XIII Workshop en Agentes Físicos

WAF 2012

Santiago de Compostela

3rd - 4th of September, 2012

Centro Singular de Investigación

en Tecnoloxías da Información (CITIUS)

University of Santiago de Compostela

Coordinators of the edition

Roberto Iglesias Rodríguez

Carlos Vázquez Regueiro

Víctor Álvarez Santos

Adrián Canedo Rodríguez

Coordinators of the edition

Roberto Iglesias Rodríguez

Centro Singular de Investigación en Tecnoloxías da Información (CITIUS).
Universidade de Santiago de Compostela (Spain).
roberto.iglesias.rodriguez@usc.es

Carlos Vázquez Regueiro

Departamento de Electrónica e Sistemas.
Universidade de A Coruña (Spain).
cvazquez@udc.es

Víctor Álvarez Santos

Centro Singular de Investigación en Tecnoloxías da Información (CITIUS).
Universidade de Santiago de Compostela (Spain).
victor.alvarez@usc.es

Adrián Canedo Rodríguez

Centro Singular de Investigación en Tecnoloxías da Información (CITIUS).
Universidade de Santiago de Compostela (Spain).
adrian.canedo@usc.es

ISBN: 978-84-940469-0-2
Legal deposit: C-1566-2012
Prints: Unidixital, s.l.

Programación visual de autómatas para comportamientos en robots

David Yunta, Jose María Cañas

***Abstract*—La utilidad y flexibilidad de un robot están determinadas en gran medida por la potencia de su software. Típicamente el software de los robots se escribe con lenguajes textuales como Python, C, C++ o Java. Para simplificar y acelerar el tiempo de desarrollo de las aplicaciones robóticas existen diferentes lenguajes visuales. Con ellos no se maneja texto, sino bloques gráficos que realizan alguna acción concreta y se pueden combinar. Los autómatas de estado finito son una técnica consolidada para programar la inteligencia de robots. Permiten generar comportamientos variados de manera compacta. En este trabajo se presenta una herramienta visual desarrollada para programar comportamientos en un robot móvil usando autómatas de estado finito. La herramienta se ha usado con éxito para generar varios comportamientos dentro de la plataforma Jderobot.**

***Index Terms*—Lenguajes Gráficos, Autómatas, Programación de Robots**

I. Introducción

En el software de los robots reside la mayor parte de su inteligencia. Una vez fijado el cuerpo físico del robot con sus sensores y actuadores, la funcionalidad reside en su programación, en el software que maneja esos recursos hardware.

No existe ninguna metodología universalmente estandarizada para programar robots. En robótica hay mucha variedad y dinamismo, tanto en el software como en el hardware. Típicamente se programan con lenguajes de texto, tanto de bajo nivel como de alto nivel. Así, es frecuente encontrar brazos robotizados programados en ensamblador y robots móviles programados con lenguajes más abstractos como C++ o Java. Una alternativa que se ha explorado es la utilización de lenguajes visuales, buscando simplificar la creación de aplicaciones robóticas. Una primera ventaja es que representa directamente la estructura de los algoritmos de control y procesamiento [3]. Por ejemplo LEGO creó el lenguaje visual código-RCX para que los niños programaran su robot Robot Invention System. Este lenguaje es completamente visual y permite programar robots mediante la unión de bloques gráficos a modo de instrucciones, comprobación de condiciones, bucles, etc. e incluso el manejo de concurrencia de modo intuitivo. Actualmente LEGO da soporte también a la programación de sus NXT con la plataforma gráfica LabView.

En los últimos años se han empezado a aplicar metodologías de ingeniería del software a la robótica y así han surgido plataformas de programación de robots para simplificar el desarrollo de aplicaciones. Estas plataformas (1) proporcionan un interfaz más o menos portable de acceso al hardware; (2) ofrecen una arquitectura software concreta a las aplicaciones; (3) incluyen bloques de funcionalidad ya resuelta y herramientas. Muchas de las plataformas que han surgido están orientadas a componentes que facilitan la reutilización de software. Por ejemplo plataformas como ROS, Orca, Microsoft Robotics Studio, RoboComp, Jderobot, etc.

