PROGRAMACIÓN DE ROBOTS MÓVILES

José M. Cañas [*,2] **Vicente Matellán** [*,2]
Rodrigo Montúfar [**,1]

[*] *Universidad Rey Juan Carlos, Móstoles, España*
[**] *Inst. Nal. de Astrofísica, Óptica y Electrónica, Tonanzintla, México*

Resumen: El objetivo de este artículo es describir los entornos de programación de robots móviles más comunes en la actualidad, sus características y las tendencias más recientes. Actualmente el software de los robots móviles se estructura en tres niveles: sistema operativo, plataforma de desarrollo y aplicaciones concretas. Las plataformas de desarrollo han surgido los últimos años con la idea de facilitar la construcción incremental de estas aplicaciones robóticas. Suelen ofrecer una interfaz uniforme de acceso al heterogéneo hardware de los robots, una arquitectura software concreta y un conjunto de funcionalidades comunes listas para su reutilización.

Palabras Clave: Programación de robots, robots móviles autónomos, herramientas software, entornos de programación, simuladores.

1. INTRODUCCIÓN

Conseguir que los robots hagan cosas útiles autónomamente es una tarea difícil. Uno de los aspectos principales para ello es su programación. Todos los robots tienen sensores, actuadores y procesadores, y el software que se ejecuta en ellos es el que da la vida a esos componentes hardware, el que enlaza los datos recibidos por los sensores con las respuestas de actuación. Desde esta perspectiva, la generación de comportamiento en un robot consiste en escribir el programa que al ejecutarse en el robot *causa* ese comportamiento. La "autonomía" y la "inteligencia" residen en ese programa. Por ejemplo, en los robots móviles el comportamiento principal es su movimiento. Los programas que se ejecutan en el robot determinan la manera en que éste se mueve por el entorno, reaccionando ante obstáculos percibidos por los sensores, acercándose a algún destino, etc. y para ello tienen que enviar continuamente las órdenes adecuadas a los motores.

Muchos de los robots que se venden actualmente son productos cerrados, programados por el fabricante e inalterables, por ejemplo, la aspiradora robótica Roomba de iRobot [3] o el perrito Aibo de Sony en sus inicios [4]. En contraste, otra parte relevante del mercado son los robots programables, cuyos principales clientes son los centros de investigación, normalmente interesados en programarlos ellos mismos. En estos casos, el fabricante vende los robots con un software que permite su programación por parte del usuario.

[1] Financiado parcialmente por la Agencia Española de Cooperación Internacional
[2] Este trabajo ha sido financiado por el Ministerio de Ciencia y Tecnología, proyecto DPI2004-07993-C03-01

[3] `http://www.irobot.com`
[4] En el verano de 2002 Sony hizo pública la interfaz de programación de sus perritos, Open-R, en parte forzados por un *hacker* la mascota robótica Aibo [5] de Sony que había conseguido desvelar algunas de sus interfaces

El modo en que se programan los robots ha ido evolucionando. Históricamente los robots eran desarrollos únicos, no se producían en serie, y los programas de control se construían empleando directamente los *drivers* para acceder a los dispositivos sensoriales y de actuación. El *sistema operativo* del robot era mínimo, básicamente una colección de *drivers* con rutinas para leer datos de los sensores y enviar consignas a los actuadores. En este contexto, el programa de aplicación invocaba directamente las funciones de la librería que ofrecía el fabricante en sus *drivers*. La aplicación se situaba directamente encima del sistema operativo, como ilustra la figura 1(a).

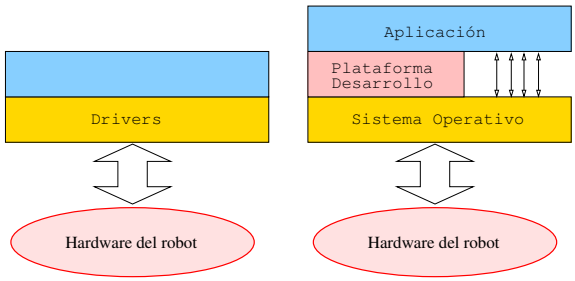

Figura 1. Programación de robots: (a) sobre *drivers* específicos de sensores y actuadores. (b) sobre una plataforma de desarrollo

Con el asentamiento de los fabricantes, el trabajo de muchos grupos de investigación y el paso de los años han ido apareciendo *plataformas de desarrollo* que simplifican la programación de aplicaciones robóticas. Estas plataformas ofrecen acceso más sencillo a sensores y actuadores, suelen incluir un modelo de programación que establece una determinada organización del software y permite manejar la creciente complejidad del código cuando se incrementa la funcionalidad del robot. El diseñador programa sus aplicaciones robóticas finales sobre esa plataforma de desarrollo, como muestra la figura 1(b).

En este artículo pretendemos reflejar nuestra experiencia de los últimos años programando diferentes tipos de robots móviles, con diferentes entornos de programación y en diferentes centros de investigación, todo ello con el objetivo de servir de punto de partida en la toma de decisiones en este contexto.

En la siguiente sección abordamos las peculiaridades de la programación de robots móviles. En la tercera se describe el nivel básico: el hardware y los sistemas operativos de robots. En la cuarta analizamos la función de las plataformas de desarrollo que han ido apareciendo los últimos años. En la quinta se detallan algunas de las plataformas concretas más extendidas, y finalizamos con unas breves reflexiones en la sección sexta a modo de conclusiones.

2. SOFTWARE DE ROBOTS

La mecánica de creación de aplicaciones para robots no difiere especialmente de la de aplicaciones en otros ámbitos del software: El programador tiene que escribir la aplicación en cierto lenguaje, compilar y enlazar su código con las bibliotecas de la plataforma y/o del sistema operativo, y finalmente ejecutarla en los computadores a bordo del robot. Aunque la mecánica sea similar, las aplicaciones de robots móviles sí presentan unos requisitos específicos que condicionan las características de esos programas.

Una característica común a muchos robots es que tienen recursos computacionales y de almacenamiento limitados: carecen de disco duro, pantalla suficiente, etc. Por eso es normal que en esos casos se utilice el PC como entorno de desarrollo y los compiladores cruzados generen en el PC el programa ejecutable que correrá en el procesador a bordo del robot.

2.1 Requisitos específicos

Escribir programas para robots móviles es una tarea complicada, ya que los robots son sistemas complejos. La programación de robots móviles suele ser más exigente que la creación de programas tradicionales como aplicaciones de ofimática, bases de datos, etc. A continuación se presentan algunos condicionantes propios, diferenciadores de otros ámbitos del software.

1. Los programas de robots móviles están directamente conectados a la realidad física, a través de sensores y actuadores. Los sensores obtienen información de esta realidad y los actuadores la modifican. Esta situación implica que el software debe ser ágil, tomar decisiones con vivacidad para controlar correctamente a los actuadores. Por esta razón, se requiere de actuación en tiempo real, si no estricto, al menos blando.

