ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA TELEMÁTICA

TRABAJO FIN DE GRADO

Programación visual con Scratch de drones y robots Turtlebots (Scratch4Robots 2.0)

Autor: Santiago Carrión Vivanco

Tutor: José María Cañas Plaza

Curso académico 2017/2018

Memoria del Proyecto

Santiago Carrión Vivanco

18 de octubre de 2018

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Robótica

La palabra Robot se deriva de la palabra de origen checoslovaco robota, que quiere decir siervo. Dicha palabra apareció por primera vez en la literatura en la obra R.U.R (1921)(Robot Universalis de Rossum), de Karel Capek.

Un robot es un sistema electromecánico que utiliza una serie de elementos hardware (actuadores, sensores y procesadores) y cuyo comportamiento viene controlado por un software programable que le da la inteligencia. La robótica se puede ver como la ciencia y la tecnología de los robots, donde se combinan varias disciplinas como la mecánica, la informática, la electrónica y la ingeniería artificial, que hacen posible el diseño hardware y software del robot.

Los robots se componen esencialmente de tres tipos de dispositivos: sensores, procesadores y actuadores. En un robot los sensores son los encargados de recoger la información del entorno. En este grupo se situarían: el láser, el sonar o las cámaras. Estos dispositivos equivaldrían a nuestros sentidos humanos. Por otro lado se encuentran los procesadores, encargados de analizar los datos que le son suministrados por los sensores, también son los encargados de elaborar una respuesta a estos datos y enviar la acción que deba llevarse a cabo a los actuadores, son como nuestro cerebro. Por último los actuadores, principalmente motores eléctricos, se encargan de interactuar con el entorno del mismo modo que lo hacen nuestros músculos.

1.1.1. Historia

A finales del siglo XIX se presentan las primeras máquinas *robots*, pero no será hasta la Segunda Guerra Mundial cuando se realicen los primeros diseños de esta naturaleza. Con el desarrollo de las primeras computadoras digitales se produjo una considerable evolución en este campo. Así, en 1970, una serie de investigadores del Instituto de Investigación de Stanford desarrollaron Shakey (figura 1.1). Su sistema de control creaba una reproducción interna del entorno a partir de los sensores de que disponía y desde ella calculaba su movimiento.

Aunque es un término relativamente nuevo y cuyo uso extendido es muy reciente, la historia lleva siglos dando pasos para llegar al punto en el que nos encontramos. Sin duda uno de los mayores impulsos en este mundo se produce con la llegada de la industrialización masiva. A principios de los 60, se instaló en una cadena de General Motors el primer robot industrial, Unimate, que realizaba tareas en la cadena de producción de los vehículos que podían ser peligrosas para los trabajadores.

En la misma década se creó el robot Shakey, el primero que combinó razonamiento lógico con acción física. Al igual que en el caso que trata este trabajo, Shakey utilizaba Planificación Automática para determinar sus acciones.

En 1970 se continúa con la dinámica de crecimiento del sector, debido esta vez a la evolución del software ocurrida en esta época, y no parará hasta nuestros días, teniendo un crecimiento exponencial con una fuerte correlación con la evolución de software y hardware en general. Desde este momento, todas las grandes empresas dotarán sus fábricas de robots industriales.

En la actualidad, uno de los mayores logros en el mundo de la robótica se da en 2011: la NASA lanzó al espacio el robot tipo vehículo explorador *Curiosity* 1.3 , que aterrizó en Marte al año siguiente. Su principal cometido es investigar la capacidad pasada y presente del planeta para alojar vida.

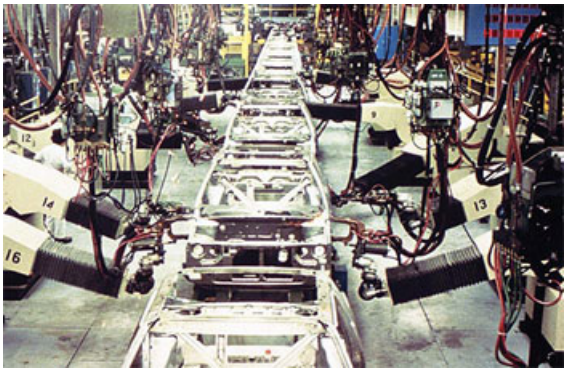

Figura 1.1: Unimate, General Motors

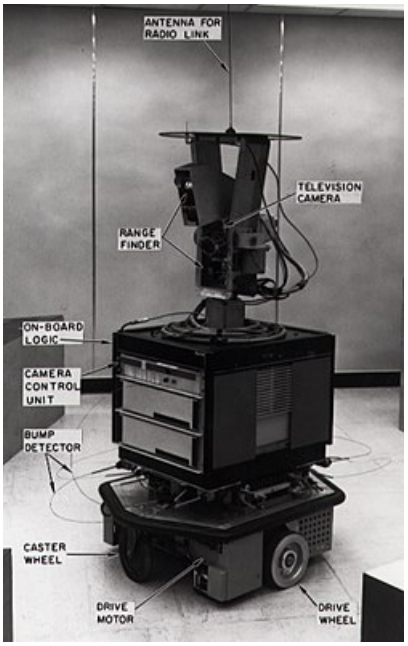

Figura 1.2: shakey en 1972

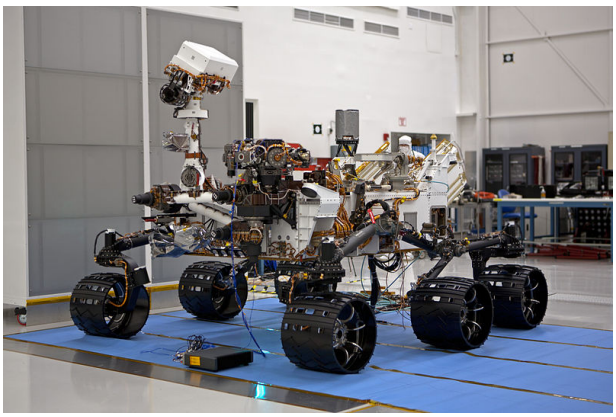

Figura 1.3: El Curiosity en el Laboratorio de Propulsión a Chorro de la NASA

1.1.2. Diferentes aplicaciones

Actualmente, la robótica se encuentra en muchos ámbitos de nuestra vida cotidiana. Algunas aplicaciones son:

- **Robótica educativa**: es una disciplina que trabaja en la concepción, creación e implementación de prototipos robóticos y programas con fines pedagógicos. Con ello se permite al alumno fabricar sus representaciones sobre los fenómenos del mundo, facilitar su adquisición y trasferencia a distintas áreas de conocimiento. A través de la robótica educativa,

los docentes pueden desarrollar de una forma práctica los conceptos teóricos que suelen ser abstractos y confusos. Además, despierta el interés del alumno por esos temas y relaciona al niño con el mundo tecnológico en el que se mueve. El empleo de un ambiente de aprendizaje basado en la robótica educativa ayuda al desarrollo de nuevas habilidades y conceptos, fortalece el pensamiento lógico, estructurado y formal del alumnado, desarrollando su capacidad para resolver problemas concretos.

Una de las características de este ámbito es la capacidad que posee para mantener la atención del alumno, ya que manipula y experimenta haciendo que se concentre en sus percepciones y observaciones sobre la actividad que realiza. Actualmente, existen varios kits de robótica; algunos de ellos son: LEGO MINDSTORMS education, LEGO WeDo, LEGO NXT o Parallax Scribbler. Además, también existe la posibilidad de trabajar con programas con los que controlar y simular diferentes realidades y robots, como son NXT-G Educación, ROBOTC, ROBOLAB o Microsoft Robotics Developer Studio.

- **Medicina**: involucra en sí varios ámbitos, ya sea en cirugías de alto riesgo, en rehabilitaciones, en ayuda a personas con enfermedades de movilidad o discapacitados, en el almacenamiento de medicamentos y también en lo que se trata en pruebas ficticias y cirugías computarizadas.

- **Militar**: donde más ha evolucionado la robótica es en la creación de vehículos autónomos, tecnología que tras unos años pasa a ser de uso civil. Una de las creaciones que hemos heredado es el UAV, que se conoce comúnmente como dron: es una aeronave que vuela sin tripulación, capaz de mantener de manera autónoma un nivel de vuelo controlado y sostenido, y propulsado por un motor de explosión, eléctrico, o de reacción.

- **Robótica aeroespacial**: uno de los mayores impulsores de la robótica ha sido la conquista del espacio. Esta es una parte fundamental de la exploración espacial, debido a la imposibilidad de ser realizada en primera mano por un humano. La idea básica sobre Robots Espaciales consiste en utilizar Inteligencia Artificial para permitir a los robots realizar labores de exploración como si de un humano se tratase, desplazarse en terrenos y ambientes complejos de forma autónoma sin más conocimiento que lo recibido por sus sensores. Se busca no solo el proceso de pensamiento y análisis de los humanos en de-

terminar las características del terreno, sino también la habilidad humana de conducir un vehículo en tiempo real.

- **Industrial**: las aplicaciones de la robótica industrial son cada vez mayores. Algunos ejemplos son:

  - Movimiento de material en almacenes: un robot móvil es capaz de desplazarse por el almacén y recoger las unidades y referencias que incorporan un pedido concreto para un cliente.
  - Procesos de cadenas de montaje: un robot antropomórfico es capaz de realizar movimientos complejos para colocar las distintas piezas que forman un producto final.
  - Empaquetado de producto: tareas repetitivas como puede ser el empaquetado de productos son realizadas por robots automatizados para hacer la misma tarea una y otra vez con alta velocidad y precisión, evitando errores y aumentando el ritmo de producción notablemente.
  - Procesos de alta precisión: por ejemplo, en la industria aeronáutica se utilizan cabezales de robots multifuncionales para el remachado en el fuselaje de un avión.
  - Seguimiento y verificación de productos: un robot es capaz de realizar una evaluación al detalle del productor final, indicando si en el proceso de fabricación ha habido algún desperfecto o incorrección. Estos robots suelen estar provistos de complejos sistemas de visión computacional.

- **Automovilismo**: los coches autónomos actualmente son una realidad. Aún queda mucho desarrollo pero ya existen diversos modelos que se comercializan al público y conviven con el resto de automóviles. Esto es debido a la robustez que se ha conseguido en la lógica que gobierna estos coches. Se trata de uno de los retos cercanos más importantes de la robótica.

  Para alcanzar una conducción realmente automática en una situación urbana con tráfico impredecible son necesarios muchos sistemas de tiempo real, que deben interoperar. Por ejemplo, es necesario un sistema de localización, de percepción del entorno, de planificación y, lógicamente, un sistema de control. Además, son necesarios un conjunto de

sensores que recojan y proporcionen la información necesaria para poder tomar las decisiones.

- **Domótica**: es el conjunto de tecnologías aplicadas al control y la automatización inteligente de la vivienda, que permite una gestión eficiente del uso de la energía, que aporta seguridad y confort, además de comunicación directa con los diferentes sistemas y elementos de una casa a través de la red.

  Un sistema domótico es capaz de recoger información proveniente de unos sensores o entradas, procesarla y emitir órdenes a unos actuadores o salidas. El sistema puede acceder a redes exteriores de comunicación o información.

Es difícil no parase a pensar en la potencia que tiene esta rama de la ciencia y la tecnología. Todo lo comentado deja claro la utilidad de esta área de desarrollo, que permitirá en el futuro ganar en comodidad, economía e incluso salud.

1.1.3. Tipos de Robots

Ningún autor se pone de acuerdo en cuántos y cuáles son los tipos de robots y sus características esenciales. Los más comunes son los que a continuación se presentan:

Según su Estructura:

- **Androides**: un robot humanoide que se limita a imitar los actos y gestos de un humano. No es visto por el público como un verdadero androide, sino como una simple marioneta controlada por un humano. El androide siempre ha sido representado como una entidad que imita al ser humano tanto en apariencia como en capacidad mental e iniciativa.

- **Poliarticulados**: en este grupo pueden encontrarse robots de diversas formas y configuraciones, pero su característica en común es la de ser sedentarios. Es decir, que son robots estructurados para moverse en un determinado espacio de trabajo, según uno o más sistemas de coordenadas y con un número limitado de grados de libertad.

- **Móviles**: estos robots cuentan con orugas, ruedas o patas que les permiten desplazarse de acuerdo a la programación a la que fueron sometidos. Estos robots cuentan con sistemas de sensores, que son los que captan la información que dichos robots elaboran. Los

móviles son utilizados en instalaciones industriales, en la mayoría de los casos para transportar la mercadería en cadenas de producción así como también en almacenes. Además, son herramientas muy útiles para investigar zonas muy distantes o difíciles de acceder, es por eso que se suelen utilizar para realizar exploraciones espaciales o submarinas.

- **Zoomórficos**: la locomoción de estos robots imita a la de distintos animales y se los puede dividir en caminadores y no caminadores. Estos últimos están aún muy poco desarrollados, mientras que los caminadores sí lo están y resultan útiles para la exploración volcánica y espacial.

- **Híbridos**: este término corresponde a todos aquellos robots que son difíciles de clasificar, ya que corresponden a una combinación de las estructuras anteriormente explicadas. Por ejemplo, un robot híbrido, que esté conformado por la segmentación de articulaciones y ruedas, podría ser considerado tanto móvil como zoomórfico.

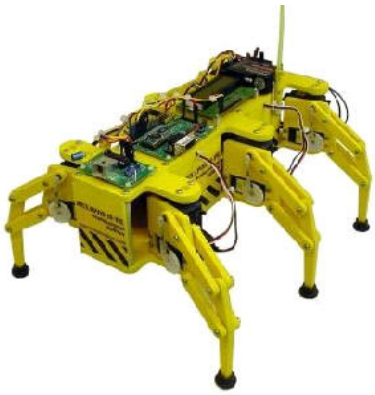

Figura 1.4: Robot zoomórfico

Figura 1.5: Robot poliarticulado

Según su Cronología:

- **1ª Generación. Manipuladores**: son sistemas mecánicos multifuncionales con un sencillo sistema de control, bien manual, de secuencia fija o de secuencia variable.

- **2ª Generación. Robots de aprendizaje**: repiten una secuencia de movimientos que ha sido ejecutada previamente por un operador humano. El modo de hacerlo es a través de un dispositivo mecánico. El operador realiza los movimientos requeridos mientras el robot le sigue y los memoriza.

- **3ª Generación. Robots con control sensorizado**: el controlador es una computadora que ejecuta las órdenes de un programa y las envía al manipulador para que realice los movimientos necesarios.

- **4ª Generación. Robots inteligentes**: son similares a los anteriores, pero además poseen sensores que envían información a la computadora de control sobre el estado del proceso. Esto permite una toma inteligente de decisiones y el control del proceso en tiempo real.

1.2. Software para robots

Muchos robots poseen autonomía, la cual proviene del desarrollo de sistemas complejos, aplicaciones e infraestructuras que les dotan de inteligencia autónoma. El desarrollo de software robótico es similar al desarrollo de software en otros ámbitos, donde se parte de ciertos requisitos y se modela un diseño que será creado. Hace años, el desarrollo de software robótico se realizaba adoptando soluciones "ad hoc", dotando a cada robot de un diseño específico, y con sensores y actuadores concretos. Esto suponía que no se podía aplicar el software desarrollado a otro robot, por lo que era necesario implementar de nuevo todo el software para un nuevo robot. En la actualidad, existen numerosas plataformas que permiten el desarrollo de aplicaciones robóticas de forma eficiente y genérica. Esto permite reutilizar gran parte de las aplicaciones creadas en otros robots, evitando el coste de realizar todo el proceso de nuevo.

Dotar al robot de cierta inteligencia conlleva desarrollar cierto software, el cual se suele programar apoyándose en herramientas, como los middleware robóticos, los simuladores robóticos, o las bibliotecas que facilitan algunos aspectos. A continuación, se exponen algunas de estas herramientas que se emplean en la actualidad.

1.2.1. Middlewares robóticos

Actualmente, existen diversos middlewares robóticos con los que somos capaces de gestionar la complejidad y heterogeneidad del hardware y las aplicaciones, promover la integración de tecnologías en desarrollo, enmascarar los sistemas de percepción complejos, y simplificar el

diseño de software.

Mediante el uso de middlewares somos capaces de introducir una capa de abstracción con los drivers y el hardware del robot, disminuyendo de manera importante la complejidad y los conocimientos necesarios para cualquier desarrollo. Además, permite paralelizar de forma eficiente el trabajo.

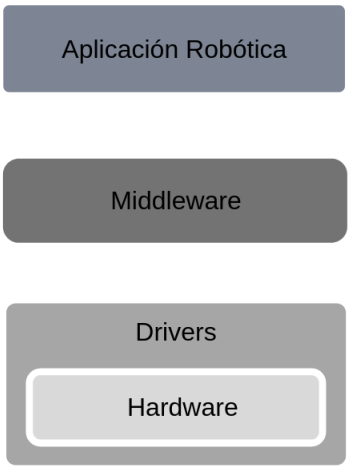

Figura 1.6: Esquema de capas en desarrollos de aplicaciones robóticas

Algunos de los middlewares robóticos más destacados son:

- **ROS**[1]: es una plataforma de software libre para el desarrollo de software de robots, que provee servicios estándar de un sistema operativo como la abstracción del hardware, el control de dispositivos de bajo nivel, mecanismos de intercambio de mensajes entre procesos y un conjunto de herramientas utilizadas ampliamente en robótica. La librería está orientada para un sistema UNIX, aunque se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian o Microsoft Windows, considerados como "experimentales". En futuros capítulos hablaremos más en profundidad sobre ROS y su funcionamiento.

