



UNIVERSIDAD REY JUAN CARLOS
INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Escuela Superior de Ciencias Experimentales y Tecnología

Curso académico 2011-2012

Proyecto Fin de Carrera

Herramienta de programación visual de autómatas de estado finito
jerárquicos para aplicaciones robóticas.

Tutor: José María Cañas Plaza

Autor: Rubén Salamanqués Ballesteros

“Creo que las máquinas podrán hacer cualquier cosa que hagan las personas, porque las personas no son más que máquinas.”

Marvin Minsky

Este documento forma parte del Proyecto Fin de Carrera de la titulación de Ingeniería Técnica en Informática de Sistemas de la Universidad Rey Juan Carlos de Madrid realizado por Rubén Salamanqués Ballesteros.

Se garantiza el permiso para copiar, distribuir y modificar este documento según los términos de la GNU Free Documentation License, Versión 1.2 o cualquiera posterior publicada por la Free Software Foundation, sin secciones invariantes ni textos de cubierta delantera o trasera.

El desarrollo de esta aplicación forma parte del Grupo de Robótica de la Universidad Rey Juan Carlos (www.robotica-urjc.es).

Copyright ©2012 Universidad Rey Juan Carlos.

Resumen.

Actualmente la robótica está experimentando una expansión y desarrollo nunca antes visto. Debido a esto muchos lenguajes de programación han ido adaptándose a esta nueva rama facilitando cada vez más su uso en programación de robots mediante la incorporación de nuevas librerías orientadas a tal efecto. Como alternativa a la programación tradicional mediante lenguajes textuales han ido surgiendo diferentes herramientas que permiten una programación más intuitiva, como son los lenguajes visuales.

El comportamiento de un robot es, en la mayoría de los casos, fácilmente representable mediante autómatas de estado finito. Dichos autómatas, compuestos por estados y transiciones, permiten de una forma muy visual centrarse en el comportamiento requerido del robot. Cada estado, que representa una o varias acciones concretas y cada transición, que marca las pautas para cambiar de un estado a otro permiten ir descomponiendo comportamientos complejos en pequeños bloques mucho más sencillos y entendibles de manera que al final podemos obtener comportamientos muy potentes representados visualmente en autómatas mucho más simples de entender.

Este trabajo tiene como objetivo la mejora de la herramienta ya existente en Jderobot 5.0, VICOE, que permite la programación visual de robots mediante autómatas de estado finito de un sólo nivel, extendiéndola, con la inclusión de jerarquía, en VisualHFSSM. Dicha inclusión añadirá a la aplicación una versatilidad mucho mayor en la representación de nuevos comportamientos, dada la potencia que nos proporciona la jerarquía en la esquematización de los comportamientos que queramos implementar.

Índice general

1. Introducción	1
1.1. Robótica	1
1.2. Software De Robots	6
1.2.1. Programación Visual	8
1.3. Programación de Robots con Autómatas Finitos	11
1.3.1. XaitControl de Xaitment	14
1.3.2. Programación visual de robots en la URJC	16
2. Objetivos	18
2.1. Descripción del problema	18
2.2. Análisis de requisitos	19
2.3. Plan de trabajo	19
3. Entorno y plataforma de desarrollo	20
3.1. GTK+	20
3.2. libGlade	21
3.3. libGnomeCanvas	23
3.4. GtkTreeView	23
3.5. libXml (o Gnome-XML)	23
3.6. Gazebo	24
3.7. JdeRobot	25
4. Descripción informática	27
4.1. Diseño General	27
4.2. Editor Gráfico	29
4.2.1. Distribución de los elementos gráficos	30
4.2.2. Elementos de un subautómata	31
4.2.3. Edición y programación explícita de un subautómata	41
4.2.4. Jerarquía de autómatas	45
4.3. Fichero intermedio	47
4.3.1. Sintáxis del fichero de guardado y carga	47
4.3.2. Guardado y carga del autómata	47

<i>ÍNDICE GENERAL</i>	III
4.4. Generador Automático de código	52
4.4.1. Información sobre las plantillas utilizadas	52
4.4.2. Rellenado de la plantilla	52
5. Resultados experimentales	60
5.1. Prueba de concepto	60
5.2. Alternativas descartadas	61
6. Conclusiones y trabajos futuros	65
6.1. Conclusiones	65
6.2. Trabajos futuros	67
A. Bibliografía	68

Capítulo 1

Introducción

En este proyecto fin de carrera hemos realizado una herramienta con la cual es posible programar visualmente el comportamiento de robots sintetizando dicho comportamiento a través de autómatas de estado finito jerárquicos. Como en los autómatas mononivel los autómatas multinivel o jerárquicos están compuestos de estados y transiciones, cada estado representa una o varias acciones a realizar en ese momento y cada transición marca las condiciones que nos harán pasar de un estado a otro. La particularidad de los autómatas jerárquicos es que cada estado puede desplegar a su vez un 'subautómata hijo' pudiendo recoger de este modo comportamientos mucho más complejos sin renunciar a la simplicidad que nos brinda la representación visual del autómata.

A continuación veremos una breve introducción sobre la robótica, el desarrollo que han ido sufriendo las herramientas de programación visual y una pequeña explicación acerca de lo que son los autómatas de estado finito para comprender la manera en la que implementamos los comportamientos en los robots.

1.1. Robótica

La robótica es la disciplina encargada del diseño, la construcción y la programación de los robots. Según *Robot Industries Association* (RIA), un robot es:

“Un manipulador reprogramable multifuncional, diseñado para mover material, partes, herramientas o dispositivos especializados mediante movimientos programados variables para la ejecución de tareas diversas.”

En robótica se combinan diversas disciplinas como la mecánica, la electrónica, la informática, la inteligencia artificial o la ingeniería de control. Otros campos importantes aplicables en robótica son el álgebra, los autómatas programables y las máquinas de estados.

Aunque la palabra 'robot' aparece por primera vez en la obra teatral de 1921 'RUR' (*Rossum Universal Robota*) y también podemos encontrar referencias a la robótica en películas como "Metrópolis" (*Fritz Lang, 1926*) o las obras literarias de Isaac Asimov (*1942*) no es hasta la década de los '50 cuando surge lo que podríamos considerar el primer robot comercial (*Unimate*,

George Devol y Joseph Engelberger, 1954) atendiendo al significado actual de la palabra. Éste robot fue instalado por primera vez en 1961 en la “*Inland Fisher Guide Plant*”, una planta de fabricación de automóviles de General Motors en Nueva Jersey y su cometido consistía en realizar soldaduras y transportar las piezas soldadas a lo largo de la cadena de montaje. Con este ejemplo entramos en uno de los principales usos a los que los robots han estado destinados hasta la actualidad, las tareas industriales.

Los robots fueron irrumpiendo cada vez con mayor fuerza en este tipo de tareas por ser sucias, peligrosas, difíciles o incluso extremadamente repetitivas como para ser realizadas por un ser humano. Actualmente el sector automotriz sigue siendo el principal exponente en cuanto a uso de robots se refiere. En el caso de una cadena de montaje de automóviles actual podemos encontrar robots que se encargan desde cometidos como soldar el chasis, hasta pintar la carrocería, pasando por tareas de logística para el servicio y transporte de piezas en los almacenes.



(a) Robot para la fabricación de automóviles.

Otra muestra del uso cada vez más común en la industria es el embalaje de productos realizado por los robots de la empresa *ABB*¹, se trata de unos brazos robóticos dotados de visión capaces de organizar y colocar distintos tipos de productos que vayan pasando por delante de ellos en una cinta transportadora.

Otro ejemplo es el de la empresa *Kiva Systems*², recientemente comprada por el gigante de la venta de productos por internet Amazon para gestionar sus almacenes. Estos robots se encargan, de forma totalmente autónoma, de mover y reubicar estanterías de varios pisos de altura llenas de paquetes, siendo capaces de coordinarse entre ellos para elegir la manera más óptima de realizar esos movimientos ahorrando así gran cantidad de recursos en esta tarea.

¹<http://www.abb.es>

²<http://www.kivasystems.com>



(b) Brazo ABB.



(c) Robots de Kiva.

Figura 1.1: Usos en la industria.

Además, los robots se usan ampliamente en actividades como limpieza de residuos tóxicos, localización y desactivación de explosivos, búsqueda y rescate de personas, exploración de volcanes activos o fondos marinos por citar algunos ejemplos. También cuentan con un uso muy destacado en la exploración espacial, donde multitud de robots han aportado la posibilidad de realizar importantísimas misiones como la exploración de Marte mediante las sondas gemelas Spirit y Opportunity o la construcción y mantenimiento de la ISS o EES (*Estación Espacial Internacional*) gracias al complejo brazo robótico *ERA*.



(a) Robot Opportunity en Marte.

Otro de los campos en los que la robótica se está desarrollando cada vez más es la medicina. Actualmente empieza a emplearse equipamiento robótico teledirigido que permite a los cirujanos

realizar operaciones muy delicadas, como por ejemplo la cirugía ocular o la neurocirugía, con una precisión altísima y sin temor a que el pulso tiemble. También se emplean robots en los laboratorios médicos para poder manejar sustancias biológicas potencialmente dañinas con el mínimo riesgo.



(b) Robot Da Vinci.

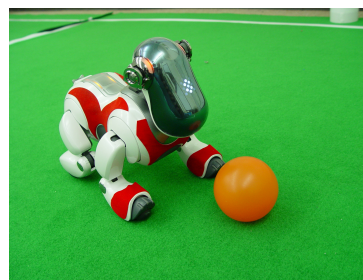
Figura 1.2: Robot en el campo de la medicina.

Teniendo en cuenta la gran expansión que como hemos podido comprobar está teniendo la robótica en multitud de campos no es de extrañar que actualmente también podamos encontrar robots domésticos al alcance de cualquiera. El principal exponente de esto pueden ser los robots aspiradora, actualmente comercializados por multitud de empresas, capaces de ser manejados por cualquier tipo de persona y que gracias a sus comportamientos autónomos son capaces de limpiar pisos enteros sin necesidad de ser programados por el usuario final.

También podemos encontrar otro tipo de robots destinados a la generalidad del público que son los robots de ocio. En los últimos años han surgido varios productos que consisten en robots técnicamente muy avanzados si nos fijamos en los estándares de hace tan sólo 30 años y que no son más que "juguetes" para gran parte de la población actual. El primer ejemplo verdaderamente influyente de esto es el Robot Aibo (*Sony, 1999*), un pequeño perro robot comercializado como mascota, o, el ladrillo NXT (*LEGO, 2006*), una pequeña unidad programable que mediante el ensamblaje de las famosas piezas de lego y una serie de sensores y actuadores diseñados para el paquete permite la creación de robots.



(a) Robots aspiradora.



(b) Aibo de Sony.

Figura 1.3: Robots en la vida cotidiana.

En el terreno de la investigación podemos destacar 2 robots de referencia. El primero es el robot NAO (*Aldebaran Robotics, 2004*) del que se han ido desarrollando versiones cada vez más capaces y que es uno de los robots principales usados en la Robocup.³ El segundo es el robot Asimo de Honda, un pequeño robot humanoide de poco más de un metro de altura que se puede considerar como la punta de lanza en cuanto a robótica humanoide y capacidades humanas aplicadas en robots se refiere, su primera versión data del año 2000 y desde entonces ha estado en continuo desarrollo pasando por varias versiones, actualmente es uno de los robots más avanzados del mundo si nos centramos en sus capacidades motrices, gestión del equilibrio, capacidad de aprendizaje y relaciones con el entorno y con humanos.



(a) NAO.



(b) Asimo.

Figura 1.4: Líneas de investigación actuales.

Vistas las aplicaciones actuales de los robots podemos dar una ligera descripción acerca de la construcción de éstos. Un robot es un ingenio mecánico compuesto por estructura, sensores y actuadores regidos o regulados todos ellos mediante un sistema de control o computador. Gracias

³<http://www.robocup.org>

a estos elementos el robot puede percibir su entorno, realizar acciones o movimientos y hacerlo siguiendo un patrón de comportamiento. Debido a la diversidad posible tanto en diseños físicos como en programaciones posibles un robot es una herramienta altamente polivalente de ahí su desarrollo cada vez mayor.

A continuación veremos una enumeración algo más detallada de las características de los elementos principales de un robot citados anteriormente:

1. **Sensores.** Constituyen el sistema de percepción del robot. Los sensores facilitan la información necesaria para que los robots interpreten el mundo real. Son dispositivos capaces de medir magnitudes físicas como distancia, sonido, temperatura, presión, velocidad... Hay que ser conscientes de que los sensores son inexactos y sensibles al ruido. Se pueden clasificar según el tipo de información que nos proporcionan en *internos* o *externos*. Los *Internos* proporcionan información sobre el propio robot como posición, velocidad o aceleración, los *externos* proporcionan información sobre el entorno del robot como proximidad, tacto, fuerza o visión.
2. **Actuadores.** Son dispositivos encargados de dotar de movimiento a los elementos del robot ejerciendo fuerzas para llevar a cabo alguna acción. Éstos dispositivos permiten al robot interactuar con el entorno. Existen diferentes tipos dependiendo de la tecnología que utilicen: pueden ser neumáticos, hidráulicos y eléctricos.
3. **Procesadores.** Circuito integrado que coordina la actuación y percepción del robot. Dicha coordinación viene determinada por la programación que se ejecute en el hardware de cómputo.

1.2. Software De Robots

En los inicios de esta disciplina el desarrollo de robots se hacía de una forma “artesanal”. No existía ninguna metodología más o menos estandarizada para garantizar una arquitectura *software* coherente. Únicamente se utilizaban los *drivers* que proporcionaba el fabricante de cada robot para realizar estas aplicaciones. En este caso el sistema operativo era mínimo, básicamente una colección de *drivers* con rutinas para leer y escribir datos de los sensores y actuadores (Figura 5.2(a)).

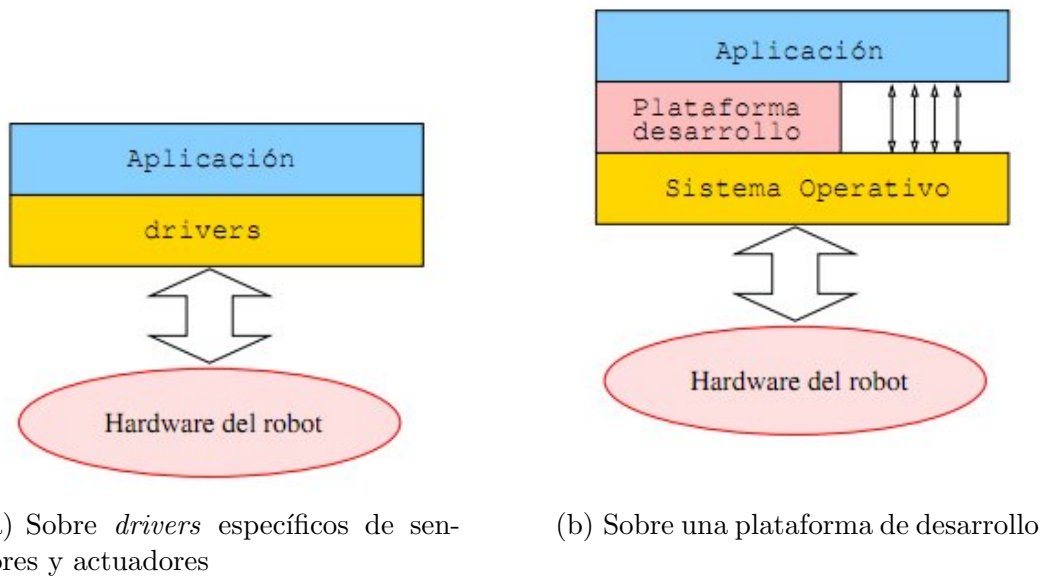


Figura 1.5: Programación de Robots

Esta forma de desarrollar software es complicada, costosa y muy poco reutilizable, es por ello que se fue sustituyendo por las plataformas de desarrollo. Actualmente el software de los robots se divide en tres niveles: sistema operativo, plataforma de desarrollo y aplicaciones. Al introducir un “puente” entre el SO y las aplicaciones el desarrollo de software se simplifica ya que la plataforma de desarrollo nos ofrece un acceso más sencillo y estandarizado a los sensores y los actuadores (como puede ser hacer girar un servomotor o leer los datos de un sensor de luz), cuentan con librerías que contienen funcionalidades comunes permitiendo, con ello se afronta mejor la creciente complejidad en los comportamientos cada vez más complejos que se pide de los robots.

El correcto comportamiento de un robot autónomo depende enteramente del software que lo gobierna. Dicho software debe cumplir una serie de requisitos que le confieren ciertas peculiaridades en comparación con los desarrollos realizados en entornos más tradicionales.

Al ser un software que regirá un objeto físico hay que tener en cuenta la capacidad de medir esa realidad física y de interactuar con ella, esto significa que el software de control de robots tiene que ser ágil para ser capaz de adaptarse a un mundo cambiante. Tiene que ser capaz de estar atento a varios elementos u objetos a la vez (tanto la multitud de sensores que componen al propio robot como a elementos externos que puedan suponer amenazas u objetivos en la realización de la tarea encomendada).

Aún con la inclusión de la arquitectura en tres niveles que hemos explicado antes, la programación de robots sigue sufriendo de alta dificultad en la reutilización del código. Esto es debido a

la gran heterogeneidad existente tanto a nivel hardware como, en consecuencia, a nivel software, haciendo que muchas veces los desarrollos robóticos tengan que empezar casi desde cero para adaptarse a las características de cada robot. Como hemos visto, las librerías de las plataformas de desarrollo intentan paliar este problema ofreciendo un nivel de abstracción y proporcionando algoritmos comunes de navegación local, procesamiento de imágenes, etc.

Otra herramienta que facilita las pruebas del software paliando la heterogeneidad del hardware son los simuladores. Éstos son capaces de proporcionar un entorno virtual donde probar las distintas soluciones programadas de forma mucho más rápida, con menos costes y con la ventaja de poder introducir cambios físicos en el robot fácilmente. Un ejemplo de este tipo de programas es el simulador Gazebo que explicaremos más adelante como parte de la infraestructura de este proyecto.

