INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Escuela Técnica Superior en Ingeniería Informática

Curso académico 2009-2010

Proyecto Fin de Carrera

Herramienta de programación visual de autómatas en Jderobot

Autor: Carlos Iván Martín Amor.

Tutor: José María Cañas Plaza.

Espero que este sólo sea el principio de un largo camino.

carlos_123_cima@hotmail.com
jmplaza@gsyc.escet.urjc.es

Agradecimientos.

Quisiera dar las gracias a todas las personas que me han apoyado durante esta larga andadura, sin ellos esto no hubiera sido posible.

En primer lugar quiero darle las gracias a Jose María Cañas, mi tutor durante el proyecto. Gracias por todo el tiempo empleado, por ese espíritu de superación y positivismo que nos contagias en cada tutoría, y por la confianza que depositas en todos nosotros.

A todos mis compañeros del laboratorio de Robótica, que en cualquier momento estan dispuestos a responder dudas y problemas ocasionados durante el desarrollo del proyecto.

Especialmente a *Cidon* por toda la comprensión que ha tenido conmigo, por el apoyo moral que me ha dado en los momentos difíciles, y porque siempre esta ahí cuando la necesito.

Quiero agradecer todo esto a mis padres, que se han esforzado durante toda una vida por darme lo mejor y por ofrecerme las oportunidades que ellos no tuvieron. También quiero destacar a mi hermano, que siempre está ahí haciendome pasar momentos divertidos incluso en los días más hostiles.

A todos mis compañeros y amigos durante la carrera, que me han hecho pasar experiencias inolvidables, sobre todo muchas risas. Especialmente a Jorge, Álvaro, *Canario*, *Reve*, Silvi, Yunta, ...

A mis amigos Cris, Fani, Adri, porque sois fantásticos, sin ninguna duda los mejores.

¡Gracias a todos!.

Resumen.

El comportamiento de un robot viene directamente determinado por la programación realizada sobre el mismo. La gran mayoría de comportamientos referentes a robots móviles se basan en un autómata finito. Estos siempre nos proporcionan un flujo de ejecución frente a un determinado estímulo.

El objetivo de este proyecto es crear *Jde_DS*, una herramienta de programación visual de autómatas en *Jderobot*. Capaz de programar un comportamiento de un robot mediante un autómata finito de estados programando de forma independiente cada estado, y transición. El comportamiento debe de ser compatible con la versión *Jderobot 4.3*.

Se ha logrado cumplir todos los objetivos fijados inicialmente, creando, como han demostrado los ejemplos de uso, una infraestructura ágil y robusta. Se ha desarrollado una plataforma que nos permite construir el autómata de forma gráfica, añadiendo transiciones y estados, y estableciendo el nodo inicial. Los estados corresponden al propio código del comportamiento, sin embargo las transiciones son condiciones que nos permiten pasar de un estado a otro dependiendo de la validación que se quiera realizar.

Para comprobar el correcto funcionamiento y el buen uso de la aplicación, se demuestran tres tipos de pruebas. Estas aplicaciones demuestran una interfaz robusta, y manejable por el usuario, debido al gran número de funcionalidades que la aplicación ofrece. Además se incluye un manual de uso facilitando la creación de comportamientos con la aplicación.

Índice general

1. Introducción **1**

1.1. El mundo de la robótica 1

1.2. La robótica y las partes de un robot 6

1.2.1. Sensores 6

1.2.2. Actuadores 8

1.2.3. Computador 9

1.3. Autómatas 9

1.4. Jderobot: Una plataforma de desarrollo para aplicaciones robóticas . . 10

2. Objetivos y Metodología **14**

2.1. Objetivos 14

2.2. Requisitos 15

2.3. Metodología 15

3. Infraestructura **17**

3.1. Jderobot 17

3.2. Gtk 22

3.2.1. libGlade y GtkBuilder 23

3.2.2. GLib 24

3.2.3. Pango 25

3.2.4. Gdk 26

3.2.5. GdkPixbuf 27

3.2.6. Cairo 27

3.3. LibXml 28

4. Descripción Informática **29**

4.1. Diseño general 29

4.2. Editor gráfico 32

4.2.1. Contenido del autómata 33

4.2.2. Dibujado . . . 38
4.2.3. Deshacer y rehacer . . . 39
4.2.4. Bloque de impresión . . . 40
4.2.5. Visualizacion del autómata . . . 40
4.2.6. Dependencias del autómata . . . 41
4.2.7. Configuración del proyecto . . . 42
4.3. Editor de texto . . . 43
4.3.1. Programación de código . . . 44
4.3.2. Bloques propios del editor . . . 46
4.4. Fichero XML . . . 48
4.4.1. Carga y guardado en XML . . . 49
4.5. Esquema Jderobot . . . 51
4.5.1. Construcción del esquema . . . 51
4.5.2. Estructura del esquema . . . 52

5. Aplicaciones 58
5.1. Manual de uso . . . 58
5.1.1. Manipulación del proyecto . . . 58
5.1.2. Creación del autómata . . . 60
5.1.3. Funcionalidades de JDE_DS . . . 65
5.2. Choca-gira . . . 65
5.3. Navegación híbrida . . . 67
5.4. Sigue Línea con Gazebo . . . 69

6. Conclusiones y trabajos futuros 72
6.1. Conclusión . . . 72
6.2. Trabajos futuros . . . 73

Bibliografía 74

Índice de figuras

1.1. Ejemplos de robots. 2
1.2. Inventos de Jacques de Vaucanson 2
1.3. Brazo *Unimate* desarrollado por G. Devol y Joseph. F. Engelberger . . 3
1.4. Robot PUMA. 4
1.5. Automatización en la industria. 4
1.6. Robots relacionados con exploraciones espaciales. 5
1.7. Robots dedicados a la medicina y al ocio 6
1.8. Ejemplos de sensores 7
1.9. Tipos de actuadores 8
1.10. Autómata finito determinista. 10
1.11. Función del middleware Jderobot 12
1.12. Esquemas desarrollados con *Jderobot* 13

2.1. Metodología del proyecto: modelo espiral. 16

3.1. Esquema de acción de jderobot. 17
3.2. Tipos de esquemas en *Jderobot* 18
3.3. Jerarquía de JDE 19
3.4. Infraestructura de JDE 20
3.5. API de *Jderobot* 20
3.6. Aplicación Glade con la interfaz gráfica de JDE_DS 24
3.7. Editor con multitud de idiomas con pango 26
3.8. Dibujo realizado con Cairo 28

4.1. Relación de partes de *Jde_DS*. 31
4.2. Bloques funcionales de la plataforma. 32
4.3. Explicación de la interfaz JDE_DS. 33
4.4. Zona de creación del autómata 36
4.5. Menús emergentes dentro de la zona de creación del autómata. 37
4.6. Marco de selección de acción. 37

4.7. Ventana de impresión del esquema del autómata 40
4.8. Ventana de visualización del autómata. 41
4.9. Activación de sensores 41
4.10. Marco de muestra de dependencias. 42
4.11. Ventana de configuración del proyecto. 42
4.12. *Widgets* de la selección del estado inicial. 43
4.13. Configuración de la velocidad del esquema. 43
4.14. Interfaz del editor de texto. 44
4.15. Zona de edición de código. 45
4.16. Botones del editor de código. 46
4.17. Menús desplegables de JDE_DS. 47
4.18. Ventana del buscador de caracteres. 48
4.19. Interfaz gráfica del esquema 54

5.1. Ventana de configuración del proyecto. 58
5.2. Advertencia al crear un nuevo proyecto. 59
5.3. Ventana de selección del fichero "XML". 60
5.4. Ejemplo de estado y ventana de propiedades. 60
5.5. Transición y menú de propiedades. 62
5.6. Ventana de selección de tiempo en transiciones temporales. 63
5.7. Representación del estado inicial y el cambio de velocidad del esquema 63
5.8. Barras de herramientas para los sensores y actuadores 64
5.9. Ventana de visualización del autómata. 65
5.10. Autómata del ejemplo choca-gra 66
5.11. Simulación del comportamiento choca-gira 67
5.12. Simulación de la navegación híbrida 68
5.13. Autómata del ejemplo de navegación híbrida 69
5.14. Simulación del esquema sigue línea con *Gazebo* 70
5.15. Autómata del test sigue línea con *Gazebo* 70

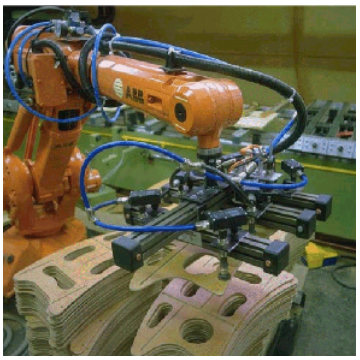
Capítulo 1
Introducción

En este primer capítulo nos adentraremos en el mundo de la robótica, hasta llegar a encontrar la motivación por la que se ha llevado a cabo este PFC y la metodología empleada para su desarrollo.

1.1. El mundo de la robótica

El origen del término robot proviene de la novela "RUR" (Rossum's Universal Robots) del autor checo Karel Capek, publicada en 1920. Este drama trata sobre un brillante científico, Rossum, y su hijo, quienes desarrollan humanos artificiales sumisos. El protagonista sigue trabajando hasta alcanzar el perfeccionamiento de dichos seres, pero al final el argumento da un giro inesperado y todos acaban rebelándose contra los humanos. Capek se dirige a estos seres artificiales mediante la palabra *robot*, derivación del vocablo checo *robota* que significa trabajo obligatorio o forzoso.
Posteriormente Isaac Asimov, escritor y bioquímico, fue quien acuño la palabra *robótica* con el sentido de disciplina científica encargada de construir y programar robots.
En la actualidad, se sigue manteniendo el enfoque de la definición. La Real Academia Española define la *robótica* como *técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales*. Dichos aparatos se les denomina *robots*, que define así: *máquina o ingenio electrónico programable, capaz de manipular objeto y realizar operaciones antes reservadas solo a las personas*.

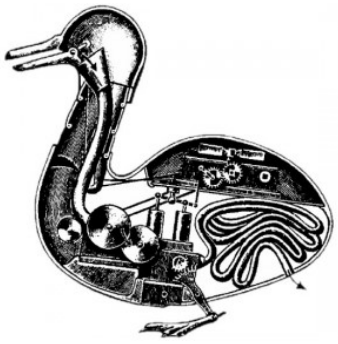
(a) Perrito Aibo de Sony



(b) Brazo transportador

Figura 1.1: Ejemplos de robots.

Como precursores históricos de los robots actuales, tenemos diversos autómatas, diseñados y construidos por grandes mentes avanzadas a su época. Como ejemplos citar el Barco con cuatro músicos robotizados construido por Al-Jazari en 1206, tratándose del primer robot humanoide programable; Leonardo Da Vinci en 1495 diseño un guerrero mecánico capaz de sentarse, mover sus brazos y cabeza, y sonreír; el pato *Digesting Duck* desarrollado por Jacques de Vaucanson en 1738, capaz de bañarse, comer y excretar (figura 1.2(a)), también fue el autor de varios muñecos animados y del primer telar mecánico (figura 1.2(b)).



(a) Digesting Duck

(b) Telar automatizado

Figura 1.2: Inventos de Jacques de Vaucanson

Hoy en día, no podrían existir robots sin el desarrollo de sistemas de computación y la tecnología de control. Estos dos factores experimentaron un fuerte desarrollo durante la segunda guerra mundial. En la década de los 60 fueron fabricados por primera vez los circuitos integrados, fue por aquella época cuando salieron al mercado a gran escala

robots industriales y empezaron a ser utilizados en los EEUU por los fabricantes de automóviles. Las primeras patentes aparecen en 1946, describiendo primitivos robots para el translado de maquinaria. Será en 1954, cuando George Devol, diseñe el primer mecanismo programable aplicado a la industria, el *Universal Automation*. George Devol junto con Joseph F. Engelberger construyeron un robot denominado *Unimate*, abreviación de *Universal Automation*, dispositivo que utilizaba un computador junto con un manipulador que conformaban una máquina que podía ser enseñada para la realización de tareas variadas de forma automática (figura 1.3). En 1962, el primer Unimate fue instalado a modo de prueba en una planta de la General Motors para funciones de manipulación de piezas y ensamblaje, con lo que paso a ser el primer robot industrial. Más tarde formarían la primera compañía comercial dedicada a la robótica, Unimation Inc., dedicada a robots industriales para cadenas de montaje.

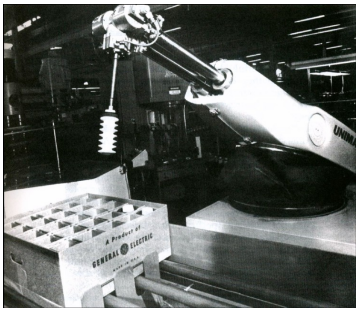


Figura 1.3: Brazo *Unimate* desarrollado por G. Devol y Joseph. F. Engelberger

Se puede considerar este punto como el inicio de la era de la Robótica tal como la conocemos, mediante la utilización de los robots programados, una nueva y potente herramienta de fabricación.

Durante la década de los 70, la investigación en robótica se centra en gran parte en el uso de sensores externos para su utilización en tareas de manipulación. Además se consolida definitivamente la presencia de robots en las cadenas de montaje y plantas industriales en el ámbito mundial. Es en estos años cuando el ingeniero mecánico Victor Scheinman desarrolló un manipulador polivalente realmente flexible conocido como PUMA (Programmable Universal Manipulator for Assembly), capaz de mover un objeto y colocarlo en cualquier orientación en un lugar deseado que estuviera en su alcance. El concepto básico multiarticulado del PUMA es la base de la mayoría de los robots actuales.

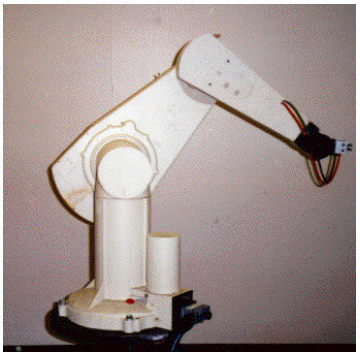
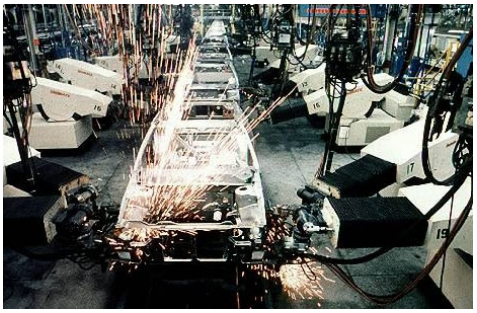
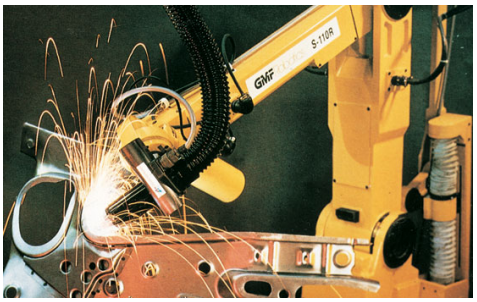


Figura 1.4: Robot PUMA.

Las principales ventajas que ofrece un brazo robotizado frente a un trabajador para estas labores repetitivas son la precisión y la rapidez. Además hay que considerar desde el punto de vista del empresario que los robots no hacen huelga, no tienen días de descanso, pudiendo trabajar las 24 horas del día, todos los días del año. Salvando cierta inversión inicial, su coste de mantenimiento puede ser menor que el de un operador humano.



(a) Automatización



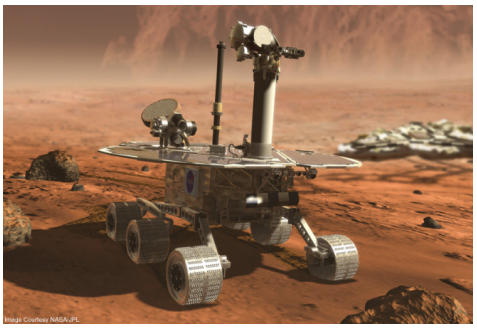
(b) Brazo Industrial

Figura 1.5: Automatización en la industria.

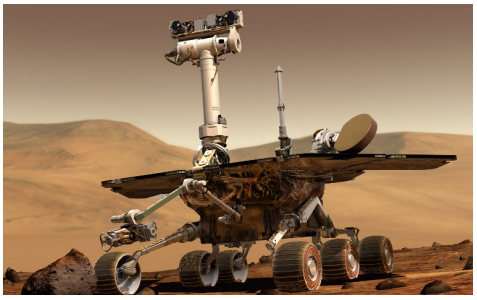
Los desarrollos actuales de robótica persiguen un gran número de objetivos y aparecen en otros escenários fuera de la fábrica. Los robots son usados hoy en día para llevar a cabo tareas sucias, peligrosas, díficiles para los humanos. Por ejemplo, se utilizan robots teleoperados para explorar zonas de alta radiación como en el interior de reactores nucleares, también nos permiten rastrear en las profundidades oceánicas, recuperar personas atrapadas entre escombros, insepccionar y desactivar bombas o minas, etc.. Estos casos el robot suele estar equipado con cámaras de video y es guiado remotamente por un operador humano.

Otros lugares donde los Robots están reemplazando a los humanos son en la exploración espacial. Este campo permite una variedad de funciones, como por ejemplo

pruebas de vuelo planetarias, brazos mecánicos para manipulación de elementos, pruebas atmosféricas o robots terrestres. Uno de los más conocidos es el robot `Pathfinder` (figura 1.6(a)): Teleoperado desde La Tierra, permitió analizar rocas e imágenes de la superficie de Marte. Tras su éxito se enviaron otro robots como el `Mars Exploration Rover` (MER). Ver figura 1.6(b).



(a) PathFinder



(b) Mars Exploration Rover

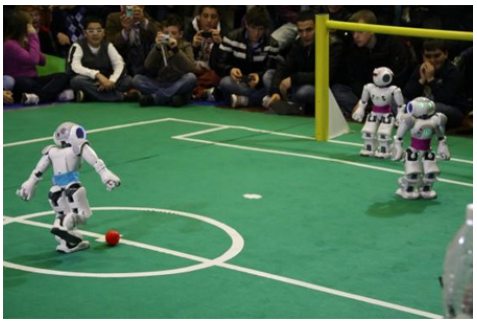
Figura 1.6: Robots relacionados con exploraciones espaciales.

Una de las áreas en la que la robótica se ha implantado fuertemente es la medicina. Se han utilizado en procedimientos de cirugía invasiva mínima, realizando tareas muy delicadas de manera exitosa. En el año 1999 el Intuitive Surgical introdujo a la medicina el *Robot Da Vinci* (figura 1.7(b)). Es un robot creado para asistir al equipo médico para llevar a cabo cirugías complicadas mediante una técnica mínimamente invasiva, con mayor precisión y un mejor post operatorio para el paciente. La automatización de laboratorios también es un área de crecimiento. Aquí, los robots son utilizados para transportar muestras biológicas o químicas entre instrumentos tales como incubadoras, manejadores de líquidos y lectores.

Uno de los campos insospechados de crecimiento de la robótica es el ámbito del ocio y el entretenimiento. Como ejemplo, el perrito Aibo de Sony se vende como mascota y los robots humanoides desarrollados por Honda y Sony en los últimos años avanzan en esta línea. Cabe destacar la RoboCup[1] (figura 1.7(a)), competición internacional anual en el que equipos de robots juegan al fútbol en distintas categorías.

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

(a) Robot Da Vinci



(b) RoboCup

Figura 1.7: Robots dedicados a la medicina y al ocio

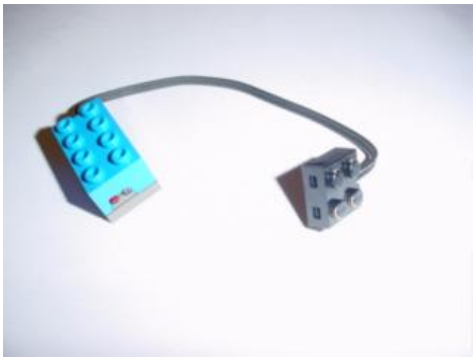
1.2. La robótica y las partes de un robot

La robótica es un área de conocimiento multidisciplinar donde confluyen muchos campos, como la inteligencia artificial, la visión computacional, la mecánica o la psicología.
Una de las ramas más interesantes quizá sea la robótica móvil, cuyo objetivo es lograr que las máquinas realicen de forma autónoma tareas con la flexibilidad y robustez que muestra el ser humano al realizarlas.

