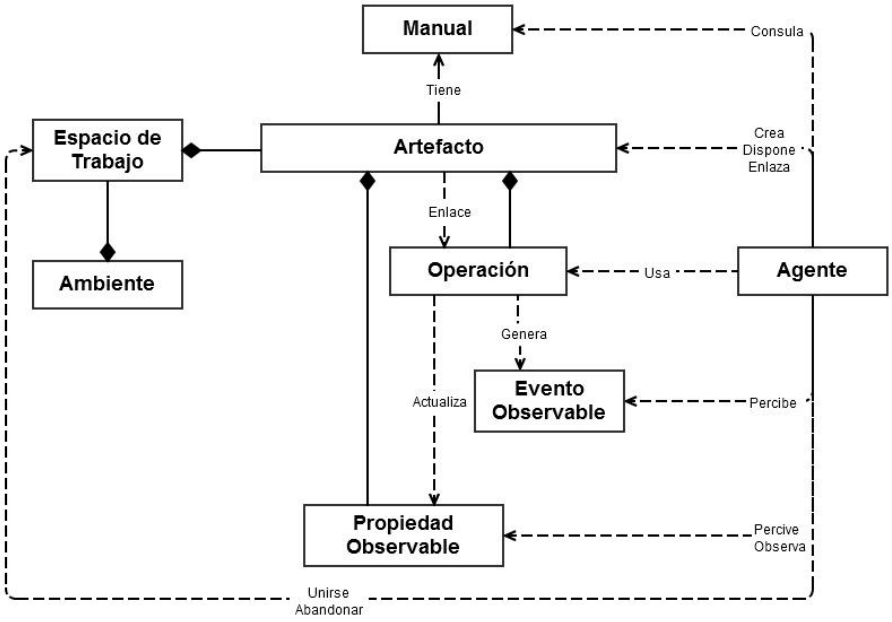

Figura 2.5: El meta-modelo A&A (Agentes y Arfefactos) en notación UML. Adaptado por Ricci et al [31]

Para adaptarse a la complejidad de los ambientes, en un ambiente basado en artefacto un agente puede seleccionar dinámicamente artefactos específicos dentro de un conjunto de artefactos, de tal forma que percibe las propiedades observables y eventos solo de las partes del ambiente que ha seleccionado. En este sentido, el modelo A&A provee un manera explícita para modularizar el ambiente, además de incluir soporte para ser extendidos y adaptados dinámicamente. Estas características, junto con la propiedad de ser adaptados para cualquier nivel de ambientes, proporcionan una nueva dimensión para resolver problemas complejos tanto en entornos computacionales como en entornos físicos.

CArtAgO (Common ARtifact infrastructure for AGent Open environments) es un framework e infraestructura para la programación y ejecución de ambientes ba-

sados en artefactos implementando el modelo A&A. Como framework, está provisto de una *API* basado en Java para la programación de artefactos y ambientes para la ejecución de ambientes basados en artefactos, junto con una biblioteca Java compuesta por un conjunto de tipos de artefactos de propósito general predefinidos. Como una infraestructura, éste proporciona el API y los mecanismos subyacentes para extender lenguajes (o frameworks) de programación de agentes para trabajar dentro de ambientes basados en *artefactos*.

2.5.1 Programación de artefactos

El API basado en Java de *CArtAgO* permite la programación de artefactos en términos de clases Java y tipos de datos básicos, sin la necesidad de usar algún nuevo lenguaje de propósito general. Mediante el uso de un ejemplo se describirá de manera general las características básicas del modelo de programación de artefactos.

```
1  package c4j;
2
3  import cartago.*;
4
5  public class Contador extends Artifact {
6
7    void init(){
8      defineObsProperty("valor",0);
9    }
10
11   @OPERATION void inc(){
12     ObsProperty prop = getObsProperty("valor");
13     prop.updateValue(prop.intValue()+1);
14     signal("tick");
15   }
16 }
```

Cuadro 2.3: Definición de un artefacto `Contador`

Un artefacto (tipo) es programado directamente mediante la definición de una clase Java extendiendo de la clase `cartago.Artifact`, y usando un conjunto básico de anotaciones Java y métodos heredados para definir los elementos de la estructura y comportamiento del artefacto. El tipo define la estructura y comportamiento de la instancia concreta que será instanciada y utilizada por los agentes

en tiempo de ejecución. En el cuadro 2.3 se muestra un ejemplo simple de la definición de un tipo de artefacto llamado `Contador`, implementando un simple contador. Éste proporciona una sola propiedad observable llamda `valor` en la que se mantiene el seguimiento del valor del conteo, y una operación `inc` para el incremento del valor del conteo. El método `init` es usado para especificar cómo debe inicializarse el artefacto al momento de ser instanciado en el espacio de trabajo.

La primiriva `defineObsProperty` permite definir propiedades observables, en ésta se especifica el nombre de la propiedad y el valor inicial (el cual puede estar compuesto por cualquier tupla de objetos de datos). Las operaciones son definidas por medio de métodos con la anotación `@OPERATION` y el valor de retorno `void`, usando parámetros de métodos como parámetros de la operación tanto de entrada como de salida. El método `updateValue` puede usarse para modificar el valor de una propiedad observable.

Al igual que las propiedades observables, las señales tienen forma de tupla, con un functor seguido de uno o multipes argumentos opcionales. En el caso de `tick`, la señal no tiene argumentos.

2.5.2 Acciones para trabajar con artefactos

Para trabajar dentro de un espacio de trabajo es necesario que el agente se una a él. Posteriormente tiene la posibilidad de salirse de dicho espacio de trabajo o unirse a otro, y trabajar simultáneamente en varios espacios de trabajo. Por defecto, cuando es inicializado el programa del SMA usando *Jason* y *CArtAgo*, todos los agentes se unen al espacio de trabajo `default`. Una vez situado en un espacio de trabajo, las acciones disponibles para trabajar con artefactos pueden categorizarse en tres grupos:

- Acciones para crear, buscar y disponer de los artefactos en el espacio de trabajo.
- Acciones para usar artefactos, ejecutar operaciones y observar propiedades y señales.
- Acciones para ligar y desligar artefactos.

Los artefactos pueden ser creados en tiempo de ejecución por los agentes del SMA, y una vez creados pueden ser descubiertos por los agentes y estos pueden destruir artefactos descubiertos. Éste es el mecanismo básico del modelo para dar soporte a la *extensión dinámica* de los ambientes. Tres clases de acciones se proveen con este fin:

- `makeArtifact(Nombre,Tipo,Params):ArtId` crea un nuevo artefacto en el espacio de trabajo. La retroalimentación de esta acción es el nombre lógico o identificador del artefacto creado.

- `disposeArtifact(ArtId)` remueve del espacio de trabajo al artefacto con identificador `ArtId`.

- `lookup(Nombre):ArtId` busca por `Nombre` un artefacto en el espacio de trabajo regresando como retroalimentación su `ArtId`. Por otra parte, también es posible buscar el artefacto por `Tipo`, mediante el uso de `lookupArtifactByType(Tipo):ArtId`. Esta acción regresa un conjunto (posiblemente vacío) de artefactos como retroalimentación.

Usar un artefacto involucra dos aspectos. Primero, el agente debe ser capaz de ejecutar las operaciones listadas en la interfaz de uso del artefacto; Segundo, el agente debe ser capaz de percibir la información observable en términos de propiedades observables y eventos. Para ello:

- `use(ArtId,NombreOp(Params)):ResOp` atiende al primer aspecto mencionado. Esta acción provee la identidad `ArtId` del artefacto que será usado; los detalles de la operación que será ejecutada, incluyendo su nombre `NombreOp` y sus parámetros `Params`. La acción tiene éxito si la operación correspondiente es terminada con éxito; y falla si la operación especificada no está incluída en la interfaz de uso del artefacto; o si un error ocurrió durante la ejecución de la operación, por ejemplo, que la operación misma haya fallado. La ejecución completa de la operación puede generar algún resultado que es devuelto al agente como la retroalimentación `ResOp`. Al ejecutar `use` la actividad o plan actual del agente se suspende hasta que un evento

reportando que la operación se ha completado, con éxito o fracaso, es recibida. Al recibir este evento, la ejecución de la acción es completada y el plan o actividad relativo a ésta se reactiva. Sin embargo, aún cuando esta actividad ha sido suspendida, el agente no se bloquea, sigue su ciclo de razonamiento normal.

El segundo aspecto del uso de artefacto, su *observación*, es atendido por:

- `focus(ArtId,Filtro` especifica la identificación `ArtId` del artefacto a observar y opcionalmente un filtro, para seleccionar el subconjunto de eventos en los que el agente está interesado. En los lenguajes de programación orientados a agentes BDI, las propiedades observables corresponden directamente con las *creencias* de los agentes. Un agente puede focalizar en diversos artefactos al mismo tiempo.

- `stopFocus(ArtId)` es el dual de `focus`. Provee el mecanismo para dejar de observar al artefacto con identificador `ArtId`.

Al observar un artefacto, las propiedades observables son mapeadas dentro de la base de creencias de los agentes observadores. De tal forma que cada vez que una propiedad observable de un artefacto cambia o una señal es generada por dicho artefacto, las correspondientes creencias de los agente son actualizadas.

Es importante destacar que un agente puede usar un artefacto sin necesidad de estarlo observando. De igual forma, si un agente ejecuta una operación de un artefacto que está siendo observado por otro agente, el agente observador tendrá eventualmente percepciones acerca de la ejecución de la operación (mientras la ejecución de `use` no se haya terminado). Una última instrucción en esta categoría es la siguiente:

- `observeProperty(Prop):ValorProp` lee el valor actual de la propiedad `Prop`. En este caso no hay eventos generados, el valor `ValorProp` regresa al agente como una señal de retroalimentación de la acción. Esto resulta útil cuando no es necesario que el agente este atento continuamente del estado observable de un artefacto; pero necesita saber el valor de la propiedad en un momento dado.

Ligar dos artefactos sirve para conectarlos de forma que el artefacto que liga puede ejecutar operaciones sobre el artefacto ligado. De manera más precisa, cuando se ligan dos artefactos, la ejecución de una operación en el artefacto que liga puede disparar la ejecución de operaciones en el instrumento ligado. Dos acciones son provistas con este fin:

- `linkArtifacts(ArtId1,ArtID2,Puerto)` El artefacto `ArtID1` liga `ArtID2`. El parámetro `Puerto` es necesario cuando se liga el mismo artefacto a multiples artefactos. En ese caso, el artefacto `ArtID1` debe proveer varios puertos etiquetados y adjuntar cada artefacto ligado a un puerto en particular.

- `unlinkArtifacts(ArtId1,ArtID2)` es el dual de la operación anterior.

Para ejemplificar los aspectos básicos de la creación y uso de artefactos, incluyendo la observación, se recurre al empleo del artefacto contador del cuadro 2.3.

Dos agentes son creados de forma que observan y comparten un artefacto que implementa un `Contador`. El agente `usuario` realiza operaciones en el artefacto, mientras que el agente `observador` percibe el estado observable y las señales del mismo. La definición del SMA para este ejemplo se muestra en el cuadro 2.4

```
1 MAS cartagoUsoArtefactos {
2   environment: c4jason.CartagoEnvironment
3   agents:
4     usuario agentArchClass c4jason.CAgentArch;
5     observador agentArchClass c4jason.CAgentArch;
6 }
```

Cuadro 2.4: Definición del SMA para el ejemplo usando el artefacto `Contador`

El agente `usuario` (cuadro 2.5) cuya meta `inicio` comprende usar y crear un artefacto identificado como `contador00` de tipo `c4j.Contador` para usarlo dos veces ejecutando la operación `inc`.