Existen muchas maneras de organizar el código de control y percepción a bordo de un robot móvil. Las arquitecturas cognitivas reactiva, deliberativa, híbrida etc. se han plasmado en cierta manera de organizar el software de los robots. Una manera concreta son los autómatas de estado finito (FSM en inglés), que se utilizan con éxito para simbolizar el comportamiento de un robot además de usarse en otros muchos ámbitos. Con ellos el comportamiento viene definido por un conjunto de estados, en cada uno de los cuales se realiza una tarea concreta. Se puede pasar de unos estados a otros mediante transiciones (condiciones de permanencia o de cambio), dependiendo de que sucedan determinados eventos, internos o externos.

En este trabajo se presenta una herramienta de programación visual de robots móviles con autómatas de finito dentro de la plataforma robótica Jderobot. El primer motivo para hacer esta herramienta fue disminuir el tiempo de desarrollo de nuevas aplicaciones, situando al programador en un lenguaje visual más abstracto. El segundo aumentar la calidad de esas aplicaciones, generando automáticamente la mayoría de su código. La herramienta permite que el ingeniero se concentre en las partes específicas de su aplicación y genera automáticamente el resto, consiguiendo así un código más robusto, menos propenso a fallos.

Este artículo se organiza del siguiente modo. En la segunda sección repasamos trabajos relacionados y otras herramientas para la creación de autómatas en robots. En la sección III se describe la herramienta desarrollada y cada una de sus partes. En la sección IV se comentan tres ejemplos de comportamientos programados con la herramienta para validarla experimentalmente. Terminamos resumiendo las conclusiones más relevantes y las líneas por las que se puede continuar este trabajo.

II. Trabajos relacionados

Uno de los trabajos más completos en robótica es el entorno MissionLab desarrollado en Georgia Tech por el profesor R.Arkin. Este entorno incluye como herramienta un editor gráfico de *configuraciones* (`CfgEdit`) [10], similar a los autómatas, que permite especificar misiones, con sus

Universidad Rey Juan Carlos.

E-mails: d.yunta@alumnos.urjc.es, jmplaza@gsyc.es

pasos, transiciones, etc. Permite generar aplicaciones para la arquitectura AuRA, desarrollada por el mismo grupo. Para representar ese autómata inventaron un lenguaje propio, que se almacenaba en fichero, *Configuration Description Language*.

Un ejemplo más reciente es el editor de autómatas dentro de la plataforma orientada a componentes RoboComp, de la Universidad de Extremadura. Por ejemplo en [2] lo han empleado para programar el comportamiento de una carretilla paletizadora.

Otra muestra es el editor gráfico de comportamientos Gostai Studio [6], dentro de la plataforma Urbi. Este editor gráfico genera código *urbiscript* como salida, incluye visualización en tiempo de ejecución del estado del autómata, permite detener y continuar la ejecución de los autómatas generados y ofrece la posibilidad de autómatas jerárquicos.

En el contexto de la RoboCup se utilizan frecuentemente autómatas para generar los comportamientos de los robots humanoides de la liga stándard SPL. Varios equipos emplean la herramienta y el lenguaje XABSL [9; 12] para especificar comportamientos, alrededor del influyente equipo alemán B-Human. La herramienta HFSM se utilizó en el equipo TeamChaos [7] para manejar jerarquías de autómatas para su arquitectura ThinkingCap, permitiendo incluso la edición en caliente del comportamiento en cada estado. En el equipo SPIteam se usa la herramienta Vicode [11] que genera autómatas para la arquitectura BICA.

Una muestra de la potencia de los autómatas es su uso para programar la inteligencia de jugadores automáticos en videojuego. Este escenario supone un entorno más simplificado que la robótica real, porque mucha información del entorno del jugador se obtiene consultado directamente alguna variable del videojuego y no hay que lidiar con dispositivos reales. Por ejemplo, en el exitoso Halo-2 de Bungie se han utilizado árboles de autómatas para desplegar un abanico de más de cien comportamientos diferentes. La empresa Xait, por ejemplo, comercializa herramientas que facilitan la programación de estos autómatas a los desarrolladores de videojuegos, como su XaitControl.

III. Herramienta de programación visual de autómatas

Muchos comportamientos de un robot se pueden representar como autómatas de estado finito. En ellos el comportamiento global del robot viene definido por estados. En cada uno de ellos realizará una acción determinada, la cual se repetirá iterativamente hasta que algo suceda y en ese caso se cambia de estado. Estos cambios de estados se denominan transiciones y dependerán de determinados eventos producidos, bien dentro del robot, bien externos o el simple paso del tiempo.