2. Una aplicación de robots móviles típicamente debe estar pendiente de varias fuentes de actividad y objetivos a la vez. El programa de un robot tiene que atender a muchas cosas simultáneamente: recoger nuevos datos de varios sensores, refrescar la interfaz gráfica, enviar periódicamente consignas a los motores, enviar o recibir datos por la red a otro proceso de la aplicación, etc. Por ello estas aplicaciones suelen ser concurrentes, lo cual les añade cierta complejidad.

3. Las aplicaciones que corren a bordo de los robots deben encargarse también de su interfaz gráfica. Aunque la interfaz gráfica no es indispensable para generar y materializar el comportamiento autónomo en el robot, normalmente resulta muy útil como herramienta de depuración.

Además de las posibilidades de interacción con el usuario, la interfaz gráfica permite la visualización en tiempo de ejecución de estructuras y variables internas (e.g. representaciones del mundo, mapas, estados, etc.).

4. El software de robots es cada vez más distribuido, tal y como apuntan Woo et al.(Woo *et al.*, 2003). Es usual que las aplicaciones de robots tengan que establecer alguna comunicación con otros procesos ejecutándose en la misma máquina o en una diferente. La distribución ofrece posibilidades ventajosas como ubicar la carga computacional en nodos con mayor capacidad o la visualización remota; y en sistemas multirrobot, hace posible la integración sensorial, la centralización y la coordinación.

5. Los programadores de robots se enfrentan a una creciente heterogeneidad, en distintos sentidos, que dificulta su tarea. En cuanto al hardware, existe una gran diversidad de dispositivos sensoriales y de actuación, y por lo tanto de interfaces. El programador debe dominarlos para acceder a ellos desde las aplicaciones. Por otro lado, mientras que en muchos campos de la informática sí hay bibliotecas que un programador puede emplear para construir su propio programa, en el software de robots no hay un marco homogéneo ni hay estándares que propicien la reutilización de código (Utz *et al.*, 2002; Montemerlo *et al.*, 2003; Côté *et al.*, 2004) y la integración. En robótica cada aplicación prácticamente ha de construirse desde cero para cada robot concreto.

6. El conocimiento sobre cómo generar y descomponer el comportamiento artificial es muy limitado. Cómo dividir el comportamiento de robots en unidades básicas sigue siendo materia de investigación, y no hay una guía universalmente admitida sobre cómo organizar el código de las aplicaciones de robots para que sea escalable y se puedan reutilizar sus partes. Cada desarrollador escribe su aplicación combinando ad hoc los bloques de código que puedan existir en su entorno.

2.2 Lenguajes

En cuanto a los lenguajes que se emplean para programar robots, no hay diferencias significativas con los utilizados en las aplicaciones informáticas tradicionales. Aunque han existido intentos de establecer lenguajes específicos para programar robots, como *Task Description Language* (TDL) (Simmons and Apfelbaum, 1998) o *Reactive Action Packages* (RAP) (Firby, 1994), no han sido nunca de uso general. El objetivo básico de estos lenguajes es incluir en la propia sintaxis del lenguaje mecanismos que resultan ventajosos para la programación de robots, tales como la descomposición de tareas, la monitorización de ejecución o la sincronización.

En los brazos robotizados industriales es frecuente el uso de lenguajes de bajo nivel, prácticamente ensamblador (Biggs and MacDonald, 2003) del microprocesador de turno. En los robots móviles, sin embargo, se usan principalmente lenguajes de alto nivel, más versátiles y potentes. De hecho, la creciente incorporación del ordenador personal como procesador principal ha dado paso a toda suerte de lenguajes: C, C++, JAVA, Python, etc.

Un ejemplo curioso es el lenguaje que incluye LEGO para que los niños programen sus robots RCX (Biggs and MacDonald, 2003). El `RCX-code` es completamente visual y en él un programa consiste de una columna que se forma encadenando en secuencia bloques gráficos que son instrucciones varias (de espera, actuación, suma a una variable, condicionales, etc). Se pueden incluir varias columnas, a modo de hilos de ejecución concurrentes.

Actualmente el lenguaje más extendido para programar robots es C. Este lenguaje supone un buen compromiso entre potencia expresiva y rapidez. Como lenguaje compilado su eficiencia temporal es superior a otros lenguajes interpretados. Hasta hace unos años gran parte del software proporcionado por los fabricantes de robots estaba codificado en C, como las librerías de Saphira con que se vendían los robots de ActivMedia, o el entorno de desarrollo de los robots de RWI (RWI, 1999). Muchos robots pequeños se programan en C, como el robot EyeBot (Bräunl, 2003) y el robot LEGO RCX, que se puede programar con una variante recortada de C llamada NQC (Baum, 2000).

Recientemente se puede observar un crecimiento en los desarrollos y aplicaciones en C++. Por ejemplo, el entorno ARIA (ActivMedia, 2002) para programar robots de ActivMedia, el entorno OPEN-R (Sony, 2003) de programación del perrito Aibo de Sony, la plataforma Miro (Utz *et al.*, 2002), Mobility (RWI, 1999), etc. El principal avance sobre C es que proporciona la abstracción de orientación objetos, con los mecanismos de herencia y polimorfismo asociados. Potencialmente también simplifica la reutilización de componentes. Una ventaja más es que la portabilidad desde C a C++ es relativamente sencilla.

2.3 Simuladores

Una herramienta muy útil en la programación de robots son los *simuladores*, los cuales ofrecen un entorno virtual en el que emulan las observaciones de los sensores y los efectos de las órdenes a los actuadores. Sirven para evaluación, depuración y

ajuste del programa de control antes de ser llevado al robot real.

Los primeros simuladores eran poco realistas y almacenaban mundos planos donde había obstáculos estáticos bidimensionales. Con los años se ha ido ganando en realismo, incorporando ruido en los sensores y en las actuaciones. Hoy en día se tienen simuladores tridimensionales, como Gazebo[6] (en la figura 2 se puede observar un mundo simulado con Gazebo) y JMR (Lozano Ortega *et al.*, 2001), capaces de simular sensores tan complejos como la visión. En los simuladores más potentes se ha incluido también la capacidad de representar un *conjunto* de robots operando en el mismo escenario de modo simultáneo.

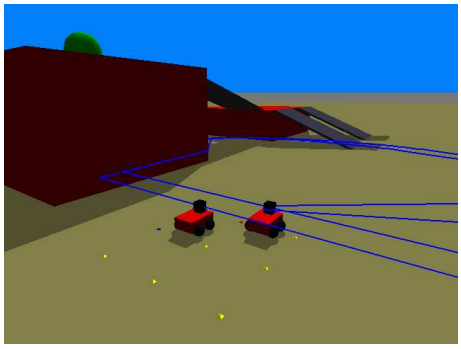

Figura 2. Simulador de robots Gazebo

Muchos fabricantes incluyen un simulador para sus robots (e.g. `EyeSim` para el robot EyeBot, `Webots` para Kephera y Koala), aunque también existen muchos desarrollos libres (`Stage`, `Gazebo`, `JMR`, etc.). Los simuladores libres tratan de incluir soporte para los robots de diversos fabricantes. La configuración concreta del conjunto de robots a simular, la disposición y parámetros de sus sensores se suele especificar en un fichero de configuración. Dos de los simuladores más relevantes de hoy en día son el `SRIsim`[7] de Kurt Konolige, que se emplea en los robots de ActivMedia y los simuladores `Stage` y `Gazebo` de software libre orientados a multirrobot (`Stage` soporta 2D y colonias numerosas de robots, y `Gazebo` soporta 3D).

