INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Diseño y desarrollo de un compilador visual para la enseñanza de la robótica básica.

T E S I S
QUE PARA OBTENER EL GRADO DE:
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :
RONNY JOSÉ TONCHE GARCÍA

D I R E C T O R D E T E S I S :
ALFONSO GUTIÉRREZ ALDANA

MÉXICO, D. F. NOVIEMBRE 2010

Agradecimientos

A mi familia por todo su apoyo incondicional.
A Rosy por su comprensión y cariño.
Al profesor Aldana por su ayuda y tolerancia.

Índice general

Índice de figuras

Índice de tablas

Resumen

En los últimos años se ha ido incrementando el uso de la tecnología de forma acelerada, la robótica ha sido una de las ramas que más auge ha tenido en cuanto a la creación de herramientas para resolver una gran variedad de problemas. Sin embargo, para que ésta rama continué con ese crecimiento es importante contar con herramientas para enseñar a las nuevas generaciones de estudiantes la aplicación de la robótica, de ésta forma los estudiantes empezarán a temprana edad con la resolución de problemas sencillos, hasta que poco a poco con la ayuda de dichas herramientas lleguen a resolver problemas complejos, por ésta razón en el presente trabajo se desarrolló un compilador visual que servirá como herramienta para apoyar en la enseñanza de la robótica básica.

La herramienta propuesta en éste trabajo consiste en un compilador visual que permitirá al estudiante realizar programas fuentes para programar el comportamiento del robot. Dicho compilador presenta una interfaz gráfica que permite a los usuarios agregar bloques o iconos que representan a un elemento (sensores, actuadores, de control o lógicos) del conjunto de construcción robótico. Además, se creó un lenguaje de programación visual por medio de una gramática de disposición de imágenes para la implementación del compilador.

El análisis y diseño del compilador visual se realizó usando técnicas de ingeniería de software como son los diagramas de casos de uso, diagramas de clase, y una vez terminado se procedió a la codificación del mismo. Y para determinar que el comportamiento del software cumpliera con los requerimientos determinados al inicio de la programación se realizaron pruebas unitarias y pruebas de integración.

Por último se inicio el proceso de registro de derechos de autor del compilador visual en el Instituto Nacional de Derechos de Autor. Además, también se creó un manual de usuario y se realizó la instalación del compilador en la Centro de Estudios Científicos y Tecnológicos (CECyT) numero 9, Juan de Dios Bátiz, con la finalidad de que sea utilizado para el propósito con el que fue creado, enseñar robótica básica.

Abstract

In recent years has been increasing the use of technology, robotics has been one of the most important industries that have had major advances on the creation of components to solve a wide variety of problems. However, for this industry continue with that growth is important to have tools to teach new generations of students the application of robotics, so students begin at an early age with simple problems solving, until that step to step with the help of these tools the students achieve to solve complex problems, for this reason in this paper we present a visual compiler as a tool to support the teaching of basic robotics.

The tool proposed in this work is a visual compiler that will allow the student to design source programs to program the robot's behavior. This compiler provides a graphical interface that allows users to add blocks or icons that represent a element (sensors, actuators, control or logic) of robotic construction set. Besides, was created a visual programming language using a grammar of images available for the implementation of the compiler.

The visual compiler analysis and design was performed using software engineering techniques such as use case diagrams, class diagrams, and once completed we proceeded to codify it. And to determine that the behavior of the software complies with the requirements determined at the beginning of the programming, unit test and integration test was performed.

Finally, start the process of copyright registration of visual compiler at the Instituto Nacional de Derechos de Autor. In addition, we created a user manual and performed the installation of the compiler in the Center for Science and Technology Studies number 9, Juan de Dios Batiz, with the aim of it is being used for the purpose for which it was created, teach basic robotics.

1

Introducción

En la actualidad los robots son utilizados por el hombre para resolver diversos problemas, tales como: la exploración de superficies de planetas distintos a la tierra, la automatización de tareas mecánicas y repetitivas, como por ejemplo: el ensamblaje de carros, entre otros.

Uno de los campos que se ha visto favorecido con la robótica es la educación. La robótica educativa se centra en la creación de robots con el fin de desarrollar habilidades creativas, motoras y grupales, favoreciendo los procesos cognitivos de la persona que los construye.

En la actualidad existen conjuntos de construcción de robots, formados por una Unidad Central de Procesamiento (CPU, por sus siglas en inglés) programable(conocidos como ladrillos programables), sensores, actuadores y piezas mecánicas. Estos conjuntos de construcción son derivados de la robótica educativa, como los conjuntos: *ROBO Mobile Set* de Fischertechnik y el *Mindstorms NXT* de LEGO, que están creados para ser utilizados por personas de 10 años en adelante. Incluso la robótica educativa ha sido utilizada en niños de edad preescolar para ayudarles en su aprendizaje, en concreto el proyecto llamado LEGO/LOGO del Dr. Seymour Papert del laboratorio de medios del Instituto Tecnológico de Massachusetts. [Mindell]

Los conjuntos de construcción de robots han sido desarrollados de acuerdo a principios educativos derivados de las teorías del desarrollo cognitivo de Jean Piaget. Este enfoque indica que en todo proceso de aprendizaje, es el papel activo de quién aprende el que amplía su conocimiento a través de la manipulación y construcción de objetos.

Estos principios sugieren que la construcción de conjuntos de robots es adecuada como herramienta de aprendizaje. [Miglino]

Dicho de otra manera, los estudiantes aprenden por si mismos a construir su propio conocimiento a través del ambiente de aprendizaje y estimulan el uso de su imaginación, prueban sus habilidades para resolver problemas y tienen cooperación con sus compañeros de clases como se menciona en [Thomaz].

Los conjuntos de construcción mencionados anteriormente vienen con un programa, que es un entorno gráfico de desarrollo, que permite programar el comportamiento que tendrá el robot. El objetivo de esta tesis es crear un programa similar, un compilador visual (en este texto se tratará de manera similar el termino compilador visual o compilador gráfico), que permita programar el comportamiento de los robots que se puedan construir a partir de dicho conjunto de construcción.

El compilador visual permite programar dos conjuntos de construcción. El primero es el conjunto *Mindstorms NXT* de la compañía LEGO y el segundo es un conjunto de construcción creado en el Centro de Investigación en Computación (CIC) y resultado del trabajo de tesis titulado como "Diseño de un sistema de desarrollo para la enseñanza de la robótica".

1.1. Antecedentes

Existen en el mercado diversos tipos de compiladores gráficos que permiten programar robots comerciales, los más importantes son:

Mindstorms de Lego. Es el software que permite programar al robot *Mindstorms NXT* de Lego. Está basado en el programa LabView de la empresa National Instruments el cual es un entorno de programación gráfico que es utilizado para procesar y analizar datos recogidos en investigación.

ROBO Pro Software de Fischertechnik. Permite programar al conjunto de construcción ROBO Mobile Set de la misma compañía.

Visual Programing Language de Microsoft Robotics Studio. Permite programar una gran variedad de hardware como: *ROBO Mobile* Set de Fischertechnik, al robot *Mindstorms NXT*, el *MobileRobots Pioneer P3DX*, etc.

Existe también una aplicación llamada *RoboEduc* y fue creada por los investigadores Sarah Thomaz, Akynara Aglae, Carla Fernandez, Renata Pitta y otros más en la Universidad Federal Río Grande del Norte en Río Grande, Brasil. RoboEduc permite controlar al robot *Mindstorms RCX* de Lego.

Una descripción mas detallada de cada una de las aplicaciones aquí mencionadas se da en la sección 2.3.

1.2. Planteamiento del problema

En la actualidad muchas instituciones educativas tienen necesidad de utilizar nuevas herramientas como apoyo para la enseñanza. Dentro de esta nueva generación de herramientas se encuentran los conjuntos de construcción de robots, los cuales pueden ser utilizados para enseñar diversas materias como son: robótica básica, introducción a la programación y muchas otras dependiendo de la creatividad de los profesores.

Estas instituciones educativas utilizan conjuntos de construcción comerciales que se encuentran en el mercado como son el *Mindstorms* de Lego o el *ROBO Mobile* de Fischertechnik.

Uno de los objetivos de este trabajo de tesis es aportar a la creación de un conjunto de construcción de robots dividido en dos partes: la primera es crear los componentes hardware del conjunto y la segunda es crear el software que permitirá programar dicho conjunto. El segundo objetivo es el tema central de este trabajo de tesis.

El compilador gráfico, que es la interfaz de usuario para crear y compilar programas que definirán el comportamiento del robot, será construido con herramientas de software libre permitiendo que el compilador sea considerado como software libre también.

El software libre ofrece cuatro tipos de libertades para los usuarios del mismo:

- La libertad de ejecutar el programa, para cualquier propósito (libertad 0).
- La libertad de estudiar cómo trabaja el programa, y adaptarlo a sus necesidades (libertad 1). El acceso al código fuente es una condición necesaria.
- La libertad de redistribuir copias para que pueda ayudar al prójimo (libertad 2).
- La libertad de mejorar el programa y publicar sus mejoras, y versiones modificadas en general, para que se beneficie toda la comunidad (libertad 3). El acceso al código fuente es una condición necesaria. [Stallman]

En otras palabras el compilador gráfico, al ser software libre, da la libertad a los usuarios de ejecutar, copiar, distribuir, estudiar, cambiar y/o mejorar el software.

Con este enfoque de libertad del código fuente del compilador ciertas partes del programa pueden ser utilizadas para aprender, por ejemplo, cómo realizar conexiones entre dos dispositivos de bluetooth o usb utilizando sus respectivas pilas de protocolos en plataformas Linux.

1.3. Objetivos

1.3.1. General

El objetivo de este trabajo es crear un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés), que tendrá la funcionalidad de programar el comportamiento del CPU programable del conjunto de construcción.

1.3.2. Específicos

- Crear un lenguaje visual de programación, estructurado en bloques(iconos) configurables.
- Realizar el programa con herramientas de software libre.
- Desarrollar una interfaz que permita la comunicación entre la PC y el robot para enviar el programa al robot.
- Elaborar la documentación necesaria, como manuales y tutoriales, para que pueda ser introducido en el ámbito educativo.

1.4. Justificación y beneficios

La necesidad de ingresar a la robótica en el ámbito educativo para que los estudiantes puedan aprender acerca de la construcción de robots, hace que las instituciones educativas se interesen por adquirir conjuntos de construcción de robots como el *Mindstorms NXT* de Lego o el *ROBO mobile* set de Fischertechnik.

Por otra parte con este compilador se ofrecerá la posibilidad de que instituciones educativas puedan crear sus propios conjuntos de construcción, por medio de la documentación resultante de este trabajo en conjunción con la otra parte del proyecto que contendrá las herramientas necesarias para la creación del hardware. Esto permitirá que dichas instituciones o que cualquier persona puedan utilizar esta herramienta para introducirla en su ámbito educativo o personal para los fines que así les convengan.

Además las personas que utilicen el IDE tienen la posibilidad de ver, modificar y/o mejorar el código fuente del mismo para agregarle alguna funcionalidad extra o para realizar cualquiera de las libertades estipuladas por la Fundación de Software Libre.

Actualmente existe una herramienta de software libre llamada Not Quite C(NQC) [1], esta herramienta permite programar a los *LEGO Mindstorms* basados en compiladores

[1] NQC, fue desarrollada por Dave Baum y está basada en el lenguaje C.

convencionales en los cuales los programas para controlar el robot, son creados con editores basados en texto. Es por esto que hacer un compilador basado en gráficos ofrecerá un ambiente de aprendizaje más fácil para los usuarios y programadores del conjunto de construcción de Robots.

Los ambientes de programación visual proporcionan factores benéficos para los programadores: El primero es que pueden ser usados por diferentes tipos de usuarios, esto quiere decir que, tanto un programador experimentado puede realizar un programa, así como un novato. Otro de los factores es que en estos ambientes cada bloque configurable específico es mapeado a una porción de código objeto, de esta manera los programadores no tienen que escribir línea por línea las sentencias para realizar ciertas tareas, sino que simplemente arrastrando y soltando los bloques en el orden deseado pueden programar diversas tareas. Y como último enfoque se puede mencionar que la retroalimentación visual inmediata permite a los programadores escribir de manera más fácil un programa. [Kim]

El compilador gráfico, que es parte del conjunto de construcción de robots, propuesto en esta tesis, será usado inicialmente en el Centro de Estudios Científicos y Tecnológicos (Cecyt) No. 9 del Instituto Politécnico Nacional. El conjunto de construcción se utilizará para enseñar materias como robótica básica, física y matemáticas a los alumnos del área técnica de sistemas digitales.

1.5. Alcances y límites

El software que se propone en esta tesis estará basado, en su diseño y funcionalidad, en el compilador visual Mindstorms de la empresa Lego.

Se pretende probar el funcionamiento del compilador gráfico con el Mindstorms NXT de la empresa LEGO.

1.6. Organización de la tesis

La estructura de la tesis está formada por seis capítulos que se describen a continuación:

- **Capítulo 1.** Introducción: presenta el tema de la robótica educativa y cómo ésta apoya al proceso de aprendizaje. También se describen los objetivos y alcances de esta tesis.
- **Capítulo 2.** Material y métodos: en este capítulo se presentan conceptos básicos, un estudio de los compiladores gráficos existentes y la propuesta de nuestro

compilador gráfico.

- **Capítulo 3.** Definición del lenguaje de programación visual: se define de manera formal el lenguaje de programación visual que utiliza el compilador visual.
- **Capitulo 4.** Análisis y diseño del compilador visual: se describe el diseño general utilizado para desarrollar el compilador.
- **Capítulo 5.** Pruebas y resultados: se presentan las pruebas unitarias y de integración que se realizaron sobre el código del proyecto. También se presenta un ejemplo de funcionalidad entre el compilador gráfico y el Mindstorms NXT de LEGO.
- **Capítulo 6.** Conclusiones: se presentan las conclusiones a las que se llegó al término de la tesis con base en los resultados alcanzados y las mejoras posibles que se podrían realizar a futuro.
- Al final se incluyen dos anexos: El programa fuente del compilador visual y el manual de usuario.
- Por último se presentan las referencias de los artículos y documentos consultados durante el desarrollo de la tesis.

