UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Control de robots humanoides a través de agentes Jason

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Rodríguez González, Darío

Tutores: Julián Inglada, Vicente Javier
Carrascosa Casamayor, Carlos

Director Experimental: Rincón Arango, Jaime Andrés

2016-2017

Resumen

En este trabajo de diseñará e implementará una *API* para *Jason* que proporcione operaciones para el manejo de un robot *NAO*. La necesidad de realizar esta *API* surgió al ver que la compañía que desarrolla este robot sólo proporciona librerías para lenguajes de programación orientados a objetos. Si se quiere dotar a este robot con un comportamiento inteligente, lo más sensato sería programarlo con un lenguaje de programación orientado a agentes inteligentes. Además del diseño e implementación de esta *API*, en este proyecto se realizarán unos ejemplos para validar la misma.

Palabras clave: API, robot NAO, agentes inteligentes, Jason, CArtAgO.

Abstract

In this work, we are goint to design and implement an API for Jason that provides operations for the management of NAO robot. The need of this API appear when the company that develops this robot only provides libraries for object-oriented programming languages. If you want to equip this robot with intelligent behavior, the most intelligent thing would be to program it with a agent-oriented programming language. In addition to the design and implementation of this API, in this project there will be some examples to validate the API.

Keywords : API, NAO robot, intelligent agent, Jason, CArtAgO.

Agradecimientos

En primer lugar, me gustaría agradecer a los tutores por darme la posibilidad de trabajar con el robot NAO.

También quería agradecer a Marisa todo el apoyo y el amor que me ha proporcionado en estos años de carrera. Sin él, probablemente no hubiera llegado hasta el final y hubiera abandonado hace tiempo.

Por último, me gustaría agradecer a Dario Jr. la felicidad que me transmite todos los días, desde que se despierta hasta que se acuesta.

Además de estos agradecimientos, me gustaría dedicar a todas esas personas que están estudiando, trabajando y además son padres de un niño menor de 2 años. Aunque todo se te haga cuesta arriba, si te lo propones todo se puede conseguir.

Tabla de contenidos

Tabla de ilustraciones

1. Introducción

En la actualidad, tecnologías como la robótica o la inteligencia artificial están teniendo gran impacto en la sociedad. Cada vez se ven más implementaciones de robots que hacen tareas como las haría una persona, como el robot *Pepper,* el cual puede ejercer las labores de recepcionista (entre muchas otras). También están acaparando mucha atención máquinas que antes eran manejadas por personas y ahora se mueven de forma autónoma gracias a una inteligencia artificial como, por ejemplo, coches autónomos o robots que manejan la mercancía de un almacén de forma completamente autónoma.

Éste trabajo se centrará en los robots humanoides, en concreto en el robot *NAO* de la empresa *Aldebaran Robotics*, y en la forma de programarlos para darle ese comportamiento inteligente.

Este robot en concreto ofrece librerías para programarlo en los siguientes lenguajes: C++, Python y Java. Todos ellos son lenguajes orientados a objetos y, por tanto, para poder dotar al robot con un comportamiento inteligente hay que seguir este paradigma. Utilizar este paradigma en pequeños proyectos no es demasiado complicado, pero a medida que se quiere introducir complejidad en el comportamiento inteligente del robot, incrementa la dificultad del código.

Por otra parte, existen lenguajes orientados a agentes inteligentes como, por ejemplo, *Jason*. Estos lenguajes proporcionan mecanismos para programar estos agentes de forma que actúen siguiendo un comportamiento inteligente.

La motivación de este proyecto es poder utilizar uno de estos lenguajes orientado a agentes para programar el comportamiento del robot NAO. Para ello, se diseñará un interfaz de programación de aplicaciones[1] (*API*) que proporcionará funciones para el manejo del robot *NAO*. Esta *API* podrá ser llamada desde un programa Jason.

Con la finalidad de detallar todo el proceso necesario para este fin, este trabajo tendrá la siguiente estructura:

- En primer lugar, se hará una introducción de las tecnologías que serán utilizadas a lo largo del desarrollo de la *API*.
- En segundo lugar, se detallará todo el desarrollo e implementación de la *API*.
- Por último, se diseñarán unos programas en Jason con el fin de validar el funcionamiento de la *API*.

[1] En el resto del documento se nombrará como *API* los interfaces de programación de aplicaciones por sus siglas en inglés (*Application Programming Interface*).

2. Objetivos

El objetivo principal de este trabajo es el diseño e implementación de una *API* para *Jason,* que permita controlar el comportamiento del robot *NAO*. Esta *API* también debe proporcionar mecanismos para percibir el entorno mediante los sensores del robot. Para ello se deben emplear artefactos *CArtAgO* [1].

Para poder llevar a cabo este desarrollo se deberán de completar unos subobjetivos, los cuales se detallarán a continuación.

2.1 Subobjetivos

- Estudio del robot *NAO*, desde la especificación técnica hasta las librerías actuales para programarlo.
- Estudio del *framework* CArtAgO y su uso junto al lenguaje *Jason*.
- Diseño e implementación de la *API* de operaciones para el lenguaje de programación *Jason*, la cual deberá de proporcionar operaciones para el manejo del movimiento del robot, operaciones básicas para procesar y manipular la entrada de video del robot, operaciones básicas para ordenarle al robot que hable y que reconozca algunas palabras, operaciones para el manejo de los leds, operaciones para el seguimiento de personas y mecanismos para obtener la información que proporcionan los sensores.
- Validar la *API* desarrollada mediante la generación de ejemplos reales sobre el robot, que permitan evaluar el uso de las diferentes funcionalidades que soporta el *API*.

3. Estado del arte

3.1 Robots humanoides

Un robot es una máquina controlada por ordenador y programada para moverse, manipular objetos y realizar trabajos a la vez que interacciona con su entorno. El robot a veces recuerda a los seres humanos, y es capaz de efectuar diversas tareas humanas complejas cuando se les indica que lo hagan, o por habérselas programado con antelación. [2]

Esta definición de robot es bastante amplia, por ello, existen una gran cantidad de máquinas que la cumplen. Para poder concretar más aún en la definición de un robot se pueden definir diferentes tipos de robots, según el entorno o la finalidad para la que fueron construidos. Una posible clasificación de los tipos de robots podría ser: robots industriales, robots domésticos o del hogar, robots médicos, robots militares, robots de entretenimiento, robots espaciales, robots educacionales y robots humanoides.

En el presente trabajo se va a tratar únicamente con robots humanoides, concretamente con el robot *NAO*. Estos robots tienen la forma del cuerpo construida para parecerse al cuerpo humano. El diseño puede ser para fines funcionales, tales como interactuar con herramientas y ambientes humanos, con fines experimentales, tales como el estudio de la locomoción o para otros fines. En general, los robots humanoides tienen un torso, una cabeza, dos brazos y dos piernas, aunque algunas formas de robots humanoides pueden modelar sólo una parte del cuerpo, por ejemplo, desde la cintura para arriba. Algunos robots humanoides también tienen cabezas diseñadas para replicar rasgos faciales humanos como ojos y bocas. [3]

Antes de detallar la estructura y funcionalidades del robot *NAO*, se expondrá brevemente la historia de los robots humanoides, para así conocer algunos de los predecesores del *NAO* y la evolución que sufrió este sector en las últimas décadas.

3.1.1 Breve repaso de la historia de los robots humanoides

Pese a que existen máquinas que podrían entrar en la definición de robot humanoide desde hace varios siglos como, por ejemplo, el robot de Leonardo [4], vamos a establecer el punto de partida en el primer robot móvil bípedo de la historia: el E0 de Honda, y a partir de él se expondrán solo los robots que el autor cree más relevantes. El artículo de la *Wikipedia* en inglés sobre robots humanoides ofrece una lista más detallada [3].

Desde el año 1986 hasta el año 1993, la compañía *Honda* desarrolló en secreto unos robots bípedos los cuales solo se componían por un par de piernas. Sus nombres fueron *E0*, *E1*, *E2*, *E3* y *E4*. Pese a ser sólo un par de piernas, estos robots marcaron un hito en la historia de la robótica, ya que eran capaces de sostenerse en pie sin perder el equilibrio y desplazarse a un ritmo constante. Todos estos robots poseían prácticamente las mismas habilidades, la única diferencia era la velocidad a la que podían andar, que evolucionó de un paso cada cinco segundos a 4,7 kilómetros hora.

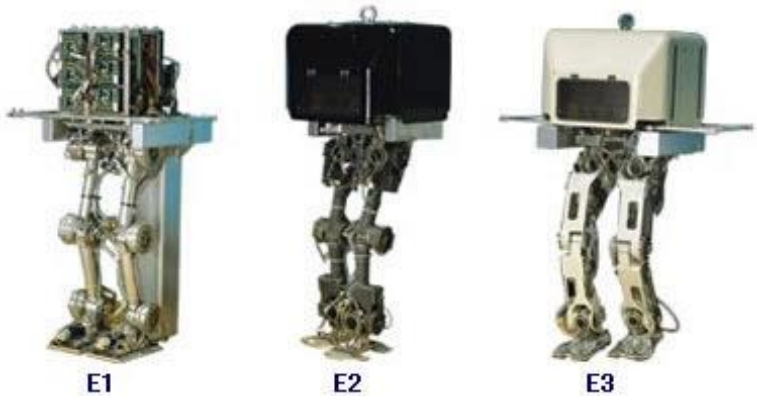


Ilustración 1 Robots E1, E2 y E3

En el año 1993 *Honda* desarrolla también los modelos *E5* y *E6*. El modelo *E5* alcanzó el hito de ser el primer robot humanoide autónomo y, unos meses después, el modelo *E6* consigue subir escaleras y superar algunos obstáculos.

Ilustración 2 Robots E4, E5 y E6

En 1993 *Honda* empezó a crear otro robot completamente distinto, al que además de piernas dotó de brazos, entre otras cosas, y lo llamó P1. El desarrollo duró de 1993 a 1997. El robot P1, con 1,91 metros de altura y 175 kilogramos de peso, es capaz de coger objetos, abrir algunas puertas y desplazarse como sus predecesores por superficies planas. Al robot *P1* le siguieron el *P2* y *P3*, los cuales poseían prácticamente las mismas cualidades que el robot *P1* pero con unos diseños más novedosos.

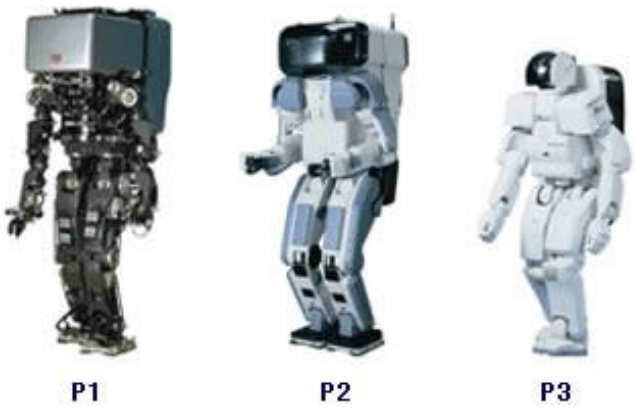


Ilustración 3 Robots P1, P2 y P3

Aunque por estos años el ritmo de los robots humanoides lo marcaba *Honda*, la empresa *AIST* también lanzó su modelo en 1998, el *HRP 1-S*. Este robot era capaz de caminar y utilizar herramientas.

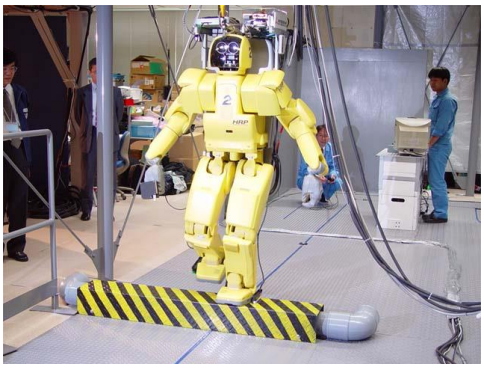


Ilustración 4 Robot HRP 1-S

En el año 2000, *Honda* presenta el que quizás sea el robot humanoide más famoso, el robot *ASIMO*. El robot se construye más pequeño que sus predecesores (1,20 metros de altura y 45 kilogramos de peso). Para darle una apariencia más amigable, goza de un diseño muy agradable y es capaz de subir y bajar escaleras.

Ilustración 5 Robot ASIMO

Después del *ASIMO* fueron muchas las empresas que se interesaron en este tipo de robots. A continuación, se nombran algunas de las implementaciones entre el año 2000 y el 2010: *HRP 2P* de *AIST*, *SDR-4X* de *Sony*, *HRP 2* de *AIST*, *Qrio* de *Sony*, *NAO* de *Aldebaran Robotics*. En 2011 *Honda* saca una nueva versión del *ASIMO*, mucho más ágil que el presentado en 2000, capaz de correr, saltar o abrir frascos y sostener y llenar un vaso de agua.

En 2011, la empresa *Boston Dynamics* desarrolla un robot que alcanza un hito no conseguido por otras empresas hasta la fecha. Esta empresa desarrolló el primer robot bípedo, al que denominaron *PETMAN*, capaz de soportar empujones mientras camina sin perder el equilibrio, o de realizar movimientos gimnásticos.

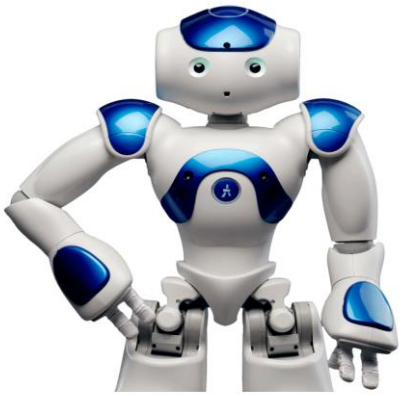
Ilustración 6 Robot PETMAN

En la actualidad, es la empresa *Boston Dynamics* la que lidera los avances más significativos en este sector. El robot humanoide más avanzado que poseen es el *Atlas*. *Atlas* y sus últimas actualizaciones son capaces de caminar por todo tipo de terrenos complejos, como ruinas o incluso un bosque nevado; pueden sortear toda clase de obstáculos, como piedras, barreras o agujeros, ayudándose con los brazos si es necesario; soportan fuertes impactos mientras mantienen el equilibrio sobre una pierna e incluso colocan cajas en estanterías con soltura.

Ilustración 7 Robot Atlas

3.1.2 Robot NAO

NAO es un robot humanoide programable y autónomo, desarrollado por *Aldebaran Robotics*. *NAO* posee tecnología puntera, programable y controlable usando *Linux*, *Windows* y *Mac*, proporcionando una interfaz de comunicación muy flexible. Permite realizar movimientos precisos y coordinados. Además, lleva incorporadas una serie de funciones de alto nivel para facilitar su uso.



Los principales usos que se le dan a este robot son académicos. Además, este robot fue elegido para una categoría de la competición *RoboCup*. Últimamente cobra importancia por su capacidad de estimular a niños autistas, entre otros.

A continuación, se expondrán sus especificaciones técnicas extraídas de la documentación oficial del robot [5], el sistema operativo del *NAO* (*NAOqi*) y una herramienta desarrollada por *Aldebaran*, llamada *Choregraphe*, que nos permite interaccionar con el robot.

3.1.2.1 Especificaciones técnicas del robot NAO

Este robot mide 574 milímetros de alto, 275 milímetros de ancho y 311 milímetros de largo, tal y como se muestra en la Ilustración 8. Pesa 5.4 kilogramos y está construido con el material ABS-PC/PA-66/XCF-30.

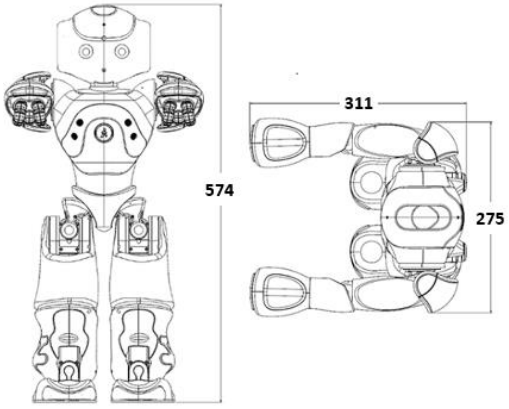


Ilustración 8 Dimensiones del robot NAO

La versión del NAO que se empleará en la práctica incorpora su propia CPU, la cual es ATOM Z530 1.6 GHz, con 1 GB de RAM, 2 GB *Flash memory* y 8 GB *Micro SDHC*. Usa una batería de litio de Un = 21,6V, con una autonomía aproximada de 60 min.

También incorpora 2 altavoces de 38 milímetros de diámetro situados en las orejas, 4 micrófonos situados en la cabeza y 2 cámaras VGA 1280x960, 30 *fps*. Respecto a los sensores, tiene cuatro de ultrasonidos situados en el pecho (dos emisores y dos receptores), 4 sensores de fuerza en cada pie, posee tres sensores de tacto en la cabeza, tres en cada mano y uno en la parte frontal de cada pie, sensores inerciales (acelerómetro y giróscopo de 3 ejes), y en cada articulación sensores de posición con una resolución de 0, 1º. Para hacerlo más vistoso, incorpora LEDs de colores en los ojos, orejas, pecho y pies. La conexión puede hacerse vía Wi-Fi IEE 802.11n (inalámbrica) o Ethernet (con cable).

Para el movimiento proporciona las articulaciones definidas en la siguiente ilustración con sus respectivos motores.