Nuestra herramienta de desarrollo permite la programación de robots utilizando autómatas de estado finito. En ella se representa el comportamiento del robot de manera gráfica en un lienzo compuesto de transiciones y estados. Además, permite incorporar el código de texto específico correspondiente a cada estado y transición y generar el código resultante en uno o varios ficheros. 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 fuente.

A. Plataforma y robot de referencia

El código resultante de la herramienta viene determinado por el robot de referencia para el cual se van a escribir las aplicaciones y por la plataforma de desarrollo elegida para esas aplicaciones. En nuestro caso se ha elegido el robot *Pioneer* (Figura 1) junto con la plataforma de desarrollo *Jderobot 5.0*. Por lo tanto nuestro código generado tiene funciones y variables propias de Jderobot.

El equipamiento sensorial de este robot de referencia consta de sensores de ultrasonido, sensor láser SICK, una pareja de cámaras, odometría en la base, un cuello mecánico, etc.. Como actuadores principales tiene los motores de la base y los del cuello mecánico que permiten orientar las cámaras a voluntad. En Jderobot existen interfaces software estandarizados para cada uno de estos dispositivos (por ejemplo interfaz Laser, Camara, Encoders, PantiltEncoders, Motors, PantiltMotors). Las aplicaciones generadas con nuestra herramienta utilizan esos interfaces para acceder a los sensores y actuadores del robot de referencia. Jderobot se encarga de cargarlos e inicializarlos al comienzo de la ejecución.

Fig. 1. Robot de referencia Pioneer.

Jderobot (http://jderobot.org) es una plataforma de desarrollo de software para aplicaciones robóticas creada por el Grupo de Robótica de la URJC. Plantea las aplicaciones robóticas como un conjunto de *componentes* que se ejecutan simultáneamente en paralelo, realizan tareas sencillas y concretas e interactúan entre sí. La ejecución concurrente de varios componentes da lugar a un comportamiento. Jderobot utiliza Ice como middleware de comunicaciones entre los distintos componentes, que pueden estar escritos en diferentes lenguajes (C++, Java, Python...) y ejecutarse en máquinas distribuidas conectadas. Las aplicaciones generadas por nuestra herramienta visual de autómatas constan de un único componente que se conecta a los interfaces estándard de sensores y actuadores que ofrecen otros componentes de Jderobot.

B. Diseño

Esta herramienta se divide en dos partes (Figura 2): el editor gráfico y un generador automático de código. Como una caja

negra, se introduce gráficamente el diseño de un autómata finito determinista y devuelve un código autogenerado en varios ficheros .cpp con el componente resultante que se podrá ejecutar en la plataforma *Jderobot*.

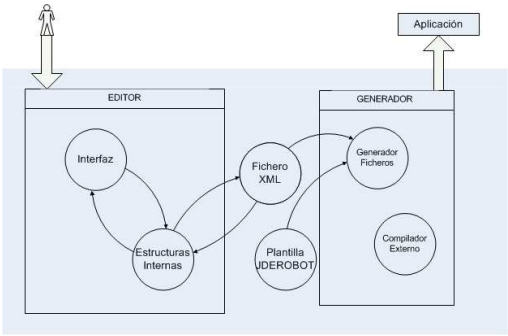

Fig. 2. Diseño de la herramienta de programación visual de autómatas

El editor gráfico (1) ofrece como interfaz la ventana principal de la herramienta, que es donde se encuentra toda la funcionalidad para crear la estructura del autómata. Contiene también (2) ciertas estructuras internas que permiten dar funcionalidad a la interfaz y guardar toda la información relacionada con el autómata y la aplicación en un fichero XML.

Dentro del generador de código hay dos partes diferenciadas: (1) 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 (2) un compilador externo, que compila el código resultante y genera el ejecutable. Todas estas partes específicas se comentan con detalle en las siguientes secciones.

C. Editor Gráfico

El editor permite al usuario representar visualmente el autómata de actuación de su robot de forma clara y sencilla. El componente está representado en forma de autómata finito en un lienzo. En él se puede visualizar cada elemento o estado, y las transiciones que les unen. El autómata puede ser dibujado y modificado según el usuario quiera. 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 en cada estado y las transiciones entre los mismos.