En el siguiente capítulo se presentan conceptos básicos de compiladores, se discuten los compiladores visuales de los conjuntos de construcción más utilizados y se propone nuestra solución.

2

Material y métodos

2.1. Introducción

En este capítulo se discutirán algunos conceptos básicos de los compiladores de lenguajes de programación, así como también los sistemas visuales y se analizarán a detalle los compiladores visuales más sobresalientes que se encuentran en el mercado. Sobre estos últimos se abordarán ventajas y desventajas, además de la descripción de cada uno de ellos.

Por último se planteará el compilador visual que se propone en esta tesis para ser utilizado por el Centro de Estudios Científicos y Tecnológicos(Cecyt) No. 9 del Instituto Politécnico Nacional y que tiene como finalidad enseñar materias como: robótica básica, física y matemáticas, a los alumnos del área técnica de sistemas digitales. También se describirán las características con las que se espera cumpla el software al final de su desarrollo.

2.2. Compiladores y sistemas visuales

Uno de los objetivos específicos de ésta tesis es crear un lenguaje de programación visual, descrito en el capítulo 3, que servirá para escribir programas que serán traducidos por el compilador visual a un lenguaje que entenderá la unidad de procesamiento del

conjunto de construcción. A continuación se dan a conocer algunos conceptos necesarios para entender los elementos que forman parte de este compilador.

Comenzaremos definiendo qué es un lenguaje de programación visual para entender porque al resultado de esta tesis se le llama compilador visual.

2.2.1. Lenguaje Visual de Programación (LVP)

Un programa en un lenguaje de programación "textual" se expresa como una cadena (unidimensional) en la cual los componentes(tokens) están concatenados para formar una sentencia cuya estructura y significado es descubierto por el análisis sintáctico y semántico, respectivamente. [Boshernitsan]

Por lenguaje visual nos referimos al uso sistemático de expresiones visuales para transmitir el significado de algo y estas expresiones se conocen como sentencias visuales. Un lenguaje visual de programación (LVP) es una distribución espacial, en dos dimensiones, de íconos (gráficos) que constituyen una sentencia visual. [Boshernitsan]

Los lenguajes visuales manejan iconos generalizados. Un icono generalizado es un objeto con una representación doble, una parte lógica (el significado) y una parte física (la imagen). [Chang]

Una visualización está formada por representaciones gráficas para ilustrar un dato, un programa, la estructura de un sistema complejo o el comportamiento dinámico de un sistema complejo. Finalmente, un sistema de programación visual, es un sistema computacional que soporta programación visual y visualización. [Chang]

2.2.2. Compilador "textual" y Compilador visual

Un compilador es un programa que lee un programa en un lenguaje, el lenguaje fuente y lo traduce a un programa equivalente en otro lenguaje, el lenguaje destino. El programa destino puede ser lenguaje máquina ejecutable o un lenguaje intermedio, que puede ser interpretado por algún programa intérprete, que es una especie de procesador de lenguajes. [Aho]

Con base en lo dicho anteriormente, el compilador visual propuesto es un sistema de programación visual que lee un programa escrito en lenguaje de programación visual y lo traduce a un programa equivalente en otro lenguaje destino.

2.2.3. Eficiencia de los LVPs

Dado que el conjunto de construcción y en específico el bloque básico de construcción tiene que ser programado por el usuario, si el programa tuviera que realizarse de la manera tradicional, es decir, por medio de un editor de texto y luego compilarlo, el usuario necesitaría tener ciertos conocimientos y/o habilidades para escribir el programa. Aquí es donde un lenguaje de programación visual realiza su principal aportación porque permite crear programas tanto a personas con conocimientos de programación como a usuarios novatos que se inician en ella.

La forma de programar con un LVP es arrastrando y soltando íconos en un ambiente de desarrollo gráfico, eliminando la cantidad de tiempo que el usuario gasta corrigiendo errores sintácticos. Y los programadores novatos también se evitan de lo frustrante que puede resultar buscar errores como olvidar poner un punto y coma(;) al final de una sentencia.

Sin embargo es necesario mencionar que no todo es ventaja en un LVP. La creación de programas simples y poco complejos es lo mejor para los LVP's ya que conforme los programas se hacen más grandes y complejos el diseño e implementación del programa es más difícil. [Ahern]

2.3. Estado del Arte

Como se mencionó anteriormente, en los compiladores visuales al construir un programa, los elementos íncluidos en éste son obtenidos de un lenguaje de programación visual, esto quiere decir que las sentencias o instrucciones son estructuras de íconos o gráficas enlazadas entre sí para definir el flujo de procesamiento del programa.

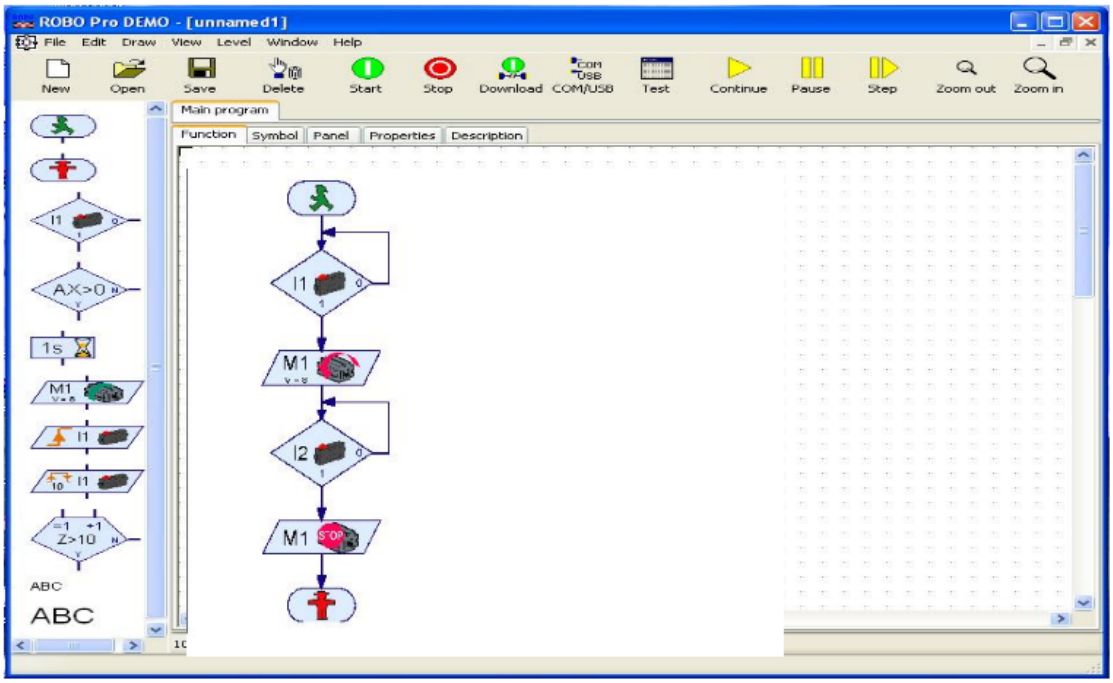

Figura 2.1: Interfaz gráfica del ROBO pro software.

Existen en el mercado diversos compiladores gráficos que permiten programar robots comerciales, los más conocidos y/o sobresalientes son:

2.3.1. ROBO Pro Software de Fischertechnik

Permite programar al robot ROBO Mobile Set de la misma compañía, su programación se basa en diagramas de flujo como se muestra en la Figura 2.1. Los módulos de programa son elementos del diagrama que están interconectados gráficamente.

Ventajas del ROBO Pro Software.

- El ROBO Mobile Set viene con un manual que muestra cómo construir siete diferentes modelos de robots. Entre ellos están un robot detector de líneas, un detector de obstáculos y un buscador de luz. Por lo cual el usuario puede aprender a usar rápidamente el conjunto de construcción.

- La interfaz del programa es parecida a un diagrama de flujo por lo que la programación puede ser intuitiva para las personas que tengan conocimientos de diagramas de flujo.

- El programa puede ser enviado a la Unidad de Procesamiento, por medio del USB o vía serial (RS-232).

Desventajas del ROBO Pro Software.

- Sólo permite programar al conjunto de construcción ROBO Mobile Set de la misma empresa.

- Existe poca documentación en línea y libros.

- No es intuitivo para personas que no tengan conocimiento de diagramas de flujos.

- Al ser una empresa alemana la propietaria del conjunto de construcción, la mayoría de la documentación existente está escrita en alemán, muy poca está traducida al inglés y casi nada al español.

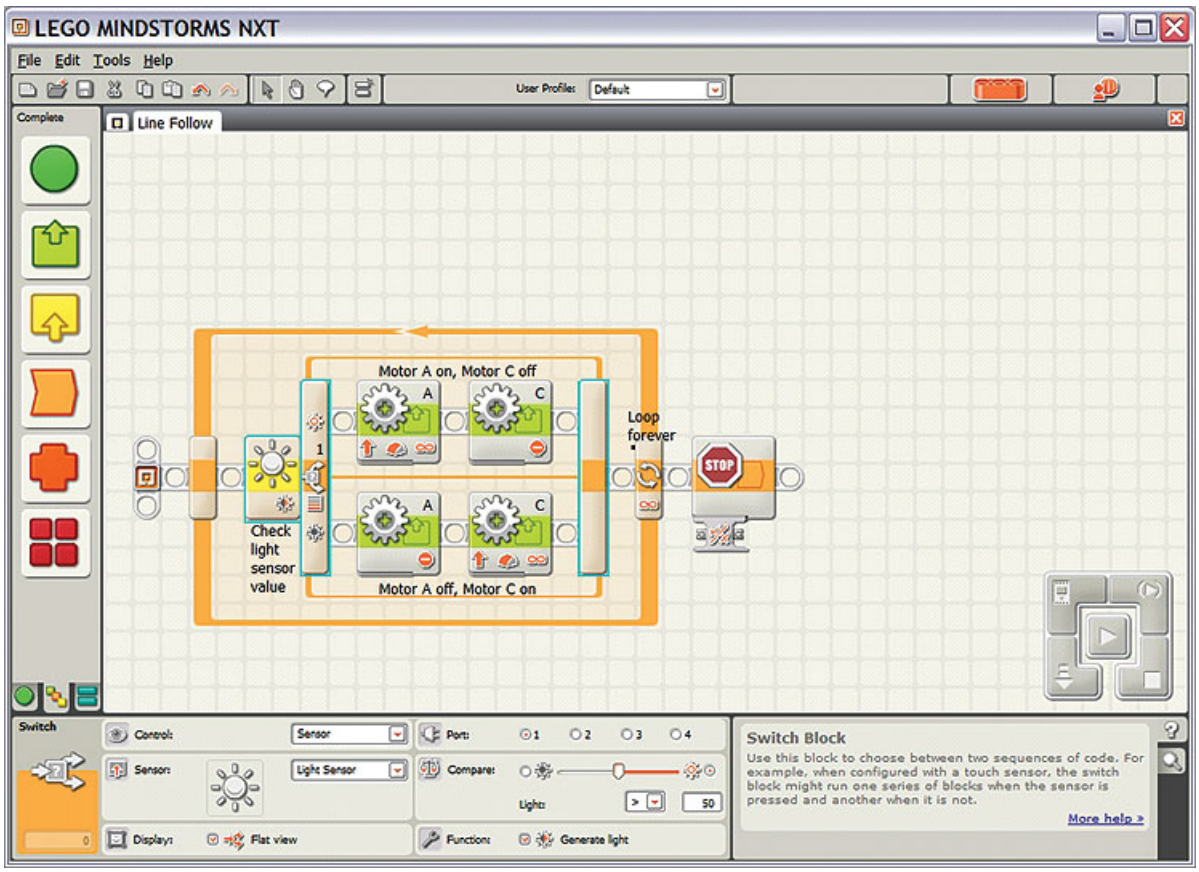

Figura 2.2: Interfaz gráfica del Mindstorms de LEGO.

2.3.2. Mindstorms de LEGO

Éste software permite programar al robot Mindstorms NXT de Lego. Esta basado en el programa LabView de la empresa National Instruments. LabVIEW es un entorno de programación gráfica usado por miles de ingenieros e investigadores para desarrollar sistemas sofisticados de medida, pruebas y control usando íconos gráficos e intuitivos y cables que parecen un diagrama de flujo. LabVIEW ofrece una integración incomparable con miles de dispositivos de hardware y brinda cientos de bibliotecas integradas para análisis avanzado y visualización de datos. La Figura 2.2 muestra la interfaz de ésta aplicación.

Ventajas de la aplicación Mindstorms.

- Extensa documentación de ayuda para usuarios del conjunto de construcción Mindstorms y de otras empresas que no son propietarias del conjunto de construcción.
- Existe una versión de la aplicación que fue creada exclusivamente para la educación, ya que la empresa Lego tiene una división de apoyo a la educación.

- También tiene una versión de la aplicación para utilizar en línea (http://www.robolabonline.com).

- Interfaz amigable de uso intuitivo.

- Aplicación hecha para ser instalada y ejecutada en la plataforma Windows 95, 98, ME, NT, 2000, XP y sistemas Macintosh 9.0 o superior.

- El programa puede ser enviado a la Unidad de Procesamiento, conocido como NXT, por medio del USB o por Bluetooth.

Desventajas de la aplicación Mindstorms.

- Sólo permite programar al conjunto de construcción Mindstorms de la misma empresa, LEGO.

- La aplicación en linea tiene costo.

2.3.3. Visual Programing Language de Microsoft Robotics Studio

El Visual Programming Language(VPL) de Microsoft es un entorno de desarrollo de aplicaciones basado en el modelo de programación de flujo de datos gráfico. Es decir, en lugar de ser una serie de instrucciones que se ejecutan secuencialmente, un programa de flujo de datos es como una serie de bloques que hacen sus tareas asignadas conforme llegan los datos.