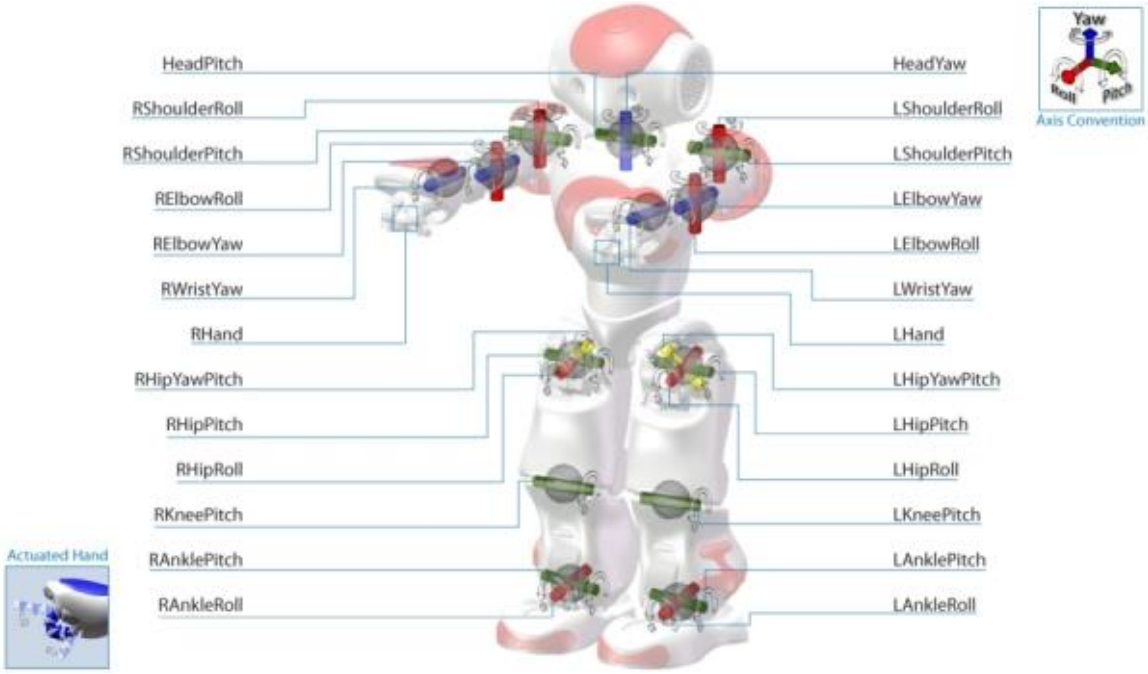


Ilustración 9 Articulaciones del NAO

3.1.2.2 NAOQi [6]

NAOQi es el nombre que le da *Aldebaran* tanto al sistema operativo del *NAO*, que está basado en Linux, como a los *frameworks* de desarrollo que dan acceso al control del robot. Este *framework* se divide en siete áreas, que están claramente diferenciadas por su funcionalidad. Estas áreas son:

- ***NAOqi Core*:** Contiene un conjunto de módulos relacionados con la memoria, el sistema y comportamientos genéricos del robot.
- ***NAOqi Motion*:** Contiene un conjunto de módulos relacionados con el movimiento del robot. Proporciona operaciones para andar, para mover las articulaciones y para establecer una postura entre otras.
- ***NAOqi Audio*:** Contiene un conjunto de módulos relacionados con el habla del robot, el reconocimiento de voz, la detección de sonidos y la reproducción de sonidos.
- ***NAOqi Vision*:** Contiene un conjunto de módulos que permiten obtener imagen desde las cámaras, detectar códigos QR, detectar una bola roja, etc.
- ***NAOqi PeoplePerception*:** Contiene un conjunto de módulos relacionados con la percepción de personas y sus características.
- ***NAOqi Sensors:*** Contiene un conjunto de módulos que nos proporcionan la información de los sensores.
- ***NAOqi Trackers:*** Contiene un conjunto de módulos que nos permiten, mediante el reconocimiento de formas, seguir unos marcadores.

Este *framework* está disponible en los siguientes lenguajes *Python*, C++, *Java* y *javascript*. Los programas en *Python* y C++ pueden ser cargados directamente en el robot y ejecutados desde el, mientras que *Java* y *javascript* sólo se puede ejecutar en un

ordenador y comunicarse vía red con el robot (Python y C++ también permiten esta modalidad de ejecución).

3.1.2.3 Choregraphe [7]

Además de los lenguajes citados anteriormente, el robot NAO puede ser programado mediante una herramienta proporcionada por *Aldebaran*, que se llama *Choregraphe*. Esta herramienta permite escribir programas que definirán un comportamiento del robot mediante un entorno visual y sencillo.

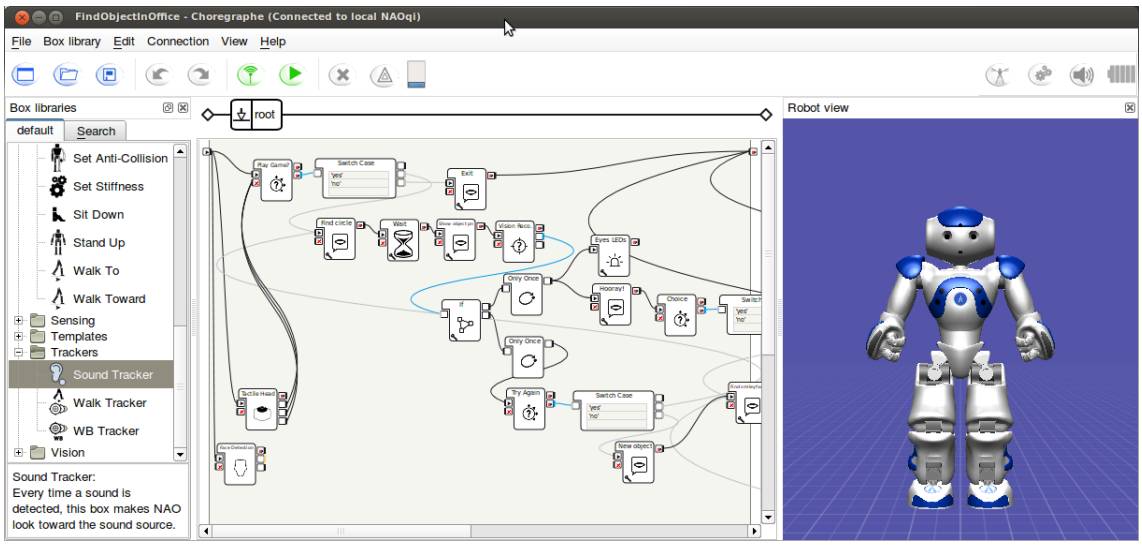


Ilustración 10 Choregraphe

Choregraphe, también contiene un robot simulado, con el que podemos trabajar como si se tratase de un robot real, pero con ciertas limitaciones. Las limitaciones vienen dadas por el hecho de que esta simulación no implementa los sensores, altavoces, etc. Su principal uso es para probar los movimientos del robot.

Otra herramienta que incorpora *Choregraphe* es el timeline, que nos permite definir o grabar directamente desde el robot, un conjunto de movimientos complejos definidos en una línea de tiempo. Esta herramienta simplifica mucho el trabajo a la hora de establecer el ángulo de las articulaciones y los tiempos de transición entre ángulos, ya que una vez definido el *timeline* nos permite exportar a código C++ o *Python*.

A continuación, se muestra una imagen de esta herramienta:

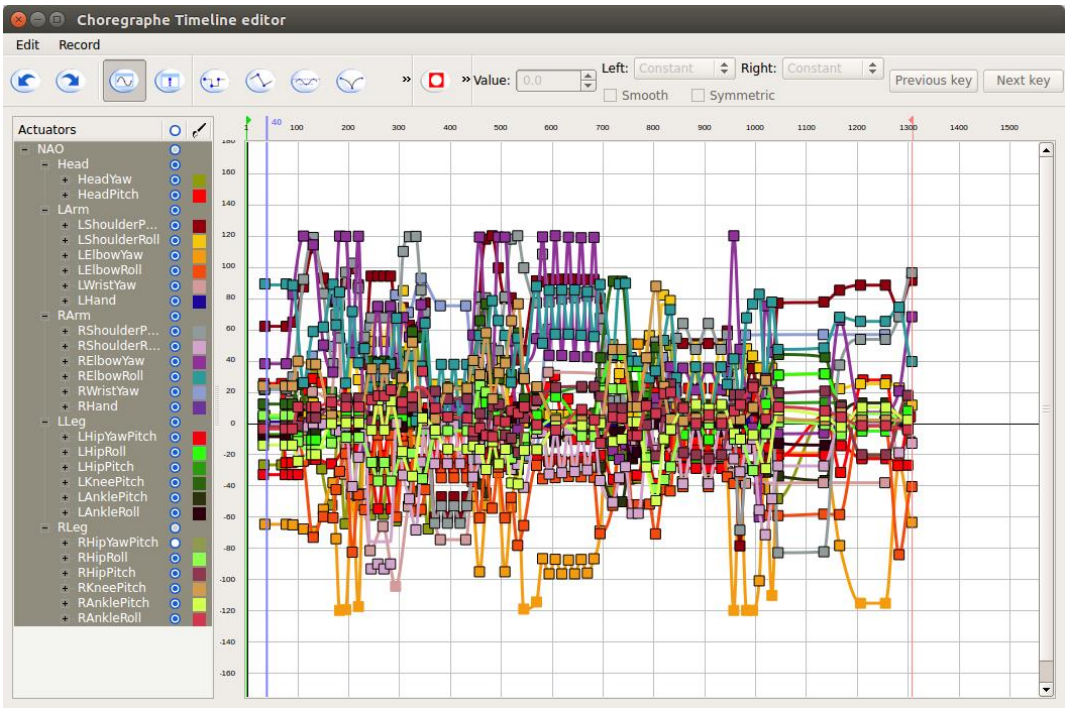


Ilustración 11 Choregraphe timeline

3.2 Agentes BDI

El modelo de software creencia-deseo-intención (generalmente referido simplemente como BDI por sus siglas en inglés) es un modelo de software desarrollado para programar agentes inteligentes. Este modelo está caracterizado por la implementación de las creencias, deseos e intenciones de un agente. Utiliza estos conceptos para resolver un problema particular en la programación orientada a agentes. En esencia, proporciona un mecanismo para separar la actividad de seleccionar un plan (de una biblioteca de planes o de planificador externo) de la ejecución de planes actualmente activos. En consecuencia, los agentes de BDI son capaces de equilibrar el tiempo dedicado a deliberar sobre los planes (elegir qué hacer) y ejecutar esos planes (hacerlo). Una tercera actividad, la creación de los planes en con mayor prioridad (planificación), no está dentro del alcance del modelo, y se deja al diseñador del sistema y programador.

A continuación, se describirán los componentes de un agente BDI:

- **Creencias**: Las creencias representan el estado informativo del agente, es decir, sus creencias sobre el mundo (incluyendo a sí mismo y otros agentes). Las creencias también pueden incluir reglas de inferencia, permitiendo encadenamiento hacia adelante para conducir a nuevas creencias. Usar el término creencia en vez de conocimiento reconoce que lo que un agente cree puede no necesariamente ser verdad (y de hecho puede cambiar en el futuro).
    - **Conjunto de creencias**: Las creencias se almacenan en la base de datos (a veces llamada una base de creencias o un conjunto de creencias), aunque esa es una decisión de implementación.
- **Deseos**: Los deseos representan el estado motivacional del agente. Representan objetivos o situaciones que el agente desearía lograr o lograr. Ejemplos de deseos pueden ser: encontrar el mejor precio, ir a la fiesta o hacerse rico.
    - **Objetivos**: Un objetivo es un deseo que se ha adoptado para la búsqueda activa por el agente. El uso del término objetivos agrega una restricción adicional. Esta es que el sistema de deseos activos debe ser constante. Por ejemplo, uno no debe tener metas simultáneas para ir a una fiesta y quedarse en casa - a pesar de que ambos podrían ser deseables.
- **Intenciones**: Las intenciones representan el estado deliberativo del agente - lo que el agente ha elegido hacer. Las intenciones son los deseos que el agente ha decidido realizar. En los sistemas implementados, esto significa que el agente ha comenzado a ejecutar un plan.
    - **Planes**: Los planes son secuencias de acciones (recetas o áreas de conocimiento) que un agente puede realizar para lograr una o más de sus intenciones. Los planes pueden incluir otros planes: mi plan para ir a dar una vuelta en coche puede incluir un plan para encontrar las llaves de mi coche.
- **Eventos**: Son los desencadenantes de la actividad reactiva del agente. Un evento puede actualizar creencias, activar planes o modificar objetivos. Los eventos pueden ser generados externamente y recibidos por sensores o

sistemas integrados. Además, los eventos pueden generarse internamente para activar actualizaciones desacopladas o planes de actividad.

3.2.1 Jason [8]

El lenguaje de programación *Jason* está inspirado en el modelo BDI. Este lenguaje de programación está basado en *AgentSpeak*. Para explicar este lenguaje, vamos a partir del siguiente ejemplo:

```
feliz(dario).

!decir(hello).

+!decir(X) : feliz(dario) <- .print(X).
```

Como podemos ver, la sintaxis muy diferente a la de programas C, *Java* o *Python*. La sintaxis está inspirada en *Prolog*, pero el objetivo es diferente (la salida no es conocimiento y el motor subyacente no se basa en resolución). Este programa se interpreta de la siguiente manera:

1. El agente tiene una creencia inicial: feliz(dario), incluida por el programador (en lugar de percibirla del entorno). Esta creencia se puede leer como "dario tiene la propiedad (o predicado) feliz".
2. El agente tiene un deseo (inicial): !decir(hola), también incluido por el programador. Lo que sigue al símbolo ! describe el deseo.
3. El agente tiene un plan para lograr el deseo decir(hola). Podemos leer este plan como "siempre que el agente tenga el deseo de decir (X) y crea que feliz(dario), ejecutando la acción .print (X) se logra el completar el deseo, para cualquier X (que es una variable ya que Comienza con una letra mayúscula, como en Prolog) ".

A su vez, este programa es interpretado en Jason de la siguiente manera:

1. La creencia inicial se agrega en la base de creencias del agente.
2. Para el deseo inicial, el evento +!decir(hola) se agrega en la cola de eventos que debe manejar el agente.
3. El plan se incluye en la biblioteca de planes del agente.
4. Se ejecuta un ciclo de razonamiento:
    - El evento +!decir (hola) se selecciona de la cola.
    - Se selecciona el plan anterior (coincide con el evento seleccionado cuando la variable X tiene el valor hola).
    - El agente cree feliz(dario).
    - Se crea una nueva intención basada en este plan y el valor de X. El agente se ha comprometido con el deseo de decir (hola).
    - Se ejecuta una acción de la intención (el comando .print (hola) en este caso).
    - Puesto que la intención ha ejecutado todas las acciones, termina.
5. El agente espera a que reaccionen los nuevos eventos.

3.3 CArtAgO

CArtAgO[2] es una infraestructura/marco de propósito general que permite programar y ejecutar entornos virtuales (también denominados entornos virtuales/aplicaciones/software) para sistemas multi-agentes.

CArtAgO se basa en el metamodelo *Agents & Artifacts* (A & A) para modelar y diseñar sistemas multi-agentes. A & A introduce metáforas de alto nivel tomadas de entornos laborales cooperativos humanos: agentes como entidades computacionales que realizan algún tipo de tarea/actividad orientada a objetivos (en analogía con trabajadores humanos) y artefactos como recursos y herramientas dinámicamente construidos, usados y manipulados por agentes para apoyar/realizar sus actividades individuales y colectivas (como artefactos en contextos humanos). En realidad, A & A se basa en estudios interdisciplinarios que involucran la Teoría de la Actividad y la Cognición Distribuida como principales marcos conceptuales de fondo.

CArtAgO permite desarrollar y ejecutar entornos basados en artefactos, estructurados en espacios de trabajo abiertos (posiblemente distribuidos a través de la red) que pueden unirse a agentes de diferentes plataformas para trabajar juntos dentro de tales entornos. Así, con *CArtAgO*, los desarrolladores de sistemas multi-agentes tienen finalmente un modelo de programación simple para diseñar y programar el entorno computacional del agente, compuesto por conjuntos dinámicos de artefactos de diferentes tipos, aparte de los modelos y plataformas utilizados para programar agentes.

CArtAgO no está vinculado a ningún modelo o plataforma de agente específico: se pretende que sea ortogonal con respecto al modelo o plataforma de agente específico adoptado para definir la arquitectura y el comportamiento del agente. Sin embargo, *CArtAgO* es especialmente útil y eficaz cuando se integra con los lenguajes de programación de agentes basados en una fuerte noción de agencia - agentes inteligentes - en particular aquellos basados en la arquitectura BDI. La última distribución de *CArtAgO* (2.0) incluye directamente un puente para el lenguaje de programación Jason.

3.3.1 Artefactos CArtAgO

La función de los artefactos *CArtAgO* es la de proporcionar a los agentes un conjunto de operaciones que puede realizar con ellos. Además, en la integración con Jason, permite el mapeo de las propiedades observables del artefacto en creencias. Para ello, el agente Jason debe realizar primero la operación "*focus*" sobre un artefacto. De esta manera, cada vez que una propiedad observable cambie su valor, también lo hará la creencia asociada.

A continuación, se mostrará un modelo que ayudará a entender mejor las funciones de un artefacto:

[2] CArtAgO toma su nombre por las siglas en ingles de: *Common ARTifact infrastructure for AGents Open environments*.

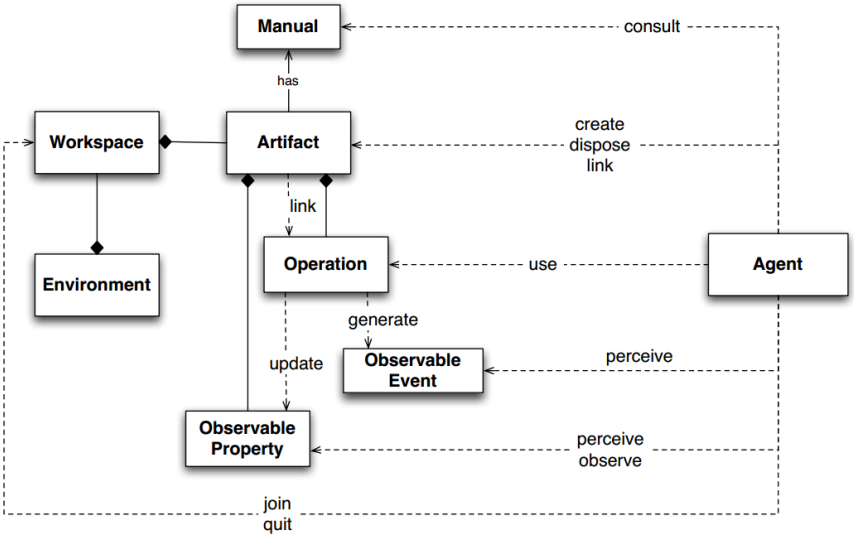


Ilustración 12 A & A metamodelo

En el presente trabajo se empleará siempre el *Workspace* por defecto, por tanto, se puede obviar esta parte del esquema anterior.

4. Diseño de la API

Tal y como se definió en el apartado 2, el objetivo de este trabajo es diseñar e implementar una API para el lenguaje de programación *Jason*. Por tanto, en ese apartado se desarrollarán esos dos puntos.

4.1 Descripción y diseño de la API

Antes de empezar con el diseño de la API, hace falta tener una visión global de lo que se desea diseñar, con lo cual, vamos a describir con un poco más de detalle el problema y así poder realizar un mejor diseño.

Para poder entender mejor el problema, nos apoyaremos en el siguiente diagrama de casos de uso:

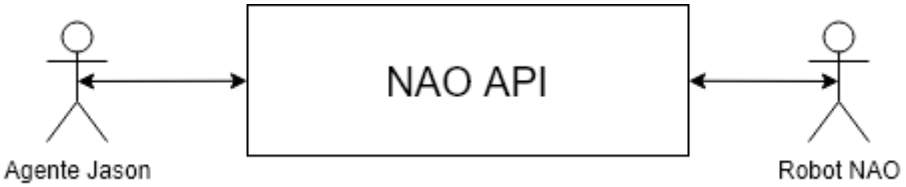


Ilustración 13 Diagrama de casos de uso inicial

Como se puede apreciar en el diagrama, lo que necesitamos hacer es un componente que llamaremos *NAO API*, al que se le puedan enviar ordenes desde un agente (o programa) Jason, y este pueda comunicar esas órdenes a un robot *NAO*. Para este fin y según lo visto en apartados anteriores, podemos utilizar los artefactos *CArtAgO*, los cuales nos proporcionarán operaciones para extender las funcionalidades de los agentes Jason. A su vez, podemos usar una librería *NAOQi* para programar el comportamiento del robot. La librería elegida será la librería *Java*. Esta librería envía las ordenes al robot a través de *proxys tcp*.