Los artefactos se crean con la acción externa `makeArtifact` que recibe como argumentos una cadena que representa el nombre el artefacto, `contador00`; otra

```
1 /* Initial goals */
2 !inicio.
3
4 /* Plans */
5 +!inicio
6   <- makeArtifact("contador00","c4j.Contador",[],Id);
7      inc;
8      inc[artifact_id(Id)];
9      inc[artifact_name("contador00")].
```

Cuadro 2.5: Agente usuario del ejemplo usando el artefacto `Contador`

que representa la clase que implementa el artefacto, `c4j.Contador`; una lista de parámetros, en este caso vacía; y una variable que unifica con el identificador lógico del artefacto, `Id`.

Hay tres formas de ejecutar una operación. En la primera, la operación `inc` se invoca sin especificar que artefacto se desea usar (línea 7, cuadro 2.5). Si ningún artefacto provee esa operación, ésta falla y en consecuencia, la acción externa del agente falla también. Si se se encuentra más de un artefacto que provee la operación en cuestión, se consideran primero los artefactos creados por el agente y se selecciona uno no deterministicamente. En la segunda forma, se especifica qué artefacto se desea usar mediante su identificador lógico `Id` (línea 8, cuadro 2.5). En la tercera, se usa el nombre del artefacto `contador00` en lugar de su identificador lógico (línea 9, cuadro 2.5).

Por otra parte, el agente `observador` (cuadro 2.6) focaliza en el contador creado por `usuario` e imprime mensajes en la pantalla cada vez que la propiedad observable `valor` es modificada o una señal `tick` es percibida.

Los agentes pueden descubrir el identificador lógico de un artefacto mediante la acción `lookupArtifact` provista por el espacio de trabajo donde se encuentra el artefacto en cuestión; especificando ya sea el nombre del artefacto o su tipo. En el último caso, si existen varios artefactos del mismo tipo, uno de ellos es elegido no deterministicamente. En el ejemplo, si el agente ejecuta la acción `lookupArtifact` antes de que el artefacto haya sido creado, entonces la acción falla y la intención falla generando el evento Jason `-?miArtefacto(...)` invocando al plan de reparación correspondiente.

```
/* Initial goals */
!observa.

/* Plans */
+!observa : true
  <- ?miArtefacto(Id);
     focus(Id).

+valor(V)
  <- println("Nuevo valor observado: ",V).

+tick
  <- println("Percibiendo un tick.").

+?miArtefacto(Id)
  <- lookupArtifact("contador00",Id).

-?miArtefacto(Id)
  <- .wait(10);
      ?miArtefacto(Id).
```

Cuadro 2.6: Agente observador del ejemplo usando el artefacto `Contador`

Al focalizar en un artefacto, las propiedades observables del artefacto se mapean en la base de creencias del agente. De esta forma, los cambios en las propiedades observables del artefacto son detectados como cambios en la base de creencias del agente. En este ejemplo, cada vez que la propiedad observable `valor` es modificada, el agente `observador` detecta un evento disparador `+valor(V)`. Las creencias relacionadas con propiedades observables de los artefactos están decoradas con anotaciones que pueden ser usadas al momento de seleccionar planes relevantes y aplicables.

Cuando un agente focaliza en un artefacto, las señales emitidas por este último se vuelven perceptibles para el agente. Al igual que con las propiedades observables, esto se ve reflejado en la base de creencias de los agentes. En el ejemplo un evento `+tick` se genera en el agente `observador` cada vez que el agente `usuario` ejecuta la operación `inc`; aunque en este caso la base de creencias no cambia.

2.6 Resumen

Como puede verse en este capítulo, la arquitectura de agentes BDI cuenta con características que son idóneas para la programación de robots, como es su razonamiento práctico, el cual puede brindarles la facultad de seleccionar las acciones apropiadas para lograr metas complejas mediante el ensamblaje de planes parciales al vuelo durante la ejecución del programa, sin perder la capacidad reactiva necesaria para poder desenvolverse en ambientes dinámicos.

Por otro lado, el intérprete Jason, además de contar con bases solidas para la implementación de agentes BDI, al utilizar programación declarativa, es posible reducir el énfasis sobre los aspectos del control de flujo en los programas, concentrándose principalmente en la definición de metas y los posibles mecanismos para lograrlas, facilitando el diseño de procesos deliberativos de los robots.

Sin embargo, dado que Jason no está diseñado para la programación de robots, es necesario contar con algún framework que permita extender el alcance de los agentes para interactuar con ambientes físicos. Es por ello que se recurre al modelo de ambiente *A&A* de *CArtAgO*, el cual brinda la posibilidad de diseñar artefactos para el control de los componentes de un robot, y a la vez permitir a los agentes utilizar únicamente aquellos que sean necesarios para cumplir sus objetivos.

Existe literatura reciente en la que se comprueba la viabilidad de utilizar la arquitectura de agentes BDI, con diferentes LPAs, para la implementación de controles robóticos. Esta literatura será descrita y analizada en el siguiente capítulo.

CAPÍTULO

3

Robótica cognitiva

3.1 Antecendentes de robótica cognitiva

De manera general, la robótica cognitiva se encarga del estudio de la representación de conocimiento y problemas de razonamiento que enfrenta un robot autónomo (o agente) dentro de un ambiente dinámico y conocido parcialmente [6]. Ésta surge a partir de la necesidad de relacionar el conocimiento, las percepciones y acciones de un robot dentro de un control de alto nivel, de tal forma que sea capaz de generar acciones apropiadas para una función presente de algún conjunto de creencias relacionadas con sus metas y el ambiente.

Previo a la formalización de la robótica cognitiva como campo de investigación de IA ya se había intentado obtener controles robóticos de alto nivel, sin embargo, éstos solían ser dependientes de los primeros enfoques de *planificación automática*, en donde dado una meta a lograr junto con la descripción de algún estado inicial del ambiente, y los prerequisito y efectos de un conjunto de primitivas de acciones, se encuentra una secuencia de acciones que satisfagan dicha meta, para que posteriormente sean ejecutadas por el robot. Sin embargo, según Levesque y Reiter [3], este tipo de controles robóticos conllevaban los siguientes perjuicios:

Ausencia de retroalimentación Se espera que el sistema de planificación genere una secuencia de acciones sin considerar los posible resultados de una retroalimentación durante su ejecución.

Ausencia de reactividad Podrían surgir situaciones excepcionales durante la ejecución como interrupciones de alta prioridad, fallas de ejecución de algún módulo o cualquier otra situación inesperada.

Dificultades computacionales Para todo sistema la planificación automática podría aparentar inviable debido al alto grado de incertidumbre que existe en un mundo dinámico, excepto en dominios muy simples; La planificación parece poco adecuado para generar secuencias muy largas de acciones.

Incompatibilidad con la robótica convencional La robótica convencional trata con microacciones donde las decisiones son realizadas varias veces por segundo en mundos caracterizados por ruido e incertidumbre.

La propuesta de Levesque y Reiter alternativa al problema que presentaban los controles robóticos dependientes de la planificación automática clásica consiste en reducir dicha dependencia por medio del uso de *intérpretes de programas* que, en vez de tomar una meta como parte de entrada del programa y utilizar un planificador para generar primitivas de acciones para lograrla, toma como entrada un *programa de alto nivel* que necesita ser ejecutado y genera primitivas de acción para ejecutarlo.

Por *programa de alto nivel* se refieren a un programa que le indica al robot (o agente) qué se necesita realizar por medio de las siguientes propiedades:

- Las declaraciones más primitivas del programa son acciones primitivas externas disponibles para el robot. Por ejemplo: `ir_al_escritorio` y después `tomar_ un_objeto`.

- Las pruebas dentro del programa pertenecen a condiciones en el mundo que son afectadas por el robot (y otros robots). Por ejemplos: si `la_puerta_está_cerrada` entonces...

- Los programas deben ser *no determinísitcos*, es decir que deben contener puntos a elegir donde el intérprete debe tomar una selección razonada (no aleatoria) que satisfaga correctamente algunas limitaciones posteriores. Por ejemplo: `entra_en_la_puerta_apropiada` y obtén el `objeto`.

Los *programas de alto nivel*, a los que hacen referencia Levesque y Reiter, se asemejan a los planes pero son considerablemente más generales. Debido a que estos no son determinísticos, es decir que éstos no pueden ser ejecutados ciegamente. La tarea del programa intérprete es averiguar la manera de cómo ejecutarlos. Para hacerlo, el intérprete necesita ser capaz de determinar cuáles son las pruebas que son verdaderas o falsas en relación a las condiciones del mundo, y qué primitivas de acción son posibles o imposibles después de haber ejecutado otras primitivas de acción.

Este enfoque dio lugar al desarrollo del lenguaje de programación lógico llamado GOLOG [33] el cual ha sido utilizado para la implementación de controles robóticos como [4, 5].

3.2 Robots cognitivos y Agentes BDI

Uno de los primeros trabajos en los que se planteó utilizar el intérprete *Jason* para la programación de controles robotícos es el de Jensen [14]. Éste no se propone como control para robots cognitivos, sino que se ve más bien desde el enfoque del diseño de un interfaz entre un agente y un robot. Específicamente consiste en una interfaz entre un agente en *Jason* y la plataforma robótica Lego Mindstorms NXT programada con leJOS [19]. Ésto se lleva a cabo mediante la implementación de una arquitectura extendida de la clase de la arquitectura base de agentes *Jason* (`AgArch`), en la que se sobreescriben los métodos heredados de acciones y percepciones para añadir secuencias para el envío y recepción de mensajes (instrucciones de motores y señales obtenidas por sensores respectivamente) entre el agente y el robot. Además añade dentro de la inicialización del agente una secuencia para el enlace con el robot y configuración de los dispositovos (motores y sensores) a utilizar. Para realizar ésto recurre a las bibliotecas para PC de leJOS[1].

[1] `http://www.lejos.org/nxt/pc/api/index.html`

En investigaciones más recientes se ha recurrido a la POA, principalmente usando el modelo BDI, para la implementación de controles robóticas cognitivos, un ejemplo de estos es el trabajo de Wei et al [16]. En el que se propone una arquitectura para robots cognitivos usando la clásica *arquitectura robótica de tres capas* [18] como arquitectura base. En ésta, la principal variación que hace sobre la arquitectura de tres capas es la sustitución de la capa deliberativa por una capa cognitiva compuesta por un agente programado en el LPA GOAL [12], el cual está basado en Prolog, adicionalmente incluye una capa de *control de comportamientos* escrita en C++, y una capa de control de hardware usando el lenguaje de programación robótico URBI[1]. Además, utiliza una capa de interfaz entre la capa cognitiva y la capa de comportamientos para la traducción de información sub-simbólica a representaciones simbólicas y viceversa. Esta interfaz esta implementada utilizando un software llamado Estándar de Interfaz de Ambiente (EIS por sus siglas en inglés) [20] que facilita la conexión entre agentes programados en diversas plataformas de agentes con ambientes. EIS es una interfaz que permite enlazar diversos LPA con otros frameworks, por medio de funciones que mapean acciones de agentes con la ejecución de funciones del ambiente, así como parámetros de ambientes con percepciones del agente.

La principal funcionalidad de ésta se concentra en la capa de comportamientos que se compone de módulos ordenados encargados del procesamiento de datos obtenidos por sensores, comunicación entre la capa deliberativa con otros robots, y ejecución de comportamientos y acciones. Ofreciendo funciones como reconocimiento de objetos (a través de visión), navegación, planificación de rutas, entre otras. Cabe mencionar que sus módulos de procesamiento de datos de sensores están divididos de acuerdo a su complejidad de procesamiento (por ejemplo: el control de sonares, micrófono, y sensores de monitoreo de motores están incluidos dentro de un solo módulo, mientras que el de la cámara se encuentra en uno separado).

En la arquitectura propuesta por Wei se distinguen los controles deliberativos y reactivos por la capa cognitiva y la capa de control de comportamientos respectivamente, los cuales están acoplados débilmente y conectados a través la capa de interfaz implementada con el EIS. Alternativamente, también existe la posibilidad

[1] http://www.urbiforge.org/

de arquitecturas con dichos controles acoplados fuertemente, lo que brinda beneficios como una integración más robusta reduciendo la pérdida de información dentro de su comunicación, permitiendo un mayor desempeño que un acoplamiento débil. Una forma de implementar este tipo de acoplamientos es mediante el uso de un componente de memoria que es compartido y directamente accesible por las capas de control, evitando la necesidad de duplicar información en las diferentes capas. La desventaja de un enfoque con acoplamiento fuerte se ve reflejada en el incremento de complejidad de la arquitectura y una interdependencia entre los componentes, acarreando como una de las consecuencias la dificultad para extender dicha arquitectura para futuras aplicaciones.