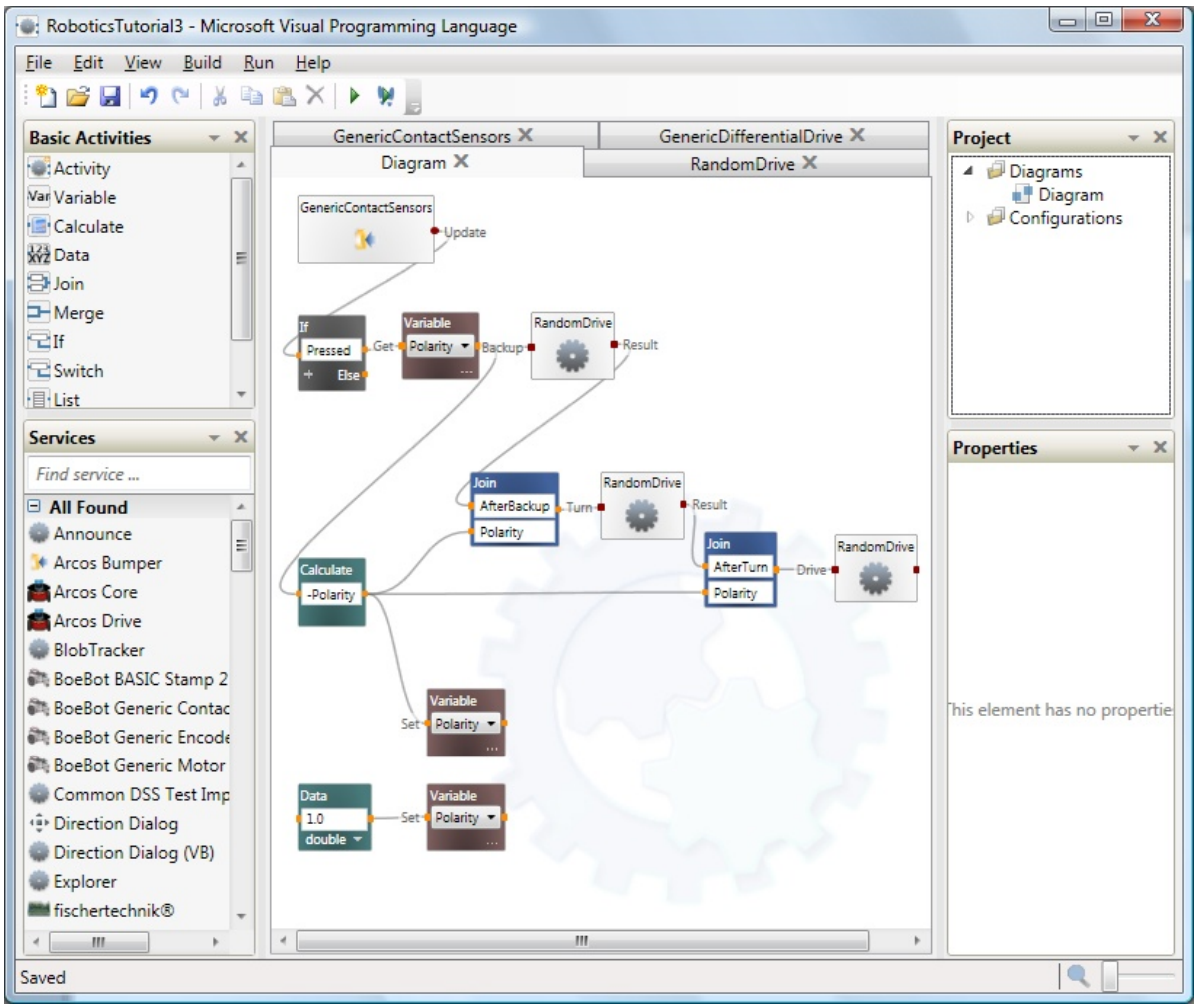

Figura 2.3: Interfaz gráfica del VPL de Microsoft.

El VPL de Microsoft permite programar una gran variedad de conjuntos de construcción de robots como: ROBO Mobile Set de Fischertechnik, al robot Mindstorms NXT, el MobileRobots Pioneer P3DX, etc. En la Figura 2.3 se muestra la interfaz de esta aplicación.

Ventajas de la aplicación VPL de Microsoft.

- Permite programar a una gran diversidad de conjuntos de construcción.
- Variada documentación en línea y libros de la aplicación.
- La edición Express de VPL está disponible de forma gratuita tanto para uso no comercial y comercial. Sin embargo, la licencia no permite la redistribución.

Desventajas de la aplicación VPL de Microsoft.

- Debido al modelo basado en flujo de datos la programación no es tan intuitiva para el usuario.
- La interfaz del mismo programa es complicada debido a que permite programar una gran cantidad de conjuntos de construcción de robots.
- Es necesario comprar el conjunto de construcción o el robot que se deseé programar por separado.

El siguiente software presentado no es comercial, es una herramienta pedagógica creada en Brasil para la enseñanza de la robótica.

2.3.4. RoboEduc

El software llamado *Roboeduc* es definido en [Thomaz] como una herramienta pedagógica para soportar robots educacionales y es enfocado en alumnos de nivel primaria. En este artículo se muestran resultados pedagógicos positivos en una comparación entre alumnos que utilizaron el *RoboEduc* contra otros que no lo usaron, obteniendo los primeros resultados favorables en materias como Ciencias, Matemáticas y Artes.

Figura 2.4: Interfaz gráfica del software RoboEduc, programación gráfica.

Figura 2.5: Interfaz gráfica del software RoboEduc, programación textual.

Las características de *RoboEduc* son:

- Permite controlar a un robot *Lego Mindstorms RCX* de manera remota y guarda la ruta en el disco duro de la PC para que ésta sea recuperada en cualquier otro momento. La Figura 2.4 muestra la interfaz de usuario para realizar esta operación.
- Permite programar al robot *Mindstorms RCX* con un lenguaje de programación textual creado especialmente para los alumnos con sentencias parecidas al portugués como se muestra en la Figura 2.5.

2.4. El compilador visual propuesto

Como se menciona en [Kim] el ambiente de programación visual del *Lego Mindstorms* permite a los usuarios novatos sin habilidades de programación o conocimientos de robótica poder aprender fácilmente a usar el robot *Mindstorms* a causa del ambiente visual que se utiliza en el software. También se hace una comparación entre el Microsoft Robotic Studio y el software del Mindstorms en la cual se dice que el Robotic Studio es más flexible que el Mindstorms pero que este último es más simple y eficiente para

programar el NXT y por lo tanto el software de Lego es un buen punto de inicio para aprender sobre robótica y programación visual.

Con base en el estudio de los compiladores visuales existentes en el mercado se ha decidido que el compilador visual propuesto en esta tesis se basará en el software Mindstorms de Lego, ya que es el más simple de usar para los usuarios y además por que el conjunto de construcción de Lego Mindstorms es muy popular en cuanto a su uso en varias instituciones educativas como se muestra en [Zhang].

A continuación se presenta las características que se esperan alcanzar al finalizar el compilador visual.

- Interfaz amigable e intuitiva para el usuario.
- Al ser software de código abierto el usuario tiene la posibilidad de ver, estudiar, analizar y modificar el código fuente para los fines que le convenga.
- Los programas creados con el compilador propuesto serán probados en el Mindstorms NXT.
- El envío del programa desde la PC a la Unidad de Procesamiento del conjunto de construcción sera por vía USB y Bluetooth.
- La documentación de la aplicación será creada especialmente para los alumnos del Centro de Estudios Científicos y Tecnológicos No. 9.

2.5. Resumen

En este capítulo se dieron a conocer algunos conceptos de compiladores visuales y se realizó un estudio de los compiladores visuales más utilizados para programar conjuntos de construcción comerciales.

En el siguiente capítulo se presenta la definición formal del lenguaje de programación visual que utilizarán los usuarios del compilador visual para crear programas fuentes que, una vez compilados, serán enviados al CPU del robot quién tomará las sentencias del programa objeto como instrucciones para realizar alguna tarea específica.

3

Lenguaje de programación visual

En el presente capítulo se define de manera formal el lenguaje de programación visual que maneja el Compilador visual propuesto en este trabajo, para formular dicha definición se utiliza una gramática de disposición de imágenes (Picture layout grammar).

En la segunda sección del captulo se muestra la manera en que es traducido el lenguaje de programación visual a dos lenguajes objetos diferentes.

3.1. Definición del lenguaje de programación visual

La gramática de disposición de imágenes (GDI), según [Golin], es un tipo de gramática que puede ser utilizada para especificar la sintaxis de lenguajes visuales, del mismo modo que las gramáticas libres del contexto son utilizadas para definir lenguajes de programación textual.

Los lenguajes de programación visual se clasifican en tres grupos [Golin], de acuerdo a las expresiones visuales utilizadas: lenguajes basados en íconos, lenguajes basados en formas y lenguajes diagrama. El lenguaje que maneja el compilador visual está basado en íconos.

En el análisis de los lenguajes de programación visual y los lenguajes de programación textual, la estructura de los elementos que lo conforman son similares, ya que éstos están compuestos por elementos léxicos terminales y no terminales. La principal diferencia es

que en los lenguajes de programación visual los elementos terminales en el programa fuente están ordenados en un espacio de dos dimensiones a diferencia de los lenguajes textuales que sólo manejan una dimensión.

3.1.1. Operadores de producción predefinidos

En la GDI las producciones corresponden a operadores de composición de imágenes y los atributos de los símbolos gramaticales representan la información espacial para un elemento de imagen. Cada símbolo gramatical tiene cuatro atributos xIzquierda, yAbajo, xDerecha, yArriba, que describen los puntos inicial y final del área ocupada por el símbolo que forman un rectángulo. Existen también restricciones en la producciones que sirven para especificar la relación entre los símbolos de una composición particular.

Para crear las producciones del lenguaje del compilador visual, se crearon operadores de producción predefinidos:

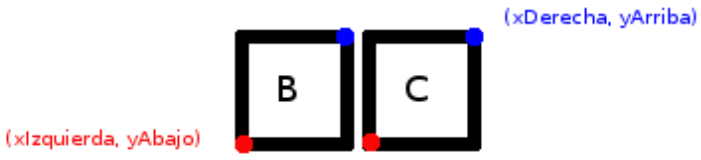

Figura 3.1: Operador predefinido Sigue.

Operador sigue. $A \rightarrow sigue\{C, B\}$ para definir que el elemento C sigue después del elemento B como en la Figura 3.1 y se declara formalmente de la siguiente manera

$A \rightarrow \{C, B\}$
donde:
C.xIzquierda > B.xDerecha,
C.yAbajo == B.yAbajo,
C.yArriba == B.yArriba.

Figura 3.2: Operador predefinido Contiene.

Operador contiene. $A \rightarrow contiene\{B, C\}$ para definir que el elemento B contiene al elemento C como en la Figura 3.2 y se declara formalmente de la siguiente manera:

$A \rightarrow \{B, C\}$
donde:
B.xIzquierda < C.xIzquierda,
B.xDerecha > C.xDerecha,
B.yAbajo == C.yAbajo,
B.yArriba == C.yArriba.

Con estas dos producciones predefinidas se puede iniciar la creación de las producciones del lenguaje visual.

3.1.2. Producciones

Las producciones que definen de manera formal el lenguaje que acepta el compilador visual son las siguientes:

(1) $INICIO \rightarrow sigue(imagenInicio, SENTENCIA)$
(2) $SENTENCIA \rightarrow sigue(BLOQUE, SENTENCIA)|BLOQUE$
(3) $BLOQUE \rightarrow imagenBloque$
(4) $BLOQUE \rightarrow REPETIR|SWITCH$
(5) $REPETIR \rightarrow contiene(imagenRepetir, SENTENCIA)$
(6) $SWITCH \rightarrow contiene(imagenSwitch, SENTENCIA)$

Los elementos terminales de las producciones están escritos en minúscula y los elementos no terminales están escritos en mayúscula.

Con estas producciones los usuarios del compilador podrán crear programas fuentes desde la ventana de edición. En la ventana de edición del compilador se podrán crear diagramas que implementan un modelo para hacer estos diseños dirigidos por la sintaxis, esto es, el compilador desde su interfaz gráfica no permite al usuario crear sentencias que no cumplan con las reglas de las producciones, y por lo tanto cuando el usuario termine de realizar su programa sólo será necesario realizar la traducción al lenguaje de bajo nivel. Es por este motivo que el compilador no cuenta con un analizador sintáctico del programa fuente.

Figura 3.3: Elementos terminales del lenguaje.

En la Figura 3.3 se muestran los elementos terminales de manera visual, los cuatro elementos terminales mostrados en la parte superior de la imagen son elementos terminales especiales que se mencionan en las producciones 1, 5 y 6. El elemento imagen Flecha es un componente léxico que es ignorado por el compilador, teniendo una función similar al espacio en blanco en los lenguajes de programación textuales, las flechas sólo sirven para mostrar cierta claridad en cuanto al flujo de las sentencias del programa fuente pero no forman parte de las producciones.

Los elementos que se encuentran dentro del cuadro son bloques sencillos que son representados en el lenguaje visual de manera general como un componente léxico llamado imagenBloque y que es mencionado en la producción 3.

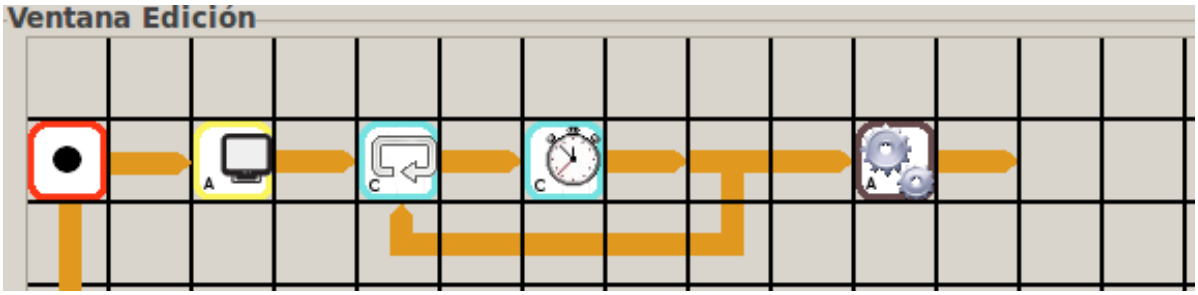

Figura 3.4: Sentencia del lenguaje visual.

El ejemplo mostrado en la Figura 3.4 es una sentencia visual, creada con las producciones, en la que se manda al conjunto de construcción a iniciar la marcha de sus motores y esperar a que el sensor de tacto sea presionado para parar los motores y terminar el programa.