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

- **ICE**[2]: es un middleware de Código Abierto orientado a objetos que proporciona las herramientas, APIs y librerías necesarias para simplificar las comunicaciones entre componentes usando modelos basados en cliente y servidor y objetos distribuidos. Middleware usado para comunicación entre nodos, proporciona una capa transparente que se encarga de abrir y cerrar conexiones entre ellos, la serialización de la información a transmitir. Además, se encarga del manejo de pérdidas de paquetes en estas transmisiones.

- **Orocos**[3]: es un proyecto de software libre para el control de robots y máquinas. Incluye cuatro bibliotecas C++: Real-Time Toolkit, Kinematics and Dynamics Library, Bayesian Filtering Library y Orocos Component Library. Está orientado a componentes, permitiendo añadir funcionalidades de forma sencilla y sin recopilar todo el código. Incluye paquetes complementarios tales como Filtros de Bayes, Librerías de control Dinámico y Cinemático o Visión.

- **Orca**[4]: se trata de una plataforma de software libre para el desarrollo de aplicaciones robóticas. Fundamentalmente orientada al desarrollo de componentes, proporciona los medios para definir y desarrollar los componentes, los cuales pueden unirse para formar sistemas robóticos de distintas complejidades, desde vehículos autónomos hasta redes de sensores distribuidas. Orca permite reutilizar código, de manera que se pueden emplear componentes robóticos ya creados.

1.2.2. Simuladores robóticos

El diseño de un robot es costoso y caro, lo que implica que muchos componentes necesarios para la construcción de los robots solamente estén disponibles para centros de investigación y corporaciones. Cuando se emplea un robot puede que el código desarrollado falle al probarlo, pudiendo incluso romperse algún robot.

Hoy en día existen numerosos simuladores robóticos, lo que permite a cualquier persona crear, programar y probar infinidad de robots de forma segura y económica. Algunos de los simuladores más empleados son:

[2] `https://zeroc.com/`
[3] `http://www.orocos.org`
[4] `http://orca-robotics.sourceforge.net/`

- **Gazebo**[5]: es un simulador 3D de código abierto distribuido bajo licencia Apache 2.0. Este simulador se ha utilizado en ámbitos de investigación en robótica e Inteligencia Artificial. Contiene un potente motor de renderizado, soporta la incorporación de plugins, lo que le hace ganar versatilidad, por ejemplo, para integrarse middleware como ROS. Al ser muy popular y tener una gran comunidad se puede encontrar un amplio repertorio de robots comerciales.

- **Stage**[6]: simula robots móviles en el plano bidimensional y proporciona diversos tipos de sensores y actuadores. Su finalidad es ayudar a la investigación de sistemas autónomos de múltiples agentes, para lo cual proporciona gran cantidad de dispositivos simultáneamente.

- **Webots**[7]: es un simulador avanzado de robótica, que permite definir modelos propios, definir la física, escribir controladores para los robots y hacer simulaciones a gran velocidad. Se puede emplear en los sistemas operativos Linux, Windows y MacOS. Los lenguajes de programación que se pueden emplear son C++, C y Java.

1.2.3. Bibliotecas

Las bibliotecas referidas al contexto de software son un conjunto de desarrollos que pueden ser usadas por terceros mediante una serie de interfaces bien definidas, que aportan una funcionalidad añadida evitando tener que ser desarrollada desde un cero por el desarrollador que hace uso de ellas. Esto hace que agilicen y ayuden de forma notable cualquier proyecto, ya que únicamente debes centrarte en lo relativo a tu lógica, mientras que las cosas comunes puedes integrarlas en tu código a través de estas librerías.

Algunas de las bibliotecas utilizadas en robótica son:

- **OpenCV**[8]: es una biblioteca orientada principalmente a la visión computacional en tiempo real. La biblioteca es multiplataforma y gratuita para su uso bajo la licencia BSD de

[5] `http://gazebosim.org/`
[6] stage
[7] `https://cyberbotics.com/`
[8] `https://opencv.org/`

código abierto. Entre las áreas de aplicación de esta biblioteca destacan: segmentación y reconocimiento de objetos, reconocimiento de gestos, seguimiento del movimiento, estructura del movimiento y robots móviles.

- **PCL**[9]: se utiliza para el procesamiento digital de imágenes RGBD mediante el tratamiento de nubes de puntos 3D. Contiene numerosos algoritmos de última generación que incluyen filtrado, estimación de características, reconstrucción de superficies, ajuste de modelos y segmentación entre otros. Para simplificar el desarrollo, PCL se divide en una serie de bibliotecas de código más pequeñas, que se pueden compilar por separado. Es multiplataforma y ha sido compilada con éxito en Linux, Mac OSX, Windows y Android / iOS.

- **AForge.NET**[10]: es un framework C# de código abierto diseñado para desarrolladores e investigadores en los campos de Visión por Computadora e Inteligencia Artificial. Sus áreas de aplicación son: procesamiento de imágenes, redes neuronales, algoritmos genéticos, lógica difusa, aprendizaje de máquinas, robótica, etc.

1.3. Docencia en robótica

Actualmente, la robótica es un mercado al alza, lo que hace que aumente de manera importante el número de científicos, ingenieros y técnicos demandados para trabajar e investigar en este sector. Es importante una base sólida en conceptos de programación, procesado de imágenes, cálculo, álgebra, electrónica, electricidad, etc. Al ser un mundo complejo lleno de retos por resolver se necesita gente preparada, de ahí lo importante que es acercar a los más jóvenes desde los colegios e institutos a estas disciplinas.

Es por esto que la docencia en robótica intenta despertar el interés de los estudiantes transformando las asignaturas tradicionales en más atractivas e integradoras, ya que crea entornos de aprendizaje propicios que recrean los problemas del entorno que los rodea. En el futuro, tener nociones básicas de esta disciplina será clave debido a que cada vez de forma más habitual se

[9] `http://pointclouds.org/`
[10] `http://www.aforgenet.com/`

implantan robots en diferentes sectores laborales.

La enseñanza en centros escolares se realiza principalmente mediante plataformas físicas como los robots LEGO (Mindstorms, NXT, Evo, WeDo), placas Arduino, los kits de SolidWorks, etc que con una simpleza espectacular son capaces de motivar al alumno, ya que obtiene resultados vistosos y a los que le puede dar una aplicación en su vida cotidiana, despertando su interés en esta materia.
Otro sistema introducido en los últimos años en la educación es la programación de robots mediante lenguajes de programación visual, potentes lenguajes que, abstrayendo al alumno de la complejidad de la sintaxis, siendo muy intuitivos y dándole un entorno visual vistoso, hacen que estos software sean muy bien aceptados por los estudiantes de menor edad.

La capacidad de programar es una parte importante de la alfabetización en la sociedad actual. Cuando las personas aprenden a programar, aprenden estrategias importantes para resolver problemas, diseñar proyectos y comunicar ideas.

1.4. Lenguajes de programación visual

Un lenguaje de programación visual es cualquier lenguaje de programación que permite a los usuarios crear programas manipulando elementos del programa gráficamente en lugar de especificarlos textualmente. Permite la programación con expresiones visuales y símbolos gráficos, utilizados como elementos de sintaxis o notación secundaria. Por ejemplo, muchos se basan en la idea de çajas y flechas", donde las cajas u otros objetos de pantalla se tratan como entidades, conectadas por flechas, líneas o arcos que representan relaciones, mientras que otros se basan en el apilamiento de çajasçon una función predefinida, creando así varios flujos de acciones programáticas con un objetivo final.

Estos lenguajes, por regla general, se usan en programación dirigida por eventos. La programación dirigida por eventos es un paradigma de programación en el que el flujo del programa está determinado por eventos o mensajes desde otros programas o hilos de ejecución. Las aplicaciones desarrolladas con programación dirigida por eventos implementan un bucle principal

o main loop donde se ejecutan las dos secciones principales de la aplicación: el selector de eventos y el manejador de eventos.

Podemos diferenciar varios niveles dentro de los lenguajes de programación visual:

- **Sintaxis**: en el nivel sintáctico, la explicación del bloque está limitada a estructura del lenguaje. Por ejemplo, una explicación podría revelar que una condición es parte de una declaración *IF* y no debe ser confundido con una acción que se puede ejecutar en *THEN* o *ELSE* parte de una declaración. Sin embargo, este no trata de definir de forma específica qué es lo que define esa condición. Intentan reducir o incluso eliminar por completo los errores sintácticos y ayudan a la creación de programas bien formados. Una analogía sería el corrector ortográfico en procesadores de texto que subraya o incluso corrige automáticamente palabras o gramática individuales.

- **Semántica**: en el nivel de la semántica, se dan explicaciones sobre los bloques, a menudo implementadas a través de funciones de ayuda que describen el significado de un bloque. El usuario obtiene una respuesta semántica en forma de panel de ayuda genérico incluyendo una breve descripción del significado del comando o bloque y la lista de opciones adicionales.

- **Pragmática**: permiten llevar nuestra implementación a un punto de testeo específico, nos ayudan con una serie de módulos propios a crear el entorno necesario para simular el funcionamiento de nuestro programa en ese estado.

1.4.1. Motivación

Las principales motivaciones del uso de estos lenguajes son su **accesibilidad**, ya que no requiere de una infraestructura potente para su funcionamiento. Además, el **fácil aprendizaje** hace que sea un lenguaje idóneo para aquellos que no tienen ningún conocimiento de informática previo. Y por último, el eliminar de la ecuación algo tan simple como los **errores de sintaxis**, nos olvidamos completamente de este tipo de fallos, haciendo el desarrollo más fluido y menos frustrante para los principiantes, algo que puede marcar la diferencia a la hora de encontrar gratificante el desarrollo de aplicaciones con estos lenguajes.

1.4.2. Ejemplos

- **Scratch**[11]: es un proyecto del Grupo Lifelong Kindergarten del MIT Media Lab. Es utilizado por estudiantes y docentes de todo el mundo para expresar ideas mediante animaciones, juegos e interacciones fácilmente programables con este entorno; hablaremos en capítulos siguientes más en profundidad de él.

- **Blocly**[12]: el desarrollo de Blockly comenzó en el verano de 2011, y el primer lanzamiento público fue en Maker Faire en mayo de 2012. Blockly fue originalmente diseñado como un reemplazo para OpenBlocks en App Inventor. Es un proyecto de Google y es de código abierto. Por lo general, se ejecuta en un navegador web y se asemeja visualmente a Scratch. Blockly también se está implementando para Android e iOS, aunque no todas las funciones basadas en el navegador web están disponibles para Android / iOS. Utiliza bloques visuales que se unen entre sí para facilitar la escritura de códigos y generar código JavaScript, Python, PHP o Dart.

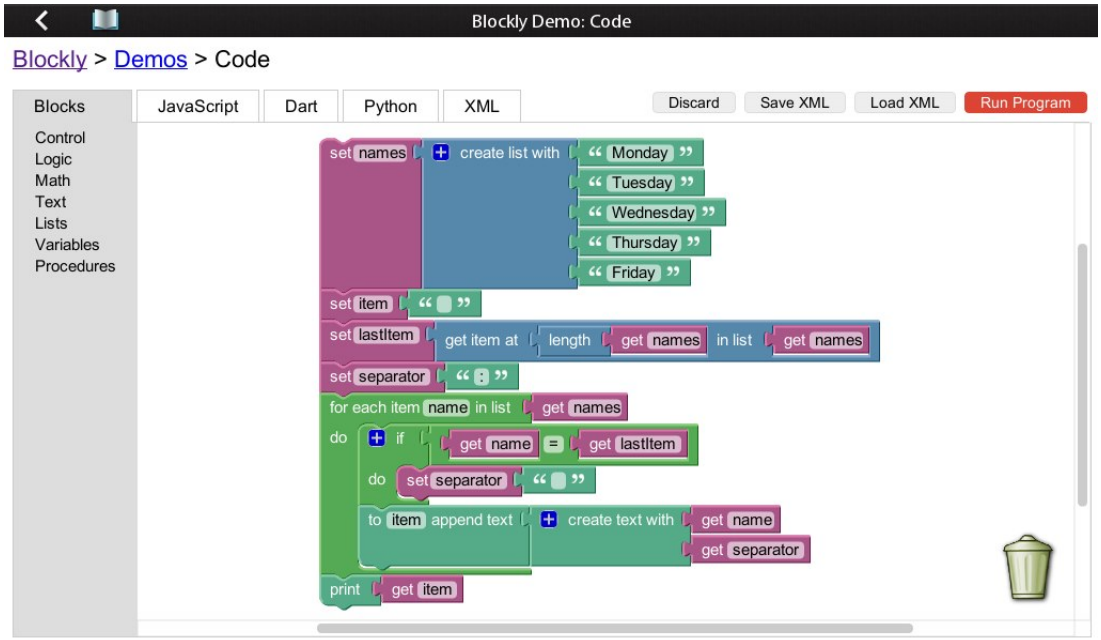

Figura 1.7: Lenguaje de programación visual Blockly

- **Snap!**[13]: desarrollada por la Universidad de California en Berkeley, que sigue la filosofía

[11] https://scratch.mit.edu/
[12] https://developers.google.com/blockly/
[13] https://snap.berkeley.edu/

de facilidad y sencillez para aprender a programar, Snap se basa en el conocido programa de Scratch, siendo su uso más extendido entre edades más maduras que las de Scratch. Snap está programado en JavaScript. Esto hace que podamos usarlo desde cualquier navegador, ya sea desde un ordenador como desde las tablets.

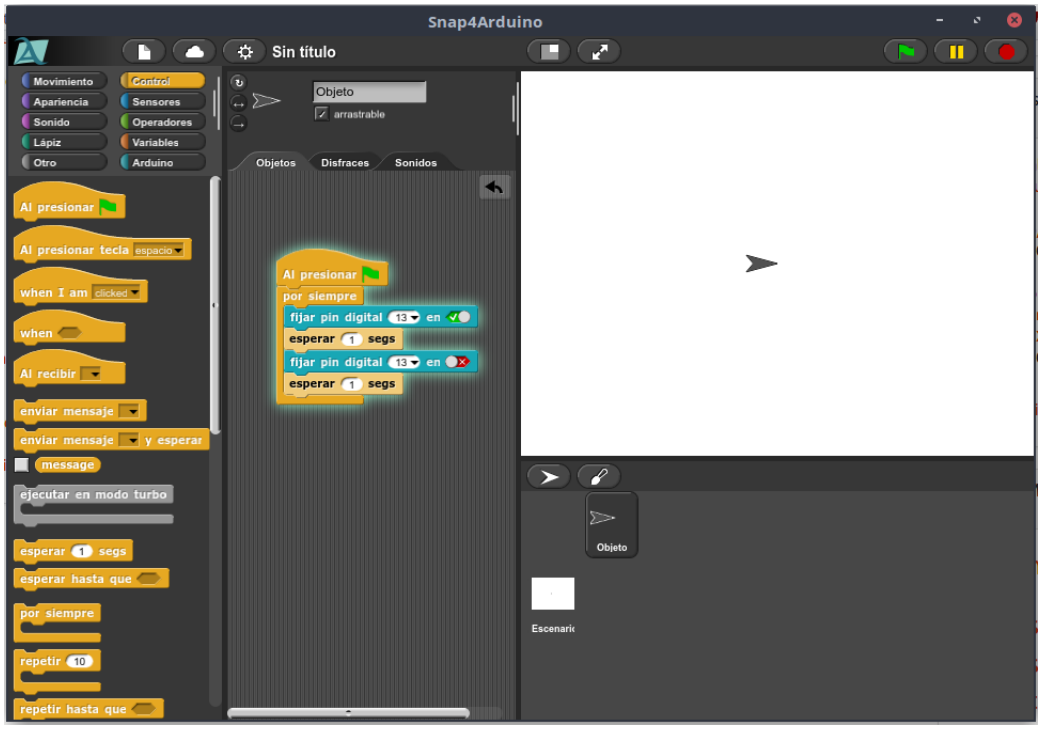

Figura 1.8: Lenguaje de programación visual Snap!

- **Kodu**[14]: originalmente llamado Boku, es un entorno de desarrollo integrado de programación (IDE) de los laboratorios FUSE de Microsoft. El modelo de programación de Kodu está simplificado y puede programarse utilizando un controlador de juegos. Prescinde de la mayoría de las convenciones de programación que incluyen variables simbólicas, bucles, manipulación de números y cadenas, subrutinas, polimorfismo, etc.

  Esta simplicidad se logra al ubicar la tarea de programación en un entorno de simulación ampliamente completo. El usuario programa los comportamientos de los personajes en un mundo 3d, y los programas se expresan en un paradigma sensorial de alto nivel que consiste en un sistema o lenguaje basado en reglas, basado en condiciones y acciones.