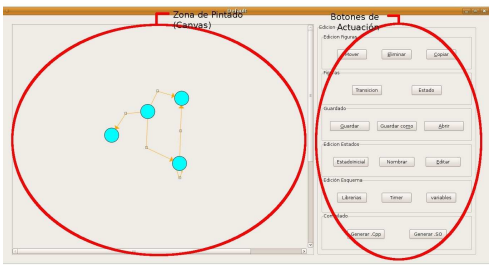

Fig. 3. Interfaz Gráfica del Editor

La interfaz gráfica está dividida en dos partes: el *canvas* y los botones de actuación (Figura 3). El canvas o lienzo de dibujo es la zona donde se representa el autómata, mientras que los diferentes botones de actuación permiten su edición y programación.

Los estados están representados como círculos y su tamaño es constante. 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 transiciones se visualizan como un arco y corresponden a las condiciones de permanencia o de cambio. Estas condiciones pueden ser temporales (pasado un determinado tiempo) o condicionales (cuando suceda un cambio interno o externo en el robot). Cada estado y cada transición lleva asociado un código en texto que define su funcionalidad específica y que se introduce desde la interfaz gráfica.

Los botones se estructuran en 5 grupos: botones de objetos o figuras (transiciones y estados), botones de edición de figuras (mover, eliminar y copiar), botones de funciones de guardado, botones de edición de estados (nombrar, editar cada objeto y definir estado inicial), botones de edición y programación del componente (librerías, *timer* y variables auxiliares) y botones de compilado.

Otra funcionalidad que permite la interfaz principal es modificar las características esenciales del autómata. A saber: establecer el estado inicial del autómata, establecer la velocidad en iteraciones por segundo de la aplicación, 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 y definir el nombre y ubicación de la aplicación, que será un componente de Jderobot.

D. Fichero intermedio XML

El editor gráfico permite guardar en el disco un fichero XML como soporte no volátil de todas las propiedades de la aplicación que se está creando: la estructura del autómata, las características de cada nodo y transición, las propiedades del componente resultante, etc.. Desde ese fichero se generará el componente equivalente en lenguaje textual. El fichero XML permite guardar el componente, incluso parcialmente desarrollado, para su posterior modificación o terminación, editarlo en otro equipo o cargarlo en versiones posteriores de la herramienta.

En el archivo XML se guardan todas las características gráficas de los elementos dibujados en el canvas (coordenadas de los puntos, nombre de los estados, conexiones entre ellos, etc) así como las características propias del componente (nombre del componente, directorio de guardado, estado inicial, librerías utilizadas, tiempo de iteración, etc) (Figura 4). Cada estado tiene asociado un identificador único que permite distinguirlos y establecer las relaciones de dependencia entre ellos. De cada estado se guardan sus coordenadas gráficas de origen y destino, nombre, código (correspondiente a las acciones realizadas) y transiciones salientes. De cada transición se guarda un identificador de destino (estado al que se conecta), coordenadas de la situación de la caja en el canvas, nombre

```
<Vicoe>
 - <Estado estado_inicial="true">
      <id>1</id>
      <origenX>329</origenX>
      <origenY>178</origenY>
      <destinoX>369</destinoX>
      <destinoY>208</destinoY>
      <nombre>NomEstado</nombre>
    - <codigo>
            /*Código del estado*/
      </codigo>
    - <transiciones>
          <transicion>
             <box x1="331" y1="242"
            x2="337" y2="248"/>
          <nombre>NomTrans</nombre>
          <destino>2</destino>
          <codigo>
              /*Código transición*/
          </codigo>
          </transicion>
      </transiciones>
   </Estado>
 +<Estado estado_inicial="false">
    </Estado>
 +<Estado estado_inicial="false">
    </Estado>
   <tiempoIteracion>
        800000
   </tiempoIteracion>
   <nombreEsquema>
        /dir/nombre
   </nombreEsquema>
 - <Librerias>
      <lib>motor</lib>
      <lib>camara</lib>
   </Librerias>
   <variables_aux>
      int i;float j
   </variables_aux>
   <funciones_aux/>
</Vicoe>
```

Fig. 4. Ejemplo Fichero XML

y código (correspondiente a la condición de permanencia o transición) o tiempo de iteración.