Por otro lado, el enfoque de las capas débilmente acopladas implementado en la arquitectura de Wei es motivado por el factor de incrementar la flexibilidad, ésto lo logran mediante la separación de objetivos de cada capa. Wei argumenta que un programador de agentes, por ejemplo, no necesita gastar tiempo en el manejo del reconocimiento de objetos. Así mismo, un programador de comportamientos no requiere considerar la toma de decisiones para programar el módulo de procesamiento de imágenes. Para lograr el acomplamiento entre estas dos capas recurren a la capa de interfaz implementada con el EIS, para la que definen un esquema de codificación que indica cómo interpretar y mapear cada dato obtenido por los sensores de una forma subsimbólica a una forma simbólica que sea útil para la toma de decisiones de la capa cognitiva.

Otro trabajo íntimamente relacionado con el de Wei es la tesis Mordenti [15], en la cual se propone un control de robot sobre una arquitectura de tres capas. En ésta, la capa cognitiva se compone por un agente en *Jason*; su capa intermedia, o capa ejecutiva, es implementada usando *CArtAgO*; y finalmente su capa más baja, o capa funcional, es representada por una plataforma robótica generada por el simulador Webots [1]. En éste, además de explorar por primera vez la utilización de *CArtAgO* como interfaz entre un agente y un plataforma robótica (al menos en simulación), también presenta un conjunto de experimentos donde compara su enfoque de programación de robots contra un paradigma imperativo, mediante la implementación de tareas simples de sistemas robóticos programados con ambos paradigmas.

[1] http://www.cyberbotics.com/

La capa ejecutiva de la implementación de Mordenti maneja la interacción entre las otras dos capas, en la que provee los mecanismos para generar representación simbólica del ambiente para la capa cognitiva y transforma las acciones de alto nivel de abstracción en secuencias de instrucciones de bajo nivel para el control de los componentes del robot. Todas estas actividades son realizadas mediante un único *artefacto CArtAgO*. Dentro del dicho artefacto se encuentran implementadas operaciones ad-hoc de acuerdo con las tareas que realiza el robot. Una deficiencia que tiene esta implementación es que la separación de actividades realizadas por el artefacto, descritas anteriormente, no es fácilmente distinguible. En otras palabras, ésta no explota la modularidad que ofrece *CArtAgO* para el diseño de ambientes.

Un trabajo reciente en el que se describen algunos requerimientos para la programación de robots cognitivos mediante algún LPA es el de Ziafati et al [17]. En éste se resaltan cuatro requerimientos principalmente. Primero, el LPA debe estar provisto por soporte incluido para su integración con algún framework robótico existente como *ROS*[1] (Robot Operating System). Segundo, debe mantenerse un balance entre las capacidades reactivas y deliberativas a través de la adaptación de una arquitectura de control robótico como la arquitectura de tres capas. Tercero, la arquitectura debe incluir algún componente sensorial para el manejo del procesamiento de los datos de entrada con la finalidad de adquirir información de alto nivel de abstracción, facilitando el razonamiento sobre eventos complejos. Cuarto, debe proveerse de un control de ejecución de planes con la capacidad de representar planes complejos y coordinar la ejecución de éstos en paralelo.

En relación al primer requerimiento mencionado en el trabajo de Ziafati, es importante resaltar que debido al rápido crecimiento en cuanto a escala y alcance de los sistemas robóticos actuales, es necesario que las controles para dichos sistemas ofrezcan la facilidad de ser reutilizados y actualizados con la misma facilidad. La ventaja que ofrece el acoplamiento con frameworks robóticos es que de esta manera se cuenta con una interfaz estándar para el acceso y manipulación de una gran variedad de dispositivos físicos para la integración en robots. Por ejemplo, *ROS* es un framework robótico libre que cuenta con un repositorio compuesto por una gran cantidad de paquetes para el manejo de diversas plataformas robóticas y hardware

[1] http://www.ros.org/

variado (como sensores y actuadores), así como para la ejecución de diferentes tareas robóticas como procesamiento de imágenes, navegación, entre otras.

El segundo punto, relacionado con el balance entre las capacidades reactivas y deliberativas que debería tener un un robot, se destaca que este punto ha sido uno de los mayores retos en las investigaciones de la robótica clásica. Por un lado, es deseable una capacidad deliberativa compleja para la autonomía de todo robot, de tal forma que se desenvuelvan en un mundo dinámico tomando las mejores decisiones para cumplir sus tareas. Por otro lado, también se requiere que los sistemas robóticos reaccionen ante eventos en un tiempo adecuado para asegurar sus seguridad, la seguridad del ambiente en que se encuentren y para cumplir con las exigencias de sus diversas tareas. Éste sigue siendo un problema abierto, sin embargo, en el trabajo de Ziafati se propone como una posible alternativa para abordarlo utilizando arquitecturas en capas. Con éstas es posible encapsular los procesos deliberativos en una capa, mientras que los procesos reactivos se encuentran separados en otra.

El tercer punto, relacionado con la incorporación de un módulo sensorial para el manejo información de entrada para facilitar el razonamiento sobre eventos complejos, tiene que ver principalmente con las deficiencias de los agentes BDI de LPA existentes para el procesamiento de datos continuos ingresados en tiempo real (información de bajo nivel de abstracción como los datos crudos de sensores). Estos agentes generalmente asumen que la información percibida del ambiente se encuentra en una representación simbólica, debido a que sus ciclos de razonamiento podrían provocar un retraso en la respuesta de los eventos generados en el ambiente. Por esta razón es recomendable recurrir a extensiones o ambientes de agentes que permitan implementar algún tipo de componente sensorial con la capacidad de transformar los datos crudos obtenidos de los sensores en representaciones simbólicas relacionadas al dominio de conocimiento del agente; asi como proporcionar diferentes métodos de acceso a los datos de sensores, tanto por consulta como por recepción de eventos; entre otras características.

El cuarto y último requisito mencionado en el trabajo de Ziafati, relacionado con la representación de planes complejos y su coordinación en paralelo, se enfoca principalmente a la necesidad de contar con operadores de planes capaces de sincronizar la ejecución de planes y acciones en arreglos de planes complejos, más

allá de la simples configuraciones secuencial o en paralelo. Entre algunos mecanismos que mencionan que podrían enriquecer a los LPA para la programación de robots cognitivos son lo siguientes:

- Contar con una *descomposición jerárquica de tareas*, la cual puede ser implementada mediante planes complejos formados por un conjunto de subplanes ordenados de manera secuencial o en paralelo a diferentes niveles jerárquicos y tener control sobre cada uno de ellos.

- Proveer soporte para posibles contingencias durante la ejecución de planes, como la capacidad para la interrupción y reestablecimiento de planes.

- Proveer soporte para acciones atómicas y continuas.

3.3 Resumen

La literatura analizada en éste capitulo ha permitido observar algunas características favorables para el diseño e implementación de arquitecturas de robots cognitivos utilizando LPAs. A pesar de que el enfoque y los alcances de cada trabajo relacionado son diferentes, tienen características comunes que resultan de utilidad como punto de partida para el desarrollo de un nuevo enfoque para la implementación de robots cognitivos.

Entre las características más relevantes se encuentra el uso de arquitecturas en capas, en las cuales se sustituye la capa deliberativa, compuesta generalmente por planificadores automáticos y aquellos componentes que requieren una mayor cantidad de recursos computacionales, por una capa cognitiva compuesta por un agente BDI. Así mismo, para facilitar los procesos deliberativos llevados a cabo por el razonamiento lógico del agente, es común observar que el conocimiento manejado en la capa cognitiva se encuentre representado simbólicamente, es decir, con un mayor nivel de abstracción que los datos manejados en cualquier otra capa de la arquitectura robótica.

Por otro lado, para que el agente de la capa cognitiva pueda tener un control sobre la plataforma robótica es necesario el uso de una interfaz que, además de

convertir datos crudos a datos simbólicos y viceversa, facilite la programación sobre los diferentes niveles de abstracción para un rápido diseño e implementación del sistema robótico. Los frameworks utilizados para el desarrollo de ambientes de agentes como *CArtAgO* y EIS están provistos de mecanismos que permiten extender LPAs para trabajar con otros framworks (como framework robóticos), lo cual los hace idóneos implementarlos como interfaz entre el agente de la capa cognitiva con alguna otra capa de la arquitectura robótica.

El trabajo de Jensen [14] no utiliza alguno de los frameworks anteriormente mencionados para implementar la interfaz entre el agente y la plataforma robótica, lo que dificulta su modificación o reutilización. Los trabajos de Wei [16] y Mordenti [15], a pesar de recurrir a los framworks EIS o *CArtAgO* respectivamente para la implementación de una interfaz entre la capa cognitiva y alguna de las otras capas, éstos son utilizadas principalmente para la transformación del nivel de abstracción de los datos manejados entre las capas involucradas.

En esta tesis, se desea aprovechar la modularidad que ofrece el modelo *A&A* de *CArtAgO* para integrar una capa de control de componentes a una arquitectura en capas para robots cognitivos. Utilizando los artefactos como módulos que representen cada uno de los componentes físicos de la plataforma robótica.

CAPÍTULO

4

Plataforma robótica Lego Mindstorms NXT

Lego Mindstorms es una serie de productos con fines de aprendizaje e investigación para el desarrollo de robots y automatización de aplicaciones en general. Consiste en un conjunto de sensores, motores, un ladrillo programable y un conjunto de piezas Lego para el ensamblaje. El ladrillo puede ser programado por el lenguaje y entorno nativo NXT-G, el cual es un entorno gráfico de programación drag-and-drop, sin embargo, es poco flexible y es difícil diseñar programas complejos con él. Por ésta razón se han desarrollado otros lenguajes entre los que destaca leJOS [19].

Lego Mindstorms fue desarrollado desde 1986 por el Grupo de Epistemología y Aprendizaje de MIT, dirigido por Seymour Papert y Mitchel Resnick y financiado por la empresa Lego. En 1998 fue lanzada al mercado la primera versión, conocida como RCX. Posteriormente, la versión NXT se lanzó al mercado en el año 2006, con la que se incluyó documentación de los esquemas de sus sensores, protocolos de comunicación y programación de sus microcontroladores a bajo nivel [34].

4.1 Características y componentes

Lego Mindstorms NXT consiste de un conjunto de sensores, motores, un ladrillo programable y un conjunto de piezas Lego para ensamblaje, dichos componentes se describen a continuación.

4.1.1 Ladrillo NXT

El componente principal del Lego Mindstorms NXT es el ladrillo programable NXT, el cual contiene dos microprocesadores que almacenan y computan los programas. El procesador principal contiene 256 Kb de memoria Flash, 64 Kb de memoria RAM y utiliza un reloj con 48Mhz de frecuencia. Además, el ladrillo NXT se compone de cuatro puertos de entrada para los sensores y tres puertos de salida para los motores. Los conectores son de tipo RJ-12. Al frente del ladrillo se encuentra un display LCD monocromático de 100X64 pixeles junto con cuatro botones que sirven de interfaz para el usuario. Así mismo, cuanta con un altavoz interno.

Figura 4.1: Ladrillo NXT del kit Lego Mindstorms NXT

Los puertos de entrada pueden recibir señales analógicas, desde 0 a 5 volts, y señales digitales utilizando el protocolo I^2C [1]. La conexión a la computadora puede ser establecida mediante cable USB, o de manera inalámbrica utilizando comunicación Bluetooth.