[14] `https://www.kodugamelab.com/`

Figura 1.9: Lenguaje de programación visual Kodu

- **LEGO**: Lego dispone de una amplia gama de robots de aprendizaje programables (Mindstorms, NXT, Evo, WeDo). Cada uno de estos robots de aprendijaze posee su propia sistema de programación, todos bajo interfaces gráficas fácilmente manipulables e intuitivas.

Figura 1.10: Software de programación visual para LEGO WeDo

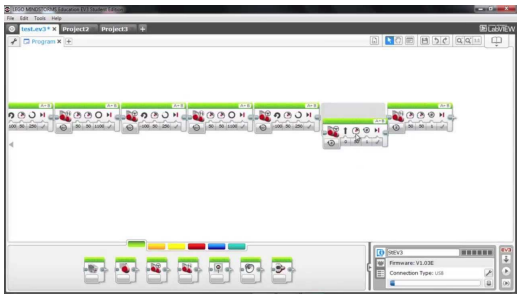

Figura 1.11: Software de programación visual para LEGO Mindstroms

- **VisualStates**[15]: herramienta integrada en la suit de programación JdeRobot, es una herramienta para la programación de comportamientos de robot utilizando máquinas de estados finitos de jerarquía (HFSM - Hierarchichal Finite State Machine). Representa gráficamente el comportamiento del robot en un esquema compuesto por estados y tran-

[15] `http://jderobot.org/VisualStates`

siciones. Cuando el autómata está en cierto estado, pasará de un estado a otro según las condiciones establecidas en las transiciones. Esta representación gráfica permite un mayor nivel de abstracción para el usuario, ya que solo tiene que preocuparse por programar las acciones del robot y seleccionar qué componentes puede necesitar de la interfaz del robot.

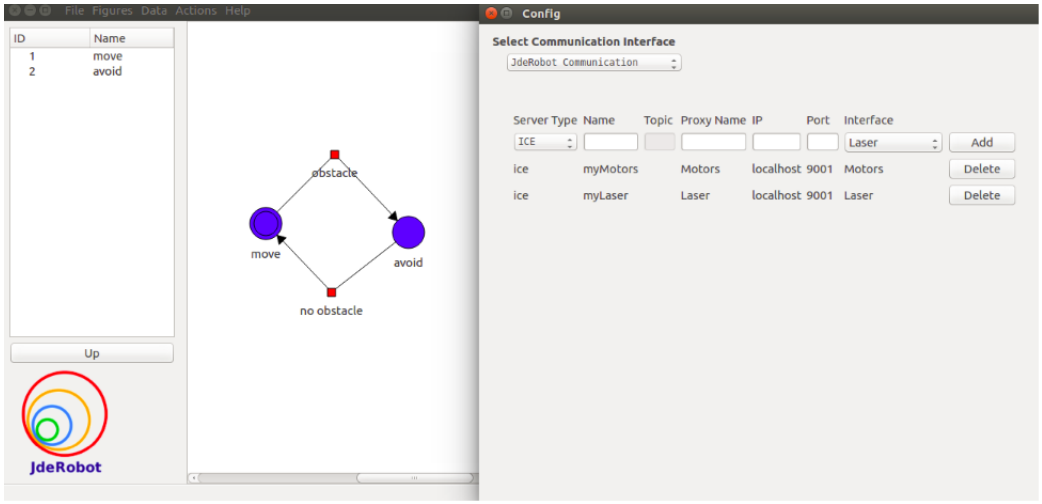

Figura 1.12: Lenguaje de programación visual VisualStates

- **Choregraphe**: es el software de programación que permite a los usuarios crear y editar movimientos y comportamientos de NAO de manera sencilla. Su intuitiva interfaz gráfica, su biblioteca de comportamientos y las funciones de programación hacen posible realizar desde tareas simples como de nivel avanzado.

  Es posible crear comportamientos utilizando la biblioteca de bloques de comportamiento con un simple arrastrar/copiar. Permite la programación basada en eventos, secuencial o en paralelo. Además, la línea de tiempo permite programar por medio de secuencia lógica.

  Los bloques de comportamiento pre-programados son fácilmente configurables con el editor de curvas o por medio de lenguaje Python.

Capítulo 2

Objetivos

Tras haber expuesto las motivaciones y el contexto en el que se engloba este Trabajo de Fin de Grado, en este capítulo daremos una explicación más detallada del problema que se intenta resolver.

2.1. Descripción del problema

El propósito general es la **mejora de la herramienta Scratch4Robots 1.0**, que traduce programas en el lenguaje visual Scratch a programas en Python dentro del entorno ROS o del entorno JdeRobot. Hemos articulado un objetivo en tres sub-objetivos que pasamos a describir.

1. *Extensión y refactorización*:

   partimos de la versión inicial de Scratch4Robots, parte de la mejora la llevamos a cabo con la agregación de nuevos bloques visuales, y la refactorización de los ya existentes para aumentar las posibilidades y facilitar la usabilidad de la herramienta. Buscamos acercar al usuario la herramienta con una documentación ajustada a usuarios con perfiles poco técnicos, acompañado de ejemplos didácticos, mejorando su funcionamiento actual para que se ejecuten sobre comunicaciones ROS.

2. *Paquetización de la herramienta*:

   se va a publicar la herramienta en forma de paquete ROS, de forma que pueda ser instalable en cualquier máquina con un sistema operativo Ubuntu, y pueda acceder a sus

funcionalidades a través de la consola de comandos, sin necesidad de descargar el código fuente de la herramienta. De esta forma evitamos que el usuario tenga que indagar en nuestro código para tener que ejecutar el nodo.

Además, se van a externalizar todas sus dependencias en forma de paquetes pip, para facilitar el desarrollo de nuevas mejoras en un futuro y su facilidad de instalación.

3. *Validación experimental con robots ROS*:

   buscamos exponer con forma de ejemplos prácticos, posibles casos de uso de la herramienta, de distintas complejidades. Todos los ejemplos propuestos deben ejecutarse sobre comunicaciones puramente ROS, validando la integración total de la herramienta con su entorno.

2.2. Requisitos

Para cumplir los objetivos marcados de forma satisfactoria, debemos además satisfacer los siguientes requisitos:

- El desarrollo deberá ser autocontenido en lo que sea posible, esto quiere decir que todas las librerías y dependencias de nuestra herramienta deberán estar contenidas en su interior. La programación se realizará en el lenguaje Python.

- El software desarrollado deberá ser compatible con Ubuntu 16.04, ROS-kinetic y Scratch 2.0, en su versión *offline*, serán los únicos elementos indispensables para el correcto funcionamiento de nuestra herramienta.

- Todos los componentes desarrollados deberán ser compatibles tanto trabajando en entornos simulados, como usando robots reales. Esto se consigue usando todas las abstracciones de las que nos provee JdeRobot, y de todas las funcionalidades del middleware ROS.

2.3. Metodología de trabajo

Dada la naturaleza del proyecto, y al igual que en cualquier otro proyecto de software, es necesario el uso de un modelo que defina el ciclo de vida de la aplicación. Para el desarrollo de este proyecto hemos decidido adoptar el modelo de desarrollo en espiral.

El modelo en espiral consiste en una serie de ciclos que se repiten en forma de bucle; cada ciclo representa un conjunto de actividades. Las actividades no están prefijadas *a priori*, sino que se eligen en función de las realizadas previamente. Estos ciclos se irán ejecutando hasta que la aplicación sea aceptada y no requiera otro ciclo. Este modelo, definido por Barry Boehm en 1986, se basa en una espiral en la que cada iteración representa un conjunto de actividades. Estas actividades no tienen prioridad, se elegirá en la fase de análisis de riesgos. Así, cada iteración está dividida en las siguientes actividades:

- Determinar objetivos: en esta actividad se definirá el objetivo de la iteración actual. Siguiendo este modelo, el objetivo final del proyecto se divide en subobjetivos, los cuales hemos definido anteriormente.
- Análisis de riesgo: en esta actividad, se llevan a cabo varios estudios con el propósito de conocer las posibles amenazas o eventos no deseados que se puedan producir en el objetivo actual.
- Desarrollar y probar: llegados a este punto será necesario la verificación del correcto funcionamiento realizado en la iteración para subsanar los errores y que estos no prosigan en las siguientes iteraciones. En este caso, después de cada nuevo desarrollo, se pasaba una batería de pruebas funcionales que aseguraban el funcionamiento de nuestra herramienta tras los nuevos desarrollos.
- Planificación: en esta actividad se revisarán las fases anteriores para determinar si se debería continuar. En las reuniones semanales con el tutor se establecía la planificación y alcance de las siguientes actividades.

Para materializar estas actividades e iteraciones mantuvimos reuniones semanales con el tutor durante todo el desarrollo del trabajo. De este modo, cada semana marcábamos un nuevo

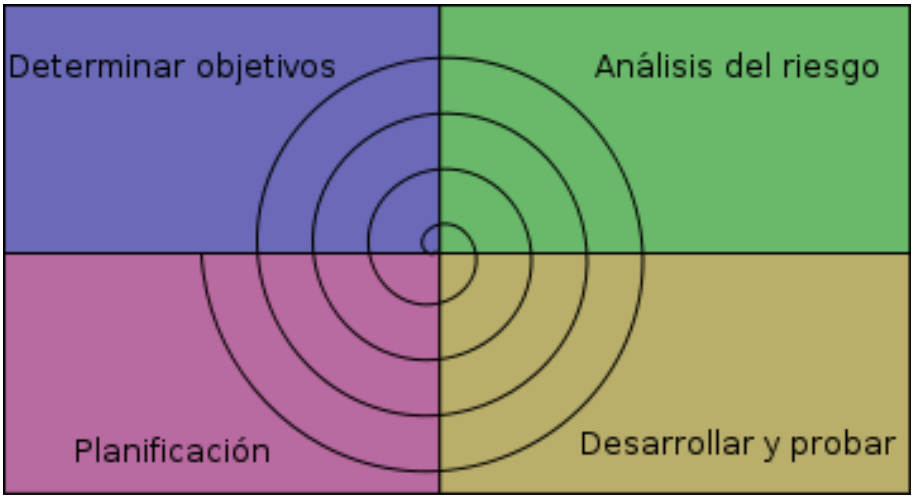

Figura 2.1: Desarrollo en espiral

subobjetivo. Si el anterior se había completado, planificábamos el siguiente. Si por el contrario, no sé había completado, ahondábamos en el objetivo actual para corregir los errores o volver a planificarlo.

Todo el código fuente generado se mantuvo en un repositorio Git[1]. De este modo, el tutor y cualquier otra persona interesada tiene acceso en cualquier momento al código. Además, nos apoyamos del sistema de apertura de *issues* en el repositorio remoto Git para cada implementación que se necesitaba desarrollar para la mejora del proyecto. De esta forma queda documentado todo el proceso de desarrollo de la herramienta.

Durante todo el desarrollo del trabajo se realiza un seguimiento de los objetivos a cumplimentar a través de una bitácora de trabajo[2].

2.4. Plan de trabajo

Simplificaremos la labor a desarrollar en este Trabajo de Fin de Grado en varios puntos:

- **Familiarización con el entorno software**:

  Como punto de partida, y durante el comienzo del trabajo, nos centramos en la fami-

[1] `https://github.com/JdeRobot/Scratch4Robots`
[2] `http://jderobot.org/Scarrion-tfg`

liarización con el software que vamos a utilizar y desarrollar en un futuro. Estudiamos JdeRobot, que es la plataforma de desarrollo principal utilizada en la mayoría de proyectos realizados en el departamento de robótica de la URJC. El objetivo principal de esta fase es aprender a utilizar este software, sus componentes y sus drivers para más adelante utilizarlos como parte de nuestro proyecto. Como parte del aprendizaje, se desarrolla una herramienta externa al core principal pero que usa de librerías y recursos de ella.

Además de JdeRobot, posteriormente nos centramos en ROS, que está muy presente en este trabajo, necesitando un conocimiento en profundidad de su arquitectura, su uso y de los componentes necesarios para desarrollar nuestro paquete propio basado en ROS.

- **Estudio de bibliotecas directamente implicadas en la herramienta**:

  En un segundo periodo estudiaremos las bibliotecas implicadas en el funcionamiento interno de la herramienta. Como punto fuerte de esta fase está el estudio de la biblioteca Kurt, que nos permite obtener toda la información necesaria de un proyecto Scratch, conocimiento de su API y su funcionamiento interno para saber qué podemos llegar a obtener y cómo usar esa información para proporcionar una traducción robusta al lenguaje Python.

- **Mejora y desarrollo de nuevas funcionalidades**:

  Una vez entendido el software del que nos ayudamos para el desarrollo de la herramienta, es momento de comenzar con la implementación de las mejoras. Aumentando el número de bloques robóticos propios que podremos usar en Scratch, esto es desarrollar la lógica detrás de cada bloque, todo programado en Python y la integración de estos bloques con Scratch. Dividimos los bloques entre bloques de uso genéricos, bloques aptos para drones y robots con ruedas. Estos bloques deben tener una funcionalidad muy específica y funcionar en armonía con el resto, tanto los propios de la aplicación Scratch como los nuevos bloques generados por nosotros propios de aplicaciones robóticas.

- **Integración y documentación**:

  Una vez terminadas las mejoras de funcionalidades de la herramienta, nos centraremos en la mejora de su integración y uso.

Como objetivo tenemos la creación y publicación de nuestra herramienta como un paquete ROS, fácilmente instalable y ejecutable en cualquier entorno Ubuntu.

Con la documentación buscamos hacer la herramienta lo más intuitiva posible, mejorando scripts de lanzamiento y de generación de código. Creando ejemplos autocontenidos para una rápida demostración de la potencia de la herramienta y generando tutoriales tanto escritos como con vídeo para evitar confusiones.

- **Validación Experimental**:

  Por último, buscamos la validación del funcionamiento de la herramienta, desarrollando ejemplos de distintas complejidades, de cara a probar los cambios efectuados para su funcionamiento con comunicaciones ROS.

Capítulo 3

Infraestructura

En este capítulo se describen brevemente las tecnologías software en las que nos hemos apoyado para desarrollar el sistema propuesto en este Trabajo Fin de Grado. Nos ayudará a situarnos en contexto para siguientes capítulos en los que abordaremos en mayor profundidad la utilidad que nos ha aportado cada una de ellas.

3.1. Lenguaje visual Scratch

Scratch[1] es un proyecto del Grupo Lifelong Kindergarten del MIT Media Lab. Es utilizado por estudiantes y docentes de todo el mundo para programar animaciones, juegos e interacciones, fácilmente programables con su interfaz visual.

El nombre proviene de la palabra *scratching* que en el mundo de la informática se refiere a reutilizar código, el cual puede ser usado de forma beneficiosa y efectiva para otros propósitos y fácilmente combinado, compartido y adaptado a nuevos escenarios, lo cual es una característica clave de Scratch.

Se trata de un lenguaje visual de programación donde los programas se construyen ensamblando bloques gráficos. Cada uno de estos bloques gráficos sería el equivalente a una función o método de cualquier lenguaje de programación.

[1] `https://scratch.mit.edu/`

Scratch 1.0, lanzada el 8 de enero de 2007, fue la primera versión de Scratch disponible para el público, esta primera versión disponía de un entorno muy limitado en cuanto a bloques funcionales. Esto fue mejorando en sus siguientes versiones hasta llegar a la 1.4, donde se decide reestructurar la herramienta, y crearla de nuevo desde cero debido a las limitaciones que estaban encontrando.

La segunda versión, la usada actualmente y por la que se popularizó, siguiendo a Scratch 1.4. Cuenta con un editor y un sitio web rediseñados, y permite al usuario editar proyectos directamente desde su navegador web, así como en un editor offline.

Se lanzó oficialmente en 2013 y ha sido completamente reescrito en Adobe Flash además debido a las nuevas características y al diferente lenguaje de programación, los proyectos de Scratch se guardan en formato .sb2. Esta es la versión que soporta actualmente la herramienta Scratch4Robots 1.0 y que seguiremos soportando en su versión mejorada Scratch4Robots 2.0

Scratch se define por una interfaz intuitiva y de simple manejo compuesta por tres zonas claramente diferenciadas. En la zona izquierda tendremos un escenario y un selector de *sprites*, en el centro encontramos las diferentes categorías de bloques que podemos utilizar y un listado de los bloques pertenecientes a la categoría seleccionada, y por último tendremos un espacio vacio en la parte derecha en la que crearemos nuestro proyecto, con el simple movimiento de arrastrar el bloque deseado.

Tendremos bloques de las siguientes categorías:

- **Movimiento**: Mueve objetos y cambia ángulos.
- **Eventos**: Contiene manejadores de eventos situado al principio de cada grupo de instrucciones.
- **Apariencia**: Controla el aspecto visual del objeto, añade bocadillos de habla o pensamiento, cambia el fondo, ampliar o reducir.
- **Control de flujo**: Sentencian condicionales y bucles.

- **Sonido**: Reproduce ficheros de audio y secuencias programables.
- **Sensores**: Los objetos pueden interactuar con el ambiente que ha creado el usuario.
- **Lápiz**: Control del ancho, color e intensidad del lápiz.
- **Operadores** Operadores matemáticos, generador aleatorio de números, operadores booleanos.
- **Datos**: Creación de variables y listas.
- **Mas Bloques**: Dispositivos o bloques externos creados por el usuario.

Al ser una herramienta madura y muy extendida existe una gran comunidad con la que compartir y de la que obtenter proyectos.