Una de las características intrínsecas y axiomáticas de los robots es el comportamiento autónomo. Este atributo se denomina a la habilidad para tomar decisiones por uno mismo y llevarlas a cabo, concretamente en el ámbito de la robótica a la capacidad para percibir la situación y actuar apropiadamente. Llevar a cabo la percepción e interacción de la realidad no es tarea fácil, y por ello se dotan de una serie de herramientas llamadas sensores, actuadores y el controlador.

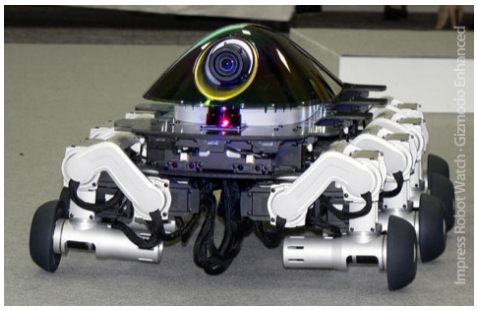
1.2.1. Sensores

Los *sensores* son los dispositivos encargados de detectar un determinado estímulo del exterior y transmitirlo de forma eficiente. Constituyen el sistema de percepción del robot, gracias a ellos podemos captar la realidad, las cantidades físicas de atributos del entorno en el que esta inmerso el robot (temperatura, velocidad, inclinación, distancias, sonido, etc.). Son dispositivos inexactos, ruidosos y limitados, y por ello se emplean varios sensores para una misma propiedad física.



(a) Sensor de luz

(b) Laser



(c) Cámara

Figura 1.8: Ejemplos de sensores

Combinando diversos sensores, llegamos a percibir el entorno con mayor exactitud comparando con un único sensor. No sólo el número de dispositivos nos permiten mejorar nuestar calidad de comunicación con el exterior, sino que depende directamente la precisión de nuestros sensores. Para ello se establecen una serie de características con las que poder comparar los sensores. Podemos caracterizarlos según su *sensibilidad*, ratio de cambio de la salida a los cambios de las entradas; *linealidad*, medida de la constancia del ratio entrada/salida; *rango*, diferencia entre el máximo y el mínimo valor medible; *tiempo de respuesta*, tiempo requerido para que un cambio de la entrada sea observable; *exactitud*, diferencia entre el valor real y el valor medido; *resolución*, incremento mínimo observable en la entrada; *repetitividad*, diferencia entre sucesivas medidas de la misma entrada; *tipo de salida*, tipo de dato que nos devuelve la información.

Todas estas características nos permiten elegir de forma adecuada el sensor más oportuno, dependiendo de las necesidades propias. Para facilitarnos aún más la elección de los sensores, se clasifican en: *internos/externos*, dependiendo si nos facilitan información sobre el propio robot o del entorno en el que se encuentra; *sencillos/complejos*, según la capacidad de cálculo que necesitan para interpretar la información; *activos/pasivos*, dependiendo si producen un estímulo para medir su

interacción con el entorno.

Cabe destacar que un sensor por sí solo no proporciona información interpretable, para ello puede hacer falta mucha capacidad de proceso. En ese ámbito interviene la sensorización de un robot, en el que se implica diversas disciplinas como la electrónica, la informática y el procesamiento de señales. En este proceso se dan todos los pasos necesarios para obtener la información legible a partir de un sensor.

1.2.2. Actuadores

Los *actuadores* son dispositivos capaces de provocar un efecto sobre un proceso automatizado. Son los encargados de generar movimiento de los elementos del robot permitiendo interaccionar con el entorno. Una propiedad que cabe destacar dentro de los actuadores son los *grados de libertad*, que son el número de movimientos independientes que puede realizar un actuador. La mayoría de los actuadores controlan un grado de libertad. Podemos distinguir dos tipos de robots dependiendo de los grados de libertad, los *holonómicos o no holonómicos*. Se dice que un robot es holonómico cuando la cantidad de grados de libertad totales y controlables son los mismos. Por otro lado los robots no holonómicos son aquellos que los grados de libertad controlables son menor que la cantidad de grados de libertado totales, cuanto mayor sea la diferencia mayor es la complejidad para controlarlo. Los tipos de actuadores más comunes que nos podemos encontrar son los neumáticos, hidráulicos y eléctricos.

(a) Neumático

(b) Hidráulico

(c) Eléctrico

Figura 1.9: Tipos de actuadores

Los primeros se caracterizan por ser seguros y robustos, aunque son dificiles de controlar debido a la capacidad de compresión del aire, y a la medición inexacta de la presión. Los actuadores hidráulicos tienen un control muy preciso, además dan una gran estabilidad en estático, sin embargo tienen el problema de fugas obligando a un mantenimiento mayor. Y por último los de tipo eléctrico, nos proporcionan poca fuerza

en comparación con los anteriores, pero su gran característica es la gran velocidad que nos ofrecen.

1.2.3. Computador

Gracias a sensores y actuadores, los robots tienen las herramientas necesarias para poder interaccionar con el entorno, ofreciendo respuestas a determinados estímulos. Son partes de gran peso en el robot, pero por sí solos no hacen del robot un ser autárquico, por ello no es un elemento anodino el computador y la programación que se lleva a cabo.
Los procesadores, son los dispositivos *hardware* que coordinan la actuación y la percepción del robot. Dicha coordinación viene directamente determinada por la programación ubicada en el *hardware* de cómputo.

1.3. Autómatas

Los sensores, actuadores, procesadores y la programación son los cuatro ingredientes indiscutibles que dotan al robot de autonomía, tal vez por ello se habla de robot como autómata.

La definición de *autómata* según la Real Academia Española es la siguiente: [1]*instrumento o aparato que encierra dentro de sí el mecanismo que le imprime determinados movimientos*,[2]*máquina que imita la figura y los movimientos de un ser animado*. Ambas definiciones corresponden a las características de los robots, pero podemos obtener una explicación más técnica de lo que es un autómata. En el campo de la robótica, nos encontramos con autómatas finitos deterministas. Un *Autómata Finito Determinista*(AFD) es un máquina de estados que tiene acceso a una secuencia de símbolos de entrada, se encuentra en cada momento en un estado determinado y puede llegar a transitar a otro estado, y para ello se realizan lo siguientes pasos: 1)Se lee de la cinta y se avanza la cabeza lectora, 2)En función del símbolo leído y del estado actual el autómata transita a otro estado. Un AFD es una quíntupla A = (Q, Σ, f, q_0, F), donde: Q es el conjunto de estados; Σ es el alfabeto de entrada; $f : Q \times \Sigma \Rightarrow Q$ es la función de transición; $q0 \ni Q$ es el estado inicial; y $F \subseteq Q$ es el conjunto de estados finales.

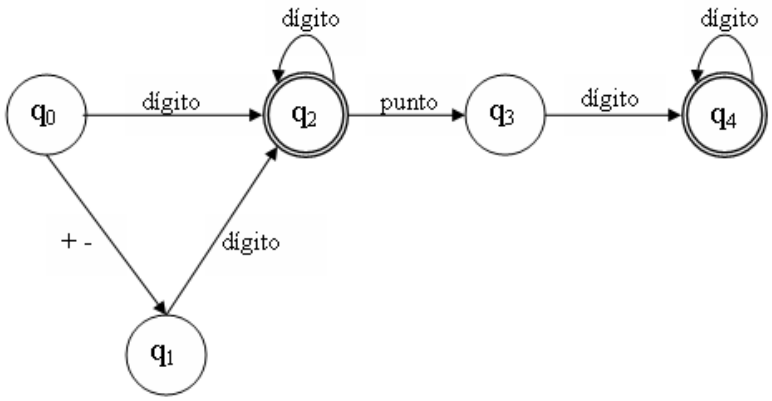


Figura 1.10: Autómata finito determinista.

La relación de un Autómata Finito Determinista con los robots es clara, los sensores corresponden con los símbolos de entrada (símbolos leídos), los estados corresponden con los movimientos o acciones que se desenvuelven en circunstancias concretas, el alfabeto concuerda con los posibles valores que nos ofrecen los sensores, no obstante el conjunto de estado finales no tiene porque existir debido a que un robot siempre se encuentra activo, detectando estímulos del exterior e intentado actuar en su entorno, preservando la característica de autonomía.

1.4. Jderobot: Una plataforma de desarrollo para aplicaciones robóticas

Una vez ubicados y comprendiendo que un robot es equivalente a un autómata, podemos sacar en conclusión que a partir de un AFD podemos programar un robot con comportamiento autónomo. Sin embargo, escribir programas para robots móviles es una tarea complicada, ya que los robots son sistemas complejos. La programación suele ser más exigente debido a que el software debe ser ágil, requeriendo una actuación en tiempo real para tomar decisiones con vivacidad y así controlar correctamente a los actuadores.

El programa de un robot tiene que atender a muchas cosas simultáneamente: recoger nuevos datos de varios sensores, refrescar la interfaz gráfica, enviar frecuentemente consignas a los motores, enviar o recibir datos por la red, etc.. Por ello estas aplicaciones suelen ser concurrentes, lo cual les añade cierta complejidad.

Con todo ello, los programadores se enfrentan a una heterogeneidad de dispositivos. Existe un gran número de dispositivos *hardware* disponibles en la robótica, aumentando

de forma considerable el número de interfaces. El programador debe manejarlos para acceder a ellos desde las aplicaciones. Por otro lado, en el campo de la robótica no hay un marco homogéneo ni de estándares que propicien la reutilización de código como en otras áreas de la informática donde se pueden utilizar librerías. En robótica cada aplicación debe prácticamente construirse desde cero para cada robot concreto.

Antiguamente los programas de control de robots se solían construir empleando directamente los *driver* para acceder a sus sensores y actuadores. La aplicación interaccionaba de forma directa con los *driver* que daba la aplicación.
Actualmente los programas con robots tienen mayor complejidad ofreciendo mayor funcionalidad. Para simplificar notablemente la programación de los mismos se han ido incorporando *middleware*, cuyo objetivo es ofrecer contextos nítidos, estructuras de datos predefinidas, bloques depurados de código de uso frecuente, protocolos estándar de comunicaciones, etc.. A medida que el desarrollo de software para robots móviles ha ido madurando han ido apareciendo diferentes plataformas *middleware*.

Hoy en día nos encontramos una gran diversidad de plataformas de desarrollo para simplificar la programación de los robots, estas normalmente son desarrolladas por los fabricantes de los mismos. Agunos ejemplos son: la plataforma ARIA[2] creada por ActivMedia para sus robots Pioneer; Sony ofrece OPEN-R[3] para sus Aibo.

Entre el gran número de *middleware* que hay en el mercado, cabe destacar *jdec*[4], fruto de una tesis doctoral, elaborado por José María Cañas, sobre generación de comportamientos autónomos en robots móviles. Desde sus inicios ha ido evolucionando adquiriendo mayor número de funcionalidades. Actualmente llamado *Jderobot* es la plataforma por escelencia para la investigación del grupo de Robótica-URJC y para la docencia de robótica en la Universidad Rey Juan Carlos.

La aplicación robótica obtiene las últimas observaciones sensoriales leyendo las variables perceptivas que la plataforma nos ofrece. Ordena comandos a los motores escribiendo en las variables de actuación, que la plataforma se encarga de materializar y hacer llegar hasta los actuadores correspondientes. *Jderobot* mantiene activo esta interfaz de variables situando el código programado encima, leyendo continuamente los datos sensoriales de ese API y escribiendo en él los valores necesarios para el movimiento de los motores.

[2]http://www.activrobots.com/SOFTWARE/aria.html
[3]http://support.sony-europe.com/aibo/index.asp
[4]http://jde.gsyc.es/index.php/Main_Page

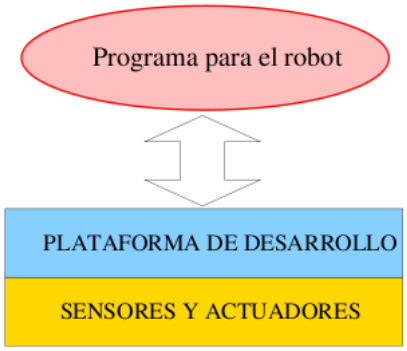


Figura 1.11: Función del middleware Jderobot

Las aplicaciones robóticas que se ejecutan bajo *Jderobot*, se les denominan esquemas concurrentes. Un *esquema* no es más que una hebra con cierto objetivo, que se ejecuta iterativamente a un ritmo controlado. Dicha hebra contiene el código del comportamiento programado, las variables perceptivas y de actuación necesarias para llevar a cabo el comportamiento del robot.

En aplicaciones complejas, *Jderobot* nos ofrece la posibilidad de dividir la funcionalidad en esquemas perceptivos y de actuación concurrentes todos al mismo nivel. Los esquemas perceptivos tratan los datos sensoriales elaborando nueva información ofreciendolo a los demás esquemas, no obstante los esquemas de actuación recogen la información sensorial o de otros esquemas perceptivos y toman decisiones de actuación.
En aplicaciones más complicadas *Jderobot* sugiere organizar los esquemas en una jerarquía de varios niveles, permitiendo inicializarlos, detenerlos, relanzarlos durante la ejecución.

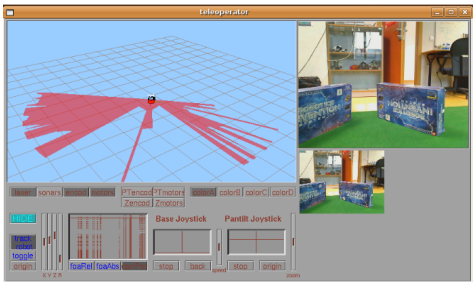
Como característica primordial destaca la encapsulación de la funcionalidad en esquemas concurrentes que se activan y modulan entre sí. Esta división tiene su sustrato teórico en la arquitectura cognitiva JDE para generar comportamiento autónomo de robots.

Algunos esquemas de gran éxito y de mayor uso dentro del grupo de robótica de la URJC, con el middleware *Jderobot* son *Teleoperator*, *FollowBall*, *NaoOperator*, etc..
El *Teleoperator* es un esquema destinado para teleoperar un robot y visualizarlo en un canvas en 3D (figura 1.12(a)). Las mediciones sensoriales del robot se muestran en la imagen. A través de la interfaz gráfica que ofrece podemos visualizar la información y controlar la información del esquema. Este esquema es aplicable tanto con un simulador

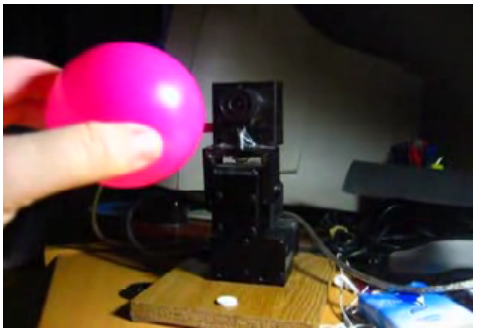
como a un robot real de forma directa. Esto es gracias a la posibilidad que nos pemrite *Jderobot* de repartir las funcionalidades entre esquemas.

FolowBall como su nombre indica es un esquema que nos permite seguir una pelota (figura 1.12(b)). Este esquema se encarga de filtrar la imagen que recibe de la cámara permitiendo mover los actuadores del cuello mecánico enfocando a la pelota a seguir.

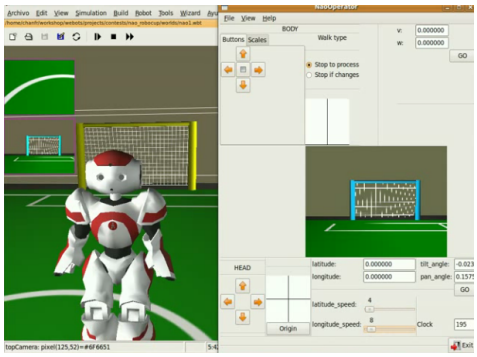
Por último *NaoOperator* es un esquema en vías de desarrollo que por el momento nos permite teleoperar el Robot Nao, destinado para la participación en la RoboCup. La interfaz que ofrece el esquema nos permite recibir la imagen de la cámara del robot y una serie de botones con el que poderle manejar (figura 1.12(c)).



(a) Teleoperator



(b) FollowBall



(c) NaoOperator

Figura 1.12: Esquemas desarrollados con *Jderobot*

Capítulo 2
Objetivos y Metodología

La gran afluencia de proyectos, prácticas e investigaciones del grupo Gsyc ligados al ámbito de la robótica obliga a trabajar de forma continua con *Jderobot*. Esta plataforma como se comento con anterioridad, nos aporta grandes ventaja a la hora de programar el comportamiento de un robot, ya que nos ofrece interactuar con el hardware del robot de un modo muy simplificado. A partir de diversos esquemas nos aporta el acceso a los sensores y actuadores mediante variables, facilitanto la programación.

Sin embargo, la programación de estas estructuras para definir un comportamiento se hace tediosa, debido a la configuración de forma repetitiva de las variables provenientes de otros esquemas aportándonos datos útiles de los sensores y actuadores. El hecho de tener que estar configurando cada esquema nos perjudica de forma sustancial con pérdidas de tiempo, con un trabajo repetitivo en muchos casos, y que nos puede llevar a confusiones en la programación. Sin ninguna duda nos dispersa de nuestro objetivo, que es programar nuestro propio comportamiento.
A parte de la carga que nos supone realizarlo de forma manual, la programación directa en el fichero del esquema es poco intuitiva y engorrosa, nada parecido a un autómata. Todo esto dificulta nuestra tarea en la elaboración de comportamientos compatibles con *Jderobot*.

La motivación para el desarrollo de este proyecto, sin ningún lugar de dudas, es establecer un primer contacto involucrado en mejorar la creación de esquemas de forma simple e intuitiva, definiendo un comportamiento, un autómata.

2.1. Objetivos

Habiendo ya recalcado las motivaciones de este proyecto, procedemos a enumerar los objetivos perseguidos para la elaboración de la plataforma *Jde_DS*:

1. Desarrollar un entorno gráfico con la habilidad de construir un autómata de forma gráfica, con sus nodos, transiciones, y estado inicial.

2. Incluir en el desarrollo de la aplicación la capacidad de programar cada estado y transición, además incorporando una sección para el código global. También se añadirá la funcionalidad para programar una transición temporal.

3. Desarrollar el cometido de construir un esquema compatible con *Jderobot* de forma automática mediante la aplicación.

4. Añadir a la construcción del esquema la creación de una interfaz gráfica para el mismo. Con el objetivo de representar en que nodo se encuentra la ejecución del esquema.

5. Comprobación del buen funcionamiento del esquema mediante pruebas pertinentes.

2.2. Requisitos

Una vez conocido todos los objetivos que se pretenden alcanzar al culminar el proyecto, hay que describir que requisitos mínimos tiene que cumplir la aplicación realizada.

- La aplicación debe ser robusta y estable, debido al gran número de estados o transiciones que pude construir el usuario.
- El entorno gráfico en el que se envuelve la aplicación debe ser intuitivo y manejable, facilitando al usuario su utilización.
- El esquema generado por *Jde_DS* debe ser compatible con la versión Jderobot4.3.. Además debe ser ligero para su ejecución.

2.3. Metodología

Para el desarrollo de este proyecto, se ha seguido un plan de trabajo basado en el modelo espiral (figura 2.1). En el que cada iteración representa un conjunto de actividades. Se ha elegido este tipo de metodología por su flexibilidad, ya que permite planear los siguientes "item" a corto plazo basados en los últimos resultados. Con cada iteración a través del espiral se construye sucesivas versiones del software cada vez más completas. Además con este modelo de trabajo , se pueden suplir carencias de conocimiento y realizar pruebas parciales del software de una forma más natural que planificando el proyecto completo desde el principio.