Centrándonos en los lenguajes utilizados para la programación de robots nos encontramos que éstos no difieren con los que podemos utilizar en la programación de cualquier otra aplicación informática, podemos usar Java, C/C++, Python y muchos otros. Lo que hace que un lenguaje pueda ser usado para programar robots son las librerías que puedan estar desarrolladas para él. Esto es posible gracias a compiladores cruzados que son capaces de generar código ejecutable en el robot.

A parte de los lenguajes tradicionales ha habido intentos de crear lenguajes específicos para programar robots que contasen con primitivas propias de la robótica. En este tipo de lenguajes se encuentran *Task Description Language* (TDL) o *Reactive Action Packages* (RAP). Pero lo que realmente se puede considerar un avance en la programación específica es el surgimiento de lenguajes de programación visual. Un ejemplo muy ilustrativo es el lenguaje código-RCX de Lego. Creado para la programación de su juguete RCX (y más tarde utilizado en su evolución, el NXT) y pensado para poder ser manejado por niños, consta de bloques visuales que representan acciones o condiciones y permite, mediante el apilamiento de dichos bloques, ir componiendo un comportamiento mas o menos complejo de una forma tremendamente sencilla.

1.2.1. Programación Visual

Con el caso de Lego y su lenguaje código-RCX hemos introducido lo que nosotros consideramos por programación visual, que sería la capacidad de realizar un desarrollo software utilizando únicamente elementos visuales. En este tipo de lenguajes, además de ser posible una programación muy intuitiva y didáctica, se puede observar de una manera muy clara aspectos esenciales en un programa informático como es el flujo de ejecución, sus condiciones, bucles, etc.

Debido a esta sencillez aportada por los lenguajes de programación visual (LPV) el aprendizaje es mucho más intuitivo permitiendo hacer llegar el desarrollo de un programa informático

a un grupo mucho más amplio de personas.

Un LPV puede definirse como:

- Un lenguaje de programación que usa una representación visual (tal como gráficos, dibujos, animaciones o iconos, parcial o completamente).
- Un lenguaje que manipula información visual o soporta interacción visual, o permite programar con expresiones visuales.
- Un conjunto de símbolos de texto y gráficos con una interpretación semántica que es usada para comunicar acciones en un entorno.
- Lenguaje de programación donde se usan técnicas visuales para expresar relaciones o transformaciones en la información.

Hay que aclarar que un LPV no es un entorno integrado de desarrollo (IDE). La diferencia es que un LPV debe ser capaz de llevar a cabo todas las tareas de programación de forma visual, sin tener que recurrir a la representación textual.

Entonces, ¿por qué insistimos en comunicarnos con las computadoras usando lenguajes de programación textuales? ¿No sería mejor comunicarnos con las computadoras usando una representación que aproveche nuestra naturaleza visual? Obviamente, los autores de los lenguajes de programación visuales/(LPV) discuten que la respuesta a ambas preguntas es sí. Las principales motivaciones para la mayoría de la investigación en LPV son:

- Mucha gente piensa y recuerda cosas en términos de cuadros.
- Se relacionan con el mundo de una manera intrínsecamente gráfica y utiliza imágenes como componente primario del pensamiento creativo.

Lenguaje Gráfico RCX-Code de Lego

Un ejemplo de lenguaje visual en robótica es el lenguaje RCX⁴ de Lego. Está basado en iconos o bloques que permiten crear diagramas que representan el comportamiento del robot.

El Sistema de Invención Robótica de Lego Mindstorms NXT fue pensado, en principio, para niños. El lenguaje oficial del RCX se caracteriza por su simplicidad y legibilidad. Este sistema propone programar robots montando bloques de comandos gráficos. Estos bloques de comandos se conocen como *Lego bricks* y representan instrucciones para el robot. Mediante estos bloques puede lograrse que el robot se mueva, emita un sonido o que reaccione a distintos eventos.

⁴http://www.ni.com/swf/lv_lego/us/lego_lv_demo.swf

El entorno de programación es una interfaz de programación visual orientada a bloques gráficos. Existen iconos que representan bloques de código que realizan acciones concretas. El usuario puede elegir de una lista de comandos que están en un cajón gráfico y colocarlos en un orden lógico que el RCX puede ejecutar componiendo así el esquema del comportamiento del robot (Figura 1.6). Según Lego, “... es un entorno de programación visual que permite a los niños recoger, soltar y apilar comandos y trozos de código.”

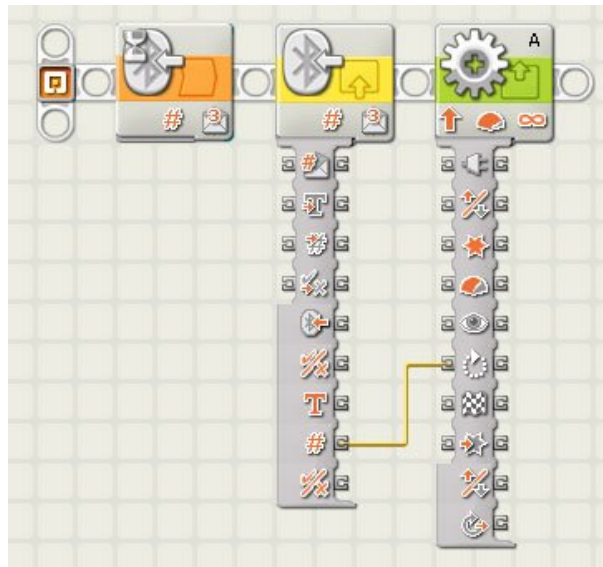


Figura 1.6: Imagen Construcción de Lego.

Los distintos bloques que se encuentran en el cajón de bloques son:

- *Motor*: Este bloque nos permite controlar todos los motores del robot.
- *Sonido*: Con este bloque el robot emitirá sonidos.
- *Pantalla*: Este bloque permite determinar lo que queremos que aparezca en el *display* del ladrillo.
- *Esperar (Wait)*: Este bloque permite que el programa espere una determinada cantidad de tiempo o que espere a que un sensor sea disparado.
- *Repetir (Bucle)*: Este bloque permite repetir un conjunto de comandos hasta que se cumpla una condición o un tiempo limitado.
- *Condicionales*: Estos bloques permiten adherir un salto en el programa basado en la lectura de un sensor. Debe decidir por cuál de las dos ramas del bloque deberá continuar, en función del valor de un dato o de un sensor.

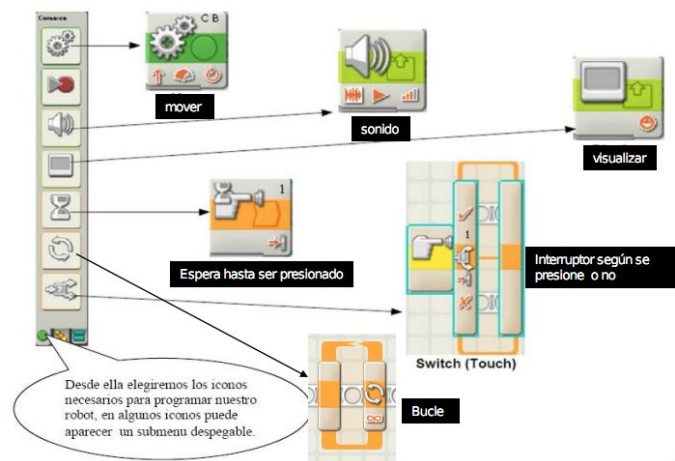


Figura 1.7: Bloques Lego.

De esta manera, usando una interfaz visual como la del Lego, se puede programar robots físicos sin introducir código en forma de texto. La interfaz visual permite al usuario independizarse de ciertos detalles de los lenguajes de programación, permitiendo reducir el tiempo de desarrollo de aplicaciones de todo tipo (no sólo en ámbitos de pruebas, control y diseño).

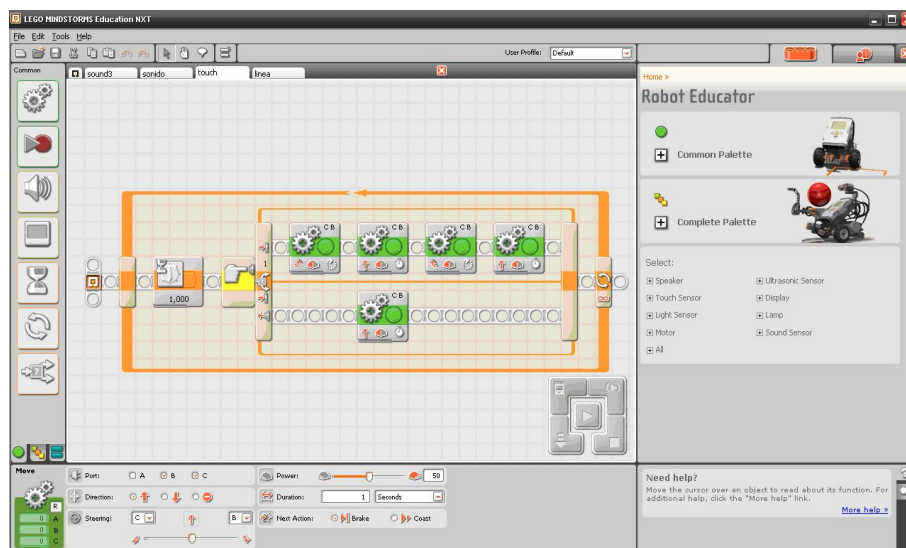


Figura 1.8: Programa general Lego.

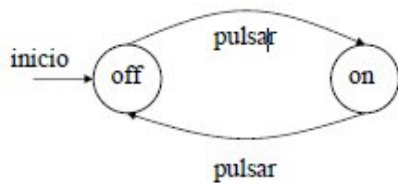
1.3. Programación de Robots con Autómatas Finitos

Un autómata finito (AF) o máquina de estado finito, es un modelo matemático que realiza cálculos de forma automática sobre una entrada para producir una salida. Es un dispositivo abstracto que es capaz de recibir información, cambiar de estado y transmitir información.

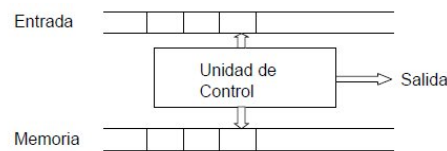
Este modelo está conformado por un alfabeto, un conjunto de estados y un conjunto de transiciones entre dichos estados. Su funcionamiento se basa en una función de transición, que recibe en un estado inicial una cadena de caracteres pertenecientes al alfabeto (la entrada), y que va leyendo dicha cadena a medida que el autómata se desplaza de un estado a otro, para finalmente detenerse en un estado final o de aceptación, que representa la salida.

Este modelo abstracto de una máquina de estado finita o FSM puede:

- Leer los símbolos en la entrada
- Produce símbolos en la salida
- Tiene una unidad de control que puede estar en uno de sus posibles estados internos
- Puede cambiar de los estados internos en función de la entrada
- Puede tener algún tipo de memoria



(a) Autómata.



(b) FSM.

Figura 1.9: Ejemplo.

Formalmente, un autómata finito es una 5-tupla $(Q, \Sigma, q_0, \delta, F)$ donde:

- Q es un conjunto de estados.
- Σ es un alfabeto.
- $q_0 \in Q$ es el estado inicial.
- $\delta : Q \times \Sigma \rightarrow Q$ es una función de transición.
- $F \subset S$ es un conjunto de estados finales o de aceptación.

Los autómatas finitos, según el tipo de proceso que ejecutan, pueden ser:

- Aceptadores: Resuelven problemas con respuesta si/no, que se modeliza normalmente como la identificación de dos estados finales, uno de aceptación y otro de rechazo.

- Generadores: Son los que generan símbolos de un alfabeto finito, llamado alfabeto de salida.
- Transductores: Son los que traducen un lenguaje de entrada a un lenguaje de salida, es decir, dado un lenguaje de entrada generan un lenguaje de salida. En general, establecen una relación entre dos lenguajes formales.

Existen dos tipos de autómatas finitos:

1. Autómata finito determinista (AFD): Cada estado de un autómata de este tipo tiene una única transición por cada símbolo del alfabeto (Figura 1.10(b)).
2. Autómata finito no determinista (AFND): Los estados de un autómata de este tipo pueden, o no, tener una o más transiciones por cada símbolo del alfabeto. El autómata acepta una palabra si existe al menos un camino desde el estado q_0 a un estado final F etiquetado con la palabra de entrada. Si una transición no está definida, de manera que el autómata no puede saber cómo continuar leyendo la entrada, la palabra es rechazada. Además de ser capaz de alcanzar más estados leyendo un símbolo, permite alcanzarlos sin leer ningún símbolo (Figura 1.10(a)).

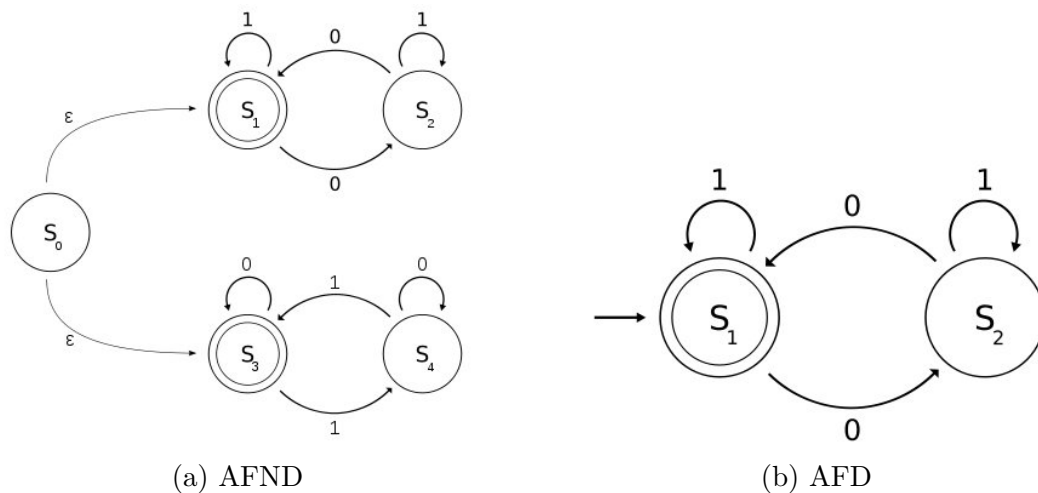


Figura 1.10: Ejemplo de Autómatas.

Utilizaremos este tipo de modelo (AFD) para simbolizar el comportamiento de un robot. El comportamiento general vendrá definido por cada estado o subcomportamiento, es decir, que tarea concreta se realizará en cada estado. Podrá pasar de unos estados a otros mediante transiciones (condiciones de permanencia o de cambio), dependiendo de determinados eventos producidos, tanto internos como externos. Así, el comportamiento de un robot cuyo objetivo es no chocarse, podría definirse tal y como se muestra en la figura 1.11.



Figura 1.11: Ejemplo de autómata que define el comportamiento de un robot.

1.3.1. XaitControl de Xaitment

XaitControl es una herramienta de programación visual de autómatas jerárquicos desarrollada por la empresa Xaitment ⁵. Su concepción e implementación es tremendamente parecida a lo que nosotros pretendemos desarrollar en este PFC aunque obviamente, al tratarse de una aplicación comercial cuenta con ciertas funcionalidades más completas y avanzadas.

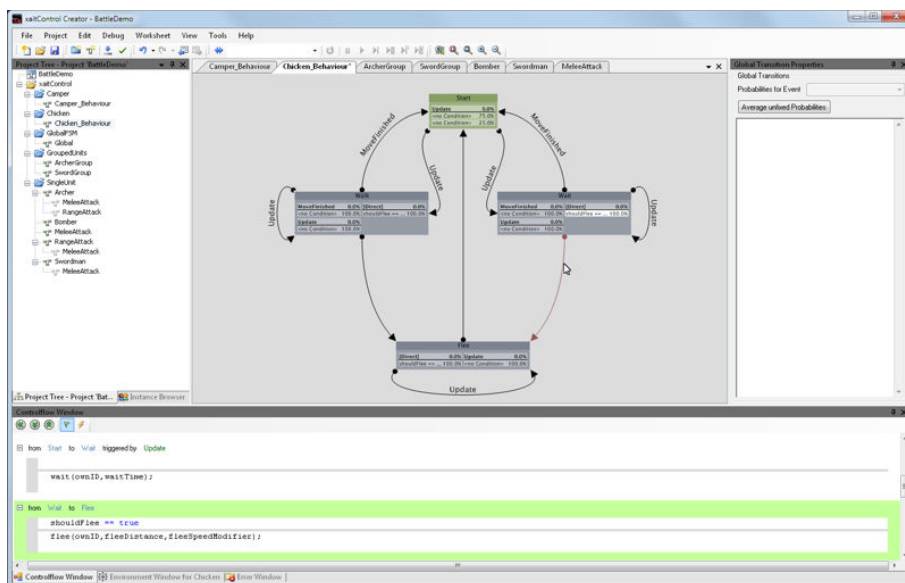


Figura 1.12: Captura de XaitControl.

La aplicación cuenta con una zona principal para el pintado de los autómatas, un vista de árbol lateral que muestra toda la estructura creada y otros paneles que muestran distintas informaciones sobre procedimientos auxiliares, control del flujo de ejecución, etc.

También cuenta con una ponderación o asignación de probabilidad de éxito en las posibles transiciones entre los diferentes estados. Esto significa que con esta herramienta se pueden

⁵<http://www.xaitment.com/>

desarrollar AFND (autómatas finitos no deterministas). Así permite, por ejemplo, que dos transiciones tengan el mismo estado de origen y el mismo destino pero que dependiendo de ciertas condiciones se transite por una o por otra ocasionando que las acciones del estado transitado sean distintas en función de la transición elegida.

Permite la creación de proyectos compuestos por uno o varios autómatas jerárquicos cada uno y permite ver todas estas dependencias mediante la columna de la izquierda que como ya habíamos comentado consiste en una vista en árbol. Además, mediante el uso de pestañas, permite tener varios subautómatas abiertos a la vez en distintos lienzos o usar también pestañas para tener un editor de texto y poder editar así los diferentes estados.

La propia herramienta cuenta con un compilador que permite lanzar la aplicación programada usándose la misma interfaz para ver los progresos, pudiendo establecer puntos de parada en el flujo de ejecución, avance instrucción por instrucción y otras funcionalidades comunes en cualquier depurador.