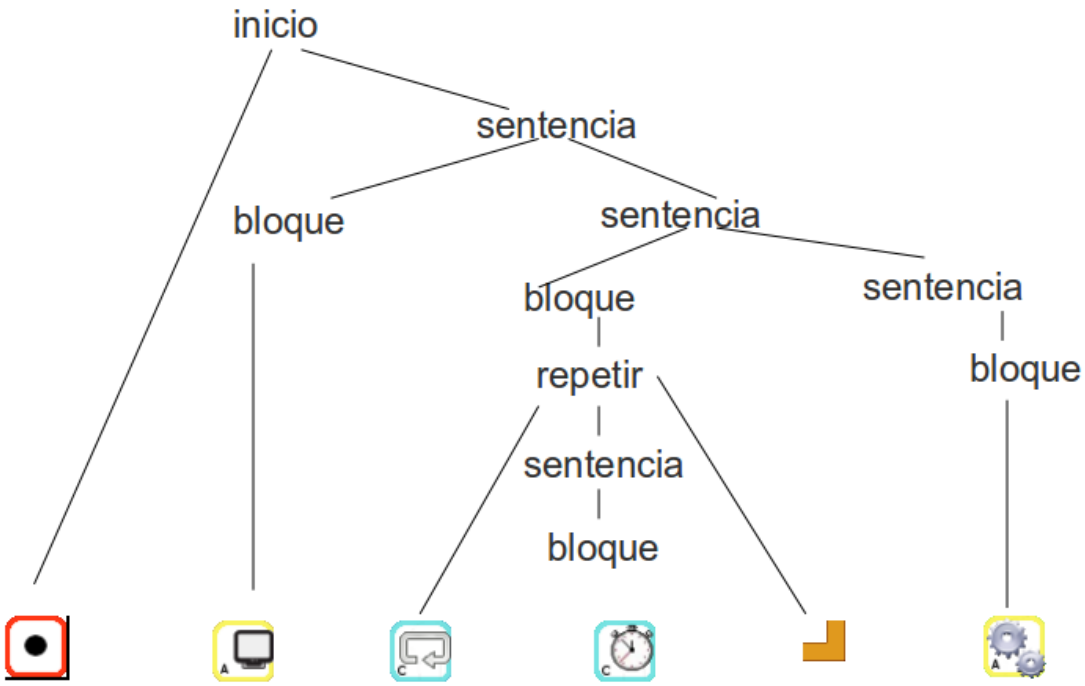

Figura 3.5: Árbol de análisis sintáctico de una sentencia.

En la Figura 3.5 se muestra un ejemplo de un árbol de análisis sintáctico para la sentencia de la Figura 3.4.

3.2. Traducción

A continuación se muestra el proceso de traducción de un programa fuente con sentencias visuales utilizando el lenguaje visual descrito anteriormente en este capítulo.

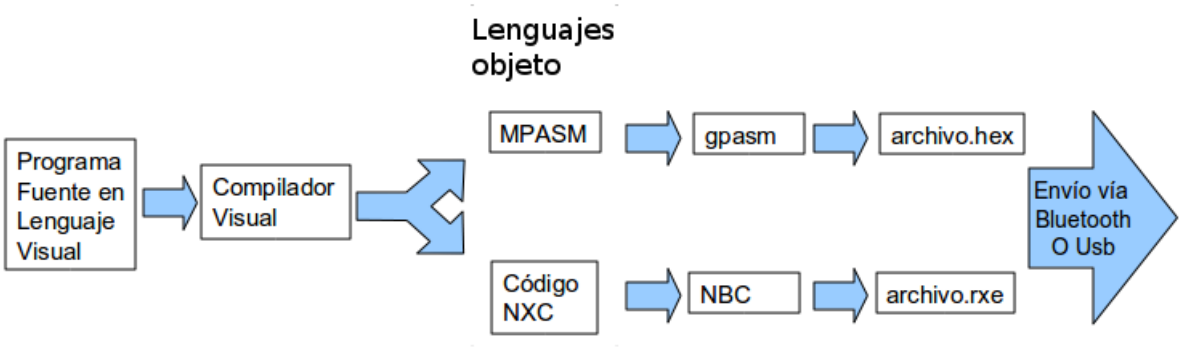

Figura 3.6: Secuencia de traducción de un programa en el compilador visual.

En la Figura 3.6 se muestra el proceso de traducción de un programa fuente (parte izquierda de la imagen) hasta obtener los archivos finales (parte derecha de la imagen) que son enviados al CPU del robot. En el compilador visual cada uno de los bloques que forman parte de una sentencia visual son traducidos a un porción de código en lenguaje ensamblador(MPASM) o en lenguaje NXC, dependiendo del CPU objetivo al que se enviará el archivo final (Mindstorms o PIC18F4550).

Los archivos traducidos tienen que pasar por otros compiladores externos que son llamados desde el compilador visual. El archivo en lenguaje ensamblador es compilado por el programa *gpasm* y se obtiene un archivo con extensión .hex, y el archivo en lenguaje NXC es compilado por el programa *nbc* del cual se obtiene un archivo con extensión .rxe.

Ejemplos de archivos creados por el compilador visual en los lenguajes ensamblador y nxc se muestran a continuación.

Código nxc: ejemplo.nxc

```
task tarea1()
{
RotateMotorEx(OUT_A, 75, 1800, 0, true, true);
}
task tarea2()
{
}
task tarea3()
{
}
task main()
{
```

```
Precedes(tarea1,tarea2,tarea3);
}
```

Código ensamblador: ejemplo.asm

```
LIST p=18f4550
INCLUDE <p18f4550.inc>

CONFIG FOSC = XT_XT ;SE CONFIGURA PARA TRABAJAR CON OSCILADOR A CRISTAL DE  4 MHZ
CONFIG  LVP = OFF ;SE DESHABILITA LA PROGRAMACION DE BAJO VOLTAJE Y RB5 COMO PIN I/O
CONFIG WDT = OFF        ;HW Disabled - SW Controlled

;************************DEFINICION DE VARIABLES**************
CBLOCK 0x00
DUTY
CONMOT1
VUELTAS
ENDC

ORG 0x00
GOTO INICIO

;*********************SUBRUTINA PARA QUE TRABAJE EL MOTOR 1*********
MOTOR1 ;{
;*******CONFIGURACION DEL MODULO CCP1 PARA TRABAJAR EN MODO PWM*****
CLRF CCP1CON ;El modulo CCP esta apagado
CLRF TMR2 ;TMR2 apagado
MOVLW 0x1E ;Cargamos el registro PR2
MOVWF PR2 ;con el valor 0x1E para obtener una frec de 2 KHz en el PWM
MOVLW 0x02 ;Asignamos el preescalador 1:16 al timer2
MOVWF T2CON
BCF TRISC,2 ;RC2 como salida del PWM1
BCF TRISD,4 ;RD4 como salida para el enable del puente L293 del motor1
BCF TRISD,2 ;RD2 como salida para la direccion del motor
MOVLW 0x0C ;Habilitamos como PWM el CCP1
MOVWF CCP1CON
CLRF PORTD
;******CONFIGURACION DEL TIMER 0 COMO CONTADOR Y CONTROL DEL MOTOR1*****
MOVLW 0xF2 ;Cargamos el registro T0CON para configurar el timer 0
MOVWF T0CON ;con entrada por el RA4 del decoder proveniente del motor
BSF TRISA,4 ;RA4 como entrada
CLRF STATUS
```

```
;************INICIA EL PROGRAMA************************************
CLRF TMR0L ;Ponemos a ceros el registro del timer0
BSF T2CON,TMR2ON ;Inicia el incremento del TIMER2
BSF PORTD,4 ;a 1 el enable del L293 motor en movimiento
BSF PORTD,2

LOOP
CALL DIREC ;llamamos a la rutina de direccion
MOVF TMR0L,W ;Pasamos el valor del registro TMR0L al registro W
XORWF VUELTAS,W ;restamos el valor de la variable vuelta de registro
BTFSS STATUS,2 ;verificamos si es UNO
GOTO  LOOP ;Si no es cero regresamos a verificar el valor de TMR0L
CLRF CCP1CON ;Apagamos el CCP como pwm
BCF PORTC,2 ;y ponemos a ceros los dos in del puente para parar el motor
BCF PORTD,2
BCF STATUS,2
CLRF TMR0L ;Ponemos a ceros el registro del timer0
BCF PORTD,4 ;desabilitamos el enable del L293 para parar el motor
RETURN
;*****************RUTINA PARA SABER LA DIRECCION*****************
DIREC BTFSC CONMOT1,5 ;REVISAMOS QUE DIRECCION SE PROGRAMAO EN EL BIT 5
GOTO REVER
BCF PORTD,2 ;SE PONE A "0" RD2 PARA EL SENTIDO ADELANTE
MOVF DUTY,W ;CARGAMOS EL VALOR DE DUTY EN REG W
MOVWF CCPR1L ;Y LO PASAMOS AL REGISTRO QUE CONTROLA EL PWM
RETURN
REVER BSF PORTD,2 ;se pone a "1" RD2 para el sentido de reversa
MOVF DUTY,W ;MOVEMOS EL VALOR DE  DUTY AL REG W
SUBLW 0x20 ;RESTAMOS EL VALOR DE DUTY A LA LITERAL
MOVWF CCPR1L ;Lo cargamos en el registro para que el PWM lo ajuste
RETURN

INICIO ;************PROGRAMA PRINCIPAL*************************
;*********CARGAMOS LOS VALORES DE LAS VARIABLES ************

BCF CONMOT1,5
MOVLW 0x75
MOVWF DUTY
MOVLW 0x5
MOVWF VUELTAS
CALL MOTOR1
GOTO $
END
```

En los dos archivos anteriores se muestra el código generado por el compilador visual para una sentencia que contiene un bloque el tipo "motor"que fue configurado para dar 5 vueltas. El primer código, escrito en nxc, esta basado en el lenguaje C y el segundo ejemplo es código ensamblador que entenderá el microcontrolador PIC18F4550.

Por último, un archivo final con extensión .rxe es enviado por el compilador visual al CPU del *Mindstorms* vía Bluetooth o USB y el archivo con extensión .hex es enviado al PIC18F4550 vía USB, aunque es necesario mencionar que solo uno de los archivos finales es creado a la vez y no los dos al mismo tiempo y por lo tanto solo es enviado a uno de los dos diferentes CPUs, dependiendo de la opción elegida por el usuario del compilador visual.

3.3. Resumen

En este capítulo se definió de manera formal el lenguaje de programación visual que maneja el compilador, para lo cual se utilizó el gramática de disposición de imágenes. Además se mostró un ejemplo de una sentencia visual y el proceso de traducción al lenguaje objeto.

En el siguiente capítulo haremos el análisis y diseño del compilador visual por medio del lenguaje de modelado unificado.

4

Análisis y diseño del Compilador Visual

El lenguaje de programación elegido para programar el compilador visual es C++, que es un lenguaje orientado a objetos. Por consiguiente el análisis y diseño del compilador visual es representado por medio del lenguaje de modelado unificado, ya que es un lenguaje que se adecua a la modelización de sistemas orientados a objetos.

Por otra parte se utilizará una arquitectura de la aplicación dividida en dos capas. La capa de presentación es la que se encarga de la interacción entre la aplicación y el usuario y la capa de lógica de negocio es la encargada de manejar todas las operaciones de administración de los programas fuentes, comunicación entre la PC y el robot, y de la traducción del programa fuente al programa objeto.

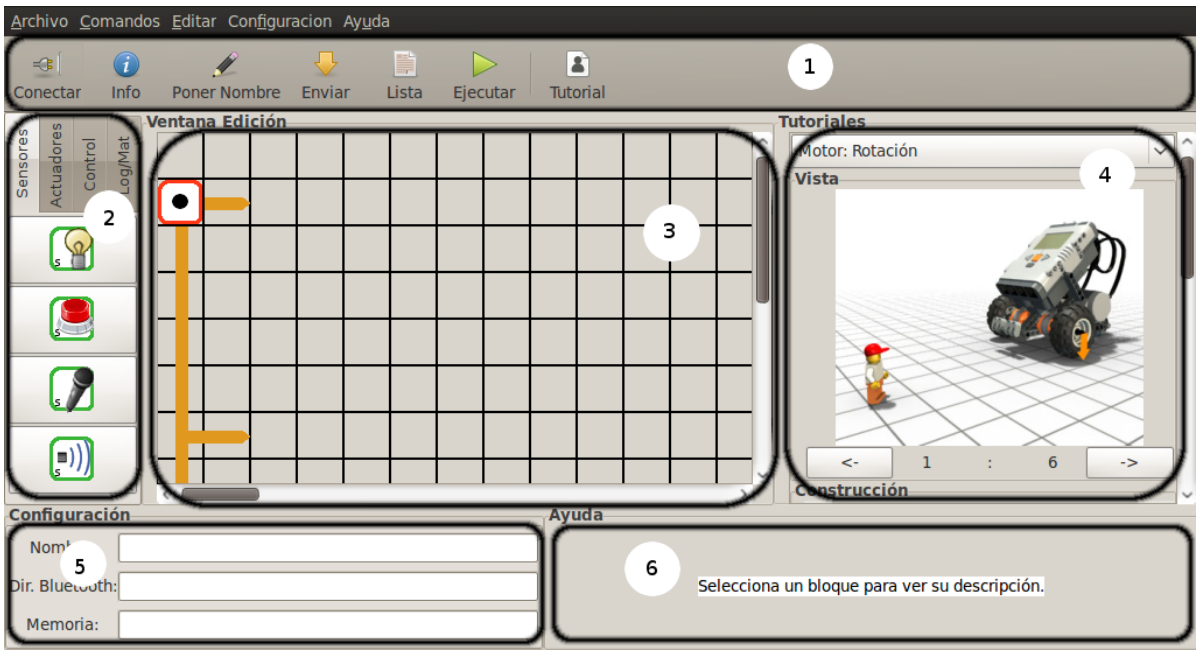

Figura 4.1: Interfaz del Compilador Visual.