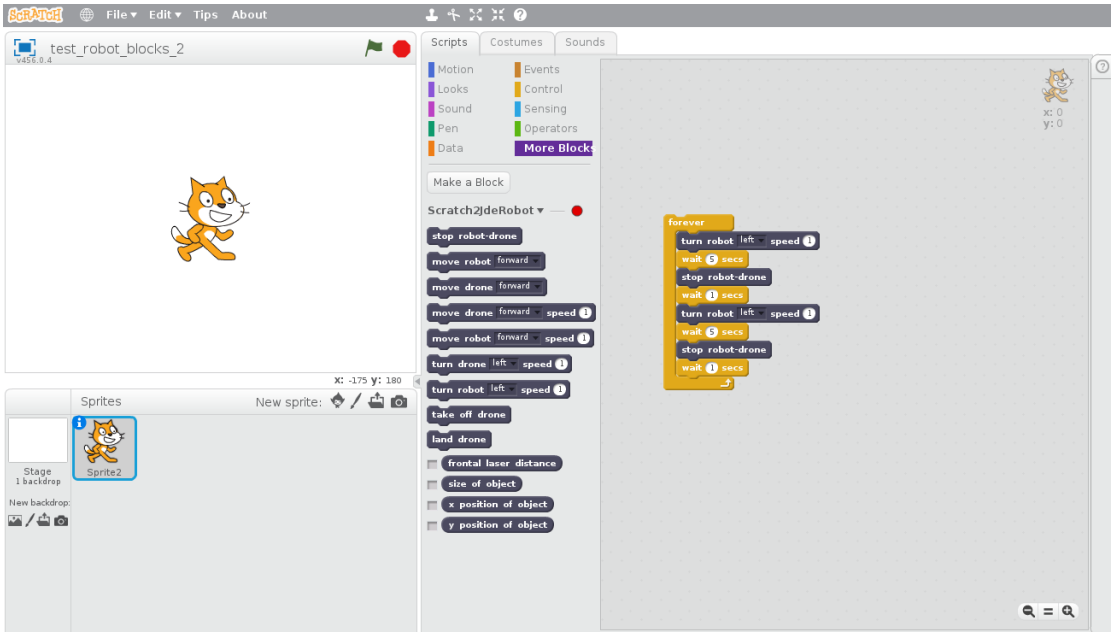

Figura 3.1: Espacio de trabajo de Scratch 2.0

Scratch sigue en constante evolución, existiendo la versión 3.0[2] en fase *Beta*. Esta versión está basada en la tecnología Blockly de Google, es capaz de integrarse en tablets, aumento de bloques funcionales, y un nuevo sistema de extensiones que permite el uso de servicios web. Aumenta su usabilidad con la agregación de una gran cantidad de videotutoriales y guías de iniciación en lenguajes de programación visual. Además todos los proyectos soportados por sus

[2] `https://en.scratch-wiki.info/wiki/Scratch_3.0`

versiones anteriores no quedarán obsoletos, sino que podrán seguir funcionando en esta nueva versión, únicamente añadiendo funcionalidades.

Esta versión saldrá de forma oficial en Enero de 2019. Scratch 3.0 está escrito en HTML5, basado en web HTML, CSS y Javascript. Scratch 3.0 utilizará principalmente las bibliotecas WebGL, Web Workers y Web Audio Javascript. Javascript es un lenguaje ampliamente soportado en todos los navegadores web, y se eligió a WebGL por su velocidad y capacidades. A diferencia de Adobe Flash, usado para Scratch 2.0, Javascript funciona sin necesidad de complementos. En esta versión se sigue manteniendo su versión offline.

Es una de las herramientas fundamentales en nuestro trabajo, serán a los proyectos generados en lenguaje de programación visual, a los que realicemos la traducción a nodos ROS escritos en Python con nuestra herramienta Scratch4Robots.

3.2. Lenguaje Python

Python[3] es un lenguaje de programación administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License.

Es un lenguaje interpretado, por lo que no se necesita compilar el código fuente para poder ejecutarlo, esto ofrece ventajas como la rapidez de desarrollo e inconvenientes como una menor velocidad, en ciertos casos, cuando se ejecuta por primera vez un código, se producen unos bytecodes que se guardan en el sistema y que sirven para acelerar la compilación implícita que realiza el intérprete cada vez que se ejecuta el mismo código.

Lenguaje muy popular en los últimos años gracias a la cantidad de librerías que contiene, tipos de datos y funciones incorporadas en el propio lenguaje, que ayudan a realizar muchas tareas habituales sin necesidad de tener que programarlas desde cero, ademas cuya filosofía hace hincapié en una sintaxis que favorezca un código legible, facilitando su aprendizaje.

[3] `https://www.python.org/`

Su sintaxis muy visual, gracias a una notación indexada (con márgenes) de obligado cumplimiento. En muchos lenguajes, para separar porciones de código, se utilizan elementos como las llaves o las palabras clave begin y end. Para separar las porciones de código en Python se debe tabular hacia dentro, colocando un margen al código que iría dentro de una función o un bucle. Esto ayuda a que todos los programadores adopten unas mismas notaciones y que los programas de cualquier persona tengan un aspecto muy similar.

Se trata de un lenguaje de propósito general y multiplataforma, Aunque originalmente se desarrolló para Unix actualmente cualquier sistema es compatible con el lenguaje siempre y cuando exista un intérprete programado para él.

Pese a ser un lenguaje multiparadigma, principalmente está orientada a objetos lo que permite en muchos casos crear programas con componentes reutilizables. También permite programación imperativa y programación funcional por lo que se adapta al estilo del programador y no al revés.

En este Trabajo de Fin de Grado usamos la versión 2.7, existiendo también en su versión 3.0, y es el lenguaje en el que basamos todo el desarrollo realizado tanto para la la propia generación del Nodo ROS, que estará desarrollado en Python, como para la traducción de bloques Scratch, cuya lógica sera en su totalidad en este lenguaje.

3.3. Herramienta Scratch4Robots 1.0

Scratch4Robots 1.0[4] desarrollado por Raúl Pérula-Martínez durante el Google Summer of Code 2017, ofrece la capacidad de traducción de bloques Scratch a código Python.

La extensión de bloques que aporta a Scratch, realizan funcionalidades básicas de movimientos tanto para drones como para robots, disponiendo de un bloque por cada tipo de movimiento a realizar, movimientos en el eje X, eje Y o movimientos de rotación. Esto abarcaba la

[4] `https://github.com/TheRoboticsClub/colab-gsoc2017-RaulPerula`

gama de movimientos de cualquier robot, pero dificultaba la integración de movimientos complejos ya que únicamente admite el movimiento en uno de los ejes a la vez.

En cuanto a traducción de bloques propios de Scratch, fuera del ámbito robótico, soporta ciertos bloques de control de flujo de programación para realizar funciones básicas como son un bucle infinito, bucles for, y sentencias condicionales.

Además el código generado a partir de la herramienta estaba basada en comunicaciones ICE, para el envió y obtención de información con el robot final.

3.4. Biblioteca Kurt

Kurt[5] es una biblioteca de Python que permite la manipulación compleja de proyectos de Scratch (archivos .sb y .sb2) a través de simples comandos de Python. Incluye un compilador y decompilador, que permite que un proyecto se cargue en un conjunto de objetos Python, y un compilador que permite cargar un conjunto de scripts de imágenes o texto en proyectos.

Al ser capaz de extraer toda la información contenida en un proyecto Scratch, y debido al parecido en la sintaxis de Scratch con Python nos sirve como principal fuente de recursos a la hora, por ejemplo, de realizar una transcripción de un bloque de Scratch a una sentencia Python.

En este Trabajo de Fin de Grado nos hemos basado en una versión[6] modificada de esta herramienta, capaz de admitir bloques definidos en extensiones externas de Scratch.

Se tratará con más profundidad el funcionamiento de esta librería en siguientes capítulos.

[5] `https://en.scratch-wiki.info/wiki/Kurt`
[6] `https://pypi.org/project/jderobot-kurt/`

3.5. Plataforma JdeRobot

JdeRobot[7] es una suite de desarrollo de software de robótica, domótica y sistemas de visión computerizados cuya última versión, la 5.6, es la usada en este proyecto y permite la integración con ROS Kinetic. Proporciona un entorno distribuido donde las aplicaciones se forman mediante una colección de componentes asíncronos concurrentes, que se conectan mediante el middleware de comunicación ICE o mensajes ROS.

Se compone de interfaces, drivers, utilidades y aplicaciones para el desarrollo de cualquier proyecto de robótica. En este proyecto nos ayudamos de librerías como pueden ser *comm, Config, JdeRobotTypes* que nos ayudan a agilizar el desarrollo de nuestra herramienta, abstraernos de ciertos niveles de complejidad y hacemos uso de una cantidad de entornos simulados en Gazebo con los que probar el correcto funcionamiento.

Nos hemos apoyado en esta plataforma para el desarrollo de este trabajo, usando diversas de sus librerías y entornos simulados predefinidos.

3.5.1. Librería Comm

Librería desarrollada por JdeRobot con versiones tanto en Python como en C que nos abstrae del tipo de comunicación utilizada por nuestros componentes. Apoyándose en las librerías ya definidas en JdeRobot para el fácil uso de nodos ROS y Ice crea una capa de abstracción que permite que una aplicación sea capaz de funcionar tanto con *ROS* como con comunicación *Ice* sin necesidad de modificar el código interno de esta. De esta forma se aprovecha el trabajo anterior, se economiza tiempo, y se reduce el código redundante.

Su funcionamiento se apoya en el uso de un fichero de configuración necesario para establecer el tipo de comunicación que usaran los sensores y actuadores de nuestro robot. De este fichero obtendremos el tipo de comunicación y toda la información necesaria para poder establecerla.

[7] `https://jderobot.org`

Se trata de un paso más en la reducción de complejidad a la hora de programar algo tan complejo como un robot.

Para nosotros esta librería es el punto de partida a la hora de buscar que nuestra librería sea totalmente funcional con ROS, nos apoyamos en ella para el desarrollo de los componentes que hacen uso de comunicaciones ROS. Desacoplando y reutilizando algunas funciones de la librería referidas puramente a ROS.

3.6. ROS

ROS[8](Robot Operating System) es un entorno para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo. Gestionado por Open Robotics Foundation. Proporciona bibliotecas y herramientas para ayudar a los desarrolladores de software a crear aplicaciones de robots. Proporciona abstracción de hardware, controladores de dispositivo, bibliotecas, visualizadores, paso de mensajes, administración de paquetes y más.

Es completamente de código abierto (BSD) y gratuito para que otros lo usen, cambien y comercialicen. Su objetivo principal es permitir que los desarrolladores de software creen aplicaciones de robots más capaces de forma rápida y fácil en una plataforma común.

En este trabajo vamos a utilizar la versión de ROS desarrollada para Ubuntu 16.04, ROS Kinetic[9]. Parte fundamental de este Trabajo Fin de Grado ya que uno de sus objetivos últimos es la generación de nodos ROS a partir de un proyecto programado en el lenguaje visual Scratch, además de la paquetización de la herramienta en una aplicación ROS. A continuación vamos a describir a muy alto nivel los conceptos básicos en los que se basa su arquitectura, esto nos va a ayudar a el proceso de funcionamiento de la herramienta Scratch4Robots 2.0.

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

[9] http://wiki.ros.org/kinetic

3.6.1. Maestro ROS

El Maestro permite que todas las demás piezas de software de ROS (Nodos) encuentren y hablen entre sí. Sin el Maestro, los nodos no podrían encontrarse, intercambiar mensajes o invocar servicios. Proporciona registro y búsqueda de nombres para el resto del gráfico de computación.

3.6.2. Nodos

Algunos robots llevan varias computadoras, cada una de las cuales controla un subconjunto de los sensores o actuadores del robot. Incluso dentro de una sola computadora, a menudo es una buena idea dividir el software del robot en pequeñas partes independientes que cooperan para lograr el objetivo general. Para esto existen los nodos de ros. Los nodos son procesos que realizan cálculos. ROS está diseñado para ser modular en una escala de grano fino, un sistema de control de robot generalmente comprende muchos nodos. Por ejemplo, un nodo controla un láser, un nodo controla los motores de las ruedas, un nodo proporciona una vista gráfica del sistema y así sucesivamente.

3.6.3. Servidor de Parámetros

El servidor de parámetros permite que los datos se almacenen por clave en una ubicación central. Actualmente es parte del Máster.

3.6.4. Mensajes

Los nodos se comunican entre sí al pasar mensajes. Un mensaje es simplemente una estructura de datos, que comprende campos tipados. Los tipos primitivos estándar (entero, punto flotante, booleano, etc.) son compatibles, al igual que las matrices de tipos primitivos. Los mensajes pueden incluir estructuras y matrices anidadas arbitrariamente (al igual que las estructuras C).

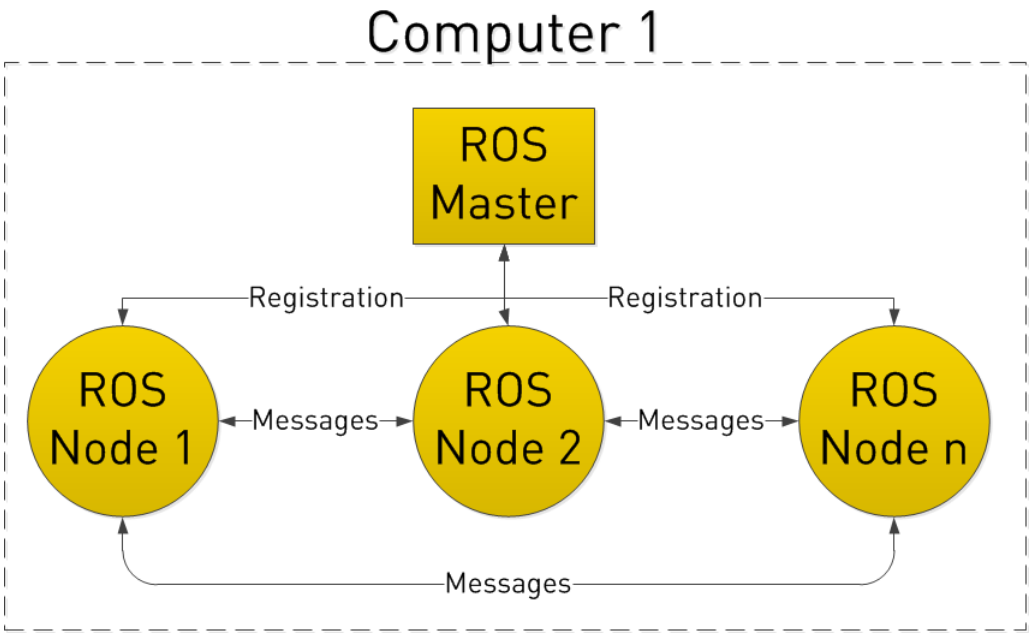

Figura 3.2: Arquitectura básica ROS maestro-nodo

3.6.5. Topics o Temas

Los mensajes se enrutan a través de un sistema de transporte con semántica de publicación suscripción. Un nodo envía un mensaje publicándolo en un tema determinado. El tema es un nombre que se usa para identificar el contenido del mensaje. Un nodo que está interesado en cierto tipo de datos se suscribirá al tema apropiado. Puede haber varios editores y suscriptores simultáneos para un único tema, y un solo nodo puede publicar y / o suscribirse a múltiples temas. En general, los editores y los suscriptores no conocen la existencia de los demás. La idea es desacoplar la producción de información de su consumo. Lógicamente, uno puede pensar en un tema como un bus de mensajes fuertemente tipado. Cada bus tiene un nombre, y cualquiera puede conectarse al bus para enviar o recibir mensajes, siempre que sean del tipo correcto.

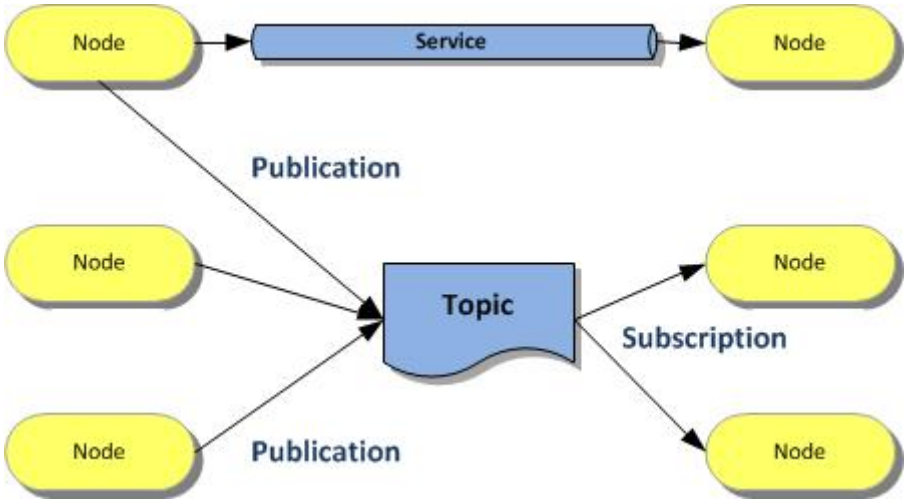

Figura 3.3: Arquitectura básica ROS publicación-suscripción

3.6.6. Servicios

El modelo de publicación / suscripción es un paradigma de comunicación muy flexible, pero su transporte unidireccional de muchos a muchos no es apropiado para las interacciones de solicitud / respuesta, que a menudo se requieren en un sistema distribuido. La solicitud / respuesta se realiza a través de servicios, que están definidos por un par de estructuras de mensaje: una para la solicitud y otra para la respuesta. Un nodo proveedor ofrece un servicio con un nombre y un cliente usa el servicio enviando el mensaje de solicitud y esperando la respuesta. Las bibliotecas cliente de ROS generalmente presentan esta interacción al programador como si fuera una llamada de procedimiento remoto.