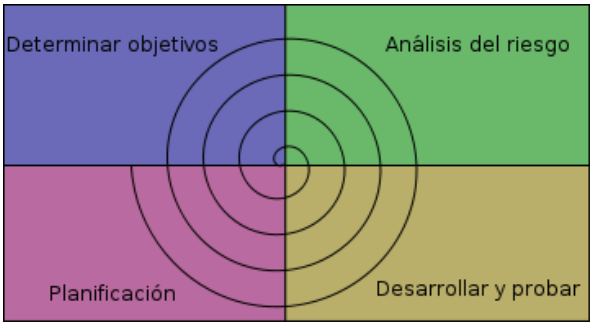


Figura 2.1: Metodología del proyecto: modelo espiral.

La coordinación del tutor con el alumno ha consistido en un seguimiento semanal. En el que cada tutoría se ha evaluado los "item" realizados, y se han fijado los hitos y objetivos a cumplir. Gracias a dicha coordinación se ha ido avanzando con pasos firmes, consiguiendo los objetivos marcados.

Las líneas generales de actuación que coinciden con cada iteración del modelo empleado son la siguientes:

1. Familiarización con la plataforma *Jderobot*. Elaboración de esquemas simples para conocer el desarrollo de los esquemas y su funcionamiento.

2. Aclimatación de la aplicación *Glade* y de las librerías Gtk+, adquiriendo los conocimientos mínimos para el desarrollo de la interfaz gráfica de la aplicación.

3. Estudio y asentamiento de la teoría de autómatas

4. Creación de la interfaz de *Jde_DS* con todas sus funcionalidades.

5. Elaboración del esquema de forma automática conectando todos los sensores y actuadores.

6. Realización de pruebas para el correcto funcionamiento de la aplicación. Este paso se realiza a lo largo del desarrollo, permitiendo la incorporación o eliminación de funcionalidades a la plataforma, y verificando el buen funcionamiento.

Capítulo 3
Infraestructura

En este capítulo se tratarán los pilares en que se basa JDE_DS. Se describirán cada unas de las bibliotecas utilizadas en el desarrollo de la aplicación.

3.1. Jderobot

Jderobot es un entorno de desarrollo de software, escrito en C y C++, para aplicaciones robóticas. Proporciona un entorno de programación donde el programa de control del robot se abastece de una colección de procesos simultáneos asíncronos con el nombre de esquemas.

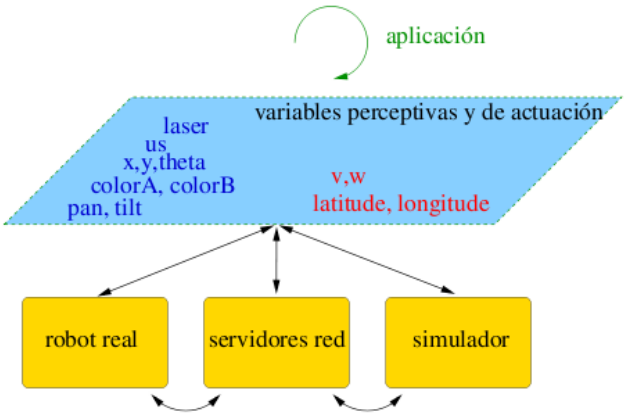
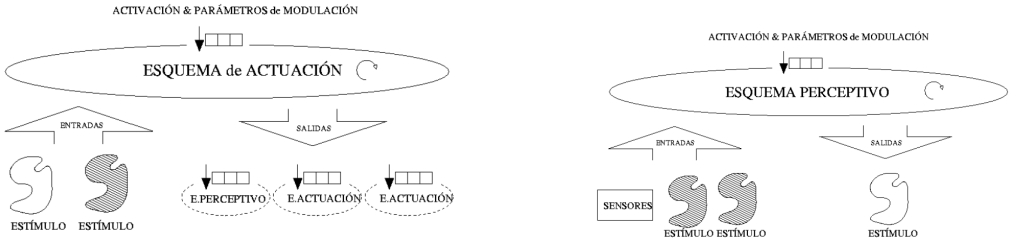


Figura 3.1: Esquema de acción de jderobot.

El esquema es la unidad básica de *Jderobot*, nos permite encapsular una determinada funcionalidad, con el objetivo de formar comportamientos mediante la agrupación de varios esquemas. Definimos esquema como un flujo de ejecución independiente con un objetivo. Podemos distinguir dos tipos de esquemas, esquemas perceptivos y esquemas motores o de actuación. Los esquemas perceptivos se encargan de elaborar los estímulos, que pueden ser leídos por otros esquemas. Las entradas de estos esquemas pueden ser los sensores del sistema, o la percepción de otros esquemas. Por otro lado los

esquemas de actuación, utilizan los datos obtenidos por otros esquemas perceptivos, para generar sus salidas, que pueden ser comandos directos hacia los actuadores del sistema o simplemente envían señales de activación y/o modulación a otros esquemas.



(a) Esquema perceptivo (b) Esquema actuador

Figura 3.2: Tipos de esquemas en *Jderobot*

Los esquemas, son por definición modulables, permitiendo el ajuste de su funcionamiento interno con dependencia de una serie de parámetros, variando el comportamiento de dicho esquema según que esquema de nivel superior lo module. Otra esencia importante es su iteratividad. Cumplen con su objetivo en iteraciones que se ejecutan periódicamente. Dicho periodo, es un parámetro de modulación principal, que ajusta la cadencia de ejecución del esquema. Cabe destacar, que un esquema puede ser activado o desactivado a voluntad, dejando de producir salidas y de consumir recursos. Esta última característica, se consigue asociando estados a cada esquema. Los esquemas perceptivos pueden encontrarse en estado *Dormido/Activo*. En el estado *Dormido*, el esquema se encuentra suspendido sin producir, ni consumir nada, mientras que en el estado *Activo*, ejecuta periódicamente su iteración. Sin embargo los esquemas motores o actuadores, poseen más estados, *Dormido/Alerta/Preparado/Activo*. Los estados que tienen en común con los esquemas perceptivos tienen la misma funcionalidad. El estado *Alerta* matiene al esquema comprobando sus precondiciones. Estas precondiciones evitan que un esquema se active cuando la situación es muy diferente de aquella para la que ha sido programado, asegurandose que la acción se ejecuta en un contexto conveniente. En *Preparado*, el esquema compite por el control contra el resto de esquemas actuadores de su mismo nivel.

La utilización de esquemas nos permiten dividir la complejidad de los problemas en subproblemas más acotados. Esta fragmentación hace más fácil reuitlizar el código y permite ejecutar distribuidamente varios esquemas en diferentes procesadores.

Una vez conocido cual es la unidad básica de *Jderobot*, falta por determinar el

tipo de distribuición que tiene cada esquema dentro de la plataforma. Los esquemas se organizan bajo una jerarquía, ofreciendo la capacidad de utilizar la funcionalidad de otros esquemas para realizar la tarea encomendada por medio de activación y modulación continua de sus hijos. Esta activación puede ser repetida recursivamente en tantos niveles como se necesite, de modo que los esquemas de bajo nivel son despertados y modulados por los de los niveles más altos.

En la jerarquía de *Jderobot* se considera la misión del hijo como un paso en plan secuencial del padre, entendiendose la jerarquía como una co-activación que solo expresa *predisposición*. Por ejemplo un padre puede despertar a varios hijos, pero no todos ellos toman el control del robot, sino que se encuentran en alerta y compiten por realizar su iteración.

JDE tiene una jerarquía de activación de la colección de esquemas, proporcionando ventajas como una complejidad acotada para la selección de acción, acomplamiento entre acción y percepción y monitorización distribuida, sin mal gastar reactividad en entornos dinámicos.

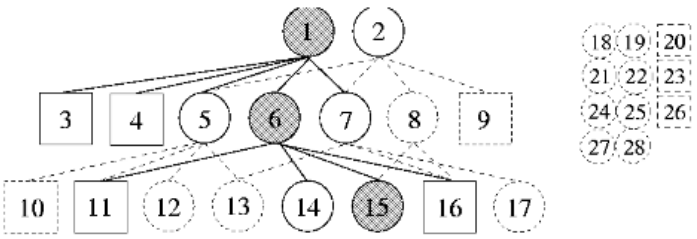


Figura 3.3: Jerarquía de JDE

La plataforma *Jderobot* es una implementación de la arquitectura cognitiva de JDE. La arquitectura es la organización de sus capacidades sensoriales, de procesamiento y de acción para conseguir un repertorio de comportamientos inteligentes interactuando con cierto entorno. Este espacio de desarrollo simplifica el acceso al hardware robótico del programa que lleva el control.

Esta implementación divide la aplicación en dos niveles: la capa de abstacción del hardware y al capa de control. La primera proporciona un interfaz estable que abstrae los sensores y actuadores del robot, ya sea real o simulado. La segunda capa ofrece el marco de trabajo en el que los esquemas interactuan y se comunican.

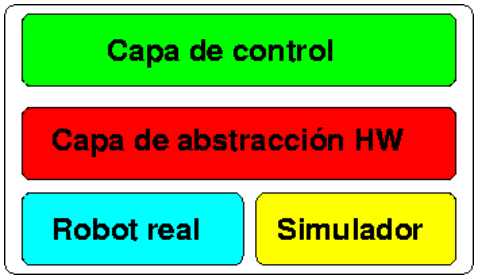


Figura 3.4: Infraestructura de JDE

Todo el hardware es abstraido y presentado como una serie de variables compartidas de lectura o escritura. *Jderobot* recoge continuamente los datos de los sensores y los escribe en las *variables sensoriales* para que los esquemas puedan acceder a ellos. También recoge las órdenes a los actuadores que los esquemas escriben en las *variables motoras* y las envía a los dispositivos a traves del "driver" correspondiente.

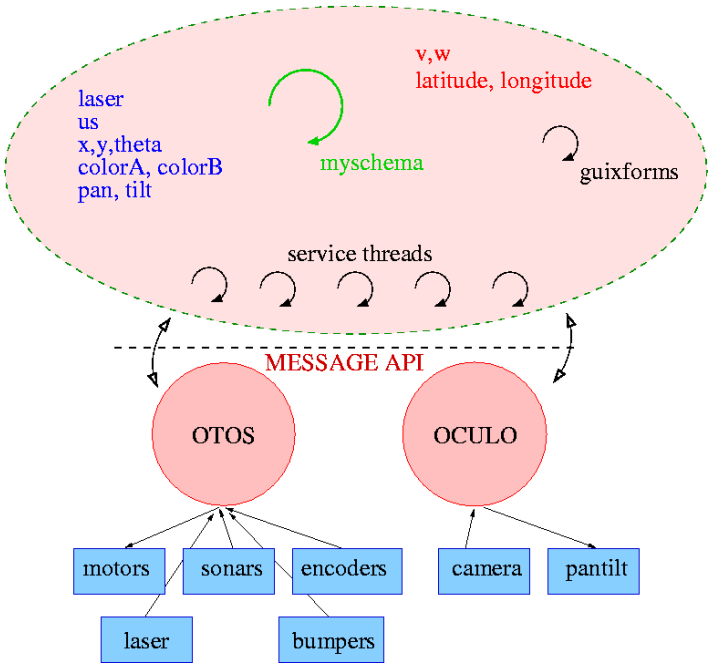


Figura 3.5: API de *Jderobot*

Jderobot contiene un fichero de configuración de texto, con extensión ".conf", para especificar la configuración particular del robot real o simulado para cada aplicación. En ese fichero se especifica en qué dispositivo local se encuentra cada sensor o actuador del robot, o en qué máquinas se encuentran los servidores de red, etc..

A continuación muestro un ejemplo muy básico del fichero de configuración para la ejecución del esquema dentro de *Jderobot*:

```
 ### PATH WHERE THE MODULES CAN BE FOUND
path .:/usr/lib:/usr/lib/jderobot/services/:
/usr/lib/jderobot/schemas/:/usr/share/jderobot/glade:
/usr/lib/jderobot/drivers/

### DRIVER CONFIGURATION
driver player
hostname localhost port 6665
##initial position X(mm) Y(mm) Theta(deg)
initial_position -6500 -6500 45
provides laser 180 1
provides sonars
provides encoders
provides motors
#provides bumpers
end_driver

### SCHEMA CONFIGURATION

### DRIVER LOAD FOR SENSOR SOURCES AND MOTOR DEVICES ------
load_driver graphics_gtk
load_driver graphics_xforms
load_driver player

### SCHEMA LOAD -------
load mastergui
load myschema

### INITIAL ACTIVATION----
#run sonars
#run introrob
run mastergui

### INITIAL GUI ACTIVATION ----
```

Para concluir sobre *Jderobot*, cabe mencionar los ficheros y las partes importantes que hay que desarrollar para tener un esquema compatible con *Jderobot 4.3*. Todos los esquemas están definidos por tres ficheros: *schema.c*, *schema.h* y *Makefile*. El *schema.c* contiene el código de las funciones que permite administrar el esquema. El fichero *schema.h* tiene las funciones de administración del esquema más importantes. Las funciones son las siguientes: *schema_cycle*, establece la velocidad de ejecución del esquema; *schema_iteration*, se encuentra el código que se ejecuta concurrentemente cuando el esquema se encuentra en modo ejecutado; *schema_init*, se encarga de cargar el

esquema en un nuevo hilo para el JDE; *schema_terminate*, destruye el hilo del esquema dentro del JDE; *schema_stop*, pone el esquema en modo espera; *schema_run*, coloca el esquema en ejecución; *schema_show*, muestra la interfaz gráfica del esquema si lo tiene; *schema_hide*, se encarga de cerrar el GUI del esquema si lo tiene. Por último el fichero *Makefile* se encarga de definir las directrices del compilado del esquema para generar el código *schema.so* que se ejecutará dentro del entorno *Jderobot*.

3.2. Gtk

GTK+ es una biblioteca gráfica (toolkit) sobre la que se sustenta todo el interfaz gráfico de Gnome. Es una biblioteca que contiene todo lo necesario para el desarrollo de interfaces gráficas. Los elementos, con el que el usuario puede interactuar, se denominan *widgets*, que son los artifices del aspecto final de la aplicación. Nos permiten la posibilidad de crear todo tipo de *widgets*, desde los más básicos como botones, etiquetas, hasta widgets más complejos como selectores de ficheros, cajas de texto multilínea, etc.

Un *widget* es un componente software visible y personalizable.
Visible porque está pensado para ser usado en los interfaces gráficos de los programas y personalizable porque el programador puede cambiar sus propiedades. La biblioteca Gtk+ está escrita en C, pero la implementación de los widgets utiliza conceptos propios de la programación orientada a objetos, como la herencia; cada *widget* hereda todas las funciones y variables de su predecesor.
La jerarquía de objetos comienza en *GObject* de la librería *Glib* del que hereda *GtkObject*. Todos los *widgets* heredan de la clase de objetos *GtkWidget*, que a su vez hereda directamente de *GtkObject*. La clase *GtkWidget* contiene las propiedades comunes a todos los *widget*; cada *widget* particular le añade sus propios atributos.
Los *widgets* se definen mediante punteros a una estructura *GtkWidget*. Todos estos objetos tienen una relación padre/hijo entre sí. Todo *widget* debe de tener un padre, llamado *contenedor*.

En GTK+ se usa el *modelo de programación por eventos*. Este modelo consiste en la asociación de determinadas acciones ("eventos") a determinadas operaciones ("funciones"). Dichos eventos marcan un cambio de estado del determinado objeto. Los eventos en GTK se refieren a señales especificadas en la definición de las clases de los distintos *widgets*.
El programador es el encargado de decidir cómo se debe reaccionar ante una

determinada acción. Debe enlazar una función (*callback*) a una determinada señal. Así, en cuanto se reciba dicha señal, se llamará automaticamente al *callback*.

Para que este tipo de sistemas sea fructuoso se usa *el bucle de eventos*. Este no es más que un bucle interno de GTK+, en el que se van comprobando, una y otra vez, los estados de cada uno de los elementos de la aplicación.Este concepto se traduce básicamente en dos funciones, *gtk_main(void)* y *gtk_main_quit(void)*.
Estas dos funciones representan el interfaz que tienen las aplicaciones para usar el bucle de eventos. La función *gtk_main(void)* ejecuta dicho bucle, esto significa que, una vez que se haya realizado la llamada se cede todo el control de la aplicació a GTK. No retorna hasta que dentro de algún manejador se haga una llamada a *gtk_main_quit(void)*, y termina el bucle de eventos de GTK+.

Dentro de GTK+ podemos encontrar unas seria de bibliotecas de las que depende, por antonomasia nos encontramos: GLib, Pango, Atk, GdkPixbuf, Gdk, Cairo.

3.2.1. libGlade y GtkBuilder

Crear una interfaz gráfica es sencillo, pero muy costoso en cuanto a codificación se refiere. Para hacer más eficiente este trabajo existen herramientas que automatizan muchas tareas y nos permiten un diseño visual de layout de la aplicación, nos aporta un entorno WYSIWYG (What You See Is What You Get).
Para definir interfaces GTK+ existe Glade (, que nos permite:

- Crear ventanas y cuadros de diálogo.
- Añadir *widgets* a la ventanas.
- Definir las propiedades de cada *widget*.
- Relacionar funciones a eventos determinados de cada *widget*.

Para enlazar la interfaz gráfica creada por Glade con una aplicación en C existen dos formas. Exportar el diseño creado en un fichero con código C, para enlazarlo posteriormente en tiempo de compilación. Esta vía permite tiempos de carga ligeros, sin embargo nos permite un menor control sobre el código generado automáticamente, ya que no debemos manipular el código generado por Glade. La segunda vía guarda el diseño en un fichero en formato XML, y se carga en la aplicación en tiempos de ejecución. Las bibliotecas que nos permiten cargar la interfaz en tiempo de ejecución son dos: *libGlade* y *GtkBuilder*.

La utilidad es la misma para ambas bibliotecas, nos aportan un conjunto de primitivas para cargar el fichero, iniciarlo, obtener *widgets* a partir de un nombre, o enlazar de forma automática cada señal con su función *callback* correspondiente.
La diferencia entre las dos bibliotecas es que *libGlade* no forma parte de *GTK*, sino que sigue un desarrollo completamente independiente. Eso limita en muchos aspectos el mantenimiento de aplicaciones que desarrollemos en referencia a GTK por ejemplo, velocidad a la hora de soportar nuevos widgets y propiedades. Aunque *libGlade* siempre funciono bien no tiene sentido que una biblioteca tan fundamental fuese un elemento externo.
Por todo esto los desarrolladores de *GTK* optaron por crear una biblioteca integrada en *GTK* y que fuese lo más parecida a *libGlade*, con el objetivo de reemplazarla se desarrolló *GtkBuilder*. La forma de comportarse *GtkBuilder* es muy similar a *libGlade*, la principal diferencia es que en *GtkBuilder* no es posible generar sólo una parte del árbol. Esto significa que es más eficiente poner cada ventana en un fichero XML independiente.