E. Plantilla de autómatas

Para materializar y transformar todos los estados y transiciones a código fuente textual se ha utilizado una plantilla de Jderobot para aplicaciones de control, se ha extendido para construir con ella una plantilla de autómatas de estado finito y en esa plantilla de autómatas se empotra el código específico insertado en la herramienta asociado a cada estado o transición.

La plantilla Jderobot para aplicaciones tiene dos partes, una parte de control y una parte gráfica. Cada una de ellas ejecuta iteraciones periódicas en un hilo concurrente respectivo. La iteración del hilo de control contiene todo el código correspondiente al comportamiento del robot. La iteración del hilo gráfico se usa para visualizar en tiempo de ejecución datos sensoriales o estructuras internas del programa, o comandar desde el GUI acciones al robot, etc.. Periódicamente actualiza y muestrea el GUI y el estado de sus objetos gráficos.

Esta plantilla inicial se ha particularizado para materializar autómatas de estado finito conformando la plantilla de autómatas (Figura 5). La herramienta construye autómaticamente esta plantilla de autómatas y en ella empotra el código de todos los estados y transiciones del autómata particular que se está editando: su código, nombres, cambios entre estados, etc.. El funcionamiento del autómata así generado ejecuta en iteraciones que se repiten periódicamente según la frecuencia indicada.

En cada iteración del hilo de control comprueba en qué estado se encuentra y ejecuta el código correspondiente. En cada estado se siguen los pasos de la Figura 5: ejecutar el código de percepción para la actuación, ejecutar el código de actuación concreta, ejecutar código de percepción para la transición, verificar las condiciones de las transiciones, y en caso de que se cumpla la transición cambiar al estado correspondiente. Los códigos específicos de percepción y de actuación los introduce el usuario humano a través de interfaz gráfico y quedan asociados a cada estado.

```
switch (Estado_Actual) {
    case estado1:
        /*Código Percepción
            Actuación*/
        ...
        /*Código Actuación
            Estadó*/
        ...
        /*Código Percepción
            Transición*/
        ...
        /*Comprobar Condiciones
            Permanencia*/
        ...
        if (cambio de estado){
          Estado_Actual = estado2;
        }else(cambio de estado2){
        }
        ...
        break;
    case estado2:
        ...
        break;
    case estadoN:
        ...
        break;
}
```

Fig. 5. Código de control de la plantilla de autómatas

En cada iteración del hilo de visualización se introducen automáticamente las funciones (Figura 6) que mantienen una ventana auxiliar similar al lienzo del propio editor gráfico, útil para la depuración. En ella se muestra el diseño del autómata, cómo cambia de estado la aplicación creada y cuál es el estado activo.

```
if (Estado_Actual != Estado_Anterior){
  this->ventana->
  change_nodo_color(Estado_Anterior,3);
}
this->ventana->
  change_nodo_color(Estado_Actual,0);
Estado_Anterior = Estado_Actual;
```

Fig. 6. Código de visualización de la plantilla de autómatas

F. Generador automático de código

Esta parte genera, compila el código en C++ resultante y así materializa el autómata diseñado visualmente generando un componente compatible con *Jderobot 5.0*.

La forma de construir una aplicación consiste en usar la plantilla del autómata comentada rellenándola con la información concreta del autómata particular que está siendo generado. Para ello se procesa el archivo XML generado desde el editor y se intercala con el código correspondiente de la plantilla, insertando el código de texto asociado a cada estado y transición correspondiente. Una vez generado se compila y se obtiene el ejecutable (Figura 7).

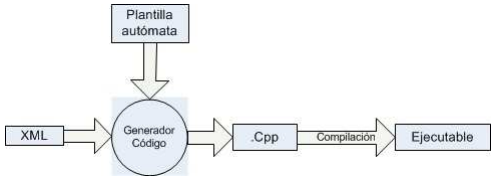

Fig. 7. Diagrama generador de código

G. Variables sensoriales y dependencias

En estas aplicaciones existen dos tipos de variables: (1) Las de sensores y actuadores de los que depende el componente, y (2) las auxiliares. Las primeras concretan en software los sensores y actuadores del robot de referencia. Vienen resueltas en nuestra aplicación al utilizar los propios interfaces estandard en Jderobot para el robot de referencia, ya sea real o simulado. La herramienta las declara e incluye automáticamente una vez seleccionadas visualmente. Son variables accesibles en todo momento a cualquier parte de la aplicación, ya sea para escritura si se trata de actuadores o para lectura si se trata de sensores.