4.1. Funcionamiento General del Compilador Visual

El compilador visual estará formado por una ventana principal que será el ambiente en el que los usuarios puedan realizar sus programas visuales. La ventana principal, que se muestra en la Figura 4.1, a su vez será dividida en 6 secciones:

Barra de Herramientas(1): contendrá las herramientas básicas para la comunicación con el CPU del Robot por ejemplo, cargar programa al Robot, recuperar información, etc.

Paleta de Bloques(2): permitirá a los usuarios seleccionar los bloques para programar el comportamiento que tendrá el Robot.

Ventana de Edición(3): será el espacio de trabajo para crear el programa por medio de los bloques.

Tutorial(4): presentará una serie de imágenes que ayudarán al usuario a crear sus primeros proyectos de programación visual.

Ventana de Configuración(5): permitirá ver y cambiar la configuración de algún bloque que haya sido seleccionado por el usuario en la ventana de edición.

Ventana de Ayuda(6): mostrará una descripción del bloque que sea seleccionado por el usuario en la ventana de edición.

Los bloques son elementos del programa que definirán el comportamiento del robot una vez que éste sea cargado en su CPU (Central Processing Unit, Unidad de Proceso

Central). Y existen tres tipos diferentes de bloques: sensores, actuadores y de control.

Con todas estas ventanas y herramientas el usuario debe ser capaz de escribir un programa que defina el comportamiento del robot y una vez que el usuario haya terminado de realizar el programa en el compilador, éste tendrá que enviar o cargar el programa en la CPU del Robot. Cuando el usuario presiona el botón de "Enviar" la aplicación tiene que traducir el programa del lenguaje visual (o de alto nivel) a un lenguaje de bajo nivel (lenguaje máquina) que será entendido por el robot y que será enviado por medio de un cable USB o conexión Bluetooth. Después de esto el usuario puede probar el programa que se ha creado, en el conjunto de construcción del robot y posteriormente realizar cambios si así lo desea.

4.2. Arquitectura del Compilador Visual

El enfoque utilizado a lo largo del desarrollo del software fue un enfoque orientado a objetos por lo que para modelar el sistema se utilizó UML (Unified Modeling Language, Lenguaje de Modelado Unificado).

El sistema fue dividido en dos capas: Interfaz del usuario y lógica de Negocio los cuáles se describen como: Capa de Presentación, que se compone de los elementos de la interfaz que interactuarán directamente con el usuario, permitiéndole acceder a los datos de la interfaz(bloques) y manipularlos. Capa de Negocio, aquí se ubican los elementos encargados de darle la funcionalidad al sistema. Dichos elementos procesarán las peticiones que son hechas por el usuario a través de la capa de presentación.

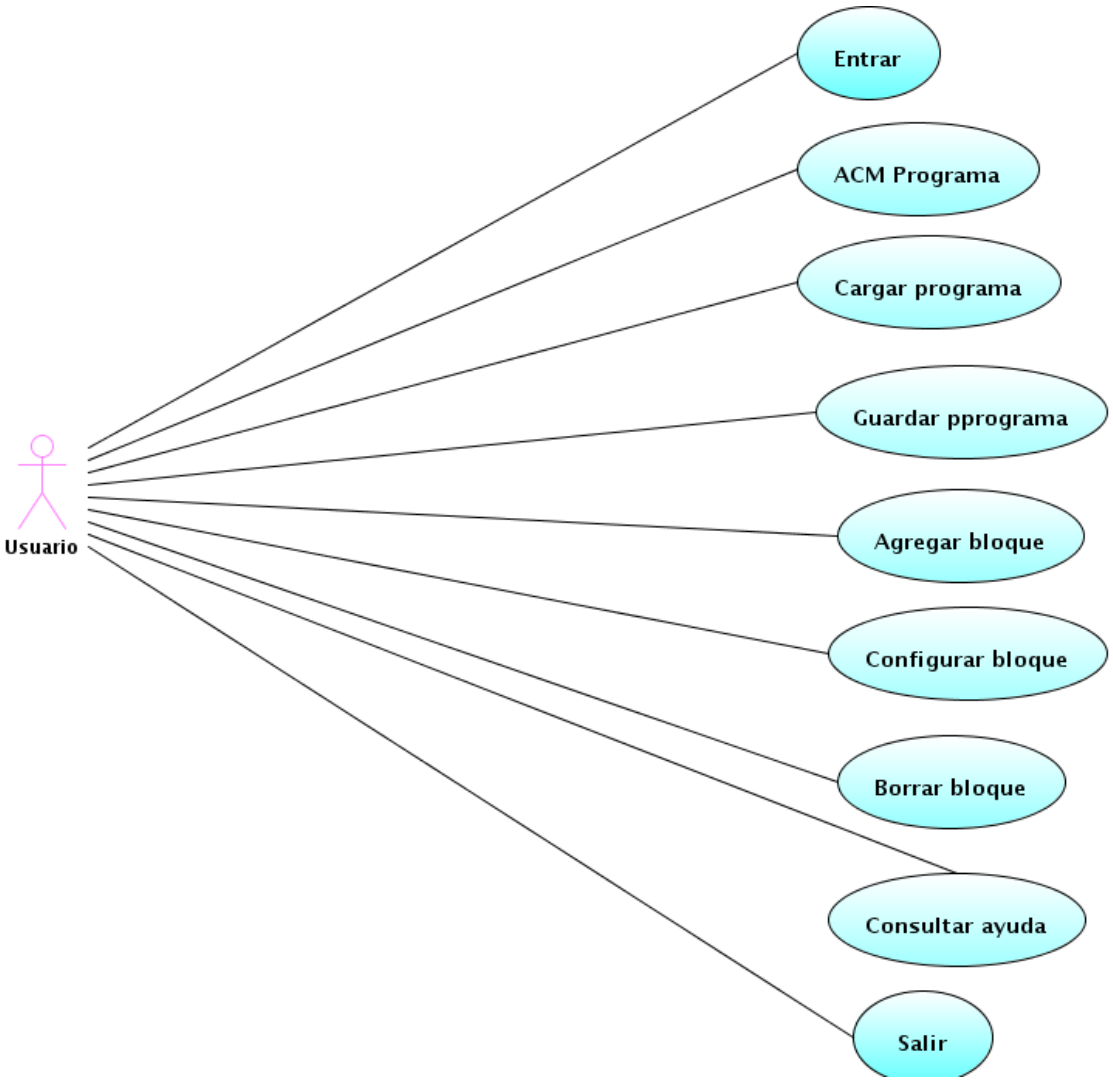

Figura 4.2: Caso de uso general del Compilador Visual.

4.3. Casos de uso

Para todos los casos de uso el único actor será el usuario, que es quién va a interactuar con el compilador visual. El caso de uso general se muestra en la Figura 4.2 y su descripción en la Tabla 4.1

Caso de Uso 0: General.
Actor: Usuario.
Descripción: El usuario ingresa al sistema.
Precondición: Que el compilador ya esté instalado y que otro usuario no esté usando la aplicación.

Tabla 4.1: Caso de uso general del Compilador Visual.

ACTOR		SISTEMA		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	El usuario accesa al sistema.	2	Muestra la pantalla con todas las funcionalidades con las que cuenta.	-
3	El usuario puede acceder a la ayuda si así lo desea.	4	Si eligió acceder a la ayuda, se muestra en la ventana de ayuda.	-
5	Elige abrir o crea un programa nuevo.	6	Si eligió abrir, muestra en ventana de edición el programa.	-
7	Selecciona, agrega y configura bloques unos detrás de otros para definir el flujo de ejecución del programa.	8	El sistema muestra los cambios en ventana de edición.	-
9	Elige guardar el programa fuente.	10	Guarda el programa fuente en la PC.	-
11	Elige enviar el programa al CPU del robot.	12	Carga el programa en la CPU del robot.	-
13	Sale del sistema.	14	Cierre del sistema.	-

4.3.1. Detalles de casos de uso

Cada uno de los casos de uso mostrados en la Figura 4.1, se muestran a detalle a continuación:

Caso de Uso 1: Entrar.
Actor: Usuario.
Descripción: El usuario ingresa al sistema.
Precondición: El usuario desea ingresar al sistema.

Tabla 4.2: Caso de uso Entrar.

ACTOR		SISTEMA		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	Accesa al Compilador Visual.	2	Muestra la pantalla con todas las funcionalidades con las que cuenta.	-

Postcondición: Al usuario se le otorgaron todas las funcionalidades con las que cuenta el sistema.

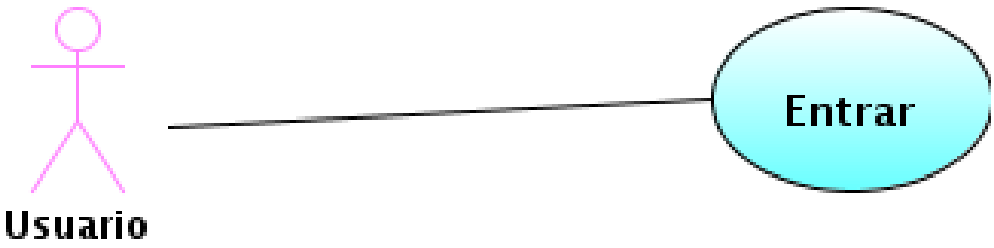

Figura 4.3: Caso de uso Entrar.

En la Figura 4.3, se presenta la forma en la que el usuario ingresará al sistema, éste únicamente deberá accesar al Compilador Visual y el sistema mostrará la pantalla donde se puede crear, abrir, guardar y/o modificar un programa.

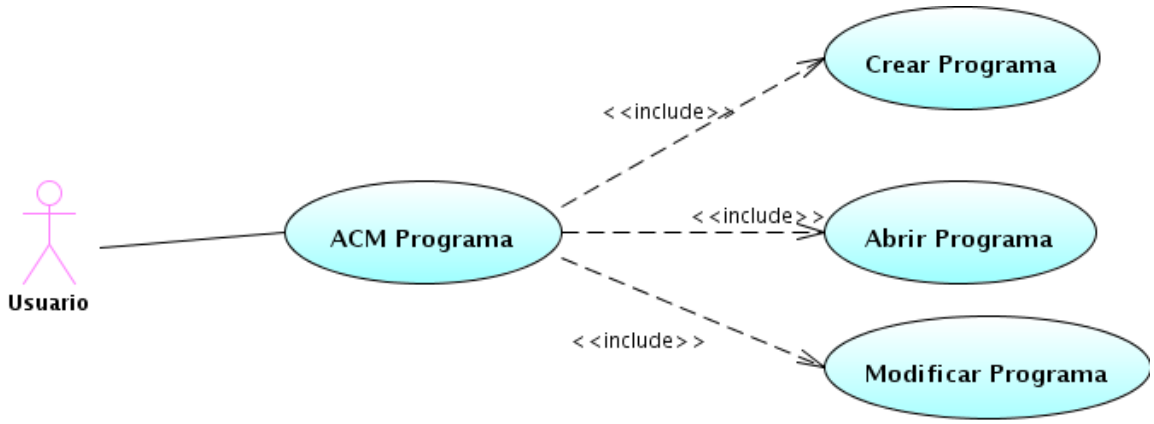

Figura 4.4: Casos de uso para Abrir, Crear y Modificar un Programa Fuente.

La Figura 4.4, muestra la administración de los programas, lo cual involucra 3 casos de uso: Abrir un programa, Crear un programa y Modificar un programa.

Caso de Uso 2.1: Abrir Programa.
Actor: Usuario.
Descripción: El usuario abre un programa que ya existe.
Precondición: El usuario desea abrir un programa fuente y el programa ya debe existir en la PC.

Tabla 4.3: Caso de uso Abrir programa.

ACTOR		SISTEMA		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	Accesa a la opción Abrir programa	2	Muestra una ventana de diálogo abrir.	-
3	Selecciona algún programa fuente ya existente en la computadora.	4	Abre el archivo y lo muestra en la ventana de la aplicación.	-

Postcondición: La aplicación muestra al usuario el archivo fuente en la ventana de edición.

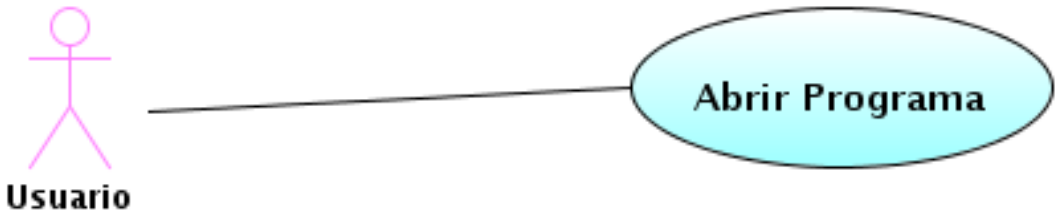

Figura 4.5: Casos de uso para Abrir un Programa.

La Figura 4.5, muestra el detalle de caso de uso Abrir Programa, donde el usuario deberá seleccionar la opción Abrir programa, la aplicación mostrará la ventana de diálogo Abrir, el usuario debe seleccionar algún archivo fuente existente y abrirlo. Por último la aplicación mostrará el programa en la ventana de edición.

Caso de Uso 2.2: Crear Programa.
Actor: Usuario.
Descripción: El usuario crea un nuevo programa.
Precondición: El usuario desea crear un nuevo programa.

Tabla 4.4: Caso de uso Crear programa.

ACTOR		SISTEMA		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	Abre la aplicación o cierra un programa creado previamente.	2	Muestra la ventana edición lista para un programa nuevo.	E1

EXCEPCIONES

ID	DESCRIPCIÓN	ACCIÓN
E1	El programa a cerrar no ha sido guardado.	El sistema envía un mensaje preguntando si se desea guardar el programa previo.

Postcondición: La aplicación permite crear un programa nuevo al usuario.

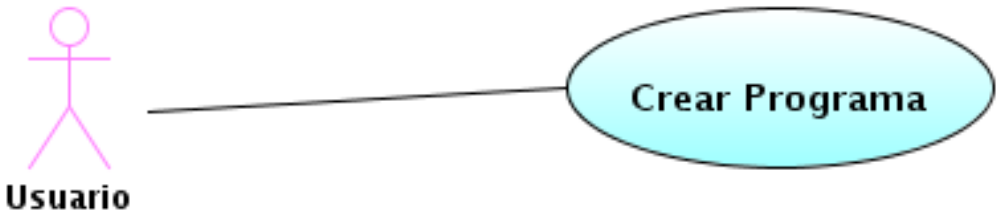

Figura 4.6: Casos de uso para Crear un Programa.