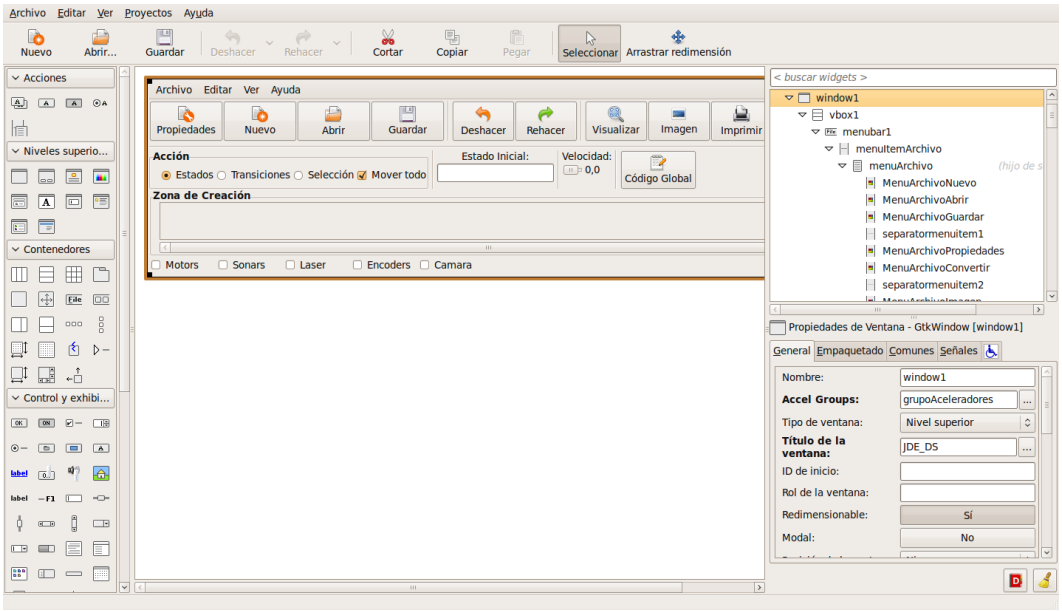


Figura 3.6: Aplicación Glade con la interfaz gráfica de JDE_DS

3.2.2. GLib

GLib proporciona los bloques básicos para construir aplicaciones y bibliotecas escritas en C. Proporciona el sistema de objetos básicos usado en GNOME, la implementación del bucle principal, y un gran conjunto de utilidades para cadenas y estructuras de datos comunes. Tiene la propiedad de mejorar la portabilidad de nuestros programas.

Debido a las diferencias entre sistemas operativos/arquitecturas en cuanto a los tamaños de tipos de datos se refiere, se vuelve a veces complejo el hacer que los programas funcionen sin cambios de S.O. o hardware. Por ello ofrece una serie de tipos de datos propios, sin embargo no son muy diferentes de los que se usa en le estándar de C.
Podemos ver algunas diferencias en la nomeclatura de los tipos de datos

Tipos GLib	Tipos estándar C
gchar	char
gint	int
gshort	short
glong	long
gfloat	float
gdouble	double

Cuadro 3.1: Tipos de GLib equiparando al estándar C.

Esta biblioteca también alberga otros tipos de datos o funciones de gran interes, caben destacar:

- Conversión de tipos.
- Cadenas de texto: GLib tiene sus propios string, del tipo *GString*. Además vienen acompañadas de un gran número de funciones de gran aceptación.
- Control de la memoria: con funciones como *g_malloc()*, *g_malloc0()*, *g_free()*, que nos ayudan en la reserva y liberación de memoria.
- Hilos: tiene a su disposición un repertorio de funciones destinadas al control de threads.
- Estructura de datos complejas: listas simplemente y doblemente enlazadas; tablas Hash.

3.2.3. Pango

Pango es una biblioteca para la composición y renderizado de texto, con énfasis en la internacionalización. Pango se usa allá donde el emplazado de texto sea necesario, sin embargo, la mayoría del trabajo en Pango se hizo usando el conjunto de herramientas para elementos de GTK+ como plataforma de prueba. Pango conforma el núcleo del manejo de texto y tipografías en GTK+-2.0.

Tiene un soporte extensivo para los variados sistemas de escritura en todo el mundo. Muchos sistemas de escritura usados en los idiomas tienen complejas reglas de dibujado de glifos y composición de caracteres. Con Pango, casi todos los idiomas pueden escribirse y mostrarse correctamente, permitiendo a los usuarios de cualquier parte ver texto en sus idiomas maternos. El soporte de Pango para múltiples sistemas de escritura es automático.
Soporta el estilo de texto usado en los típicos documentos e interfaces, incluyendo cursivas, tamaños de tipografía y subrayado. Esta biblioteca usa un vocabulario parecido a XML llamado PangoMarkup que le permite establecer el tamaño de la tipografía, el color, los estilos y otros atributos del texto.

Podemos definirla como una biblioteca para la salida de texto internacionalizado. Se encarga de todo lo relacionado al manejo de fuentes, internacionalización y afines, es la biblioteca que se encarga de la impresión del texto en pantalla.
Mediante Pango, se ha realiazado la parte de editor de texto que viene intrínseca con la aplicación, JDE_DS. Se ha reconocido el idioma del sistema, y establecido el tipo de fuente para dicho editor.

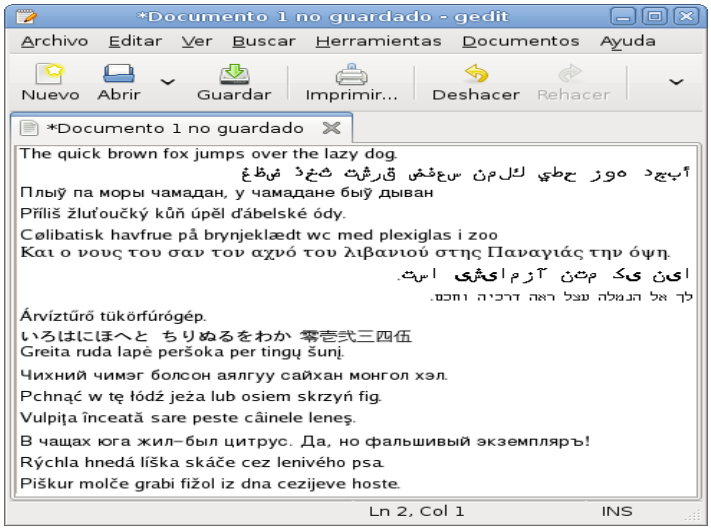


Figura 3.7: Editor con multitud de idiomas con pango

3.2.4. Gdk

Se trata de la biblioteca de bajo nivel que usa GTK+ para interactuar con el sistema para los dispositivos gráficos y de entrada. GDK contiene toda la funcionalidad necesaria para dibujar objetos y texto en la pantalla e interactuar con el usaurio con varios dispositivos de entrada.
Posee características de contexto gráfico y primitivas de dibujado que son adecuadas

para dibujar objetos simples y renderizar imágenes en la pantalla. Nos sirve de gran utilidad para la representación de los nodos y transiciones en la aplicación desarrollada. A través de GDK podemos acceder a eventos del teclado, ratones y otros dispositivos de entrada. También nos proporciona rutinas de bajo nivel para acceder a los datos de arrastrar y soltar y del portapapeles del sistema.

3.2.5. GdkPixbuf

GdkPixbuf es una biblioteca para la carga, manipulación y visualización de imagenes en distintos formatos (png, gif, jpeg, etc). En la versión 1.x, esta librería no forma parte de GTK+, aunque, al ser la sustituta de imlib, se ha procedido a la distribución junto con GTK+.
Esta biblioteca nos ofrece funciones para el dibujo de imagenes en diversos *widgets*, el escalado de las mismas, información de bajo nivel de las imágenes, etc. El tipo de dato con el que trabaja se denomina *GdkPixbuf*, es la estructura con la que representa la imagen, y guarda toda su información.
Con el conjunto de funciones que lleva consigo la librería, se ha desarrollado el zoom de la vista prevía y la impresión del autómata.

3.2.6. Cairo

Es una biblioteca gráfica 2D que cuenta cun una API sofisticada para dibujar vectores gráficos, imágenes compuestas y renderizar texto con antialias.*Cairo* proporciona soporte para muchos dispositivos de salida, incluyendo el sistema X Window, Microsoft Windows y búferesd de imagen, permitiéndole escribir código para dibujar gráficos en diferentes medios independiente de la plataforma.

El modelo de dibujado de *Cairo* es similar al que proporcionan *PostScript* y PDF. La API de *Cairo* proporciona operaciones de dibujado tales como pincelar y rellenar splines cúbicos de Bézier, componer imágenes y realizar transformaciones afines. Estas operaciones con vectores permiten altos gráficos con antialias sin tener que usar el caro dibujo basado en píxeles en el código de la aplicación.
También permite renderizado de alta calidad en múltiples dispositivos, como verbigracia la creación de salidas modificadas adecuadas para una impresión.

Figura 3.8: Dibujo realizado con Cairo

3.3. LibXml

Es una biblioteca, desarrollada por el proyecto *Gnome*, de funciones en C cuyo objetivo es tratar archivos XML. Se caracteriza por su gran portabilidad, pues depende sólo de la biblioteca estándar de ANSI C. Además de la interfaz nativa en C, provee interfaces para otros lenguajes de programación como: C++, C#, Perl, Python, Ruby y otros. Aunque pertenezca al proyecto *Gnome* puede usarse de forma independiente. Provee todas las funciones necesarias para trabajar archivos xml, abrir documentos, recorrer, crear y modificar árboles (xml), validar contra DTDs y manejar diferentes juegos de caracteres como ISOxxxx, UTF8 o UTF16, entre otras.

Capítulo 4
Descripción Informática

En este capítulo comentaremos las soluciones desarrolladas para cumplir los objetivos marcados en el capítulo 2, para ello se utilizaron las herramientas descritas en el capítulo anterior. Plantearemos las diversas partes de la estructura de *Jde_DS* en una vista general y su alto nivel de cohesión que hay entre ellas, posteriormente describiremos en mayor detalle el funcionamiento de cada una de las partes.

4.1. Diseño general

Jde_DS es una aplicación destinada a la creación de esquemas compatibles con *Jderobot* a partir de autómatas gráficos. Esta aplicación se organiza en varias secciones, facilitando su desarrollo y contraatacando los objetivos marcados a lo largo de la creación del proyecto.

Las partes más significativas son la GUI del editor gráfico, la GUI del editor de texto, el volcado del contenido del proyecto en un fichero XML, la generación de código autómatico desembocando en un esquema, y la compilación de dicho esquema para su ejecución con la plataforma *Jderobot*.

El editor gráfico, es una interfaz que nos permite crear un autómata con todos sus componentes: nodos, transiciones, estado inicial. También nos permite modificar las características esenciales del proyecto del autómata como su ubicación en el disco duro, el nombre del proyecto, y la velocidad del esquema en la ejecución con *Jderobot*. Es una interfaz que ofrece ayudas al usuario, como la capacidad de impresión del autómata, el guardado del gráfico del autómata en una imagen, etc..
Esta GUI es la ventana padre de la aplicación, en ella se crea el 100 % de la estructura del autómata, manejando todos los atributos del esquema y del proyecto en cuestión. Aunque prácticamente todo el desarrollo se lleva a cabo en el editor gráfico, la funcionalidad del autómata y el comportamiento del mismo tiene lugar en otra interfaz de gran importancia, es la GUI perteneciente al editor de texto.
A partir de la ventana principal podemos programar cada estado o transición de nuestra

estructura, simplemente "clickeando" en la opción de *Programar estado* o *Programar transición* dependiendo del tipo de objeto, donde se llamará a la interfaz del editor de texto.

El editor de texto nos permite la posibilidad de insertar, eliminar o modificar el código de cada objeto. La programación que demos a cada parte del autómata nos definirá el comportamiento del esquema dentro del *middleware Jderobot*.
La GUI de programación nos proporciona las funcionalidades básicas para una escritura y manipulación de texto cómoda. Nos ofrece botones de "Undo/Redo", de guardado, etc.. Una vez acabado con la definición del comportamiento en cada parte del autómata, podemos guardar dicho código dentro de la estructura del autómata, salvando los cambios realizados en el editor de texto.

Tras describir la fucionalidad de programación que ofrece el editor gráfico, pasaré a describir otra funcionalidad importantísima, se trata del volcado del proyecto en un fichero XML.
Este atributo nos permite guardar en el disco duro las propiedades del proyecto, la estructura del autómata con las propiedades de cada nodo o transición, la velocidad del esquema en tiempo de ejecución, etc..
Gracias a ello podemos transportar nuestro proyecto de un ordenador a otro, cargar proyectos a nuestra aplicación y almacenarlos. Todo mediante un fichero de carácter XML, como ya se conoce, este lenguaje nos aporta un estándar para el intercambio de información de forma estructurada.

Otra funcionalidad sustancial que nos dota la interfaz principal, es la capacidad de generación de código de forma automática. Este cometido nos crea un fichero ".c", con un esqueleto compatible con la plataforma *Jderobot*. En dicho fichero, se verá reflejado la representación del autómata con su programación, todo ello en un código legible para su posterior compilación.
Tras una compilación satisfactoria, nos encontramos con un fichero ejecutable por la aplicación *Jderobot*, con extensión ".so". Dentro de *Jderobot* podremos comprobar el correcto funcionamiento de nuestro autómata creado por la aplicación *Jde_DS*.

A continuación muestro un esquema que nos detalla a la perfección las relaciones establecidas entre las distintas partes de la aplicación, permitiendonos una comprensión general más completa (figura 4.1).

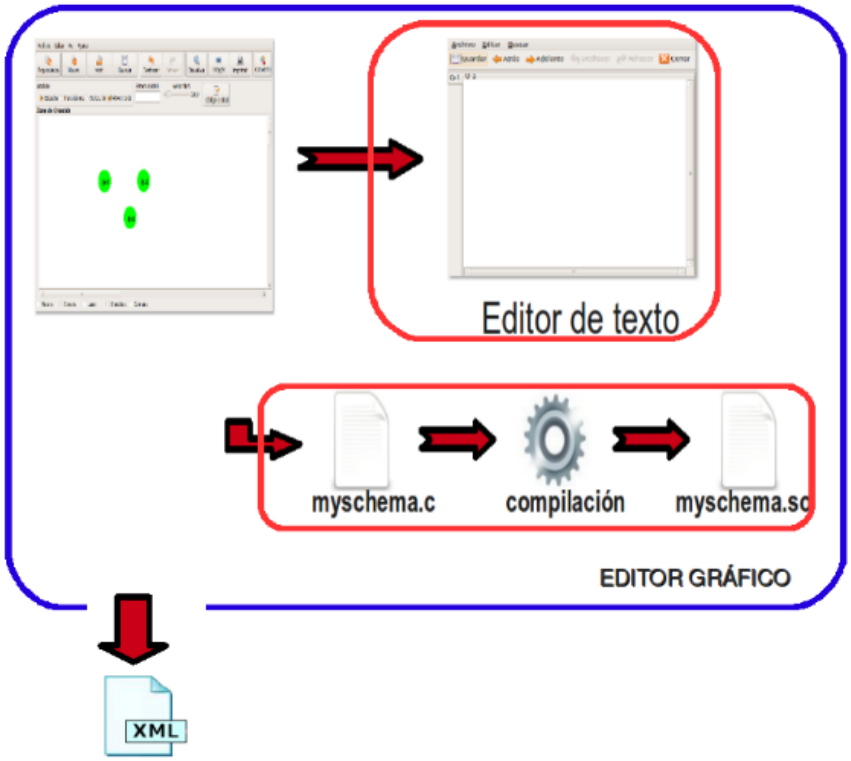


Figura 4.1: Relación de partes de *Jde_DS*.

Toda esta visión es bastante anodina en cuanto al contenido de la aplicación, sin entrar en detalles de los elementos funcionales que la componen. Los elementos funcionales se distribuyen en bloques funcionales permitiendo dar vida a la plataforma desarrollada. Esta división en bloques nos ofrece una organización de los de las estructuras empleadas para abastecer cada objetivo propuesto, este tipo de distribución corresponde con el modelo de "divide y venceras".

Para dar mayor recreatividad a todos estos bloques, se pueden visualizar en la figura 4.2.

Figura 4.2: Bloques funcionales de la plataforma.

4.2. Editor gráfico

El editor gráfico es la parte más importante de la aplicación, es donde el usuario posee todas las funcionalidades para la construcción del autómata y posteriormente convertirlo a esquema.

Este editor se ubica dentro del fichero */gui/JDE_DS.glade*, desarrollado por la plataforma *Glade*. Gracias a esta aplicación se ha diseñado completamente la interfaz añadiendo funcionalidad a cada uno de los *widgets* que la componen.

La interfaz la podemos dividir en varias partes: el menú desplegable, los botones de la aplicación, el selector de acción, las dependencias de los nodos, el indicador del estado inicial, el editor de velocidad del esquema, la zona del autómata y por último la activación de sensores u actuadores.

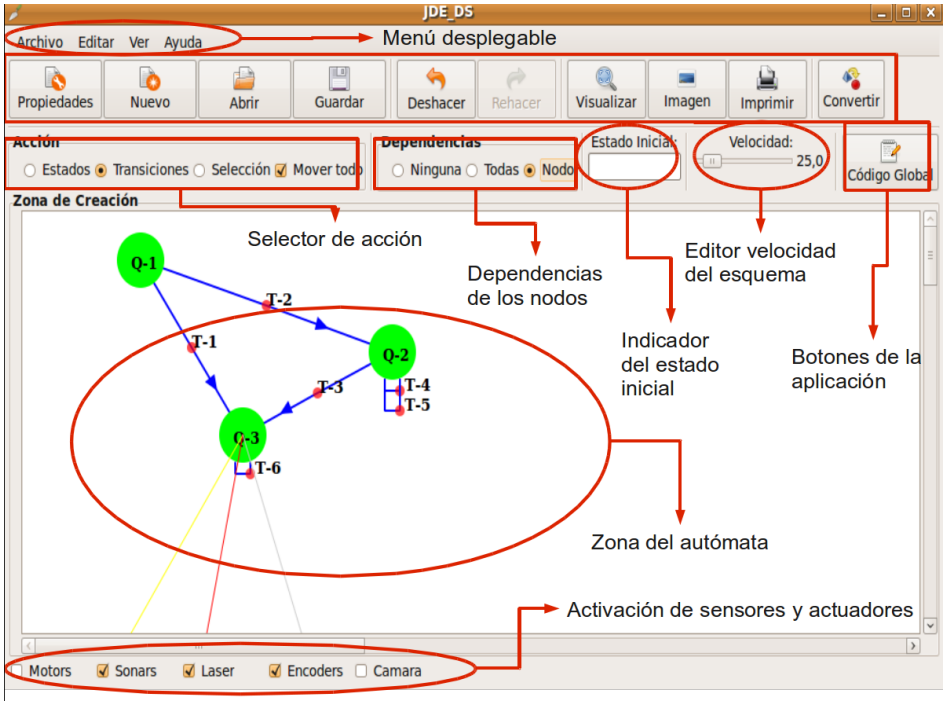


Figura 4.3: Explicación de la interfaz JDE_DS.

En este apartado me dedicaré a detallar los bloques funcionales que se encuentran dentro del editor gráfico de la aplicación, comentando las estructuras fundamentales de cada sección, y los *widget* de Gtk+ implicados en dicha funcionalidad.

4.2.1. Contenido del autómata

Este bloque comprende la funcionalidad de abastecer las estructuras y los métodos necesarios para implementar el almacenado de las partes del autómata. A parte de almacenar los datos propios del autómata, se encarga de enlazar la representación gráfica de cada transición o estado con la referencia del propio objeto.

El almacenamiento de las características de cada estado y transición que el usuario construye en la zona de creación del autómata se almacena en dos tablas hash distintas, una para cada tipo de objeto. La estructura en la que se albergan es de tipo *GHashTable* perteneciente a la biblioteca *GLib*. Esta estructura consiste en almacenar dentro de una tabla todos los objetos de forma ordenada, utilizando para ello una clave. Dentro de dicha tabla se almacena dos valores, la clave con la que permite ubicar el objeto, y el objeto en sí.

Para crear una variable del tipo *GHashTable*, nos obliga a declarar dos funciones de tipo *GHashFunc* y *GEqualFunc*, las cuales se les pasa como parámetro al constructor. La función *GHashFunc* permite identificar en que posición de la tabla ubica el

objeto que queremos añadir. La implementación que realizo para esta función es de calcular el módulo del índice del objeto entre 10, cuyo valor es la posición donse se almacenará dicho objeto. La función *GEqualFunc* tiene como objetivo verificar si dos claves son iguales. Para ello comparo los identificadores de los objetos, devolviendo 1 si son iguales y 0 en caso contrario.