Las variables auxiliares son propias del usuario, se deben definir e inicializar desde la herramienta. Por ejemplo son variables utilizadas para cálculos intermedios o de ayuda en los estados o transiciones. Se pueden incorporar mediante una ventana auxiliar de texto del editor gráfico, donde habrá que declararlas e inicializarlas.

IV. Experimentos

Para validar la herramienta desarrollada se han realizado varios autómatas, programando con ellos varios ejemplos de menor a mayor complejidad que generan distintos comportamientos de robots. Se han probado sobre el robot de referencia en el simulador Gazebo (Figura 11). Gazebo es un simulador 3D multirrobot, forma parte del proyecto Player/Stage y es el simulador de referencia elegido por la plataforma ROS, que se está convirtiendo en un estandard de facto en la comunidad robótica. Gazebo permite simular cámaras, láser, robots con ruedas, con patas, un conjunto variado de sensores y actuadores, y todo tipo de terrenos 3D.

El componente `gazeboserver` de Jderobot se conecta al simulador y ofrece los mismos interfaces que otros componentes (drivers) ofrecen al conectarse al laser real, cámaras reales, robot físico, etc. La aplicación generada con el autómata ejecuta indistintamente sobre el robot simulado y el real, no es consciente de sobre qué está funcionando, simplemente se conecta a los interfaces respectivos.

A. Comportamiento Choca-Gira

El primer ejemplo genera el comportamiento de un robot capaz de no chocarse con los obstáculos que se encuentra. El robot debe avanzar recto hasta que en su trayectoria se encuentre con algún obstáculo de frente, entonces debe esquivarlo retrocediendo un poco y girar hasta poder seguir avanzando.

Este comportamiento se ha simplificado en tres estados: *anda*, *retrocede* y *gira*. En el estado *anda*, el robot avanza en línea recta hasta encontrar un obstáculo. Cuando encuentre un obstáculo con el sensor láser, el robot pasará al estado *retrocede*, en el cual retrocederá durante 3 segundos. Una vez haya retrocedido cierta distancia, el robot pasará al estado *gira*, en el que gira hasta que no se encuentre con ningún obstáculo. Cuando eso ocurra regresará al estado *anda*, en el que seguirá andando. En la figura 8 puede verse el diseño de este autómata.

Fig. 8. Autómata del comportamiento choca-gira.

Para esta aplicación se ha utilizado el sensor láser y los motores (valores V y W correspondientes a la velocidad lineal y angular del robot). El código insertado en el estado *avanza* es el siguiente:

```
this->controller->setV((float)100);
this->controller->setW((float)0);
this->navegacion->cogerLaser0(\&laser);
```

B. Comportamiento Sigue-Línea

Este segundo ejemplo genera el comportamiento de un robot que sigue una línea roja pintada en el suelo usando exclusivamente el sensor cámara. Si en algún momento se pierde de vista la línea roja, debe localizarla y seguir avanzando sobre ella.

Se ha simplificado este comportamiento en tres estados: *sigue*, *para* y *corrige*. En el estado *sigue*, el robot avanza por la línea roja mientras que esté situada más o menos en el centro. Si en algún momento la línea roja se sale del centro de la imagen, el robot pasa al estado *para*. El robot se detiene

y seguidamente pasa al estado *corrige*, donde girará a la izquierda o derecha dependiendo de la posición de la línea en la imagen de la cámara. La Figura 9 muestra el interfaz gráfico desplegado por la aplicación generada automáticamente, que contiene el propio autómata y el estado activo en cada momento.

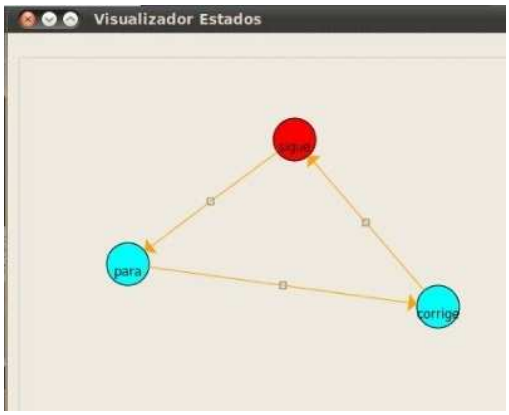

Fig. 9. Autómata del comportamiento Sigue-línea.