Esta aplicación es muy parecida a lo que será la creada por nosotros ya que no es estrictamente programación visual como sí es el código-RCX de Lego visto anteriormente pero se apoya en elementos visuales para la representación de autómatas facilitando enormemente el entendimiento y la programación de comportamientos y, gracias a que incorpora jerarquía, la complejidad de éstos puede ser muy alta sin renunciar a esa simplicidad visual.

Actualmente se usan herramientas como esta en el control de los comportamientos de la IA (Inteligencia Artificial) de los adversarios en los videojuegos, viendo así que estas herramientas no tienen que estar directamente ligadas a robots.

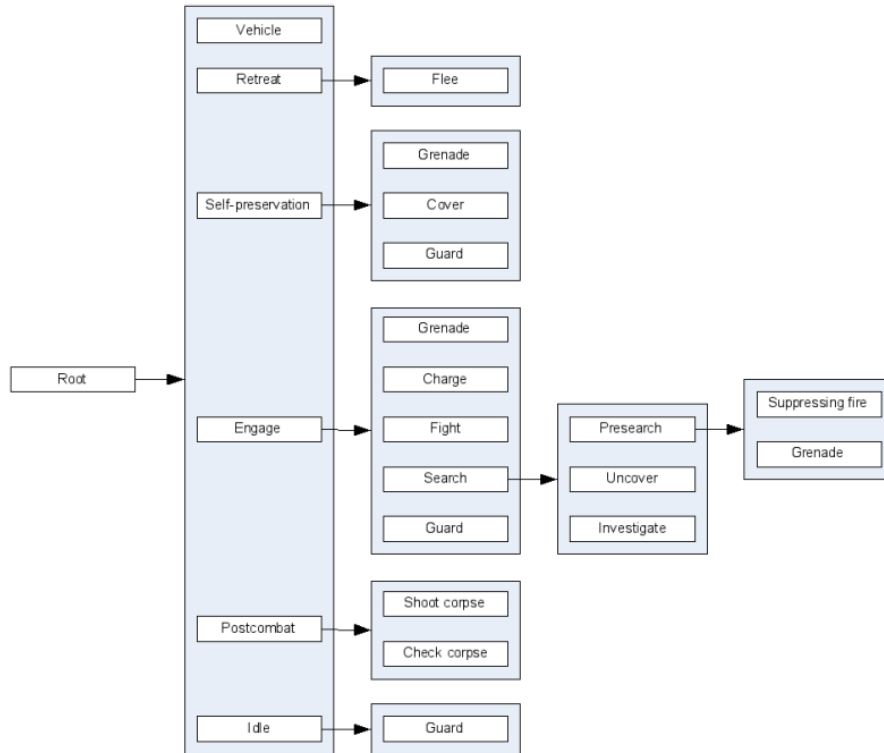


Figura 1.13: Ejemplo de autómata del videojuego Halo 2.

En la captura se observa un ejemplo de comportamiento programado mediante este tipo de herramientas en el que vemos como los enemigos van actuando en base a un autómata jerárquico. Cada acción del primer nivel despliega uno o más niveles especializando con cada paso las acciones a ejecutar.

1.3.2. Programación visual de robots en la URJC

Centrándonos cada vez más en nuestro entorno hay que hacer mención de las herramientas en las que hemos basado este proyecto. La primera de ellas es la aplicación creada por Carlos Iván Martín (<http://jde.gsync.es/index.php/Cmartin-pfc-itis>). Esta aplicación es la primera versión completa desarrollada en la Universidad Rey Juan Carlos para trabajar en la plataforma jderobot, en este caso su versión 4.3. Permitía el desarrollo y la programación de robots mediante autómatas mononivel que funcionaban sobre el componente básico de aquella versión de jderobot.

La segunda versión fue desarrollada por David Yunta Garro (<http://jde.gsync.es/index.php/Dyunta-pfc-itis>). La principal mejora introducida fue la migración de la aplicación a la nueva versión de jderobot, la 5.0. Esta versión nos permitía seguir desarrollando autómatas mononivel y generaba código ejecutable basado en el componente *Introrob*. Contaba con una interfaz rediseñada y una gran mejora en los componentes generados al incorporar un GUI automático. En este GUI mostraba en tiempo de ejecución los estados por los que el robot iba transitando a medida que ejecutaba su comportamiento programado. Gracias a esto se facilitaban enormemente las tareas de depuración.

Una vez finalizado el capítulo de introducción, en el segundo capítulo describimos los objetivos concretos y fijaremos los requisitos que debe cumplir la herramienta a desarrollar en este proyecto. En el capítulo de entorno y plataforma de desarrollo se analizan en detalle las herramientas *software* que se han empleado a lo largo de la implementación. En el cuarto capítulo, dedicado a la descripción informática, se cuenta en profundidad la solución adoptada para llevar a cabo los objetivos planteados. En el siguiente capítulo comprobamos experimentalmente el funcionamiento de la infraestructura desarrollada mostrando varios comportamientos de un robot generados con la herramienta. Finalizamos la memoria describiendo las conclusiones y las líneas futuras de trabajo a partir de este proyecto.

Capítulo 2

Objetivos

Tras haber presentado el contexto general del proyecto pasamos a fijar sus objetivos y los requisitos que planteamos al inicio del proyecto.

2.1. Descripción del problema

El objetivo general de este proyecto es realizar una herramienta de desarrollo para facilitar la programación de robots de manera gráfica basada en autómatas jerárquicos, en concreto para un determinado robot de referencia con un conjunto conocido de sensores y actuadores (ver 3.6).

Esta herramienta representará el comportamiento del robot de manera gráfica en un lienzo mediante un autómata finito de estados jerárquico compuesto de transiciones y estados, incorporará el código correspondiente a cada etapa o estado y generará el código resultante en varios ficheros. De esta forma se consigue una representación visual del funcionamiento del robot, teniendo así una aproximación a un LPV (lenguaje de programación visual). Aunque en este caso es necesario introducir parte del código en forma de texto, el desarrollo queda más sencillo en un entorno gráfico y fácil de interpretar.

Para conseguir esto, el problema se ha dividido en varios subobjetivos:

- Programar un editor Gráfico: Es la parte principal de la herramienta, en él se plasmará en forma de autómata el comportamiento deseado. Debe ser capaz de mostrar y navegar entre los distintos niveles del autómata jerárquico y por supuesto editar cada estado y transición para añadir el código correspondiente.
- Programar un generador automático de código: Será el encargado de traducir desde la implementación gráfica (a través de una plantilla jerárquica “general”) a código en texto plano compatible con la plataforma *Jderobot 5.0*, concretamente generará código en C++.
- Validación experimental: Se generarán varias aplicaciones robóticas que materialicen distintos comportamientos de un robot para comprobar el correcto funcionamiento de la herramienta.

2.2. Análisis de requisitos

Una vez comentados los objetivos, los requisitos de partida a los que deberá ajustarse el desarrollo del proyecto son:

1. La herramienta debe representar el autómata de forma visual y simplificada, de forma que permita su comprensión.
2. La herramienta debe soportar un gran número de estados y transiciones y debe permitir su manipulación de forma ágil así como la navegación entre niveles.
3. El código generado por la herramienta debe ser compatible con la plataforma de aplicaciones robóticas *Jderobot 5.0*.

2.3. Plan de trabajo

El desarrollo del proyecto ha seguido la planificación siguiente:

1. *Familiarización con la plataforma Jderobot y el simulador Gazebo*: Empezamos el proyecto instalando la plataforma y ejecutando componentes sencillos (introrob) ya hechos con el simulador.
2. *Familiarización con las librerías Gtk+ y la aplicación Glade*: Modificamos el componente básico introrob para añadirle elementos que nos servirían en el desarrollo posterior de la herramienta (como por ejemplo un canvas) y así familiarizarnos con su manipulación.
3. *Realización del editor gráfico*: Para ello se ha utilizado Gtk y libGnomeCanvas para generar toda la estructura y funcionalidad de la interfaz.
4. *Diseño del formato intermedio*: Se utilizará el formato XML para guardar toda la información del esquema a generar. Se definirá la forma en que la información será guardada.
5. *Generación automática del código*: Se procesará la información guardada para elaborar de forma automática el esquema resultante.
6. *Experimentos y realización de ejemplos*: Aquí se probará toda la plataforma desarrollada con varios ejemplos concretos. Para ello se estudiará la plataforma *Jderobot* y se adaptarán los ejemplos a esta. Para simular los robots se utilizará *Stage* o *Gazebo*.

A lo largo del proyecto se han tenido reuniones semanales con el tutor para ir planificando cada paso, corregir fallos de etapas anteriores e ir avanzando e incorporando nuevas mejoras.

Capítulo 3

Entorno y plataforma de desarrollo

En este capítulo presentamos las herramientas en las que nos hemos apoyado para realizar el proyecto, el robot típico en el que se ejecuta la aplicación, así como la plataforma software sobre la que se ha realizado.

3.1. GTK+

GTK+ o The GIMP Toolkit es un conjunto de bibliotecas multiplataforma para desarrollar interfaces gráficas de usuario (GUI). GTK es una interfaz orientada a objetos para programadores de aplicaciones (API). Está escrita en C principalmente, pero tiene enlaces a muchos otros lenguajes de programación como C++, Python y C# entre otros.

GTK+ se basa en dos colecciones de librerías principales entre otras: GLib y GDK. GLib es una biblioteca de bajo nivel que envuelve la mayor parte de las funciones de la biblioteca estándar de C para el manejo de estructura de datos, portabilidad, interfaces para funcionalidades de tiempo de ejecución como ciclos, hilos, carga dinámica o un sistema de objetos. Por otro lado está GDK, biblioteca que actúa como intermediario entre gráficos de bajo nivel y gráficos de alto nivel.

Cada pieza o elemento de una interfaz gráfica es denominada Widget. Un widget puede representar un botón, textos, ventanas, una barra de desplazamiento, cuadros de diálogos... Existen muchos tipos de widgets, los cuales derivan todos de la misma clase (herencia), como se ve en la figura. Una interfaz se podría definir como un contenedor de widgets.

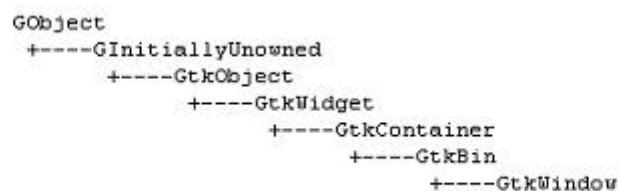


Figura 3.1: Clase GtkWidget.

La forma de interactuar con estos widgets es mediante el uso de eventos. Cuando ocurre un evento, tal como presionar un botón, o cerrar una ventana, la señal apropiada será emitida por el widget afectado. Esta señal podrá ser capturada y conectada a una función apropiada (*callback*) mediante el manejador de señales. Esta función será llamada cuando la señal sea detectada. Así, para que un botón realice una determinada acción, sólo se tendrá que conectar dicha señal a la función correspondiente.

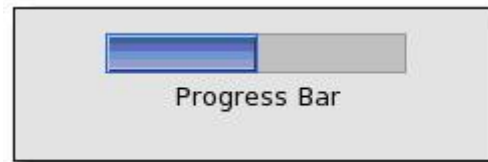


Figura 3.2: Ejemplo de GtkWidget.

Gtk+ dormirá en un bucle principal `gtk_main ()` hasta que se produzca un evento y entonces el control se pasará a la función o callback apropiado. Una vez finalizado el callback, el control vuelve al bucle principal y espera a que se produzca otro evento. Este bucle principal es quien espera la entrada de datos en la interfaz.

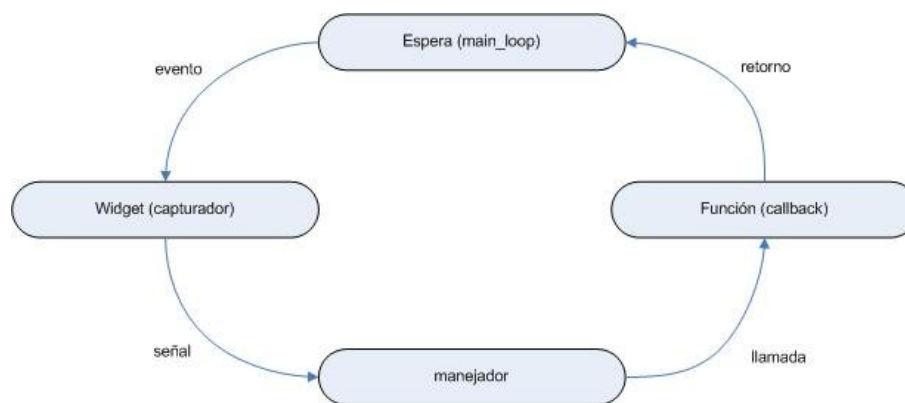


Figura 3.3: Bucle de Gtk.Main().

Hemos usado Gtk para realizar todo el diseño y funcionalidad del editor gráfico de autómatas.

3.2. libGlade

Glade es un RAD (desarrollo rápido de aplicaciones) para el diseño de aplicaciones GTK+. Es una herramienta para simplificar el proceso de elaboración de una interfaz gráfica.

Generar una aplicación desde cero puede resultar costoso en cuanto a tiempo de codificación si se hace directamente desde código, y más aún si constantemente se van haciendo modifica-

ciones y ampliaciones. Por ello, resulta más fácil si a la hora de realizar el diseño se hace mediante un LPV o una plataforma de desarrollo. De esta forma, se podrá ir viendo el diseño del GUI a medida que se va construyendo (WYSIWYG o What you see is what you get).

¿Por qué Glade? Glade permite al desarrollador un diseño rápido y eficiente de una aplicación visual y entonces concentrarse en la implantación del programa en vez de preocuparse de los problemas del interfaz de usuario.

Glade originalmente generaba código C a partir de sus diseños de interfaz en Glade. Glade versión 2 también proporciona este servicio. Esta opción no es recomendable ya que sólo se aplica al lenguaje C. Glade también permite generar o guardar el proyecto en un fichero .glade (documento XML) la cual es preferible, ya que permite cargar el interfaz gráfico en tiempo de ejecución, lo que resulta mucho más mantenible y flexible.

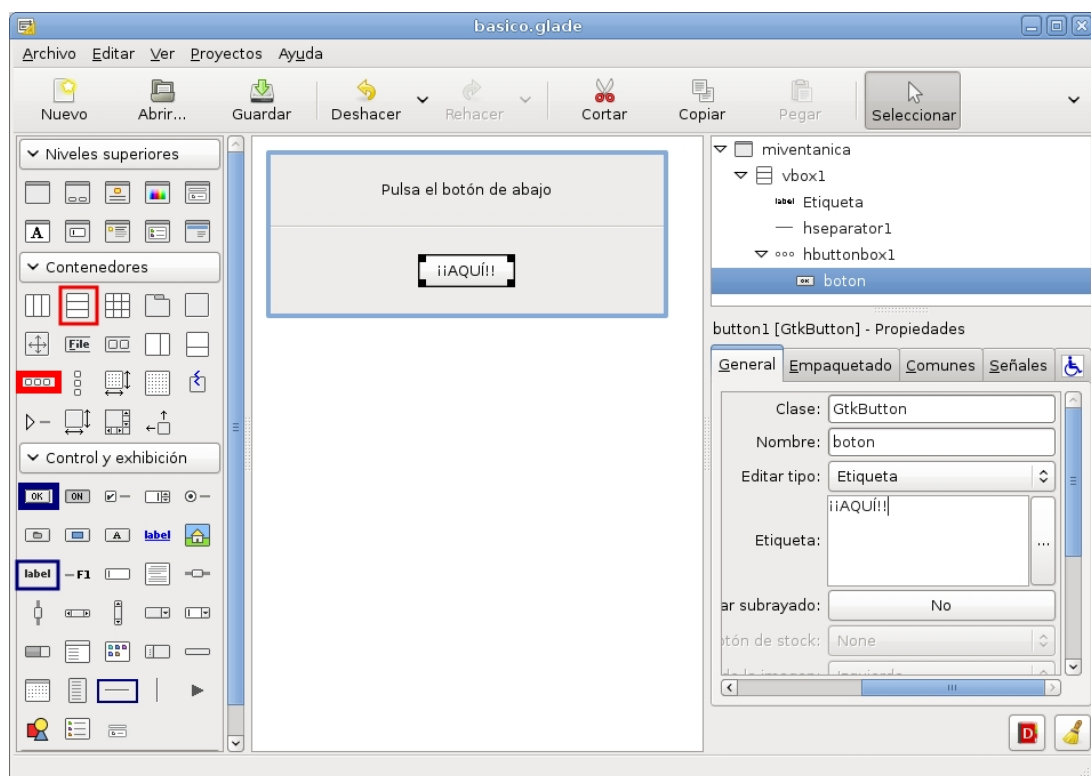


Figura 3.4: Interfaz Glade.

Todas las ventanas auxiliares a la del editor gráfico principal han sido diseñadas utilizando Glade.

3.3. libGnomeCanvas

Un canvas es un lienzo o superficie en la que se puede pintar. Por lo normal puede generar gráficos 2D (líneas, puntos, polígonos...), juegos, animaciones y composición de imágenes.

Debido a que Gtk no dispone de un canvas, se ha recurrido a una biblioteca externa para poder insertarlo en una aplicación GTK+. LibGnomeCanvas es un motor de gráficos estructurados y una de las bibliotecas esenciales de GNOME.

Un canvas se simplifica, al igual que los elementos de Gtk, en un widget. Este widget se puede utilizar para la visualización de los gráficos y es flexible para la creación de elementos interactivos de interfaz de usuario. Los ítems o figuras constan de elementos gráficos como líneas, elipses, polígonos, imágenes, textos y curvas.

Hemos usado este tipo de *widget* para la representación gráfica del autómata.

3.4. GtkTreeView

Un GtkTreeView es un *widget* que nos permite representar información en forma jerárquica (o de árbol) separándola por niveles. Para poder representar la información este widget hace uso a su vez de un GtkTreeModel (un modelo de datos con el que se puede especificar el modo en el que los datos se representarán) y de un GtkTreeStore (necesario para guardar los datos propiamente dichos).

Al igual que el canvas, el GtkTreeView puede manejar eventos, siendo esta funcionalidad muy útil de cara a las necesidades de la aplicación.