La Figura 4.6, muestra el detalle de caso de uso Crear Programa, donde el usuario deberá dar click al icono de Crear programa, la aplicación mostrará una ventana de diálogo Crear, el usuario debe seleccionar la ruta donde se creará el nuevo archivo y dar el nombre para este archivo. Por último la aplicación mostrará el programa fuente vacío en la ventana de edición.

Caso de Uso 3: Guardar programa.
Actor: Usuario.
Descripción: El usuario guarda el programa en la PC.
Precondición: El usuario desea guardar el programa en la PC y debe tener un programa abierto o recien creado.

Tabla 4.5: Caso de uso Guardar programa.

ACTOR		**SISTEMA**		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	Selecciona la opción guardar programa desde la aplicación.	2	Muestra ventana de diálogo guardar.	-
3	Selecciona ruta donde quiere guardar el programa y le pone un nombre al programa	4	Guarda el programa en la PC.	E1

EXCEPCIONES

ID	**DESCRIPCIÓN**	**ACCIÓN**
E1	Existe un programa en la PC con el mismo nombre elegido por el usuario.	El sistema envía un mensaje preguntando si desea sobreescribir el programa.

Postcondición: La aplicación guarda el programa en la PC.

Figura 4.7: Casos de uso para Guardar un Programa.

Caso de Uso 4: Cargar programa.
Actor: Usuario.
Descripción: El usuario carga el programa en la CPU del robot.
Precondición: El usuario desea cargar el programa en la CPU del robot, debe existir un programa abierto o recién creado.

Tabla 4.6: Caso de uso Cargar programa.

ACTOR		SISTEMA		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	Selecciona la opción cargar programa desde la PC.	2	Traduce el programa del lenguaje visual a uno de bajo nivel que entiende la CPU del Robot.	-
		3	Una vez traducido el programa se envía a la CPU del robot por medio de un cable USB.	E1

EXCEPCIONES

ID	DESCRIPCIÓN	ACCIÓN
E1	El CPU del robot no está conectado a la computadora.	El sistema envía un mensaje de error al usuario indicando que la CPU del Robot no está conectada a la computadora.

Postcondición: La aplicación carga el programa en la CPU del Robot y el usuario puede probar su funcionamiento.

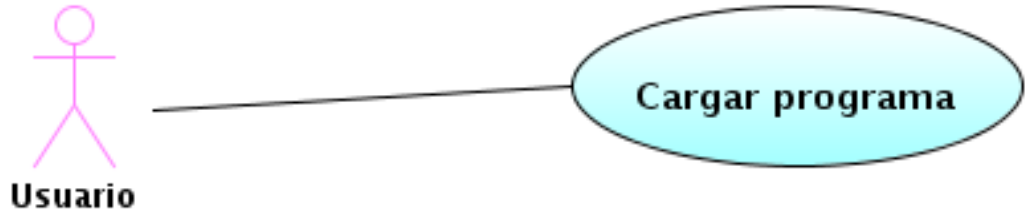

Figura 4.8: Casos de uso para Cargar un Programa.

Caso de Uso 5: Seleccionar y Agregar Bloque.
Actor: Usuario.
Descripción: El usuario selecciona un bloque de la paleta de bloques y la agrega a la ventana de edición.
Precondición: El usuario desea seleccionar un bloque de la paleta de bloques y agregarlo a la ventana de edición.

Tabla 4.7: Caso de uso Seleccionar y agregar bloque al programa.

ACTOR		SISTEMA		
PASO	ACCIÓN	PASO	ACCIÓN	EXCEPCIÓN
1	Selecciona un bloque y lo arrastra a la ventana de edición.	2	Crea una instancia del bloque seleccionado por el usuario y lo dibuja en la ventana de edición.	E1

EXCEPCIONES

ID	DESCRIPCIÓN	ACCIÓN
E1	La posición donde fue colocado el bloque no está permitida sintácticamente.	Ninguna

Postcondición: El usuario obtiene un bloque, que definirá algún comportamiento del Robot, en la ventana de edición.

Figura 4.9: Casos de uso para Seleccionar y agregar un bloque al Programa.

La Figura 4.9, muestra el detalle de caso de uso Seleccionar y Agregar Bloque, donde el usuario seleccionará en la paleta de bloques un bloque y lo agregará a la ventana de edición, una vez hecho esto el bloque se mostrará en la ventana de edición.

Caso de Uso 6: Configurar Bloque.
Actor: Usuario.
Descripción: El usuario configura un bloque que se encuentra en la ventana de edición.
Precondición: El usuario desea configurar un bloque, el bloque ya debe existir en la ventana de edición y el bloque debe ser configurable.

Tabla 4.8: Caso de uso Configurar bloque.

ACTOR		SISTEMA		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	Selecciona un bloque en la ventana de edición.	2	Muestra la configuración por omisión del bloque seleccionado en la ventana de configuración.	
3	Modifica la configuración del bloque en la ventana de configuración	3	Registra los cambios del bloque.	

Postcondición: El usuario modificó la configuración de algún bloque y puede continuar programando.

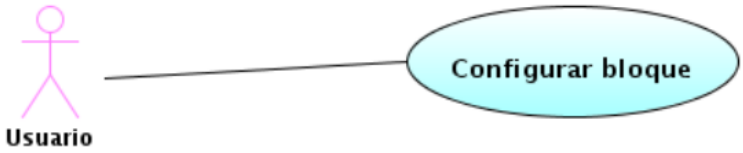

Figura 4.10: Casos de uso para Configurar un bloque.

La Figura 4.10 presenta el caso de uso para configurar un bloque, donde el usuario seleccionará en la ventana de edición un bloque y en la ventana de configuración el usuario podrá realizar modificaciones en el comportamiento del elemento que representa dicho bloque.

Caso de Uso 7: Consultar ayuda.
Actor: Usuario.
Descripción: El usuario consulta la ayuda seleccionando un bloque que se encuentra en la ventana de edición.
Precondición: El usuario desea consultar la descripción un bloque y el bloque debe

existir en la ventana de edición.

Tabla 4.9: Caso de uso Consultar ayuda.

ACTOR		SISTEMA		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	Selecciona un bloque en la ventana de edición.	2	Muestra la descripción del bloque seleccionado en la ventana de ayuda.	

Postcondición: El usuario consultó la descripción de algún bloque y puede continuar programando.

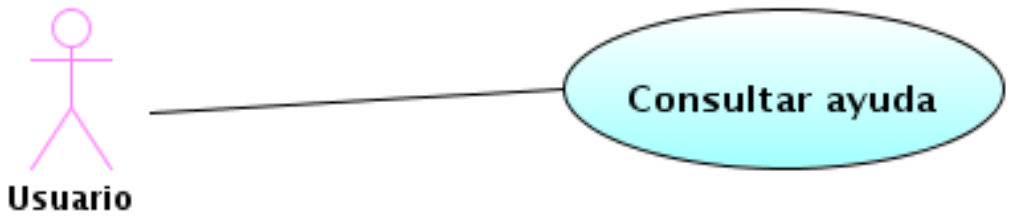

Figura 4.11: Casos de uso para Consultar la ayuda de un bloque.

La Figura 4.11, presenta el caso de uso para Consultar la ayuda de un bloque, donde el usuario debe seleccionar un bloque, entonces el sistema le muestra una descripción del bloque seleccionado en la ventana de ayuda.

Caso de Uso 8: Salir.
Actor: Usuario.
Descripción: El usuario cierra el sistema.
Precondición: El usuario desea desea salir del sistema.

Tabla 4.10: Caso de uso Salir del Programa.

ACTOR		SISTEMA		
PASO	**ACCIÓN**	**PASO**	**ACCIÓN**	**EXCEPCIÓN**
1	Cierra la aplicación.	2	Termino del sistema.	E1

EXCEPCIONES

ID	DESCRIPCIÓN	ACCIÓN
E1	El programa no ha sido guardado.	El sistema envía un mensaje preguntando si desea guardar el programa. En caso que se deseé guardar el programa se realizan los pasos del caso de uso 3.

Postcondición: El usuario cierra el sistema.

4.4. Diagrama detallado de clases

Los diagramas de paquetes serán divididos en 2 niveles de aplicación de acuerdo al número de capas de la aplicación mencionadas al inicio de este capítulo.

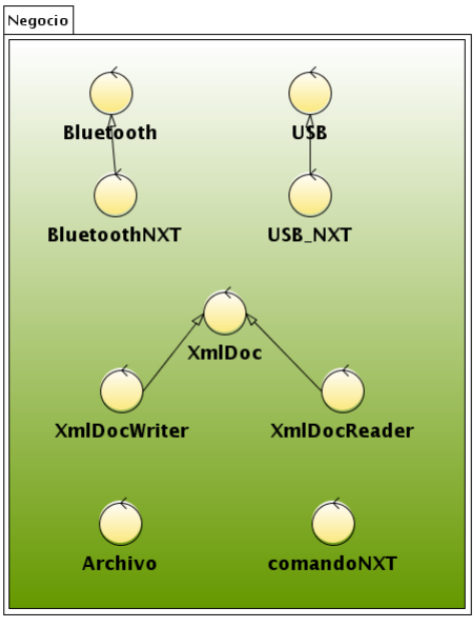

Figura 4.12: Capa de lógica de negocio.

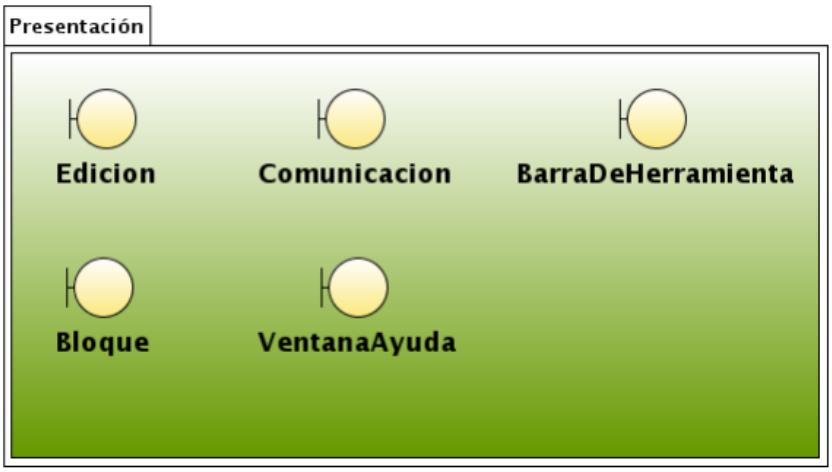

Figura 4.13: Capa de Presentación.

El paquete de lógica de negocio que aparece en la Figura 4.12 está formado por las clases que se encargan de los procesos de comunicación con la CPU del robot y para la administración de los programas fuentes. El paquete de presentación tiene a las clases que están directamente relacionadas con la interacción entre la interfaz del programa y el usuario y se muestra en la Figura 4.13.

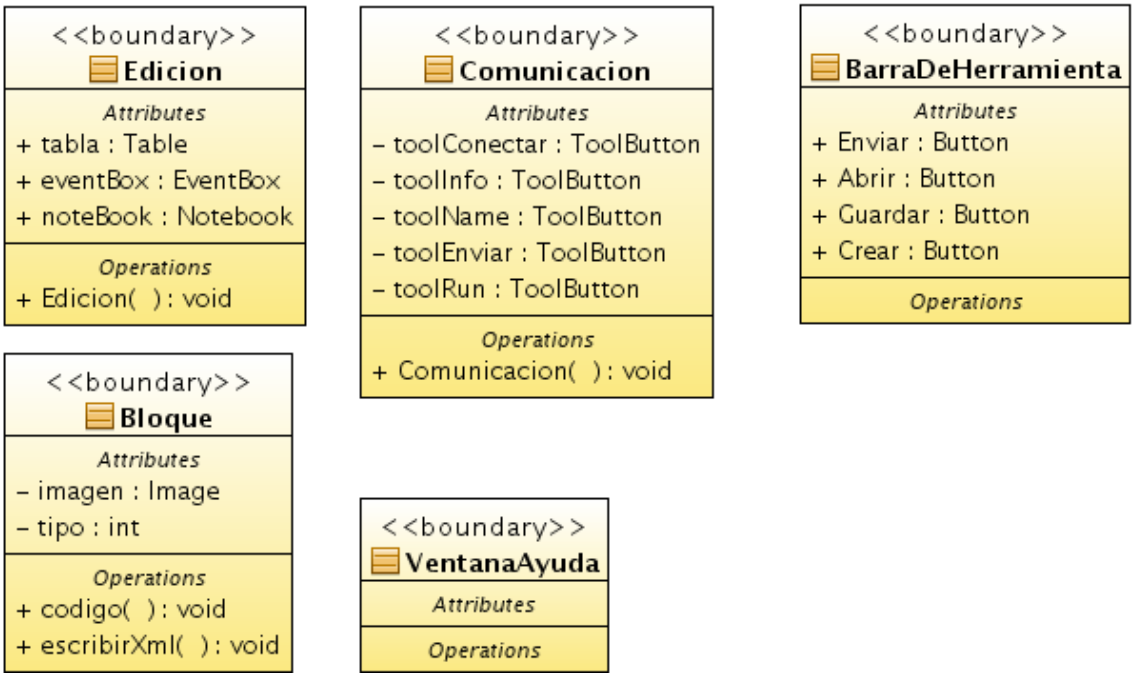

Figura 4.14: Clases de capa de presentación.