En este ejemplo la acción realizada en cada estado depende de la percepción sensorial realizada en él, a diferencia del ejemplo anterior donde la acción de cada estado es constante. Por ejemplo, en el estado *sigue* y *corrige* el valor de giro es variable dependiendo de dónde se aprecie la línea roja de la imagen, materializando un control proporcional.

C. Comportamiento Ratón y Gato

En este tercer ejemplo se ha programado un robot (considerado *gato*) para que persiga a su congénere (considerado *ratón*) por el mundo. El ratón es teleoperado y si se para, el gato se detendrá junto a él a una cierta distancia. Si en algún momento pierde de vista al ratón, el gato deberá buscarlo, localizarlo y avanzar hasta situarse de nuevo enfrente suyo. Además el robot gato tiene que esquivar los obstáculos que puedan aparecer en su persecución del ratón.

El comportamiento del robot gato se ha diseñado con un autómata de 4 estados: *parado*, *buscar*, *obstáculos* y *avanzar* (Figura 10). En el estado *avanzar*, el robot avanza por el mundo. Mientras avanza comprobará varias circunstancias: si hay obstáculos en su camino y si el robot ratón se encuentra proximo a él. En caso de que éste esté próximo y no haya obstáculos, pasará al estado *centrar* para posicionarlo en medio y posteriormente avanzar hacia él. En caso de que en su trayectoria haya obstáculos pasará al estado *obstáculos*, donde su función será precisamente esquivarlos. Una vez sorteados, seguirá en su trayectoria. En caso de no localice al ratón, seguirá avanzando por el mundo hasta encontrarlo.

En las imágenes de la Figura 11 se puede observar cómo el robot gato llega a alcanzar el objetivo. En la segunda imagen el robot avanza por el mundo sorteando los obstáculos, todavía no tiene localizado al ratón. En la tercera imagen ya tiene localizado al robot ratón y avanza hacia él. En la cuarta ya ha

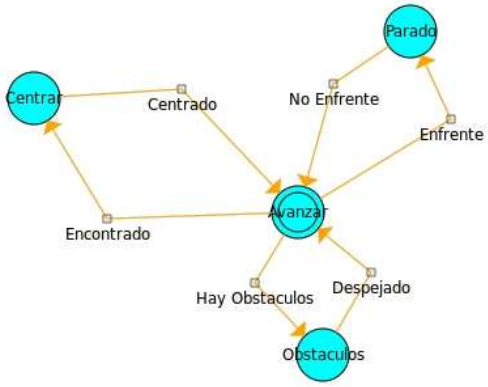

Fig. 10. Autómata del comportamiento del robot *Gato*.

llegado al objetivo y está parado detrás de él. En caso de que el robot ratón ande, el robot gato le persigue hasta volverse a situar detrás suyo.

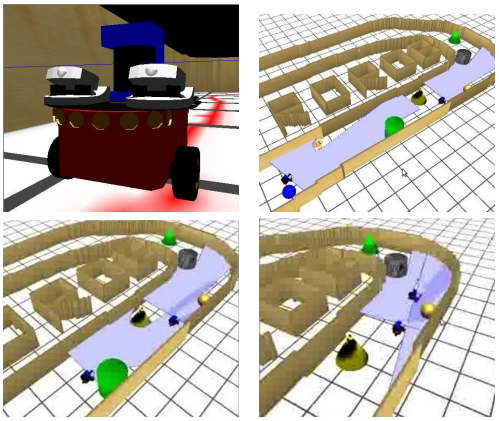

Fig. 11. Pioneer en Gazebo y escenas del robot Gato persiguiendo al ratón.

V. Conclusiones

En este artículo hemos presentado una herramienta para la programación visual de comportamientos en robots móviles usando autómatas de estado finito. La herramienta se ha dividido en un editor gráfico de autómatas y un generador automático de código. El editor permite la introducción por el usuario humano de la estructura del autómata, sus estados y sus transiciones. Además crea un fichero XML que guarda de modo no volátil esa estructura. El generador automático de código usa el código fuente de una plantilla de autómatas en C++ y sobre ella empotra el código específico asociado a cada estado y cada transición, construyendo así el código fuente de la aplicación.