El ladrillo NXT puede conectarse a través de Bluetooth con otros tres dispositivos al mismo tiempo, pero sólo puede comunicarse con uno a la vez. De este modo,

[1] http://www.nxp.com/documents/user_manual/UM10204.pdf

es posible transmitir datos entre diferentes ladrillos NXT durante la ejecución de programas.

La comunicación Bluetooth dentro del ladrillo NXT es establecida mediante canales maestro-esclavo. Esto significa que un dispositivo debe funcionar como maestro mientras que los otros tres funcionan como esclavos, estos últimos únicamente pueden comunicarse con el dispositivo maestro. Cabe mencionar que un ladrillo NXT no puede funcionar como un dispositivo esclavo y maestro simultáneamente.

4.1.2 Motores

Los motores de Lego Mindstorms NXT, se consideran servomotores, debido a que éstos cuentan con un motor de corriente directa, un sistema reductor formado por engranes y sensores que le permiten obtener un retroalimentación de desplazamiento. La señal obtenida por sus sensores es enviada al ladrillo NXT para ser procesada y posteriormente se ajusta la señal emitida al motor.

Figura 4.2: Motor del kit Lego Mindstorms NXT

La velocidad de estos motores es regulable, y tienten la capacidad de rotar a una posición determinada con un error de 2 grados, o simplemente rotar en cualquier dirección con una velocidad constante de hasta 900 grados por segundo.

Otra capacidad de estos motores, es la capacidad de oponer una resistencia al movimiento mientras éstos se mantienen encendidos para una posición establecida, y permitir su libre rotación cuando se encuentran apagados, mientras que sus sensores internos se mantienen en funcionamiento registrando el movimiento.

4.1.3 Sensores

Sensor de contacto: También llamado interruptor de límite, se considera un sensor digital debido a que éste entrega un valor booleano, sin embargo su respuesta y funcionamiento son completamente analógicos. Es un dispositivo

mecánico constituido por un botón sensible a la presión. Generalmente se ocupa para la detección de obstáculos o posiciones extremas de movimientos, donde al cambiar de estado, se apaga el actuador correspondiente, impidiendo posibles daños.

Figura 4.3: Sensor de contacto del kit Lego Mindstorms NXT

Sensor de luz: Está constituido por un fototransistor, y un LED de color rojo. El fototransistor utilizado detecta un rango de luz con longitudes de onda desde 400nm hasta 1200nm, con una mayor sensibilidad para la luz infraroja (longitud de onda mayor a 750nm). El sensor funciona de dos maneras, activa y pasiva. Cuando se utiliza de manera pasiva, el LED se mantiene apagado, y se detecta de forma lineal la luz visible al ojo humano. Cuando se usa de manera activa, el LED se enciende y permite detectar la intensidad de luz roja reflejada en una superficie cercana.

Figura 4.4: Sensor de luz del kit Lego Mindstorms NXT

Sensor ultrasónico: Se considera como un sensor digital, debido a la señal detectada es preprocesada antes de enviarse al ladrillo NXT y ésta no es transmitida en función de un voltaje, sino que se envía en forma de mensaje usando el protocolo de comunicación I^2C. Cuenta con un emisor sonoro, un receptor y

un circuito digital que procesa la información obtenida. Su principio se basa en enviar pulsos sonoros de alta frecuencia (200KHz) hacia una superficie, y medir el periodo de tiempo entre los pulsos enviados y los pulsos de eco detectados por el receptor, calculando la distancia recorrida por el sonido. Con este sensor se pueden detectar obstáculo con un rango entre 5 y 150 centímetros.

Figura 4.5: Sensor ultrasónico del kit Lego Mindstorms NXT

Sensor de sonido: Está constituido por un micrófono y un circuito analógico de amplificación y filtrado. La señal obtenida es el nivel de presión sonora. Su rango de operación es de 50dB a 90dB.

Figura 4.6: Sensor de sonido del kit Lego Mindstorms NXT

Encoder: Son sensores internos localizados dentro de cada uno de los motores, los cuales permiten determinar la dirección, posición y velocidad de rotación. Se componen de un disco con escalas de cuadrículas espaciadas uniformemente, éstos giran junto con la flecha de su respectivo motor. De un lado de la escala se encuentra una fuente de luz, del otro hay fototransistor. El fototransistor genera un pulso cada vez que un rayo de luz atraviesa una cuadrícula, de este modo, por medio de software se lleva un contador para tomar el registro

de rotación del motor. Para determinar el sentido, se encuentra un sistema idéntico en desfase, de tal modo que al sumar los dos trenes de pulsos se obtenga una señal diferente para cada sentido.

Sensor de brújula: Contiene una giroscopio digital de dos ejes, basado en tecnología microelectromecánica (MEMS). Éste permite medir el campo magnético de la tierra y devuelve el ángulo al que apunta sobre el eje horizontal. Utiliza el protocolo I^2C para transmitir las mediciones obtenidas.

4.2 Lenguajes de programación

Una de las razones por la que Lego Mindstorms NXT se ha vuelto popular, principalmente en el ámbito educativo, es por la variedad de lenguajes con los que es posible programarlo. A pesar de que éste cuenta con un lenguaje oficial, la versatilidad con la que fue diseñado ha permitido que se amplíen sus posibilidades de programación con diversos lenguajes.

En esta sección se muestran los lenguajes más relevantes que han sido utilizados para la programación de robots usando la plataforma Lego Mindstorms NXT.

NXT-G Lego Mindstorms NXT dispone de un lenguaje oficial llamado NXT-G [35], diseñado por la empresa National Instrument. Éste utiliza un entorno de programación gráfico, el cual ofrece herramientas con iconos intuitivos (fig. 4.7). Sus instrucciones o comandos, representadas por bloques de instrucciones, son manipulados mediante un enfoque ”drag and drop”. Sus bloques de instrucciones se dividen en dos tipos: bloques de control de componentes de la plataforma robótica (sensores, motores, display LCD, etc), y bloques de control de flujo, estos últimos son las estructuras de control clásicas de la programación estructurada.

Los bloques de instrucción de NXT-G representan funciones que se enlazan secuencialmente sobre un riel que representa el flujo principal del programa, además, éstos cuentan con entradas y salidas de las señales adquiridas por los sensores, las cuales son tipificadas por medio de colores. Cada bloque de instrucción contiene un panel de configuración que permite configurar sus opciones de funcionamiento particulares.

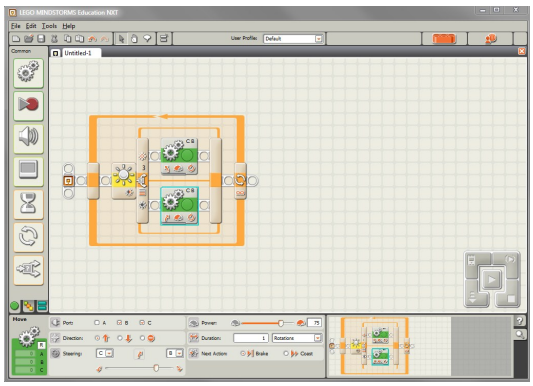

Figura 4.7: Entorno gráfico de NXT-G.

Un solo bloque de instrucción del lenguaje NXT-G, traducido a algún otro lenguaje de programación basado en texto, podría equivaler a varias líneas de código. Esto proporciona un beneficio para los programadores novatos, simplificando el tiempo de programación. Sin embargo, la cantidad de instrucciones generadas en código de bytes, por cada bloque de instrucción, es mucho mayor que la que se podrían generar con otros lenguajes [36]. Cabe destacar que los programas desarrollados en NXT-G se trasladan a la memoria del ladrillo NXT, y se ejecutan sobre éste.

RobotC RobotC es un lenguaje de programación basado en C, éste cuenta con su propio entorno de trabajo para escritura y depuración de programas para diversas plataformas robóticas, en las que se encuentra Lego Mindstorms NXT. El entorno de desarrollo integrado (IDE por sus siglas en inglés) incluye un revisor de sintaxis, que compila durante la escritura en tiempo real mostrando ayuda contextual y auto-completa funciones y variables (fig. 4.8). Su depurador permite ejecutar un programa paso a paso, establecer puntos de paro y observar variables, o solo observar la ejecución del código sobre el ladrillo NXT. Además, RobotC utiliza un remplazo del firmware original del ladrillo NXT, con el cual proporciona algunas mejoras de rendimiento como ejecuciones de código mas rápidas, mejor gestión de memoria, mejoras en los controladores de los motores, entre otras.

RobotC es un producto comercial, dirigido al mercado de la educación, desarrollado por la academia de robótica de Universidad Carnegie Mellon. Su sitio oficial en Internet [37] cuenta con una completa documentación, además de un foro en el que la comunidad comparte tanto información de ayuda, así como proyectos completos. Dado que éste es un producto comercial, su uso es exclusivo para aquellos que han comprado una licencia, sin embargo, existe una versión de prueba de 30 días.

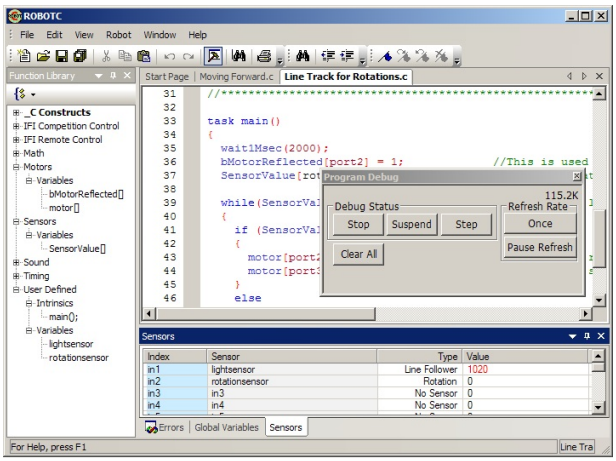

Figura 4.8: IDE de RobotC

NexT Byte Codes (NBC) y Not eXactly C (NXC) NBC Fue uno de los primeros lenguajes de programación basados en texto desarrollados para la primer versión de la plataforma robótica Lego Mindstorms, aunque se desarrolló también una versión para Lego Mindstorms NXT. NBC utiliza el firmware original del ladrillo NXT para la ejecución de sus programas. Su sintaxis se basa en lenguaje ensamblador, sin embargo, no es un lenguaje ensamblador de propósito general, debido a que el firmware original del ladrillo NXT posee algunas restricciones para el manejo de sus microcontroladores [38].

El lenguaje NXC es un lenguaje de alto nivel similar a C. Los programas escritos en éste, al compilarse, son convertidos en codigo de NBC, por lo cual posee las mismas restricciones que NBC. Tanto NBC como NXC utilizan el IDE Brix Command Center, el cual al igual que RobotC, también cuenta con herramientas de depuración en tiempo real.

leJOS NXJ LeJOS es un ambiente de programación basado en Java, de código abierto, para Lego Mindstorms. Éste utiliza un remplazo de firmware sobre el ladrillo NXT que incluye una pequeña máquina virtual de Java proporcionando mejoras sobre el manejo de motores y sensores así como sobre el rendimiento de memoria gracias a su recolector de basura. Cuenta con una biblioteca de clases Java para el ladrillo NXT y otro para programas de computadora que se comunican con el ladrillo NXT vía alámbrica o inalámbrica. Además incluye algunas herramientas para dar soporte durante el proceso de programación [19]. A diferencia de los lenguajes anteriormente mencionados, basados en programación estructurada, éste esta basado en programación basada en objetos.

Parte II

Implementación

CAPÍTULO

5

Implementación

En la sección 3.2 se presentaron los trabajos relacionados con los controles de robots cognitivos más recientes que utilizan un LPA BDI para su parte cognitiva, así como algunos requerimientos que deberían tener los LPA para la programación de robots cognitivos. Estos trabajos tienen aspectos en común que son relevantes para la implementación de la capa de control de componentes para el control de robot cognitivos del presente proyecto. Uno de los principales aspectos es el uso de una arquitectura en capas, que además de proporcionar de alguna manera un equilibrio entre la reactividad y deliberación necesarias para un robot, ésta también proporciona un esquema de separación ordenado sobre el manejo de los recursos y sus respectivas funcionalidades dentro del sistema. En este capítulo se describe la manera en que se ha implementado la arquitectura de tres capas para el control de un robot cognitivo, haciendo énfasis en la integración de la capa de control de componentes dentro de la arquitectura.