3. SISTEMAS OPERATIVOS

Como mencionamos en la introducción, la primera opción para el desarrollador de aplicaciones robóticas consiste en escribir su código sobre la funcionalidad que le proporciona el sistema operativo del robot. La misión principal del sistema operativo es ofrecer a los programas un acceso básico al hardware del robot, permitir su manipulación y uso desde los programas. Fundamentalmente debe permitir recoger lecturas de los sensores y enviar órdenes a los actuadores incorporando *drivers* que dan soporte software de bajo nivel a los dispositivos físicos.

En cuanto a la multitarea, los sistemas operativos suelen proporcionar mecanismos que nos permiten la programación con múltiples hilos de ejecución, así como la comunicación y coordinación entre ellos. El sistema operativo también suele incluir soporte para el hardware de comunicaciones (e.g. tarjetas de red inalámbricas) y para los elementos de interacción del robot, ya sean botones físicos, pantallas pequeñas, pantallas de ordenador, etc.

3.1 Sistemas operativos dedicados y generalistas

Gran parte de los robots actuales incluyen sistemas operativos ad hoc. Sin embargo, desde hace algún tiempo se ha extendido el uso en los robots de sistemas operativos de propósito general y de ordenadores personales portátiles o de sobremesa empotrados.

Así, los sistemas operativos de propósito general como Linux o MS-Windows han entrado en el mundo de los robots. Estos sistemas, además de los *drivers* del hardware, incluyen un conjunto muy extenso de bibliotecas genéricas, útiles para la programación de robots, tales como las bibliotecas e interfaces para la multitarea, las comunicaciones y las interfaces gráficas.

A modo de ejemplo, en esta sección vamos a analizar un sistema operativo específico, ROBIOS para el robot EyeBot (figura 3), y un sistema operativo de propósito general, GNU/Linux, muy extendido en los centros de investigación.

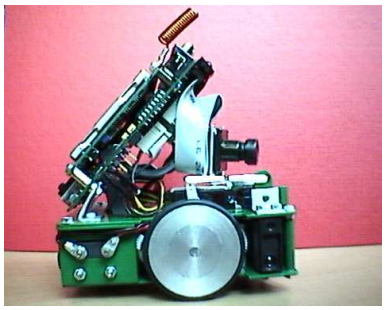

Figura 3. Robot EyeBot

ROBIOS El sistema operativo del robot EyeBot (Bräunl, 2003) se llama ROBIOS. Este robot es de tamaño pequeño y está equipado con un microprocesador Motorola 68332 a 35 MHz, al cual se le conectan dos motores con *encoders*, tres sensores de infrarrojos y una cámara digital. Su hardware también incluye una pequeña pantalla, cuatro botones y un radioenlace. Este sistema operativo permite la carga de programas a través de un puerto serie y su ejecución pulsando los botones. Como acceso básico al hardware desde

[6] `http://playerstage.sourceforge.net`
[7] `http://www.ai.sri.com/~konolige/saphira`

los programas, incluye una API (*Application Programming Interface*) con funciones para recoger las lecturas de los sensores de infrarrojos, para obtener las imágenes de la cámara y para hacer girar los motores a cierta velocidad.

Además incluye dos funciones que le permiten enviar y recibir bytes a través de un radioenlace, a otros EyeBots o un PC que se encuentren dentro del alcance radio de su antena. En cuanto a la interfaz gráfica, ROBIOS incluye primitivas para muestrear si los botones están pulsados, pintar y refrescar la pantalla.

Con respecto a la multitarea, ROBIOS tiene primitivas para crear hebras, detenerlas durante un tiempo, matarlas, etc. Ofrece dos modos de repartir el tiempo de procesador entre hebras: con desalojo (en rodajas de tiempo) y sin desalojo (con un testigo que va pasándose de una hebra a la siguiente), correspondientes a las hebras de kernel o de usuario típicas de sistemas operativos tradicionales. También ofrece semáforos para coordinar la ejecución concurrente de esas hebras y el acceso a variables compartidas.

GNU/Linux Un sistema operativo generalista que ha ganado gran aceptación en la comunidad robótica es GNU/Linux (RWI, 1999; Gerkey *et al.*, 2003; Montemerlo *et al.*, 2003). Incluye un amplio abanico de herramientas de desarrollo, como el compilador de C/C++ `gcc`, el editor `emacs`, etc.

En el ámbito de la multitarea, la biblioteca más usual es *Pthreads* que proporciona hebras POSIX de kernel que se pueden comunicar a través de memoria compartida. Pthreads también ofrece semáforos para controlar el acceso a esas variables compartidas o resolver problemas de concurrencia que puedan aparecer. Estos mecanismos se suman a los que ya ofrece GNU/Linux como creación de procesos, tuberías, `fifos`, memoria compartida, etc..

El mecanismo de comunicaciones más extendido en GNU/Linux son los *sockets*, que permiten la comunicación intermáquina entre programas, una abstracción que le permite olvidarse de los detalles de red, protocolos de acceso al medio, etc. En concreto soporta IP para el nivel de red, y los protocolos del nivel de transporte TCP y UDP. De esta manera el programador de la aplicación no debe preocuparse de localizar la máquina destino, ni modificar su código cuando, por ejemplo, el robot se conecta a la red por ethernet en vez de usar la red inalámbrica.

GNU/Linux ofrece también múltiples librerías para crear y manejar interfaces gráficas con `X-Window`, que es el sistema más extendido en el mundo Unix. Encima de las librerías básicas (`Xlib` o `Xt`), que son flexibles pero complejas, GNU/Linux dispone de varias librerías que simplifican el manejo de la interfaz gráfica, como `Qt`, `XForms`, `GTK`, etc.

El potente soporte de GNU/Linux para la multitarea, las comunicaciones remotas y las bibliotecas gráficas existentes en ese entorno permiten abordar con éxito la naturaleza concurrente, distribuida y la necesidad de visualización típicas de los programas para robots, tal como vimos en la sección 2.

Una de las desventajas del uso de GNU/Linux en robótica es que no es un sistema operativo de tiempo real, pues no permite acotar plazos. No ofrece *garantía* de plazos (tiempo real *duro*) porque su sistema de planificación de procesos no las asegura. En este sentido contrasta con sistemas operativos similares como QNX, o la variante RT-Linux. Sin embargo, GNU/Linux resulta lo suficientemente ágil para los requisitos de aplicaciones no críticas. Por ejemplo, permite detener hebras durante milisegundos.