Debido a la inclusión de la jerarquía hemos recurrido al uso de este tipo de *widget* para representar toda la estructura del autómata creado pudiendo tener así no sólo la representación gráfica del canvas sino un árbol completo con toda la información accesible en una columna vertical.

3.5. libXml (o Gnome-XML)

XML es un formato estándar avalado por el W3C (World Wide Web Consortium), que permite estructurar la información lógicamente. En contra de otros lenguajes de "marcación", como HTML, por ejemplo, XML no especifica la forma en la que la información debe ser mostrada, sino que sólo se ocupa de estructurarla de una forma lógica.

Esta característica (almacenamiento de información estructurada), añadida a la independencia de la forma de presentación de la información, hace de XML un formato ideal para almacenar datos en las aplicaciones. A la vez, al ser un estándar reconocido internacionalmente, y, por tanto, soportado en multitud de aplicaciones, es también ideal para la compartición de datos entre aplicaciones, incluidas aplicaciones para distintos sistemas operativos. Por todo esto, siguiendo con su afán de únicamente utilizar estándares reconocidos, XML es la herramienta ideal para el almacenamiento de datos.

Para almacenar el formato intermedio del autómata hemos elegido XML y para crear y leer ficheros XML se ha elegido libXml, una librería que implementa un analizador sintáctico XML que permite fácilmente hacer uso de XML en aplicaciones. Con libXml se puede leer, validar y modificar documentos XML, todo ello de una forma bastante clara y sencilla.

3.6. Gazebo

La robótica es un campo que genera un gran interés, sin embargo, la inversión que hay que hacer para acceder al hardware resulta en ocasiones prohibitiva. La simulación por ordenador es una alternativa válida y de bajo coste. No sólo eso, a pesar de que el coste sea un motivo más que suficiente para interesarse por la simulación por ordenador, hay razones adicionales que motivan a interesarse por los simuladores:

1. *Gestión de recursos*: Un simulador, usado junto con una capa de abstracción al hardware, permite desarrollar software para un robot que existe pero al que no se tiene acceso en un determinado momento. Esto es especialmente útil cuando hay varios desarrolladores y un único robot.
2. *Prototipado*: Mediante la simulación es posible implementar el software de un robot antes de que éste haya sido construido. Esto es útil para realizar cualquier tipo de prueba antes de que el robot se construya.
3. *Reusabilidad*: Dado que el uso de simuladores suele ir de la mano de algún tipo de capa de abstracción, el código generado puede ser compartido con otras personas.
4. *Control del tiempo*: Algunos simuladores permiten parar, o acelerar el tiempo (positiva o negativamente) de la simulación. Esto puede ser útil para inspeccionar el estado de los robots en cierto momento o para hacer pruebas de durabilidad.

Gazebo es un simulador 3D orientado a robótica actualmente mantenido y desarrollado por Willow Garage ¹. Al ser un simulador 3D es posible simular cámaras y todo tipo de terrenos 3D. Permite cargar diferentes tipos de sensores y actuadores y ofrece las primitivas para leer

¹www.gazebosim.org

de los primeros e interactuar con los segundos dentro de un mundo artificial (lasers, sonars, cámaras, motores...). Además permite simulaciones multirobot, adjuntar cámaras, crear mundos virtuales... En nuestro caso ha sido utilizado para simular diferentes entornos de actuación de un robot.

El modelo de robot que se emplea en este proyecto es un Pioneer con pleno realismo y proporciona un mundo en el que probar nuestros algoritmos. Consiste en un robot triciclo que consta de un cuerpo principal en el que van montadas dos cámaras acopladas sobre un cuello mecánico, un sensor láser de 180°, motores para las dos ruedas principales y motores para mover el cuello mecánico. Además todos los motores citados cuentan con encoders como sensores internos.

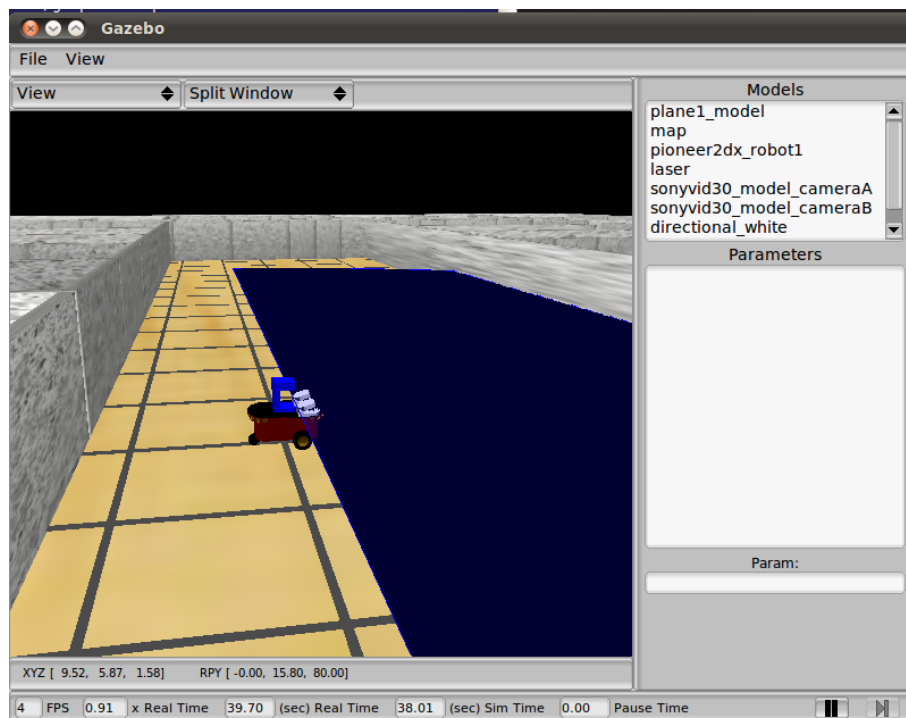


Figura 3.5: Gazebo

3.7. JdeRobot

Este Proyecto Fin de Carrera ha sido desarrollado empleando la plataforma JdeRobot. Ésta es el resultado de la Tesis Doctoral de José María Cañas Plaza y ha sido mantenida y mejorada a lo largo de los años por el Grupo de Robótica de la Universidad Rey Juan Carlos.

La filosofía de esta plataforma es crear una herramienta que proporcione un método sencillo de desarrollar aplicaciones de robótica y visión artificial. Otro de los pilares importantes es la sencillez de reutilización de componentes implementados anteriormente.

JdeRobot esta programado fundamentalmente en C/C++, aunque existen componentes es-

critos en otros lenguajes de programación como pueden ser Python ó Java. Se organiza como un conjunto de componentes comunicándose con el esquema cliente-servidor, donde cada aplicación se ejecuta como un proceso, ofrece una interfaz sencilla para la programación de sistemas de tiempo real y resuelve problemas relacionados con la sincronización de los procesos y la adquisición de datos.

JdeRobot simplifica el acceso a los dispositivos hardware desde el programa, permitiendo obtener el valor de un sensor únicamente leyendo una variable local. Véase el caso de las cámaras que poseen la misma interfaz de imagen para distintas fuentes de vídeo, lo cual proporciona una gran flexibilidad.

JdeRobot hace uso de ICE, un middleware desarrollado por ZeroC orientado a objetos distribuidos bajo una doble licencia GNU GPL y una licencia propietaria. Puede ser utilizado para aplicaciones de Internet sin la necesidad de utilizar los protocolos HTTP y es capaz de atravesar cortafuegos a diferencia de la mayoría de middleware de este tipo. Se trata de una de las características esenciales de la versión actual de JdeRobot 5.0. Es compatible con diferentes lenguajes de programación como C++, Java, Python y la mayoría de sistemas operativos como Windows, MAC OS X, Linux y Solaris. Una variante de ICE es ICE-E que puede ejecutarse dentro de teléfonos móviles. Utilizando ICE se ha diseñado un conjunto de interfaces para interactuar unos componentes con otros, siendo este el único mecanismo de comunicación entre los componentes.

Las aplicaciones creadas para el este proyecto estarán programadas bajo la pataforma JdeRobot 5.0.

Descripción informática

En este capítulo describimos la solución desarrollada para alcanzar el objetivo planteado en el capítulo 2, empleando las herramientas que se explicaron en el capítulo 3. La idea es describir el proyecto en detalle y definir cada una de las partes en las que se basa. Primero damos una idea general de cómo y por qué se ha estructurado así la aplicación. Después explicamos en detalle cada una de las partes en las que se ha dividido.

4.1. Diseño General

Para resolver el objetivo planteado, y buscando una forma fácil para que el programador pueda configurar y programar robots, se han tenido en cuenta tres razones:

1. A la hora de realizar códigos y programas, resulta más fácil ver el diseño del comportamiento del robot de forma visual y organizada que recurrir a la representación textual de todo el código.
2. El comportamiento de un robot se puede simplificar mediante estados. En cada estado realizará una acción determinada, la cual se repetirá iterativamente hasta que algo en el entorno cambie. Estos cambios de estados se denominan transiciones. Una transición puede darse o bien porque se satisface una condición o bien porque se ha cumplido un límite de tiempo establecido por el programador, siendo en este caso transiciones temporales.
3. Un estado puede desplegar, mediante jerarquía, acciones más concretas recogidas en forma de subautómata hijo. Gracias a esto los *hijos* pueden materializar otra serie de acciones encaminadas a enriquecer el comportamiento de su estado *padre*.

Por estas razones se ha decidido realizar una herramienta que represente de forma visual y simplificada el comportamiento del robot y genere automáticamente una aplicación robótica en forma de componente¹ de *Jderobot*. En resumen, la idea del proyecto se puede representar como una caja negra, en la que se introduce unos datos y un diseño de un autómata finito determinista (jerárquico o no) y devuelve un código autogenerado en varios ficheros *.cpp* con todo el componente resultante que se podrá ejecutar en la plataforma *Jderobot* (Figura 4.1).

¹A partir de la versión 5.0 a los esquemas se les llama componentes



Figura 4.1: Entradas y salidas de la aplicación.

Esta herramienta se divide en dos partes: el editor gráfico (con su diseño y sus componentes visuales requeridos) y un generador automático de código. El diseño del editor gráfico se basa en una ventana con diferentes botones de actuación, un espacio en el que se representa el nivel dentro del que nos encontramos en el autómata finito y una vista en árbol que nos muestra la información de todo el autómata jerárquico. En el espacio principal, denominado *canvas*, se pueden pintar dibujos, figuras, imágenes, etc... En nuestro caso se usa para representar los estados (en forma de círculos) y las transiciones (en forma de flechas). Además, mediante la inclusión de menus contextuales, podremos editar, renombrar o eliminar los elementos representados en dicho canvas. Los diferentes botones de actuación en el editor se usan para pintar el autómata o navegar entre los distintos niveles jerárquicos. Además, nos permiten editar y definir las características esenciales del componente resultante diferenciando cada nivel de jerarquía pudiendo asignar librerías dependientes, velocidad, variables o funciones auxiliares del nivel en el que nos encontremos. También encontramos botones para funciones generales del proyecto, como son guardarlo o cargarlo y generar y compilar el código a partir de la información que hayamos introducido en el autómata.

Esta GUI es la ventana principal de la aplicación y es donde se encuentra el 100% de la funcionalidad para crear la estructura del autómata, manejando todos los atributos del componente y del proyecto en cuestión. Para la configuración de las características del componente y para programar cada estado y transición se utilizan varias ventanas auxiliares.

Una funcionalidad que ofrece el editor gráfico es guardar en el disco un fichero XML con todas las propiedades del proyecto: la estructura del autómata, todos sus niveles, las características de cada nodo y transición, las propiedades del componente resultante, etc... Esto es importante ya que además de guardar el componente para su posterior modificación o enviarlo a otro equipo, es necesario que esté previamente guardado para poder generar el componente resultante en lenguaje textual. Por otro lado, al guardar la información en un fichero XML, permite cargar los componentes en versiones posteriores de la herramienta.

El generador de código nos genera el código resultante en *C++* de forma automática. Éste nos crea un código *.cpp* específico compatible con la plataforma *Jderobot 5.0* a partir del fichero XML. Este código tiene funciones y variables propias de *Jderobot* y funciones y variables es-

pecíficas del componente creado. El fichero `.cpp` se podrá compilar generando así un fichero ejecutable listo para usarse en la plataforma *Jderobot*.

Se ha decidido dividir el proyecto en estas dos partes ya que son las más significativas y generales. A su vez la parte de la interfaz se ha dividido en varios módulos que permiten mayor legibilidad. Dentro del editor gráfico se puede distinguir dos partes: la propia interfaz del editor y las estructuras internas. Las estructuras internas permiten dar funcionalidad a la interfaz y guardar toda la información relacionada con el autómata y el componente. Dentro del generador de código también hay dos partes diferenciadas: el generador de ficheros que a su vez se apoya en el fichero xml generado por el editor y en una plantilla de *Jderobot*, y el compilador. Todas estas partes más específicas se comentarán en las siguientes secciones.

El desarrollo completo del proyecto está en <http://jde.gsy.com/index.php/Rubensb-pfc-itis> y el código fuente está disponible en la web <https://svn.jde.gsy.com/users/rubensb/pfc-itis/trunk/visualHFSM/>.

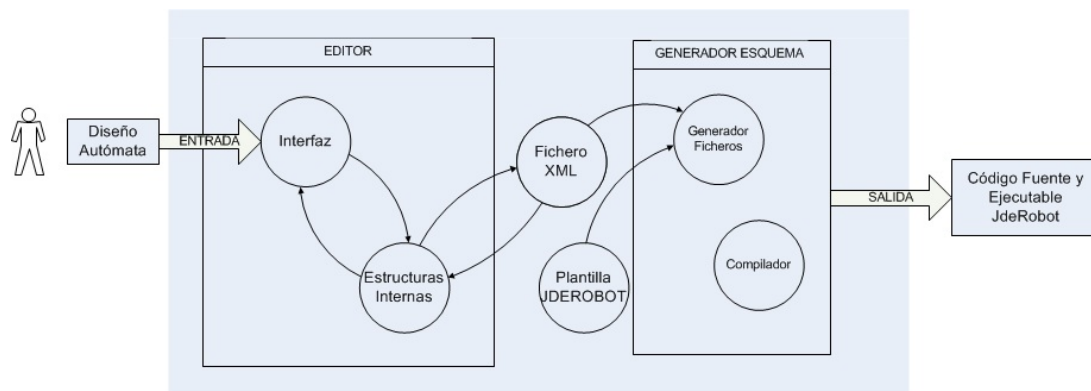


Figura 4.2: Esquema detallado de los elementos de la aplicación.

4.2. Editor Gráfico

El principal objetivo del editor es proporcionar un entorno visual sencillo para permitir la comunicación entre la aplicación y el usuario. El editor permite al usuario representar visualmente el autómata de su robot de forma clara. Se puede manipular estados (crear, borrar, mover, nombrar...) y definir en cada uno de ellos el comportamiento que se quiere realice el robot. También permite establecer las condiciones de permanencia de cada estado, así como las transiciones entre los mismos. Asimismo es posible desplegar niveles inferiores o *subautómatas* haciendo doble click en un estado y creando así comportamientos más complejos.

Para la programación de este editor hemos utilizado la librería *Gtk*. Para posteriores ventanas se utiliza la herramienta de diseño de interfaces *Glade*, programando todos los manejadores asociados a cada señal mediante código.



Figura 4.3: Interfaz gráfica del editor.

4.2.1. Distribución de los elementos gráficos

La interfaz gráfica del editor se divide en tres partes. Empezando por la izquierda podemos ver la vista de árbol, en el centro y como elemento principal tenemos el *canvas* y a la derecha contamos con la botonera. Figura 4.3). En el *canvas* se va desarrollando y dibujando todo el autómata realizado, y mediante los botones de actuación podemos navegar entre niveles, crear más elementos o cambiar propiedades del nivel en el que estemos. Además, también hay botones que se encargan del guardado y la carga del proyecto así como de la generación del código y de su compilación.

En el editor los botones se estructuran en 5 grupos: *Navigation* con un único botón que nos permite subir de nivel dentro del autómata jerárquico, *Figures* que nos permite la creación de nuevos estados y transiciones, *Save/Open* para las opciones de guardado y carga, *Subautomata data* con los botones que nos permiten cambiar las características del nivel en el que estemos, las interfaces ICE que usa, su velocidad de iteración y sus variables/funciones auxiliares y *Code and compile* para poder generar el código y compilarlo.

Por otra parte, la zona del *canvas* es la zona central y más importante de la herramienta, como se puede observar en la captura anterior representa los estados y las transiciones, esta zona es sensible a eventos y gracias a ello podemos editar las características de los estados y transiciones que ahí se muestran accediendo a las opciones mediante menús contextuales.

Por último la vista jerárquica en árbol de la derecha muestra en todo momento la rama del autómata en la que nos encontremos y contiene la información de todo el esquema, también es sensible a eventos de modo que nos permite interactuar expandiendo o contrayendo ramas a voluntad y, mediante un doble click en un nombre, navegamos hasta ese nivel directamente.

El componente está representado en forma de autómata finito en un canvas o lienzo y en una vista jerárquica de árbol. En el primero se puede visualizar cada elemento o estado, y las transiciones que les unen de un nivel en concreto mientras que en el segundo podemos disponer de la información de toda la estructura creada hasta el momento, viendo desplegada únicamente la rama que corresponda al subautómata que vemos en el lienzo. El autómata puede ser dibujado y modificado según el usuario quiera.

4.2.2. Elementos de un subautómata

Transiciones y estados

Un autómata es una colección de lo que podríamos llamar *subautómatas*, están conectados entre sí mediante relaciones de jerarquía padre-hijo. Estos subautómatas están compuestos a su vez por una colección de estados (nodos) y transiciones. Los estados están representados como círculos y su tamaño es constante. Todos los estados tienen el mismo tamaño y serán azules si no cuentan con hijos, es decir, no despliegan niveles más profundos, o verdes en caso contrario. Estos podrán ser nombrados en cualquier momento (opción accesible en menú contextual). El nombre aparecerá sobre el estado correspondiente. Las transiciones están representadas mediante dos líneas consecutivas que unen los dos estados y una caja. La caja está situada entre las dos líneas y nos permite interactuar para modificar la transición. Una de las líneas está representada mediante una flecha para indicar la orientación de la transición (Figura 4.4).