5.1 Descripción general de la arquitectura empleada

La arquitectura que hemos utilizado está inspirada en la arquitectura clásica de tres capas [18]. Las capas de nuestra arquitectura se encuentran organizadas jerárquicamente de arriba hacia abajo en función de su nivel de abstracción 5.1. La capa

cognitiva está asociada a los procesos de deliberación y control de comportamientos; la capa de control de componentes está asociada al modelado del ambiente y gestión de los componentes del robot; y la capa funcional está asociada al control del hardware de la plataforma robótica.

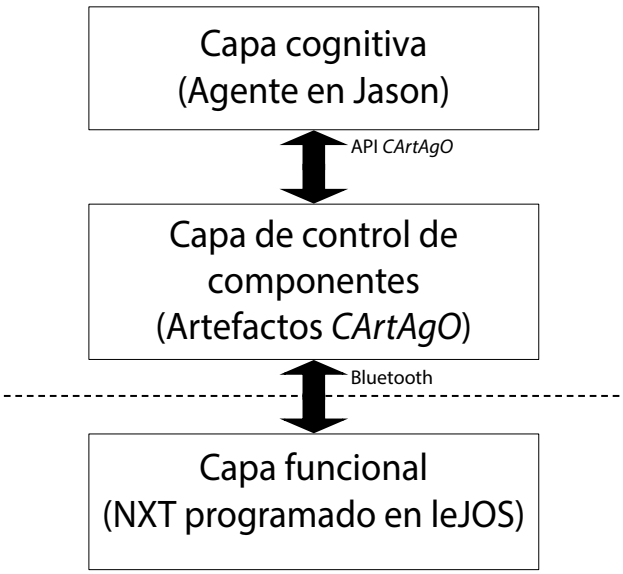

Figura 5.1: Representación de la arquitectura, vista desde la interacción entre las capas.

La capa cognitiva, implementada en Jason, utiliza el ciclo de razonamiento de *AgentSpeak(L)* para llevar a cabo la toma de decisiones, así como la selección y representación de comportamientos por medio de eventos y planes respectivamente.

La capa de control de componentes utiliza el modelo de ambiente de agentes *A&A* de *CArtAgO*, en el cual se modela cada componente físico del robot dentro del ambiente del agente por medio de artefactos especializados. Los artefactos permiten al agente manipular y atender de manera individual cada uno de los componentes de la plataforma robótica. Estos también proporcionan los mecanismos para reconocer las señales adquiridas de los sensores y convertirlas en percepciones simbólicas para la capa cognitiva, así como identificar las acciones del agente y transformarlas en instrucciones para trasmitirlas a la capa funcional.

Finalmente, la capa de ejecución, programada en lenguaje leJOS sobre la plataforma robótica Lego Mindstorms NXT, es encargada de activar y manipular los componentes físicos del robot, en lo que esta involucrado el movimiento de motores y la adquisición de señales de los sensores.

5.2 Capa cognitiva

La capa cognitiva es implementada con el lenguaje *AgentSpeak(L)* dentro de Jason utilizando una arquitectura de agente extendida de la clase *CAgentArch* provista por *CArtAgO*. Ésta es utilizada para generar, durante la inicialización del programa del SMA, una espacio de trabajo particular para cada agente definido en el archivo MAS del proyecto en Jason. Este espacio de trabajo no excluye la posibilidad para un agente de cambiarse a otro espacio de trabajo. La función de éste es poner a disposición del agente la capacidad de instanciar los artefactos para el control de un solo robot. Los artefactos instanciados dentro un espacio de trabajo solo pueden ser manipulados por los agentes que se encuentre dentro del mismo.

Con lo mencionado anteriormente podemos definir que una agente es asignado al control de un solo robot siempre que el agente se mantenga en el mismo espacio de trabajo. Esto no quiere decir que un agente no pueda manipular más de un robot simultáneamente, debido a que no se impide la posibilidad de cambio de espacios de trabajo. Sin embargo, con la finalidad de evitar una sobrecarga de trabajo es recomendable asignar el control de un solo robot específico para cada agente.

5.3 Capa de control de componentes

La capa de control de componentes está compuesta por un conjunto de artefactos relacionados, cada uno, a un componentes de la plataforma robótica Lego Mindstorms NXT. A continuación se describen los artefactos involucrados.

Artefacto administrador de componentes Para que la capa cognitiva pueda tener control sobre cada artefacto de componente de la plataforma robótica es necesario inicializar el correspondiente controlador de artefacto, crear un enlace bidireccional entre dicho artefacto y el artefacto de comunicación (exceptuando al propio artefacto de comunicación), y posteriormente enviar la

directiva a la plataforma robótica para la inicialización a bajo nivle del comoponente en cuestión. Todo este proceso es simplificado por Artefacto Administrador de componentes, creado automáticamente durante el arranque del programa principal.

Artefacto de NXT Es el encargado de establecer la comunicación Bluetooth, es indispensable para el funcionamiento de la arquitectura, debido a que éste proporciona todos los mecanismos para la interacción entre el hardware de la plataforma robótica y Jason. Todos los artefactos se encuentran ligados a éste, de tal modo que al utilizar algún otro artefacto, por medio de operaciones de artefactos, las instrucciones será enviadas a éste y se envíen posteriormente por medio de Bluetooth de manera automática. De igual forma, reconoce los mensajes enviados por el ladrillo NXT, y las redirige hacia el artefacto correspondiente encargado de reconocer la señal. A éste se liga un artefacto auxiliar, el cual puede ser utilizado únicamente por el artefacto de comunicación. Este artefacto auxiliar se encarga de estar a la escucha de mensajes enviados por el ladrillo NXT y enviarlos al artefacto principal de comunicación para su procesamiento.

Artefactos de Sensores A cada sensor se le asigna un artefacto que sirve para reconocer su señal y convertirla en mensaje simbólico para el agente, en el cual se indica el tipo de sensor utilizado, el puerto al que está conectado en el ladrillo NXT y la señal adquirida. Además, para el caso de los sensores que tienen más de un modo de operación, su artefacto correspondiente, proporciona operadores de agente que permiten cambiar entre los diferentes modos de operación.

Artefactos de Motores Los artefactos de motores cuentan con operadores de agente que permiten manipular los motores así como convertir las señales de los encoders para convertirlas en mensajes simbólicos y así permitirle al agente identificar la posición de cada uno de los motores, su estado de actividad o inactividad, y su velocidad de rotación.

Artefacto DifferentialPilot Este artefacto es utilizado para el control de motores para una estructura con motores diferenciales. Con éste es posible utilizar

operaciones que permiten mover el robot como si se tratara de un vehículo, cuyos parámetros utilizan unidades relacionadas a la superficie y no a las motores individuales. Como por ejemplo se le indica al robot moverse 10cm en vez de calcular cuántos grados necesitan girar las motores para que el motor se desplace 10cm. Para utilizar este artefacto es necesario indicar el diámetro de las ruedas, la distancia entre las ruedas y los puertos a los que se conecta cada motor. Al usar este artefacto no se recomienda el uso de algún otro artefacto de motores.

Los artefactos de sensores son inicializados por defecto en modo asíncrono, con ello no se genera ningún evento provocado por señales de sensores hasta que sen focalizados y se solicite un dato sensado por medio de una consulta, para ellos se usa la operación `readValue`. El valor obtenido se verá reflejado en una propiedad observa con el nombre del artefacto en cuestión.

Por otro lado si se desea utilizar un sensor en modo síncrono es necesario cambiar su modo de operación mediante la operación setSynchronMode. Con ello la propiedad observable correspondiente a dicho artefacto sensor será actualizada de manera continua y la transmisión de datos de dicho sensor será controlada desde la capa de control de componentes.

Cabe mencionar que artefactos de motores y DifferentialPilot contiene un conjunto de operaciones que representan instrucciones sobre los motores y un conjunto de operaciones que representan mensajes de consulta cuya respuesta es reflejada en una propiedad observable. Éstas operaciones envuelven instrucciones de las clases Motor y DifferntialPilot de leJOS cuyas especificaciones se encuentran en [19]. Las operaciones de instrucciones no esperan a que la instrucciones hayan sido realiza, por lo que es posible interrumpir el movimiento de motores con otra operación de instrucción. Para que un agente pueda esperar a que termine un movimiento es posible mediante un plan que verifique si los motores se encuentran en movimiento o no.

5.4 Capa funcional

La capa funcional, encargada de control directo del hardware, está implementada mediante el lenguaje leJOS [19] sobre el ladrillo NXT de la plataforma robótica

Lego Mindstorms NXT. El programa implementado para ésta capa se ha basado, en gran parte, en la implementación del trabajo de Jensen [14], el cual se compone de por un conjunto de objetos, funcionan bajo hilos separados, que controlan cada componente. El problema de la implementación de Jensen es que éstos objetos tienen que ser creados y configurados al momento en que se inicializa la conexión con el agente. Y una vez generados no pueden ser reconfigurados para diferentes modos de uso.

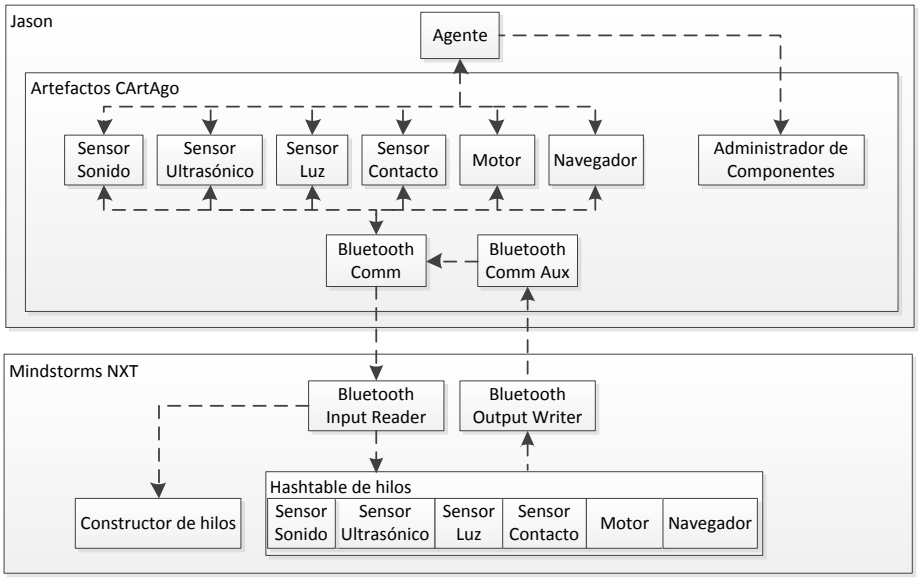

Figura 5.2: Relación entre *capa de control de componentes* y *capa funcional*.

En nuestra implementación se ha añadido un administrador de hilos, equivalente al artefacto administrador de componentes de la capa de control de componentes, encargado de generar los hilos cuando se reciba la solicitud correspondiente por parte de la capa de control de componentes en cualquier momento. Ofreciendo un manejo dinámico de cada componentes, como se hace con los artefactos dentro de la capa de control de componentes. Esta capa refleja la equivalencia de los artefactos de componentes con los hilos como se muestra en la figura 5.2. Las flechas dentro de ésta indican dirección de flujo de datos, tema que será abordado en

la sección 5.6. Además, ésta implementación tiene una mayor funcionalidad, debido a que se han incorporado un conjunto mayor de funciones para el manejo de motores.

5.5 Integración entre capa de control de componentes y capa funcional

Debido a que el único enlace entre la capa de control de componentes y la capa funcional es un canal de comunicación unidereccional con un ancho de banda limitado, es necesario una interfaz confiable capaz de garantizar que la capa funcional ejecute las instrucciones dadas por la capa de control de componentes y con suficiente rendimiento que permita al robot tener la reactividad necesaria para llevar a cabo sus tareas.