Los objetos que se almacenan en las tablas son nodos o transiciones, de tipo *TypNodo* o *TypTran* respectivamente.

El tipo *TypNodo* tiene la siguiente estructura:

```
typedef struct {
  int sup_izq[2]; //Esquina superior izquierda
  int sup_der[2]; //Esquina superior derecha
  int inf_izq[2]; //Esquina inferior izquierda
  int inf_der[2]; //Esquina inferior derecha
  int centro[2]; //Centro del nodo
  int id; //Identificador único
  GString *nombre; //Nombre del nodo
  int modificado; //Variable de modificado texto
  GSList *Lista; //Lista de transiciones
  TypSensores *sensores; //Valor de los sensores
  GString *codigo; //Codigo del esquema en el nodo
} TypNodo;
```

En ella cabe destacar las variables que se encargan de almacenar la posición del nodo en la zona de creación del autómata, son: *sup_izq*, encargada de representar el vértice superior izquierdo; *sup_der*, almacena el vértice superior derecho; *inf_izq*, el vértice inferior izquierdo; *inf_der*, el vértice inferior derecho; y por último la variable *centro*, encargada de almacenar el centro del estado. Como se puede observar, la representación de un estado se realiza de forma cuadrada, aunque graficamente se visualice de tipo circular. También nos encontramos con la variable *id* encargada de almacenar el identificador único para cada nodo; la variable *modificado* destinada para conocer si se encuentra en estado de edición o no el código del nodo; la variable de tipo *GSList* es una lista cuyo objetivo es almacenar los puntero de las transiciones que se encuentran enlazadas con el estado; *sensores* permite representar las dependencias que tiene dicho estado mediante una matriz de booleanos; Y por último nos encontramos con las variables de tipo *GString*, cuya misión es albergar el nombre y la programación del nodo.

El otro tipo de estructura que se ve almacenado en una tabla hash, es de tipo *TypTran*.

```
typedef struct {
```

```
  int IdentNodoA; //Identificador NodoA
  int IdentNodoB; //Identificador NodoB
  int Ident; //Identificador transición
  int PosA[2]; //Posición NodoA
  int PosB[2]; //Posición NodoB
  int PosTrans[2]; //Posición transición
  int PosTransL[2]; //Posición en caso de bucle
  int PosTransR[2]; //Posición en caso de bucle
  int modificado; //Variable modificado
  int temporal; //Transición temporal
  long msec; //Milisegundos
  GString *nombre; //Nombre transición
  GString *codigo; //Código transición
}TypTran;
```

Las variables *IdentNodoA* y *IdentNodoB*, almacenan el identificador del los nodos que se encuentran en la transición, origen y destino respectivamente. El *Ident* contiene el identificador único para cada transición (es independiente a los nodos). *PosA* y *PosB* albergan la posición de los nodos. Las variables *PosTrans*, *PosTransL* y *PosTransR* llevan la posición del nodo, la primera el la posición donde se ubica la transición y las dos últimas posiciones son auxiliares para representar una transición de tipo bucle. La variable *modificado*, al igual que en los nodos, sirve para verificar si la transición se encuentra abierta en el editor de código. *temporal* y *msec* nos indican si es una transición temporal y en caso de serlo guardar los milisegundos. Y por último nos encontramos con los tipos *GString* de las variables *nombre* y *codigo* que almacenan el texo para el nombre de la transición y para el código programado dentro de la misma.

Otro tipo de estructura perteneciente a los pilares de la aplicación es aquella que nos permite reconocer los estados y transiciones dentro de la zona de creación de autómata. El tipo es el siguiente:

```
typedef struct {
    int num;
    int nodo;
} TypPos;
TypPos Indice [2500][2500];
```

La variable *Indice* es un array del tamaño de la zona de creación del autómata, compuesto en cada celda por un registro de dos enteros. El *num* nos almacenará el identificador del objeto (Objeto/Nodo/Transición). Y la variable *nodo* nos almacenará los siguiente valores dependiendo del tipo de objeto contenido en esa posición. Tomará valor -1 en caso de que esté vacío, 1 si es un nodo, 2 si es una transición, y 3 si es un tipo objeto.

Por tanto *Indice* nos permite la habilidad de saber si el usuario hace "click" en un objeto o no, y en caso haberlo saber con exactitud el objeto en cuestión. Esto es debido a que existe una relación directa entre la posición donde se dibuja el objeto en la zona de creación, y la posición en la variable *Indice*. La creación de esta variable fué fruto de una gran reflexión, ya que ganamos en velocidad perdiendo en memoria. Otra de las opciones que le hizo frente fué comparar todos los objetos construidos para saber si estaban en el rango del puntero, sin embargo la velocidad disminuye considerablemente con el número de objetos que el usuario puede construir. Tras la meditación me decidí por la primera opción debido a que actualmente el tema de memoría no resulta un gran problema.

Estas estructuras son de gran importancia, debido a que en ellas se encuentran informaciones valiosas para el autómata. Además permite interactuar con el usuario, abasteciendo más del 80 por ciento de la interactividad de la plataforma.

Los widget que se encuentran relacionados de forma directa con la estructura que nos permite reconocer los objetos del autómata se encuentran dentro de la zona de creación del autómata.
Esta zona está diseñada por varios *widgets*: un *GtkFrame*, un *GtkScrolledWindow*, un *GtkViewPort*, dos *GtkAdjustment* y el más importante un *GtkDrawingArea* denominada en el fichero XML de la interfaz gráfica como *ZonaDibujo*, ver figura 4.4.

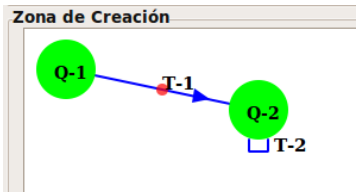
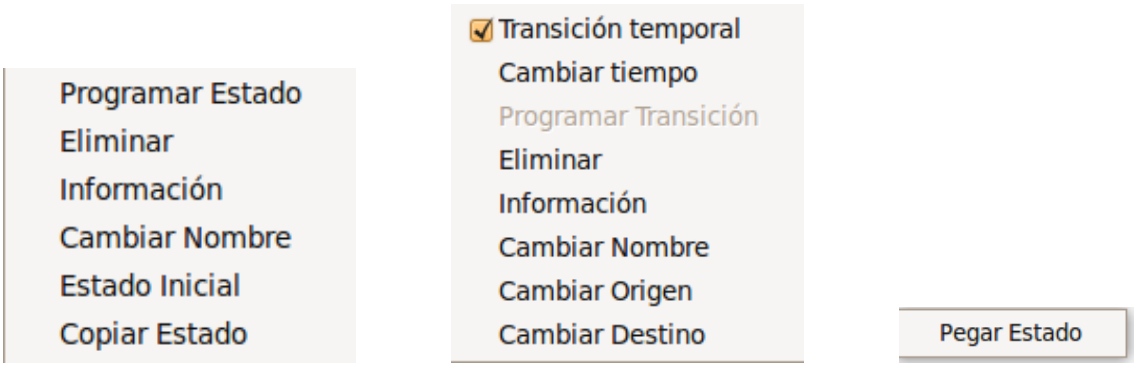


Figura 4.4: Zona de creación del autómata

Es la parte más importante de la aplicación, porque es dónde se centra el mayor nivel de interactividad con el usuario. En está zona es donde se dibujan los estados, transiciones dando la posibilidad de eliminarlos, moverlos, programarlos, cambiarlos de nombre, etc..
Esta zona es sensible a los movimientos del ratón, concretamentes a sus botones y desplazamientos. Al hacer un "click" podemos crear estados, transiciones o seleccionarlos, todo ello dependiendo de la opción de acción activada y de la parte del editor del autómata dónde se realice. Al pulsar el botón izquierdo podemos arrastrar objetos o crear transiciones. Y por último con "click" del botón derecho nos muestra

un menú emergente con opciones para el usuario (figura 4.5), dependiendo donde se realice, si encima de un estado, transición o zona vacía, nos mostrará un tipo de menú u otro.



(a) Menú Estado (b) Menú Transición (c) Menú Vacío

Figura 4.5: Menús emergentes dentro de la zona de creación del autómata.

Sólo se ofrece al usuario tres tipos de acción, los cuales determinan la funcionalidad del ratón dentro de la zona de creación del autómata. La selección de acción es un conjunto de botones de tipo *GtkRadioButton* englobados por un marco cuyo *widget* es un *GtkFrame*. Dentro del marco también encontramos un botón de tipo *GtkCheckButton*, ver figura 4.6.

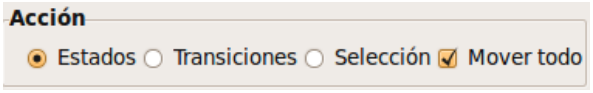


Figura 4.6: Marco de selección de acción.

Todos los *GtkRadioButton* que se encuentran están agrupados en un mismo conjunto, este hecho hace que sólo y solamente uno sea el seleccionado. Estos botones nos ofrencen la posibilidad de crear estados, transiciones o permitir la selección de los objetos que hay en la zona de creación del autómata.
Para la creación de un nodo sólo basta con hacer "click" con el botón izquierdo del ratón en una zona vacía dentro del marco de creación del autómata. En caso de realizarlo en algún estado o transición, quedará seleccionado el objeto "clickeado".
Sin embargo para la creación de una transición, hay que pulsar el botón izquierdo en el nodo origen y soltarlo en el nodo destino. A continuación se mostrará la transición creada. Si se trata de una transición de tipo bucle, mismo origen y destino, basta con hacer "click" en el nodo deseado. Al igual que con la acción de estados podemos

seleccionar objetos, pero en este caso sólo de carácter transición.
La acción de selección solamente nos permite seleccionar los objetos del autómata. Aunque esta acción se encuentra respaldada con las opciones anteriores, aporta al usuario mayor tranquilidad a la hora de seleccionar y mover objetos debido a que en ningún momento podremos añadiremos objetos nuevos sin quererlo.

Dentro de este *frame* también tenemos un botón de tipo *GtkCheckButton*, con la etiqueta de *Mover todo*. Este botón nos da la posibilidad de elegir si queremos la opción de mover las transiciones cuando se muevan los estados. Con la opción activada, cuando el usuario mueva una transición se organizarán de forma automática las transiciones que se encuentren relacionadas con el estado en cuestión.

4.2.2. Dibujado

En este bloque se trabaja capacidad de representar los objetos dentro de la zona del autómata. Se encarga de implementar los métodos necesarios para dibujar los estados y transiciones, las líneas de las dependencias, y además de refrescar cuando sea necesario el *widget* encargado de representar en la interfaz gráfica el autómata.

Para la representación gráfica del autómata se utiliza un pixmap secundario, es decir, el dibujo del autómata se pinta inicialmente en una estructura llamada pixmap del tipo *GdkDrawable*, permitiendo un carácter de doble buffer. Para mostrarlo por pantalla en la aplicación se vuelca dicho *widget* en la zona de creación del autómata. De esta manera ganamos una mejora en cuanto al refresco de la imagen, evitando ciertos parpadeos y mostrando todo el autómata de forma simultánea.
Una variable de carácter importante dentro de la aplicación es de tipo *TypDibujo*, en ella se almacenan los objetos esenciales para el pintado y la representación del autómata en la ventana del editor gráfico. Esta variable tiene la siguiente estructura:

```
typedef struct {
    GdkDrawable * pixmap; //Pixmap para el autómata
    GtkWidget * zonaDibujo; //Almacena GtkDrawingArea
    GtkWidget * scroll; //Almacena el ScrolledWindow
    int longitudX; //Longitud del GtkDrawingArea
    int longitudY; //Altrua del GtkDrawingArea
    GdkGC *penWhite; //Colores Blanco
} TypDibujo;
```

En el *pixmap* sólo se dibuja la estructura del autómata, es decir nodos y transiciones. La *zonaDibujo* almacena el *widget* de la zona de creación del autómata. Las dos variables de tipo "int" almacenan el tamaño del *GtkDrawingArea* para aplicarselo al pixmap. Y

por último el tipo *GdkGC* almacena el color blanco para mostrarlo como fondo de la zona de creación.

Para el dibujo de estados, transiciones y sensores se emplea las herramientas de la biblioteca de *Cairo*, ya mencionada en el capítulo anterior. Gracias a las funciones proporcionadas por dicha biblioteca, obtenemos una representación más nítida y de un mayor nivel de representación.

4.2.3. Deshacer y rehacer

Se trata de un bloque de gran importancia, ya que cumple con la acción "Undo/Redo" típica de cualquier editor. Los botones que nos permiten ejecutar este par de acciones se encuentran tanto en los botones de la aplicación como en el menú desplegable dentro de la opción de *Editar*.

Los botones relacionados con la acción "Undo/Redo" tienen gran importancia en la aplicación. Tienen como función deshacer y rehacer los cambios ocasionados por el usuario. Sin embargo no almacena todos los cambios producidos, ya que tiene un límite que alberga las últimas 10 acciones realizadas.
Para dar fruto a esta propiedad tan importante y amigable para el usuario, se ha conseguido desarrollandola bajo una lista circular. Se basa en almacenar la estructura del proyecto justo antes de realizar el cambio en una secuencia de ficheros númerados del 0 al 9 dentro de la carpeta */tmp/schemas/*. El código de cada objeto se explicará dentro del editor de texto. Fuera de la lista circular también se guarda un fichero con el proyecto actual pero con la información de los nodos y transiciones actuales, y los objetos anteriores que se hayan eliminado.
Tenemos tres tipos de índices que nos permiten implementar la simulación de la lista circular. Estos índices nos aportan el número de fichero temporal en el que se almacena, o se carga la información del proyecto. El índice *Escritura* nos da el número del fichero en el que vamos a realizar la copia del proyecto, todo esto se realiza justo antes de hacer el cambio el usuario; el nombre del fichero en el que se almacena es del tipo *schemaX.xml* donde la X es el índice en cuestión. El índice *Lectura* es el encargado de almacenar el número del proyecto que se desea cargar, este se encuentra en una posición por detrás del de escritura. Y por último caba mencionar el índice *Inicio*, encargado de albergar el número del primer fichero guardado dentro de los diez posibles.
Como he iniciado anteriormente para hacer esto posible, nos apoyamos en otro fichero llamado *codigoActual.xml*. Este fichero alberga la información de los nodos y transiciones recientes. Se utiliza para actualizar el proyecto una vez cargada la

estructura por parte del índice *Escritura*. Sólo actualiza la velocidad del esquema, el código programado de cada objeto, las transiciones temporales, y los sensores establecidos por los estados.
Gracias a estas estructuras, se hace posible la acción "Undo/Redo" relacionada con la estructura del autómata.

4.2.4. Bloque de impresión

Dentro de esta agrupación, se trata de implementar la capacidad de imprimir el esquema del autómata tanto a una impresora física como a un fichero "PDF". En este bloque carece de una gran estructura, debido a que no tiene que almacenar ningún tipo de dato, simplemente crear una ventana de impresión dando la posibilidad al usuario de elegir el tipo de impresora, y de cambiar las propiedades básicas necesarias.

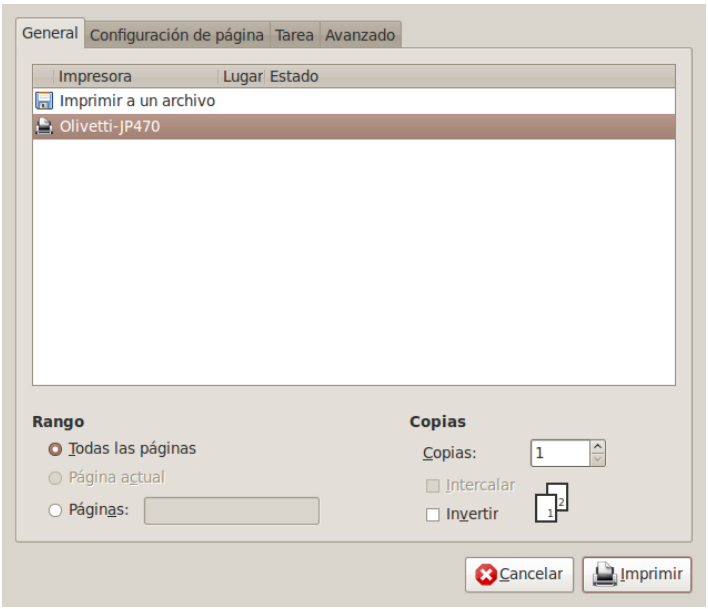


Figura 4.7: Ventana de impresión del esquema del autómata

4.2.5. Visualizacion del autómata

Este bloque abastece todos los métodos necesarios para representar el esquema del autómata, sin relación de dependencias, en una ventana nueva ofreciendo la funcionalidad de zoom. Tiene como objetivo construir una visión general de nuestro autómata en caso de tratarse de un esquema con gran cantidad de nodos y transiciones.

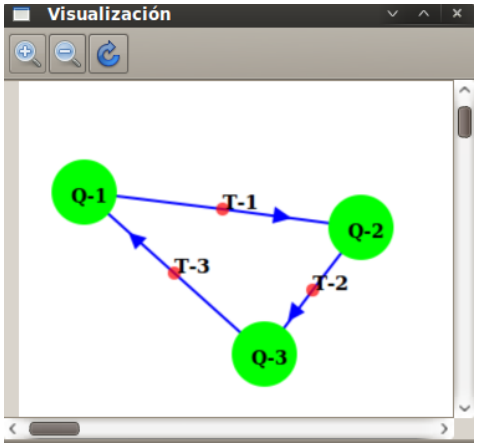


Figura 4.8: Ventana de visualización del autómata.

4.2.6. Dependencias del autómata

En esta agrupación de funciones y estructuras, se encargan de representar internamente las dependencias de cada estado, calcular las dependencias totales de todo el autómata, interactuar y representar dentro de la zona de activación de sensores y actuadores.

La parte de activación de sensores u actuadores (figura 4.9), está organizada por 5 *GtkCheckButton*, correspondientes por cada sensor u actuador que se puede insertar en el esquema. Nos encontramos con los *Motors*, *Sonars*, el *Laser*, los *Encoders* y la *Camara*.

Figura 4.9: Activación de sensores

Está relacionada irrefutablemente con la opción marcada en la parte de dependencias. Dependiendo de la opción seleccionada, se podrá habilitar o deshabilitar la información mostrada de los sensores y actuadores.
La parte de las dependencias de los nodos esta formado con *widgets* de tipo *GtkRadioButton*. Tenemos tres botones encargados de representar en la zona de dibujo todas las dependencias, ninguna o de un nodo seleccionado. Esta opción ayuda al usuario a detectar las dependencias de cada estado y también del autómata en general. La opción *Ninguna*, se deshabilida la zona de sensores y actuadores ya que no se establece ninguna relación con los estados. La opción *Todas* mantiene desactivada la zona de sensores y actuadores debido a que muestra por pantalla las dependencias de todos los nodos del autómata sin posibilidad de ser modificadas, es decir, muestra las

dependencias del autómata de forma global. Y por último la opción de *Nodo* permite modificar las dependencias de cada nodo, por ello se encuentra activada dicha zona. Esta última opción es la única que permite modificar las dependencias de los estados.

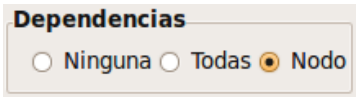


Figura 4.10: Marco de muestra de dependencias.

4.2.7. Configuración del proyecto

Dentro de este bloque determinamos la funcionalidad que nos permite configurar las características de un proyecto. Aunque son bastante reducidos los atributos a configurar, son bastante importantes. Determinan la ubicación del proyecto actual, y la dirección dónde se creará el esquema final con el código programado del autómata. Este tipo de configuración se relaciona con el botón *Propiedades*, permitiendo cambiar los atributos del proyecto actual (figura 5.1). Dichas propiedades son: el nombre del proyecto y su ubicación dentro del disco duro.