Según estas consideraciones, podemos concretar un poco más el diagrama de uso anterior, que quedaría de la siguiente forma:

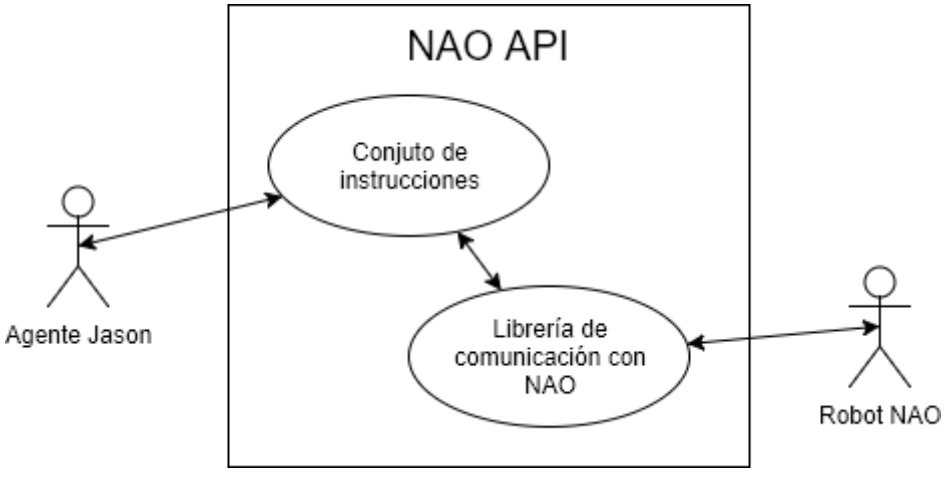


Ilustración 14 Diagrama de casos de uso

Del diagrama anterior, podemos concluir que lo que haría falta es utilizar la librería actual del NAO, proporcionada por la empresa que comercializa el robot, y añadirle unos componentes que nos permitan hacer una llamada desde *Jason* a esa librería. Por lo tanto, se utilizará una arquitectura en capas que permitan transformar una orden *Jason* a una llamada a la librería *NAO*. Las capas necesarias serían las siguientes: un puente entre *Jason* y *CArtAgO*, una librería de operaciones basada en artefactos *CArtAgO* y un interfaz de conexión entre los artefactos *CArtAgO* y la librería *NAO*.

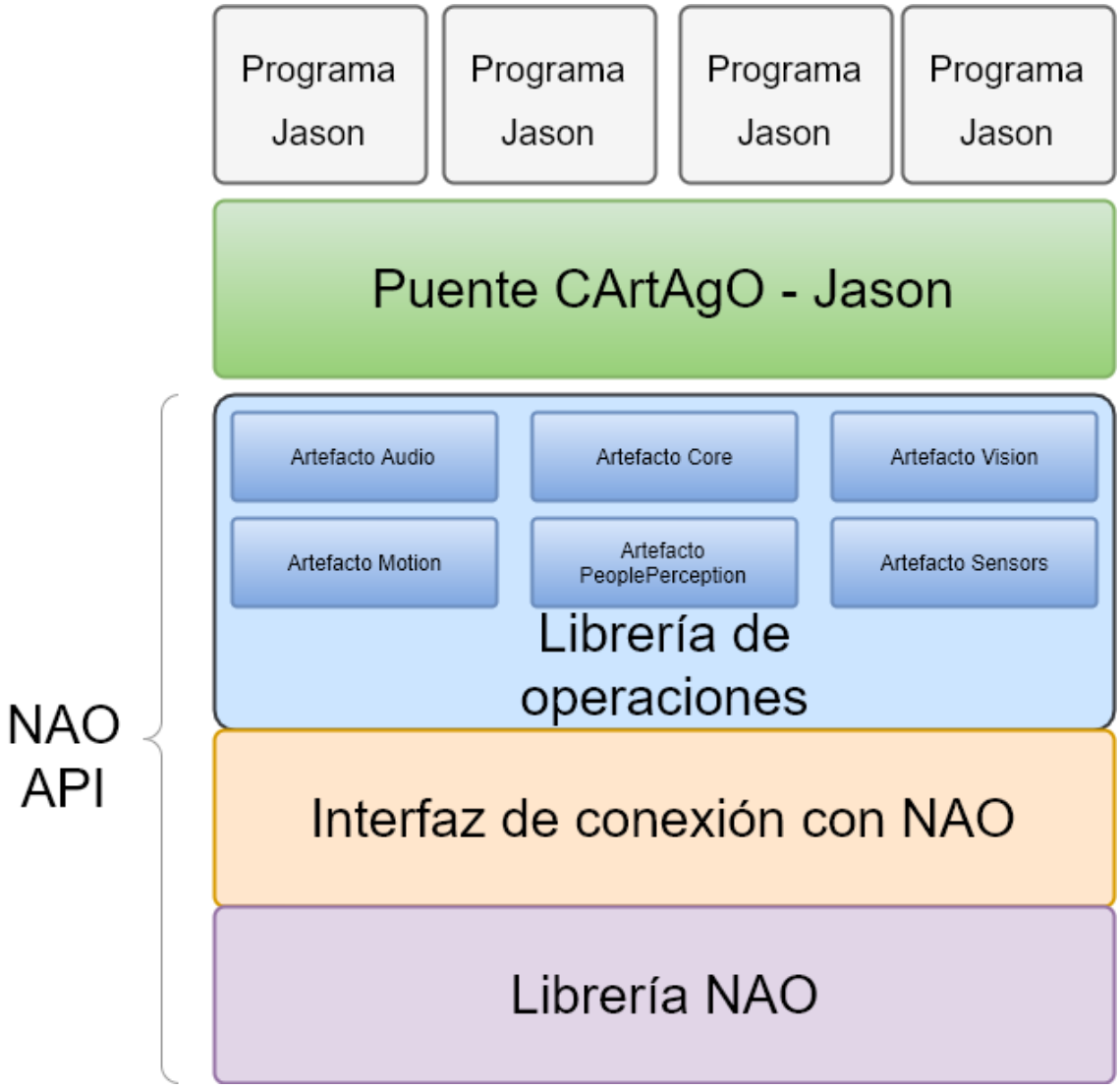


Ilustración 15 Conexión entre Jason y la librería NAO

4.1.1 Descripción de las capas de la librería

A continuación, se describirán las capas definidas anteriormente, las cuales se pueden ver en la Ilustración 15.

4.1.1.1 Capa librería NAO

En esta capa se encuentra la librería de programación del *NAO*. En este proyecto en particular, la definición dice que se debe utilizar la librería *NAOqi* para *Java*. Esta librería está basada en *proxys tcp* y nos permite enviar ordenes al robot a través de una conexión de red. Gracias a esto, no es necesario cargar el programa en el robot para ejecutarlo, por tanto, esto nos permite controlar tanto un robot real como un robot simulado siempre que este permita conexiones *tcp*.

Esta librería, tal y como se vio el apartado 3.1.2.2, está estructurada en módulos. En ese mismo apartado se ofrece una descripción general de cada módulo.

A continuación, se enumerarán las clases java que proporciona cada módulo, que poseen los métodos encargados de enviar las ordenes al robot:

- ***NAOqi Core*:**
  - *ALAutonomousLife*: Mantiene el ciclo de vida del robot y gestiona el lanzamiento de las actividades.
  - *ALBehaviorManager*: Inicia y detiene comportamientos.
  - *ALConnectionManager*: Administra la conexión a una red y su configuración.
  - *ALMemory:* Obtiene e inserta datos para que otros módulos los puedan usar.
  - *ALModule*: Permite crear tus propios módulos.
  - *ALNotificationManager*: Administra las notificaciones
  - *ALPreferenceManager*: Lee y guarda las configuraciones de robot
  - *ALResourceManager*: Manejo de los recursos.
  - *ALStore*: Recupera aplicaciones de la tienda *Apps 2.1*
  - *ALSystem*: Administra el sistema del robot.
  - *ALUserSession*: Administra el estado de los usuarios activos y los enlaces a sus datos
  - *ALTabletService*: Carga la aplicación web, reproduce vídeos y administra la propia tableta.
  - *ALWorldRepresentation*: Almacena datos a largo plazo sobre objetos detectados en una base de datos espacialmente estructurada.
  - *PackageManager*: Administra paquetes: instalación, desinstalación.
- ***NAOqi Motion*:**
  - *ALAutonomousMoves*: Gestiona los movimientos autónomos.
  - *ALRobotPosture*: Hace que el robot vaya a una postura predefinida.
  - *ALNavigation*: Hace que el robot se mueva con seguridad, deteniéndose si se detecta algún obstáculo.
  - *ALRecharge*: Hace que el robot se mueva a su estación de carga.
  - *ALMotion*: Es la herramienta principal que permite al robot moverse. Los desarrolladores y animadores avanzados pueden aprovechar sus numerosos métodos para desarrollar su propio código de movimiento.

- ***NAOqi Audio*:**
  - *ALAudioDevice*: Administra las entradas y salidas de audio. Este módulo es utilizado por todos los demás módulos de audio excepto *ALAudioPlayer*.
  - *ALAudioPlayer*: Reproduce archivos de audio en el robot.
  - *ALAudioRecorder*: Graba archivos de audio en el robot.
  - *ALSoundDetection*: Detecta eventos de sonido.
  - *ALSoundLocalization*: Localiza los sonidos detectados por el módulo *ALSoundDetection*.
  - *ALSpeechRecognition*: Consigue que el robot entienda lo que dice un humano.
  - *ALTextToSpeech*: Consigue que el robot hable.
  - *ALAnimatedSpeech*: Combina el habla y los gestos.
  - *ALDialog*: Crea una base de conocimientos básicos para las habilidades de conversación.
  - *ALVoiceEmotionAnalysis*: Identifica la emoción expresada por la voz del hablante.
- ***NAOqi Vision*:**
  - *ALPhotoCapture*: Toma fotos y las guarda en el disco.
  - *ALVideoDevice*: Administra las entradas de vídeo.
  - *ALVideoRecorder*: Graba video.
  - *ALBacklightingDetection*: Comprueba si la imagen de la cámara está retroiluminada.
  - *ALBarcodeReader*: Detecta y lee un código de barras en una imagen.
  - *ALColorBlobDetection*: Detecta gotas (circulares o no) de un color determinado.
  - *ALDarknessDetection*: Comprueba si el ambiente está oscuro.
  - *ALLandMarkDetection*: Detecta puntos de referencia visuales específicos.
  - *ALMovementDetection*: Detecta un poco de movimiento y dice de dónde viene.
  - *ALRedBallDetection*: Detecta objetos rojos y circulares.
  - *ALVisionRecognition*: Hace que el robot aprenda y reconozca patrones visuales: objetos, imágenes ...
  - *ALVisualSpaceHistory*: Construye un mapa con fecha y hora de las posiciones de la cabeza.
  - *ALVisualCompass*: Utiliza una imagen como una brújula.
  - *ALCloseObjectDetection*: Detecta objetos que están cerca para ser detectados directamente por el sensor 3D.
  - *ALSegmentation3D*: Segmenta las imágenes de profundidad devueltas por el sensor 3D.
- ***NAOqi PeoplePerception*:**
  - *ALBasicAwareness*: Hace que el robot reaccione ante los estímulos de su entorno.
  - *ALEngagementZones*: Analiza la posición de una persona con respecto al robot.
  - *ALFaceCharacteristics*: Proporciona características humanas estimadas basadas en el análisis facial.

- *ALFaceDetection*: Hace que el robot detecte y reconozca las caras humanas.
- *ALGazeAnalysis*: Analiza la dirección de la mirada de una persona.
- *ALPeoplePerception*: Consigue que el robot guarde pistas sobre la gente que la rodea.
- *ALSittingPeopleDetection*: Consigue que el robot detecte si una persona está sentada o no.
- *ALWavingDetection*: Hace que el robot detecte si una persona se está agitando.

- ***NAOqi Sensors:***
  - *ALSensor*: Es responsable de generar los eventos correspondientes a los sensores del robot.
  - *ALBattery*: Es responsable de generar eventos relacionados con el *hardware* de la batería del robot.
  - *ALBodyTemperature*: Es responsable de generar eventos cuando alguna parte del hardware, capaz de sobrecalentarse, ha alcanzado un nivel relativo de temperatura.
  - *ALChestButton*: Es responsable de generar los eventos relacionados con el botón del pecho del robot.
  - *ALFsr*: Genera el evento *footContactChanged()* cuando cambia el contacto del pie.
  - *ALTouch*: Genera el evento *TouchChanged()* cuando se toca el robot.
  - *ALInfrared*: Permite la comunicación IR (Infra-Red) con el robot. Tiene 3 propósitos diferentes:
    - Utilizar el robot como control remoto,
    - Configura el robot para recibir órdenes de un mando a distancia,
    - Hacer que varios robots se comuniquen juntos (no se recomienda).
  - *ALLaser*: Recupera los datos enviados por la cabeza láser (dispositivo opcional) y los almacena en una clave *ALMemory* llamada: *Device/Laser/Value*.
  - *ALsonar*: Recupera el valor del sensor ultrasónico de ALMemory, lo procesa y genera eventos según la situación.
  - *ALLeds*: Permite controlar los LEDs del robot.
- ***NAOqi Trackers:***
  - *ALTracker*: Rastreador genérico que permite al robot rastrear diferentes objetivos (bola roja, cara, hito, etc) usando diferentes medios (cabeza solamente, cuerpo entero, movimiento, etc).

4.1.1.2 Capa interfaz de conexión con NAO

El objetivo de esta capa es poder proporcionar a las conexiones tcp con el NAO a las capas superiores. La finalidad de la misma es poder crear estas conexiones al arrancar la aplicación y así luego poder ser recuperadas y utilizadas por las capas superiores. Gracias a esto, se reducirá el tiempo de ejecución de las operaciones, ya que no se perderá tiempo estableciendo una nueva conexión y, además, se reducirá también el número de

conexiones debido a que todas las necesarias son creadas al iniciar la aplicación, por lo tanto no hay necesidad de establecer nuevas conexiones.

Las conexiones con el robot son gestionadas por las clases java definidas en el apartado anterior, por tanto, esta capa necesita crear una instancia de esas clases. Estas instancias estarán guardadas en esta capa en toda la ejecución del programa y, además, se deben proporcionar unos métodos que permitirán recuperarlas.

4.1.1.3 Capa librería de operaciones

En esta capa se definen las operaciones que estarán disponibles en los agentes Jason. Para ello se hará uso de los artefactos *CArtAgO*. Estos artefactos fueron definidos en el apartado 3.3.1.

Para seguir una estructura idéntica a la que sigue la librería *NAOqi*, se definirán los siguientes artefactos: artefacto Core, artefacto *Motion*, artefacto Audio, artefacto *Vision*, artefacto *PeoplePerception* y artefacto *Sensors*. No se definirá un artefacto Trackers, ya que se considera que implementar las funcionalidades de este módulo queda fuera de los objetivos del proyecto.

4.1.1.4 Capa puente Cartago – Jason

La función de esta capa es transformar las llamadas a operaciones desde un agente *Jason* a su correspondiente llamada a una operación proporcionada por un artefacto *CArtAgO*. Otra función de esta capa es la de transformar las propiedades observables y eventos de *CArtAgO* en creencias del agente Jason. También deberá hacer la transformación de los tipos de datos que utiliza Jason a los tipos de datos utilizados por *CArtAgO*. Estas transformaciones serán:

- De *CArtAgO* a *Jason*:
  - Los *boolean* se transforman en *boolean*
  - *Int, long, float* y *double* se transforman en *doubles* (*NumberTerm*)
  - Los *String* se transforma en *String*
  - *Null* se transforma en una variable no definida
  - Los arrays se transforman en listas
- De *Jason* a *CArtAgO*:
  - Los *boolean* se transforman en *boolean*
  - Un término numérico (*NumberTerm*) se transforma al tipo de número más pequeño que es suficiente para contener los datos
  - Los *Strings* se transforman en objetos *String*
  - Las estructuras se transforman en objetos *String*
  - Las variables no definidas se transforman en parámetros de salida (representados por la clase *OpFeedbackParam*)
  - Las listas se transforman en *arrays*

Para realizar todo esto, se utilizará la librería *c4jason* proporcionada por *CArtAgO*.

Cabe destacar que si se desean utilizar las capas inferiores para proporcionar un conjunto de operaciones a otro lenguaje que no sea Jason, solo sería necesario cambiar la implementación de esta capa por otra que transformase las propiedades observables, los eventos y los tipos de los datos de la misma forma que lo haría c4jason. Por tanto,

cambiando *c4jason* por *c4jadex*, se podrían utilizar todas las operaciones definidas en la librería de operaciones desde el lenguaje *Jadex,* el cual es otro lenguaje de programación orientado a agentes inteligentes basado en la arquitectura BDI.

4.2 Implementación de la API

Una vez diseña la API, se va proceder a su implementación. Como la arquitectura de la *API* es en capas, en este apartado se definirá la implementación de cada una de ellas. Cabe destacar que tal y como se definió en el diseño, la capa librería *NAO* y la capa puente *CArtAgO – Jason* corresponden a herramientas ya hechas. La librería del NAO fue implementada por los desarrolladores del robot, *Aldebaran robotics*, y en el caso de puente *CArtAgO – Jason*, el cual utiliza c4jason, fue implementado por el equipo de desarrollo de *CArtAgO*. Como consecuencia, estas dos capas se omitirán en este apartado.

Una vez iniciado el proceso de implementación, se ha visto que la librería que proporciona *Aldebaran* para programar el NAO es muy extensa, y en algunas partes bastante compleja. El objetivo de este trabajo no es abarcar toda esta librería, por tanto, a continuación se nombrarán las clases que se utilizarán:

- *ALMotion*
- *ALTextToSpeech*
- *ALRobotPosture*
- *ALMemory*
- *ALSonar*
- *ALNavigation*
- *ALAnimatedSpeech*
- *ALVideoDevice*
- *ALAudioPlayer*
- *ALLeds*
- *ALBasicAwareness*
- *ALSpeechRecognition*
- *ALSystem*
- *ALVisualCompass*

Destacar que las clases *ALBasicAwareness, ALSpeechRecognition, ALSystem* y *AlVisualCompass* no están disponibles en robots virtuales, por tanto, si se intenta crear una instancia de esta clase pasándole como argumento la dirección de un robot virtual saltará una excepción en tiempo de ejecución, lo cual se debe prevenir.

4.2.1 Implementación de la capa interfaz de conexión con NAO

Para implementar esta capa, se ha decidido utilizar una clase Java que contendrá todas las instancias de las clases de la librería *NAO*. Esta clase a su vez proporcionará unos métodos los cuales permitirán obtener estas instancias.