4. PLATAFORMAS DE DESARROLLO

Las aplicaciones con robots presentan cada vez mayor complejidad y ofrecen mayor funcionalidad. En muchos campos del software se ha ido implantando *middleware* que simplifica el desarrollo de nuevas aplicaciones en esas áreas. Este *middleware* proporciona contextos nítidos, estructuras de datos predefinidas, bloques muy depurados de código de uso frecuente, protocolos estándar de comunicaciones, mecanismos de sincronización, etc. Análogamente, a medida que el desarrollo de software para robots móviles ha ido madurando han ido apareciendo diferentes plataformas *middleware* (Utz *et al.*, 2002).

Hoy en día los fabricantes más avanzados incluyen plataformas de desarrollo para simplificar a los usuarios la programación de *sus* robots. Por ejemplo, ActivMedia ofrece la plataforma ARIA (ActivMedia, 2002) para sus robots Pioneer, PeopleBot, etc.; iRobot ofrecía Mobility (RWI, 1999) para sus B14 y B21; Evolution Robotics vende su plataforma ERSP; y Sony ofrece OPEN-R (Martín *et al.*, 2004) para sus Aibo.

Además de los fabricantes, muchos grupos de investigación han creado sus propias plataformas de desarrollo. Varios ejemplos son la suite de navegación CARMEN [8] (Montemerlo *et al.*, 2003) de Carnegie Mellon University, Orocos (Bruyninckx, 2001), `Player/Stage/Gazebo` (PSG) (Gerkey *et al.*, 2003; Vaughan *et al.*, 2003) , Miro (Utz *et*

[8] `http://www-2.cs.cmu.edu/~carmen`

al., 2002), JDE (Cañas and Matellán, 2002), MARIE (Côté *et al.*, 2004), etc..

El objetivo fundamental de estas plataformas es hacer más sencilla la creación de aplicaciones para robots. Hemos identificado varias características comúnes entre ellas: uniforman y simplifican el acceso al hardware, ofrecen una arquitectura software concreta y proporcionan un conjunto de bibliotecas o módulos con funciones de uso común en robótica que el cliente puede reutilizar para programar sus propias aplicaciones.

4.1 Abstracción del hardware

Normalmente las plataformas ofrecen un acceso a sensores y actuadores más abstracto y simple que el proporcionado por el sistema operativo. Por ejemplo, si se dispone de un robot Pioneer equipado con un sensor láser SICK, la aplicación puede acceder a sus medidas a través de las funciones de la plataforma ARIA o pedirlas y recogerlas directamente a través del puerto serie. Utilizando ARIA basta invocar un método sobre cierto objeto de una clase y la plataforma se encargará de mantener actualizadas las variables con las lecturas. Utilizando sólo el sistema operativo, la aplicación debe solicitar y recoger periódicamente las lecturas al sensor láser a través del puerto serie, y debe conocer el protocolo del dispositivo para componer y analizar correctamente esos mensajes de bajo nivel.

El acceso abstracto también se ofrece para los actuadores. Por ejemplo, en vez de ofrecer comandos de velocidad para cada una de las dos ruedas motrices de un robot Pioneer, la plataforma Miro (Utz *et al.*, 2002) ofrece una sencilla interfaz de V-W (velocidad de tracción y de giro) para la actuación motriz, la cual se encarga de hacer las transformaciones oportunas, de enviar a cada rueda las consignas necesarias para que el robot consiga esas velocidades comandadas de tracción y de giro.

Hattig et al. (Hattig *et al.*, 2003) apuntan que la uniformidad en el acceso al hardware es el primer paso para favorecer la reutilización de software dentro de la robótica. Esta característica está presente en varias plataformas, aunque cada una lo hace a su manera. En la plataforma `ERSP` de Evolution Robotics (figura 4) el acceso al bajo nivel recibe el nombre de *Hardware Abstraction Layer* (HAL), y en Miro, *Service Layer*. En `OPEN-R`, en Mobility y en `ARIA` la API de acceso a los sensores y actuadores viene dada por los métodos de un conjunto objetos. En `JDE` (Cañas and Matellán, 2002) y en PSG el acceso abstracto al hardware lo marca un protocolo entre las aplicaciones y los servidores.

4.2 Arquitectura software

La arquitectura software de la plataforma de desarrollo fija la manera concreta en la que el código de la aplicación debe acceder a las medidas de los sensores, ordenar a los motores, o utilizar una funcionalidad ya desarrollada.

Existen muchas alternativas de software para ello: llamar a funciones de biblioteca, leer variables, invocar métodos de objetos, enviar mensajes por la red a servidores, etc. Por ejemplo, la plataforma CARMEN (Montemerlo *et al.*, 2003) presenta interfaces funcionales, Miro usa la invocación de métodos de objetos distribuidos, TCA (Simmons and Apfelbaum, 1998) requiere del paso de mensajes entre distintos módulos, JDE requiere la activación de procesos y la lectura o escritura de variables. Esta apariencia de las interfaces depende de cómo se encapsule en cada plataforma la funcionalidad ya desarrollada.

Al escribirse dentro de la arquitectura software de la plataforma, el programa de la aplicación adopta una organización concreta y usualmente un lenguaje determinado. Así, se puede plantear como una colección de objetos (p.e. en OPEN-R), como un conjunto de módulos dialogando a través de la red (p.e. en TCA), como un proceso iterativo llamando a funciones, etc.. Hay plataformas muy cerradas, que obligan estrictamente a cierto modelo. Por ejemplo, en la plataforma RAI (RWI, 1999) los clientes han de escribirse forzosamente como un conjunto de módulos RAI (hebras sin desalojo), con ejecución basada en iteraciones. Otras arquitecturas software son deliberadamente abiertas, restringen lo mínimo, como PSG o CARMEN.

Las arquitecturas software de las plataformas más avanzadas establecen mecanismos concretos para que la aplicación se pueda distribuir en varias unidades concurrentes. El mecanismo multitarea que ofrece la plataforma envuelve y simplifica la interfaz del kernel subyacente para la multiprogramación, en la cual se apoyan siempre. Al igual que ocurre con la interfaz abstracta de acceso al hardware, la interfaz abstracta de multitarea facilita la portabilidad.

4.3 Funcionalidades de uso común

Además de las librerías sencillas de apoyo, como filtros de color y demás, estas funcionalidades engloban técnicas relativamente maduras, ya sea de percepción o de algoritmos de control: localización, navegación local segura, navegación global, seguimiento de personas, habilidades sociales, construcción de mapas, etc.

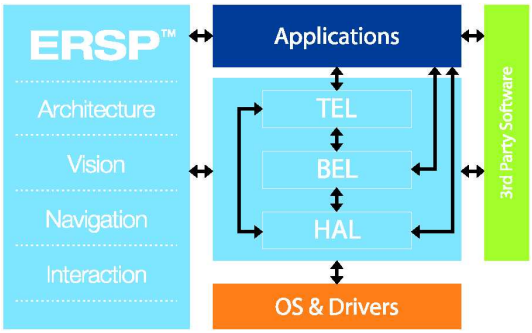

Figura 4. Módulos de navegación, visión e interacción en la plataforma ERSP©

La ventaja de integrarlas en la plataforma es que el usuario puede reutilizarlas, enteras o por partes, lo cual permite acortar los tiempos de desarrollo y reducir el esfuerzo de programación necesario para tener una aplicación. Al incluir funcionalidad común, el desarrollador no tiene que repetir ese trabajo y puede construir su programa reutilizándolas, concentrándose en los aspectos específicos de su aplicación. Además, suele estar muy probada, lo cual disminuye el número de errores en el programa final.