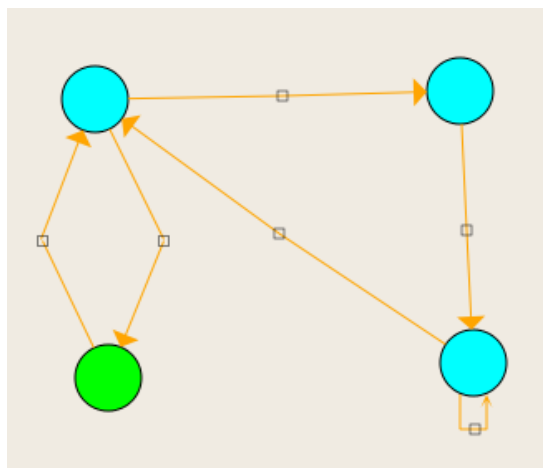


Figura 4.4: Ejemplo estados y transiciones.

El autómata completo se he representado mediante una lista de subautómatas. Como ya habíamos indicado esta formado por estados y transiciones y además por atributos propios. Se ha creado el tipo *tSubAut* que contiene la siguiente estructura:

```
typedef struct tSubAut {
    list <tNodo> ListaElementosSub;    //Lista de estados del subautómata
    list <tTransicion> ListaTransicionesSub; //Lista de transiciones
                                         //del subautómata
    int tiempoIteracionSub; //Velocidad de iteración para el subautómata
    string variablesSub; //Variables auxiliares del nivel
    string funcionesSub; //Funciones auxiliares del nivel
    importar impSub; //Variable para guardar las interfaces ICE
                    //del subautómata
    int idSub; //Identificador del subautómata
    int idPadre; //Identificador de su padre
} tSubAut;
```

Los dos primeros elementos *ListaElementosSub* y *ListaTransicionesSub* son dos listas que contienen, gracias a dos estructuras que serán explicadas más adelante, la información visual y el código de los elementos representados en el lienzo para el subautómata que se esté mostrando. El siguiente elemento, *tiempoIteracionSub*, especifica la velocidad a la que el hilo de ejecución que luego implementará este subautómata iterará. *variablesSub* y *funcionesSub* nos permitirán añadir funcionalidad extra común a todo el subautómata. Con *impSub* podremos saber qué interfaces ICE serán necesarias en ese nivel. Y por último *idSub* y *idPadre* son dos identificadores, el primero el del propio subautómata y el segundo el de su padre.

Dentro de un subautómata cada estado puede estar unido a otro por un número ilimitado de transiciones. Además, un estado puede estar unido a si mismo mediante una autotransición. Las condiciones de permanencia o transición pueden ser temporales, pasado un determinado tiempo, o condicionales, cuando suceda un cambio tanto interno como externo en el robot.

En cada subautómata hay que definir el estado inicial en el cual empezará el comportamiento del robot para ese nivel. Para ello sólo hay que seleccionar la opción "*Mark as initial*" en el menú contextual que se despliega al hacer click con el botón derecho sobre un estado. El estado de inicio está resaltado con un círculo interno al estado.

Cada nodo y transición tendrá unas características determinadas que se almacenarán en una lista. Para ello se han creado dos tipos: *TipoNodo* y *TipoTransición*. Esas estructuras incluyen

información lógica del propio subautómata e información gráfica para su visualización y edición.

El *TipoNodo* tiene la siguiente estructura:

```
typedef struct nodo {
    GnomeCanvasItem * item; // Item Nodo
    GnomeCanvasItem * estado_inicial; // Item círculo interno.
                                //Por defecto NULL.
    string nombre; // Nombre de la figura. "" por defecto
    string codigo; // Código del estado
    GnomeCanvasItem * item_nombre; // Item que muestra el nombre del estado
    list <GnomeCanvasItem *> listaAdyacentes; // Lista de transiciones
                                //adyacentes
    int idHijo; //Id. del subautómata hijo de este estado
} nodo;
```

En ella hay que destacar las variables encargadas de la representación del estado en el *canvas*. Estas son: *item*, *estado_inicial* e *item_nombre*. La variable *item* representa el estado en sí, mientras que las variables *estado_inicial* e *item_nombre* representan características propias a los estados, en este caso si son estados iniciales y su nombre. La variable *item*, es un puntero del item creado, la cual se utiliza para obtener y modificar sus propiedades. Las variables *nombre* y *codigo* tienen la finalidad de guardar el nombre y la programación del nodo (código a ejecutar en cada iteración del subautómata). La variable *listaAdyacentes* se encarga de almacenar todas las transiciones conectadas a él. y por último la variable *idHijo* representa la identificación, dentro de la lista de subautómatas que componen el autómata completo, del hijo asociado a ese estado.

El otro tipo de estructura que se ha creado es el *TipoTransición*:

```
typedef struct transicion {
    GnomeCanvasItem * item; // item transición
    GnomeCanvasItem * origen; // Nodo origen
    GnomeCanvasItem * destino; // Nodo destino
    string nombre; // Nombre de la transición. "" por defecto
    GnomeCanvasItem * item_nombre; // Item del nombre.
    string codigo; // Código de la transición
    int tiempo; // Tiempo de la transición
} transicion;
```

La variable *item* contiene el objeto *item* de la transición creada (al igual que en los nodos). Ésta a su vez está formada por tres items, dos líneas y una caja, agrupadas en una variable de tipo *GnomeCanvasGroup*. Las variables *origen* y *destino* contienen los estados o nodos que conectan. También tiene dos variables que hacen relación al nombre de la transición: *nombre* e *item_nombre*. Y por último tenemos dos variables correspondientes a la condición de permanencia: *codigo* y *tiempo*. *codigo* lleva la programación de la transición en caso de que ésta dependa de un cambio externo, y *tiempo* indica el tiempo de permanencia en la transición en caso de que esta sea temporal. Si la transición depende de un cambio externo, entonces por defecto *tiempo* tiene el valor -1.

En ambos tipos no se ha guardado la posición de los elementos en el canvas. Esto no es necesario ya que la variable *item* contiene todas las propiedades del elemento representado en el canvas.

Coordenadas del canvas

En un canvas se manejan varios sistemas de coordenadas. El primero es un sistema lógico y abstracto de coordenadas en números reales de coma flotante en el espacio llamado coordenadas del mundo. Cuando el canvas comienza a *renderizar* hacia un pantalla se usa el sistema de coordenadas por pixel del canvas (también conocido como coordenadas del canvas). El usuario puede definir el factor de escala y de desplazamiento, los cuales son usados para convertir entre el mundo de coordenadas hacia el sistema de coordenadas de canvas.

Además, cada item en el canvas tiene su propio sistema de coordenadas llamado coordenadas de elemento. Este sistema se especifica en las coordenadas del mundo, pero son relativas a un elemento (0,0, 0,0 sería la esquina superior izquierda del elemento).

Por último, tenemos el sistema de coordenadas de la ventana. Estas son como las coordenadas del canvas pero están desplazadas dentro de la ventana donde el canvas será mostrado. Estas coordenadas son útiles cuando se manejan eventos de GDK como arrastrar y soltar elementos dentro del canvas.

Para recuperar las coordenadas en las que se presionó en el canvas, se usa la función *coordenadas*. La función *coordenadas* es llamada cada vez que el botón izquierdo del ratón es presionado (señal *button_press_event*). Esta función se caracteriza por tener un argumento más en la función de respuesta: *GdkEvent *event*. Este argumento contiene, además de otras características, las coordenadas de la ventana en donde se pulsó el botón. Esto se guardará y se usará posteriormente para dibujar las figuras en el canvas.


```

void coordenadas(GtkWindow *window, GdkEvent *event, gpointer data)
{
    origenX = event->button.x;
    origenY = event->button.y;
    printf("Click Press %d, %d\n", origenX, origenY);

    //Con esta distinción podemos mostrar un menú contextual que nos
    //permitirá pegar un estado que haya sido copiado previamente.
    if (event->button.button == 3){
        gtk_menu_popup(GTK_MENU(menu_pegar),NULL,NULL,NULL,NULL,
            event->button.button,event->button.time);
    }
}

```

Pintado de nodos

La función *pinta_nodo* es la responsable de dibujar los nodos del componente. Esta función es llamada posteriormente a la función *coordenadas* (señal *button_release_event*), y comprobará que el botón “Estado” se presionó previamente para dibujar en el canvas un estado en las coordenadas guardadas anteriormente. Comprobará además que el botón del ratón presionado fue el izquierdo.

Se creará una nueva estructura de *TipoNodo* y se conectará la función correspondiente a la señal “event” emitida por el item (objeto “estado”) recién creado, para cuando éste sufre algún evento darle funcionalidad correspondiente. Esto permite su posterior edición. Además, debido a que esta función crea un elemento nuevo, al final se realiza la gestión necesaria para que ese cambio quede reflejado en la vista en árbol del panel izquierdo.

```

void pinta_nodo(GtkWindow *window, GdkEventMotion *event, gpointer data)
{
    int x, y;
    GnomeCanvasItem *item;
    GnomeCanvasGroup *group;

    list<tSubAut>::iterator posSub; //Añadido para poder iterar en la lista
    //de subautomatas y actualizar el treeview con cada nuevo estado

```

```
x = event->x;
y = event->y;

points = gnome_canvas_points_new(2); /* 2 puntos */
points->coords[0] = origenX;
points->coords[1] = origenY;
points->coords[2] = x;
points->coords[3] = y;

printf("Click %d, %d\n", x, y);

if ((strcmp(botonPulsado.c_str(), "Cuadrado")==0) and
    (event->state & GDK_BUTTON1_MASK))
{

    group = GNOME_CANVAS_GROUP (gnome_canvas_item_new (root,
                                                         gnome_canvas_group_get_type (),
                                                         "x", 0,
                                                         "y", 0,
                                                         NULL));

    item = gnome_canvas_item_new(group,
                                  gnome_canvas_ellipse_get_type(),
                                  "x1", (double) origenX-20,
                                  "y1", (double) origenY-20,
                                  "x2", (double) (origenX + 20),
                                  "y2", (double) (origenY + 20),
                                  "fill_color_rgba", 0x00ffffff,
                                  "outline_color", "black",
                                  "width_units", 1.0,
                                  NULL);

    registro.item = item;
    registro.item_nombre = NULL;
    registro.estado_inicial = NULL;
    registro.nombre = "";
    registro.codigo = "";
    registro.idHijo = 0; // Por defecto un estado no tiene hijos
```

```
    id ++;

    ListaElementos.push_back(registro);

    /*Añadimos control (señal) al item creado*/
    g_signal_connect (group, "event",
                      (GtkSignalFunc) item_event,
                      NULL);

    //Actualizamos datos necesarios para mostrar cambios en el tree view
    posSub = ListaSubAutomatas.begin();

    while(posSub->idSub != subautomata_mostrado){
        posSub++;
    }

    //guardamos el estado de la lista de elementos en el nodo del
        //subautomata para poder recargar bien el tree view
    posSub->ListaElementosSub = ListaElementos;

    actualizar_tree_view();

    gtk_tree_view_expand_to_path(GTK_TREE_VIEW(tree_view), arbolPath);
}
}
```

Pintado y Orientación de las transiciones

Para dibujar las transiciones, se han tenido en cuenta dos cosas: los puntos de origen y destino, y la orientación. Al ser los estados un círculo, la transición será la recta que pase por el punto medio de ambos estados. Para que no haya ninguna transición que atraviese el círculo de un estado, se calcula la intersección de dicha recta con las circunferencias. La transición resultante será la línea que une ambos puntos resultantes. Posteriormente se podrá mover la transición para que no se solape con otra.

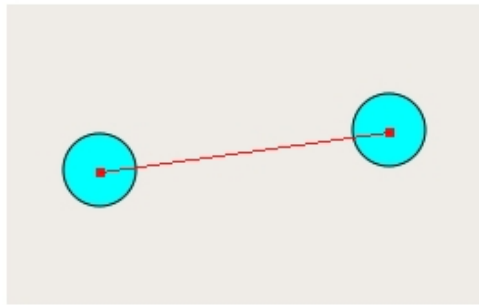
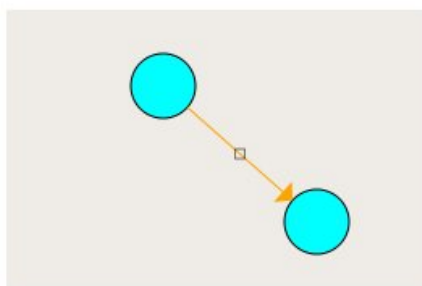


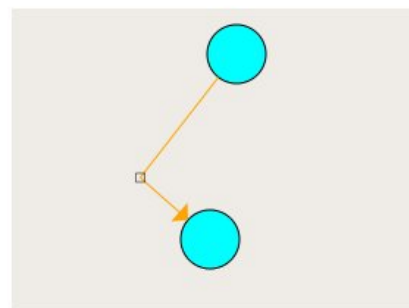
Figura 4.5: Recta que une las circunferencias.

Mover estados y transiciones

A la hora de mover cada estado hace falta tener en cuenta los estados conectados a él. Para no perder ninguna conexión, cada estado tendrá asociado una lista de estados adyacentes. En primer lugar se moverá el estado afectado y después de realizará el pintado de todas las transiciones conectadas a él. Para mover los estados sólo hace falta cambiar sus correspondientes valores $x1$, $y1$, $x2$ y $y2$. Para repintar las transiciones hay que buscar dentro de la lista de adyacentes correspondientes al nodo todas las transiciones conectadas, se borrará parte de la transición y se volverá a pintar (Figura 4.6).



(a) Antes



(b) Después

Figura 4.6: Mover Nodos.

En el caso de mover transiciones, se haría lo mismo que en los nodos. En este caso se movería la caja de la transición y las líneas se volverían a calcular y repintar (Figura 4.7).

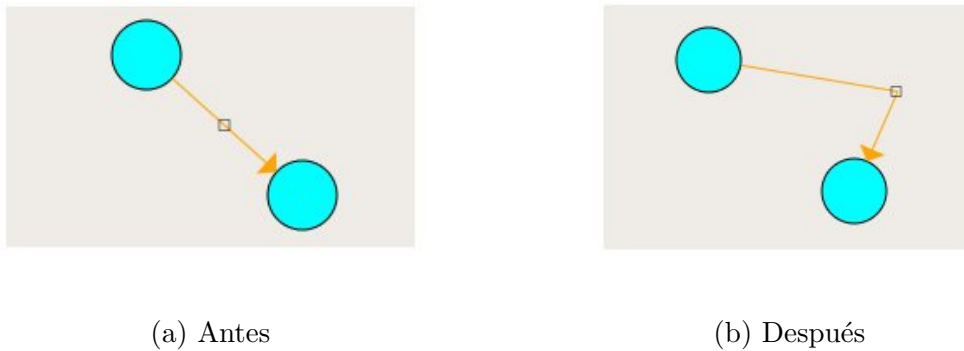


Figura 4.7: Mover Transiciones.

Estos movimientos se pueden realizar simplemente pinchando y arrastrando el elemento que se quiera mover.

Zoom

Una función que tiene el editor es la de ampliar y reducir la zona de creación del autómata o canvas. Esto es útil a la hora de crear componentes con gran número de estados, ya que permite tener una visión más general durante el desarrollo del autómata. Para ampliar los elementos se pulsará en el teclado la flecha hacia arriba y para reducirlo la flecha hacia abajo.

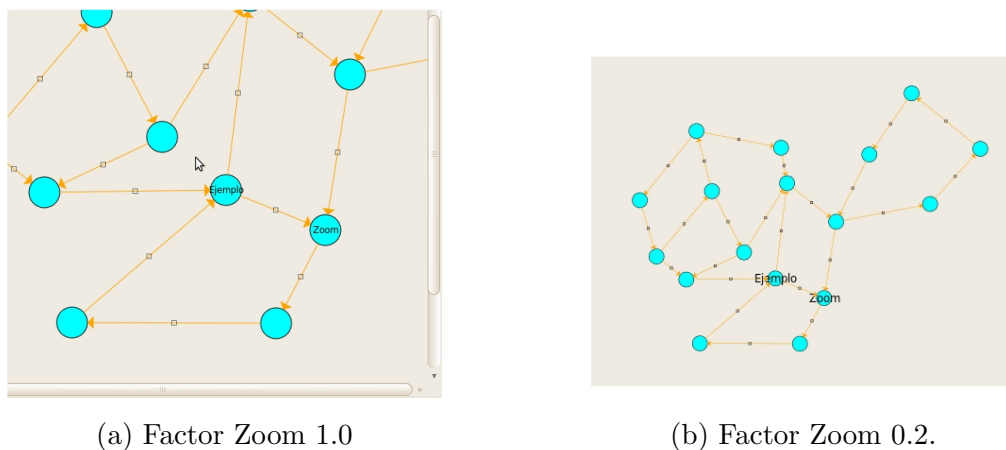


Figura 4.8: Zoom.

Menús contextuales

Con esta nueva versión hemos añadido otra característica para hacer que la aplicación sea más usable, esta característica es la inclusión de menús contextuales. Anteriormente todas las opciones que se ofrecían eran accesibles a través del botón correspondiente en el panel derecho de la aplicación. Ahora este acceso se ha trasladado a una serie de menús desplegables mediante click con el botón derecho sobre los elementos directamente en el canvas, esta acción es mucho más

intuitiva y común en el uso de diversas aplicaciones y acerca la herramienta a los usuarios finales.

Son 3 los menús contextuales que se han implementado, el *menu_estado*, el *menu_transicion* y el *menu_pegar*. Como podemos deducir por su nombre el primero se despliega cuando hacemos click derecho sobre un estado, el segundo cuando lo hacemos sobre una transición y el tercero cuando el click es en una zona libre del canvas.

El *menu_estado* nos permite acceder a las siguientes funcionalidades:

- *Rename*: Esta opción es la que nos permitirá poner nombre al estado.
- *Edit*: Seleccionado esta opción podremos editar el código del estado para dotarle de funcionalidad.
- *Mark as Initial*: Marcamos ese estado como el inicial para el nivel en el que nos encontremos.
- *Copy*: Copiamos ese estado al portapapeles para luego permitir su pegado en ese u otro nivel.
- *Delete*: Borramos ese estado y todo lo que ese estado pudiese desplegar, es decir, se “poda” el árbol jerárquico a partir de ese estado.