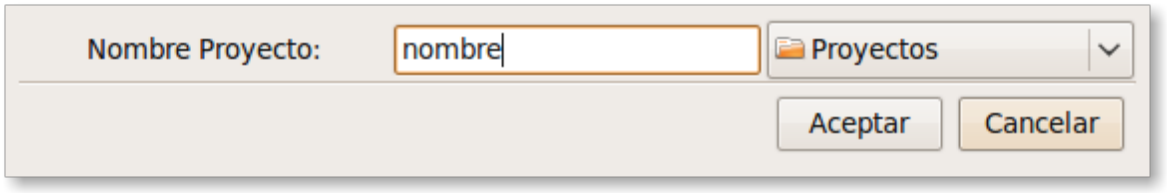


Figura 4.11: Ventana de configuración del proyecto.

A parte de este tipo de configuración, también se debe establecer los parámetros que influyen de manera notoria en el esquema resultante. Se trata del estado inicial donde se empieza a ejecutar el código implementado en el autómata, y la velocidad de ejecución del esquema por la plataforma *Jderobot*.

El indicador del estado inicial está formado por dos *widgets*. Uno de ellos es un *GtkLabel*, el cuál muestra el texo de "*Estado Inicial:*"; el otro *widget* es un *GtkEntry* con la propiedad de no modificable, ya que sólo nos muestra el nombre del estado inicial. Ver figura 5.7(a).

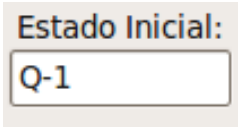


Figura 4.12: *Widgets* de la selección del estado inicial.

El editor de velocidad está formado por un *GtkLabel* y un *GtkHScale*. Este último *widget* es el encargado de modificar la velocidad del esquema, por lo que debe ser sensible tras su cambio de valor. El *GtkHScale* se encuentra ajustado mediante un objeto de tipo *GtkAdjustment*, este se encarga de establecer los rangos de valores, el valor inicial y el incremento de valores por cada movimiento. Ver en la figura 4.13.

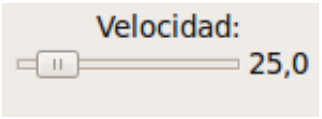


Figura 4.13: Configuración de la velocidad del esquema.

4.3. Editor de texto

Una vez detallada la interfaz principal de la plataforma, la cual da pie a todas las demás ventanas de la aplicación, procederé a hablar sobre el editor de texto.

Este editor tiene almacenada su interfaz gráfica dentro de un fichero "XML" denominado: *Editor.glade*. En dicha ventana podemos apreciar las partes de menú desplegable, los botones del editor, y la zona de edición de texto (figura 4.14).

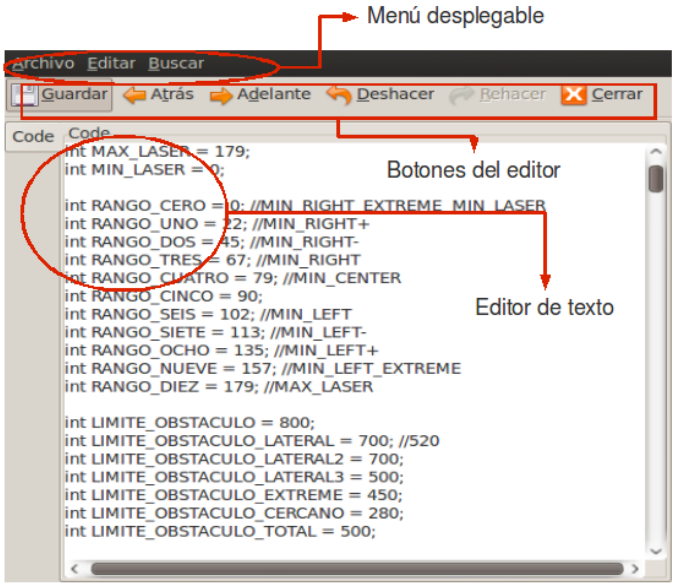


Figura 4.14: Interfaz del editor de texto.

En esta sección hablaré sobre los bloques de funcionalidad principales de la aplicación, relacionados o implementados dentro el editor de texto.

4.3.1. Programación de código

Este bloque general de la aplicación se encarga de implementar las funciones requeridas para programar de forma cómoda y sencilla cada objeto de nuestro autómata. Ofreciendo camodidades al usuario, para facilitar su uso.

Para llevar a cabo toda esta acción dentro del editor de texto, necesitamos una serie de estructuras básicas para su implementación. La estructura principal que da pie al editor de texto, y en el que se basa la interfaz es una lista doble, del tipo *GList* dentro de la biblioteca *Glib*, comentada en el capítulo de infraestructura (3.2.2).
El objetivo de implementar una lista doblemente enlazada, es para desarrollar con mayor eficiencia el recorrido de páginas dentro del *Notebook* del editor de texto.

Dentro de dicha lista, almacenamos un dato de tipo *TypTxt*, en el que se almacenan las características esenciales para poder identificar al objeto abierto en el editor, y para controlar su estado dentro del mismo.
Este tipo de dato es el siguiente:

```
typedef struct {
    GString *Label;
    GtkWidget *texto;
    int Id;
```

```
    int TipoObjeto;
    int guardado;
    int linea;
    int columna;
} TypTxt;
```

La variable *Label* almacena el nombre del objeto abierto, sirve para representar el título de la página dentro del editor de texto. *Texto* es una variable de tipo *GtkWidget* encargada de almacenar el *GtkTextView* de cada página. Este tipo de dato es el encargado de visualizar el texto con la ayuda de la biblioteca *Pango* (3.2.3). Las variables *Id* y *TipoObjeto* nos permiten identificar el tipo de objeto que se encuentra abierto, necesario para almacenar el código programado en el *GtkTextView* y para conocer si se encuentra abierto el objeto en el editor de texto. La variable *guardado* nos permite llevar el control si se encuentra actualizado el contenido del código del objeto en relación con su pestaña abierta. Y por último las variables *linea* y *columna* sirven para almacenar la ubicación del cursor dentro de la pestaña.

Esta estructura es vital para el desarrollo completo del editor de texto. Se encuentra intrínsecamente relacionada con la zona de edición de código. Esta es la parte más importante de esta interfaz gráfica, ya que recae toda la atención del usuario (figura 4.15).

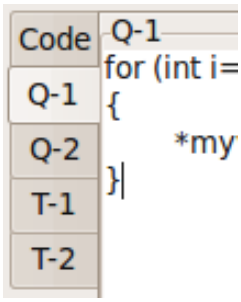


Figura 4.15: Zona de edición de código.

Esta zona de la interfaz gráfica está compuesta por un tipo *Notebook*, encargado del soporte de las páginas del editor. Este *widget* crea y elimina las páginas según la intención del usuario. Cada una de ellas se representa mediante solapas en el *Notebook* con el nombre del objeto vinculado.
Cada página está constituida por un widget de tipo *GtkTextView* que está relacionado con otro widget de tipo *GtkTextBuffer*. El primer *widget* tiene como objetivo representar el texto ubicado en el segundo. Este último es un buffer encargado de almacenar el texto que el usuario escribe en la página. Estos dos tipos de objetos se encuentran repetidos

por cada página, ya que hacer posible su programación.
La representación del texto se ve ayudada por las funciones pertenecientes a la biblioteca *Pango* (3.2.3), encargada de dar formato al texto con las propiedades: tipo de texto, tamaño, y color.

4.3.2. Bloques propios del editor

A parte de los bloques generales de la aplicación, este ámbito contiene subbloques con funcionalidad exclusivo para el editor de texto. La variedad de funcionalidad que tiene este editor se encuentran relacionada con los botones de la ventana gráfica que representa al editor de texto.
El número de botones es de seis y los más importantes son los siguientes (figura 4.16):

Figura 4.16: Botones del editor de código.

- El botón de guardado nos permite almacenar el código de la pestaña actual a la estructura del objeto de dicha pestaña. Esto se realiza cogiendo la información de la pestaña actual que se encuentra en el doble buffer que nos ofrece el *widget* donde se muestra el texto.

- Los botones *Atrás* y *Adelante* nos permiten recorrer de arriba a bajo o viceversa todas las páginas abiertas en el editor. Como ya se comento anteriormente, esto fue la razón por el creado de la lista doblemente enlazada para almacenar los datos esenciales de cada página.

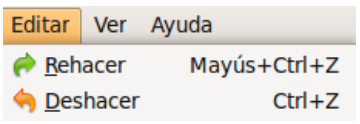
- *Deshacer* y *Rehacer* son los botones encargados de hacer factible la acción de "Undo/Redo" del editor. Este par de botones son globales para la interfaz, es decir, las acciones modificadas de cada página son almacenadas en una lista circular comun para todas las páginas, por lo que el hecho de deshacer y rehacer cambios no tienen porque realizarse en la página actual. Se cuenta una nueva acción por el usuario cada vez que se añade una línea de texto o se deshace, además se le añaden las operaciones de cortar y pegar.
  La estructura y el funcionamiento del "Undo/Redo" en el editor de texto es un anhelo de la interfaz gráfica principal de JDE_DS. Se basa también en una lista circular simulada por tres índices de tipo *Escritura*, *Lectura* e *Inicio*. Como podemos observar tienen los mismos nombres que la interfaz comentada

con anterioridad, por lo que también realizan las mismas funciones. En este caso a diferencia de la otra el nombre del fichero temporal tiene el tipo: *schemaCodigoX.xml* siendo la X el valor del índice oportuno para cada acción. Estos ficheros se encargan de almacenar sobre todo el código de cada objeto programado.

- Por último nos encontramos el botón *Cerrar*, encargado de eliminar del *Notebook* del editor la página actual. Antes de ello, comprueba si se encuentra guardado el código de dicha página en su objeto correspondiente, y en caso de no ser así se limita a mostrar un mensaje de diálogo en el que el usuario tendra toda la responsabilidad de avanzar con el cierre de la página o de cancelar dicha acción.

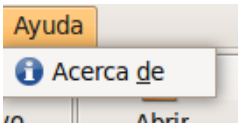
No obstante, podemos añadir también las funcionalidades que aportan los menús deplegables del editor. Las opciones que nos encontramos en cada menú se pueden ver en la figura 4.17 y todos ellos están creados mediante objetos de tipo *GtkMenuImageItem*.

(a) Menú Archivo



(b) Menú Editar

(c) Menú Ver



(d) Menú Ayuda

Figura 4.17: Menús desplegables de JDE_DS.

Dentro de todas las opciones que nos encontramos en los menús desplegables destacan las siguientes:

- La opción de "Cortar" como su nombre indica nos permite cortar una cadena de texto seleccionada. En esta acción se almacena en el portapapeles el string cortado.

- "Pegar" nos permite copiar en la posición del cursor la cadena de texto del portapapeles.
- La opción "Copiar" realiza una copia del texto seleccionado ubicándola en el portapapeles del editor.
- Por último cabe destacar al opción de "Buscar" la cual nos permite buscar una cadena de texto en la página actual. En caso de encontrarla seleccionará todas las similitudes (figura 4.18).

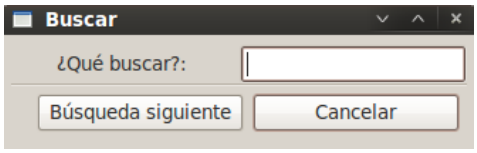


Figura 4.18: Ventana del buscador de caracteres.

Este tipo de funcionalidad se establece para crear un entorno más amigable con el usuario en el ámbito de edición de texto, siendo de forma independiente a la estructura general de *Jderobot*.

4.4. Fichero XML

En esta sección hablaré de la importancia que tiene la manipulación de ficheros "XML" dentro de la aplicación y del tipo de estructura empleado.

La importancia del lenguaje XML

Los ficheros "XML" tienen una gran importancia dentro de la plataforma, nos permiten almacenar de forma completa cualquier proyecto vigente en la aplicación. Al igual que la aplicación permite el volcado de un proyecto, también es factible la carga de proyectos mediante ficheros de carácter "XML". Estos procesos son útiles para el guardado en sí del proyecto, y para hacer posible la funcionalidad "Undo/Redo" de las dos interfaces gráficas importantes de la aplicación.
Se decidió utilizar este tipo de lenguaje debido a su característica de almacenar información auto-descrita. La información almacenada dentro de un fichero de estas propiedades se encuentra bien estructurada, con la capacidad de recoger o añadir información pertienente mediante el uso de bibliotecas. La biblioteca utilizada para la lectura y escritura de estos ficheros es *Libxml* ya comentadas en capítulos anteriores (3.3).

4.4.1. Carga y guardado en XML

Este bloque se centra en los objetivos relacionados con la carga y el almacenamiento del proyecto en ficheros "XML". Para hacerlo posible implementamos un tipo de estructura para la colocación de los datos del proyecto dentro del fichero. Esta organización es la siguiente:

```
<PruebaXml>
    <velocidad vel="45" />
    <inicial estado="1" />
    <nodos>
        <nodo id="1">
            <posicion x="91" y="79" />
            <nombre>EstadoUno</nombre>
            <sensores>
                <motors valor="1" />
                <sonars valor="0" />
                <lasers valor="1" />
                <encoders valor="0" />
                <camara valor="1" />
            </sensores>
            <codigo>Código del estado uno</codigo>
        </nodo>
        <nodo id="2">
            <posicion x="333" y="85" />
            <nombre>EstadoDos</nombre>
            <sensores>
                <motors valor="0" />
                <sonars valor="0" />
                <lasers valor="0" />
                <encoders valor="0" />
                <camara valor="0" />
            </sensores>
            <codigo>Código del estado dos</codigo>
        </nodo>
    </nodos>
    <transiciones>
        <transicion id="1">
            <posicion x="214" y="42" />
            <nodoA id="1" x="91" y="79" />
            <nodoB id="2" x="333" y="85" />
            <temporal opcion="1" tiempo="2000" />
            <nombre>TransiciónUno</nombre>
            <codigo />
        </transicion>
        <transicion id="2">
            <posicion x="346" y="120" />
            <nodoA id="2" x="333" y="85" />
```

```
                <nodoB id="2" x="333" y="85" />
                <posicionL x="325" y="120" />
                <posicionR x="341" y="120" />
                <temporal opcion="0" tiempo="0" />
                <nombre>TransiciónBucle</nombre>
                <codigo>Código de la transición</codigo>
            </transicion>
    </transiciones>
    <global>
        <codigo>Código global</codigo>
    </global>
</PruebaXml>
```

Dentro de la estructura podemos destacar cuatro partes a comentar. El fichero "XML" tiene como raiz el nombre del proyecto. Dentro de dicha raiz nos encontramos tres zonas bien diferenciadas. En primer lugar tenemos las propiedades relacionadas con el autómata y el esquema: la velocidad y el nodo inicial. Posteriormente consta de una etiqueta que nos indica el almacenamiento de todos los nodos del autómata, otra etiqueta que nos indica el empiece de las transiciones, y finalmente nos encontramos con el código global del autómata.

Como ya he comentado todos los nodos se almacenan dentro de la etiqueta *nodos*. En la etiqueta principal de cada nodo contiene el identificador de cada uno, y tiene como etiquetas hijas *la posición*, el *nombre*, los *sensores* y el *código*. La posición se almacena con dos coordenadas mediante los atributos x e y. La etiqueta de los sensores tiene descendientes en el que muestran las dependencias del nodo en particular. Y por último tenemos el *codigo*, encargado de albergar el código programado de cada nodo.

Dentro del árbol "XML" nos encontramos a parte de los nodos, las transiciones. Cada una de ellas almacena como atributo el identificador y el resto de sus propiedades como nodos hijos. Las etiquetas hijas son: la *posición*, encargada de almacenar la posición central de la transición; *nodoA* y *nodoB* almacenan el identificador y la posición de los nodos origen y destino respectivamente; en caso de ser una transición bucle nos encontramos con las etiquetas *posicionL* y *posicionR* en el que se almacena los valores auxiliares para posicionar la transición en la zona de dibujo; la etiqueta *temporal* almacena si es o no mediante el atributo *opcion* y en caso de serlo almacena en el atributo *tiempo* los microsegundos de la transición temporal; posteriormente tenemos *nombre* encargada de guardar el nombre de dicha transición; y por último al igual que en los nodos tenemos la etiqueta *codigo*, cuya función es almacenar el código programado por el usuario en la transición.

Como nota todas las funciones encargadas de la recogida de datos o la grabación de los mismos en los ficheros "XML" se encuentran en *xml.c* y *xml.h*. Todas las funciones dependiendo del tipo de dato a recoger o escribir, deben posicionarse en primer lugar en la raíz del fichero, para posteriormente buscar el nodo concreto donde halla que almacenar o cargar la información.

Toda esta funcionalidad se encuentra relacionada con los botones de *Guardar* y *Abrir* dentro del editor gráfico de la aplicación. El botón *Abrir* nos carga en la aplicación un proyecto almacenado en el disco duro. Esta actuación se encarga de borrar el proyecto actual y de cargar por completo el fichero en formato "XML" del proyecto seleccionado. *Guardar* es un botón bastante importante dentro de la aplicación, ya que se encarga de almacenar en la ubicación del proyecto un fichero de formato "XML" con toda la información relacionada con el proyecto actual de la plataforma.

4.5. Esquema Jderobot

La última sección que podemos apreciar dentro de la aplicación, es la parte de la construcción del esquema compatible con *Jderobot 4.3*. Comentaré el proceso de construcción de dicho esquema y la estructura que lleva para dar vida a dicho autómata.

4.5.1. Construcción del esquema

En primer lugar para ubicarnos, los ficheros de código donde se encuentran las funciones que hacen posible dicha conversión tienen lugar en *Construir.c* y *Construir.h*. Tenemos una función principal denominada *construirSchema* que es la encargada de realizar todo el proceso, cuyos pasos son preparar el fichero "XML", insertar los estados, las transiciones, crear la función de ejecución del esquema, y por último compilarlo para obtener el ejecutable.

Antes de comentar los pasos citados con mayor detenimiento, deseo mencionar que para la conversión del esquema debe estar configurado el proyecto, ya que a la hora de hacerlo se almacenan en la dirección del proyecto los datos necesarios para la ejecución y la creación del esquema, como por ejemplo el fichero del esquema incavado (sin construir), la interfaz del esquema, el *Makefile* necesario para la compilación, y el fichero *myschema.conf* obligatorio para la ejecución de *Jderobot*; sólo un fichero es añadido en el proceso de construcción, se trata de la imagen que servirá para el GUI del esquema.

1. En la parte de la preparación del esquema, se encarga de almacenar dentro del fichero *myschema.c* la declaración de las variables relacionadas con los sensores y actuadores del autómata, la declaración e inicialización de la variable *Estado* y *myschema_cycle* encargadas de almacenar el estado que se va a ejecutar y de la velocidad del esquema en la ejecución respectivamente. También esta parte se encarga de almacenar el código global del esquema, de la inicialización de las dependencias, y la inclusión de las mismas en las funciones de *stop* y *thread* del esquema.

2. El segundo paso que realiza la función de construcción es la insercción de los estados en el esquema con su código programado, la forma de como se encuentran implementadas todas las funciones se describirá en el apartado de estructura del esquema.

3. De forma anhela al paso anterior, este se encarga de insertar las funciones que aportarán las condiciones programadas en cada transición.

4. La creación de la función de ejecución del esquema es la última parte que se escribe dentro del fichero *myschema.c*. Es la encargada de ejecutar el comportamiento programado.

5. El último paso para la construcción del esquema, es su compilado, el cuál nos aportará el ejecutable para iniciar nuestro comportamiento, todo ello si es factible.

4.5.2. Estructura del esquema

La estructura del esquema debe ser compatible con el *middleware Jderobot*. por lo tanto requiere unos métodos de programación específicos para su buen funcionamiento. Debido a la gran cantidad de código que se encuentra en un ejemplo de un comportamiento, procedo a explicar las estructuras más importantes paso a paso encontradas en las diferentes partes de un esquema.