El protocolo de comunicación implementado entre la capa de control de componentes y la capa funcional usa un modelo esclavo maestro. El artefacto de comunicación dentro de la capa de control de componentes toma el rol de control de dispositivo maestro, mientras que un hilo dentro del programa del ladrillo NXT toma el rol de control de dispositivo esclavo. En este protocolo el dispositivo maestro está encargado de crear el canal, iniciar la comunicación y del envío de mensajes que pueden contener instrucciones o solicitudes. El dispositivo esclavo envía mensajes de respuesta al dispositivo maestro, y si éste necesita enviar un mensaje sin la existencia de alguna solicitud del dispositivo maestro espera a que el canal se encuentre libre. Cuando el dispositivo maestro envía un mensaje de instrucción, espera por un mensaje de reconocimiento de parte del dispositivo esclavo para continuar con el envío de mensajes. Cuando el dispositivo esclavo recibe un mensaje de instrucción primero ejecuta dicha instrucción y posteriormente envía el mensaje de reconocimiento al dispositivo maestro.

El mensaje de reconocimiento proporciona una garantía sobre la funcionalidad de la comunicación y permite al dispositivo maestro mantener una sincronía de envío de mensajes. Por otro lado, sin el uso de mensajes de reconocimiento, el dispositivo esclavo podría mantener una gran pila de instrucciones enviadas por el

dispositivo maestro esperando por ser ejecutadas, provocando un posible desface en la ejecución de acciones.

5.6 Control del flujo de datos

El ladrillo cuenta con dos dispositivos de comunicación: cable USB y Bluetooth. Sin embargo, solo es posible usar un canal de comunicación a la vez entre el ladrillo NXT y la computadora. Cada uno tiene sus beneficio y perjuicios. El primero tiene la ventaja de ser un medio de comunicación rápido y con un nivel de pérdidas de datos no significativo, sin embargo, es de corto alcance (limitado por la longitud del cable), además de disminuir la libertad de movimiento. Para el segundo, los beneficios y perjuicios son a la inversa, disponiendo de un mayor alcance con total libertad de movimiento sobre el ambiente, pero con una comunicación más lenta y con posibilidad de pérdidas de datos en un nivel significativo [39].

Para efectos prácticos, se ha elegido el modo de comunicación Bluetooth para la comunicación entre la plataforma robótica Lego Mindstorms NXT y Jason, sin embargo, no se descarta la posibilidad de implementar una extensión que permita también la comunicación por medio de cable USB.

Como se ha mencionado en la sección 5.3, la comunicación por el lado de Jason, se lleva a cabo por medio de un artefacto principal y uno auxiliar. El artefacto de comunicación principal, al momento de ser creado dentro de la capa cognitiva, se encarga de generar un canal de comunicación entre Jason y el dispositivo Bluetooth del ladrillo NXT identificado con el nombre del dispositivo y su dirección Bluetooth. Del mismo modo, genera y enlaza de manera automática el artefacto auxiliar que le permitirá recibir mensajes enviados por ladrillo NXT.

Cuando se genera algún otro artefacto de componente, éste es enlazado con el artefacto de comunicación. Con lo cual, este último envía un mensaje al ladrillo NXT para la activación del componente correspondiente. Así, cuando se recibe un mensaje desde el ladrillo NXT, se identifica el componente y la señal es redireccionada al artefacto que provoco su activación.

5.7 Modalidades de adquisición de datos de sensores

Una manera de administrar los datos que son enviados desde la capa funcional a la capa de control de componentes es eligiendo un modo de transmisión de datos para cada necesidad. Se utilizan dos modos de transmisión, estos son de manera síncrona y asíncrona.

Al enviar los datos de los sensores desde la capa funcional de manera síncrona, está no requiere de solicitudes para enviar los datos, sino que los envía de manera continua, no sin antes verificar que éstos no sean datos redundantes, es decir, si el último dato adquirido es igual al dato anterior significa que las capas superiores ya han recibido esa información. Al hacer esto, además de reducir los datos enviados por el canal de comunicación entre la capa funcional y la capa de control de componentes, se disminuye la carga de eventos que deben ser procesados por la capa cognitiva.

Por otro lado, también es posible manipular los sensores de manera asíncrona, es decir, que la adquisición de datos de los sensores sea administrada desde la capa cognitiva. De ésta manera los datos obtenidos por un sensor son enviados únicamente cuando se requieren.

El modo de adquisición de datos de los sensores puede cambiarse de manera dinámica en tiempo de ejecución según lo requiere la aplicación.

CAPÍTULO

6

Experimentos

6.1 Movimiento aleatorio

Para observar los movimientos básicos del robot se ha diseñado un experimento en el que un robot, con una estructura física como la de un vehículo con motores diferenciales, debe moverse aleatoriamente.

Para esta práctica se ha implementado el programa del agente mostrado en el cuadro 6.1. En éste se distinguen tres planes que serán explicados a continuación.

Debido a que la meta inicial del agente es `!configuraRobot`, el plan aplicable a intentar primero es *p_configura*. Con el cual, por medio de operaciones del artefacto `ComponentManager` (instanciado por defecto desde el inicio del SMA usando la arquitectura de agente implementada en esta proyecto de investigación), añade un componente `NXT` para la comunicación con el ladrillo NXT y un componente `differentialPilot` para el control de los motores en modo diferencial. Posteriormente focaliza el artefacto del componente `differentialPilot` para percibir sus propiedades observables. Una vez añadidos los componentes necesarios, establece las velocidades de movimiento del robot usando las operaciones `setTravelSpeed(...)` y `setRotateSpeed(...)`, estableciendo con esto una velocidad de 7 centímetros/segundo para el movimiento en línea recta y 60

grados/segundo para la rotación sobre el eje central del robot, y al final se plantea la meta `!move`.

```
/* Metas iniciales */
!configuraRobot.

/* Planes */
[p_configura]
+!configuraRobot: true
  <- addNXT("BTComm","nxt","00165305CC20");
    addComponent("DifferentialPilot","pilot","nxt",5.6, 11.3,"B","C");
    lookupArtifact("pilot",PilotID); focus(PilotID);
    setTravelSpeed(7.0); setRotateSpeed(60.0); !move.

[p_aleatorio]
+!move: true
  <- .random(R0); travel(R0*30);
    !esperaFinMovimiento; .random(R1);
    if(R1*3 < 1){.random(R2); rotate(R2*110); }
    if(R1*3 >= 1 & R1*3 < 2){.random(R2); rotate(R2*(-110)); }
    !esperaFinMovimiento; !!move.

[p_espera]
+!esperaFinMovimiento: true
  <- isMoving; while(moving(1)){ isMoving; .wait(10); }.
```

Cuadro 6.1: Agente del experimento `Movimiento aleatorio`

De nuevo solamente hay un plan aplicable para la nueva meta a intentar, este plan es *p_aleatorio*, donde al ejecutar el cuerpo de este plan el robot primero intentará moverse en línea recta de 0 a 30 centímetros aleatoriamente. Posterior a ésto el agente se plantea la meta alcanzable `esperarFinMovimiento` y no continuará hasta que el movimiento indicado haya terminado. Cabe aclara que la ejecución de las operaciones implementadas que requieran usar los motores no esperan a que éstos terminen de moverse, sino que devuelven el control inmediatamente después de haber verificado que el ladrillo NXT haya recibido la instrucción dada. Posteriormente intentará girar a la izquierda o a la derecha de 0 a 110 grados, también aleatoriamente, y volverá a esperar a que terminen de moverse los motores mediante el planteamiento de la meta `esperarFinMovimiento` para que finalmente el procedimento se repita recursivamente mediante el planteamiento de la meta `!!move`.

El plan *p_espera* se encarga de verificar que los motores del robot hayan dejado de moverse mediante el uso de la operación `isMoving` y la propiedad observable `moving(Int)` (ambas del artefacto componente `differentialPilot`), dentro del ciclo *while*.

6.2 Evasor de obstáculos

En ésta practica se propone utilizar el movimiento aleatorio del robot implementado en la práctica anterior, además de añadirle un comportamiento extra.

Al colocar al robot de la práctica anterior sobre el suelo, es fácil que éste colisione con cualquier obstáculo o simplemente contra una pared, cuando ésto ocurre lo más común es que el robot pierda movilidad y dependa de alguien que lo auxilie para cambiarlo de posición para continuar con su movimiento normal. A modo de incrementar la autonomía del robot de la práctica anterior, se propone añadir los componentes y planes necesarios para que éste pueda evadir obstáculos.

Al añadirle un sensor ultrasónico al frente de la estructura física, y del mismo modo añadiendo el control del componente correspondiente dentro del programa, es posible programar un conjunto de planes que permitan al robot interrumpir el movimiento aleatorio para evadir los obstáculos que se le presenten en su camino. Para lograrlo se ha modificado el plan *p_configura* añadiendo el componente `UltrasonicSensor` y activado su modo de adquisición de datos síncronamente, a modo que los cambios en los valores obtenidos por dicho sensor generen eventos. El nuevo plan *p_configura* puede observarse en el cuadro 6.2.

```
1 [p_configura]
2 +!configuraRobot: true
3   <- addNXT("BTComm","nxt","00165305CC20");
4     addComponent("DifferentialPilot","pilot","nxt",5.6, 11.3,"B","C");
5     addComponent("UltrasonicSensor","us","4","nxt");
6     lookupArtifact("pilot",PilotID); focus(PilotID);
7     lookupArtifact("us",USID); focus(USID);
8     setSynchronMode[artifact_name("us")];
9     setTravelSpeed(7.0); setRotateSpeed(60.0); !move.
```

Cuadro 6.2: Plan *p_configura* de la práctica `Evasor de obstáculos`

Del mismo modo, el plan `p_aleatorio` de la práctica anterior es modificado. Para que pueda compartir el mismo evento disparador con otros planes se ha cambiado la primitiva `true` de su contexto por `not obstaculo`, como se observa en la el cuadro 6.3, con lo que se establece que el plan será aplicable únicamente cuando el agente no contenga `obstaculo` dentro de sus creencias.

```
[p_aleatorio]
+!move: not obstaculo <- // cuerpo del plan ...
```

Cuadro 6.3: Plan *p_aleatorio* de la práctica `Evasor de obstáculos`

Adicionalmente se han añadido los planes del cuadro 6.4 para el manejo de los eventos para la detección evasión de obtáculos.

```
[p_evadir]
+!move: obstaculo
  <- travel(-5); !esperaFinMovimiento;
    rotate(180); !esperaFinMovimiento;
    !move.

[p_obstaculo_encontrado]
+us(Distancia): not obstaculo & Distancia < 20
  <- .drop_intention(move); .drop_intention(esperaFinMovimiento);
    stop; +obstaculo; !move.

[p_obstaculo_evadido]
+us(Distancia): obstaculo & Distancia > 20
  <- .drop_intention(move); .drop_intention(esperaFinMovimiento);
    stop; -obstaculo; !move.
```

Cuadro 6.4: Planes adicionales de la práctica `Evasor de obstáculos`

El plan *p_obstaculo_encontrado* es aplicable cuando se encuentre un obstáculo a una distancia menor de 20cm del sensor y además cuando el agente no tenga la creencia `obstaculo`. Ésto permite que el plan no vuelva a ser aplicable con cualquier cambio en el dato sensado mientras el obstáculo siga presente. Dentro del cuerpo de este plan se utiliza la acción interna `.drop_intention(...)` para eliminar las intenciones del conjunto de intenciones del agente relacionadas

con el movimiento del robot. Con ésto no se detiene el movimiento presente de los motores, es por ello que tras eliminar dichas intenciones se forza a detener los motores usando la operación `stop`. Posteriormente añade `obstaculo` a la base de creencias y vuelve plantearse la meta `!move`, con lo que ahora el plan aplicable es *p_evadir*.