3.7. Simulador Gazebo

Gazebo[10] es un simulador 3D para robots, es un proyecto de Open Source Robotics Fundation distribuido bajo la licencia Apache 2.0, con un motor de físicas y cinemáticas muy potente, es la principal herramienta en la que se apoya el desarrollador para verificar de forma segura que su implementación cumple con el objetivo determinado. Este tipo de simuladores han conseguido una fuerte evolución del mundo de la robotica ya que nos permite desarrollar componentes

[10] `http://gazebosim.org/`

para dispositivos y robots complejos de forma barata y rápida. Se trata de una herramienta altamente personalizable y moldeable que admite plugins que permiten una gestión más fina de los recursos de cara a conseguir una simulación más precisa o simplemente perimitir al desarrollador trabajar con mayor comodidad sobre el simulador. Al ser un proyecto open source existe una comunidad muy activa detrás de esta herramienta compartiendo conocimientos y publicando nuevas funcionalidades de esta. Cabe destacar que Gazebo se compone principalmente de un cliente y un servidor. El servidor es el encargado de realizar los cálculos y la generación de los datos de los sensores,además de poder ser usado sin necesidad de una interfaz gráfica. El cliente proporciona una interfaz gráfica basada en QT que incluye la visualización de la simulación y una serie de controles de multitud de propiedades. Esta configuración permite lanzar múltiples clientes sobre un servidor, consiguiendo múltiples interfaces de la misma simulación. Destacar que en 2013, este simulador se utilizó en la Virtual Robotics Challenge, un componente del DARPA Robotics Challenge. En este proyecto Gazebo se emplea para probar la solución desarrollada en un entorno controlado para después, pasar a un entorno real.

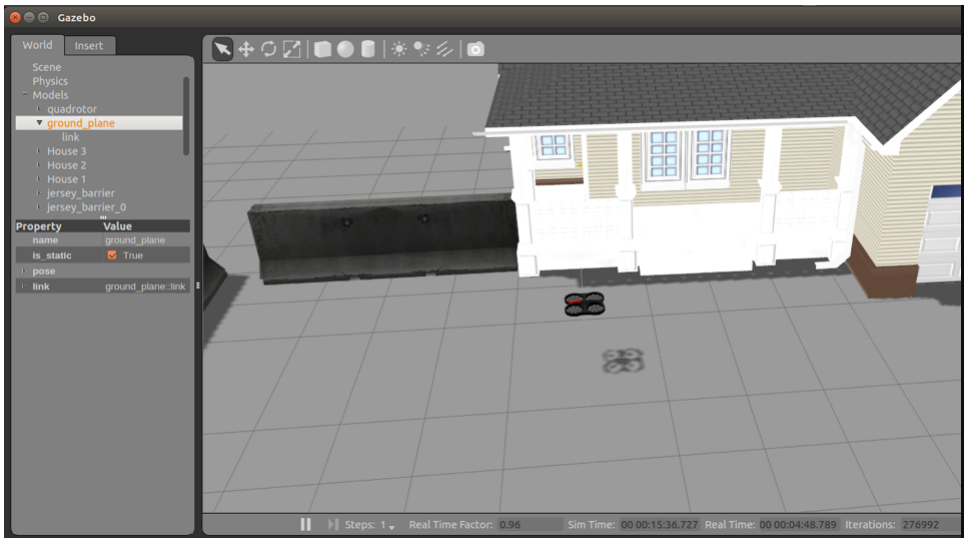

Figura 3.4: Entorno de simulación Gazebo

En este proyecto se hace uso de la versión 7 de Gazebo, siendo el entorno simulado elegido para la validación de todos los desarrollos sobre robots simulados.

Capítulo 4

Scratch4Robots 2.0

Una vez presentado el contexto, los objetivos, así como las herramientas empleadas, en este capítulo se detalla la mejora desarrollada sobre la herramienta Scratch4Robots. Primero presentamos el diseño global y después analizaremos en detalle que aportaciones se han realizado y como se han llevado a cabo. Este apartado nos ayudará a entender en profundidad y el alcance de las mejoras realizadas sobre Scratch4Robots 1.0.

4.1. Diseño

Como diseño global de la herramienta, y detallando poco el funcionamiento su funcionamiento interno, podemos esquematizar Scratch4Robots 2.0 de la siguiente forma:

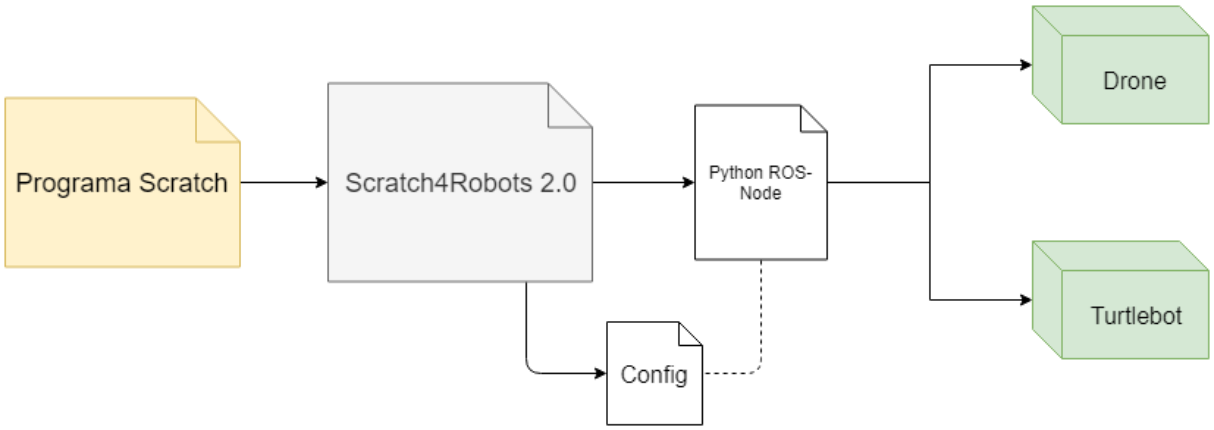

Figura 4.1: Esquema de Scratach4Robots 2.0

Esta es la poca complejidad con la que el usuario final tiene que lidiar, apreciándose su sencillez en el diseño y entendimiento a simple vista.

El diseño básico consta de un programa, codificado mediante el lenguaje visual Scratch 2.0, que nuestra herramienta, Scratch4Robots 2.0, procesará y traducirá. Como resultado de este proceso obtenemos un nodo ROS, programado en Python, y un fichero de configuración, necesario para su ejecución. Este nodo ya estará preparado para ser ejecutado sobre drones y robots Turtlebots.

Después de ver el diseño superficial, vamos a detallar como funciona de forma interna Scratch4Robots 2.0. En el siguiente esquema podemos apreciar los componentes internos en los que se basa la herramienta.

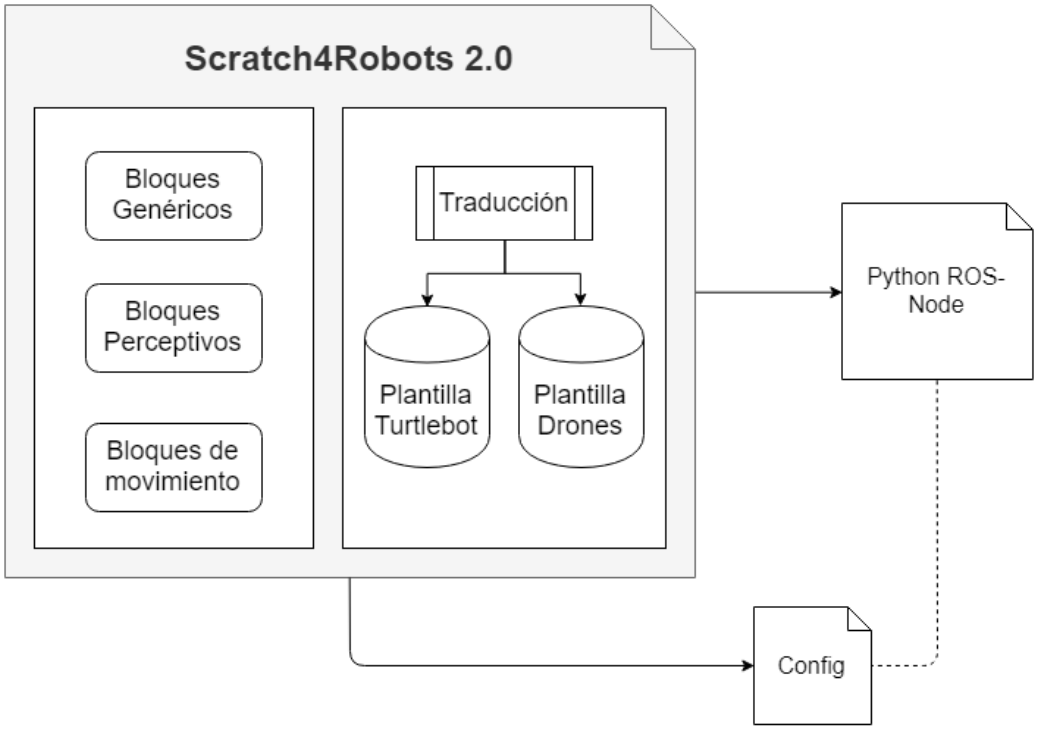

Figura 4.2: Esquema de Scratch4Robots 2.0 en detalle

En un primer apartado podemos ver los bloques visuales que soporta nuestra herramienta y de los que contiene la lógica suficiente para poder hacer uso de ellos. Estos bloques están divididos en tres grandes grupos, bloques genéricos, bloques perceptivos y bloques de movimiento.

Por otro lado vemos el módulo encargado de la traducción y generación del nodo ROS final. La traducción obtenida a partir del programa Scratch, será encapsulada en una plantilla que contiene toda la lógica necesaria para establecer las comunicaciones ROS con el drone o Turtlebot.

4.1.1. Plantillas

En el esquema en detalle definido anteriormente se introduce un elemento nuevo, de vital importancia en la ejecución de la herramienta y del que hasta ahora hemos hablado poco. Este elemento es la plantilla en la que se encapsula el código traducido del programa Scratch.

Esta plantilla tiene una función fundamental, que es la creación del nodo ejecutable final.

Esto conlleva, tanto la suscripción automática a los tópicos ROS necesarios, como el envío, procesado y lectura de mensajes ROS.

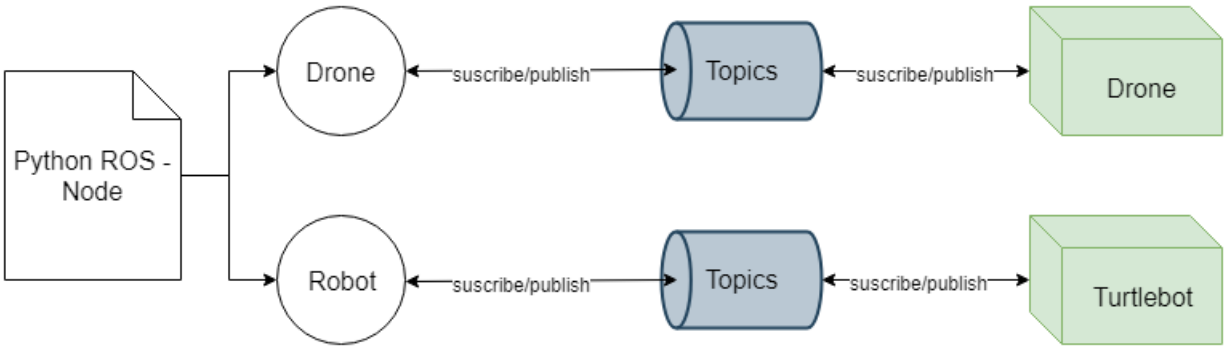

Figura 4.3: Esquema de comunicaciones ROS a través del nodo generado

Apoyándonos en la figura 4.3, vamos a explicar la lógica implementada tras esta plantilla.

Como hemos planteado anteriormente, la traducción final se encapsula en una plantilla común a drones y Turtlebots, que no es ni más ni menos que el Nodo ROS que vamos a ejecutar. De cara a simplificar el código de salida generado por nuestra herramienta, hemos agrupado la lógica referida al procesado de comunicaciones ROS en dos librerías, una para Turtlebots y otra drones. De ahí a que en la figura 4.3 se hayan dividido estas dos partes. Entrando en detalle mostramos el código simplificado del nodo final generado.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-n
import ...
# Establece la subscripcion a topics ROS para drones
from drone import Drone
# Establece la subscripcion a topics ROS para robots
from robot import Robot
# Ejecucion del nodo
def execute(robot):
    try:
    # Desde codigo, insertamos la traduccion obtenida aqui
    // Scratch Code
```

```
    except KeyboardInterrupt:
        raise
if __name__ == '__main__':
    # carga de configuraciones, topics ros a los que nos suscribimos, a traves de fichero yml
        cfg = config.load(open_path + filename)
        yml_file = yaml.load(stream)
    for section in yml_file:
    # En el yml de configuracion se define si se trata de robot o drone.
        if section == 'drone':
        # Si es drone se inicia la clase que hace la subscripcion a todos los topics ROS
            robot = Drone(cfg)
        elif section == 'robot':
        # De forma analoga si se trata de un robot sobre ruedas
            robot = Robot(cfg)
    execute(robot)
```

Entrando en más detalle de estas librerías, vamos a ver como realiza la suscripción y publicación en los tópicos desde una de ellas, con fragmentos de códigos obtenidos de la aplicación.

```
import config
import rospy

class Robot():
    def __init__(self, cfg):
        # inicia nodo ROS Robot
        self.__node = rospy.init_node("robot", anonymous=True)
        # hacemos la subscripcion a los topics ROS
        topic = cfg.getProperty("robot.Pose3D.Topic")
        self.__pose3d_client = ListenerPose3d(topic)
        topic = cfg.getProperty("robot.Motors.Topic")
        self.__motors_client =  PublisherMotors(topic)

        def getPose3d():
                return self.__pose3d_client.getPose3d()
```

Describiendo el fragmento de código anterior y como se aprecia en el esquema de más arriba, nuestro nodo ROS en Python, a través del objeto Python *Robot*, que al iniciarse por primera vez hace la carga de propiedades del fichero de configuración que se le pasa como argumento. Estas propiedades como hemos comentado anteriormente serán los tópicos a los que se conectará nuestro nodo. Se realizará la suscripción o publicación a cada tópico de manera individual. En este caso y observando el código, usamos el objeto *ListenerPose3d*, para conectarnos al tó-

pico de el que obtendremos la información publicada por el sensor de odometría del robot, del mismo modo se hace con *PublisherMotors* para el uso de los motores del robot.

En resumidas cuentas, al iniciarse el objeto Python, *Robot* o *Drone*, según se esté usando un tipo u otro, éste realiza la suscripción a todos los tópicos a través de la creación de un *cliente* por cada una de estas suscripciones. Estos *clientes* ya contienen métodos desde los que obtener mediante comunicaciones ROS, la información final tal y como la esperamos.

En el siguiente extracto de código se aprecia simplificado el detalle de una de los clientes que realizan la suscripción, y de los obtenemos información del robo, a través de mensajes ROS, publicados en un tópico.