En un primer momento, se decidió hacer esto siguiendo el patrón de diseño *singleton*, así estaríamos seguros de que se cumpliría la funcionalidad principal de esta capa: crear las conexiones tcp, las cuales están implementadas en las clases de la librería NAO, una

única vez en todo el programa. Esta implementación se puede ver en el siguiente diagrama de clases:

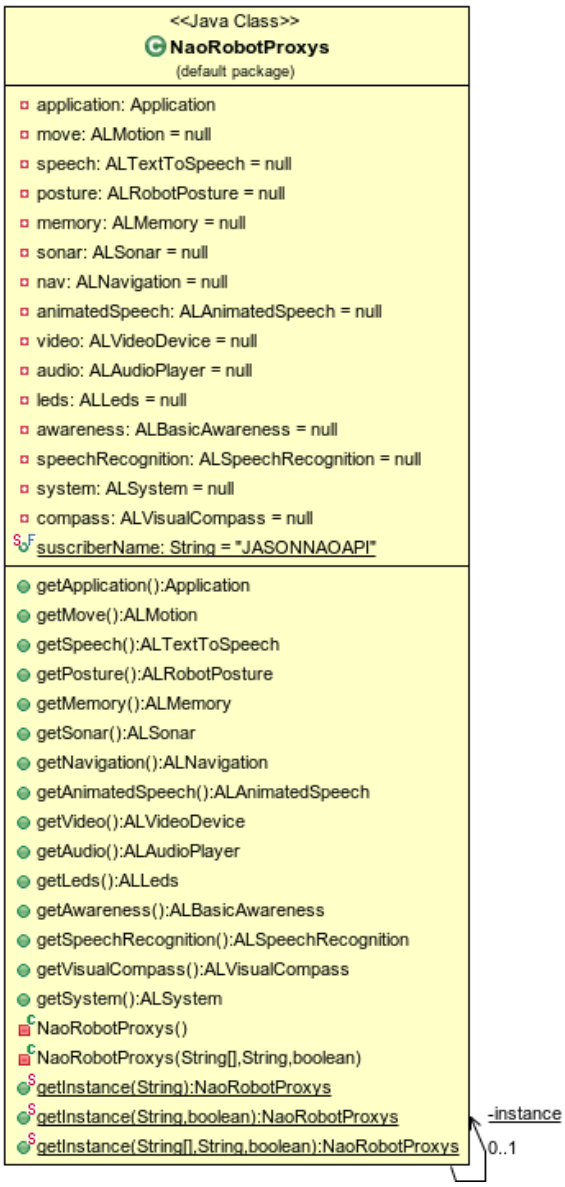


Ilustración 16 Diagrama de clases inicial

Como se puede observar, esta clase a la que se ha nombrado como *NaoRobotProxys* contiene todas las clases de la librería *NAOqi* que se van a necesitar. Posee un atributo *instance* privado, el cual solo se puede obtener mediante un método *getInstance*. Este método está sobrecargado y se puede llamar de tres formas distintas, pero la función de todas ellas es devolver el atributo *instance* si este ya ha sido inicializado, sino se inicializará según los parámetros proporcionados. Una vez la instancia está creada, los parámetros proporcionados se ignorarán y se devolverá la instancia guardada.

Como se puede observar, el método *getInstance* se puede llamar de 3 formas:

- Pasándole como argumento un *String*, que será la *url* al robot
- Pasándole como argumentos un *String* y un *boolean*, que indicarán la *url* del robot y si el robot es virtual o no.

- Pasándole como argumentos una lista de *String*, un *String* y un *boolean*. Estos indicarán los parámetros para crear la conexión con el robot, la *url* del robot y si el robot es virtual o no.

Los parámetros para crear la conexión y la *url* del robot son necesarios siempre que se crea una conexión con un robot. Si los parámetros no se proporcionan en la llamada a *getInstance*, estos tomarán como valor { "" }. Por otra parte, el parámetro que indica si es un robot virtual hace falta para para saber si crear todas las conexiones del robot u omitir aquellas que no son compatibles con los robots virtuales.

Esta implementación cumple con los requerimientos definidos en el diseño de la *API*, pero tiene un gran inconveniente: sólo se puede crear una conexión con un robot. Si se quisiera crear un programa en *Jason* en el cual interactúan dos o más robots no lo podríamos hacer con esta implementación, por tanto, aunque cumple con el diseño necesita ser reformulada para poder interactuar con más de un robot.

Siguiendo esta primera implementación, se ha optado por reformularla de la siguiente manera:

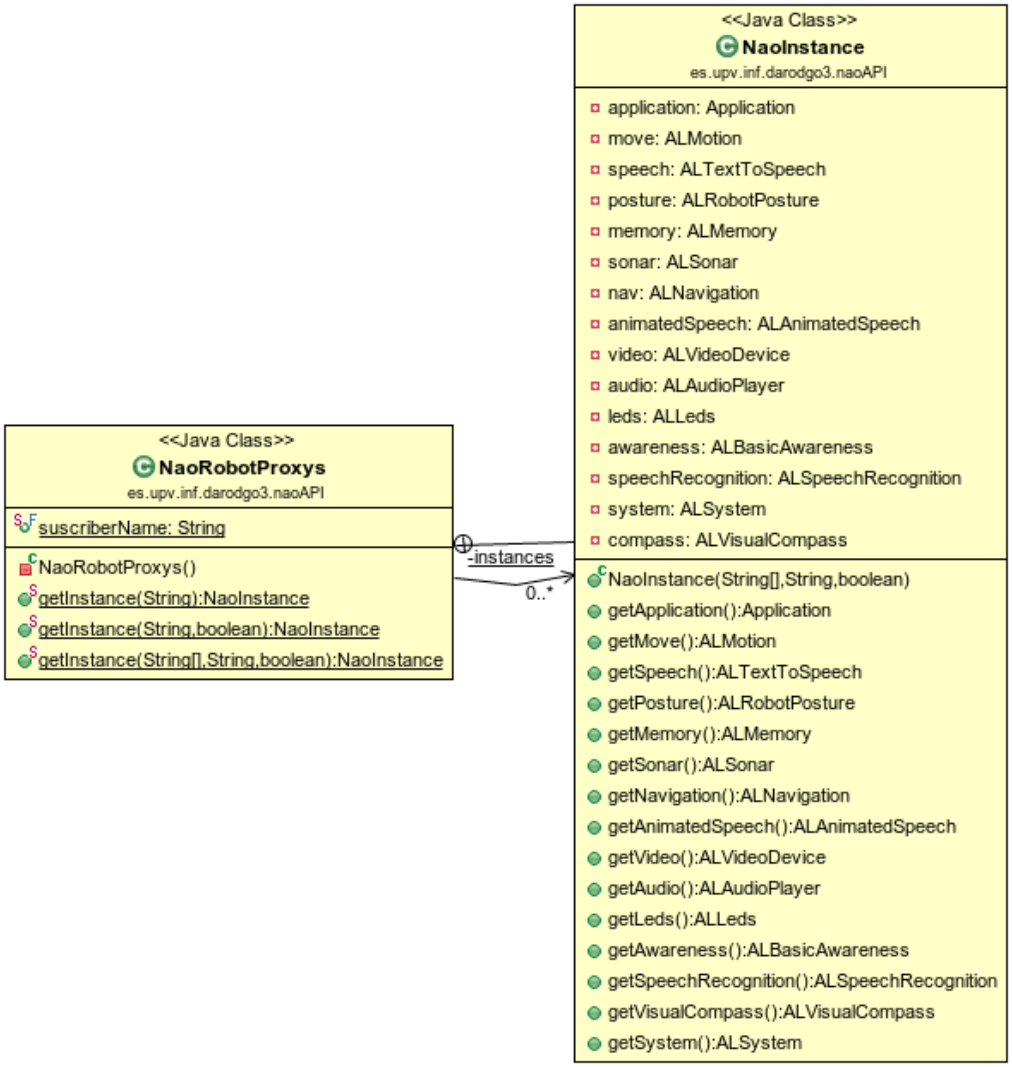


Ilustración 17 Diagrama de clases final

En esta implementación en vez de guardar una única instancia, que en este caso pasa a llamarse *NaoInstance*, se ha optado por guardar una instancia por cada *url* de robot, la cual se suministra de igual manera llamando al método *getInstance*. La instancia para una *url* seguirá siendo única y se creará al llamar al método *getInstance*, si esta no existiera. Para mantener todas estas instancias y poder acceder a ellas en función de la *url* se empleará un *Map*, que tiene como clave la *url* del robot y como valor la instancia.

En todo lo demás esta implementación es igual a la anterior. A continuación, se muestra un ejemplo de cómo emplear esta clase para recuperar la clase *ALMotion*:

```
NaoRobotProxys.getInstance(url, isVirtual).getMove();
```

4.2.1 Implementación de la capa librería de operaciones

Una vez tenemos ya los *proxys* de conexión con el robot, en este apartado vamos a proceder a implementar todo el conjunto de operaciones que estarán disponibles en *Jason*. Como ya se describió en el apartado 4.1.1.3, estas operaciones se implementarán mediante artefactos *CArtAgO*, los cuales nos permiten definir operaciones, actualizar la base de creencias del agente Jason, etc.

Pero antes, vamos a describir todas las funciones y clases de utilidades que se han implementado para facilitar y ayudar en la implementación de los artefactos.

4.2.1.1 Utilidades

Como se ha visto en el apartado 4.1.1.4, cuando desde *Jason* se le envía un número a *CArtAgO*, este es representado en el menor tipo que lo pueda contener. Esto puede generar problemas a la hora de usar estos números en código Java, ya que puede lanzar una excepción si no coinciden los tipos. Para solucionar esto, se ha creado una clase, a la que le hemos llamado *NumberParser*. Esta clase contiene los siguientes métodos estáticos:

NumberParser:

cartagoNumberToDouble

Convierte un número proporcionado por *Jason*, que a priori no se sabe de que tipo es, lo convierte a *Double* y lo retorna.

Recibe: un *Object*, que es el parámetro proporcionado por *Jason*.

Devuelve: el objeto recibido convertido a *Double*.

cartagoNumberToFloat:

Convierte un número proporcionado por *Jason*, el cual a priori no se sabe de que tipo es, lo convierte a *Float* y lo retorna.

Recibe: un *Object*, que es el parámetro proporcionado por *Jason*.

Devuelve: el objeto recibido convertido a *Float*.

ListParser:

También muchas veces necesitaremos convertir una lista Jason, que se convierte en un array Java, en un *List* Java. Para ello se ha definido la esta clase con los siguientes métodos estáticos:

cartagoStringListToList

Convierte una lista Jason de *Strings* en una lista Java de *Strings*.

Recibe: una lista de *Strings* proporcionada por Jason.

Devuelve: un *List* de Java que contiene todos los elementos de la lista de entrada.

cartagoNumberListToDoubleList

Recibe una lista de números Jason, los convierte a *Double* utilizando las funciones definidas en la clase *NumberParser*, los añade a una lista Java y la retorna.

Recibe: una lista de números *Jason*.

Devuelve: un *List* de Java en el que todos sus elementos son de tipo *Double*.

cartagoNumberListToFloatList

Recibe una lista de números Jason, los convierte a *Float* utilizando las funciones definidas en la clase *NumberParser*, los añade a una lista Java y la retorna.

Recibe: una lista de números Jason.

Devuelve: un *List* de Java en el que todos sus elementos son de tipo *Float*.

javaStringListToCartagoString

Recibe un *List* Java de *Strings* y lo convierte a un *String* que se compone de la siguiente manera: [“elemento1”, “elemento2”, ...]

Recibe: un *List* Java de *Strings*.

Devuelve: un *String*

Picture

Cuando se recupera información de la cámara, la imagen viene representada como un array de bytes. Una de las operaciones muestra esa imagen en una ventana, pero para ello primero necesita convertir ese array de bytes en un *java.awt.image.BufferedImage*. Para ayudar en este propósito se ha implementado esta clase. Los métodos que posee son:

toPicture

Método estático que convierte un array de bytes, en el que está representada una imagen, en un *java.awt.image.BufferedImage*. Cabe destacar que este método sólo está pensado para convertir imágenes de 640x480.

Recibe: un array de bytes.

Devuelve: un objeto *java.awt.image.BufferedImage* que contiene la imagen.

Para finalizar con este apartado, se han diseñado unas clases de tipo enumerados, en las que una de ella nos servirá para poder hacer la conversión de un número de forma cuantitativa a un número real, y la otra nos servirá para convertir una velocidad expresada en forma cuantitativa a un número real. Estas clases son las siguientes:

Quantity

Clase enumerado que hace relación entre un adjetivo cuantitativo a un número real. Los adjetivos cuantitativos son: todo, mucho, medio, poco y nada. Estos adjetivos se asocian con los siguientes valores: 1.0, 0.75, 0.5, 0.25 y 0 respectivamente. Además, esta clase posee los siguientes métodos:

getQuantity

Este método es un método de instancia que retorna el valor numérico del enumerado. Este método se usa de la siguiente manera:

```
Quantity.MUCHO.getQuantity();
```

Esta instrucción devuelve 0.75

FromString

Este es un método estático, el cual recibe un *String* y retorna el tipo enumerado asociado a ese *String* en el caso de que exista, sino devuelve *null*. Este método se utiliza de la siguiente manera:

```
Quantity.fromString("todo");
```

Esta instrucción devuelve el enumerado Quantity.TODO.

Velocity

Clase enumerado que hace relación entre un adjetivo cuantitativo de velocidad a un número real. Los adjetivos cuantitativos son: muyrapido, rapido, normal, lento y muylento. Estos adjetivos se asocian con los siguientes valores: 1.0, 0.8, 0.5, 0.3 y 0.1 respectivamente. Además, esta clase posee los siguientes métodos:

getVelocity

Este método es un método de instancia que retorna el valor numérico del enumerado. Este método se usa de la siguiente manera:

```
Velocity.MUYRAPIDO.getVelocity();
```

Esta instrucción devuelve 1.0

FromString

Este es un método estático, que recibe un *String* y retorna el tipo enumerado asociado a ese *String* en el caso de que exista, sino devuelve *null*. Este método se utiliza de la siguiente manera:

```
Velocity.fromString("muyrapido");
```

Esta instrucción devuelve el enumerado Velocity.MUYRAPIDO.

4.2.1.2 Artefactos

Los artefactos son la parte más importante de este trabajo. En ellos podemos definir operaciones que luego podrán ser usadas desde un programa *Jason*, que nos permite abordar el objetivo principal.

En este apartado se implementarán los artefactos que se definieron en el apartado 4.1.1.3, que son los siguientes: artefacto Audio, artefacto Core, artefacto *Motion*, artefacto *PeoplePerception*, artefacto *Sensors* y el artefacto *PeoplePerception*. Además de estos, durante el proceso de implementación se observó que podría ser útil implementar un artefacto que nos permitiese crear todos los demás artefactos en Jason empleando una sola instrucción. A este artefacto se denominó *NaoRobot*. Por último, en el proceso de validación de la aplicación surgió la necesidad de poder ejecutar una acción ante un determinado estimulo. Para ello se implementó un artefacto que generaría una creencia *Jason* cada vez que se pulsase un botón de un mando de consola. A este artefacto se le denominó *GamePad*.

A continuación, se detallará la implementación de cada uno de estos artefactos, las operaciones que proporcionan y las creencias que generan en el agente *Jason*.

Antes de eso, y con el objetivo de entender mejor las operaciones del robot, se va a detallar como se llevan a cabo las operaciones síncronas (o bloqueantes) y las operaciones asíncronas (o no bloqueantes)

Operaciones síncronas y asíncronas

Las operaciones que se ejecutan en el Nao pueden ser de dos tipos: síncronas y asíncronas.

Las operaciones síncronas lo que hacen es esperar a que la operación finalice para devolver la ejecución al programa principal. Esto se puede visualizar mejor en el siguiente diagrama de secuencia:

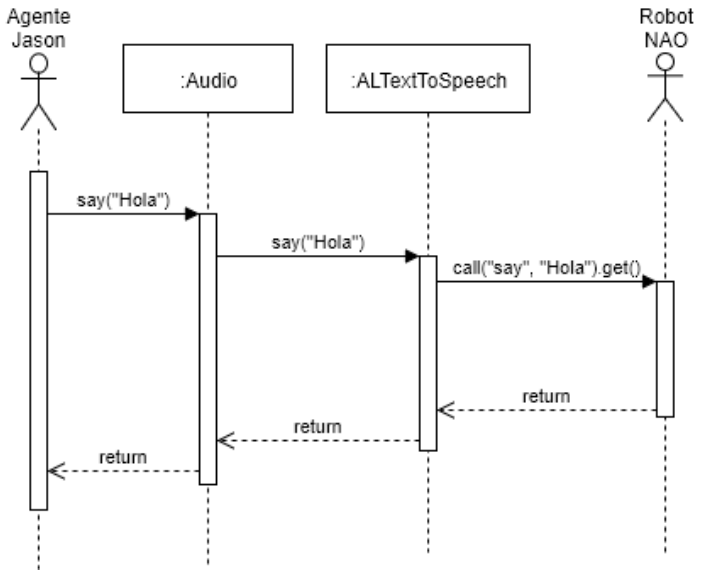


Ilustración 18 Diagrama de secuencia de una operación síncrona

Para simplificar este ejemplo, se ha omitido la capa puente entre CArtAgO – Jason, ya que complicaría bastante el diagrama y no ayudaría a comprender este tipo de operaciones, y también se ha simplificado la llamada al robot por el mismo motivo.

Como se puede ver en el diagrama, el agente Jason hace una llamada a una operación, en este caso *say*, y el hilo principal de ejecución se queda esperando hasta que el robot termine de decir la palabra "Hola".

Ahora vamos a ver el mismo caso, pero con una operación asíncrona.

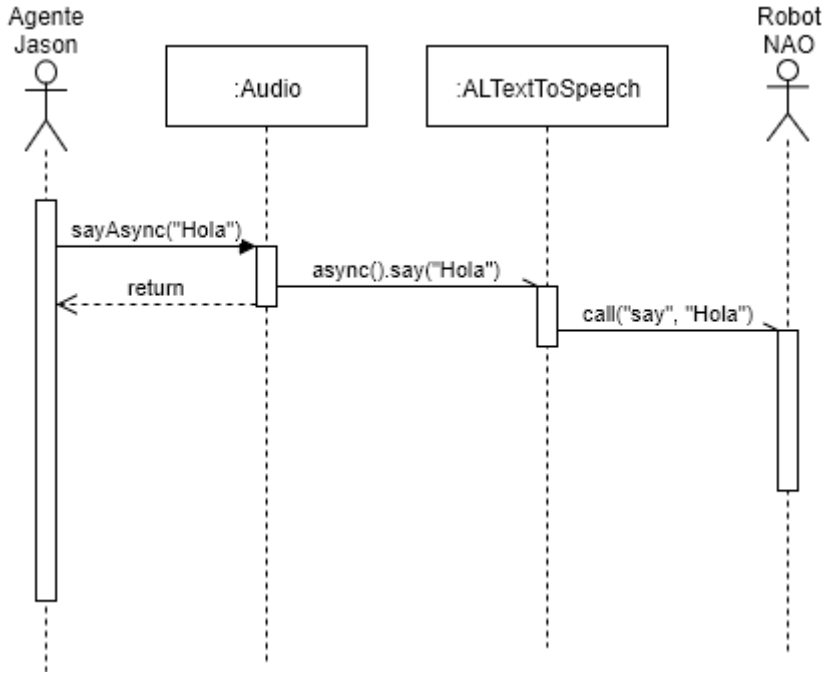


Ilustración 19 Diagrama de secuencia de una operación asíncrona

Como se puede observar en este caso, el agente Jason hace una llamada a la operación *sayAsync*, pero en vez de esperar a que el robot diga la palabra "Hola", devuelve la ejecución del programa al agente Jason que puede ejecutar más operaciones, aunque el robot no haya terminado de decir "Hola".