Cuando el obstáculo haya sido evadido, el plan *p_obstaculo_evadido* será aplicable y el agente volverá a eliminar las intenciones relacionadas con el movimiento, detendrá los motores, eliminará la creencia `obstaculo` y finalmente volverá a plantearse la meta `!move` para continuar con el movimiento aleatorio del plan `p_aleatorio`.

6.3 Robot Aprendiz

El caso de estudio para este experimento se define a continuación: Un organismo que busca alimento sin una técnica de búsqueda específica tiene que aprender a identificar cuál de los alimentos disponibles en su entornos son benéficos para él y cuales no. La forma de diferenciar los alimentos es debido a un conjunto de características o atributos, algunas de estas características son más representativas que otras. El organismo debe aprender a distinguir, en base a prueba y error, cuáles son las características representativas para identificar el alimento no benéfico más rápidamente y así evadirlo sin necesidad de evaluar las demás características de manera innecesaria.

Haciendo una analogía con el ejemplo anterior, nuestro robot representa el organismo que busca alimento, mientras que el alimento esta representado por un conjunto de círculos esparcidos por un escenario. Cada círculo esta compuesto por 3 anillos que representan los atributos y un círculo interior que representa el tipo de alimento o su clase. Por practicidad, tanto los atributos como la clase se han designado con valores binarios. El escenario propuesto para la práctica es similar al de la figura 6.1.

La práctica se divide en dos etapas, la primera es la etapa de aprendizaje en la que el robot debe recolectar las características de cada alimento como ejemplos de una base de datos y generar con estos un árbol de decisiones mediante el algoritmo J48. La segunda etapa es la fase de prueba, en la que el robot deberá encontrar

y alimentarse únicamente del alimento benéfico, identificando también el alimento no benéfico para rechazarlo al reconocerlo mediante clasificación con el árbol generado, evitando la necesidad de ver todos sus atributos.

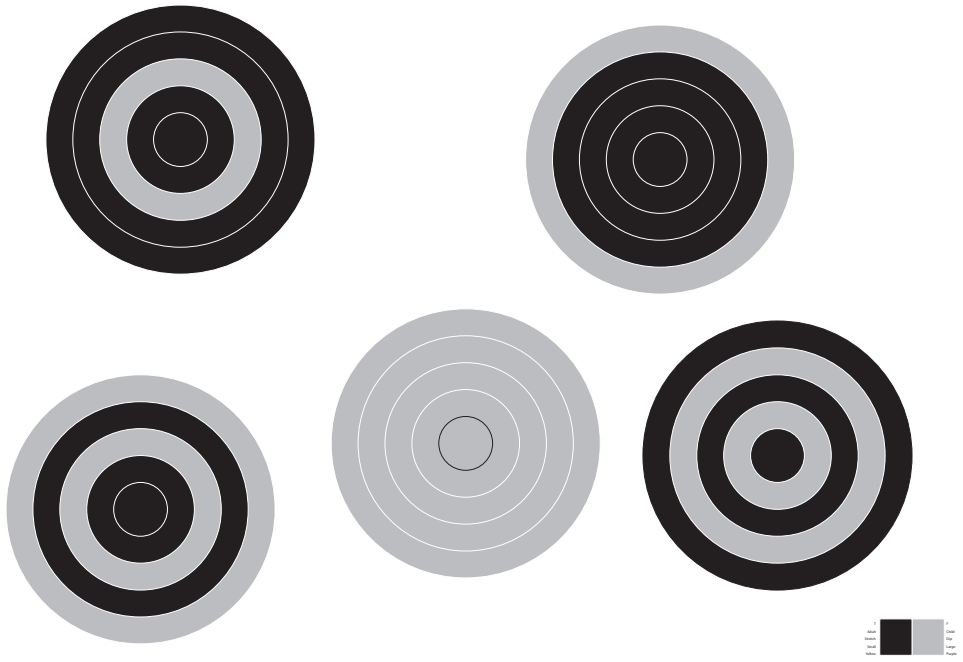

Figura 6.1: Ejemplo de parte del escenario utilizado para el experimento del robot aprendiz

La estructura física del robot (fig. 6.2) utiliza dos motores manipulados como un vehículo con motores diferenciales, un sensor de luz utilizado para el reconocimiento del alimento, y un sensor de contacto para mecanismo adicionales como calibración y cambio de fases. Análogamente, el robot es construido dentro del programa de la capa cognitiva, con lo que se activan los correspondientes artefactos componentes asociados dentro de la capa de control de componentes.

Para el aprendizaje y clasificación se ha implementado un artefacto llamado ClassifierJ48 [40] (fig.6.3) que encapsula funciones de minería de datos del software WEKA. Las operaciones implementadas en el artefacto `ClassifierJ48` son `addInstance`, `buildClassifier` y `classifyInstance`, las cuales

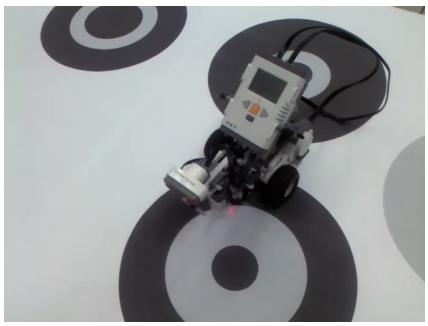

Figura 6.2: Estructura física del robot utilizado en el experimento de aprendizaje

se utilizan para añadir ejemplos a la base de datos, construir un modelo J48 con los ejemplos de la base de datos, y clasificar una instancia con el modelo construido.

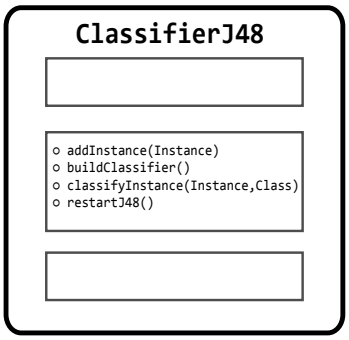

Figura 6.3: Artefacto ClassifierJ48 extraído del trabajo de Limón et al [40]

Tras la inicialización de los artefactos componentes, el artefacto ClassifierJ48 y una calibración para identificar los colores del escenario, el agente controlador obtiene la meta !move, la cual puede activar cualquier plan relacionado con el movimiento definiendo un comportamiento específico.

Las creencias que definen el comportamiento del agente controlador son: `wondering`, `aligning` `reading_of_rings`, `learning` y `testing`. La creencia `wondering` está relacionada con un comportamiento de movimiento aleatorio dentro del escenario siempre que no se encuentre un círculo de alimento; `aligning` se relaciona con el comportamiento de alineación al anillo exterior de

algún circulo de alimento encontrado; `reading_of_rings` está relacionada con un comportamiento de lectura de cada anillo del circulo de alimento encontrado; `learning` y `testing` definen la fase del experimento y el procesamiento de la información adquirida durante el comportamiento de lectura de anillos.

Mientras que el valor obtenido por el sensor de luz no sea el de algún color que represente la presencia de alimento, se mantendrá la creencia de comportamiento `wondering` recurriendo al mismo plan de movimiento usado en el experimento de movimiento aleatorio. Invocando el mismo plan recursivamente hasta que la creencia de comportamiento cambie o hasta sea interrumpido por alguna de las acciones internas `.drop_intention(...)` o `.drop_all_intentions` activadas dentro de otro plan.

Las señales adquiridas por los artefactos de componentes sensores son usados en modo síncrono para activar planes que definen las creencias de comportamiento. Este plan es activado cuando el valor obtenido por el sensor de luz sea el de algún color que represente la presencia de alimento. Cuando se activa remplaza la creencia de comportamiento wondering por aligning, desecha las intenciones de movimiento y vuelve a generar la meta !!move de manera recursiva.

Cuando la creencia de comportamiento presente es `aligning` el agente controlador intentará obtener una alineación tangente a la curva del anillo exterior del alimento encontrado. Para lograrlo avanza únicamente con uno de sus motores hasta que el sensor de luz obtenga una señal que represente que ha salido del círculo de alimento, inmediatamente después detiene el motor y activa el otro haciendo que regrese al interior del círculo de alimento. Estos movimientos son repetidos siguiendo el contorno del anillo exterior mientras se mide la distancia de desplazamiento de un lado a otro. Cuando la distancia de desplazamiento es menor a un valor de umbral, que representa la distancia de desplazamiento mínima aproximada, termina la alineación y cambia al comportamiento `reading_of_rings`.

Al comienzo del comportamiento `reading_of_rings` el sensor de luz se cambia a modo asíncrono debido a que solo se usará para obtener los valores de los colores de los anillos que representan los atributos. Para ello el robot se desplazara desde el perímetro del círculo hacia su centro por pasos con distancias específicas para la lectura de cada anillo, dichas distancias son obtenidas con mediciones previas al experimento. Al terminar cada paso, toma una lectura del sensor de luz y

continúa con el siguiente hasta terminar en el círculo interior. Al finalizar la lectura añade el ejemplo obtenido a la base de datos de entrenamiento, genera un nuevo modelo J48 con el ejemplo añadido, sale del círculo de alimento y regresa el modo de sensor de luz a síncrono para retomar el comportamiento `wondering`.

Debido a que no es posible definir un parámetro para indicar al robot cuantos ejemplos son suficientes para finalizar la fase de aprendizaje por utilizar un método de aprendizaje supervisado, se ha utilizado un procedimiento manual utilizando el sensor de contacto. Al presionarlo la creencia de comportamiento learning es remplazada por testing. Con esto el comportamiento de reading_of_rings se ve afectado haciendo que tras la lectura de cada anillo de un círculo de alimento se clasifique el ejemplo semiconstruido y salga del círculo si la el resultado de la clasificación representa la clase del alimento no benéfico, de lo contrario continuará con la lectura del siguiente anillo sucesivamente hasta el círculo interior, donde el robot se alimentado con el alimento benéfico de manera metafórica. Al terminar el proceso de alimentación sale del círculo de alimento y continúa el proceso de exploración aleatoria en busca de más alimento.

6.4 Resultados y Discución

A pesar de que los experimentos que se describen en este capítulo parecieran ”mundos de juguete”dentro de un ambiente físico, estos demuestran la viabilidad de usar una capa de control de componentes para manejar la interacción entre una capa cognitiva y una capa funcional dentro de la arquitectura de un robot cognitivo. También se demuestra la viabilidad de la separación del control de componentes físicos dentro de módulos representados por artefactos *CArtAgO* y con el mismo enfoque también es posible extender las capacidades del robot, no solo para otros componentes físicos, sino también para softwares externos que auxilien en las tareas de un robot, como puedo observarse en el experimento de aprendizaje (sección 6.3) donde se añadió un artefacto utilizando bibliotecas de Weka.

Como puede observarse en los primeros dos experimentos (secciones 6.1 y 6.2), únicamente se presenta el programa del agente en Jason de capa cognitiva de la arquitectura, en el que se hace uso de operaciones de los artefactos *CArtAgO* implementados en la capa de control de componentes. Ésto es debido a que tanto la

capa de control de componentes, como la capa funcional de nuestra implementación han sido diseñadas para que el programador final no requiere modificarlas, a no ser que requiere mejorarlas, depurarlas o extenderla como en el caso del tercer experimento (sección 6.3). Con ello, la programación mínima requerida para que un robot pueda desempeñarse en una tarea específica se concentra básicamente en el uso de un solo lenguaje. Facilitando la puesta en marcha de la experimentación.

Otro aspecto importante a resaltar es la facilidad de programar un robot de manera incremental mediante la reutilización de planes, con lo que es posible disminuir el tiempo de diseño, programación y depuración. Como ejemplo, se observa que los planes de movimiento de los experimentos realizados son muy similares, con algunas diferencia en sus contexto debido a que en cada programa las circunstancias para que éstos sean aplicables pueden variar.

CAPÍTULO

7

Conclusiones

7.1 Comparación con trabajos relacionados

La arquitectura para el control de robots cognitivos implementada en este trabajo de investigación tiene algunos puntos relacionado con los trabajos mencionados en la sección 3.2 que deben ser explorados para verificar las principales aportaciones obtenidas así como aquellas deficiencias que podrían cubrirse en trabajos futuros.