```
# modulo python oficial ros
import rospy
# mensajes de comunicacion ROS para la lectura de odometria
from nav_msgs.msg import Odometry
class ListenerPose3d:
    def __init__(self, topic):
        # Objeto Pose3d() contiene los datos devueltos por el sensor
        self.data = Pose3d()
        # Al suscribirnos como argumento se introduce el topico,
        # el tipo de mensaje ROS que se lee del topico
# y una funcion de callback en la que recogemos las lecturas del topico
        self.sub = rospy.Subscriber(self.topic, Odometry, self.__callback)

    def __callback (self, odom):
        pose = odometry2Pose3D(odom)
        self.lock.acquire()
        self.data = pose
        self.lock.release()

    # metodo a traves del cual obtendremos la pose 3d
    def getPose3d(self):
        self.lock.acquire()
        pose = self.data
        self.lock.release()
        return pose
```

Este mismo procedimiento hay que seguirlo para cada uno de los posibles tópicos ROS que necesite nuestra aplicación, por ejemplo, uno para dar velocidad a los distintos motores, otro

para el sensor láser, cámaras incrustadas en el robot etcétera.

Con todo esto se puede apreciar la complejidad de la que evadimos al usuario, siendo para él, un simple ejecutable o nodo, completamente operativo, generado a través de nuestra aplicación. Una vez lanzado el nodo habremos sido capaces de aplicar una lógica programada en Scratch sobre un robot en un entorno simulado.

4.2. Traducción de bloques

Una vez entendido el funcionamiento de la plantilla encargada de toda la lógica de funcionamiento sobre comunicaciones ROS, y volviendo a la figura 4.2, vamos a especificar como se lleva a cabo la traducción del programa Scratch y las mejoras que hemos llevado a cabo en este proceso.

Esta labor de traducción la realizamos a través del uso de la bibliotea Kurt, vamos a estudiar más a fondo como trabaja la esta biblioteca y cómo nos ha ayudado en la traducción de bloques.

Kurt es una biblioteca de Python que permite la manipulación compleja de proyectos Scratch (archivos .sb o .sb2) a través de simples comandos de Python. Incluye un decompilador, que permite que un proyecto se cargue en un conjunto de objetos de Python, y un compilador que permite el empaquetamiento de un conjunto de *scripts* de imágenes / texto en proyectos Scratch.

En este proyecto se ha utilizado una versión[1] mejorada de esta biblioteca, con ciertas adaptaciones para el tratamiento de extensiones externas como la nuestra. Kurt define los bloques usados por Scratch como objetos JSON, en esta nueva versión se agrega un fichero con la configuración de nuestros bloques como objetos JSON. De esta forma se agrega la capacidad de obtener la información de los bloques de nuestra extensión.

La clase principal de kurt almacena el contenido de un archivo de proyecto Scratch. Los contenidos incluyen variables globales y listas, el escenario y los *sprites*, cada uno con sus pro-

[1] `https://pypi.org/project/jderobot-kurt/`

pios *scripts*, sonidos, variables y listas.

Una vez obtenido el contenido de un proyecto Scratch en un objeto Python, con el siguiente fragmento de código somos capaces de obtener el *strings* representativo de cada bloque.

```
# load the scratch project
p = kurt.Project.load(open_path + sys.argv[1])

# show the blocks included
for scriptable in p.sprites + [p.stage]:
        for script in scriptable.scripts:
                # exclude definition scripts
                if "define" not in script.blocks[0].stringify():
                        s = script
print("Stringify:")
sentences = []
for b in s.blocks:
        print(b.stringify())
```

Una vez tenemos la lista de los *strings* equivalentes a los bloques del proyecto Scratch, hay que realizar una tarea de transformación de estas a código Python.

En este trabajo se modifica de forma notable la forma en la que se realizaba de forma inicial esta traducción, agregando una gran cantidad de bloques soportados y añadiendo un grado de recursividad que permite la traducción de bloques anidados. El soporte de bloques anidados nos permite realizar acciones de mayor complejidad dentro del entorno Scratch, como pueden ser bucles en los que se definen flujos condicionales complejos.

En la traducción nos apoyamos de unos diccionarios Python en los que mapeamos el string equivalente al bloque Scratch con el equivalente Python que queremos sustituir, esto nos lleva al último punto de nuestro desarrollo. Para ciertos bloques de Scratch existe una traducción directa a un comando Python como ya hemos visto antes. Mientras que para los bloques robóticos definidos por nuestra extensión debemos de crear un método específico que contenga toda la lógica a implementar como hemos visto anteriormente en la descripción de nuestros bloques.

4.3. Desarrollo de bloques

Una de las aportaciones de este trabajo a la herramienta ha sido la refactorización de bloques ya existentes y la agregación de nuevos bloques funcionales. En la programación visual entendemos por bloque a cada componente visual que contiene una lógica específica, con la agrupación de bloques podemos conseguir un flujo de programación complejo. En primer lugar vamos a describir como hemos agregado estos nuevos bloques a Scratch, siguiendo con su definición e implementación.

4.3.1. Extensión de Scratch

Para la agregación de nuevos bloques, Scratch facilita el uso de extensiones externas, estas extensiones definen mediante el uso de ficheros .s2e compuesto por una serie de objetos JSON. Este tipo de ficheros se creó para la comunicación mediante HTTP de bloques con aplicaciones auxiliares, por ejemplo algún tipo de hardware. Nosotros no vamos a utilizar esta funcionalidad, únicamente nos ayudamos de este documento .s2e para definir nuestra extensión y pueda ser usada desde el IDE offline de scratch.

En este documento se define un objeto JSON, el cual será la definición de la extensión, en el que se incluye el nombre de la extensión, un puerto usado para la comunicación de componentes externos en caso de ser necesario y una lista de bloques de Scratch.

A continuación se define un ejemplo de una extensión para Scratch.

```
{
  "extensionName": "Extension Example",
  "extensionPort": 12345,
  "blockSpecs": [
    [" ", "beep", "playBeep"],
        [" ", "set beep volume to %n", "setVolume", 5],
        ["r", "beep volume", "volume"],
  ]
}
```

El campo “blockSpecs” describe los bloques de extensión que aparecerán en el apartado “Más bloques” en la aplicación de Scratch. En en este caso, hay tres bloques:

- Un bloque de comandos que reproduce un pitido.
- Un bloque de comando que establece el volumen del pitido.
- Un bloque que devuelve un valor, que informa del volumen de un pitido.

Cada bloque se describe mediante una matriz con los siguientes campos:

- **Tipo de bloque**:
  - ' ' - bloque de comandos
  - 'w' - bloque de comandos que esperan
  - 'r' - bloque que retorna un valor
  - 'b' - bloque que retorna un booleano
- **Formato de bloque**:

  El formato de bloque es una cadena que describe las etiquetas y ranuras de parámetros que aparecen en el bloque. Las ranuras de parámetros están indicadas por una palabra que comienza con ' %' y puede ser una de:
  - %n - parámetro de número
  - %s - parámetro de cadena
  - %b - parámetro booleano
- **Operación o nombre de variable remota**:

  El campo de operación en una especificación de bloque se usa de dos maneras. Para bloques de comandos, se envía a la aplicación auxiliar, junto con cualquier valor de parámetro, para invocar una operación. O para retornar bloques, es el nombre de una variable de sensor. Los valores de la variable del sensor se guardan en un diccionario. La ejecución de un bloque simplemente devuelve el valor reportado más recientemente para esa variable de sensor.
- **Parámetros predeterminados**:

  Se pueden añadir cero o más valores de parámetros predeterminados

- **Menús desplegables**:

  Los bloques que definimos pueden hacer uso de parámetros de menú, los cuales definiremos de dos formas:

  - %m.menuName - parámetro de menú (no editable), proporciona un sencillo espacio para los parámetros del menú desplegable.
  - %d.menuName - parámetro de número editable con menú, proporciona una ranura de parámetro numérico con un menú auxiliar.

Con todo esto podemos entender la definición de alguno de nuestros bloques como podemos en el siguiente extracto de código, en el que se ve la definición de algunos de nuestros bloques.

```
{
  "extensionName": "Scratch4Robots",
  "extensionPort": 12345,
  "blockSpecs": [
    ["", "stop robot-drone", "stop"],
    ["", "move robot %m.robotDirections speed %n", "robot/move/speed", "forward", 1],
    ["r", "color detection %m.color", "camera/all","red"],
    ["r", "frontal laser distance", "laser/frontal"],
  ],
  "menus": {
    "robotDirections": ["forward", "back"],
    "color": ["red", "blue"]
  }
}
```

4.3.2. Bloques genéricos

Antes de comenzar con los bloques propios a nuestra extensión vamos a necesitar una serie de bloques genéricos que nos ayuden a realizar lógica básica de programación como pueden ser operadores matemáticos, operadores lógicos y sentencias básicas de programación. Scratch ya nos ofrece estos bloques por definición, por lo que no hará falta agregarlos a nuestra extensión específica, solamente nos encargaremos de su posterior traducción a código Python.

A continuación mostramos todos los bloques genéricos de los que podemos hacer uso desde la herramienta Scratch4Robots, ya que hemos implementado su traducción a Python.

- **Bloques de operadores matemáticos**:

Bloques fundamentales de cara a realizar una lógica posterior con los datos obtenidos desde los sensores de nuestros robots, estos bloques son una de las mejoras ofrecidas por este trabajo.

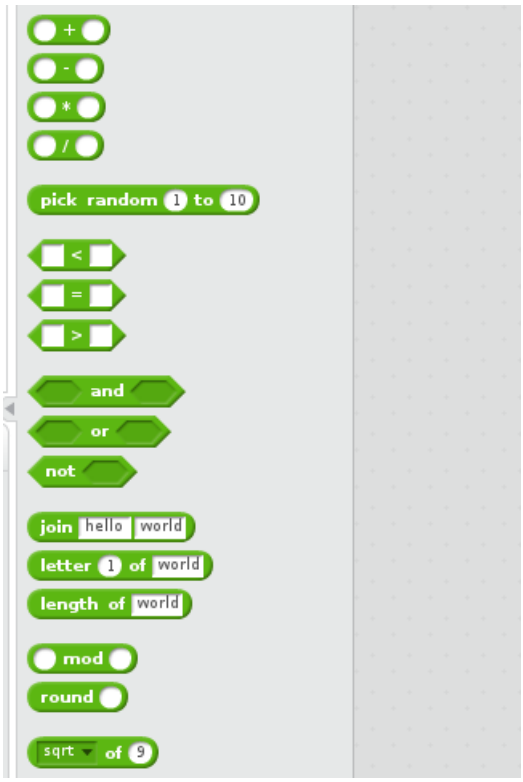

Figura 4.4: Bloques matemáticos y lógicos

- **sqrt of** (): Realiza la operación raíz cuadrada de un número dado
- **sin of** (): Realiza la operación seno de un número dado
- **cos of** (): Realiza la operación coseno de un número dado
- **tan of** (): Realiza la operación tangente de un número dado
- **asin of** (): Realiza la operación arcoseno de un número dado
- **acos of** (): Realiza la operación arcocoseno de un número dado
- **atan of**(): Realiza la operación arcotangente de un número dado
- **log of** (): Realiza la operación logaritmo de un número dado
- **ln of** (): Realiza la operación logaritmo neperiano de un número dado
- **abs of** (): Devuelve el valor absoluto de un número

- **mod of** (): Devuelve el módulo de un número dado

- **Bloques de operadores lógicos**: Permiten construir expresiones lógicas, se obtiene como resultado booleanos.

  - **And**: Operador de conjunción.
  - **Or**: Operador de disyunción.
  - **NOT**: Operador de negación.
  - **Mayor que y Menor que**: Operadores de comparación numérica.

- **Bloques de control**:

  Estructuras de control básicas de programación.

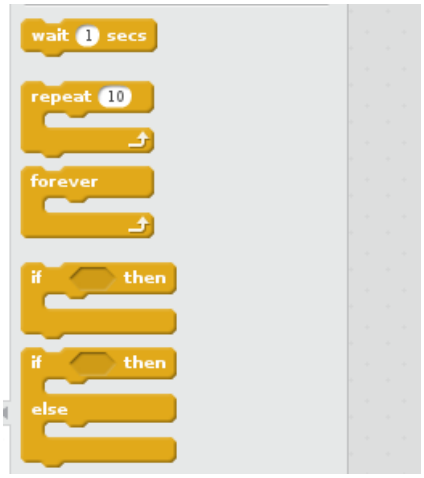

Figura 4.5: Bloques de control

- **Wait () secs**: Pausa la ejecución el tiempo especificado, equivalente a la sentencia *time.sleep()* de Python.
- **Forever**: Bucle infinito, equivalente a *while(True)* en lenguaje Python.
- **If () then**: Comprueba la condición para que si la condición es verdadera, los bloques dentro de ella se ejecuten.
- **If () Then, Else**: Comprueba la condición para que si la condición es verdadera, los bloques dentro de la primera condición se activen y si la condición es falsa, los bloques dentro de la segunda condición se activarán.

- **Repeat** (): Un ciclo que repite la cantidad de veces especificada, sería la equivalencia al bucle *for* en Python.

- **Otros**

  - **say** (): Imprime lo que le añadas como argumento, equivalente a *print*.
  - **Set** () **to** (): Utilizado para dar valor a una variable en concreto.

- **Bloques de listas**:

  Esta serie de bloques desarrollados en esta versión han sido de gran ayuda a la hora de la refactorización y creación de otros bloques de mayor complejidad.

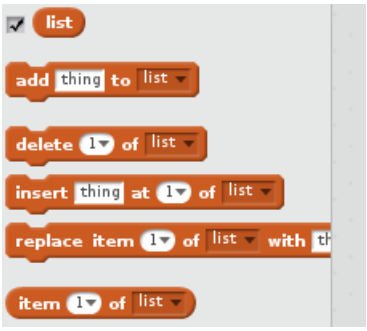

Figura 4.6: Bloques de listas

  - **Insert** () **at** () **of** (): Inserta elemento en la posición seleccionada de la lista indicada.
  - **Item** () **of** (): Devuelve el elemento almacenado en la posición indicada de la lista.
  - **Add** () **to** (): Inserta en la lista un elemento.
  - **Delete** () **of** (): Elimina el elemento en una posición determinada de la lista.

4.3.3. Bloques para drones

Estos bloques ya pertenecen a la extensión creada para la herramienta. En este trabajo se ha realizado una refactorización completa de todos los bloques referentes a drones y la agregación de nuevos bloques.

Cabe destacar de estos bloques que una vez traducidos al nodo final, que se consigue implementar el funcionamiento de drones bajo comunicación ROS, ésto es algo elaborado completa-

mente en este trabajo.

Para ello nos apoyamos en MAVROS[2], puente oficial entre nodos ROS y Mavlink[3], el protocolo de comunicación estándar para drones.

- **Bloques perceptivos**: Nos permiten obtener información de sensores incorporados en el drone.

  - **Get pose3D**: Obtiene el valor de la posición 3D del robot.

Figura 4.7: bloque que retorna la posición del robot

Simplificando la totalidad del código, la función Python en la que quedaría traducida sería *getPose3d*, como se muestra:

```
1      def getPose3d(self):
2          # client_pose3d se trata del cliente que hace la comunicacion ROS
3          return client_pose3d.getPose3d
```

  - **Color detection**: Nuevo componente añadido en esta versión, que haciendo uso de una cámara en el robot, detecta objetos de un determinado color, introducido como argumento, devolviendo una lista que contendrá las posiciones en los ejes x e y, además del tamaño del objeto, en la imagen capturada por la cámara.

Figura 4.8: bloque que detecta objetos de un determinado color

[2] http://wiki.ros.org/mavros
[3] https://github.com/mavlink

Usando la biblioteca OpenCV de Python y con técnicas de análisis computacional de imágenes, de una simple imagen somos capaces de obtener su posición en la imagen y su tamaño:

```
    def detect_object(self, color):
        # define the lower and upper boundaries of the basic colors
        color_range = __get_color_range(color)
        # get image type from camera ROS client
        image = self.__camera_client.getImage()
        # apply color filters to the image
        filtered_image = cv2.inRange(image.data, color_range[0], color_range[1])
        rgb = cv2.cvtColor(image.data, cv2.COLOR_BGR2RGB)
        # Apply threshold to the masked image
        ret,thresh = cv2.threshold(filtered_image,127,255,0)
        im,contours,hierarchy = cv2.findContours(thresh,cv2.RETR_TREE,cv2.
    CHAIN_APPROX_SIMPLE)
        # Find the index of the largest contour
        for c in contours:
            if c.any != 0:
                areas = [cv2.contourArea(c) for c in contours]
                max_index = np.argmax(areas)
                cnt=contours[max_index]
                if max(areas) > 0.0:
                    x,y,w,h = cv2.boundingRect(cnt)
                    x_position = (w/2)+x
                    y_position = (h/2)+y
                    size = w*h
        return size, x_position, y_position
```

- **Bloques de movimiento**

  - **Stop robot-drone**: Pone a su valor inicial todas las velocidades del robot.

Figura 4.9: bloque stop-robot

Con el código que se ejecutará tras su traducción:

```
    def stop_robot(self):
        self.client__vel.sendVelocities(0,0,0,0,0,0)
        time.sleep(1)
```

- **Drone take off**: Hace que el drone despegue.

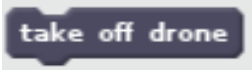

Figura 4.10: bloque que realiza el despegue del drone

Mediante el armado del drone y el envío de una velocidad ascendente conseguimos su despegue:

```
    def takeoff(self):
        # arming realiza el encendido del robot
        self.client_extra.arming()
        self.client_vel.sendVelocities(0,0,2,0,0,0)
        time.sleep(1)
        self.client_vel.sendVelocities(0,0,0,0,0,0)
```

- **Drone land**: Realiza un aterrizaje controlado del drone.

Figura 4.11: bloque que realiza el aterrizaje del drone

Para realizar el aterrizaje del drone a través de ROS, tiene su propio tópico al cual se le envía una llamada con argumentos cero, como se puede observar:

```
    def land(self):
        self.lock.acquire()
```

```
3           self.land_client.call(0,0,0,0,0)
4           self.lock.release()
```

- **Move drone**: Bloque que tras la refactorización admite una lista con las velocidades del drone en los diferentes ejes (velocidad en el eje x, velocidad en el eje z, velocidad yaw), pare realizar giros se debe combinar la velocidad en el eje x con la velocidad de rotación en yaw.

Figura 4.12: bloque que realiza el despegue del drone

Lo que se traduce a python con la siguiente función:

```
1  def move_vector(self, velocities):
2      # cliente que realiza la suscripcion al topico que recibe la pose
3      pose = self.client__pose3d.getPose3d()
4      yaw = pose.yaw
5      vx = velocities[0]
6      vz = velocities[1]
7      az = velocities[2]
8      vxt = vx*math.cos(yaw)
9      # cliente que realiza la suscripcion al topico que publica velocidades
10     self.client__vel.sendVelocities(vxt,0,vz,0,0,az)
```

4.3.4. Bloques para Turtlebots

- **Bloques perceptivos**

  - **Pose3D** :Obtiene el valor de la posición 3D del robot.

Figura 4.13: bloque pose del robot

Se traduce de forma equivalente a su homólogo para drones, la única diferencia es el tópico desde el que recibe la información.

- **Color detection**: Bloque homólogo al del drone, dado un color nos devuelve su posición en la imagen, y su tamaño.

Figura 4.14: bloque que detecta colores en una imagen

Se trata del mismo bloque perceptivo ya definido para drones.

- **Frontal distance**: Obtiene la medida promedio de los datos del láser frontal.

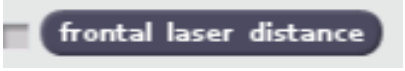

Figura 4.15: bloque que devuelve datos de un laser

Se traduce al siguiente método Python:

```
def get_laser_distance(self):
        # get laser values from ROS client
        laser = self.__laser_client.getLaserData()
        l = [x for x in laser.values if str(x) != 'nan' and x < 10]
        try:
            avg = sum(l) / len(l)
        except ZeroDivisionError:
            avg = 0
        return avg
```

- **Bloques de movimiento**

- **robot move** (): Admite como parámetro una lista con las velocidades en el eje x e y.

Figura 4.16: bloque de movimiento para robots

Con su traducción:

```
    def move_vector(self, velocities):
        vx = float(velocities[0])
        vz = float(velocities[1])
        print "velocities:",vx,vz
        self.__reset()
        self.__vel.vx = vx
        self.__vel.vz = vz
        if vz>0:
            self.turn("left",vz)
        if vz<0:
            self.turn("right",vz)
        self.__publish(self.__vel)
```

- **robot turn** (): Permite la rotación sobre el propio eje del robot.

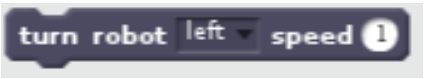

Figura 4.17: bloque que realiza el giro del robot

Con su traducción Python:

```
    def turn(self, direction, vel):
        self.__vel.az = vel
        # set different direction
        if direction == "right":
            self.__vel.az = -self.__vel.az
        # publish movement
        self.__publish(self.__vel)
```

Con esto se definen los bloques pertenecientes a nuestra extensión, de la forma en que Scratch los muestra y la traducción de cada uno de ellos llevada, de una forma simplificada.

Se aprecia que se trata de robots sofisticados, con cierta complejidad. La cual hacemos transparente para el usuario, simplificando toda la lógica que existe tras cada robot, en simples e intuitivos bloques gráficos.

En este trabajo hemos refactorizado la totalidad de la lógica tras los bloques robóticos, dándole una mayor coherencia a todos ellos. Además de crear lógica completamente nueva para ciertos bloques agregados en esta versión. Cabe destacar el bloque referente a la detección de objetos de un cierto color.

Capítulo 5

Integración

Una vez explicado los nuevos desarrollos efectuados para esta versión de la herramienta, vamos a centrarnos en uno de los puntos claves de este trabajo, la paquetización de la herramienta, llegando a conseguir una fácil instalación e integración de esta. Esto lo conseguimos con un análisis de nuestras dependencias, desacoplando todas estas dependencias lo más que podamos para darle una mayor flexibilidad, y una vez desacopladas permitir la instalación de la herramienta, junto con sus dependencias de una forma sencilla, rápida y homogénea.

5.1. Análisis de dependencias

Partimos de una versión fuertemente acoplada con JdeRobot, siendo necesaria la instalación de toda la suit para el correcto funcionamiento de la herramienta. Esto añade una complejidad de instalación innecesaria a nuestra aplicación, cosa que no queremos ya que buscamos un uso educativo de ella, por lo que buscamos la mayor facilidad de instalación posible.

Como principal dependencia nos encontramos la biblioteca *comm* de JdeRobot, biblioteca que como ya hemos explicado antes, abstrae al programador del tipo de comunicación que se quiera establecer con el robot final, pudiendo con un mismo punto de entrada, entablar una comunicación vía ROS o vía ICE. En este trabajo nos hemos centrado en la comunicación mediante ROS, por lo que se ha realizado una tarea de desacople de las funcionalidades ROS de la biblioteca *comm*. El resultado de esta tarea de desacople es la creación de una nueva biblioteca, basada puramente en comunicación ROS.

Esta biblioteca llamada JDRros será la encargada de crear y gestionar las comunicaciones entre nodos ROS de nuestras aplicaciones. Desde la publicación y suscripción de nodos, hasta el envío de mensajes a través de ellos.

A su vez, la propia biblioteca *comm* tenía dependencias con otras librería del entorno JdeRobot, en algunos casos hemos optado por refactorizar las clases puramente ROS, ya desacopladas de *comm*, para que no hagan uso de estas dependencias. Mientras que en algunos casos, viendo la utilidad de estas librerías, se han optado por incorporar a nuestro proyecto. Un ejemplo de esta reutilización es la biblioteca *Config* de JdeRobot.

Para comprender la importancia de esta dependencia vamos a explicar brevemente su funcionalidad. Esta biblioteca nos ayuda a la carga de propiedades mediante un fichero .yml, que mediante una simple API, busca y retorna cualquier propiedad que se haya definido en este fichero. En nuestro caso es extremadamente útil, ya que podemos adaptar nuestro nodo ROS a cualquier robot y situación. Como hemos explicado anteriormente, ROS establece comunicación a través de nodos mediante la publicación de Topics, a los que el resto de nodos podrán subscribirse. En este fichero de comunicación establecemos los Topics de los que se va a nutrir nuestro nodo, mediante la librería *Config* los cargamos y con las utilidades de la biblioteca ya mencionada *JDRros*, nos subscribimos o publicamos en esos Topics.

Éstas serían las dependencias con respecto a JdeRobot, pero nuestra herramienta contiene dependencias externas.

La más importante es con la ya conocida biblioteca Kurt, parte esencial en nuestro trabajo de traducción, y por lo tanto en el funcionamiento de la herramienta.

5.2. Creación y publicación de paquetes pip

Una vez hecho el análisis y desacoplo en mayor medida de nuestro código con estas dependencias, buscamos la forma de facilitar la instalación de nuestra herramienta.

Con todo esto hemos dejado el camino preparado para conseguir que nuestra herramienta no sea un programa monolítico, en el que se encuentran incrustadas todas sus librerías y funcionalidades, teniendo que actualizar toda herramienta cada vez que ésta sufra un mínimo cambio en cualquiera de sus dependencias.

Para dar el paso de una aplicación monolítica, a una distribuida, lo conseguimos mediante la paquetización de estas dependencias en paquetes Pip[1] independientes.

Pip es un sistema de administración de paquetes utilizado para instalar y administrar paquetes de software escritos en Python. Todos los paquetes pip pueden encontrarse en el Python Package Index (PyPi)[2], siendo accesibles para todo el mundo en cualquier momento. Es fácil ver el potencial de este sistema, subiendo nuestras dependencias a PyPi somos capaces de instalar todas ellas de una sola atacada.

Para crear estos paquetes pip se genera en un directorio una estructura de ficheros específica, entre las que se encontrarán por su puesto nuestro módulo y un archivo llamado *setup.py*. Aquí se configura todo nuestro paquete, desde su nombre, versión, dirección del código fuente o incluso dependencias que necesite y se instalarán a su vez de forma automática.

El siguiente paso es generar *paquetes de distribución* para el paquete. Estos son los archivos que se cargan en el índice de paquetes (PyPi) y pueden ser instalados por pip por el usuario final.

Para este trabajo se han generado los siguientes paquetes pip para cada una de las dependencias que tenía nuestra herramienta:

- **jderobot-ros**[3]:

  Contiene todo lo referente a la publicación y suscripción de Tópicos ROS.

[1] `https://pip.pypa.io/en/stable/`
[2] `https://pypi.org/`
[3] `https://pypi.org/project/jderobot-ros/`

- **config-jderobot**[4]:

  Ayuda a la carga de propiedades desde un fichero de configuración, estas propiedades en nuestro caso serán los Topicos que necesita nuestro nodo para funcionar.

- **jderobot-jderobottypes**[5]:

  Abstracción de tipos de datos, simplificándolos para un más fácil uso de ellos.

- **jderobot-kurt**[6]:

  Contiene la versión adaptada de la biblioteca para nuestro proyecto, haciendo posible la obtención de información de los bloques definidos en la extensión Scratch que hemos creado para nuestra herramienta.

- **robot-scratch4robots**[7]:

  Modulo que contiene toda la lógica tras los bloques específicos de robots sobre ruedas que hemos creado para esta herramienta.

- **drone-scratch4robots**[8]:

  Modulo que contiene toda la lógica tras los bloques específicos de drones que hemos creado para esta herramienta.

Con esto hemos modularizado al máximo nuestra herramienta, consiguiendo poder mejorar y ampliar su funcionalidad sin necesidad de generar una nueva versión completa de nuestra herramienta, además de conseguir la instalación de todas nuestras dependencias con una facilidad asombrosa.

5.3. Creación y publicación de paquete ROS

Hasta ahora hemos externalizado nuestros módulos Python en paquetes pip, pero aún no hemos resuelto como facilitar el uso e instalación de nuestra herramienta, que como hemos expuesto, se trata de un nodo ROS. Antes de comenzar este trabajo la herramienta se basaba en un

[4] `https://pypi.org/project/config-jderobot/`
[5] `https://pypi.org/project/jderobot-jderobottypes/`
[6] `https://pypi.org/project/jderobot-kurt/`
[7] `https://pypi.org/project/robot-scratch4robots/`
[8] `https://pypi.org/project/drone-scratch4robots/`

repositorio con el código fuente, el cual había que descargar al completo y manualmente instalar todo el entorno necesario para su correcto funcionamiento. Esto llegaba a ser engorroso e incluso complicado, ya que eran necesarias una gran cantidad de herramientas complementarias.

La solución aportada en este trabajo es la creación de un paquete ROS y su posterior publicación a los repositorios públicos. Con esto, al igual que hemos hecho con los paquetes pip conseguimos que nuestra herramienta sea instalable como cualquier otro programa en un sistema operativo Ubuntu, mediante un solo comando.

Para la creación del paquete nos ayudamos de *catkin*[9]. Es el sistema de compilación oficial de ROS. Combina macros de CMake y scripts de Python para proporcionar alguna funcionalidad además del flujo de trabajo normal de CMake. El flujo de trabajo de *catkin* es muy similar al de CMake, pero agrega soporte para la infraestructura automática de 'encontrar paquetes ROS' y al mismo tiempo construir múltiples proyectos dependientes. Esto se consigue gracias al sistema de compilación personalizada que usa *catkin*.

Tras generar los ficheros de configuración necesarios de forma correcta, en los que indicamos las dependencias de la herramienta, los nodos que queremos exponer una vez esté instalado el paquete, e información referente a la distribución de ROS para la cual ha sido creado, nuestro paquete estará listo para ser generado.

En nuestro caso el nodo principal que queremos exponer, es el encargado de la traducción de Scratch a Python y genera el nodo final preparado par ser ejecutado sobre robots. Este nodo podrá ser ejecutado desde linea de comandos sin necesidad de disponer del código fuente de la aplicación, ya que estará instalado.

Para el último paso de nuestro proceso, la publicación del paquete, nos apoyamos de la herramienta *bloom*[10] proporcionada por la Open Source Robotics Foundation, que si hemos configurado correctamente nuestro paquete, automatizará la generación de los binarios que serán descargado por los usuarios finales, como se hace con cualquier otra instalación en Ubuntu.

[9] `http://wiki.ros.org/catkin/conceptual_overview`
[10] `http://wiki.ros.org/bloom`

Tras realizar una petición al equipo que gestiona ROS y cumplir con los requisitos de calidad que ellos establecen, nuestro paquete será publicado.

De esta forma hemos conseguido que Scratch4Robots[11] esté publicado de forma oficial como un paquete ROS, para su distribución Kinetic[12], en repositorios públicos, además de todas sus dependencias.

5.4. Documentación de la herramienta

Por último, pero no por ello menos importante, cabe destacar la labor de documentación que se ha llevado a cabo en este trabajo. Una de las partes más importantes para el futuro uso de esta herramienta. La simpleza de uso, ayudada de una documentación clara, concisa y con ejemplos disponibles, hacen de nuestra aplicación idónea para aquellas personas con menos conocimientos técnicos.

En el repositorio oficial de Scratch4Robots[13] se indica paso a paso como realizar instalación y un primer ejemplo de uso, apoyado todo vídeos explicativos.

Además de los pasos para su instalación, se suministra un ejemplo para cada tipo de robot, robots con ruedas, y drones. Estos ejemplos contienen todo lo necesario para su ejecución en minutos, desde los proyectos Scratch ya generados, los ficheros de configuración necesarios para su ejecución, así como los comandos necesarios para la ejecución de los entornos simulados y una descripción detallada de su uso.

En la página oficial de JdeRobot, Scratch4Robots[14] se documenta en más detalle, en la que además de la guía de instalación y uso de la herramienta, se muestran todos los bloques disponibles, su forma de uso e incluso como puedes desarrollar tus propios bloques.

[11] `http://wiki.ros.org/scratch4robots`
[12] `http://wiki.ros.org/kinetic`
[13] `https://github.com/JdeRobot/Scratch4Robots`
[14] `http://jderobot.org/Scratch4Robots`

Con esto finalizamos la fase de integración de la herramienta, consiguiendo un entorno modular, de fácil instalación, y con guías de uso preparadas para usuarios con poco nivel técnico. Idóneo para niños y adultos con ganas de aprender a programar robots de forma sencilla y sin necesidad de gastar un tiempo desorbitado en el entendimiento en profundidad de éstos.

Capítulo 6

Experimentos

En esta sección vamos a demostrar el potencial de la herramienta con algunos ejemplos de su funcionamiento. Validando el funcionamiento de ésta, y sus mejoras funcionales con respecto a la versión de Scratch4Robots de la que partimos.

Además de los ejemplos mostrados en este apartado, se ha creado una batería de pruebas con las que ver un ejemplo de funcionamiento de cada uno de nuestros bloques. Estas pruebas son proyectos Scratch, que el usuario puede cargar en el entorno de Scratch y usar como punto de partida para sus próximos proyectos.

6.1. Evitar obstáculos con Turtlebot

En este ejemplo hacemos uso de un robot con ruedas, para ser más exactos de un Turtlebot. Para ejecutar las pruebas utilizamos Gazebo como simulador, cargando un entorno con obstáculos repartidos como se aprecia en la figura.

El objetivo directo de este experimento es la programación de un robot Turtlebot, capaz de evitar obstáculos de forma autónoma, con la ayuda de un sensor láser incorporado en su parte frontal. Esto lo vamos a conseguir usando únicamente la lógica de la que disponemos en Scratch junto con la extensión que aporta nuestra aplicación Scratch4Robots.

Se puede apreciar con estos ejemplos el poder didáctico de la herramienta. Su simpleza y

sencillez hacen de ella perfecta para la enseñanza a personas con pocos conocimientos técnicos.

Además este ejemplo podemos verlo en nuestro día a día con las famosas aspiradoras autónomas (iRobot Roomba), por lo que vemos una aplicación directa que podría tener nuestra aplicación fuera del ámbito didáctico.

Vamos a seguir el ciclo de vida de nuestra aplicación, explicando el proceso que lleva a la consecución de nuestro objetivo.

En primer lugar y con ayuda de nuestros bloques, desarrollamos la aplicación en Scratch. Una vez tenemos nuestro proyecto acabado y guardado, con la herramienta Scratch4Robots, ya instalada en nuestro equipo y mediante el uso de un solo comando realizaremos la traducción del proyecto.

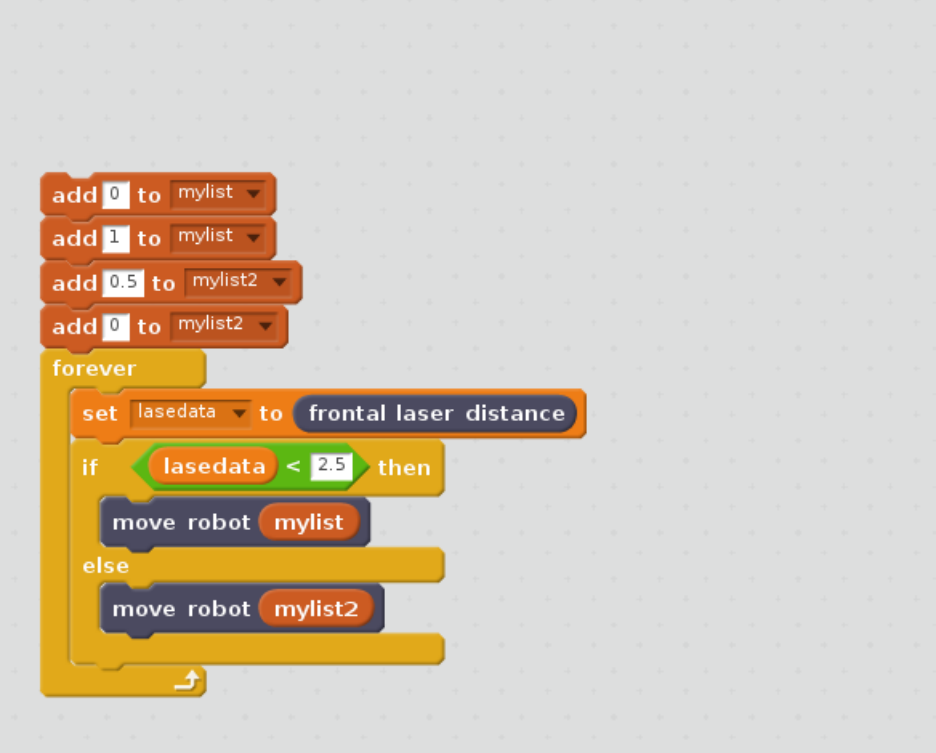

Figura 6.1: Programación en Scratch del Turtlebot

Esta simpleza en la ejecución la conseguimos mediante las funcionalidades que nos aporta ROS para la ejecución de nodos, esto se consigue tras la labor de integración realizada. El comando a ejecutar sería:

```
~$ rosrun scratch4robots scratch2python myscratchaproyect.sb2
```

Explicando un poco el comando anterior, *rosrun* sería el comando que nos aporta ROS, capaz de ejecutar nodos de un paquete ROS, *scratch4robots* es nuestro paquete ya instalado en la máquina, *scratch2python* sería el nodo interno de nuestra aplicación encargada de la traducción a Python, y por último se añade el proyecto Scratch que queremos traducir. Observamos a continuación el código ya traducido a Python que será incrustado en el nodo ROS para su ejecución.