La forma concreta en que se reutilizan las funcionalidades depende nuevamente de la arquitectura software y de cómo se encapsulen en ella: módulos, objetos distribuidos, objetos locales, funciones, etc.

Los fabricantes suelen vender esas funcionalidades por separado o incluirlas como valor añadido de su propia plataforma. Por ejemplo, la plataforma ERSP (figura 4) incluye tres paquetes en su arquitectura básica: uno de interacción, otro de navegación y otro de visión. En el módulo de interacción se incluye el reconocimiento del habla y la síntesis de voz, para interactuar de modo verbal con su robot. En el módulo de navegación se incluye la construcción automática de mapas, la localización en ellos y su utilización para planificar trayectorias.

4.4 Arquitectura cognitiva

Se denomina *arquitectura cognitiva* de un robot a la organización de sus capacidades sensoriales y de actuación para generar un repertorio de comportamientos. Para comportamientos simples casi cualquier organización resulta válida, pero para comportamientos complejos se hace patente la necesidad de una buena organización cognitiva. De hecho, puede llegar a ser un factor crítico: con una buena organización sí se pueden generar ciertos comportamientos y con una mala no, porque la complejidad se vuelve inmanejable. A lo largo de la historia de la robótica han surgido diversas escuelas cognitivas o paradigmas para orientar la organización del sistema.

La división del comportamiento artificial en unidades reutilizableses una cuestión muy complicada. Hattig et al.(Hattig *et al.*, 2003) opinan que aún es demasiado pronto para buscar estándares ahí y recomiendan ceñir por ahora la estandarización exclusivamente al acceso a los sensores y actuadores.

La relación entre las arquitecturas cognitivas y las de software es múltiple. Las arquitecturas cognitivas se materializan en alguna arquitectura software, de manera que los comportamientos generados siguiendo cierto paradigma acaban implementándose con algún programa concreto. Las propuestas cognitivas más fiables cuentan con implementaciones prácticas relevantes en arquitecturas software concretas: deliberativas (e.g. SOAR), híbridas (e.g. TCA (Simmons and Apfelbaum, 1998), Saphira, etc.), basadas en comportamientos (subsunción, JDE, etc.), etc. Una buena arquitectura cognitiva favorece la escalabilidad de la plataforma.

Hay plataformas software debajo de las cuales subyace un modelo cognitivo, pero también hay otras en las que no. No obstante, unas arquitecturas software cuadran mejor con ciertas escuelas cognitivas que con otras. Los sistemas deliberativos clásicos cuadran con la programación lógica, con una descomposición funcional en bibliotecas (módulos especialistas) y con las aplicaciones monohilo con un sólo flujo iterativo de control (sensar-modelar-planificar-actuar). Por el contrario, los sistemas basados en comportamientos cuadran mejor con la programación concurrente, donde se tienen varios procesos funcionando en paralelo y que colaboran al funcionamiento global. Resulta natural asimilar cada unidad de comportamiento al concepto software de proceso, o incluso al de objeto activo.

4.5 Plataformas de software libre

Un gran número de las plataformas de desarrollo, principalmente las creadas por grupos de investigación, se publican con licencia de software libre, con la idea de contribuir al libre intercambio de conocimiento en el área y con ello al avance de la disciplina robótica.

Player/Stage/Gazebo (PSG) La plataforma PSG [9] fue creada inicialmente en la Universidad de South California (Gerkey *et al.*, 2003). Está formada por los simuladores Stage y Gazebo, y por el servidor Player al que se conecta el programa de

[9] `http://playerstage.sourceforge.net/`

la aplicación para recoger los datos sensoriales o comandar las órdenes a los actuadores. El soporte a robots variados y los simuladores que incorpora la convierten en una plataforma muy completa. Además, cuenta con una creciente comunidad de desarrolladores, muy activa, que continuamente añade nuevas capacidades y amplía el hardware soportado.

En PSG los sensores y actuadores se contemplan como si fueran ficheros (Vaughan *et al.*, 2003), dispositivos de modo carácter al estilo de Unix, que se manipulan con cinco operaciones básicas: abrir, cerrar, leer, escribir y configurar. Para usar un sensor, las aplicaciones tienen que abrir el dispositivo, configurarlo y leer de él, cerrándolo al final. De manera análoga se utilizan los actuadores. Cada tipo de dispositivo se define en PSG con una *interfaz*, de manera que los sensores sónar de algún robot en particular son instancias de la misma interfaz. El conjunto de posibles interfaces presenta una máquina virtual, con toda suerte de sensores y actuadores. El robot concreto instancia las interfaces relativas a los dispositivos realmente existentes.

PSG tiene un diseño cliente/servidor: las aplicaciones establecen un diálogo por TCP/IP con el servidor Player, que es el responsable de proporcionar las lecturas sensoriales y materializar los comandos de actuación. Además de permitir acceso remoto, este diseño proporciona a las aplicaciones construidas sobre PSG gran independencia de lenguaje y mínimas imposiciones de arquitectura. La aplicación puede escribirse en cualquier lenguaje, y con cualquier estilo, simplemente respetando el protocolo de comunicaciones con el servidor.

PSG se orienta principalmente a ofrecer una interfaz abstracta del hardware de robots y no a la identificación de bloques comunes de funcionalidad. No obstante, se puede incorporar cierta funcionalidad adicional con nuevos mensajes del protocolo, y servicios añadidos a Player. Por ejemplo, la localización probabilística se ha añadido como una interfaz más, `localization`, que proporciona múltiples hipótesis de localización. Esta nueva interfaz supera a la tradicional, `position`, que acarrea la posición estimada desde la odometría.

ARIA Otra plataforma de software libre muy utilizada es ARIA (*ActivMedia Robotics Interface for Applications*) (ActivMedia, 2002). ARIA está impulsada y mantenida por la empresa ActivMedia Robotics como interfaz de acceso al hardware de sus robots, pero se distribuye con licencia GPL y por tanto su código fuente está accesible. ARIA ofrece un entorno de programación orientado a objetos, que incluye soporte para la programación multitarea y para las comunicaciones a través de la red. Las aplicaciones han de escribirse forzosamente en C++. ARIA está soportada en Linux y en Win32 OS, por lo que una misma aplicación escrita sobre su API puede funcionar en robots con uno u otro sistema operativo. Es un claro ejemplo de portabilidad.

En el bajo nivel, ARIA tiene un diseño de cliente-servidor: el robot está gobernado por un microcontrolador que hace las veces de servidor. Ese servidor establece un diálogo a través del puerto serie con la aplicación, escrita utilizando ARIA, que actúa como cliente. En ese diálogo se envían al cliente las medidas de ultrasonido, odometría, etc. y se reciben las órdenes de actuación a los motores.