Variables del esquema

La definición de las variables de las dependencias vienen predeterminadas por *Jderobot*[1], ya que deben ser compatibles con dicho *middleware*. Estas aparecen por demanda, es decir, dependiendo si el autómata lo utiliza o no. Un ejemplo de este tipo de variables es el siguiente:

[1] http://www.jde.gsyc.es

```
//Sonar
float *mysonars=NULL;
runFn sonarsrun;
stopFn sonarsstop;

//Camara
char **mycolorA=NULL;
runFn colorArun;
stopFn colorAstop;
char **mycolorB=NULL;
runFn colorBrun;
stopFn colorBstop;
```

Otro tipo de función tipada por el *middleware* es la ocupada para el tiempo de ejecución del esquema dentro dicha plataforma. Esta es un tipo entero cuyo nombre obligado es *myschema_cycle*. Una variable *int Estado* nos permite almacenar el identificador del estado que debe ejecutar, por defecto toma el valor del estado inicial. A continuación tenemos una estructura encargada de almacenar el tiempo de ejución del esquema *TypeTiempo*, este tipo se utiliza para conocer el tiempo de ejecución dentro de un nodo del autómata y así poder satisfacer la transición temporal. Las estructuras habladas son las siguientes:

```
int Estado = 1; /* aplicacion */
int myschema_cycle = 27; /* ms */

typedef struct{
        struct timeval inicio;
        int estado;
}TypeTiempo;

  TypeTiempo *tiempo;
```

Funciones y variables del código global

Una vez declarado todas las variables del esquema, nos encontramos con el código global definido en la aplicación *JDE_DS*. Este código será referenciado por las funciones de las transiciones y de los estados, siempre dependiendo de la programación realizada por el usuario.

Funciones auxiliares del esquema

En esta sección tenemos las funciones que nos permiten representar y hacer funcionar la interfaz gráfica del esquema en tiempo de ejecución. Estas utilizan la biblioteca *Cairo* para la respresentación (figura **??**).

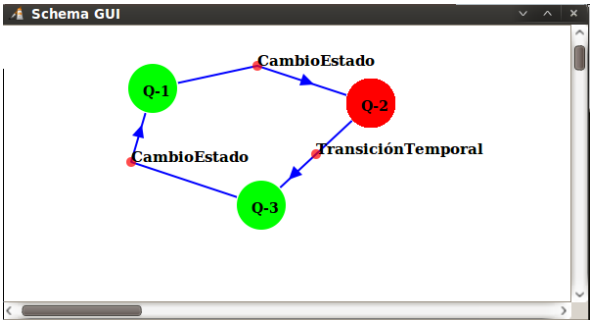


Figura 4.19: Interfaz gráfica del esquema

La interfaz gráfica del esquema nos ayuda a depurar errores que estamos realizando en la estructura del autómata, también nos permite realizar un seguimiento sobre el flujo que está ejecutando el esquema. Esto es debido a que la GUI nos muestra la estructura del comportamiento, señalando los estados en ejecución en rojo.

También nos encontramos con las funciones de acceso a la estructura *TypeTiempo* y de calcular el tiempo transcurrido en cada estado. La forma realizada para calcular el tiempo dentro de cada estado, es almacenar la primera vez que entra en un nodo el tiempo de ejecución y el estado, esta última variable nos permite saber si es la primera vez que entra o no, ya que se le pasa como parámetro el estado actual que se va a ejecutar.

Funciones propias de Jderobot

Estas funciones son las encargadas de lanzar el esquema dentro de la plataforma *Jderobot*, también se ocupan de parar la ejecución, de mostrar la interfaz gráfica y de controlar el tiempo de ejecución del esquema. Este conjunto de funciones se encuentran definidas dentro del manual de *Jderobot* haciendo posible un esquema compatible y ejecutable con dicho *middleware*.

Funciones de estados y transiciones

Las funciones de estados y transiciones dentro del esquema son las encargadas de dar vida a nuestro autómata. En ellas se encuentran el código programado por el usuario en cada nodo o transición. La disposición que tiene este código en las funciones es la siguiente:
En cuanto a los estados se refiere no tiene una estructura especial, simplemente se encarga de almacenar de forma literal el código del nodo. La función se define *void EstadoX_Function()*, donde la X pertenece al identificador del nodo.

Las funciones de las transciones tienen otro tipo de estructura, no obstante su nombrado es muy parecido *int TransicionX_Function(int estadoActual)* donde X corresponde al identificador de la transición perteneciente. El parámetro de salida es un entero, el cual nos marca el estado que se debe ejecutar tras evaluar la condición de la transición, en caso de ser cierta devuelve el identificador del nodo destino, y en caso contrario nos devuelve el estado actual, que corresponde con el estado origen. El parámetro *estadoActual* corresponde a la variable que marca el identificador del estado que se va a ejecutar. En caso de que sea distinto del origen de la transición se devolverá literalmente como salida de la función; el objetivo de esta acción es poder mantener la evaluación que ha tenido otra transición evaluada anteriormente a esta. Se verá con mejor detalle al explicar la función de ejecución del esquema.
Para ubicar e informar al usuario que función corresponde con cada objeto del autómata antes de cada función se escribe un comentario con el nombre del estado correspondiente.

A continuación muestro un ejemplo para poder ilustrar las estructuras empleadas para la representación de dichas funciones:

```
//Q_1
void
Estado1_Function()
{
    imprimirEstadoActual("Uno");
}

//CambioEstado
int
Transicion1_Function(int estadoActual)
{
    if(estadoActual != 1) return estadoActual;
    if(TRUE){
        return (2);
    }
    else{
        return(1);
    }
}
```

Función de ejecución del autómata

La ejecución del autómata se basa en la función *myschema_iteration()*, esta es la encargada de ejecutarse cada vez que *Jderobot* le asigna CPU al esquema. En ella se

debe centrar toda la programación del autómata, por ello es donde se realizan todas las llamadas de las transiciones y estados.

Se tomó la decisión de estructurarlo mediante un *switch* de la variable que marca el identificador del estado a ejecutar, como ya comentamos con anterioridad se trata de la variables *Estado*. Dentro de cada opción nos encontramos en primer lugar el pintado en la GUI del esquema en caso de que se encuentre abierta, seguida de la función que permite controlar el tiempo de ejecución en cada estado, posteriormente se da la función de ejecución del nodo. Una vez acabada la iteración del estado se valuan las transiciones que tienen como origen dicho estado. El valor devuelto de la variable *Estado* tras comprobar todas las transiciones, nos da el estado siguiente que se va a ejecutar. Antes de acabar la opción del *switch* nos encontramos con una función encargada de borrar en el GUI el estado ejecutado, siempre que la variable *Estado* nos de un nodo diferente al actual.

Para poder recrearnos de la estructura realizada, muestro un ejemplo:

```
//ITERATION
void myschema_iteration()
{
    switch(Estado)
    {
        case 1:
            //Función de pintado en la GUI
            dibujar_resalto_estado( 154, 64, "Q-1");
            //Función que controla el tiempo de ejución
            set_tiempo_estado_TypeTiempo (tiempo, 1);
            //Ejecución del código del estado
            Estado1_Function(); //Q_1
            //Verificación de las condiciones de las transiciones
            Estado = Transicion1_Function(Estado); //CambioEstado
            //Borrar el pintado en la GUI
            borrar_resalto_estado( 154, 64, "Q-1", 1);
            break;

        case 2:
            dibujar_resalto_estado( 377, 79, "Q-2");
            set_tiempo_estado_TypeTiempo (tiempo, 2);
            Estado2_Function(); //Q_2
            //TransiciónTemporal
            Estado = Transicion2_Function(Estado);
            borrar_resalto_estado( 377, 79, "Q-2", 2);
            break;
```

```
            case 3:
                dibujar_resalto_estado( 265, 183, "Q-3");
                set_tiempo_estado_TypeTiempo (tiempo, 3);
                Estado3_Function(); //Q_3
                Estado = Transicion3_Function(Estado); //CambioEstado
                borrar_resalto_estado( 265, 183, "Q-3", 3);
                break;

        }
}
```

Capítulo 5
Aplicaciones

En este capítulo mostraremos una serie de pruebas con el fin de demostrar la robustez y el buen funcionamiento de nuestra aplicación, haciendo frente a los objetivos propuestos. Para ello, se han realizado tres ejemplos de comportamientos distintos, con niveles de dificultad diversos. A parte, estos ejemplos han servido para pulir cierta funcionalidad de la plataforma como por ejemplo mejorar la interactividad con el usuario, aumentar las facilidades en la creación del autómata, etc.

Antes de comentar las aplicaciones realizadas con la aplicación, hablaré de las instrucciones necesarias para la construcción de un autómata mediante la aplicación, y la configuración del proyecto para la elaboración final del esquema compatible con *Jderobot*.

5.1. Manual de uso

5.1.1. Manipulación del proyecto

En este sección se hablará todas las posibilidades que ofrece la aplicación para la configuración, carga y almacenamiento de los proyectos.

Configuración del proyecto

La configuración del proyecto se realiza en: *Archivo–>Propiedades*. Nos aparecerá una ventana en la que debemos editar el nombre del proyecto y la ubicación donde deseamos almacenarlo (figura 5.1).

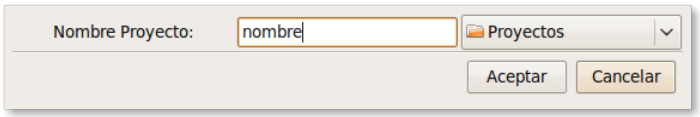


Figura 5.1: Ventana de configuración del proyecto.

En esa ruta se almacenará el proyecto en formato “XML”, los archivos relacionados

con el esquema: “myschema.c”, “myschema.h” y el ejecutable “myschema.so”. También nos encontraremos el fichero de configuración para arrancar *Jderobot* llamado “myschema.conf”, este fichero debe editarse según las necesidades del usuario.

NOTA: Cabe destacar que la ejecución del esquema debe estar acompañado siempre de la imagen “myschemaGUI.png”, y del fichero “SchemaGUI.xml”. Sin estos dos ficheros sería imposible la ejecución de la interfaz gráfica de nuestro esquema dentro de *Jderobot*.

Nuevo proyecto

La creación de un nuevo proyecto se realiza en: *Archivo–>Nuevo*. A continuación se mostrará una ventana de advertencia (figura 5.2), que nos indica si queremos crear el nuevo proyecto advirtiendo que se borrará el actual de la aplicación.

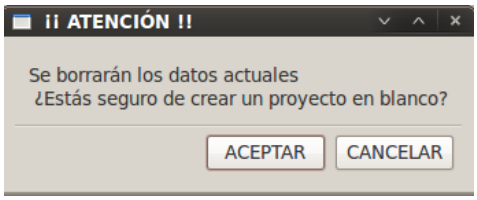


Figura 5.2: Advertencia al crear un nuevo proyecto.

Una vez aceptado nos aparecerá la ventana de configuración del proyecto por si deseamos hacerlo.

Guardar proyecto

La acción de guardar el proyecto nos permite almacenar en el disco duro todo el proyecto actual de la aplicación, en formato “XML”. La forma de realizarlo es mediante: *Archivo–>Guardar*.

NOTA: Para realizar esta acción se necesita haber configurado antes el proyecto.

Cargar proyecto

La forma de cargar un proyecto xml es mediante: *Archivo–>Abrir*. A continuación se muestra una ventana para la selección del fichero “XMl”, donde se encuentra el proyecto almacenado que queremos cargar (figura 5.3).

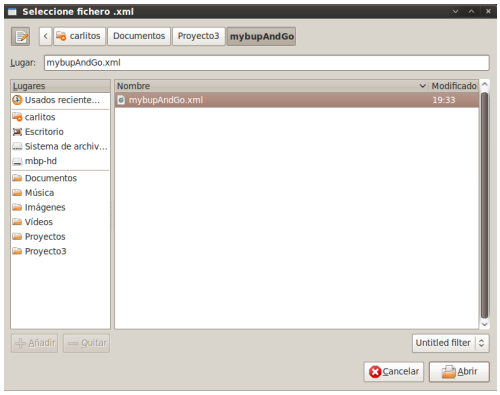


Figura 5.3: Ventana de selección del fichero “XML”.

NOTA: En cuanto se realiza la carga, el proyecto del fichero sustituye al proyecto actual de la aplicación. En esta acción no se necesita configurar el proyecto manualmente, ya que se realiza de forma automática.

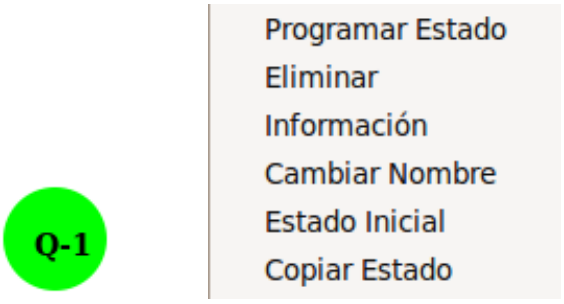
5.1.2. Creación del autómata

En esta sección se comentará las opciones necesarias para la creación y manipulación del autómata de nuestro proyecto.

Los estados

Para trabajar con los estados debemos tener marcada la opción de estados, mediante: *Acción–>Estados*.

NOTA: El menú de propiedades de un nodo se abre posicionando el cursor en el estado (círculo verde) y haciendo “click” con el botón derecho del ratón.



(a) Estado (b) Menú Estado

Figura 5.4: Ejemplo de estado y ventana de propiedades.

- **Crear un estado**: debemos posicionar el cursor sobre la zona de creación del autómata y hacer "click".

- **Eliminación de un estado**: debemos posicionarnos dentro de la zona de creación donde se encuentre el estado a eliminar. Una vez posicionado el cursor, hacer "click" con el botón derecho del ratón y seleccionar la opción *Eliminar*.
  NOTA: Al eliminar un estado se eliminarán todas las transiciones que pertenecen al estado en cuestión.

- **Mover estado**: se debe posicionar encima del mismo y pulsar el botón izquierdo del ratón y sin soltar arrastrar el cursor al lugar deseado.
  NOTA: En esta acción tenemos una propiedad de gran utilidad, que nos permite reorganizar las transiciones de forma automática al mover es estado. Para activarla, debe estar marcada la opción de *Mover todo*, para ello: *Acción–>Mover todo*.

- **Cambiar el nombre**: abriendo el menú de propiedades del nodo y haciendo "click" en la opción *Cambiar Nombre*.

- **Información del estado**: para ver las características de un estado vamos al menú de propiedades del nodo, realizando "click" en *Información*.

- **Programar estado**: Dentro del menú de propiedades tenemos la opción de *programar estado*, donde se abrirá el editor de código con la información de dicho nodo. Una vez abierto el editor, sólo nos falta ponernos a programar el comportamiento que deseamos ejecutar en cada estado.

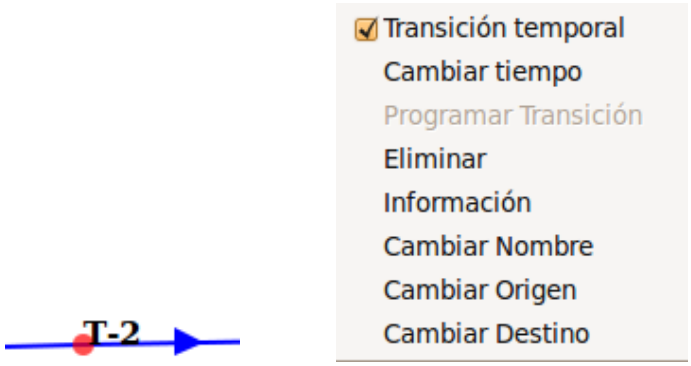
- **Establecer nodo inicial**: se debe ir al menú de propiedades del estado, y seleccionar la opción *Estado inicial*.

- **Copiar un nodo**: para ello el usuario debe seleccionar la opción de *Copiar Estado* dentro del menú de propiedades. Si queremos posteriormente pegarlo, debemos hacer "click" con el botón derecho del ratón en una parte sin objetos dentro de la zona de creación del autómata, donde nos aparecerá un menú emergente con la opción de *Pegar*.
  NOTA: Al copiar estamos haciendo una réplica del nodo sin sus transiciones.

Las transiciones

A la hora de crear, modificar, y eliminar las transiciones y sus propiedades debemos trabajar en el modo transición: *Acción–>Transiciones*.

NOTA: El menú de propiedades de una transición se abre posicionando el cursor en la transición (círculo rojo) y haciendo "click" con el botón derecho del ratón.



(a) Transición (b) Menú Transición

Figura 5.5: Transición y menú de propiedades.

- **Crear transición**: debemos pulsar el botón izquierdo del ratón en el estado origen y soltarlo cuando estemos posicionados en el estado final de la transición.
- **Eliminar transición**: dentro del menú de propiedades, elegimos la opción *Eliminar*.
- **Mover transición**: para ello debemos posicionarnos en la transición (círculo rojo), pulsar el botón izquierdo de ratón y soltarlo en el lugar donde se desea mover.
- **Cambiar nombre**: seleccionar la opción *Cambiar nombre* dentro del menú de propiedades.
- **Características de la transición**: podemos ver las características de la transición en el menú de propiedades, seleccionando *Información*.
- **Cambiar Origen** o **cambiar destino**: dentro del menú de propiedades tenemos la opción de poder cambiar el origen o el destino, para ello debemos seleccionar *Cambiar origen* o *Cambiar destino* y posteriormente hacer "click" en el estado por el que queremos cambiarlo. Automáticamente se nos cambiará la transición.
- **Crear una transición temporal**: podemos transformar cualquier transición a una de carácter temporal, sólo tenemos que activar la opción *Transición temporal* dentro del menú de propiedades. A continuación aparecerá una ventana en la que marcamos el tiempo en milisegundos de la transición (figura 5.6).

- **Cambiar tiempo de la transición temporal**: para ello debemos hacer "click" en *Cambiar tiempo* dentro del menú de propiedades. Sólo esta permitido cuando se encuentra activada la opción de transición temporal.

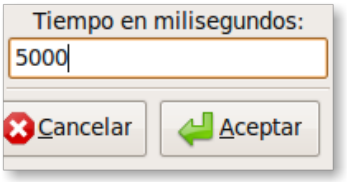
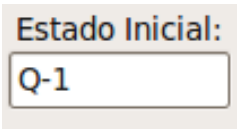


Figura 5.6: Ventana de selección de tiempo en transiciones temporales.

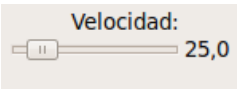
- **Programar una transición**: Debemos seleccionar *Programar Transición* dentro del menú de propiedades. Esta opción sólo se encuentra disponible en transición no temporales. Una vez abierto el editor debemos programar la condición de la transición, para ello sólo debemos poner la condición sin ninguna sentencia "if".

Características del esquema

Para la creación completa del autómata, y poder convertirlo debemos establecer los atributos del esquema. Tenemos que configurar el estado inicial, ya explicado con anterioridad, y debemos establecer la velocidad con que se ejecuta el esquema dentro de *Jderobot*.



(a) Estado inicial



(b) Velocidad del esquema

Figura 5.7: Representación del estado inicial y el cambio de velocidad del esquema

El código global

Una funcionalidad que nos ofrece la aplicación es la posibilidad de compartir código en cada estado o transición. Mediante el botón *Código global* se nos abrirá el editor de código, donde podremos programar variables, constantes o funciones visibles para todos los estados o transiciones.

Las dependencias

Las dependencias son los sensores y actuadores que corresponden a cada nodo. Los tipos que podemos relacionar son *Motors*, *Sonars*, *Laser*, *Encoders* y *Camara*. Par visualizarlas o modificarlas debemos manejar el menú de dependencias.

(a) Sensores y actuadores

(b) Visualización de dependencias

Figura 5.8: Barras de herramientas para los sensores y actuadores

Activación de dependencias

Para establecer las dependencias de cada estado o transición el usuario debe tener marcado la opción de *Nodo* dentro de *Dependencias*. Con esta opción marcada, se muestran las dependencias del nodo seleccionado, por lo tanto sólo basta con pulsar el botón izquierdo del ratón en el estado deseado. Al hacerlo se activarán los botones de los sensores y actuadores para poder modificarlos.