Llegados a este punto ya estamos listos para describir los artefactos y sus operaciones.

Artefacto Audio

Este artefacto *CArtAgO* permite controlar los componentes de audio del robot. Puede hacer hablar al robot, cambiar el idioma del robot, cambiar la voz del robot, permite habilitar el reconocimiento de voz, etc. Una vez creado el artefacto se pueden utilizar las siguientes operaciones:

init(String url)
Igual que init(url, false)

Parámetros:
url - La url del robot

init(String url, boolean isVirtual)

Método que se ejecuta al crear el artefacto. Este método recupera de *NaoRobotProxys* los proxys necesarios para que el artefacto pueda funcionar, los cuales son: *ALSpeechRecognition*, *ALMemory*, *ALTextToSpeech*, *ALAnimatedSpeech*. Una vez recuperados los almacena en atributos de clase, así las operaciones podrán hacer uso de los mismos sin tener que volver a recuperarlos.

Parámetros:
url - La url del robot
isVirtual - Indica si el robot es virtual o no

say(String text2say)

Convierte un texto a sonido: toma un *String* como entrada, lo convierte en sonido y lo emite en ambos altavoces. Ésta es una operación síncrona.

Parámetros:
text2say - El texto que será convertido en sonido

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),say_async(text2say)]

sayAsync*(String text2say)*

Igual que *say*, pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
text2say - El texto que será convertido en sonido

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),say_async(text2say)]

sayAsyncWithDelay(String text2say, long delay)

Convierte un texto en sonido y lo reproduce al cabo de un tiempo, el cual es indicado como parámetro. Ésta es una operación asíncrona.

Parámetros:
text2say - El texto que será convertido en sonido
delay - El tiempo de retraso, en milisegundos.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),say_async_with_delay(text2say, delay)]

sayToFile(String text2say, String fileName)

Convierte un texto en sonido y lo guarda en un fichero. Ésta es una operación síncrona.

Parámetros:
text2say - El texto que será convertido en sonido
fileName - Nombre del fichero donde se guardará el sonido.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),say_to_file(text2say, fileName)]

sayToFileAsync(String text2say, String fileName)
Igual que *sayToFile*, pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
text2say - El texto que será convertido en sonido
fileName - Nombre del fichero donde se guardará el sonido.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),say_to_file_async(text2say, fileName)]

sayHelloAnimated()
Hace que el robot diga "Hola" mientras mueve el brazo como si estuviera saludando. Ésta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),say_hello_animated]

sayAnimated(String animation, String text2say)
Hace que el robot hable a la vez que mueve su cuerpo siguiendo una animación, la cual es proporcionada como parámetro. Ésta es una operación síncrona.

Parámetros:
animation - La animación que efectuará mientras habla. Todas las animaciones están disponbiles en: http://doc.aldebaran.com/2-1/naoqi/audio/alanimatedspeech_advanced.html#animated-speech-list-behaviors-nao
text2say - El texto que se convertirá a voz.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),say_animated(animation,text2say)]

sayAnimatedAsync(String animation, String text2say)
Igual que *sayAnimated*, pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
animation - La animación que efectuará mientras habla. Todas las animaciones están disponbiles en: http://doc.aldebaran.com/2-1/naoqi/audio/alanimatedspeech_advanced.html#animated-speech-list-behaviors-nao
text2say - El texto que se convertirá a voz

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),say_animated(animation,text2say)]

setVocabulary(Object[] vocabulary, boolean enabledWordSpotting)

Establece la lista de palabras (vocabulario) que debe ser reconocidas por el motor de reconocimiento de voz. Ésta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
vocabulary - Lista de palabras que deben ser reconocidas
enabledWordSpotting - Si está deshabilitado, el motor espera escuchar una de las palabras especificadas, nada más, nada menos. Si está activada, las palabras especificadas se pueden pronunciar en medio de un flujo de voz entero, el motor intentará detectarlas.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),set_vocabulary(vocabulary,enabledWordSpotting)]

startWordRecognition()

Empieza el proceso de reconocimiento de palabras. Cuando una palabra es detectada, esta genera una creencia en Jason la cual sigue la siguiente forma: *wordRecognized(word)*, donde *word* es la palabra reconocida. Ésta es una operación asíncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),start_word_recognition]

stopWordRecognition()

Detiene el reconocimiento de palabras. Después de ejecutar esta operación no se reconocerá ninguna palabra más hasta que se vuelva a ejecutar *startWordRecognition*. Ésta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),stop_word_recognition]

speechGetAvailableLanguages(OpFeedbackParam<String> res)

Obtiene la lista de idiomas instalados en el sistema del robot, y los devuelve en el parámetro de salida res como un *String*. Ésta es una operación síncrona.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_get_available_languajes]

speechSetLanguage(String lang)

Cambia el idioma del motor que traduce el texto a sonido. Este idioma debe estar instalado en el robot. Para ello puede hacer uso de la operación *speechGetAvailableLanguages*. Ésta es una operación síncrona.

Parámetros:
lang - Nuevo idioma del robot

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_set_languaje(lang)]

speechGetAvailableVoices(OpFeedbackParam<String> res)

Obtiene la lista de voces instaladas en el sistema del robot, y las devuelve en el parámetro de salida res como un *String*. Ésta es una operación síncrona.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_get_available_voices]

speechGetParameter(String parameter, OpFeedbackParam<Float> res)

Devuelve el valor de uno de los parámetros de voz en el parámetro de salida res. Los parámetros disponibles son: "*pitchShift*", "*doubleVoice*", "*doubleVoiceLevel*" y "*doubleVoiceTimeShift*". Ésta es una operación síncrona.

Parámetros:
parameter - Nombre del parámetro a obtener
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_get_parameter(parameter)]

speechSetParameter(String parameter, float value)

Cambia los parámetros de la voz. Ésta es una operación síncrona. Los parámetros disponibles son:

- *pitchShift*: aplica un cambio de tono a la voz. El valor indica la relación entre las nuevas frecuencias fundamentales y las anteriores (ejemplos: 2.0: una octava arriba, 1.5: una quinta arriba). El rango correcto es (1.0 - 4), o 0 para desactivar el efecto.
- *DoubleVoice*: añade una segunda voz a la primera. El valor indica la relación entre la frecuencia fundamental de la segunda voz y la primera. El rango correcto es (1.0-4) o 0 para desactivar el efecto.
- *doubleVoiceLevel*: el valor correspondiente es el nivel de la voz doble (1.0: igual a la voz principal). El rango correcto es (0 - 4).
- *DoubleVoiceTimeShift*: el valor correspondiente es el retardo entre la voz doble y la principal. El rango correcto es (0 - 0.5) Si el valor del efecto no está disponible, el parámetro del efecto permanece sin cambios.

Parámetros:
parameter - Nombre del parámetro
value - Nuevo valor

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_set_parameter(parameter,value)]

speechGetVoice(OpFeedbackParam<String> res)

Obtiene la actual del robot, y la devuelve en el parámetro de salida res como un *String*. Ésta es una operación síncrona.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_get_voice]

speechSetVoice(String voice)

Cambia la voz del robot. Esta voz debe estar instalada en el robot. Para ello puede hacer uso de la operación *speechGetAvailableVoices*. Ésta es una operación síncrona.

Parámetros:
voice - Nuevo idioma del robot

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_set_voice(voice)]

speechGetVolume(OpFeedbackParam<Float> res)

Obtiene el volumen actual del texto a voz y lo devuelve en el parámetro de salida res. El volumen es un número entero que va de 0 a 100. Ésta es una operación síncrona.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_get_volume]

speechSetVolume(float volume)

Establece el volumen de la salida de texto a voz. El volumen es un número en coma flotante el cual va de 0.0 a 1.0. Ésta es una operación síncrona.

Parámetros:
volume - Nuevo valor de volumen.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_set_volume(volume)]

speechStopAll()

Esta operación detiene la actual y todas las tareas de conversión te texto a sonido pendientes inmediatamente. Ésta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),speech_stop_all]

endAudio()

Esta operación termina todas las conexión y subscripciones a eventos establecidas anteriormente, con el fin de poder terminar el programa correctamente. Por último, deshace el artefacto.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg)]

Artefacto Core

Este artefacto proporciona un conjunto de operaciones relacionadas con la memoria, el sistema y algunos comportamientos genéricos del robot.

init(String urlRobot)

Igual que *init(url, false)*

Parámetros:
urlRobot - La *url* del robot

init(String urlRobot, boolean isVirtual)

Método que se ejecuta al crear el artefacto. Este método recupera de *NaoRobotProxys* los proxys necesarios para que el artefacto pueda funcionar, los cuales son: *ALSystem* y *ALMemory*. Una vez recuperados los almacena en atributos de clase, así las operaciones podrán hacer uso de los mismos sin tener que volver a recuperarlos.

Parámetros:
urlRobot - La *url* del robot
isVirtual - Indica si el robot es virtual o no

getRobotName(OpFeedbackParam<String> res)

Obtiene el nombre del robot y lo devuelve en el parámetro de salida res como un *String*. Ésta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),get_robot_name]

shutdownRobot()

Apaga el robot. Ésta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),shutdown_robot]

rebootRobot()
Reinicia el robot. Ésta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),reboot_robot]

getFreeMemory(OpFeedbackParam<Integer> res)
Obtiene la cantidad de memoria disponible en el *heap* del robot y la devuelve en el parámetro de salida res como un *Integer*. Ésta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),get_free_memory]

getTotalMemory(OpFeedbackParam<Integer> res)
Obtiene la cantidad de memoria total del robot y la devuelve en el parámetro de salida res como un *Integer*. Ésta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),get_total_memory]

getMemoryData(String key, OpFeedbackParam<Object> res)
Obtiene el valor almacenado en la memoria bajo la clave "*key*" proporcionada. Este valor se devuelve en el parámetro de salida *res*. Ésta es una operación síncrona.

Parámetros:
key - Nombre del valor a recuperar
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),get_memory_data(key)]

endCore()

Esta operación termina todas las conexiónes y subscripciones a eventos establecidas anteriormente, con el fin de poder terminar el programa correctamente. Por último, deshace el artefacto.

Artefacto Motion

Este artefacto proporciona un conjunto de operaciones relacionadas con el movimiento del robot. Desde el movimiento de las articulaciones, indicando el ángulo, hasta operaciones de más alto nivel que consiguen que el robot camine.

init(String url)

Igual que *init(url, false)*

Parámetros:
url - La *url* del robot

init(String url, boolean isVirtual)

Método que se ejecuta al crear el artefacto. Este método recupera de *NaoRobotProxys* los proxys necesarios para que el artefacto pueda funcionar, estos son: *ALMotion*, *ALNavigation* y *ALRobotPosture*. Una vez recuperados los almacena en atributos de clase, así las operaciones podrán hacer uso de los mismos sin tener que volver a recuperarlos. Además, esta operación hará todo lo necesario para que el robot se levante automáticamente después de cada caída. Por último, crea la siguiente propiedad observable, la cual se transforma automáticamente en una creencia en el agente Jason:

awaked: Indica si el robot esta despierto o si está dormido.

Parámetros:
url - La *url* del robot
isVirtual - Indica si el robot es virtual o no

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),motion_init(url,isVirtual)]

goRest()

El robot descansará: establece a una posición de relajación y seguridad y además apagará todos los motores. Ésta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),go_rest]

wakeUp()

El robot se despertará: enciente todos los motores y establece una posición inicial si es necesario. Ésta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),wake_up]

posture(String posture, Object maxSpeedFraction)

Hace que el robot establezca una postura predeterminada, en una fracción de tiempo dada. Ésta es una operación síncrona.

Parámetros:
posture - Nombre de la postura. Puede ser: *Crouch*, *LyingBack*, *LyingBelly*, *Sit*, *SitRelax*, *StandInit*, *Stand*, *StandZero*
maxSpeedFraction - Fracción de tiempo, que puede ser representada como un valor numero el cual va de 0 a 1, o como un elemento del enumerado *Velocity*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),posture(posture,maxSpeedFraction)]

openHandNAO(String handName)

Abre una mano del NAO. Ésta es una operación síncrona.

Parámetros:
handName - Nombre de la mano, puede ser *RHand* o *LHand*

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),open_hand_NAO(handName)]

closeHandNAO(String handName)

Cierra una mano del NAO. Ésta es una operación síncrona.

Parámetros:
handName - Nombre de la mano, puede ser *RHand* o *LHand*

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),close_hand_NAO(handName)]

moveBothHandNAO(Object amplitude, Object vel)

Mueve las dos manos del NAO a una posición a la vez a una velocidad regulable. Ésta es una operación síncrona.

Parámetros:
amplitude - Amplitud a la que se quieren colocar las manos. Puede ser un *float* o un *String*. Si es un *String* se debe proporcionar un valor del *enum Quantity* (todo, mucho, medio...)
vel - Velocidad con la que se colocará la nueva amplitud. Puede ser un *float* o un *String*. Si es un *String* debe proporcionar un valor del *enum Velocity* (muyrapido, rapido...).

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),move_both_hand_NAO(amplitude,vel)]

angleInterpolationNAO(Object[] names, Object[] keys, Object[] times, boolean isAbsolute)

Interpola una o varias articulaciones a un ángulo de objetivo a lo largo de trayectorias temporizadas. Ésta es una operación síncrona.

Parámetros:
names - Nombres de las articulaciones
keys - Ángulos de las articulaciones (radianes)
times - Tiempos en los cuales las articulaciones tienen que alcanzar la nueva posición
isAbsolute - true si los ángulos son absolutos, false si son relativos.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),angle_interpolation_NAO(names,keys,time,isAbsolute)]

getAnglesNAO(Object[] names, Boolean useSensors, OpFeedbackParam<Object[]> res)

Obtiene los ángulos (en radianes) de las articulaciones y se devuelve por el parámetro de salida. Ésta es una operación síncrona.

Parámetros:
names - Nombres de las articulaciones
useSensors - Si es true, se devolverán los ángulos usando los sensores
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),get_angles_NAO(names,useSensors)]

setAnglesNAO(Object[] joints, Object[] angles, Object vel)

Establece los ángulos (en radianes) de las articulaciones proporcionadas a la velocidad indicada. La velocidad se puede indicar como un número entre 0 y 1, o como un valor del enumerado *Velocity*. Ésta es una operación asíncrona.

Parámetros:
joints - Nombres de las articulaciones.
angles - Angulo de destino de las articulaciones
vel - Velocidad a la cual se moverán las articulaciones.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),set_angles_NAO(names,angles,vel)]

setAnglesNAOBlocking(Object[] joints, Object[] angles, Object vel)

Igual que *setAnglesNAO*, pero en vez de ser una operación asíncrona es una operación síncrona.

Parámetros:
joints - Nombres de las articulaciones.
angles - Angulo de destino de las articulaciones
vel - Velocidad a la cual se moverán las articulaciones.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),set_angles_NAO_blocking(names,angles,vel)]

getFallManagerEnabled(OpFeedbackParam<Object> res)

Devuelve el estado del gestor de caídas por el parámetro de salida. True si el gestor de caídas está activado. Ésta es una operación síncrona.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),get_fall_manager_enabled]

setFallManagerEnabled(boolean newValue)

Activar/desactivar la protección del gestor de caídas para el robot. Cuando se detecta una caída, el robot adopta una configuración conjunta para protegerse y reducir la rigidez. Un evento de memoria llamado "*robotHasFallen*" se genera cuando el *fallManager* se ha activado. Para obtener este evento desde Jason ver el artefacto *Sensors*. Ésta es una operación síncrona.

Parámetros:
newValue - true para activar el gestor de caídas.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),set_fall_manager_enabled(newValue)]

moveInit()

Inicializa el proceso de movimiento. Comprueba la pose del robot y toma la postura correcta. No es necesario llamarla antes de caminar, ya que esta operación se utiliza internamente. Ésta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),move_init]

robotIsWakeUp(OpFeedbackParam<Object> res)

Comprueba si el robot esta levantado o no y devuelve esa comprobación por el parámetro de salida. Ésta es una operación síncrona.

Parámetros:
res - Parámetro de salida.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),robot_is_wake_up]

moveNAO(Object x, Object y, Object theta)

Hace que el robot se mueva a la velocidad normalizada dada. Ésta es una operación asíncrona.

Parámetros:
x - La velocidad normalizada a lo largo del eje x (entre -1 y 1).
y - La velocidad normalizada a lo largo del eje y (entre -1 y 1).
theta - La velocidad normalizada a lo largo del eje z (entre -1 y 1).

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),move_NAO(x,y,theta)]

moveToNAO(Object x, Object y, Object theta)

Hace que el robot se mueva en la posición dada. Ésta es una operación asíncrona.

Parámetros:
x - La posición a lo largo del eje x [m].
y - La posición a lo largo del eje y [m].
theta - La posición a lo largo del eje z [grados].

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),move_to_NAO(x,y,theta)]

setMoveArmsEnabled(boolean leftArmEnabled, boolean rightArmEnabled)

Establece si los movimientos de brazos están habilitados durante el proceso de movimiento. Es una operación síncrona.

Parámetros:
leftArmEnabled - Si true - Los movimientos del brazo izquierdo son controlados por la tarea de movimiento.
rightArmEnabled - Si true - Los movimientos del brazo derecho son controlados por la tarea de movimiento

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera:
[error_msg(Msg),set_move_arms_enabled(leftArmEnabled,rigthArmEnabled)]

waitUntilMoveIsFinished()

Espera hasta que finalice el proceso de movimiento: Esta operación se puede utilizar para bloquear la ejecución del programa hasta que la tarea de movimiento esté totalmente terminada, o lo que es lo mismo, si se usa después de una operación de movimiento hace que esta sea síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),wait_until_move_is_finnished]

moveIsActive(OpFeedbackParam<Boolean> res)

Comprueba si el proceso de movimiento es activo y devuelve el resultado por el parámetro de salida. Ésta es una operación síncrona.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),move_is_active]

stopMoveNAO()

Detiene el proceso de movimiento del robot que se está ejecutando. Ésta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),stop_move_NAO]

navigationGetSecurityDistance(OpFeedbackParam<Float> res)

Obtiene la distancia en metros desde la cual el robot debe detenerse si hay un obstáculo, y la devuelve por el parámetro de salida. Ésta es una operación síncrona.

Parámetros:
res - Parámetro de salida.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),navigation_get_security_distance]

navigationSetSecurityDistance(Object dist)

Establece la distancia en metros desde la cual el robot debe detenerse si hay un obstáculo. Ésta es una operación síncrona.

Parámetros:
dist - Distancia en metros.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),navigation_set_security_distance(dist)]

navigationMove(Object x, Object y, Object theta)

Hace que el robot se mueva a la velocidad dada en unidades S.I. esquivando los obstáculos que se va encontrando. Ésta es una operación síncrona.