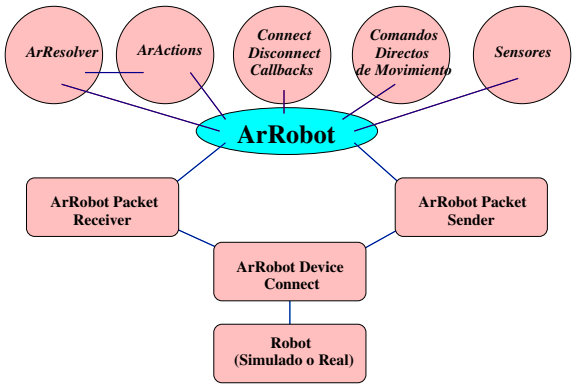

Figura 5. Estructura del API de ARIA

En cuanto al acceso al hardware, ARIA ofrece una colección de clases, que configuran una API articulada según muestra la figura 5. La clase principal `ArRobot` contiene varios métodos y objetos relevantes asociados. `Packet Receiver` y `Packet Sender` están relacionados con el envío y recepción de paquetes por el puerto serie con el servidor. Dentro de la clase `ArRangeDevices`, se tienen clases más concretas como `ArSonarDevice` o `ArSick` cuyos objetos contienen métodos que permiten a la aplicación acceder a las lecturas de los sensores de proximidad (sónares o escáner láser, respectivamente).

A diferencia de otras plataformas orientadas a objetos, los objetos de ARIA no son distribuidos, están ubicados en la máquina que se conecta físicamente al robot. No obstante, ARIA permite programar aplicaciones distribuidas utilizando `ArNetworking` para manejar comunicaciones remotas, que es un recubrimiento de los *sockets* del sistema operativo subyacente.

En cuanto a la multitarea, las aplicaciones sobre ARIA pueden programarse en monohilo o multihilo. En este último caso ARIA ofrece infraestructura tanto para hebras de usuario (`ArPeriodicTask`) como para hebras kernel (`ArThreads`). Las `ArThreads` son un recubrimiento de las `Linux-pthreads` o `Win32-threads`. Para resolver los problemas de sincronización y concu-

rrencia, ofrece mecanismos como los `ArMutex`, y las `ArCondition`.

ARIA contiene comportamientos básicos como la navegación segura, sin chocar contra los obstáculos. Pero no incluye funcionalidad común como la construcción de mapas o la localización, que se venden por separado. Por ejemplo, ActivMedia vende el paquete `MAPPER` para la construcción de mapas, `ARNL` (*Robot Navigation and Localization*) para la navegación y localización, y `ACTS` (*Color-Tracking Software*) para la identificación de objetos por color y su seguimiento.

Miro Miro [10] (Utz *et al.*, 2002) es una plataforma basada en objetos distribuidos para la creación de aplicaciones para robots, desarrollada en la Universidad de Ulm y publicada bajo licencia GPL. En cuanto a la distribución de objetos, Miro sigue el estándard CORBA, empleando la implementación TAO de CORBA, así como la librería ACE.

Miro consta de tres niveles: una capa de dispositivos, una capa de servicios y el entorno de clases. La capa de dispositivos (*Miro Device Layer*) proporciona interfaces en forma de objetos para todos los dispositivos del robot. Es decir, sus sensores y actuadores se acceden a través de los métodos de ciertos objetos, que dependen de la plataforma física. Por ejemplo, la clase `RangeSensor` define una interfaz para sensores como los sónares, infrarrojos o láser. La clase `DifferentialMotion` contiene métodos para desplazar un robot con tracción diferencial. La capa de servicios (*Miro Services Layer*) proporciona descripciones de los sensores y actuadores como servicios especificados en forma de CORBA IDL (*Interface Definition Language*). De esta manera su funcionalidad se hace accesible remotamente desde objetos que residen en otras máquinas interconectadas a la red, independientemente de su sistema operativo subyacente. El entorno de clases (*Miro Class Framework*) contiene herramientas para la visualización y la generación de históricos, así como módulos con funcionalidad de uso común en aplicaciones robóticas, como la construcción de mapas, la planificación de caminos o la generación de comportamientos.

Dentro de Miro la aplicación robótica tiene la forma de una colección de objetos, remotos o locales. Cada uno ejecuta en su máquina y se comunican entre sí a través de la infraestructura de la plataforma. Los objetos se pueden escribir en cualquier lenguaje que soporte el estándar CORBA, ya que Miro no impone ninguno en concreto, aunque ha sido enteramente escrita en C++.

[10] `http://smart.informatk.uni-ulm.de/MIRO`

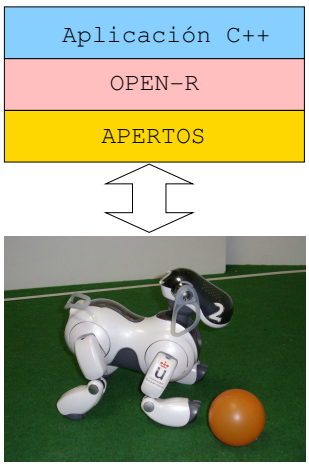

Figura 6. Arquitectura de software para el Aibo

5. ENTORNOS DE PROGRAMACIÓN

Dentro del mercado de robots móviles hay varios proveedores mayoritarios a nivel mundial: ActivMedia [11], iRobot, Robosoft [12], K-Team [13], Sony, LEGO, etc. Cada uno de estos fabricantes incluye un entorno de programación para sus robots, y lo actualizan a medida que surgen nuevos diseños o dispositivos. Por ejemplo, ActivMedia comercializa los robots Pioneer, AmigoBot, etc.; para los cuales están los entornos Saphira y ARIA. iRobot comercializaba los robots B21, Magellan, etc.; que utilizaban los entornos RAI, Beesoft, y recientemente Mobility. K-Team comercializa los robots Khepera y Koala, así como sus plataformas de programación y simulación. En esta sección vamos a analizar aquellos en los que tenemos experiencia más directa. Todos ellos confirman la organización del software de robots en tres niveles diferenciados: sistema operativo, plataforma y aplicaciones.

5.1 Aibo de Sony

El robot Aibo (figura 6) se vende como mascota, con un programa que gobierna su comportamiento para exhibir conductas de perro (seguir pelota, buscar hueso, bailar, etc.) y le permite incluso aprender con el tiempo. Desde el verano del año 2002 se abrió la posibilidad de programarlo, disparándose su uso como plataforma de investigación. El robot tiene como sensores principales una cámara situada en la cabeza, distintos sensores de contacto (apoyo en las patas, en la cabeza y lomo) y odometría en cada una de sus articulaciones. Como actuadores principales tiene las patas, el cuello y la cola.