El *menu_transicion* nos permite acceder a las siguientes funcionalidades:

- *Rename*: Ponemos nombre a la transición.
- *Edit*: Editamos su condición o su tiempo de espera.
- *Delete*: Borramos la transición.

Por último el *menu_pegar* tan sólo cuenta con la siguiente opción:

- *Paste*: Con esta opción podremos pegar en cualquier parte del autómata un estado previamente copiado. Sólo tendrá efecto si efectivamente se ha realizado esa copia y además creará una copia de todo lo que el estado original pudiese desplegar. De modo que si copiamos un estado que tuviese un hijo que a su vez desplegase dos “nietos” cuando pegamos en cualquier parte del esquema se nos generará no sólo el estado sino también su hijo y sus dos nietos. Esto nos permite duplicar comportamientos en distintas zonas del autómata teniéndolos que programar sólo una vez.

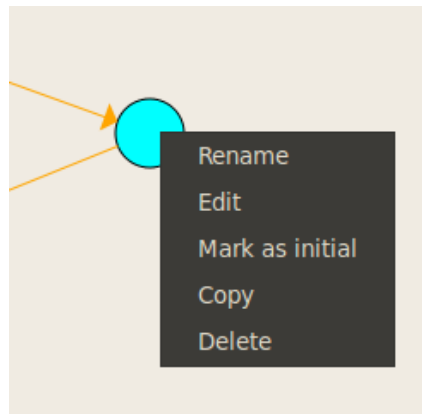


Figura 4.9: Menu contextual de un estado.

4.2.3. Edición y programación explícita de un subautómata

Otra funcionalidad que nos permite la interfaz principal es la de modificar las atributos auxiliares esenciales del proyecto y de cada uno de los subautómatas. Estas características son: establecer el estado inicial del subautómata, establecer la velocidad de iteraciones por segundo de cada hilo de la aplicación en la ejecución con *JdeRobot* (velocidad de iteración de cada nivel o subautómata), asignar un nombre a los estados, insertar el código de cada estado, definir las condiciones de permanencia o de transición de cada estado, definir las variables dependientes y auxiliares de cada nivel y definir el nombre y ubicación del componente. Para ello ha sido necesario la creación de ventanas auxiliares y menús contextuales para acceder a gran parte de ellas.

Debido a la necesidad de utilizar ventanas auxiliares para incorporar el comportamiento del robot y otras características, se ha utilizado el diseñador de interfaces Glade (Ver 3.2), ya que simplifica la programación de una ventana gráfica. Esta forma de generar interfaces o ventanas resulta mucho más rápida y fácil que la programación de una interfaz directamente en código, sin embargo, toda la funcionalidad de cada objeto gráfico ha sido incorporada mediante código. De esta forma tenemos el diseño en un fichero XML realizado con Glade y sus correspondientes señales y *callbacks* (funciones de respuesta) que se conectan mediante código para realizar las correspondientes tareas. Estas funciones son las que se ejecutan cada vez que sucede un evento.

Estado inicial

El estado inicial define dónde se empieza a ejecutar el código implementado en el autómata, es decir, en el comienzo de la ejecución, el estado en el que se encuentra el autómata. Para marcar el estado inicial tanto del subautómata raíz (el que será el estado inicial 'total' del autómata completo) como del resto de subautómatas o niveles sólo es necesario seleccionar la opción "Mark as Initial" del menú contextual que se despliega al hacer click con el botón derecho sobre un estado.

El estado seleccionado está representado con un círculo interno para diferenciar que ese es el estado de inicio.

Velocidad del subautómata

La velocidad del componente define el tiempo de iteración del nivel activo, es decir, cada cuánto tiempo se ejecutará la iteración principal del hilo que se creará para ese nivel en la plataforma *JdeRobot*. Está formado por un *GtkVBox*, que contiene un *GtkHButtonBox* con los botones Aceptar y Cancelar, y un *GtkFrame*. Dentro del *frame* hay un *GtkLabel* con la etiqueta “Iteration Time” y *GtkEntry* donde se escribirá el tiempo de ciclo en milisegundos (Figura 4.10).

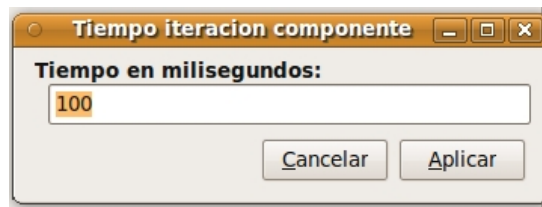


Figura 4.10: Ventana edición del tiempo.

Interfaces sensoriales y de actuación

El botón de ‘Interfaces’ permite definir qué interfaces de sensores y actuadores son propias del subautómata que se generará para *JdeRobot*. Se tienen que cargar en la ejecución del mismo con el autómata porque depende de ellos para funcionar. Cada subautómata lleva asociadas sus propias interfaces ya que cada nivel puede hacer uso sólo de unas cuantas de todas las disponibles. Está formado por un *GtkVBox* que contiene dos *frames* y un *GtkHButtonBox*. El widget *GtkHButtonBox* contiene los botones de aceptar y cancelar. Los *frames* contienen dichas variables, uno correspondiente a los sensores y otro correspondiente a los actuadores del robot de referencia. Cada *frame* contiene una etiqueta (*GtkLabel*) con el nombre del tipo de la variable y varios *GtkCheckButton*. Estos *GtkCheckButton* son botones de selección o activación, es decir, permiten elegir una o varias opciones dentro de los sensores y actuadores disponibles en el robot de referencia (Figura 4.11).

Esto permite la activación y desactivación en el robot de referencia de sensores y actuadores y su uso desde el autómata. Los sensores que tenemos son el Laser, las Camaras, los Encoders y los PtEncoders (encoders del cuello mecánico), y los actuadores son el Motor (de la base) y el PtMotor (motores del cuello mecánico). Estos sensores los proporciona *Jderobot* el cual se encarga de cargar únicamente los interfaces indicados como necesarios e inicializarlos al inicio de la ejecución (ver 3.6).

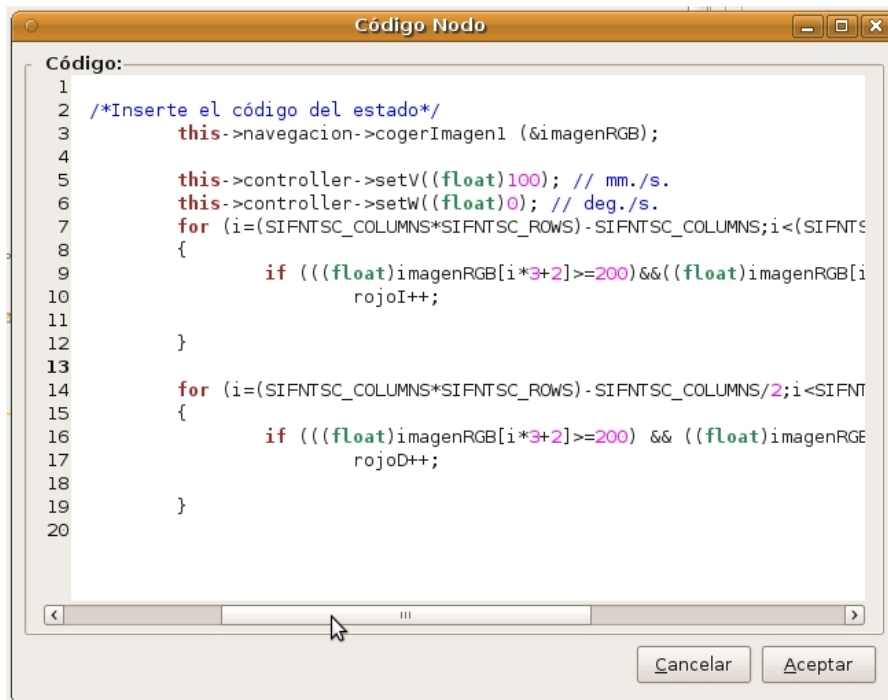


Figura 4.11: Ventana de selección de interfaces sensoriales y de actuación.

Programación explícita de estados y transiciones

El código del estado es el que se ejecuta en cada iteración de la aplicación si el autómata se encuentra en ese estado. Es el correspondiente a las acciones del robot. De cada estado pueden salir varias transiciones y a su vez dicho estado puede “validar” el despliegue de su hijo. El código de dichas transiciones se ejecuta en cada estado correspondiente y en cada iteración del subautómata, y es el que verifica si hay que cambiar de estado. Además, en cada estado se realizarán las operaciones de percepción necesarias, tanto para condiciones de permanencia o transición como para tomar acciones concretas.

La inserción del código en lenguaje de texto asociado a estados y transiciones se realiza a través de dos ventanas auxiliares. La primera permite insertar código en los estados y está compuesta por un *GtkSourceView* que está dentro de un *GtkScrolledWindow*. El *scroll* permite que la zona de texto se pueda ampliar. En este caso se ha decidido usar un *GtkSourceView* en vez de un *GtkTextView* ya que además de permitir visualizar y editar el texto, posee las funciones de *undo/redo* y permite cambiar el color del texto en función del lenguaje usado (Figura 4.12).



```

Código:
1
2 /*Inserte el código del estado*/
3     this->navegacion->cogerImagen1 (&imagenRGB);
4
5     this->controller->setV((float)100); // mm./s.
6     this->controller->setW((float)0); // deg./s.
7     for (i=(SIFNTSC_COLUMNS*SIFNTSC_ROWS)-SIFNTSC_COLUMNS;i<(SIFNTS
8     {
9         if (((float)imagenRGB[i*3+2]>=200)&&((float)imagenRGB[i
10            rojoI++;
11
12     }
13
14     for (i=(SIFNTSC_COLUMNS*SIFNTSC_ROWS)-SIFNTSC_COLUMNS/2;i<SIFNT
15     {
16         if (((float)imagenRGB[i*3+2]>=200) && ((float)imagenRGE
17            rojoD++;
18
19     }
20
  
```

Figura 4.12: Ventana de inserción de código en los nodos.

La otra ventana permite insertar código en las transiciones. Está compuesta por dos *GtkRadioButton*. Estos son botones de elección, los cuales permiten elegir entre dos o más opciones. En nuestro caso permiten elegir entre insertar texto, en caso de que sea una transición condicional, o insertar tiempo, en caso de que sea una transición temporal. En ambos casos se mostrará debajo un *GtkEntry* (Figura 4.13). Aquí no se ha usado un *GtkSourceView*, ya que sólo hay que definir la condición que debe cumplir, el resto de cálculos perceptivos hay que hacerlos en el estado correspondiente.



(a) Transición temporal.

Figura 4.13: Ventana edición de transiciones.

Declaración de variables y funciones auxiliares

Todas las variables utilizadas en el componente deberán ser declaradas e inicializadas desde el editor. Estas variables podrán ser variables dependientes o variables auxiliares que se necesiten para realizar cálculos. Además se podrán añadir todas las funciones auxiliares que sean nece-

sarias. Tanto las variables como las funciones auxiliares son las que el usuario necesite definir, escribiendo su código explícito para poder usarlas en los estados o las transiciones.

Está formado por un *GtkNoteBook* con dos pestañas, una para las variables y otra para las funciones. Cada pestaña está compuesta por un *GtkScrolledWindow* y un *GtkSourceView* al igual que en la programación de los estados (Figura 4.14). Cada nivel tiene asociadas sus propias variables y funciones auxiliares.

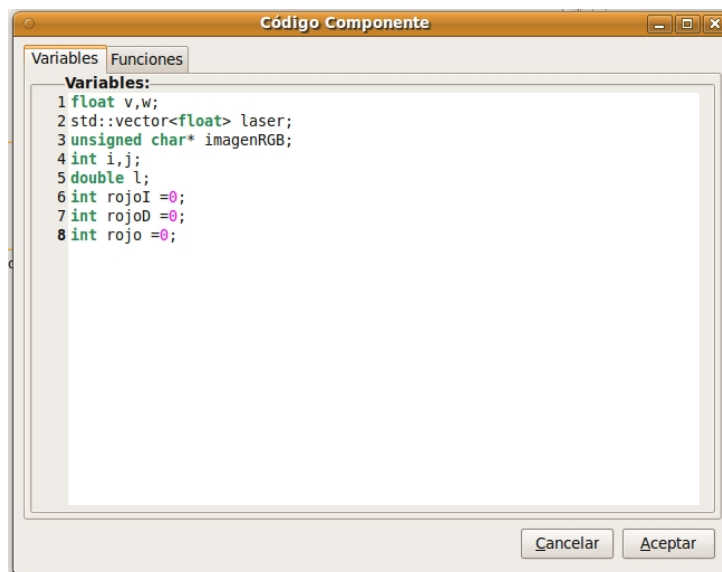


Figura 4.14: Ventana de inserción de variables y funciones auxiliares.

4.2.4. Jerarquía de autómatas

Comportamiento e interacción con la vista en árbol

Al haber incorporado la posibilidad de crear autómatas jerárquicos se hacía necesario poder tener un mapa global del proyecto que fuera visible en todo momento. Esto lo hemos conseguido con la inclusión de una columna vertical a la izquierda de la interfaz que nos muestra una vista en árbol de todos los estados y niveles que tengamos en nuestro autómata.

Esta vista es interactiva, es decir, según nosotros naveguemos entre los niveles (bien hacia abajo, haciendo doble click en un estado para desplegar su hijo, bien hacia arriba, mediante el botón “ÜP” de la botonera derecha) la información en ella mostrada reacciona a esa navegación contrayéndose o expandiéndose únicamente la rama del árbol en la que nos encontremos en ese momento. Gracias a esta funcionalidad podemos saber en todo momento la parte del autómata en la que nos encontramos, siendo obviamente más útil cuanto más grande sea el autómata con el que estemos trabajando.

Además se ha incorporado otra funcionalidad que es la de poder navegar a una parte del

autómata haciendo doble click directamente en la vista. La vista permite interactuar directamente al usuario, se puede expandir o contraer manualmente las ramas que se quieran. De este modo podemos buscar un nivel concreto y si lo deseamos navegar directamente a él haciendo doble click en cualquiera de sus estados.

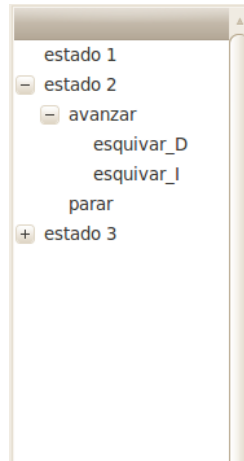


Figura 4.15: Vista de árbol jerárquica.

Extensión de las estructuras internas.

En todo momento se ha tenido muy presente el aprovechar lo máximo posible las estructuras existentes. Sin embargo ha sido necesario realizar una extensión de estas para que fuesen capaces de implementar jerarquía.

Un cambio realizado para conseguir esto ha sido enriquecer la información guardada por un estado con un *idHijo*. Este identificador representa el subautómata dentro de la lista de subautómatas que conforman el componente completo que es hijo de ese estado. De este modo podemos establecer una relación hacia abajo padre-hijo.

También hemos creado la estructura de lista de subautómatas. Cada nodo de esta lista contiene básicamente la misma información que antes (*en VICOE*) se utilizaba para el autómata mononivel. Pero se ha enriquecido con un *idSub*, es decir, un identificador para cada subautómata y con un *idPadre* que es otro identificador de subautómata, en este caso, el de su padre. Gracias a este identificador podemos establecer la relación en sentido contrario hijo-padre.

Extensión de los atributos auxiliares.

Debido a que ahora tenemos varios subautómatas tenemos que poder asociar a cada uno sus propios atributos auxiliares. Al haber creado una lista de subautómatas este trabajo sólo ha consistido en incluir en cada nodo de esa lista la información necesaria. En nuestro caso esta información consiste en tiempo de iteración, variables/funciones auxiliares e interfaces ICE

utilizados por el subautómata.

4.3. Fichero intermedio

En esta parte se explica cómo se guarda el autómata desarrollado parcialmente para su futura modificación. Se explica además la forma del fichero resultante. Para guardar los datos hemos elegido el formato XML.

4.3.1. Sintáxis del fichero de guardado y carga

El formato XML tiene la ventaja de permitirnos estructurar cualquier tipo de información de forma lógica. Gracias a esto podemos estructurar por niveles los datos necesarios para recoger toda la información contenida en el componente que se esté creando.

Hemos creado un primer nivel que contiene los subautómatas y la dirección y nombre del componente generado. Dentro de cada subautómata contamos con su identificador y el de su padre, los estados que lo componen, su tiempo de iteración, variables y funciones auxiliares e interfaces que usa.

Cada estado se expande a su vez recogiendo si es o no el inicial en ese subautómata, su identificador, su posición en el lienzo, nombre, identificador de su hijo, código y las transiciones asociadas.

Por último el nivel más interno dentro del fichero lo componen las transiciones. Estas se expanden para albergar su posición en el lienzo, su nombre, código o tiempo y el identificador de su estado destino.

4.3.2. Guardado y carga del autómata

En el archivo *XML* se guardan todas las características de los elementos dibujados en el canvas (coordenadas de los puntos, nombre de los estados, conexiones entre ellos, etc), las características de cada subautómata como el estado inicial, las librerías utilizadas, el tiempo de iteración, etc así como las características propias del componente (nombre del componente, directorio de guardado).

Cada subautómata tiene un identificador único. Gracias a esto podemos, a la hora de generar el código, establecer las relaciones padre-hijo necesarias para poder implementar el comportamiento jerárquico ya que, como hemos ido explicando ligeramente hasta ahora, cada subautómata será implementado como un hilo y este hilo se “activará” en función del estado de su

padre. En cada subautómata se guarda, además de sus estados y transiciones, su identificador, los interfaces ICE de los que hace uso, las variables auxiliares y las funciones auxiliares.

Del mismo modo, dentro de cada subautómata, cada estado tiene asociado un identificador único. Esto permite identificarlos y establecer las relaciones de dependencia entre ellos. De cada estado se guarda las siguientes características: coordenadas gráficas de origen y destino, nombre, código (correspondiente a las acciones realizadas) y transiciones salientes. De cada una de estas transiciones se guarda las siguientes características: un identificador de destino (estado al que se conecta), coordenadas de la situación de la caja en el canvas, nombre y código (correspondiente a la condición de permanencia o transición) o tiempo de iteración.