Parámetros:
x - La velocidad a lo largo del eje x [m/s]
y - La velocidad a lo largo del eje y [m/s]
theta - La velocidad a lo largo del eje x [rad/s]

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),navigation_move(x,y,theta)]

navigationMoveTo(Object x, Object y, Object theta)
Hace que el robot se mueva a la posición proporcionada, esquivando los obstáculos que se va encontrando. Ésta es una operación síncrona.

Parámetros:
x - La posición a lo largo del eje x [m]
y - La posición a lo largo del eje y [m]
theta - El ángulo alrededor del eje z [grados]

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),navigation_move_to(x,y,theta)]

navigationMoveToward(Object x, Object y, Object theta)
Hace que el robot se mueva a la velocidad dada en la fracción de velocidad normalizada, esquivando los obstáculos que se va encontrando. Ésta es una operación síncrona.

Parámetros:
x - La velocidad a lo largo del eje x [0.0-1.0]
y - La velocidad a lo largo del eje y [0.0-1.0]
theta - La velocidad a lo largo del eje x [0.0-1.0]

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),navigation_move_toward(x,y,theta)]

navigateTo(Object x, Object y, Object theta)
Hace que el robot navegue a un objetivo métrico relativo expresado en 2D. El robot calcula un camino para evitar obstáculos. Es una operación asíncrona.

Parámetros:
x - La posición a lo largo del eje x [m].
y - La posición a lo largo del eje y [m].
theta - Orientación del robot (grados)

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),navigate_to(x,y,theta)]

stopNavigateTo()
Detiene la operación *navigateTo* de forma segura.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),stop_navigate_to]

attackNAO()

Nao simula realizar un ataque. Ésta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),attack_NAO]

movementClappingTwo()

Nao dice "buen trabajo" mientras aplaude. Ésta es una operación síncrona.

El movimiento original fue diseñado por el FUN lab de la universidad de Notre Dame, empleando el timeline de Choregraphe. Ese timeline se exportó a código Java utilizando el ChoregrapheTranspiler.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_clapping_two]

movementExcelent()

NAO levanta el puño con su mano derecha y empuja su brazo derecho hacia arriba en el aire mientras dice "excelente". Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_excelent]

movementYouAreRight()

NAO alza ambas manos por encima de la cabeza, baja los brazos y mueve sus manos de un lado a otro aplaudiendo. NAO dice: "Tienes razón", para aplaudir y animar a los niños. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_you_are_right]

movementYouAreSoSmart()

NAO apunta con la mano derecha sobre la cadera, alza la mano izquierda, y dice: "Eres muy inteligente". Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_you_are_so_smart]

movementYouAreSoSmartWithCheering()
NAO apunta con la mano derecha y mueve el brazo izquierdo hacia arriba y hacia abajo, con el codo doblado y el puño cerrado. Nao dice: "Eres muy inteligente". Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_you_are_so_smart_with_cheering]

movementyesYouGotIt()
NAO dobla las rodillas y tira los dos brazos hacia abajo mientras dice: "¡Sí, lo tienes!". Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_yes_you_got_it]

movementWow()
Nao mueve la cabeza de arriba a abajo mientras dice: "*Wow*". Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_wow]

robotDance()
NAO mueve los brazos arriba y abajo con los codos doblados y balancea el brazo izquierdo en otro movimiento de la danza del robot. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),robot_dance]

taiChiChaun()

Este movimiento fue creado por *Aldeberan* y viene con *Choregraphe*. NAO realiza una serie de Movimientos de Tai Chi. Ésta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),tai_chi_chaun]

rockOn()

NAO levanta el brazo derecho hacia arriba con ligera curva en el codo y puño cerrado. Esto también podría ser usado para refuerzo positivo. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),rock_on]

thriller()

Nao baila la canción Thriller igual que lo hacía Michael Jackson. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),thriller]

goIrish()

NAO mueve sus brazos hacia atrás como si quisiera pelea. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),go_irish]

wearingOut()

NAO mueve el brazo derecho para limpiar el sudor de su frente. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),wearing_out]

danceMove()
NAO balancea los brazos hacia adelante haciendo un baile. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),dance_move]

movementBow()
NAO dobla las rodillas y se inclina hacia adelante en un movimiento de reverencia. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_bow]

blowKisses()
NAO mueve su mano izquierda hasta su boca y la mueve hacia fuera y hacia adentro para simular que está lanzando besos. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),blow_kisses]

birthdayDance()
NAO hace el giro y balancea sus brazos arriba y abajo en un movimiento de danza. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),birthday_dance]

airGuitar()
Nao mueve los brazos en una posición de guitarra de aire y comienza a tocar.Esta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),air_guitar]

discoDance()
NAO hace un baile de discoteca. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),disco_dance]

macarenaDance()
Nao baila la macarena. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),macarena_dance]

sprinklerDance()
Nao realiza el baile "*sprinkler*". Esta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),sprinkler_dance]

workOut()
NAO recoge pesos invisibles y ejercicios con ellos. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),work_out]

armDance()
NAO gira sus brazos de delante a atras y a cada lado. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),arm_dance]

movementPushups()
NAO se pone en una posición de flexión y luego hace varias flexiones antes de levantarse de nuevo. Ésta es una operación síncrona.

El movimiento original fue diseñado por el *FUN lab* de la universidad de *Notre Dame*, empleando el *timeline* de *Choregraphe*. Ese *timeline* se exportó a código *Java* utilizando el *ChoregrapheTranspiler*.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),movement_pushups]

endMotion()
Esta operación termina todas las conexiónes y suscripciones a eventos establecidas anteriormente, con el fin de poder terminar el programa correctamente. Por último, deshace el artefacto.

PeoplePerception

Este artefacto proporciona un conjunto de métodos relacionados con la percepción de personas y sus características.

init(String url)
Igual que *init(url, false)*

Parámetros:
url - La *url* del robot

init(String url, boolean isVirtual)

Método que se ejecuta al crear el artefacto. Este método recupera de *NaoRobotProxys* los proxys necesarios para que el artefacto pueda funcionar son: *ALBasicAwareness*. Una vez recuperados los almacena en atributos de clase, así las operaciones podrán hacer uso de los mismos sin tener que volver a recuperarlos. Además de esto, se habilitarán los siguientes estímulos que harán reaccionar al robot en la búsqueda de las personas: sonido, movimiento, gente y tacto. Por último, establece el tipo de seguimiento a las personas, el cual será: *SemiEngaged*. Este actúa de la siguiente manera: Cuando el robot

está en contacto con una persona, sigue escuchando los estímulos y, si obtiene un estímulo, mirará en su dirección, pero siempre volverá a la persona con la que está comprometido. Si pierde a la persona, escuchará los estímulos nuevamente y podrá involucrarse con otra persona.

Parámetros:
url - La *url* del robot
isVirtual - Indica si el robot es virtual o no

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),awareness_init(url,isVirtual)]

startAwareness(String trackingMode)

Inicia el proceso se seguimiento de personas. El tipo de seguimiento se indica por parámetro. Estos pueden ser:

- Head: El seguimiento sólo utiliza la cabeza
- BodyRotation: El seguimiento utiliza la cabeza y la rotación del cuerpo
- WholeBody: El seguimiento utiliza todo el cuerpo, pero no lo hace girar
- MoveContextually: El seguimiento utiliza la cabeza y realiza de forma autónoma pequeños movimientos, como acercarse a la persona rastreada, retroceder, girar ...

Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
trackingMode - Tipo de seguimiento

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),start_awareness]

stopAwareness()

Termina el seguimiento de personas. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),stop_awareness]

endPeoplePerception()

Esta operación termina todas las conexiónes y subscripciones a eventos establecidas anteriormente, con el fin de poder terminar el programa correctamente. Por último, deshace el artefacto.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg)]

Sensors

Este artefacto proporciona un conjunto de operaciones relacionadas con los sensores del robot. Además, se suscribe a algunos eventos de los sensores, y genera o actualiza creencias en el robot, para así tener en el disponibles lo datos de los sensores.

init(String url)

Igual que *init(url, false)*

Parámetros:
url - La *url* del robot

init(String url, boolean isVirtual)

Método que se ejecuta al crear el artefacto. Este método recupera de *NaoRobotProxys* los proxys necesarios para que el artefacto pueda funcionar, los cuales son: *ALMemory*, *ALSonar* y *ALLeds*. Una vez recuperados los almacena en atributos de clase, así las operaciones podrán hacer uso de los mismos sin tener que volver a recuperarlos. Además, crea las siguientes propiedades observables, que pueden transformarse automáticamente en creencias del agente *Jason*:

- *sonarLeft*: indica el valor de *SonarLeftDetected*. Cada vez que el robot cambie este valor, este será también actualizado en la base de creencias del agente Jason.
- *sonarNothingLeft*: indica el valor de *SonarNothingLeftDetected*. Cada vez que el robot cambie este valor, este será también actualizado en la base de creencias del agente Jason.
- *sonarRight*: indica el valor de *SonarRightDetected*. Cada vez que el robot cambie este valor, este será también actualizado en la base de creencias del agente Jason.
- *sonarNothingRight*: indica el valor de *sonarNothingRightDetected*. Cada vez que el robot cambie este valor, este será también actualizado en la base de creencias del agente Jason.
- *sonarLateralRight*: indica el valor de *SonarLateralRightDetected*. Cada vez que el robot cambie este valor, este será también actualizado en la base de creencias del agente Jason.
- *sonarLateralRight*: indica el valor de *SonarLateralRightDetected*. Cada vez que el robot cambie este valor, este será también actualizado en la base de creencias del agente Jason.
- *sonarMiddle*: indica el valor de *SonarMiddleDetected*. Cada vez que el robot cambie este valor, este será también actualizado en la base de creencias del agente Jason.
- *gyroscope*: contiene los valores del giroscopio 3D
- *accelerometer*: contiene los valores del acelerómetro 3D
- inglés: contiene los ángulos 3D

También genera las siguientes creencias:

- *frontTactilTouched*: genera esta creencia cada vez que detecta que el sensor táctil frontal de la cabeza es accionado.

- *middleTactilTouched*: genera esta creencia cada vez que detecta que el sensor táctil del medio de la cabeza es accionado.
- *rearTactilTouched*: genera esta creencia cada vez que detecta que el sensor táctil de detrás de la cabeza es accionado.
- *leftBumperPressed*: genera esta creencia cada vez que detecta que el *bumper* del pie izquierdo es accionado.
- *rightBumperPressed*: genera esta creencia cada vez que detecta que el *bumper* del pie derecho es accionado.
- *handLeftTouched*: genera esta creencia cada vez que los sensores de presión de la mano izquierda detectan presión.
- *handRightTouched*: genera esta creencia cada vez que los sensores de presión de la mano derecha detectan presión.
- *footContactChanged*: genera esta creencia cada vez que los sensores FSR detectan que el robot no está en contacto

Parámetros:
url - La *url* del robot
isVirtual - Indica si el robot es virtual o no

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),sensors_init]

fadeLeds(String name, float intensity, float duration)
Ajusta la intensidad de un *LED* o grupo de *LEDs* durante un tiempo determinado. Ésta es una operación síncrona.

Parámetros:
name - Nombre del led o del grupo de leds
intensity - La intensidad del led o del grupo de leds (un valor entre 0 y 1).
duration - Duración en segundos

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),fade_leds(name,intensity,duration)]

fadeLedsAsync(String name, float intensity, float duration)
Idéntica a *fadeLeds* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
name - Nombre del led o del grupo de leds
intensity - La intensidad del led o del grupo de leds (un valor entre 0 y 1).
duration - Duración en segundos

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),fade_leds_async(name,intensity,duration)]

fadeRGBLeds(String name, float red, float green, float blue, float duration)

Establece el color de los leds durante un tiempo determinado. El color se indica como porcentaje de rojo, verde y azul. Ésta es una operación síncrona.

Parámetros:
name - Nombre del led o del grupo de leds
red - Intensidad del canal rojo
green - Intensidad del canal verde
blue - Intensidad del canal azul
duration - Duracion en segundos

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),fade_RGB_leds(name,red,green,blue,duration)]

fadeRGBLedsAsync(String name, float red, float green, float blue, float duration)

Idéntica a *fadeRGBLeds* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
name - Nombre del led o del grupo de leds
red - Intensidad del canal rojo
green - Intensidad del canal verde
blue - Intensidad del canal azul
duration - Duración en segundos

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),fade_RGB_leds_async(name,red,green,blue,duration)]

fadeColorLeds(String name, String colorName, float duration)

Establece el color de los leds durante un tiempo determinado. Los colores soportados son: "*white*", "*red*", "*green*", "*blue*", "*yellow*", "*magenta*", "*cyan*". Esta es una operación síncrona.

Parámetros:
name - Nombre del led o del grupo de leds
colorName - El nombre del color.
duration - Duracion en segundos

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),fade_color_leds(name,colorName,duration)]

fadeColorLedsAsync(String name, String colorName, float duration)

Idéntica a *fadeColorLeds* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
name - Nombre del led o del grupo de leds
colorName - El nombre del color.
duration - Duración en segundos

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),fade_color_leds_async(name,colorName,duration)]

getIntensityLeds(String name, OpFeedbackParam<Object> res)

Obtiene la intensidad de un led o de un grupo de leds y la devuelve por el parámetro de salida. Esta es una operación síncrona.

Parámetros:
name - Nombre del led o del grupo de leds
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),get_intensity_leds(name)]

getIntensityLedsAsync(String name, OpFeedbackParam<Object> res)

Idéntica a *getIntensityLeds* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
name - Nombre del led o del grupo de leds
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),get_intensity_leds_async(name)]

ledsOn(String name)

Enciende un led o un grupo de leds. Esta es una operación síncrona.

Parámetros:
name - Nombre del led o del grupo de leds.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),leds_on(name)]

ledsOnAsync(String name)

Idéntica a *ledsOn* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
name - Nombre del led o del grupo de leds.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),leds_on_async(name)]

ledsOff(String name)
Apaga un led o un grupo de leds. Esta es una operación síncrona.

Parámetros:
name - Nombre del led o del grupo de leds.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),leds_off(name)]

ledsOffAsync(String name)
Idéntica a *ledsOff* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
name - Nombre del led o del grupo de leds.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),leds_off_async(name)]

setIntensityLeds(String name, Object intensity)
Establece la intensidad de un led o un grupo de leds. El nuevo valor de intensidad puede ser indicado como un número entre 0 y 1, o como un valor del enumerado *Quantity*: todo, mucho, medio, poco, nada. Esta es una operación síncrona.

Parámetros:
name - Nombre del led o del grupo de leds.
intensity - Nueva intensidad

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),set_intensity_leds(name,intensity)]

setIntensityLedsAsync(String name, Object intensity)
Idéntica a setIntensityLeds pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
name - Nombre del led o del grupo de leds.
intensity - Nueva intensidad

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),set_intensity_leds_async(name,intensity)]

earLedsSetAngle(int degrees, float duration, boolean leaveOnAtEnd)
Una animación para mostrar una dirección con las orejas. Esta es una operación síncrona.

Parámetros:
degrees - El ángulo que desea mostrar en grados (*int*). 0 está hacia arriba, 90 está hacia adelante, 180 está hacia abajo y 270 está de vuelta.
duration - La duración en segundos de la animación.
leaveOnAtEnd - Si es true, el último led se deja encendido al final de la animación.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),ear_leds_set_angle(degrees,duration,leaveOnAtEnd)]

earLedsSetAngleAsync(int degrees, float duration, boolean leaveOnAtEnd)

Idéntica a *earLedsSetAngle* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
degrees - El ángulo que desea mostrar en grados (*int*). 0 está hacia arriba, 90 está hacia adelante, 180 está hacia abajo y 270 está de vuelta.
duration - La duración en segundos de la animación.
leaveOnAtEnd - Si es true, el último led se deja encendido al final de la animación.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera:
[error_msg(Msg),ear_leds_set_angle_async(degrees,duration,leaveOnAtEnd)]

randomEyeLeds(float duration)

Iniciar una animación aleatoria en los les de los ojos. Esta es una operación sincrona.

Parámetros:
duration - Duración aproximada de la animación en segundos.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),random_eye_leds(duration)]

randomEyeLedsAsync(float duration)

Idéntica a *randomEyeLeds* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
duration - Duración aproximada de la animación en segundos.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),random_eye_leds_async(duration)]

rastaLeds(float duration)

Lanzar una animación *rasta* verde / amarillo / rojo en los leds de todo el cuerpo. Esta es una operación síncrona.

Parametros:
duration - Duración aproximada de la animación en segundos.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),rasta_leds(duration)]

rastaLedsAsync(float duration)
Idéntica a *rastaLeds* pero en vez de ser una operación síncrona es una operación asíncrona.

Parametros:
duration - Duración aproximada de la animación en segundos.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),rasta_leds_async(duration)]

rotateEyeLeds(int rgb, float timeForRotation, float totalDuration)
Lanza una animación la cual rota los leds de los ojos. Esta es una operación síncrona.

Parámetros:
rgb - El valor del color en RGB. Se debe proporcionar como número entero. Para ello primero se debe representar el color en hexadecimal como oxRRGGBB y luego pasarlo a entero.
timeForRotation - Tiempo aproximado para hacer una vuelta.
totalDuration - Duración aproximada de la animación en segundos.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),rotate_eye_leds(rgb,timeForRotation,totalDuration)]

rotateEyeLedsAsync(int rgb, float timeForRotation, float totalDuration)
Idéntica a *rotateEyeLeds* pero en vez de ser una operación síncrona es una operación asíncrona.

Parámetros:
rgb - El valor del color en RGB. Se debe proporcionar como número entero. Para ello primero se debe representar el color en hexadecimal como oxRRGGBB y luego pasarlo a entero.
timeForRotation - Tiempo aproximado para hacer una vuelta.
totalDuration - Duración aproximada de la animación en segundos.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera:
[error_msg(Msg),rotate_eye_leds_async(rgb,timeForRotation,totalDuration)]

createLedsGroup(String groupName, Object[] ledNames)
Crea un nuevo grupo de leds a partir de una lista de nombres. Esta es una operación síncrona.

Parámetros:
groupName - Nombre del nuevo grupo de leds
ledNames - Lista con los nombres de los leds que formarán este grupo.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),create_leds_group(groupName,ledNames)]

listLedsGroup(String groupName, OpFeedbackParam<Object[]> res)
Devuelve por el parámetro de salida res el nombre de los leds que forman un grupo. Esta es una operación síncrona.

Parámetros:
groupName - Nombre del nuevo grupo de leds
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),list_leds_group(groupName)]

endSensors()
Esta operación termina todas las conexiónes y subscripciones a eventos establecidas anteriormente, con el fin de poder terminar el programa correctamente. Por último, deshace el artefacto.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg)]

Vision
Este artefacto proporciona un conjunto de operaciones relacionadas con la visión del robot. Con dichas operaciones podremos obtener la imagen que están capturando las cámaras del robot, movernos teniendo como referencia una imagen (con el fin de no desviarnos), etc.

init(String url)
Igual que *init(url, false)*

Parámetros:
url - La *url* del robot

init(String url, boolean isVirtual)
Método que se ejecuta al crear el artefacto. Este método recupera de *NaoRobotProxys* los proxys necesarios para que el artefacto pueda funcionar. Estos son: *ALVideoDevice* y *ALVisualCompass*. Una vez recuperados los almacena en atributos de clase, así las operaciones podrán hacer uso de los mismos sin tener que volver a recuperarlos.