El sistema operativo que regula los dispositivos hardware del Aibo es `Aperios` [14], un sistema

[11] `http://www.activmedia.com`
[12] `http://www.robosoft.fr/`
[13] `http://www.k-team.com/`
[14] Antes conocido como Apertos, y previamente como Muse

Robot	*Procesador*	*Sistema Operativo*	*Plataforma*	*Simulador*
LEGO RCX	micro Hitachi	BrickOS	no	legosim webots
Sony Aibo	RISC	Aperios	Open-R	webots
ActivMedia Pioneer	micro + laptop	Linux/MS-Windows	ARIA Saphira JDE PSG	PSG SRIsim JMR
iRobot (aka RWI) B21	PCs	Linux	Mobility CARMEN Miro	PSG
EyeBot	micro Motorola	ROBIOS	librerías	EyeSim
K-Team Kephera y Koala	micro Motorola	KT BIOS	SysQuake Korebot	webots
Robuter	micro	Linux/MS-Windows	librerías	no
Nomad	PC	Linux	librerías	no
ER1	PC	Linux/MS-Windows	ERSP	-

Cuadro 1. Tabla resumen de robots y sus entornos de programación

operativo de tiempo real basado en objetos. Sobre este sistema, Sony incluye la plataforma OPEN-R, que incorpora varios objetos específicos, los cuales establecen una interfaz de acceso al hardware del robot. Existen objetos para el acceso básico a las imágenes y las articulaciones (objeto `OVirtualRobotComm`), para el manejo de la pila TCP/IP (objeto `ANT`) y para el manejo del altavoz y micrófono (objeto `OVirtualRobotAudioComm`).

OPEN-R obliga a que la aplicación se escriba en C++, y su arquitectura software hace que consista en uno o varios objetos, que utilizan los servicios de los objetos básicos y que pueden intercambiarse datos entre sí (Martín *et al.*, 2004; Serra and Baillie, 2003). Adicionalmente, OPEN-R permite la multitarea y la programación orientada a eventos.

Para programar al Aibo se utiliza el PC como entorno de desarrollo y un compilador cruzado para el procesador RISC. El programa generado se escribe desde el PC en una barra de memoria (*memory stick*), y ésta se inserta en el propio perro para su ejecución a bordo.

5.2 RCX de LEGO

Este robot (figura 7) se vende como juguete creativo y plataforma educativa. Está compuesto por un procesador central (el ladrillo RCX) y un conjunto de piezas LEGO que se ensamblan mecánicamente para montar el cuerpo. Entre los sensores, que se conectan como piezas LEGO, los hay de luz, de choque, de rotación, etc. Los actuadores son principalmente motores. El procesador es un micro Hitachi H8/3297 de 8 bits, con 32 KB de memoria RAM, sobre el que se ejecuta un sistema operativo propio de LEGO (Ferrari *et al.*, 2001).

Hay varias alternativas para programarlo, y todas utilizan el PC como entorno de desarrollo, descargando el programa en el RCX a través del enlace de infrarrojos que posee (por puerto serie o USB). LEGO ofrece un entorno visual de programación orientado a los niños, llamado `RCX-code`, ya mencionado en la seccion 2.2. Otro entorno gráfico con el que se puede programar es Robolab [15], una adaptación de LabView, de National Instruments.

[15] `http://www.ni.com/company/robolab.htm`

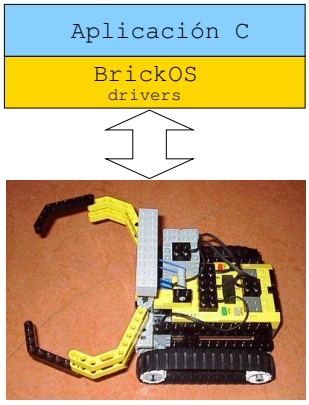

Figura 7. Arquitectura de software para el robot LEGO

Una posibilidad más de programación es el lenguaje NQC, creado por Dave Baum (Baum, 2000). Este lenguaje textual es una variante recortada de C que dispone de instrucciones propias para el acceso a sensores y actuadores. Tanto los programas en RCX-code como en NQC se compilan a código intermedio que se interpreta en tiempo de ejecución en el propio ladrillo.

Debido a la popularidad de este robot, algunos aficionados han desarrollado sistemas operativos alternativos que reemplazan al nativo de LEGO y exprimen mejor las posibilidades del hardware. El sistema operativo libre BrickOS [16] [17], desarrollado por Markus L. Noga (Nielsson, 2000), permite la programación del ladrillo en lenguaje C. En este caso, el compilador cruzado genera código no interpretado, que ejecuta directamente sobre el micro, por lo que se gana en eficiencia temporal. Del mismo modo, el sistema operativo LeJOS [18] permite la creación de aplicaciones en JAVA. Todos estos sistemas operativos, incluyendo al original de LEGO, ofrecen la capacidad de multitarea.

En todos los casos el entorno de programación es muy básico, de sistema operativo. Como es un robot muy limitado en cuanto a número y calidad de sensores y de actuadores, los comportamientos generables con él tienden a ser sencillos. Por eso no es necesaria una plataforma de desarrollo y las

[16] `http://brickos.sourceforge.net/`
[17] antes conocido como LegOS
[18] `http://lejos.sourceforge.net/`

aplicaciones se pueden hacer sin problemas programando directamente sobre la API del sistema operativo.

5.3 Pioneer de ActivMedia

El robot Pioneer es un robot de tamaño mediano, según se aprecia en la figura 8, con 26 cm de radio y 22 cm de alto. Consta de una base móvil con un par de motores, sensores de ultrasonido, odómetros y opcionalmente, sensores de contacto y sensor láser de proximidad. Tiene un microcontrolador que se encarga de recoger las medidas sensoriales y enviar las órdenes de movimiento a los motores, además de estar gobernado por un sistema operativo de bajo nivel (P2OS o AROS, según modelos). En cuanto a la arquitectura de procesadores, el montaje más frecuente del Pioneer incorpora un ordenador portátil a bordo, en el que se ejecutan las aplicaciones, y en él un sistema operativo generalista como Linux. Las aplicaciones dialogan con el microcontrolador a través del puerto serie utilizando un protocolo propio llamado SIP.

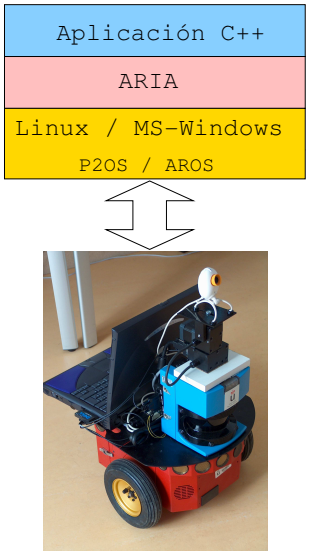

Figura 8. Arquitectura de software para el Pioneer

Para crear aplicaciones se dispone de la plataforma ARIA, que ya hemos descrito en la sección 4.5.

El fabricante incluye el simulador `SRIsim` acompañando a ARIA. Este simulador bidimensional permite emular un único Pioneer en un entorno estático y sus sensores más comunes: odometría, sónares y láser. Recientemente han incorporado a su software el simulador `Stage` de PSG, renombrándolo como `MobileSim`, y programando un recubrimiento para que funcione con el código ya existente de ARIA. Esto es un ejemplo nítido, a nivel de empresa, de reutilización de código en aplicaciones de robots, favorecido porque `Stage` es software libre.