Empezando por comparar este trabajo con el de Wei [16], se observa que ambas se centran en el diseño modular para el control de robots cognitivos utilizando arquitecturas base similares. Sin embargo, debido a que evidentemente el alcance de cada trabajo es diferente, esta comparación se enfoca únicamente en aquellos factores que tienen en común. Por un lado, en el trabajo de Wei se contempla un diseño global proponiendo una arquitectura completa, mientras que el presente trabajo se concentra principalmente en una de las capas. Partiendo de esto, las comparaciones consecuentes se centrarán en las capas que comparten funciones similares dentro de cada arquitectura. Estas son la capa de control de componentes de nuestra arquitectura y la capa de interfaz que se utiliza en el trabajo de Wei entre su capa cognitiva y su capa de control de comportamientos.

Por otro lado, la capa de control de componentes de nuestra implementación pretende cumplir con algunas de las funciones integradas en la capa de control de

comportamientos de la implementación de Wei, como es el manejo de las señales de sensores y el control de motores, así como funciones adicionales provistas por bibliotecas de softwares externos. Además, en ésta capa se lleva a cabo la tarea de generar percepciones para el agente de la capa cognitiva, tarea que en el trabajo de Wei se recurre al uso de otra capa. El concepto de un acoplamiento, que podría ser considerado como débil, entre las capas cognitiva y de control de componentes de nuestra implementación se observa en los mecanismos que utilizan los artefactos CArtAgO para poder ser manipulados por los agentes.

Una carencia que se observa en nuestra implementación es la falta de un componente de control de comportamientos dentro de la capa de control de componentes, como la que se usa en la la implementación de Wei. Es importante conciderar esta modificación en trabajos futuros, lo que permitiría aumentar el desempeño de la capa cognitiva.

En relación al trabajo de Mordenti [15], se observa la gran similitud en el uso de los mismos frameworks (*Jason* y *CArtAgO*) para la implemetación de las dos capas superiores de una arquitectura basada en tres capas. Sin embargo, la capa intermedia de nuestra propuesta tiene como propósito principal mantener organizado el control de cada componente mediante la separación de módulos usando los artefactos, lo cual es una carencia en el trabajo de Mordeni. Aunque, al igual de lo que ocurre ante el trabajo de Wei, la implementación de Moridenti presenta un mejor manejo de comportamientos que la nuestra.

Finalmente, para concluir las comparaciones con trabajos relacionados, los requerimientos mencionados en el trabajo de Ziafati [17] que se han tomado a consideración en nuestro trabajo ha sido los siguientes:

- La implementación de la arquitectura de tres capas (aunque de una manera pobre), con la que se espera facilitar en trabajos futuros el balance entre las capacidades reactivas y deliberativas del robot.

- La incorporación de un módulo sensorial representado principalmente por la capa de control de comportamientos. En ésta se resalta la facilidad que ofrece para seleccionar los componentes dinámicamente para llevar a cabo una tarea, así como los modos de acceso a los datos de los sensores proporcionados.

Con lo que se obtiene un mejor uso de los recursos tanto de comunicación, procesamiento y por consiguiente el de la energía.

7.2 Trabajos Futuros

Los controles de componentes mencionados en la sección 5.3 son implementaciones exclusivamente para los componentes de la plataforma robótica Lego Midstorm NXT como un caso de estudio. Sin embargo, no se descarta la posibilidad de proveer soporte para la integración con algún framework robótico existente para soportar más plataformas robóticas, a manera de cumplir con uno de los requisitos propuestos por Ziafati et al [17] además de extender las posibilidades de experimentación con aplicaciones robóticas complejas, como en las que se involucren componentes de visión y navegación.

Otro trabajo futuro que se requiere contemplar es el de diseñar artefactos para el manejo de comportamientos como parte de la capa de control de componentes, con lo que disminuiría el procesamiento de información de la capa cognitiva aumentando su eficiencia.

En este trabajo se ha explorado una alternativa para el control de robot similar al de los trabajos mencionados, pero con un enfoque en el que se explotan las capacidades del ambiente de agentes *CArtAgO* con la finalidad de proveer una manera modular de manejar los componentes físicos dentro de una arquitectura de tres capas para el control de robots cognitivos. A pesar de que la implementación actual es compatible tan solo con una plataforma robótica, no se descarta la posibilidad de explorar el uso de ésta como herramienta para el desarrollo de escenarios experimentales tanto para la investigación como en el campo educativo.

Referencias

[1] D. Vernon, G. Metta, and G. Sandini, "A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents," *Evolutionary Computation, IEEE Transactions on*, vol. 11, no. 2, pp. 151–180, 2007. 1

[2] Y. Lespérance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl, "A logical approach to high-level robot programming: A progress report," in *Control of the physical world by intelligent systems: papers from 1994 AAAI fall symposium*, 1994, pp. 79–85. 1

[3] H. Levesque and R. Reiter, "High-level robotic control: Beyond planning. a position paper," in *AIII 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap*, 1998. 37

[4] G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati, "A theory and implementation of cognitive mobile robots," *Journal of Logic and Computation*, vol. 9, no. 5, pp. 759–785, 1999. 39

[5] M. Shanahan and M. Witkowski, "High-level robot control through logic," in *Intelligent Agents VII Agent Theories Architectures and Languages*. Springer, 2001, pp. 104–121. 39

[6] H. Levesque and G. Lakemeyer, "Cognitive robotics," *Foundations of Artificial Intelligence*, vol. 3, pp. 869–886, 2008. 1, 37

[7] V. Lifschitz, L. Morgenstern, and D. Plaisted, "Knowledge representation and classical logic," *Foundations of Artificial Intelligence*, vol. 3, pp. 3–88, 2008. 2

[8] A. S. Rao, M. P. Georgeff *et al.*, "BDI Agents: From Theory to Practice." in *ICMAS*, vol. 95, 1995, pp. 312–319. 2, 13

[9] M. Dastani, "2APL: a practical agent programming language," *Autonomous agents and multi-agent systems*, vol. 16, no. 3, pp. 214–248, 2008. 2

[10] A. S. Rao, "AgentSpeak (L): BDI agents speak out in a logical computable language," in *Agents Breaking Away*. Springer, 1996, pp. 42–55. 2, 18

[11] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007. 2, 11, 16, 20, 22

[12] K. Hindriks, "Programming rational agents in goal," in *Multi-Agent Programming: Languages, Tools and Applications*, A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, Eds. Springer US, 2009, pp. 119–157. 2, 5, 40

[13] R. H. Bordini, L. Braubach, M. Dastani, A. El Fallah-Seghrouchni, J. J. Gomez-Sanz, J. Leite, G. M. O'Hare, A. Pokahr, and A. Ricci, "A survey of programming languages and platforms for multi-agent systems," *Informatica (Slovenia)*, vol. 30, no. 1, pp. 33–44, 2006. 2

[14] A. S. Jensen, "Implementing Lego Agents Using Jason," *Computing Research Repository*, vol. abs/1010.0150, 2010. [Online]. Available: http://arxiv.org/abs/1010.0150 2, 4, 6, 39, 45, 64

[15] A. Mordenti, "Programming Robots with an Agent-Oriented BDI-based Control Architecture: Explorations using the JaCa and Webots Platforms," Bologna, Italy, Tech. Rep., 2012. [Online]. Available: http://amslaurea.unibo.it/4803 5, 41, 45, 80

[16] C. Wei and K. V. Hindriks, "An Agent-Based Cognitive Robot Architecture," in *Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems, ProMAS'12*. M. Dastani, B. Logan & J.F. Huebner, 2012, pp. 55–68. 2, 4, 40, 45, 79

[17] P. Ziafati, M. Dastani, J.-J. Meyer, and L. Van Der Torre, "Agent Programming Languages Requirements for Programming Cognitive Robots," in *Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems, ProMAS'12*. M. Dastani, B. Logan & J.F. Huebner, 2012, pp. 39–54. 2, 6, 42, 80, 81

[18] E. Gat, "On three-layer architectures," in *Artificial Intelligence and Mobile Robots*. MIT Press, 1998. 3, 40, 59

[19] B. Bagnall and J. A. Breña, "LEJOS, Java for LEGO Mindstorms." [Online]. Available: http://lejos.sourceforge.net/ 4, 6, 39, 47, 55, 63

[20] T. M. Behrens, J. Dix, and K. V. Hindriks, "Towards an environment interface standard for agent-oriented programming," Clausthal University of Technology, Tech. Rep., 2009. 5, 40

[21] A. Gerra-Hernández, "Notas del curso Sistemas Multiagentes," Maestría en Inteligencia Artificial, Universidad Veracruzana, Tech. Rep., 2013. 11

[22] Z. Manna and A. Pnueli, *Temporal Verifications of Reactive Systems –Safety*. Springer-Verlag, 1995. 12

[23] S. Russell and P. Norvig, *Artificial intelligence : a modern approach*, 3rd ed. Prentice Hall, 2010. 12

[24] M. Wooldridge, *An introduction to multiagent systems*, 2nd ed. Wiley Publishing, 2009. 12

[25] M. Wooldridge, N. R. Jennings *et al.*, "Intelligent agents: Theory and practice," *Knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995. 12

[26] Y. Shoham, "Agent-oriented programming," *Artificial intelligence*, vol. 60, no. 1, pp. 51–92, 1993. 13

[27] M. E. Bratman, "Intention, plans, and practical reason," 1999. 14

[28] D. C. Dennett, *The international stance*. The MIT press, 1989. 14

[29] J. R. Searle, "Meaning and speech acts," *The Philosophical Review*, vol. 71, no. 4, pp. 423–432, 1962. 14, 21

[30] R. H. Bordini, A. L. C. Bazzan, R. d. O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser, "Agentspeak(xl): efficient intention selection in BDI agents via decision-theoretic task scheduling," in *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multi-agent systems*. New York, NY, USA: ACM, 2002, pp. 1294–1302. 20

[31] A. Ricci, M. Piunti, and M. Viroli, "Environment programming in multi-agent systems: an artifact-based perspective," *Autonomous Agents and Multi-Agent Systems*, vol. 23, no. 2, pp. 158–192, 2011. 21, 26, 27, 28

[32] T. Finin et al., "An overview of KQML: A knowledge query and manipulation language," University of Maryland, CS Department, Tech. Rep., 1992. 21

[33] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl, "Golog: A logic programming language for dynamic domains," *The Journal of Logic Programming*, vol. 31, no. 1, pp. 59–83, 1997. 39

[34] The Lego Group, *LEGO® MINDSTORMS® NXT: Hardware Developer Kit version 1.00*. [Online]. Available: http://www.lego.com/es-ar/mindstorms/downloads/nxt/nxt-hdk/ 47

[35] J. F. Kelly, *LEGO MINDSTORMS NXT-G: Programming Guide*. Technology in Action Press, 2010. 52

[36] M. Gasperi, P. Hurbain, and I. Hurbain, *Extreme NXT: Extending the LEGO MINDSTORMS NXT to the Next Level*. Technology in Action Press, 2007. 53

[37] D. Swan, "RobotC, a C Programming Language for Robotics." [Online]. Available: http://www.robotc.net 54

[38] J. Hansen, *NeXT Byte Codes (NBC) Programmer's Guide*. [Online]. Available: http://bricxcc.sourceforge.net/nbc/ 54

[39] S. Toledo, "Analysis of the NXT Bluetooth-Communication Protocol," 2006. [Online]. Available: http://www.tau.ac.il/~stoledo/lego/btperformance.html 66

[40] X. Limón, A. Guerra-Hernández, N. Cruz-Ramírez, and F. Grimaldo, "An agents and artifacts approach to distributed data mining," in *MICAI 2013: Twelfth Mexican International Conference on Artificial Intelligence*, ser. Lecture Notes in Artificial Intelligence, vol. 8266. Berlin Heidelberg: Springer-Verlag, 2013, pp. 338–349. 74, 75