A la hora de cargar un proyecto, primero se borra el proyecto actual y posteriormente se carga el proyecto seleccionado. Se utilizan los identificadores establecidos para dibujar las transiciones. A la hora de dibujar en el canvas el componente guardado hay que tener en cuenta que hasta que no se le pasa el control a la aplicación, ésta no dibuja ninguna figura. Este problema se puede solventar mediante una función que permite actualizar el canvas en el momento deseado. Debido al uso de jerarquía tenemos que ser capaces de visualizar sólo el nivel adecuado, esto se resuelve de forma fácil con la creación de un par de funciones `mostrar_subautomata(idSub)` y `ocultar_subautomata(idSub)`. Como se puede deducir estas funciones muestran y ocultan el subautómata requerido, dado que cada nivel tiene asociados sus items en el canvas sólo tenemos que mostrarlos u ocultarlos centrándonos en aplicar esos cambios en los items del subautómata indicado.

Este archivo *XML* será utilizado posteriormente también para generar el código fuente resultante del componente (*.cpp*). Por lo tanto es necesario guardar el proyecto antes de compilarlo.

La estructura del fichero resultante es parecida a la siguiente:

```
<VisualHFSM>
  <Subautomata>
    <idSub>1</idSub>
    <idPadre>0</idPadre>
    <Estado estado_inicial="true">
      <id>1</id>
      <origenX>329</origenX>
      <origenY>178</origenY>
      <destinoX>369</destinoX>
      <destinoY>208</destinoY>
```

```
<nombre>sigue</nombre>
<hijo>2</hijo>
<codigo>
/*Código del estado*/
</codigo>
<transiciones>
  <transicion>
    <box x1="331" y1="242" x2="337" y2="248"/>
    <nombre>HayObstaculo</nombre>
    <destino>2</destino>
    <codigo>/*Código transición*/</codigo>
  </transicion>
</transiciones>
</Estado>
<Estado estado_inicial="false">
  <id>2</id>
  <origenX>223</origenX>
  <origenY>291</origenY>
  <destinoX>263</destinoX>
  <destinoY>311</destinoY>
  <nombre>para</nombre>
  <hijo>0</hijo>
  <codigo>
/*Código del estado*/
</codigo>
<transiciones>
  <transicion>
    <box x1="383" y1="322" x2="389" y2="328"/>
    <nombre></nombre>
    <destino>3</destino>
    <codigo>/*Código transición*/</codigo>
  </transicion>
</transiciones>
</Estado>
<Estado estado_inicial="false">
  <id>3</id>
  <origenX>556</origenX>
  <origenY>290</origenY>
```

```

    <destinoX>586</destinoX>
    <destinoY>310</destinoY>
    <nombre>corrige</nombre>
    <hijo>0</hijo>
    <codigo>
    /*Código del estado*/
    </codigo>
    <transiciones>
        <transicion>
            <box x1="458" y1="261" x2="464" y2="267"/>
            <nombre>Giro Completado</nombre>
            <destino>1</destino>
            <tiempo>80000</tiempo>
        </transicion>
    </transiciones>
</Estado>
<tiempoIteracion>800000</tiempoIteracion>
<Librerias>
    <lib>motor</lib>
    <lib>camara</lib>
</Librerias>
<variables_aux>int i;</variables_aux>
<funciones_aux/>
</Subautomata>
<Subautomata>
    <idSub>2</idSub>
    <idPadre>1</idPadre>
    <Estado estado_inicial="true">
        .
        .
        .
    </Estado>
    <Estado estado_inicial="false">
        .
        .
        .
    </Estado>
<tiempoIteracion>400000</tiempoIteracion>

```



```
<Librerias>
  <lib>laser</lib>
  <lib>motor</lib>
  <lib>camara</lib>
</Librerias>
<variables_aux>int j;</variables_aux>
<funciones_aux/>
</Subautomata>
<nombreEsquema>/dir/nombre</nombreEsquema>

</VisualHFSM>
```

Nombrado y ubicación del componente

La forma de guardar o cargar el proyecto se realiza a través de un *GtkFileChooserDialog*. Éste *widget* es un cuadro de dialogo específico de selección de archivos, indicado para guardar ficheros o abrir ficheros (Figura 4.16). Esto permite además nombrar y ubicar el componente.

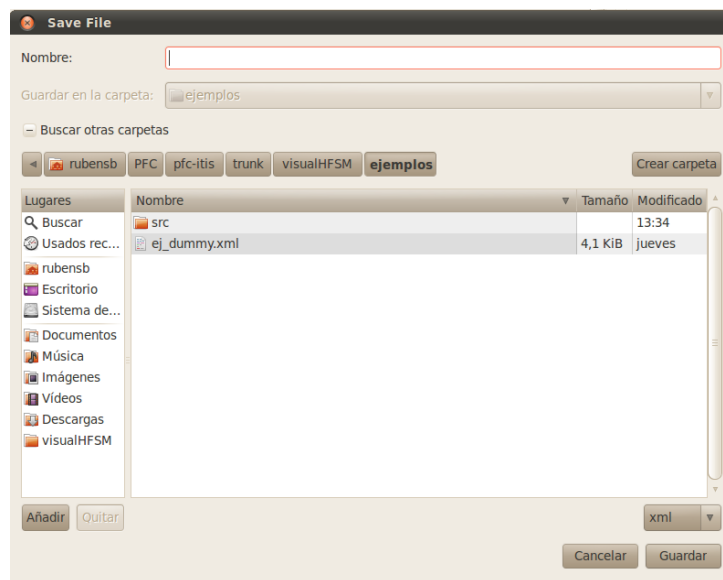


Figura 4.16: Ventana de Guardado.

4.4. Generador Automático de código

Esta parte es la encargada de generar y compilar el código en *C++* resultante, de materializar el autómata diseñado visualmente generando un componente compatible con *JdeRobot 5.0*. Se explicará el proceso de construcción del componente y las partes de las que está compuesto.

4.4.1. Información sobre las plantillas utilizadas

Hemos visto que la generación del código ejecutable para su uso en *JdeRobot 5.0* requiere de una plantilla. A continuación veremos los pasos que hemos ido dando hasta obtener la plantilla definitiva.

Empezamos tomando como esqueleto principal el nuevo *BasicComponent* desarrollado para *JdeRobot*. Este componente implementa de forma totalmente independiente mediante distintos hilos de ejecución un hilo gráfico y un hilo de control. Ambos iteran constantemente a una velocidad concreta ejecutando repetidamente el código programado en ellos.

Para adaptar este esquema a nuestras necesidades empezamos editando el hilo de control. Creamos una estructura capaz de recoger el comportamiento de un autómata mononivel. Esta estructura consiste en un doble *switch* y un motor temporal. El primero de los *switch* es el de evaluación de transiciones. Al principio de cada iteración lo que hacemos es comprobar las condiciones de permanencia en el estado en el que nos encontremos y transitar al siguiente si fuese necesario. El segundo *switch* es el de actuación, aquí ejecutamos el código del estado que corresponda. Gracias a esta estructura evitamos iteraciones vacías ya que conseguimos realizar en un mismo bloque evaluación, transición y acción.

El paso para adaptar esta plantilla mononivel para jerarquía fue el de extender el único hilo de control original a tantos hilos de control como subautómatas tengamos. En el caso de subautómatas hijos su comportamiento se verá limitado a que su padre esté en el estado correcto. El resto es igual a si fueran el subautómata raíz; *switch* de evaluación y *switch* de actuación.

4.4.2. Rellenado de la plantilla

La forma de construir un componente consiste en usar una plantilla de la aplicación *Jderobot* que se va rellenando con la información concreta del autómata que está siendo generado. Esta plantilla tiene dos partes, la parte de control y la parte gráfica del componente. La parte de control está compuesta por una plantilla con el esqueleto genérico de un autómata finito determinista en *C++*, el cual se irá completando con la información y el código específico de cada estado y transición definidos en el editor gráfico (Figura 4.17). Mientras que la parte gráfica se usa para visualizar en tiempo de ejecución los estados activos mediante texto a través de la

consola.

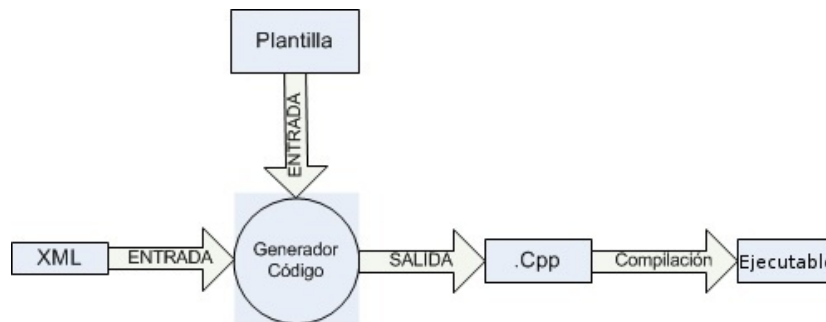


Figura 4.17: Resumen Generador Código.

Para ello se ha tenido como referencia el nuevo componente básico que ha sido desarrollado paralelamente a este proyecto por Maikel González Baile ². Este nuevo componente básico tiene la particularidad de separar mediante distintos hilos de ejecución la parte de control de la parte gráfica siendo muy útil para nuestros propósitos dado que en nuestro caso editamos casi en exclusiva la parte de control para poder implementar el comportamiento recogido por el autómata.

En nuestro caso prescindimos de la interfaz gráfica que este componente básico posee y la sustituimos por un hilo gráfico que escribe texto en la consola.

Para poder implementar correctamente el comportamiento de nuestro autómata es necesario editar, dentro de todos los archivos que componen el componente básico, tres ficheros que son, *control_class.cpp*, *control_class.h* y *mycomponent.cfg*. En los dos primeros cargaremos la información necesaria para todos los hilos de ejecución (que son un hilo para el gui y un hilo de control por cada nivel distinto o subautómata que haya en nuestro esquema) y en el tercero cargaremos la configuración para los interfaces ICE que nuestro componente necesite.

Antes de realizar esta edición el proceso de generación del código comienza copiando todos los archivos del componente básico en la carpeta `../src` y, como ya hemos explicado anteriormente, es necesario guardar el esquema del autómata antes de poder generar el código.

Edición de los ficheros necesarios

A continuación vamos a explicar los cambios que hay que realizar sobre los archivos base que hemos copiado del componente para adaptarlos a nuestras necesidades.

²<http://jde.gsync.es/index.php/Mikel-pfc-itis>

Edición de control_class.h

El primer fichero que editamos para incorporar información generada por nuestro esquema es el control_class.h . Esta primera edición es muy sencilla y consiste únicamente en definir las variables de tipo *pthread_t* que serán los distintos hilos de ejecución. Se define una variable por cada nivel del autómata, así si tenemos por ejemplo un autómata con sólo dos niveles, es decir, la raíz tiene X estados y sólo uno de ellos despliega un hijo, en este fichero añadimos en su zona de declaración de variables lo siguiente:

```
pthread_t thr_sub_1;
pthread_t thr_sub_2;
```

Edición de control_class.cpp

El siguiente fichero a editar es el control_class.cpp, esta es la parte sin duda más importante del generador de código ya que es aquí donde necesitamos crear la estructura multihilo que dará vida a nuestro autómata.

Lo primero que tenemos que declarar para poder implementar nuestro autómata jerárquico son las estructuras de memoria que guardan el estado en el que se encuentra cada nivel. En este punto diferenciamos entre dos tipos de estructuras, la creada para el nivel raíz y la creada para el resto. En el caso del nivel raíz se crea una estructura muy simple que sin más contiene todos los estados posibles de ese nivel.

```
typedef enum Estado_Sub_1 {
    estado1,
    estado2
}Estado_Sub_1;
```

Pero en el caso de los niveles inferiores esta estructura difiere ligeramente incorporando lo que hemos llamado “estados fantasma”. Estos estados fantasma son estados que no se corresponden con ninguna acción y nos son útiles para recuperar el punto en el que un nivel se estaba ejecutando cuando su padre haya dejado de estar en el estado que le permite ejecutarse y tiempo después vuelve a ese estado “llave”. De este modo obtenemos algo como esto:

```
typedef enum Estado_Sub_2 {
    estadoA,
    estadoA_ghost,
```

```

    estadoB,
    estadoB_ghost
}Estado_Sub_2;

```

Obsérvese que en nuestra herramienta hemos pintado un nivel con únicamente dos estados, estadoA y estadoB, pero a la hora de implementar el código añadimos sus correspondientes estados fantasma para poder controlar la ejecución jerárquica.

A continuación declaramos las variables que guardarán los estados de cada nivel inicializándolas a sus correspondientes estados iniciales marcados en la herramienta.

```

Estado_Sub_1 Sub_1 = estado1;
Estado_Sub_2 Sub_2 = estadoA;

```

Seguidamente pasamos a crear la estructura capaz de recoger el correcto comportamiento que queramos conseguir. Para ello ha sido necesario desarrollar una plantilla jerárquica básica que, como ya hemos explicado, mediante hilos de ejecución consiga nuestro propósito. Esta plantilla básica aglutina todos los elementos que un subautómata necesita para funcionar, esto es, zona de decisión (para poder elegir si saltar o no a un estado nuevo), zona de acción (donde se ejecutarán las acciones programadas para el estado activo) y motor temporal (para controlar la velocidad de iteración del hilo). Dentro de estas tres partes las que realmente permiten crear el comportamiento jerárquico son las dos primeras. Así cada hilo queda finalmente compuesto siguiendo este esqueleto:

```

void *motor_th1(){

    variables básicas
    variables auxiliares //Las que puede introducir el usuario
                        //desde la herramienta en cada nivel

    //El siguiente while hace las veces de iterationControl
    //agrupa comportamiento y motor temporal
    while(true){

        //Switch de evaluación de transiciones
        switch (Sub_1){
            case estado1:
            {

```

```
        //Se evalúan las condiciones y si se cumplen
        //se salta al estado correspondiente
    }
    case estado2:
    {
        //Se evalúan las condiciones y si se cumplen
        //se salta al estado correspondiente
    }
}

//Switch de actuación
switch (Sub_1){
    case estado1:
    {
        //Se ejecuta el código programado para
        //este estado
    }
    case estado2:
    {
        //Se ejecuta el código programado para
        //este estado
    }
}

//Motor temporal del hilo
//Aquí hacemos uso del tiempo de iteración indicado
//para este nivel por el usuario.

}
}
```

Gracias al uso de este esqueleto que se ejecuta de manera iterativa podemos aprovechar cada iteración sin tener ninguna vacía ya que lo primero que hace el hilo es evaluar las condiciones de salto pudiendo ejecutar inmediatamente después el código correspondiente se haya o no se haya saltado según la evaluación.

En el caso de encontrarnos con subautómatas hijos que dependen del estado de su padre para poder ejecutarse el esqueleto anterior sufre una ligera modificación para recoger el comportamiento jerárquico de manera correcta. Si suponemos que el subautómata 2 es hijo del

subautómata 1 tenemos:

```
void *motor_th2(){

    variables básicas
    variables auxiliares //Las que puede introducir el usuario
                        //desde la herramienta en cada nivel

    //El siguiente while hace las veces de iterationControl
    //agrupa comportamiento y motor temporal
    while(true){

        if(Sub_1==estado2){ //Suponemos que es el estado2
                            //el que despliega esta rama.
            if(Sub_2==estadoA_ghost || Sub_2==estadoB_ghost){
                Sub_2=(Estado_Sub_2)(Sub_2-1);
            }

            //Switch de evaluación de transiciones

            //Switch de actuación

        }else{
            if(Sub_2==estadoA || Sub_2==estadoB){
                Sub_2=(Estado_Sub_2)(Sub_2+1);
            }
        }

        //Motor temporal del hilo
        //Aquí hacemos uso del tiempo de iteración indicado
        //para este nivel por el usuario.

    }
}
```

Como vemos se han introducido varios elementos que pasamos a explicar. El primero de ellos es una condición *if(thr_sub_1==estado2)* que limita toda la ejecución del hijo a que su padre se encuentre en el estado correcto. La siguiente condición hace uso de los estados fantasma que explicamos con anterioridad recuperando el punto de ejecución que el nivel tuviese cuando se

le retiró la "llave". Hemos decidido implementar este comportamiento en vez de la alternativa de volver al estado inicial del nivel cada vez que su padre lo active porque es posible que ese nivel se hubiese quedado a medias en cualquier acción y con este método permitimos recuperar la ejecución en el punto correcto para continuar con su tarea. El último punto a explicar es la rama del *else* que simplemente coloca al hilo en un estado fantasma guardando así su punto de ejecución para cuando su padre lo vuelva a activar.

En ambas variantes del esqueleto observamos como se han incluido las variables auxiliares introducidas por el usuario para ese nivel, en el caso de las funciones auxiliares estas se introducen justo antes de escribir las funciones `motor.th*` explicadas arriba para que así puedan ser utilizadas en los esqueletos.

Con esto hemos explicado las mayores modificaciones que se han de realizar para implementar el comportamiento jerárquico pero nos queda realizar otras dos tareas antes de terminar con la edición de `control.class.cpp`. La primera de ellas es la creación del hilo "visual" que mostrará por consola los estados activos, es decir, el estado del nivel raíz y, por orden de profundidad, el resto de estados activos diferenciándolos de los inactivos por no llevar la terminación "_ghost". La segunda es la inclusión del código necesario para la importación de los interfaces ICE usados por el autómata, para ello se recopilan todos los interfaces marcados por el usuario en todos los niveles y se añade el código correspondiente únicamente de los marcados.

Edición de `mycomponent.cfg`

La última tarea del generador de código es editar el fichero de configuración del componente que se va a crear para añadir las líneas necesarias referentes a los interfaces ICE que el autómata vaya a utilizar.