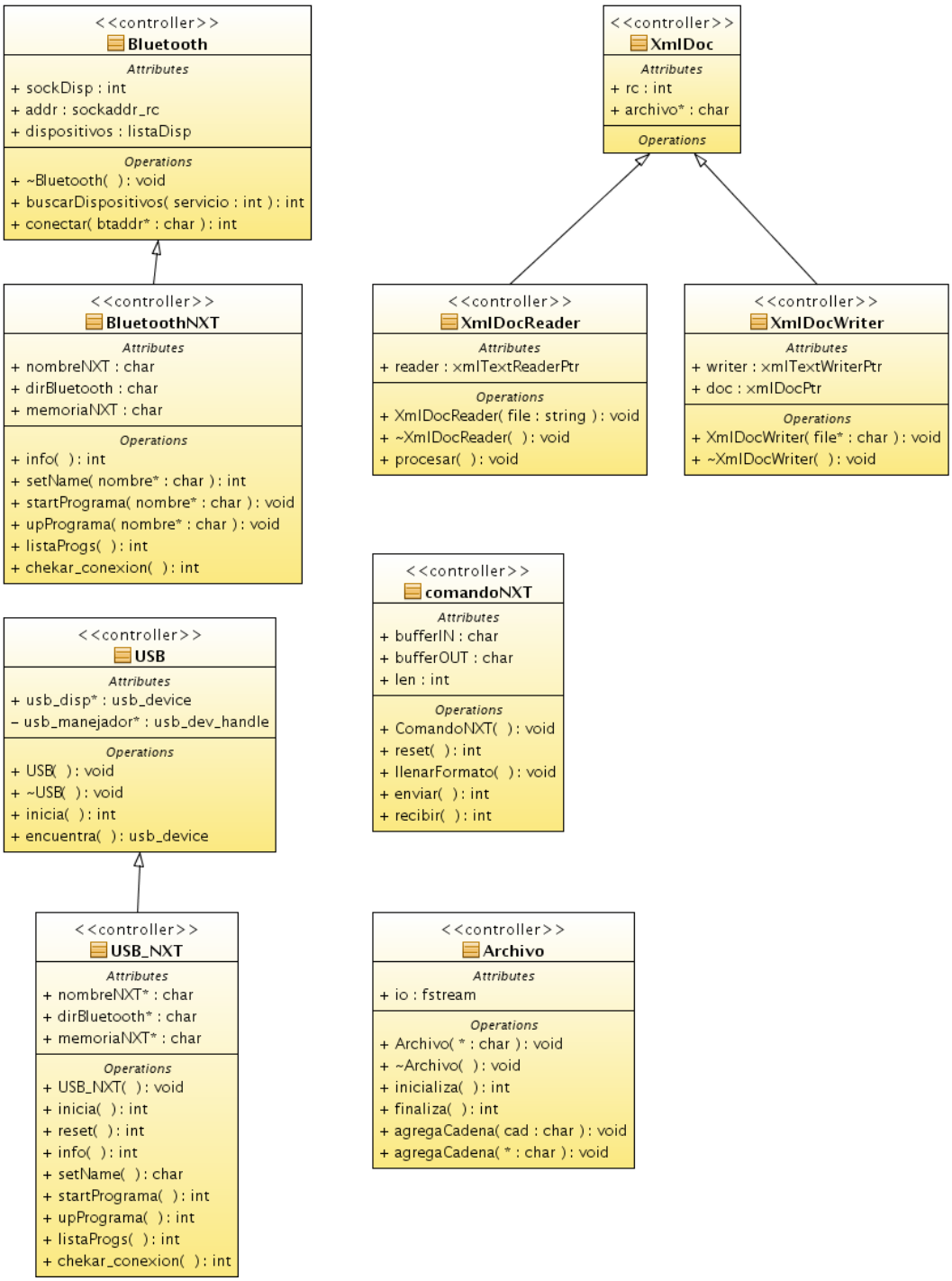

Figura 4.15: Clases de capa de lógica de negocio.

En las Figuras 4.14 y 4.15 se presenta el diseño detallado de las clases. Y a continuación se describe brevemente la utilidad de cada una de las clases.

Clases del paquete de presentación.

- **Clase Edición.** Se encarga de las operaciones de edición del proyecto como son: copiar, cortar, borrar y pegar bloques.

- **Clase Comunicación.** Es la clase encargada de llamar a las otras clases correspondientes de la comunicación cuando el usuario solicita una de estas operaciones por medio de un evento.

- **Clase Bloque.** Es una clase virtual de la cual se derivan todos los demás tipos de bloques(i.e. sensores, actuadores y control) y define los atributos y operaciones que comparten todos los bloques.

- **Clase Ayuda.** Es la clase encargada de presentar una descripción de los bloques cuando éstos son seleccionados.

Clases del paquete de lógica de negocio.

- **Clase Archivo.** Es la clase encargada de guardar la traducción del lenguaje de programación visual al lenguaje objeto(en este caso NBC).

- **Clase USB.** Se encarga de las operaciones de comunicación entre la computadora y la CPU del robot por medio de un dispositivo USB.

- **Clase bluetooth.** Se encarga de las operaciones de comunicación entre la computadora y la CPU del robot por medio de un dispositivo bluetooth.

- **Clase XmlDoc.** Se encarga de realizar el guardado y recuperación de un programa fuente creado por algún usuario del compilador visual.

- **Clase ComandoNXT.** Guarda el formato específico que requiere el protocolo de comunicación del robot lego mindstorms.

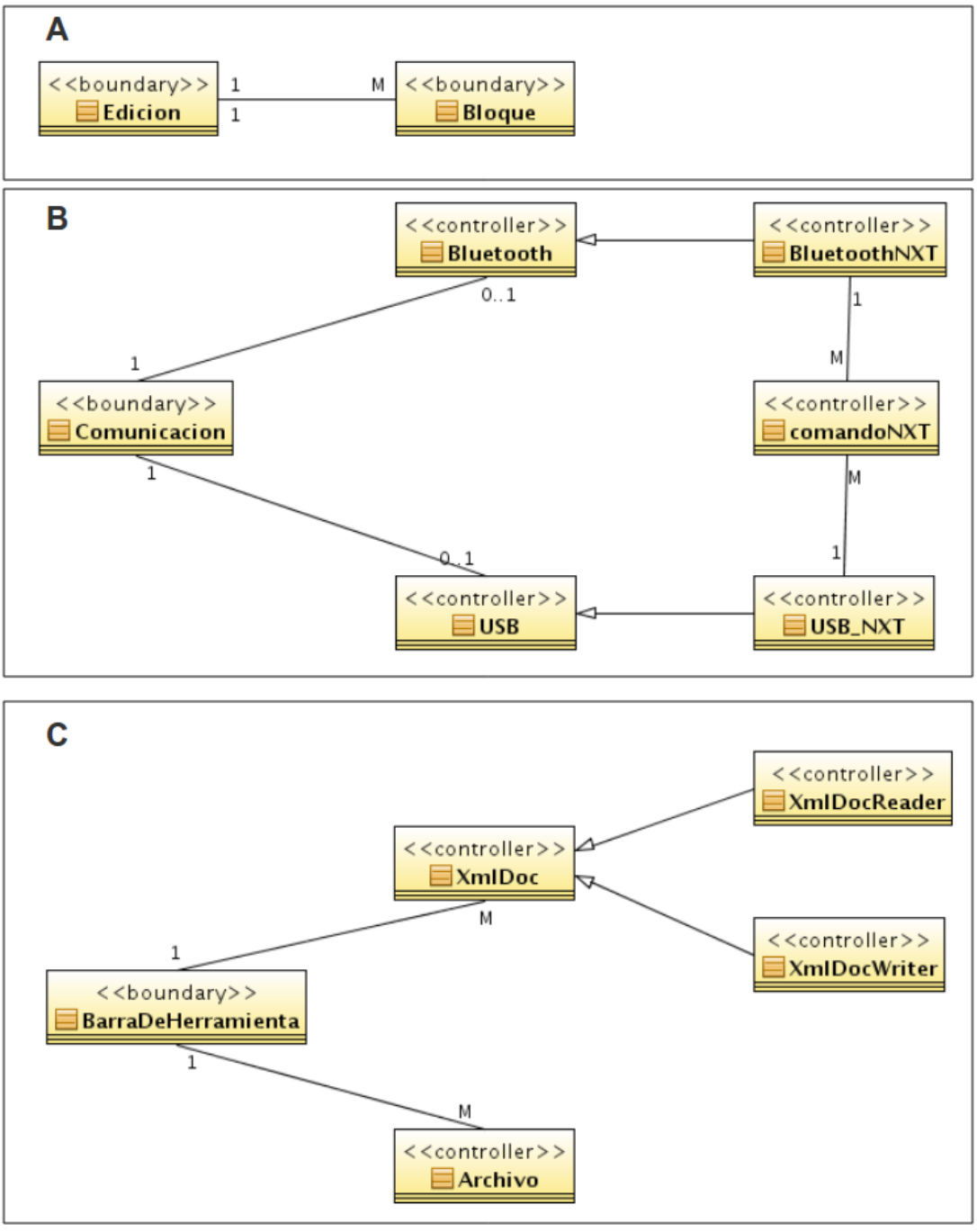

Figura 4.16: Relaciones entre las clases del proyecto.

4.5. Relaciones entre las clases

En esta sección se presentan las relaciones que existen entre las clases del proyecto. En la Figura 4.16 se muestran las clases separadas en tres cuadros (A, B y C), en las que se demuestra la relación directa entre las clases que se encuentran dentro de un mismo cuadro.

Cuadro A.

La relación entre la clase Edición y la clase Bloque se da por el motivo de que en la ventana de edición los usuarios crean sentencias visuales arrastrando bloques o iconos a dicha ventana.

Cuadro B.

La relación de la clase Comunicación entre la clase Bluetooth y la clase Usb se da por el hecho de que existe una comunicación por parte del compilador visual hacia y desde la CPU del robot Mindstorms. Las clases BluetoothNXT y Usb-NXT son especializaciones de las clases Bluetooth y Usb respectivamente. La clase ComandoNXT es la clase especial que maneja el protocolo específico del Mindstorms NXT y es por eso que tiene una relación directa con las clases BluetoothNXT y Usb-NXT.

Cuadro C.

La clase BarraDeHerramientas tiene relación con la clase Archivo, que es utilizada para realizar la traducción del programa fuente al programa objeto; y también tiene relación con la clase XmlDoc, que es la clase encargada de guardar el programa fuente en un archivo Xml para su posterior recuperación. Las clases XmlDocReader y XmlDocWriter son especializaciones de la clase XmlDoc y son utilizados para recuperar el programa fuente desde el disco duro y para el guardado del programa en disco respectivamente.

4.6. Diagramas de secuencia

En esta sección se presentan los diagramas de secuencia que muestran el escenario donde el Usuario desea realizar un programa dentro del sistema. Se podrán observar los mensajes de intercambió entre las clases y la funcionalidad que tiene cada una de ellas, dichas clases se presentaron en el diagrama de paquetes mostrado en la sección anterior.

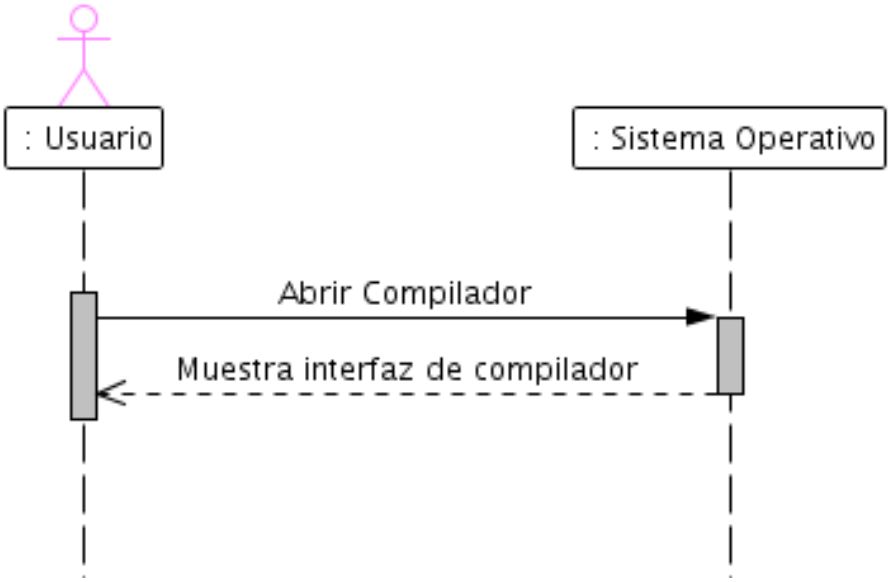

Figura 4.17: Diagrama de secuencia para el caso de uso Entrar.

Los diagramas de secuencia que se muestran a continuación se crearon a partir de los diagramas de casos de uso de la sección 4.3.

4.6.1. Diagrama de secuencia Entrar

El proceso de abrir el compilador visual, se muestra en la Figura 4.17, donde el usuario envía la orden de ingresar al sistema, haciendo doble click en el archivo lanzador de la aplicación. El sistema abre la interfaz del compilador visual con la que el usuario se comunicará directamente y en la que podrá utilizar las funcionalidades con las que cuenta el compilador.

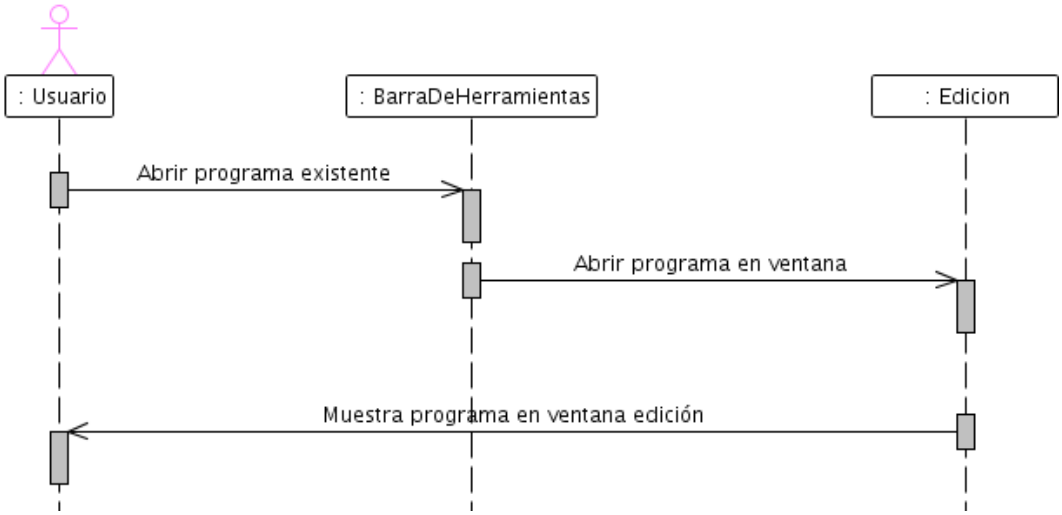

Figura 4.18: Diagrama de secuencia Abrir programa.