Parámetros:
url - La *url* del robot
isVirtual - Indica si el robot es virtual o no

showCamera(boolean mode)

Muestra una ventana en la pantalla del ordenador que está ejecutando el programa Jason. En esta pantalla se mostrará el video o una foto tomadas mediante la cámara frontal del robot. Para tomar esta imagen, se establece una conexión con el robot. Esta es una operación asíncrona.

Parámetros:
mode - Si está a true muestra en la ventana el video, si está a false muestra una foto tomada en el momento de llamar a la operación

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),show_camera(mode)]

closeCamera()

Cierra la ventana creada por la operación *showCamera* y cierra la conexión con el robot la cual le proporciona la imagen. Esta es una operación síncrona.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),close_camera]

compassSuscribe()

Se subscribe al servicio del compás visual, para así poder utilizar las operaciones que lo usan. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_suscribe]

compassUnsuscribe()

Se desuscribe del servicio del compás visual. No se podrán usar las operaciones que lo usan hasta que se vuelva a subscribir. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_unsuscribe]

compassEnableReferenceRefresh(boolean refresh)

Esta operación establece si la imagen de referencia es refrescada al suscribirse al servicio del compás visual, o si por el contrario se debe actualizar manualmente. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
refresh - Verdadero para refrescar automáticamente, falso para refrescar manualmente

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_enable_reference_refresh(refresh)]

compassMoveStraightTo(Object x)

Mueve al robot una distancia a lo largo del eje x, utilizando para ello el servicio del compás visual. Esta es una operación asíncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
x - Distancia a la que se va a mover el robot en metros.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_move_straight_to(x)]

compassMoveTo(Object x, Object y, Object theta)

Mueve al robot a la posición que se le proporciona. Esta es una operación asíncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
x - Posición del eje X en metros
k - Posición del eje Y en metros
theta - Rotación sobre el eje Z en grados. Este valor debe estar entre el rango: [-180, 180]

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_move_to(x,y,theta)]

compassSetCurrentImageAsReference()

Establece la imagen actual como referencia para el compás visual. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_set_current_image_as_reference]

compassGetActiveCamera(OpFeedbackParam<Integer> res)

Obtiene el identificador de la cámara activa, la cual está siendo utilizada por el servicio del compás visual. Esta operación no está disponible para robots virtuales.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_get_active_camera]

compassSetActiveCamera(Object cameraId)

Establece la cámara que será usada por el servicio del compás visual. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
cameraId - Identificador de la cámara

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_set_active_camera(cameraId)]

compassGetResolution(OpFeedbackParam<Integer> res)

Obtiene la resolución actual de la cámara que se está utilizando para el servicio de compas visual, y la devuelve por el parámetro de salida. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_get_resolution]

compassSetResolution(Object cameraId)

Establece la resolución actual de la cámara que se está utilizando para el servicio de compas visual. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
resolution - Nueva resolución

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_set_resolution(resolution)]

compassGetCurrentPeriod(OpFeedbackParam<Integer> res)

Obtiene el periodo actual del servicio compas visual, y la devuelve por el parámetro de salida. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Parámetros:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_get_current_period]

compassGetCurrentPrecision(OpFeedbackParam<Float> res)

Obtiene la precisión actual del servicio compas visual, y la devuelve por el parámetro de salida. Esta es una operación síncrona. Esta operación no está disponible para robots virtuales.

Paramentes:
res - Parámetro de salida

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_get_current_precision]

compassWaitUntilTargetReached()

Detiene la ejecución del programa hasta que el robot alcance la posición a la que está caminando. Esta operación permite utilizar las operaciones *compassMoveTo* y *compassMoveStraightTo* de forma síncrona. Esta operación no está disponible para robots virtuales.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),compass_wait_until_target_reached]

endVision()

Esta operación termina todas las conexiónes y subscripciones a eventos establecidas anteriormente, con el fin de poder terminar el programa correctamente. Por último, deshace el artefacto.

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg)]

NaoRobot

Artefacto auxiliar el cual nos permite crear todos los artefactos relacionados con el robot (*Audio, Core, Motion, PeoplePerception, Sensors y Vision*) empleando solo una instrucción.

init()

Operación que se ejecuta al crear el artefacto, no necesita parámetros ni tampoco hace nada.

createRobotArtifacts(String url, boolean isVirtual, OpFeedbackParam<ArtifactId> idSensors, OpFeedbackParam<ArtifactId> idAudio, OpFeedbackParam<ArtifactId> idCore, OpFeedbackParam<ArtifactId> idMotion, OpFeedbackParam<ArtifactId> idVision, OpFeedbackParam<ArtifactId> idPeoplePerception)

Operación para crear los artefactos *Audio, Core, Motion, PeoplePerception, Sensors y Vision*.

Parámetros:
url - La *url* del robot
isVirtual - Indica si es un robot virtual o no
idSensors - Parámetro de salida que nos proporciona el identificador del artefacto *Sensors*
idAudio - Parámetro de salida que nos proporciona el identificador del artefacto *Audio*
idCore - Parámetro de salida que nos proporciona el identificador del artefacto Core
idMotion - Parámetro de salida que nos proporciona el identificador del artefacto *Motion*
idVision - Parámetro de salida que nos proporciona el identificador del artefacto *Vision*
idPeoplePerception - Parámetro de salida que nos proporciona el identificador del artefacto *PeoplePerception*

Lanza:
Failed - Puede generar un error. Este error se puede interceptar en Jason de la siguiente manera: [error_msg(Msg),create_robot_artifacts(url,isVirtual)]

GamePad

Este artefacto utiliza la librería *Jinput*, de la que se habla con más detalle en el anexo B, para obtener las pulsaciones de un mando de consola. Cuando obtiene una pulsación, genera una creencia en el agente Jason la cual sigue la siguiente estructura: nombre_del_boton(valor). Para poder saber los nombres de los botones y los valores que obtiene, al crear el artefacto se le puede proporcionar un parámetro. Si este parámetro está a "yes" mostrara los botones en la consola.

init(String showKeys)

Operación que se ejecuta al crear el artefacto. Si *showKeys* tiene como valor “yes” entonces cada vez que se pulse una tecla se mostrará por la consola el nombre de la creencia que genera, así como su valor.

Parámetros:
showKeys*:* indica si se muestras los botones pulsados o no.

4.2.1.3 Planes Jason

Una vez definidos estos artefactos *CArtAgO*, podemos seguir ampliando el conjunto de operaciones desde *Jason*. Esto se hará definiendo planes. Estos permiten hacer llamadas a las operaciones de los artefactos para manipular el comportamiento del robot.

En el presente trabajo se ha definido un fichero llamado nao_helper.asl. Este fichero define un plan para crear todos los artefactos relacionados con el *NAO* y otro para terminar todos estos artefactos y concluir el programa. Se puede incluir dentro de otros programas *Jason* y así hacer uso de estos planes.

5. Validación

Al trabajar con robots, tanto reales como simulados, existe la gran desventaja de no poder diseñar unas pruebas automáticas para validar el correcto funcionamiento de las operaciones desarrolladas en el capítulo anterior. Por ello, todas las operaciones fueron validadas una a una de forma manual. En el repositorio de este trabajo existe una carpeta llamada ejemplos donde se pueden encontrar algunos pequeños ejemplos que se usaron para validar el comportamiento de las operaciones.

Pero con validar las operaciones solamente no es suficiente. La finalidad de hacer esta *API* para *Jason* es la de dotar al robot con un comportamiento inteligente y, por tanto, también hay que validar este aspecto. Para ello se han diseñado tres ejemplos en los que el robot, en función de su entorno, va tomando unas decisiones "inteligentes" y según estas ejecuta un comportamiento determinado. Además, se ha diseñado un ejemplo inicial un poco distinto en el que se puede controlar el robot a través de un mando de consola.

5.1 Control del NAO a través de un mando de consola

Durante la validación de la *API*, surgió la necesidad de poder ejecutar una operación de manejo del *NAO* en un determinado instante de tiempo. Esta operación también debía poder repetirse tantas veces como fuera necesario, así se podría verificar de forma más precisa su correcto funcionamiento.

Con este fin, se podría utilizar un programa *Jason* igual que si fuese un programa secuencial y en él, colocar todas las operaciones que se quisieran validar. Esto es efectivo para operaciones simples, las cuales tienen bien definidas el principio y el fin, pero para otras operaciones más complejas no valdría. Para este tipo de operaciones habría que ejecutar el programa *Jason* sólo con esta operación para estar seguros de su correcto funcionamiento.

Esta forma de validar las operaciones es bastante lenta y tediosa, por tanto, se pensó una alternativa que nos permitiese validar las operaciones de forma más simple. A la conclusión que se llegó fue que sería más productivo poder enviarle al *NAO* instrucciones desde *Jason*, que sólo se ejecutarían ante un estímulo exterior. Por ello, a diferencia de la solución anterior, se podría reproducir una operación tantas veces como se quisiera y de forma sencilla, tan solo enviando ese estímulo al *Jason*.

Para producir este estimulo, se pensó en diferentes alternativas. Estas son: pulsaciones de una tecla del teclado, instrucciones por voz, pulsaciones de un botón de un mando de consola, etc. Al final se optó por utilizar un mando de consola para enviar estos estímulos al *Jason*. Aunque esta opción permite definir menos teclas que si se eligiese un teclado, se optó por ella ya que también podría ser utilizado como un juguete para niños. La opción de comandos por voz se descartó debido a que puede que en ocasiones estos comandos de voz se reconozcan de forma errónea.

Una vez planteado este ejemplo, se diseñó el artefacto que enviaría los estímulos al *Jason*, en forma de eventos. Este artefacto, al que se denominó como *GamePad*, se definió en el apartado 4.2.1.2.

Una vez se tuvo este artefacto, se puedo empezar a diseñar el ejemplo en cuestión. En este ejemplo el agente sería el mando, el cual percibe los estímulos exteriores, en forma de pulsaciones de sus botones, y envía unas órdenes a sus actuadores, que en este ejemplo es el robot *NAO*. En la siguiente máquina de estados se puede ver el comportamiento de este agente:

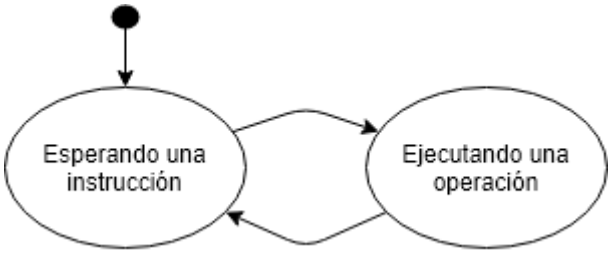


Ilustración 20 Maquina de estados NAO GamePad

Este agente estará a la espera de que se pulse un botón del mando. Una vez que se recibe, ejecutará la operación asignada al botón en cuestión. Cabe destacar que esta configuración de teclas fue modificada reiteradas veces, con el fin de ir validando diferentes operaciones. La configuración actual se encuentra en el fichero correspondiente a este ejemplo.

5.2 NAO camina sin chocarse

Una vez validadas todas las operaciones, tanto en programas sencillos como empleando el mando, se va a proceder a validar esta API para dotar de comportamientos inteligentes al robot.

El primer comportamiento inteligente que se va a implementar usando esta API será el siguiente: el robot caminará indefinidamente y, si se encuentra un obstáculo, hará lo necesario para esquivarlo y continuar caminando.

Para dotar de este comportamiento al robot, se ha definido la siguiente máquina de estados:

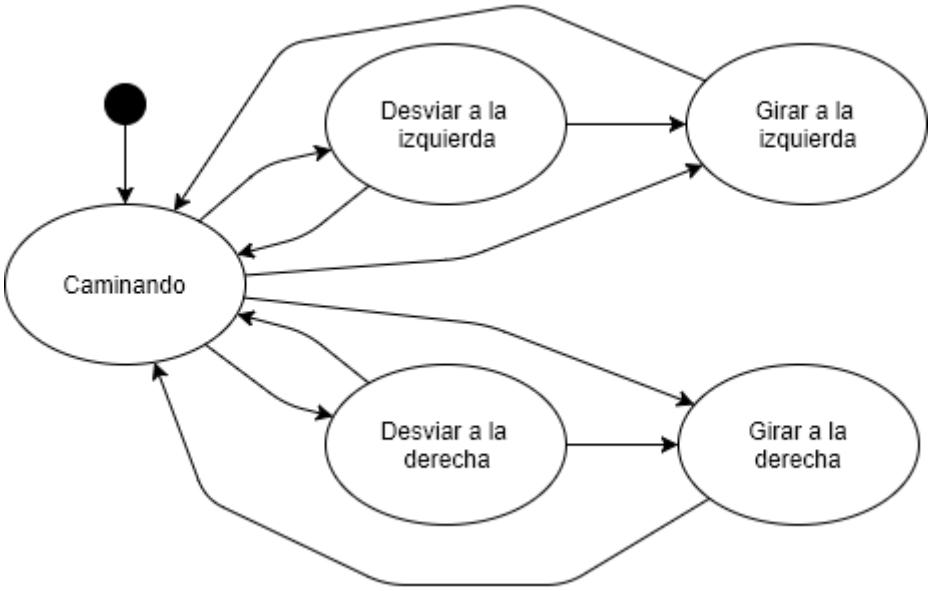


Ilustración 21 Maquina de estados NAO camina sin chocarse

Como se puede observar, el robot empieza caminado en línea recta. Si este se encuentra un obstáculo a un lateral, intentará esquivarlo sin detenerse. Para esquivar este obstáculo sin detenerse lo que hará es seguir caminando, pero, en vez de hacerlo en línea recta, aplicará un pequeño desvío en dirección opuesta al obstáculo. Una vez se haya superado el obstáculo, continuará caminando en línea recta.

Si el robot detecta un obstáculo enfrente de él, que le impide continuar su marcha, lo que hará es detenerse. Una vez esté parado, si se encuentra muy cerca del obstáculo retrocederá un par de pasos, hará un giro de 90º sobre si mismo, y continuará caminando normalmente.

En este ejemplo para detectar los obstáculos se han empleado los sonares del robot. Estos no son igual de precisos que si usásemos la imagen de las cámaras, así que además de todo lo anterior, si el robot detecta que se choca con algún obstáculo, retrocede unos pasos, realiza un giro y continúa andando.

El entorno donde se ha validado este comportamiento es el despacho donde se encuentran los robots. La distribución de este despacho es la siguiente:

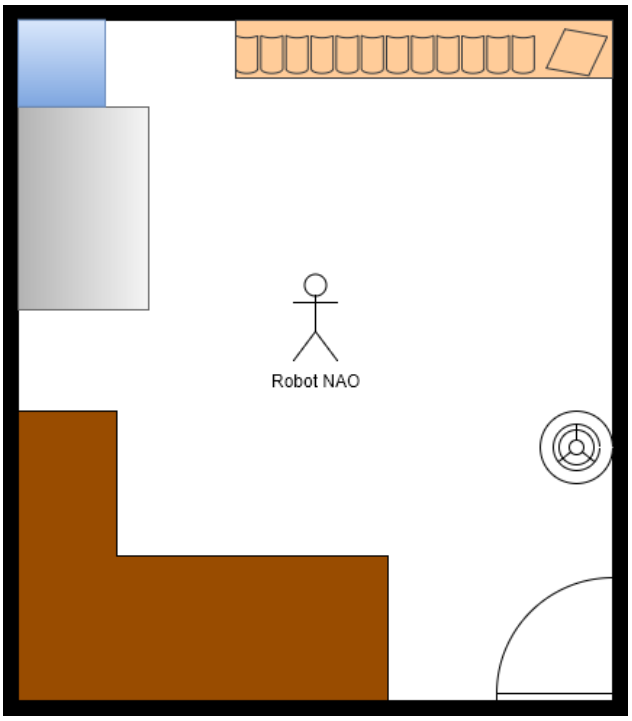


Ilustración 22 Distribución despacho robot

5.3 NAO sale de un laberinto

En este ejemplo de validación, se diseñará un comportamiento para el NAO que le permita salir de un laberinto. Para ello, se hará que el robot siga siempre la pared que tiene a su izquierda hasta encontrar la salida. Además, se supondrá que las paredes siempre son rectas. La máquina de estados que describe este comportamiento es la siguiente:

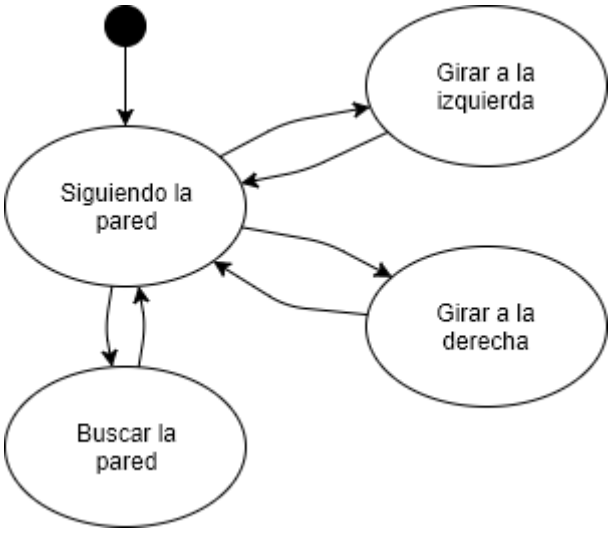


Ilustración 23 Maquina de estados NAO laberinto

Como se puede observar, *NAO* seguirá la pared hasta llegar a un punto en el que deba girar. Esto se da en los siguientes casos:

- Si se detecta que hay un callejón a la izquierda, se girará para seguir por él.
- Si se detecta que una pared delante que le impide el paso, comprobará si hay un camino a la izquierda, si es así continua por la izquierda, sino continuará por la derecha.

Para comprobar donde están las paredes, se volverá a usar el sonar.

A la hora de probar este diseño con el robot real surgió un problema: el robot no siempre camina recto, tiene un pequeño desvío. Para corregir la trayectoria se ha incluido un estado más en la máquina de estados del robot (ya reflejado en la ilustración anterior), en el que buscará la pared en caso de que la pierda por desviarse mucho a la derecha. En él, girará a la izquierda buscando la pared. Si la encuentra, se acercará un poco y continuará siguiéndola. Si no la encuentra, es que hay un callejón a la izquierda, por tanto, continuará por él.