6. PERSPECTIVAS

El mercado de los robots móviles tiene actualmente una pujanza creciente, cada vez hay más movimientos empresariales alrededor de la robótica. Los prototipos de investigación cada vez tienen mayor funcionalidad y se aproximan a su uso en el hogar como robots de entretenimiento o de servicio. Los robots Roomba, Aibo y LEGO son varios ejemplos de éxitos de ventas, superando el primero el millón y medio de unidades vendidas, y más de cien mil los dos últimos.

Con el paso de los años, el modo de programar los robots ha ido evolucionando y se ha ido articulando en tres niveles diferenciados: sistemas operativos, plataformas de desarrollo y aplicaciones concretas. En la sección 5 hemos descrito el entorno de programación de tres robots reales muy difundidos: Aibo, RCX y Pioneer. En todos ellos hemos encontrado esos tres niveles de software.

Otra tendencia confirmada es la extensión del ordenador personal (bien PC de sobremesa, bien PC portátil) como procesador principal de los robots. Más allá de los sistemas operativos dedicados, esta incorporación ha traído la irrupción en los robots de sistemas operativos generalistas, como Linux o MS-Windows, y el uso creciente de lenguajes de alto nivel con sus herramientas asociadas.

También hemos identificado varias líneas comunes entre plataformas: uniforman y simplifican el acceso al hardware, ofrecen una arquitectura software determinada e incorporan funcionalidad robótica común como la navegación, la construcción de mapas o la localización. A grandes rasgos, las plataformas tratan de fijar una base estable sobre la cual desarrollar aplicaciones robóticas, y eso supone un paso hacia la madurez de la programación de robots. Primero, favorecen la portabilidad de aplicaciones entre robots diferentes. Segundo, fomentan la reutilización de código, con lo cual disminuyen los tiempos de desarrollo de las nuevas aplicaciones. Y tercero, con su arquitectura software ofrecen una manera de organizar el código, lo cual permite afrontar la complejidad creciente de las aplicaciones de robots.

Hemos presentado una muestra representativa tanto de las plataformas de software libre (ARIA, PSG, Miro) como de las propietarias (ERSP, OPEN-R). Habrá que ver cuáles sobreviven de aquí a unos años, y eso dependerá enormemente de cuantos usuarios y desarrolladores sean capaces de atraer cada una de ellas.

La aparición de las plataformas de desarrollo ha supuesto un relevante paso hacia adelante en el camino de la programación de robots. Sin embargo, aunque la necesidad de estándares en robótica ha sido identificada por muchos autores, en la

actualidad existen *muchas* plataformas diferentes, tal vez demasiadas, reflejo de que no hay aún un estándar universalmente admitido.

REFERENCIAS

ActivMedia (2002). ARIA reference manual. Technical Report version 1.1.10. ActivMedia Robotics.

Baum, Dave (2000). *Dave Baum's definitive guide to LEGO MINDSTORMS*. APress.

Biggs, Geoffrey and Bruce MacDonald (2003). A survey of robot programming systems. In: *Proceedings of the 2003 Australasian Conference on Robotics and Automation* (Jonathan Roberts and Gordon Wyeth, Eds.).

Bruyninckx, Herman (2001). Open robot control software: the OROCOS project. In: *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-01)*. Vol. 3. Seoul (Korea). pp. 2523–2528.

Bräunl, Thomas (2003). *Embedded robotics*. Springer Verlag.

Cañas, José M. and Vicente Matellán (2002). Dynamic schema hierarchies for an autonomous robot. In: *Advances in Artificial Intelligence - IBERAMIA 2002* (José C. Riquelme y Miguel Toro Francisco J. Garijo, Ed.). Vol. 2527 of *Lecture notes in artificial intelligence*. pp. 903–912. Springer.

Côté, Carle, Dominic Létourneau, François Michaud, Jean-Marc Valin, Yannick Brosseau, Clément Raïevsky, Mathieu Lemay and Victor Tran (2004). Code reusability tools for programming mobile robots. In: *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-04)*. Sendai (Japan).

Ferrari, Mario, Giulio Ferrari and Ralph Hempel (2001). *Building robots with LEGO MINDSTORMS: The Ultimate Tool for Mindstorms Maniacs*. Syngress.

Firby, R. James (1994). Task networks for controlling continuous processes. In: *Proceedings of the 2nd International Conference on AI Planning Systems AIPS'94*. Chicago, IL (USA). pp. 49–54.

Gerkey, Brian P., Richard T. Vaughan and Andrew Howard (2003). The Player/Stage project: tools for multi-robot and distributed sensor systems. In: *Proceedings of the 11th International Conference on Advanced Robotics ICAR'2003*. Coimbra (Portugal). pp. 317–323.

Hattig, Myron, Ian Horswill and Jim Butler (2003). Roadmap for mobile robot specifications. In: *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*. Vol. 3. Las Vegas (USA). pp. 2410–2414.

Lozano Ortega, Miguel Ángel, Ignacio Iborra Baeza and Domingo Gallardo López (2001). Entorno Java para simulación y control de robots móviles. In: *Actas de la IX Conferencia de la Asociación Española Para la Inteligencia Artificial, CAEPIA 2001*. Gijón. pp. 1249–1258.

Martín, Francisco, Rafaela González-Careaga, José M. Cañas and Vicente Matellán (2004). Programming model based on concurrent objects for the AIBO robot. In: *Proceedings of the XII Jornadas de Concurrencia y Sistemas Distribuidos*. pp. 367–379.

Montemerlo, Michael, Nicholas Roy and Sebastian Thrun (2003). Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (CARMEN) toolkit. In: *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*. Vol. 3. Las Vegas (USA). pp. 2436–2441.

Nielsson, Stig (2000). Introduction to the LegOS kernel. Technical report.

RWI, Real World Interface (1999). Mobility Robot Integration software, User's guide. Technical Report version 1.1. IS Robotics, Inc.

Serra, Francois and Jean-Christophe Baillie (2003). Aibo programming using OPEN-R SDK. tutorial. Technical report. ENSTA (Francia).

Simmons, Reid and David Apfelbaum (1998). A Task Description Language for robot control. In: *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-98)*. Vol. 3. Victoria (Canada). pp. 1931–1937.

Sony (2003). OPEN-R SDK, Programmer's guide. Technical Report 20030201-E-003. Sony Corporation.

Utz, Hans, Stefan Sablatnög, Stefan Enderle and Gerhard Kraetzschmar (2002). Miro – middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures* **18**(4), 493–497.

Vaughan, Richard T., Brian P. Gerkey and Andrew Howard (2003). On device abstractions for portable, reusable robot code. In: *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*. Vol. 3. Las Vegas (USA). pp. 2121–2127.

Woo, Evan, Bruce A. MacDonald and Félix Trépanier (2003). Distributed mobile robot application infraestructure. In: *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*. Vol. 2. Las Vegas (USA). pp. 1475–1480.