```
        mylist = []
        mylist2 = []
                # Agrega velocidades que realizan el giro del Turtlebot
        mylist.append('0')
        mylist.append('1')
        # Agrega velocidades que hacen que el Turtlebot avance
        mylist2.append('0.5')
        mylist2.append('0')
        while True:
            lasedata = robot.get_laser_distance()
            if lasedata < 2.5:
                robot.move_vector(mylist)
            else:
                robot.move_vector(mylist2)
```

Tras la traducción obtenemos un nodo ROS completamente preparado para su ejecución sobre el robot Turtlebot, únicamente con la dependencia de un fichero de configuración que describiremos a continuación:

```
robot:
  Motors:
    Server: 2 # ROS
    Topic: "/mobile_base/commands/velocity"
    Name: robotMotors
    maxW: 0.7
    maxV: 4

  Laser:
    Server: 2 # ROS
    Topic: "/scan"
    Name: robotLaser

  Pose3D:
    Server: 2 # ROS
    Topic: "/odom"
    Name: robotPose3d

  Camera1:
    Server: 2 # ROS
    Format: RGB8
    Topic: "/cameraL/image_raw"
    Name: robotCamera1

  NodeName: robot
```

En este fichero se observa los distintos actuadores y sensores que necesitará nuestro nodo, además por cada sensor se define el Tópico al que nuestro nodo se subscribe para enviar y recoger los datos que necesitemos de nuestro robot. En este caso darle especial importancia a la obtención de los datos del sensor laser, protagonista de nuestro ejemplo.

Con todo esto preparado solo faltaría ejecutar nuestro nodo sobre la simulación para ver los resultados que podemos apreciar en el siguente detalle.

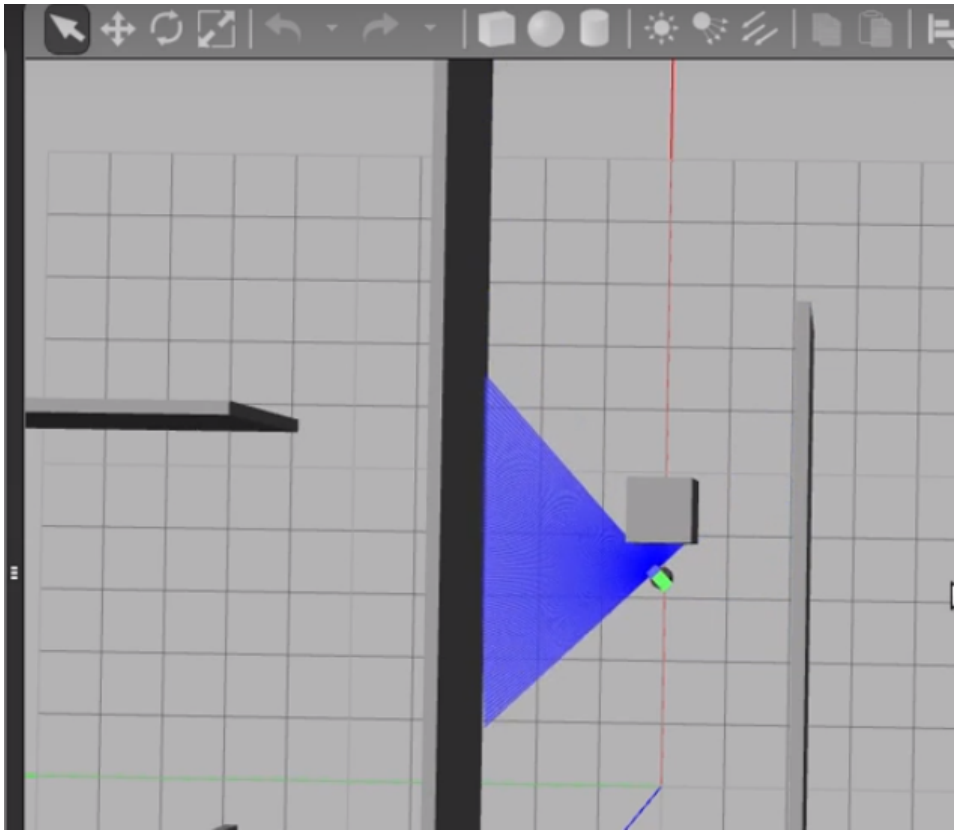

Figura 6.2: Ejecución del nodo sobre Turtlebot

De esta forma, en pocos y sencillos pasos conseguimos programar un robot de forma autónoma para que evite cualquier obstáculo que se encuentre en su camino.

6.2. Persecución entre drones

En este ejemplo por el contrario, como robot final utilizamos un dron, mucho más interesante desde el punto de vista práctico, aunque igual de interesante desde el punto de vista didáctico. Al igual que con la prueba anterior hemos utilizado Gazebo como motor de simulación.

El ciclo de vida es el mismo que el que hemos seguido para la ejecución del robot. Primero se genera el proyecto en Scratch, el cual mostramos en la siguiente imagen:

Figura 6.3: Programación en Scratch del drone perseguidor

Mediante las funcionalidades que nos aporta ROS para el lanzamiento de aplicaciones bajo un solo comando ejecutamos la traducción, obteniendo nuestro nodo programado en Python. Mostramos el detalle de la traducción de este proyecto, la cual se integrará en el nodo final ROS como hemos descrito en apartados anteriores.

```
                mylist = []
                myvelocity = []
        robot.take_off()
        while True:
            mylist.insert(0, robot.color_detection('red'))
```

```
            size = mylist[0][0]
            x = mylist[0][1]
            y = mylist[0][2]
            if size > 0:                    # Tras detectar el objetivo, estima velocidades
                if size > 700:
                    velx = '-2'
                else:
                    velx = '2'
                if x > 165:
                    velyaw = '-2'
                else:
                    velyaw = '2'
                if y > 110:
                    velz = '-1'
                else:
                    velz = '1'
            else:                           # Gira sobre si mismo buscando el objetivo
                robot.stop()
                velx = '0'
                velz = '0'
                velyaw = '2'
            myvelocity.insert(0, velx)
            myvelocity.insert(1, velz)
            myvelocity.insert(2, velyaw)
            robot.move_vector(myvelocity)
```

Y por último, ya con nuestro nodo generado, agregando la configuración pertinente estará todo listo para su ejecución.

```
drone:
  Camera:
    Format: RGB8
    Topic: "/solo/cam_frontal/image_raw"
    Name: UAVViewerCamera

  Pose3D:
    Topic: "/mavros/local_position/odom"
    Name: UAVViewerPose3d

  CMDVel:
    Topic: "/mavros/setpoint_velocity/cmd_vel"
    Name: UAVViewerCMDVel

  Navdata:
    Topic: "/IntrorobROS/Navdata"
```

```
    Name: UAVViewerNavdata

  Extra:
    TopicArming: "mavros/cmd/arming"
    TopicLand: "mavros/cmd/land"
    TopicSetMode: "mavros/set_mode"
    TopicVel: "/mavros/setpoint_velocity/cmd_vel"
    Name: UAVViewerExtra
NodeName: drone
```

Tras ejecutar el nodo junto con el fichero de configuración, obtenemos el resultado deseado, como observamos en la imagen superior. Algo tan complejo como una persecución entre dos drones, totalmente programado en un lenguaje tan simple como Scratch, y esto, es solo una prueba del potencial de esta herramienta.

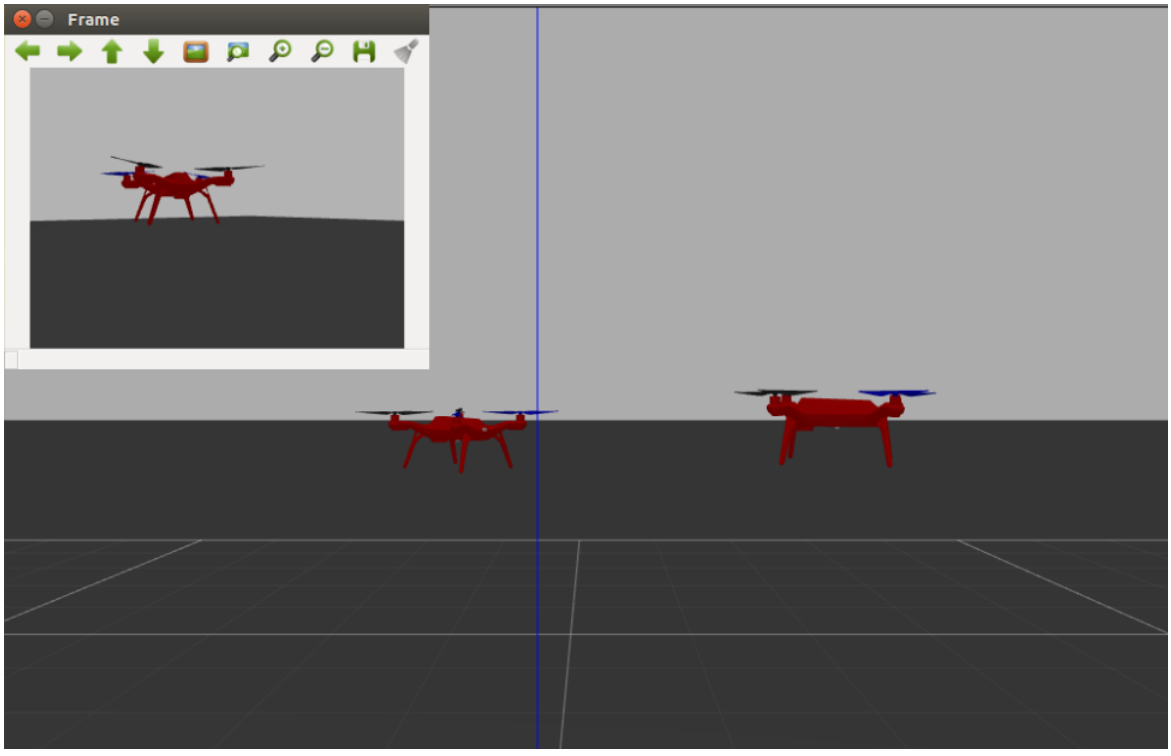

Figura 6.4: Ejecución del drone perseguidor

Con estos ejemplos hemos recorrido el ciclo de vida de nuestra herramienta, explicando su funcionamiento, además de validar experimentalmente todas las mejoras en el funcionamiento de Scratch4Robots 2.0. Las mejoras en la ejecución de la herramienta mediante simples comandos, sin necesidad de indagar en el código fuente. Y completo funcionamiento de la herramienta,

con comunicaciones ROS.

Capítulo 7

Conclusiones

Tras detallar en profundidad las mejoras aportadas a la herramienta Scratch4Robots, este capítulo se ha reservado para ver hasta qué punto se han logrado los objetivos establecidos, para resumir los conocimientos adquiridos durante su desarrollo y para exponer las posibles mejoras que se pueden introducir en trabajos futuros.

7.1. Conclusiones

El objetivo global de este proyecto era la mejora de la herramienta Scratch4Robots 1.0. Este objetivo se ha alcanzado con éxito a través de una serie de subobjetivos. Estos subobjetivos se dividen en la refactorización de la herramienta, y la extensión en funcionalidad de ésta con el desarrollo de nuevos bloques, una paquetización completa de la herramienta, y la validación experimental con robots ROS.

El primer subobjetivo consiste en la refactorización y extensión en funcionalidad. Esto se consigue haciendo modificaciones tanto en el proceso de traducción, en la generación del nodo final ejecutable y en los propios bloques ya existentes en la herramienta. La traducción que aportamos ofrece una mayor capacidad de detección de bloques anidados, además de soportar una mayor cantidad de bloques propios de Scratch. El código generado, ahora es un único nodo ROS programado en Python, con toda la lógica para funcionar de forma autónoma, a este nodo ROS se le aporta la flexibilidad de ser configurado a través de un fichero de configuración, en el que se indican las necesidades de nuestro desarrollo, como pueden ser sensores, actuadores

etcétera. Otra de los cambios que aporta esta refactorización es la adaptación a comunicaciones ROS en su totalidad. En cuanto a la extensión de funcionalidades, aportamos nuevos bloques, como bloque destacado se agrega el bloque perceptivo de detección de objetos de un color determinado, que usa una cámara montada sobre el robot. Se aportan otros bloques de carácter general, como son los bloques matemáticos y lógicos, además de todos los bloques para el manejo de listas. Estos bloques amplían las posibilidades de uso de la herramienta de forma considerable.

En segundo lugar se busca una paquetización completa de la herramienta. Esto se consigue de dos formas. La primera y principal, es la generación de un paquete ROS, instalable en forma de binario desde linea de comandos en un sistema operativo Ubuntu. Este paquete además de ser fácilmente instalable, hace uso de todas las funcionalidades que nos aporta el entorno ROS, como por ejemplo la ejecución de nodos pertenecientes a un paquete ya instalado, desde linea de comandos. Esta funcionalidad es en la que nos basamos para lanzar el nodo que realiza la traducción del bloque Scratch al nodo ROS final. Además de esta paquetización ROS, realizamos otra labor de desacople de dependencias en forma de paquetes pip, igualmente instalables desde linea de comandos. Esto aporta una mayor flexibilidad a la herramienta, pudiendo agregar y modificar funcionalidad sin necesidad de modificar el core de la herramienta.

Por último queremos hacer una validación experimental de la herramienta con robots puramente ROS. Este subobjetivo se lleva a cabo con la total incorporación de comunicaciones ROS, tanto para Turtlebots como para drones. Estas comunicaciones son validadas con una serie de ejemplos funcionales sobre robots simulados.

El objetivo se ha conseguido cumpliendo con los requisitos previos que nos habíamos propuestos. Todo el código se realiza en lenguaje Python. El paquete ROS generado, se trata de un ROS en su versión Kinetic, versión que está enfocada a su uso sobre Ubuntu 16.04, otro de los requisitos que establecíamos. Además no necesitamos de más software, aparte del mencionado anteriormente y el propio Scratch 2.0 en su versión *offline*.

7.2. Trabajos Futuros

El desarrollo de este trabajo ha generado la versión 2.0 de Scratch4Robots, pero aún quedan muchas cosas que se pueden desarrollar para mejorar la herramienta.

El más destacado y de necesaria implementación para que la herramienta se mantenga actualizada, es la integración con Scratch 3.0. Siguiente versión que sale a la luz a principios de 2019. Ayudaría a mantener al día la herramienta con la tecnología del momento. Esto es de vital importancia si buscamos la usabilidad de ésta, una herramienta desactualizada hará que los usuarios se decanten por otras de mayor valor, además de impedirnos el desarrollo de funcionalidades de mayor complejidad por las limitaciones que supone el mantenernos en le versión 2.0 de Scratch. El ser compatibles en un futuro con Scratch 3.0 nos permite crecer tanto cuantitaviamente como cualitativamente, pudiendo ganar más usuarios y además ampliando nuestras funcionalidades, ya que podríamos incorporar funcionalidades como el uso de servicios web que proporciona Scratch 3.0.

Otra mejora sería la ampliación de la gama de robos soportados. En esta versión nos hemos centrado en drones y en Turtlebots, pero existe una completa gama de robots de diferentes características a las que podríamos dar soporte. Uno podría ser el robot humanoide Pepper, muy popular actualmente. Esto supondría la generación de bloques para las necesidades específicas de estos robots.

De cara a su usabilidad, se podría ampliar el repertorio de tutoriales y guías didácticas. Enfocandolas al aprendizaje de la programación de robots sofisticados para usuarios con un perfil técnico bajo.