La herramienta permite una creación rápida de aplicaciones robóticas y genera un código menos propenso a fallos, pues la mayor parte de código se reutiliza automáticamente de una plantilla libre de errores. Las aplicaciones así creadas

no pierden la naturaleza reactiva, pues ejecutan iteraciones de control en ciclos periódicos. Además, el abanico de comportamientos generables es muy extenso debido a la potencia de la abstracción de los autómatas de estado finito. Una característica positiva es que la aplicación generada muestra automáticamente en su interfaz gráfico el diagrama de estados del autómata, resaltando el estado activo en cada momento, lo cual resulta muy útil para depuración.

La herramienta se ha validado experimentalmente creando con ella tres comportamientos diferentes sobre el robot de referencia. En los ejemplos se han probado autómatas de cuatro estados, pero la herramienta puede manejar muchos más y es ahí cuando su aporte es mayor frente a la alternativa de programar directamente en código fuente el autómata completo.

La versión actual sólo trabaja con autómatas de un único nivel , estamos trabajando en la extensión de la herramienta a autómatas jerárquicos, donde la respuesta en un estado pueda ser a su vez la dada por otro autómata con subestados. Con esta mayor potencia representativa se aumentaría aún más el abanico de comportamientos generables y favorecería la posible reutilización de subautómatas entre distintas aplicaciones. Otra línea futura es la generación de aplicaciones para otras plataformas diferentes a Jderobot y otros robots, por ejemplo la arquitectura BICA que hemos desarrollado para aplicaciones con el robot humanoide Nao.

AGRADECIMIENTOS

Este trabajo está financiado en parte por el proyecto S2009/DPI-1559, RoboCity2030-II, de la Comunidad de Madrid y por el proyecto 10/02567 del Ministerio de Ciencia e Innovación.

REFERENCES

[1] S. Carpin, M. Lewis, J. Wang, S. Balarkirsky, and C. Scrapper. Usarsim:a robot simulator for research and education. In *Proceedings of the IEEE 2007 International Conference on Robotics and Automation, pp. 1400-1405*, 2007.

[2] R. Cintas, L. Manso, L. Pinero, P. Bachiller, and P. Bustos. Robust behavior and perception using hierarchical state machines: A pallet manipulation experiment. *Journal of Physical Agents*, 5(1):35–44, 2011.

[3] P.T. Cox and T.J. Smedley. Visual programming for robot control. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 217–224, sep 1998.

[4] Jeff Craighead, Robin Murphy, Jenny Burke, and Brian Goldiez. A survey of commercial open source unmanned vehicle simulators. In *Proceedings of the IEEE 2007 International Conference on Robotics and Automation, pp. 852-857*, 2007.

[5] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics ICAR-2003, pp. 317-323*, Coimbra (Portugal), 2003.

[6] Gostai. Gostai studio suite. http://www.gostai.com/products/studio/gostai_studio/, 2012.

[7] D. Herrero-Perez, F. Bas-Esparza, H. Martinez-Barbera, F. Martin, C.E. Aguero, V.M. Gomez, V. Matellan, and M.A. Cazorla. Team chaos 200. In *Proceedings of the IEEE 3rd Latin American Robotics Symposium, 2006. LARS '06*, pages 208–213, 2006.

[8] T. Laue, K. Spiess, and T. Röfer. Simrobot - a general physical robot simulator and its application in robocup. In *In A. Bredenfeld, A. Jacoff, I. Noda, Y. Takahashi (Hrsg.), RoboCup 2005: Robot Soccer World Cup IX, Nr. 4020, S. 173–183, Lecture Notes in Artificial Intelligence. Springer*, 2006.

[9] M. Lötzsch, J. Bach, H.D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language xabsl. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*. Springer, 2004.

[10] D C MacKenzie and R C Arkin. Evaluating the usability of robot programming toolsets. *The International Journal of Robotics Research*, 17(4):381, 1998.

[11] F. Martín, C. Agüero, J.M. Cañas, and E. Perdices. Behavior-based iterative component architecture for soccer applications with the nao humanoid. In *Proceedings of WAF2010 XI Physical Agents Workshop*, pages 131–142, Valencia, september 2010.

[12] Max Risler. *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines*. PhD thesis, Fachbereich Informatik, Technischen Universitat Darmstadt, 2009.