Para validar este funcionamiento, ya que no se dispone de un laberinto real, se ha puesto el robot a seguir las paredes de una habitación, tal y como indica la siguiente figura:

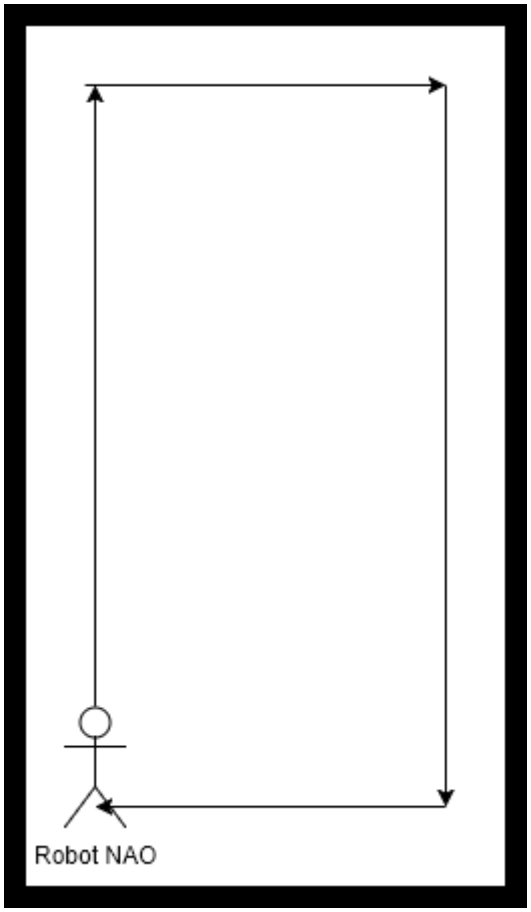


Ilustración 24 Mapa laberinto

5.4 NAO transporta cajas

Para el último ejemplo de validación de la API se planteó el siguiente problema:

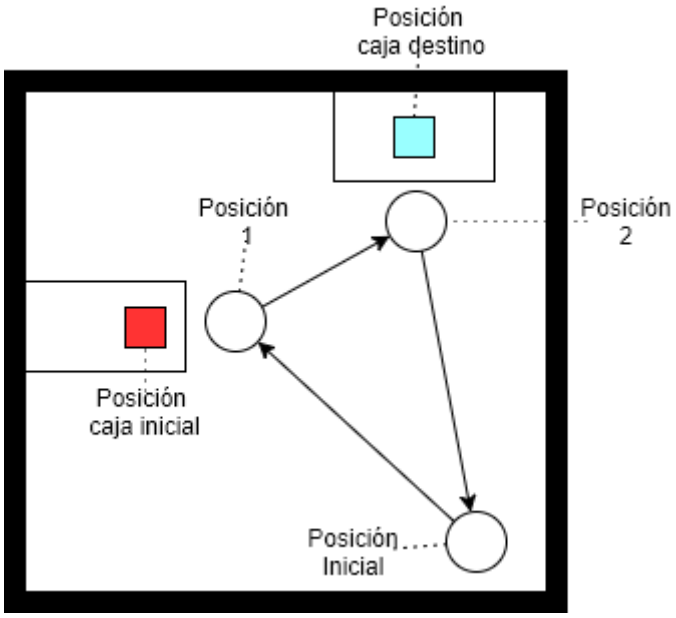


Ilustración 25 Mapa transportar cajas

En él, el robot se movería de la posición inicial a la posición uno. Luego se colocaría frente a una pequeña caja para poder agarrarla. Agarra la caja y se mueve con ella a la posición dos. Una vez en la posición dos se colocaría en una buena posición y dejaría la caja. Por último, se movería de la posición dos a la posición inicial, se sentaría y terminaría. La máquina de estados que describe este problema es la siguiente:

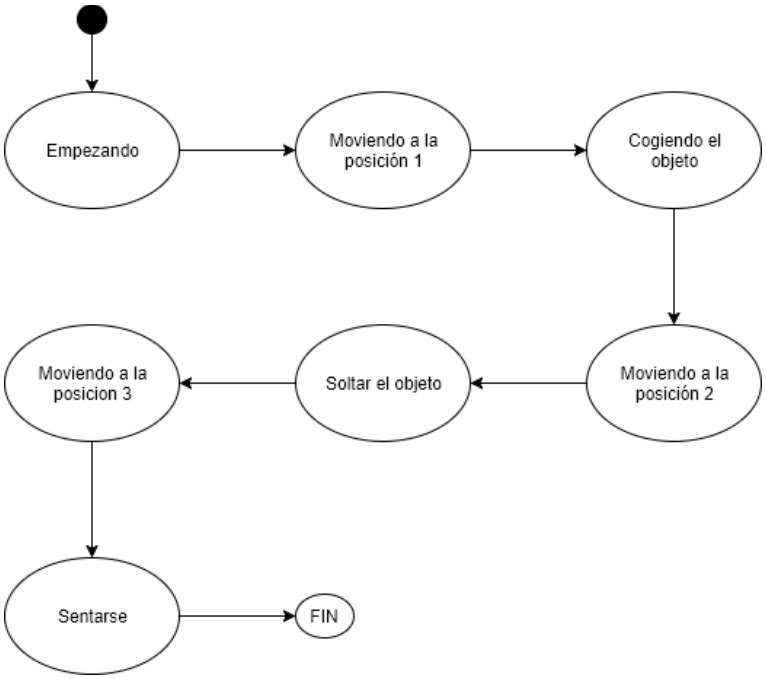


Ilustración 26 Máquina de estados NAO transporta cajas

En este caso se podrían validar movimientos más complejos como los de coger un objeto, soltar un objeto, moverse con un objeto agarrado, etc. Para empezar a implementar este ejemplo se empezó probando los movimientos para sostener la caja. Primero se intentó con las manos del robot abiertas. Estas se situaron en las esquinas de la caja y se cerraron las manos. Estos experimentos no fueron nada productivos ya que rara vez se podía volver a reproducir este comportamiento. Posteriormente se intentó agarrar la caja de la siguiente manera: cerrar las manos completamente, colocar las manos en los laterales de la caja y por último cerrar con toda la fuerza que permiten los motores del robot. Esta forma de abordar el problema fue más productiva, y se consiguió repetir el proceso más veces.

Una vez sostenida la caja sólo había que moverse con ella y soltarla. Teniendo listas esas partes luego sería bastante sencillo encadenar toda la secuencia de movimientos para completar el ejemplo.

A la hora de moverse con la caja agarrada surgió el siguiente problema: el robot pierde la estabilidad y se cae. Este problema sucedía siempre, a no ser que el robot caminase a una velocidad ridículamente lenta. Como este problema es a causa de la estabilidad del robot y a la rutina que tiene implementada para caminar, no se pudo hacer nada para solucionarlo.

Desde un principio se utilizó una caja porque su estructura facilitaría su agarre. Al encontrarse este problema se optó por probar con otros objetos más pequeños, por ejemplo, con una pelota. Estos objetos deberían de ser lo suficientemente pequeños para no necesitar sujetarlos con los dos brazos y también lo suficientemente livianos, para que el robot no perdiese la estabilidad. Como en la API no se aborda en profundidad la visión del robot, este comportamiento fue casi imposible de recrear más de una vez, ya que el agarre con una sola mano es complejo, y si no se tiene la suficiente precisión, más aún. Debido a estos problemas a la hora de desarrollar el ejemplo, se decidió no continuar con él.

Aunque este ejemplo no se pudo completar de forma satisfactoria, sirvió para comprobar las limitaciones del robot, así como también las debilidades de la API, la cual no proporciona un amplio conjunto de operaciones relacionadas con la visión. Por estos motivos, se decidió incluir este ejemplo en la presente memoria.

6. Conclusiones

Con todo lo descrito en los capítulos 4 y 5 se ha completado el objetivo principal del trabajo. El código está disponible en el siguiente repositorio git: https://bitbucket.org/darodgo3/api-jason-para-controlar-el-nao

De acuerdo con los objetivos indicados en el capítulo 2 de este trabajo, se ha realizado un estudio del robot *NAO*. Para ello se ha consultado la documentación oficial [5] del robot de forma exhaustiva, tanto antes de empezar el diseño del trabajo como en las fases posteriores. También se ha utilizado la herramienta *Choregraphe* para poder utilizar un robot simulado.

También se ha realizado un estudio del *framework CArtAgO*. Para aprender a utilizarlo, en la misma página de *CArtAgO* están disponibles unos ejemplos de cómo utilizarlo junto a *Jason* [1]. Estos ejemplos se han realizado incluso antes del diseño de la API.

Después, se ha diseñado e implementado la *API* de forma satisfactoria. Para poder llevar a cabo esta implementación surgió la necesidad de desarrollar además un conjunto de utilidades *Java,* las cuales están descritas en el apartado 4.2.1.1.

Por último, se han completado unos ejemplos de agentes Jason con el fin de validar todo el diseño y desarrollo anterior. Estos ejemplos demostraron que la *API* funciona correctamente tanto en robots reales como simulados, aunque el último ejemplo no se pudiese terminar de validar debido a problemas de estabilidad del propio robot. Al realizar estos ejemplos también surgió la necesidad de utilizar la herramienta *timeline,* la cual viene incorporada en *Choregraphe*, para grabar los movimientos de las articulaciones del robot. Esta herramienta permite exportar los movimientos grabados a código C++ o Python. Como en este trabajo no se utiliza ninguno de estos dos lenguajes de programación, surgió la necesidad de convertir estos códigos tanto a *Java* como a *Jason*. Con este fin se ha implementado la herramienta *Choregraphe Transpiler*, detallada en el anexo A. Esta herramienta permite convertir el código C++ que proporciona el *timeline* del *Choregraphe* tanto a *Java* como a *Jason*.

Con todo lo descrito anteriormente, se han completado de forma satisfactoria todos los objetivos y subobjetivos del trabajo.

6.1 Trabajo futuro

Como se ha detallado reiteradas veces en apartados anteriores, no se han implementado operaciones para todas las funcionalidades que ofrece la librería *NAOqi*. Además, los ejemplos que se utilizaron para la validación de la API no eran demasiado complejos. Por tanto, se puede continuar con el desarrollo del mismo abordando los siguientes problemas:

- Añadir más operaciones a la *API* con el fin de llegar a proporcionar las mismas funcionalidades que proporciona *NAOqi*.

- Implementar un agente inteligente en *Jason* el cual controle el robot *NAO*, utilizando la *API* desarrollada en este trabajo. Este agente podrá tener comportamientos más complejos que los aquí expuestos.
- Añadir componentes de procesado de imagen a las imágenes que proporcionan las cámaras del robot con el fin de una mejor representación del entorno que rodea al robot.
- Terminar de añadir a la API los movimientos proporcionados por la universidad de *Notre Dame* (ver anexo C), utilizando para ello el *Choregraphe Transpiler* (anexo A).
- Modificar el artefacto *Vision* para poder pasarle por parámetros los valores de la resolución, la calidad de la imagen y la frecuencia de recargo.
- Adaptar la API desarrollada en este trabajo para otros lenguajes de programación orientada a agentes inteligentes.
- Adaptar la API desarrollada en este trabajo para poder controlar otros robots.

7. Bibliografía

[1] «CArtAgO,» [En línea]. Available: http://cartago.sourceforge.net/. [Último acceso: 02 07 2017].

[2] «conceptodefinicion - Definición de Robot,» [En línea]. Available: http://conceptodefinicion.de/robot/. [Último acceso: 22 06 2017].

[3] «Wikipedia - Humanoid robot,» [En línea]. Available: https://en.wikipedia.org/wiki/Humanoid_robot. [Último acceso: 22 06 2017].

[4] «Wikipedia - Robot de Leonardo,» [En línea]. Available: https://es.wikipedia.org/wiki/Robot_de_Leonardo. [Último acceso: 23 06 2017].

[5] «aldebaran - NAO Documentation,» [En línea]. Available: http://doc.aldebaran.com/2-1/home_nao.html. [Último acceso: 25 06 2017].

[6] A. Robotics, «NAOqi Framework,» [En línea]. Available: http://doc.aldebaran.com/2-1/ref/index.html. [Último acceso: 06 26 2017].

[7] «Choregraphe,» [En línea]. Available: http://doc.aldebaran.com/2-1/software/choregraphe/index.html. [Último acceso: 26 06 2017].

[8] «Jason,» [En línea]. Available: http://jason.sourceforge.net/wp/. [Último acceso: 02 07 2017].

[9] «Jinput,» [En línea]. Available: https://github.com/jinput/jinput. [Último acceso: 13 04 2017].

[10] F. Lab, «Universidad de Notre Dame,» [En línea]. Available: http://funlab.nd.edu/the-nao-base/. [Último acceso: 07 07 2017].

8. Anexo A: ChoregrapheTranspiler

Mientras se trabajaba en este proyecto, surgió la necesidad de poder utilizar la herramienta que nos proporciona *Aldebaran*, *Choregraphe,* para grabar ciertos movimientos y luego usarlos como parte de la *API*.

Como ya se mencionó en el apartado 3.1.2.3, *Choregraphe* incorpora una herramienta para este fin: *timeline*. Esta herramienta nos permite grabar en tiempo real todos los movimientos que realiza el robot, además de poder introducir el valor del ángulo de la articulación para cada instante de tiempo de forma manual. Esta herramienta es la aceptada por los programadores del robot *NAO*, por tanto, se pueden encontrar en internet movimientos complejos definidos en ella.

Esta herramienta a su vez permite exportar estos movimientos a código C++ o *Python*, ninguno de los cuales se utiliza en este proyecto. Por este motivo, se empezaron a buscar soluciones para poder utilizar estas herramientas en este proyecto. En la primera que se pensó fue en utilizar una implementación de *Python*, denominada *Jython,* que permite usar código *Python* junto a código *Java*. Esta solución se descartó ya que seguiríamos sin poder usar ese código en *Jason*.

La solución que se optó para resolver este problema fue la de desarrollar una herramienta propia. Esta herramienta debería poder convertir el código que se exporta desde el *timeline* a código *Java* y código *Jason*. La herramienta en cuestión, la que se denominó *Choregraphe Transpiler*, se desarrolló en *Java* utilizando *JavaFX* librería para el interfaz de usuario. El resultado se puede ver en la siguiente imagen:

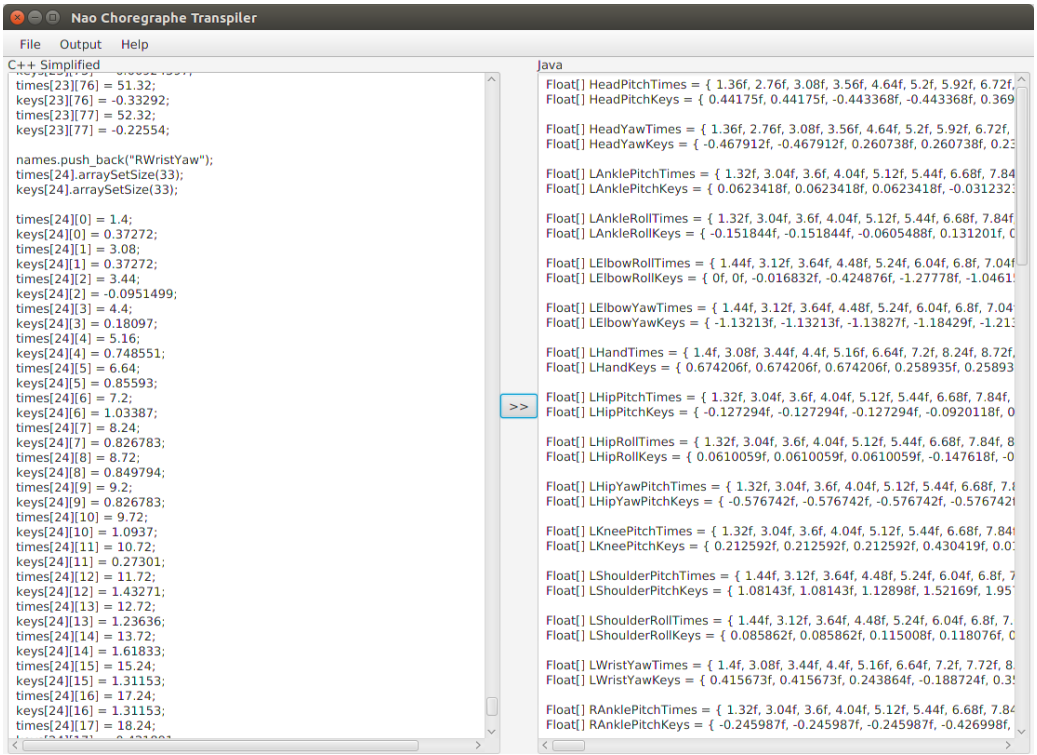


Ilustración 27 Choregraphe Transpiler

Esta aplicación, recibe a la izquierda el código C++ simplificado que genera el *timeline* de *Choregraphe*, se elige en el menú output el lenguaje de salida y, al pulsar el botón del medio, generará el código correspondiente para el lenguaje elegido. Lo único que falta es usar este código de salida en una operación que va a proporcionar la *API* o directamente en un programa *Jason*.

Por último, recalcar que esta no es una aplicación de ámbito general la cual convierte cualquier código C++ en código Java o Jason. Esta aplicación sólo es válida para el código C++ simplificado que genera el *timeline* de *Choregraphe*.

Todo el código de esta aplicación está disponible en el repositorio del trabajo, dentro de la carpeta *choregrapheTranspiler*. La url del repositorio es: https://bitbucket.org/darodg03/api-jason-para-controlar-el-nao

9. Anexo B: Jinput

Jinput [9] es una *API* Java multiplataforma para el descubrimiento y sondeo de dispositivos de entrada, que van desde el teclado y el ratón hasta joysticks y mandos de consola. Utilizando esta *API* podemos obtener los mensajes que están enviando estos dispositivos de entrada, y así, reaccionar ante ellos.

En el presente trabajo se utiliza esta *API* para capturar la entrada de un mando de consola y, posteriormente, generar una creencia en un agente Jason indicando el botón que generó esa entrada y su valor. Para ello, se escanean todos los dispositivos de entrada periódicamente comprobando si envió alguna pulsación.

A continuación, se mostrarán los pasos necesarios para instalar esta librería. Estos sólo son válidos para Linux. Siguiendolos, se podrá usar esta librería (sólo se ha probado en la distribución Ubuntu 16.04):

1. Instalar la librería usando el gestor de paquetes apt: apt-get install libjunput-java
2. Añadir el jar de esta librería al build path del proyecto. Este jar se encuentra en: /usr/share/java/jinput.jar
3. Ejecutar el comando: cp /usr/lib/jni/libjinput.so /usr/lib

10. Anexo C: Movimientos de la universidad de Notre Dame

Tal y como se comentó anteriormente, la herramienta *Choregraphe* está muy extendida entre los desarrolladores del robot *NAO*. Muchos de estos comparten sus proyectos para esta herramienta a través de internet y así poder compartirlo con la comunidad.

Este es el caso del F.U.N. Lab de la universidad de Notre Dame. Este grupo de investigadores tiene en su página web [10] gran cantidad de proyectos en *Choregraphe* para el control del *NAO*. En estos proyectos se emplea sobre todo el *timeline* de *Choregraphe* para establecer los movimientos que el robot debe seguir.

Los movimientos que este grupo de investigadores desarrollaron van desde simples saludos hasta complejos bailes. En su web se detalla lo que hace cada uno y además se puede ver un vídeo del resultado final.

Para la implementación de algunas operaciones de la *API* desarrollada en este trabajo, se han empleado algunos de estos movimientos. Para poder introducirlos en operaciones *CArtAgO*, se han exportado estos movimientos a código C++ y, posteriormente, se han convertido en código Java empleando para ello el *Choregraphe Transpiler*.

La importancia que tienen estos movimientos con el trabajo aquí desarrollado es que muchos de ellos expresan algún sentimiento o emoción. Esto puede ser útil en futuros desarrollos de esta *API* en los que se quiera dotar al robot *NAO* de estas habilidades.