Dado que como hemos visto anteriormente ya hemos realizado un barrido de los interfaces necesarios en la edición del `control.class.cpp` ya sabemos qué interfaces necesitamos y con esa información editamos el fichero `mycomponent.cfg`. Por ejemplo si un autómata programado por el usuario sólo hiciese uso del láser, las cámaras y los motores este fichero sólo contendría lo siguiente:

```
Mycomponent.Motors.Proxy=motors1:tcp -h localhost -p 9999
Mycomponent.Camera1.Proxy=cameraA:tcp -h localhost -p 9999
Mycomponent.Camera2.Proxy=cameraB:tcp -h localhost -p 9999
Mycomponent.Laser.Proxy=laser1:tcp -h localhost -p 9999
```


Variables

En los componentes existen dos tipos de variables. Las variables propias del componente o variables compartidas, y las variables auxiliares. Estas últimas variables se deben definir e inicializar desde la herramienta.

Las variables compartidas son las que están disponibles para todas las partes del componente, concretan los sensores y actuadores del robot de referencia. Vienen resueltas en nuestro componente al utilizar las interfaces estandar ICE para esos sensores y actuadores que son materializadas por otros componentes (por ejemplo *GazeboServer*). Definen el comportamiento de los mecanismos de actuación y percepción del robot. Para incorporar y utilizar estas variables sólo hace falta seleccionar a través de la ventana que aparece al pulsar el botón *Interfaces* las que queramos usar. La herramienta las declara e incluye automáticamente una vez seleccionadas visualmente. Son variables accesibles en todo momento a cualquier componente, ya sea para escritura si se trata de actuadores (Tabla 4.2) o para lectura si se trata de sensores(Tabla 4.1).

Nombre		Recurso
ld		Láser - vector de X distancias (mm.) vertidos por el láser.
ed		Encoders - Array que contiene la posición del robot de acuerdo al sistema de referencia inicial.
data1		Cámara - vector correspondiente a la imagen izquierda. Contiene una instantánea de la cámara A.
data2		Cámara - vector correspondiente a la imagen derecha. Contiene una instantánea de la cámara B.
pted1, pted2	longitude	PTEncoders - Ángulo medido en pan (deg.) del cuello mecánico.
	latitude	PTEncoders - Ángulo medido en tilt (deg.) del cuello mecánico.

Cuadro 4.1: Sensores del robot de referencia

Nombre		Recurso
mprx	v	Motores - Velocidad lineal (mm./s.).
	w	Motores - Velocidad rotacional (deg./s.).
ptmprx1, ptmprx2	longitude	PTMotor - Ángulo en pan (deg.) a comandar al cuello mecánico.
	latitude	PTMotor - Ángulo en tilt (deg.) a comandar al cuello mecánico.

Cuadro 4.2: Actuadores del robot de referencia

Las variables auxiliares son propias del usuario. Por ejemplo son variables para cálculos intermedios o de ayuda. No están relacionadas con actuadores ni sensores. Se podrá incorporar mediante una ventana auxiliar de texto del editor gráfico, donde habrá que declararlas e inicializarlas.

Resultados experimentales

En este capítulo vamos a exponer como hemos sido capaces de comprobar el correcto funcionamiento de la herramienta creada a través de un sencillo ejemplo a modo de prueba de concepto que nos mostrará la capacidad de ésta para generar comportamientos jerárquicos. Además mostraremos algunas ideas que fueron descartadas a lo largo del desarrollo del proyecto.

5.1. Prueba de concepto

Con el siguiente ejemplo de funcionamiento vamos a ver como se usa la herramienta para generar un comportamiento jerárquico. Hemos implementado un autómata tremendamente sencillo para demostrar esto.

El autómata se compone de 3 niveles, el raíz tiene dos estados, *ChocaGira* y *Vueltas*, la transición entre ambos es temporal y será cíclica en intervalos de 30 segundos. Cada uno de estos estados despliega a su vez un hijo.

En el caso de *ChocaGira* su hijo consta de 4 estados que implementan un comportamiento de choca-gira, estos son *avanzando*, *parando*, *marcha_atras* y *girando*.

En el caso de *Vueltas* su hijo tiene otros dos estados, *aIzquierda* y *aDerecha*, esta rama sólo alterna entre dos sentidos de giro del robot y nos permitirá ver como se ha dejado de ejecutar el comportamiento de choca-gira para pasar a ejecutarse el de las vueltas.

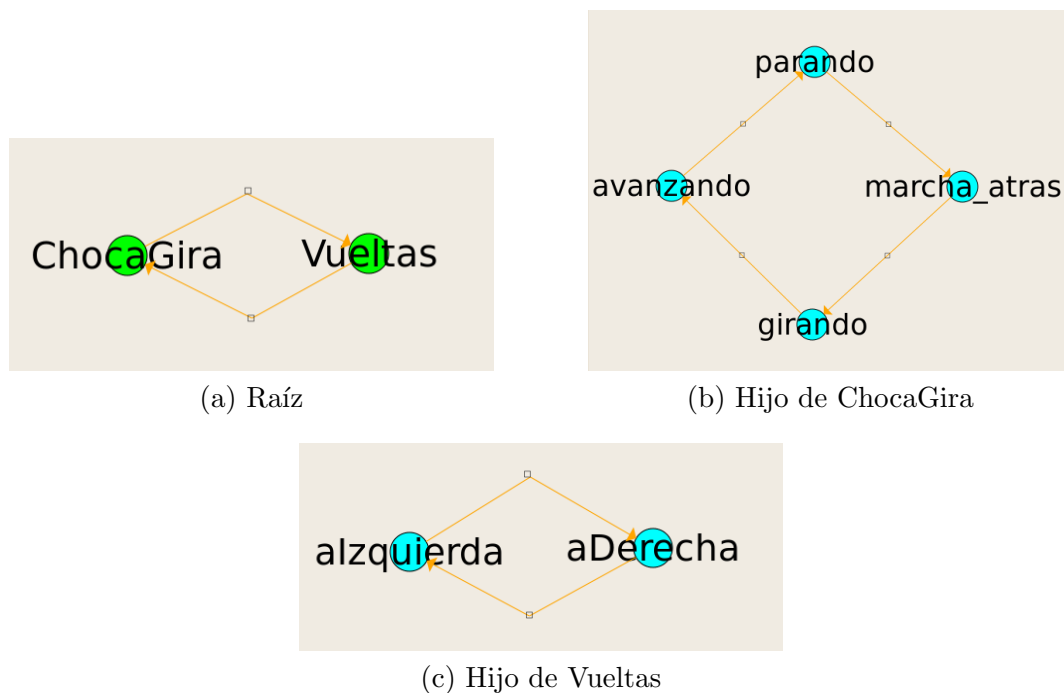


Figura 5.1: Prueba de concepto

Al ejecutar el componente generado podemos observar como el hilo de *GUI* nos muestra por consola los estados activos.

```

Archivo Editar Ver Terminal Solapas Ayuda
rubensb@rubensb-lucid: ~
ChocaGira
Avanzando

```

Figura 5.2: Visualización de estados en tiempo de ejecución

5.2. Alternativas descartadas

Vamos a explicar algunas ideas que fueron descartadas a lo largo del desarrollo del proyecto porque no cumplían los requisitos necesarios para cubrir la funcionalidad deseada.

El aspecto del proyecto que ha ido sufriendo más cambios a medida que se avanzaba en su desarrollo ha sido sin duda la creación de la plantilla. Se fue pasando por distintas soluciones que aunque en un primer momento parecían cumplir con lo deseado tenían una serie de inconvenientes que nos hicieron decidir cambiarlas.


```

        estado_actual = estadoM;
        estado_cambiado = true;
    }
    if(!estado_cambiado){
        /*Código del estado*/
        estado_cambiado = false;
    }
    break;
}
}while(estado_cambiado = true);

```

Segunda plantilla mononivel

Para solucionar el problema de las iteraciones vacías creamos una segunda plantilla mononivel, añadimos junto al cambio de estado una variable *booleana*, prioridad, que si estaba activa ejecutaba el código del estado antes de evaluar las transiciones. Se activaba en la iteración anterior al haber evaluado las transiciones. Con esto conseguimos evitar las transiciones vacías.

Sin embargo surgió otro problema, con este método podíamos ejecutar la acción de un estado en un momento en el que las condiciones podían ya no satisfacerse. Al haber dado la prioridad en la iteración anterior al ejecutar el código de un estado teniendo en cuenta sólo a esta variable podría ocurrir que en esa iteración las condiciones ya no se cumpliesen y en realidad no debiésemos haber ejecutado esas acciones.

```

do{
    switch(estado_actual){
        case estado1:
            /*Percepcion propia del estado para poder actuar*/
            if (prioridad){
                prioridad = false;
                /*Código del estado*/
            }else{
                if (/*Condición de la transición 1*/){
                    estado_actual = estado2; //se cambia al estado que corresponda
                    estado_cambiado = true;
                }
                if (/*Condición de la transición 2*/){ /*si añadimos "&& !estado_cambiado"
                    damos prioridad a las primeras transiciones,

```

```

                                si no lo añadimos damos prioridad a las últimas
                                en caso de que varias puedan solaparse*/
    estado_actual = estado3; //se cambia al estado que corresponda
    estado_cambiado = true;
    prioridad = true;
}
if (!estado_cambiado){ //no se ha cumplido ninguna condicion de transición
    /*Código del estado*/
    estado_cambiado = false;
}
}
break;
.
.
.
case estadoN:
    /*Medicion del tiempo*/
    if(){
    }else
        if(/*Tiempo transcurrido = Tiempo marcado*/){ //para los casos de
                                                    //transiciones temporales

            estado_actual = estadoM;
            estado_cambiado = true;
            prioridad = true;
        }
        if(!estado_cambiado){
            /*Código del estado*/
            estado_cambiado = false;
        }
    }
    break;
}
}while(estado_cambiado = true);

```

Finalmente subsanamos todos estos problemas con la versión definitiva con doble *switch*, evaluación y actuación.

Capítulo 6

Conclusiones y trabajos futuros

Tras exponer la solución desarrollada para los objetivos fijados y presentada la prueba de funcionamiento con el ejemplo de concepto expuesto en el capítulo anterior vamos a evaluar los resultados obtenidos y a exponer posibles líneas de trabajo futuras que permitan mejorar la herramienta.

6.1. Conclusiones

El objetivo principal de este proyecto era mejorar la versión anterior de la herramienta existente hasta ese momento (*VICOE, David Yunta, 2011*) incluyendo como principal característica la posibilidad de desarrollar autómatas jerárquicos. También se ha pretendido dotar a la herramienta de la mayor usabilidad posible para facilitar su manejo e incluso poder utilizarla en el ámbito docente. Además se quería que el código generado hiciese uso de la última plantilla básica para aplicaciones robóticas ofrecida por JdeRobot, *BasicComponent*.

Dado que este proyecto se ha basado fuertemente en la versión anterior de la herramienta una de las primeras tareas fue la de familiarizarse con VICOE, entender su interfaz, funcionalidad y desarrollo para tener claros los puntos reutilizables y los que era necesario cambiar. En todo momento se ha desarrollado intentando aprovechar al máximo el entorno existente de modo que la programación de los puntos comunes entre las dos versiones restase el menor tiempo posible a las tareas de mejora de la aplicación. Buen ejemplo de esto es el modo con el que se ha resuelto la jerarquía, hemos creado una lista de subautómatas que básicamente está formada por nodos que contienen casi la misma información que VICOE generaba para un único nivel, permitiéndonos de esta manera manejar cada nivel con los procedimientos existentes hasta entonces como si de un autómata mononivel se tratase y dejando que nos centrásemos en los aspectos de 'enlace' entre cada nodo de la lista de subautómatas para tener así una estructura jerárquica.

Para acercar al usuario a la aplicación también se ha trabajado activamente en la interacción de éste con la interaz gráfica de la herramienta. Para ello se ha simplificado la botonera de la versión anterior, trasladando la mayor parte de su funcionalidad a menús contextuales asociados a los elementos pintados en el lienzo. Estos menús ofrecen una forma muy común de interactuar con cualquier aplicación informática de modo que ahora la edición del esquema

resulta más intuitiva. Además, funciones simples como mover los elementos del lienzo ahora se realizan simplemente pinchando y arrastrando y no, como ocurría anteriormente, pinchando un botón concreto y realizando el movimiento para luego tener que pinchar otro botón para hacer otra acción.

Otra mejora que va en la línea de la usabilidad es la inclusión de la vista en árbol a la izquierda de la interfaz. Este nuevo elemento se nos reveló como totalmente necesario debido a la necesidad de tener un mapa global del componente, sin esta vista y con un supuesto componente lo suficientemente extenso es fácil imaginar la posibilidad de que el usuario acabase perdido entre los niveles sin saber a ciencia cierta en que rama se encontraba. Con la vista en árbol hemos conseguido solucionar todos estos problemas y además añadir otra funcionalidad interesante que es la de poder navegar a cualquier punto del esquema sin tener que seguir un orden "lineal" subiendo o bajando nivel a nivel.

Como paso intermedio entre los dos puntos principales (la interfaz gráfica y la generación de código) tuvimos que entender la generación y el manejo de ficheros *XML* para poder usarlos, tal como ocurría en la versión anterior, para guardar los datos del esquema y poder generar el código a partir de él. Aquí adoptamos una solución parecida a la adoptada para las estructuras de memoria usadas por la aplicación, es decir, se añadió un nuevo nivel al fichero *XML* que no era otro que *Subautomata*, teniendo una organización interna prácticamente igual a la de un autómata mononivel guardado por vicoe.

Con todos estos elementos nos centramos en la generación del código final. Para ello hemos usado como plantilla base el nuevo componente básico que se ha creado paralelamente a este proyecto para la plataforma *jderobot* (basic component, Maikel González, 2012). Este componente se encarga de implementar de manera transparente la separación entre hilos visuales e hilos de control de modo que nos sirvió para poder implementar el comportamiento de nuestros autómatas de forma muy clara editando para ello la parte de control. Obviamente y debido a que este nuevo *basic component* sólo proporciona un único hilo de control nosotros tuvimos que introducir mejoras para que soportase la ejecución multihilo que nuestros autómatas jerárquicos requerían pero dado a la nueva arquitectura empleada fue una tarea sencilla al simplemente tener que añadir más hilos de tipo control.

Se han cumplido todos los requisitos marcados, implementando una herramienta donde se representa un autómata de forma visual y simplificada, que ayuda a la comprensión del usuario, que soporta gran número de estados y facilitando el desarrollo de los componentes compatibles con la versión 5.0 de *Jderobot*.

Se puede decir que el método seguido ha ayudado en el aprendizaje de conocimientos. Se

han desarrollando interfaces desde diferentes vías de trabajo, se han empleado diferentes tipos de *widgets* y empaquetados y se ha incorporado estructuras y controles de Gtk+ añadiendo bibliotecas auxiliares. Además, se ha aprendido a utilizar las diferentes señales y funciones de respuesta para dar la funcionalidad deseada a la interfaz. También se ha aprendido conocimientos sobre los ficheros “XML” y su creación y manipulación mediante las bibliotecas.

La programación de la herramienta ha estado más relacionada con la ingeniería de software que con la generación de comportamientos sobre robots, aunque el objetivo de la estructura fuese este. Siempre se han tenido en cuenta los objetivos y la funcionalidad final de la plataforma, pero no se ha desarrollado nada en cuanto a software que mejore el comportamiento o funcionalidades de los robots se refiere, sino que facilite su programación.

El proyecto completo cuenta con unas 8000 líneas de código de las cuales han sido escritas específicamente como mejoras sobre la versión anterior unas 800-1000.

6.2. Trabajos futuros

En este apartado se detallan algunas posibles mejoras que podrán realizarse sobre este proyecto y que pueden servir como nuevas líneas de trabajo para otros proyectos fin de carrera futuros.

Una mejora que se podría incluir es el rediseño completo de la interfaz gráfica de la aplicación para adaptarla a tipos de interfaces más comunes. Esto es, básicamente, la sustitución de la botonera lateral y el traslado de su funcionalidad a una barra superior de menús. Este cambio permitiría ampliar la zona del lienzo que es al fin y al cabo la zona sobre la que gira toda la interfaz.

Otra mejora que se podría incorporar es la posibilidad de copiar y pegar no sólo un único estado (y todos sus hijos) como ocurre ahora sino varios de ellos implementando algún tipo de “buffer de copiado” en el que se puedan añadir varios elementos.

También hay que mencionar que con la inclusión de la jerarquía hemos perdido una característica muy interesante que sí incluía la versión anterior y es la de mostrar un canvas equivalente al programado en tiempo de ejecución del componente generado por la herramienta, nosotros hemos resuelto esta falta indicando mediante consola los estados activos, pero sin duda la utilidad de ese gui automático en tiempo de ejecución es altamente interesante.

Apéndice A

Bibliografía

David Yunta Barro, Herramienta de programación visual de autómatas de estado finito para aplicaciones robóticas., Proyecto Fin de Carrera de Ing. Tec. Informática Sistemas, URJC, 2011

David Lobato, jde+ Una plataforma de desarrollo para aplicaciones robóticas, Proyecto Fin de Carrera de Ing. Informática, URJC, 2005

David Lobato, jderobot 5: Entorno de desarrollo basado en componentes para aplicaciones robóticas, Máster Universitario en Sistemas Telemáticos e Informáticos, URJC, 2010

Carlos Iván Martín Amor, Herramienta de programación visual de autómatas en Jderobot, Proyecto Fin de Carrera de Ing. Tec. Informática Sistemas, URJC, 2010

Ricardo Palacios Maya, Representación rica de la escena 3D alrededor de un robot móvil., Proyecto Fin de Carrera de Ing. Téc. Informática de Sistemas, URJC, 2006

Jesús Ruiz-Ayúcar Vázquez, Jdeneo.c - Una plataforma de desarrollo de aplicaciones robóticas., Proyecto Fin de Carrera de Ing. Téc. Informática de Sistemas, URJC, 2006

José María Cañas Plaza, Programación de robots móviles, Universidad Rey Juan Carlos, 2004

J. M. Cañas and M. A. Cazorla and V. Matellán., Uso de simuladores en docencia de robotica móvil., IEEE Revista Iberoamericana de Tecnologías del Aprendizaje. Volume 4, Number 4, pp 268-277, 2009

GNOME Canvas Library Reference Manual, "<http://developer.gnome.org/libgnomecanvas/2.30/>", 2007

GNOME XML Reference Manual, "<http://xmlsoft.org/>"

GTK Reference Manual, "<http://developer.gnome.org/gtk/>", 2011