Visualización de dependencias

El usuario tiene la posibilidad de visualizar las dependencias a nivel global o individual. Las opciones que tenemos son las siguientes:

- Ninguna dependencia: si deseamos no mostar las dependencias. Para ello tener seleccionado *Dependencias–>Ninguna*.
- Ámbito general del autómata: si se desea visualizar las dependencias de todos los nodos a la vez y así tener una visión general de las dependencias del autómata. Se consigue seleccionando *Dependencias–>Todas*.
- Dependencias de forma individual: si queremos seleccionar esta opción *Dependencias–>Nodo*, nos mostrará sólo las dependencias del nodo seleccionado.

Conversión del autómata al esquema

Para terminar de desarrollar el autómata, sólo nos falta convertirlo al esquema compatible con *Jderobot*. Para ello sólo debemos pulsar el botón *Convertir*. Una vez convertido sólo nos queda ejecutarlo mediante *Jderobot*, si su compilación ha sido satisfactoria.

5.1.3. Funcionalidades de JDE_DS

La aplicación nos aporta funcionalidades de gran utilidad, y estas son:

- Visualización del autómata: nos permite visualizar la estructura del autómata en una ventana nueva, con la funcionalidad de zoom, para verlo con una visión más general. Para ello pulsar: *Ver–>Visualizar*.

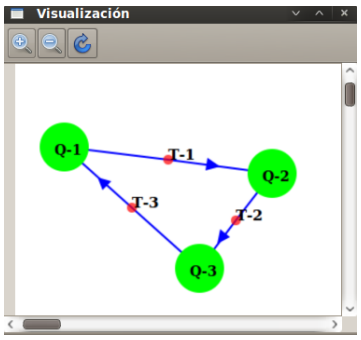


Figura 5.9: Ventana de visualización del autómata.

- Almacenar imagen del autómata: mediante el menú *Archivo–>Guardar Imagen* nos permite guardar en el disco duro en formato "PNG" la imagen de la estructura del autómata.

- Imprimir: esta funcionalidad nos permite imprimir la estructura del autómata. Se realiza mediante *Archivo–>Imprimir*.

5.2. Choca-gira

El objetivo de este primer ejemplo consiste en definir el comportamiento de un robot capaz de no chocarse con los obstáculos encontrados en un entorno, siendo indiferente la dirección tomada por dicho autómata. El autómata creado para desarrollar el comportamiento se aprecia en la figura 5.10.

Para la creación de dicho autómata se elaboró tres estados: *Avanza*, *Atrás*, *Giro*. El primer estado se encarga de avanzar hacia adelante con el robot dando valores a las variables relacionadas con los motores. La labor del segundo estado es dar marcha atrás hasta que se encuentre el robot a una distancia prudencial del obstáculo, esta comprobación se hace dentro de la transición que la precede. Y por último, el estado tres permite realizar un giro determinado a la izquierda. Con estos nodos, tenemos la funcionalidad para poder mover el robot en el entorno, pero sin la adquisición de

los sensores empleados, ni de las transiciones entre estados, no podemos evitar los obstáculos.

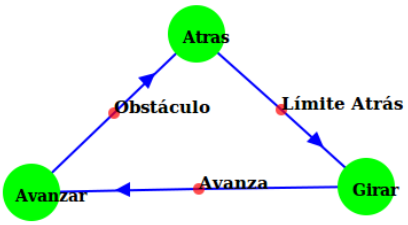


Figura 5.10: Autómata del ejemplo choca-gra

Las transiciones que nos permiten completar el autómata, y poder mover la ejecución de un estado a otro son: *Obstáculo*, *LimiteAtras* y *Avanzar*. La transición primera nos permite pasar del *Avanza* al *Atrás*, esto sucede siempre que se encuentre un objeto a una distancia determinada, gracias a esta comprobación evitamos que el objeto se choque contra el obstáculo. La transición *LimiteAtras* comprueba la distancia que se ha alejado del obstáculo, encaso de ser la suficiente la ejecución pasará del estado *Atrás* al *Giro*. Y por último, la transicióin *Avanzar* nos permite pasar directamente al estado *Avanzar*, debida a que esta transición tiene como condición *True* por lo que siempre cambiará de estado.

Los sensores empleados para dicho autómata depende del robot que elijamos en la ejecución del esquema, en este caso se utiliza un *Pioneer*, en un entorno ejecutado con el simulador *Player*[1] compatible con *Jderobot*. Para la detección de obstáculos se utiliza el sensor *Sonar* y como actuadores los *motors* que nos permiten desplazar el robot en el entorno.

[1]http://playerstage.sourceforge.net/

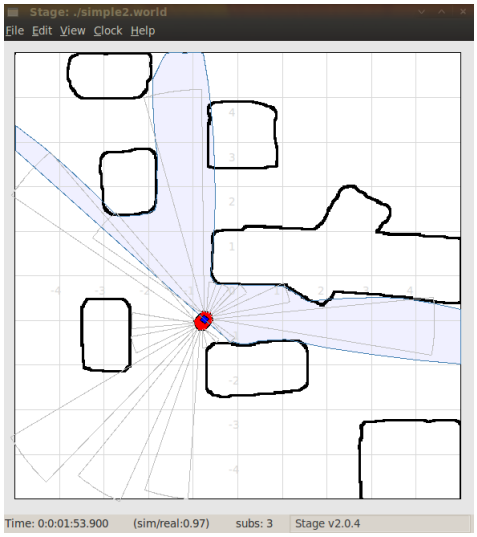


Figura 5.11: Simulación del comportamiento choca-gira

Aunque este ejemplo es bastante sencillo, es idóndeo para apreciar el funcionamiento de la aplicación *JDE_DS*, resaltando el desarrollo del esquema y en particular su propia GUI, en el que podemos apreciar el estado en ejecución. Gracias a esta prueba se ha corroborado la buena estructura elegida dentro del esquema, ya comentada anteriormente, en el que se cambia de un estado a otro sin problemas y con gran fiabilidad.

5.3. Navegación híbrida

El segundo ejemplo creado para la comprobación de la aplicación, se trata de una navegación híbrida. El objetivo es recorrer un circuito de carreras mediante el robot *Pioneer*, evitando el choque del mismo. En este ejemplo el autómata debe recoger el valor de los sensores, los *Sonars*, los que posteriormente se manipulan para recoger la información necesaria y así evitar el choque del robot consiguiendo dar una vuelta completa al circuito.
Al igual que en la prueba anterior, se ha utilizado el simulador *Player* para la ejecución del esquema (figura 5.12).

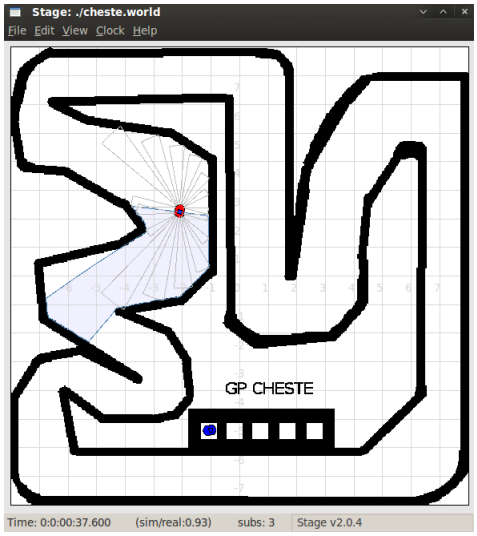


Figura 5.12: Simulación de la navegación híbrida

La estructura del autómata desarrollada mediante *JDE_DS* para dar vida al comportamiento del robot se aprecia en la figura 5.13. Para explicar con mayor detalle su funcionalidad, detallaré un ejemplo. El estado inicial del autómata es *Go*, el cual nos permite ir hacia adelante en caso de que no se encuentre ningún obstáculo. En caso contrarío hay varias posibilidades dependiendo de la cercanía y del lugar del obstáculo encontrado.

- Si nos lo encontramos en un lado del robot y su lado opuesto no hay obstáculo, el autómata obligará hacer un giro hacia el lado que tenga libre. El grado del giro dependerá de la posición del obstáculo en el lateral. Irá de mayor a menor grada según esté más al centro o a un extremo para evitar el choque.

- Si el pioneer encuentra un impedimento por el centro, recorrerá hacia atrás una breve distancia, y calculará el lado donde tenga menor obstáculo para realizar el giro.

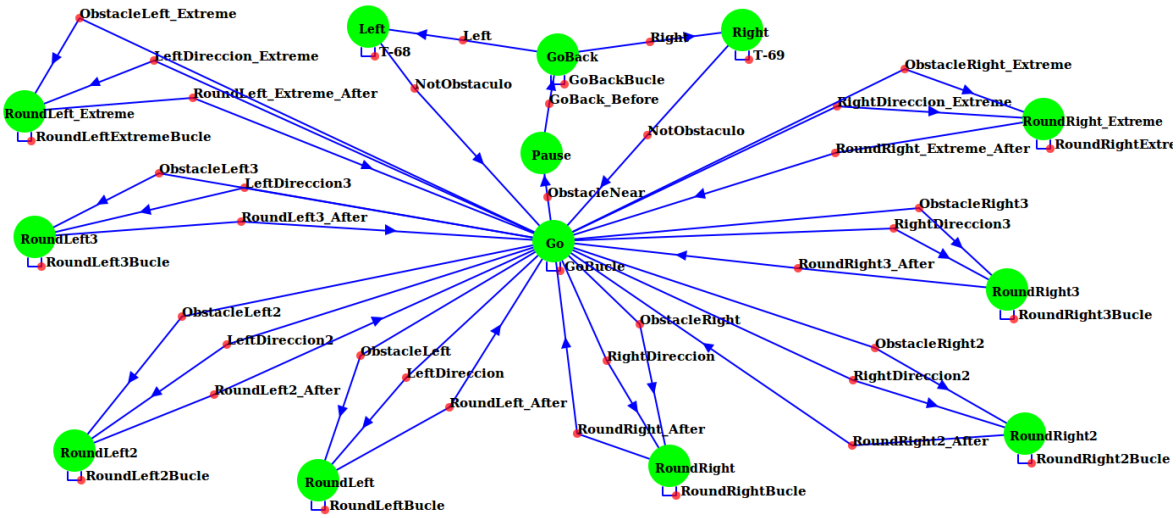


Figura 5.13: Autómata del ejemplo de navegación híbrida

Este autómata lo podemos dividir en cuatro secciones importantes. Las partes de la izquierda y de la derecha permiten mover al robot hacia un lado u otro, dentro de cada una estas dos partes podemos apreciar cuatro nodos encargados de girar en mayor o menor medida al *Pioneer* dependiendo del lugar donde se encuentre el obstáculo. La zona central es la encargada de mover el robot hacia adelante, y por último la zona superior del autómata es la encargada de parar al *Pioneer* en caso de encontrar un obstáculo frontal y calcular el lado con menor obstáculo para girar.

El objetivo de este comportamiento a nivel de la plataforma es demostrar la robustez de *JDE_DS* con un gran número de estados y transiciones, realizando un autómata bastante más complejo que el anterior y viendo su correcto funcionamiento. Con estos dos tests realizados se afirma la gran diversidad de esquemas que se pueden desarrollar con la aplicación.

5.4. Sigue Línea con Gazebo

Por último cabe destacar la última prueba de uso de la aplicación. Como su nombre indica, tiene el robot como objetivo seguir una línea dentro de un entorno simulado con *Gazebo*[2](figura 5.14). Para la realización de este seguimiento se realiza mediante el robot *Pioneer*, en este caso concreto no resulta útil la utilización de los *Sonars*, por lo que se lleva a cabo mediante el sensor *Camara* del robot.

[2]http://playerstage.sourceforge.net/index.php?src=gazebo

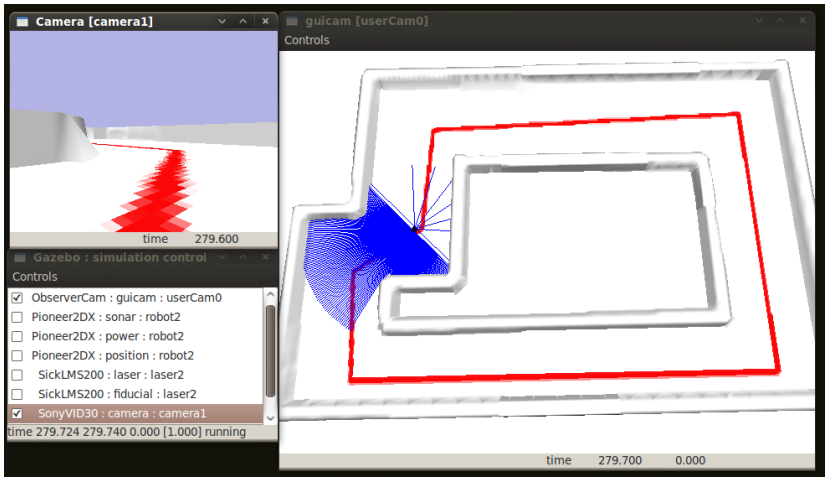


Figura 5.14: Simulación del esquema sigue línea con *Gazebo*

La estructura desarrollada para la construcción del autómata es muy parecida a las pruebas anteriores (figura 5.15) , pero en este caso la recogida de los sensores tiene un estado propio. Dependiendo de la información evaluada por este estado se determinará a que estado se debe transitar. Los estados que mandan información a los actuadores son tres, encargados de realizar un giro a la derecha, a la izquierda o avanzar con el *Pioneer*.

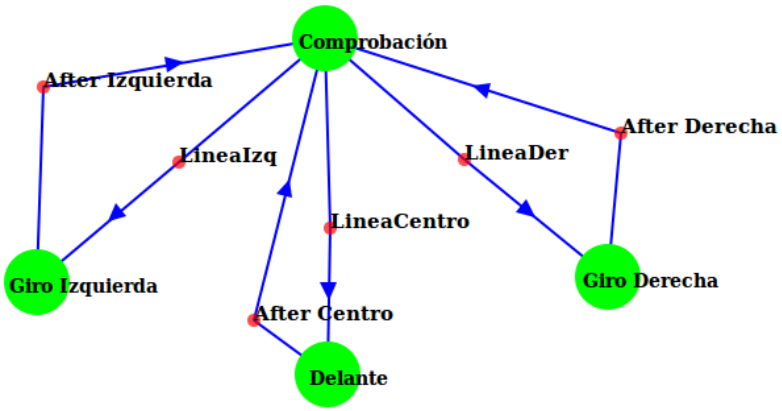


Figura 5.15: Autómata del test sigue línea con *Gazebo*

Dentro del autómata programado cabe destacar la reducción y el aumento de la velocidad dependiendo del grado de giro. En caso de ser muy pronunciado, la velocidad disminuye constantemente; la velocidad aumenta de forma gradual en el caso de no detectar ningún tipo de giro, como en una recta. Estas acciones relacionadas a la velocidad mejoran notablemente el rendimiento en la simulación, ya que evitan que el robot se pase en algún giro.

Una vez desarrollado el comportamiento para poder seguir una línea mediante el robot *Pioneer*, comento los objetivos cumplidos en cuanto a la aplicación se refiere. Gracias al desarrollo de este y de los anteriores ejemplos podemos afirmar que la aplicación funciona de forma eficiente, con gran robustez soportando gran cantidad de nodos y transiciones, y además ofrece al usuario las herramientas cómodas y oportunas para el desarrollo de los comportamientos deseados. Con este último ejemplo se termina la serie de pruebas realizadas con gran éxito para la aplicación.

Capítulo 6
Conclusiones y trabajos futuros

En el presente capítulo se va a exponer las conclusiones obtenidas en la realización de *JDE_DS* y las posibles mejoras que se pueden incorporar en un futuro, todo ello comentando los puntos más relevantes que han permitido el éxito en el proyecto y las dificultades encontradas.

6.1. Conclusión

Como conclusión principal, cabe destacar la culminación de los objetivos propuestos al principio del documento. Se ha desarrollado una aplicación capaz de crear comportamientos para robots mediante la creación de autómatas. Esto se ha conseguido con un autómata programable, dónde cada estado y transición tienen un código definido por el usuario. Los comportamientos creados por la aplicación son compatibles con el *middleware Jderobot*, facilitandonos su simulación.

Se ha cumplido todos los requisitos marcados, implementando una plataforma robusta y de gran intuitividad para el usuario, facilitando el desarrollo de los esquemas, y ayudando a su programación.

Dentro del desarrollo de la aplicación nos hemos encontrado con dificultades. Uno de los objetivos con mayor esfuerzo de realización ha sido la coordinación de la interfaz gráfica de la aplicación con las estructuras internas que llevan toda la información del autómata, como verbigracia la información de los nodos y transiciones almacenados en las tablas hash.
Otro subobjetivo bastante costoso fue el pintado de las transiciones dentro de la zona de creación del autómata, debido a su reorganización autómatica y la existencia de transiciones de tipo bucle. Para ello se tuvo que pensar la funcionalidad que permite su reorganización, abarcando diferentes problemas, como la coincidencia de objetos en una misma posición o la posibilidad de crear transiciones fuera de la zona del autómata.

Durante todo el tiempo de desarrollo de *JDE_DS* he podido enriquecer mi conocimiento. He aprendido sobre bibliotecas gráficas, concretamente con GTK+ donde

he trabajado con una gran diversidad de *widgets*, en primer lugar desarrollando la interfaz gráfica por código y posteriormente mediante el programa *Glade* el cual ha facilitado posteriormente la creación y modificación de interfaces. También he conseguido aprender sobre los ficheros "XML" y su manipulación mediante las bibliotecas. A lo largo del desarrollo he trabajado con un sistema de gestión de versiones "SVN", el cual me ha facilitado el trabajo diario y la organización de diferentes versiones. Entre la gran diversidad de cosas que he aprendido cabe mencionar la herramienta make, el diseño de documentos mediante la sintaxis de LATEX, y la comprensión de la plataforma *Jderobot*.

6.2. Trabajos futuros

JDE_DS no está ni mucho menos paralizado en cuanto a su evolución se refiere, ya que durante su implementación se han pensado la posibilidad de grandes mejoras que aumentarían su funcionalidad. Las mejoras que caben mencionar son las siguientes:

1. Modificar la compatibilidad de los esquemas con la nueva versión *Jderobot 5.0*, ya que se empieza a trabajar con ella.

2. Incluir en la aplicación la capacidad de importar autómatas dentro de otros, permitiendo así la reuitilización de comportamientos.

3. Mejorar la interactividad con el usuario, permitiendo la selección de varios objetos, la posibilidad de arrastar nodos o transiciones de otros proyectos, etc..

4. Incluir mejoras en el editor de texto de la aplicación, como el resalto de la sintaxis, la inclusión del número de líneas, etc..

5. Añadir la funcionalidad del zoom en la zona de creación del autómata, consiguiendo una visión más general durante el desarrollo del autómata.

Bibliografía

[API Gtk+, 2010] Gnome API Gtk+, GLib. *API de Gtk+, GLib, Gnome: http://library.gnome.org*. 1. 2010.

[Harlow, 1999] Eric Harlow. *Desarrollo de aplicaciones Linux con GTK y GDK+*. 1. 1999.

[León,] Luis Alvarez León. Tutorial de latex. *Universidad de Las Palmas de Gran Canaria*.

[Plaza, 2003] José María Cañas Plaza. Jerarquía dinámica de esquemas para la generación de comportamiento autónomo. *Tesis doctoral, Universidad Politécnica de Madrid*, 2003.

[Plaza, 2004] José María Cañas Plaza. Manual de programación de robots con jde. *URJC*, 2004.

[Plaza, 2006] José María Cañas Plaza. Programación de robots móviles. *URJC*, 2006.

[Plaza, 2007] José María Cañas Plaza. Jdec: arquitectura basada en esquemas para aplicaciones roboticas. *URJC*, 2007.

[Vázquez, 2007] Jesús Ruiz-Ayúcar Vázquez. Jdeneo.c- una plataforma de desarrollo de aplicaciones robóticas. *URJC*, 2007.