4.6.2. Diagrama de secuencia Abrir programa

El diagrama de secuencia mostrado en la Figura 4.18, presenta la forma en la qué un usuario solicita a la barra de herramientas Abrir un programa existente y ésta es cargada en la ventana de edición. En el diagrama de clases que se muestra en la sección 4.4, la clase relacionada con la ventana de edición es llamada *Edición*.

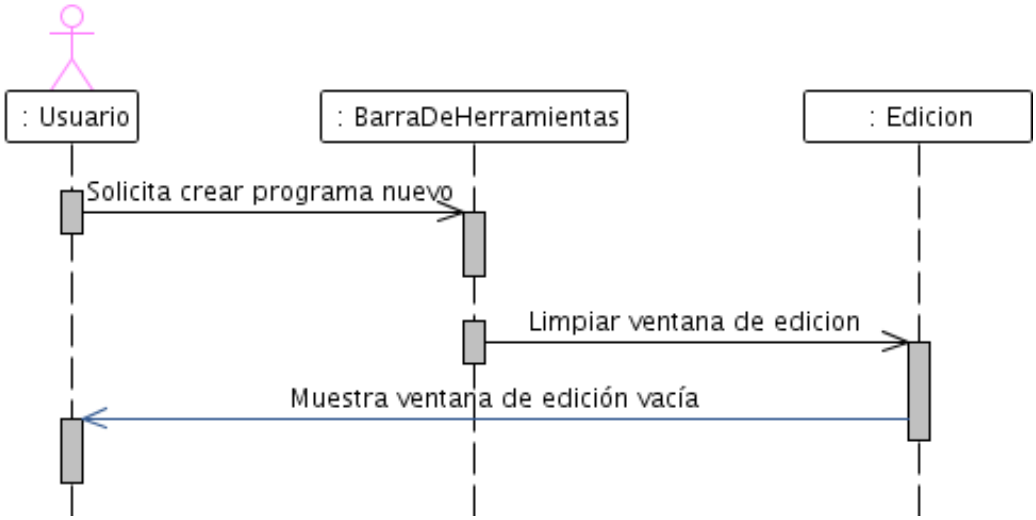

Figura 4.19: Diagrama de secuencia Crear programa.

4.6.3. Diagrama de secuencia Crear programa

En la Figura 4.19, se presentan las acciones qué deben ser ejecutadas para poder crear un nuevo programa, donde el usuario debe solicitar crear un nuevo programa a la barra de herramientas y ésta a su vez le notifica a la ventana de edición para que esta realice la operación de limpiar la ventana.

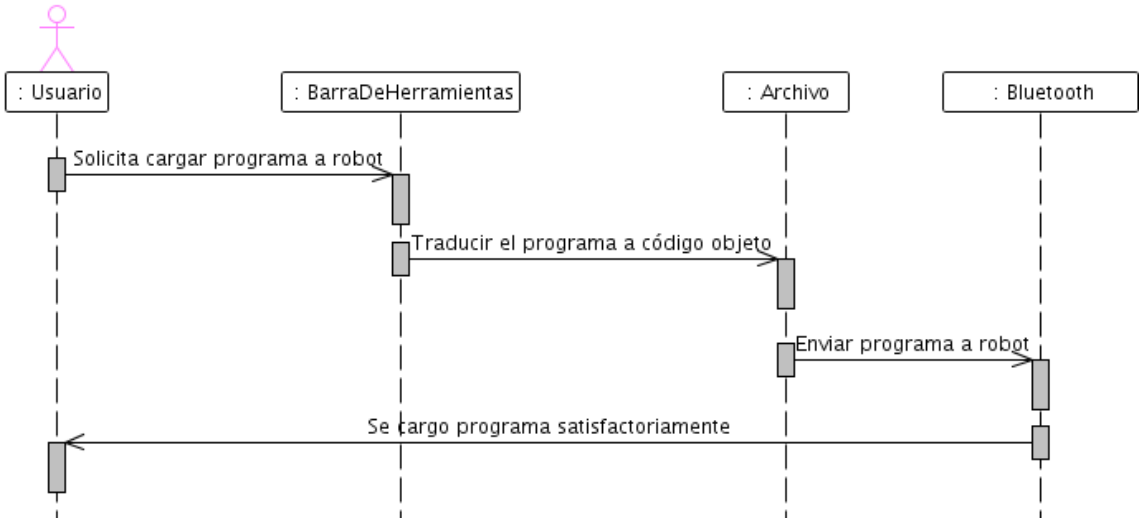

Figura 4.20: Diagrama de secuencia Cargar programa.

4.6.4. Diagrama de secuencia Cargar programa

El diagrama de secuencias mostrado en la Figura 4.20, corresponde al caso de uso cargar programa. El programa compilado se carga al robot por lo cual se presenta la forma en la que interactúan cada una de las clases encargadas de hacer ésta labor y los mensajes que intercambian.

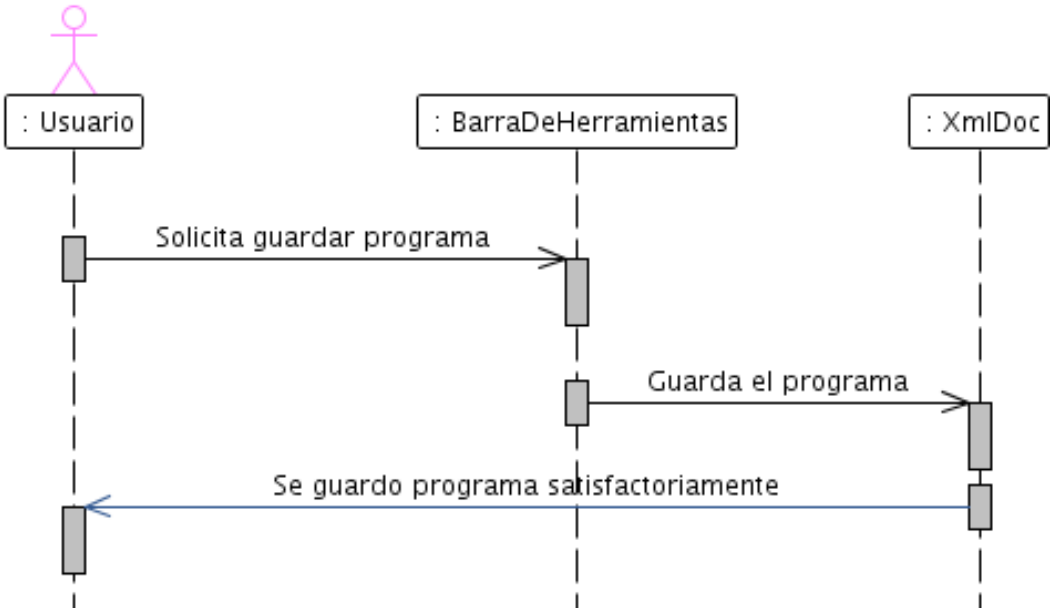

Figura 4.21: Diagrama de secuencia Guardar programa.

4.6.5. Diagrama de secuencia Guardar programa

En la Figura 4.21 se muestra la secuencia de acciones que realizan varias clases para guardar el programa fuente en el disco duro de la PC.

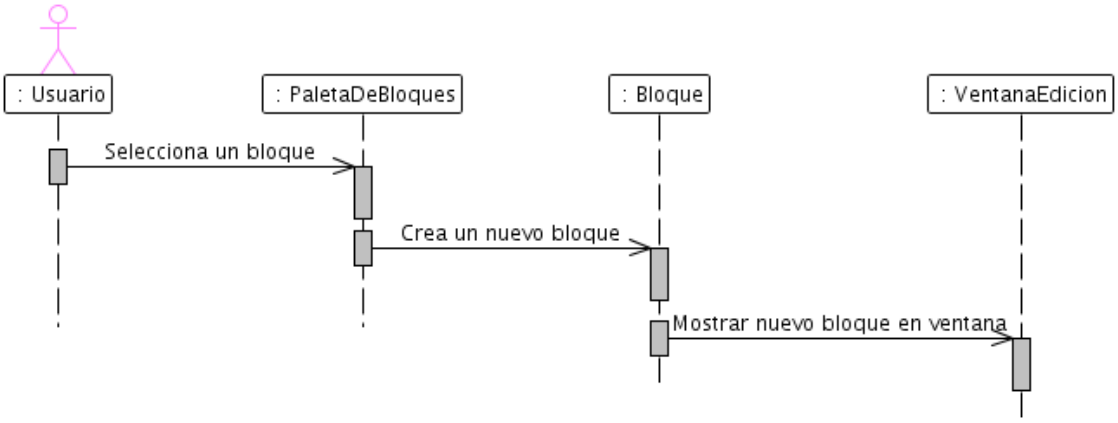

Figura 4.22: Diagrama de secuencia Agregar bloque.

4.6.6. Diagrama de secuencia Agregar bloque

La Figura 4.22, muestra la secuencia en la que se selecciona un bloque de la paleta de bloques y se agrega en la ventana de edición del compilador. Los bloques son agregados en la ventana de edición para crear las sentencias visuales que programarán el comportamiento del robot cuando el usuario mande el archivo compilado a la CPU.

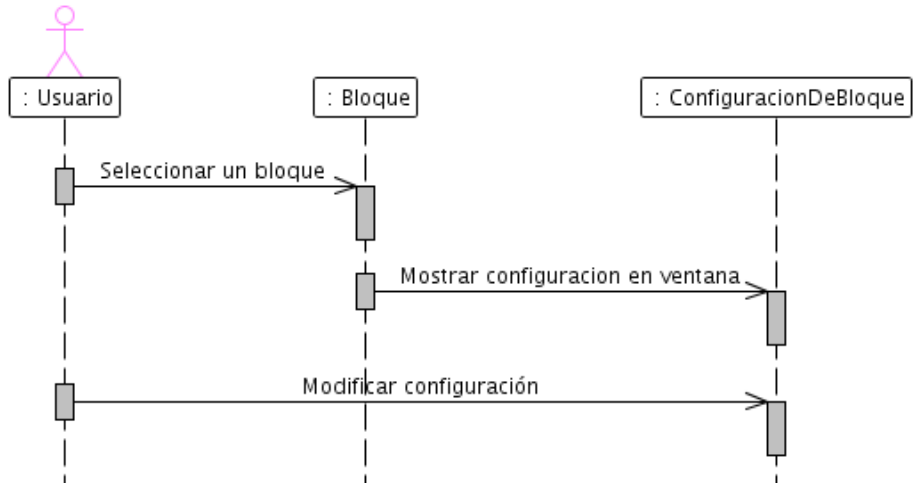

Figura 4.23: Diagrama de secuencia Configurar bloque.

4.6.7. Diagrama de secuencia Configurar bloque

Todos los bloques cuando se agregan a la ventana de edición tiene un configuración por defecto, la secuencia de uso que se muestra en la Figura 4.23 es la que realiza el usuario para cambiar la configuración del bloque seleccionado.

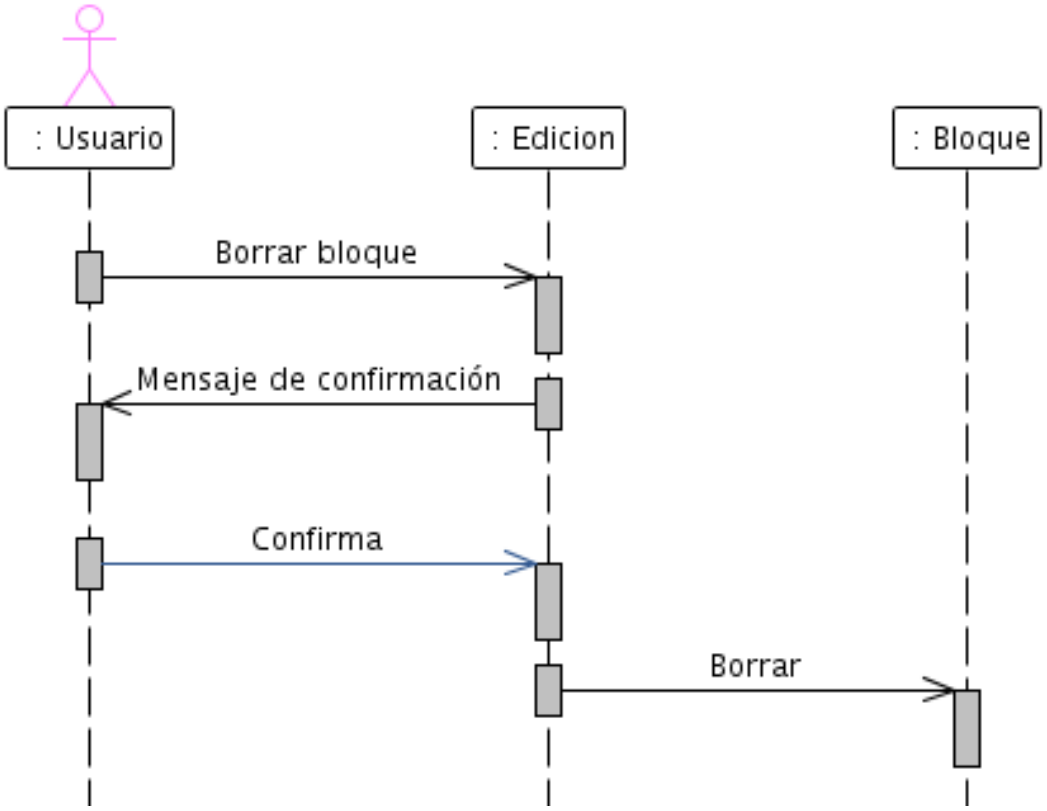

Figura 4.24: Diagrama de secuencia Borrar bloque.

4.6.8. Diagrama de secuencia Borrar bloque

En la Figura 4.24 se muestra el diagrama de secuencia para que el usuario realice el borrado un algún bloque que se encuentre dentro de la ventana de edición.

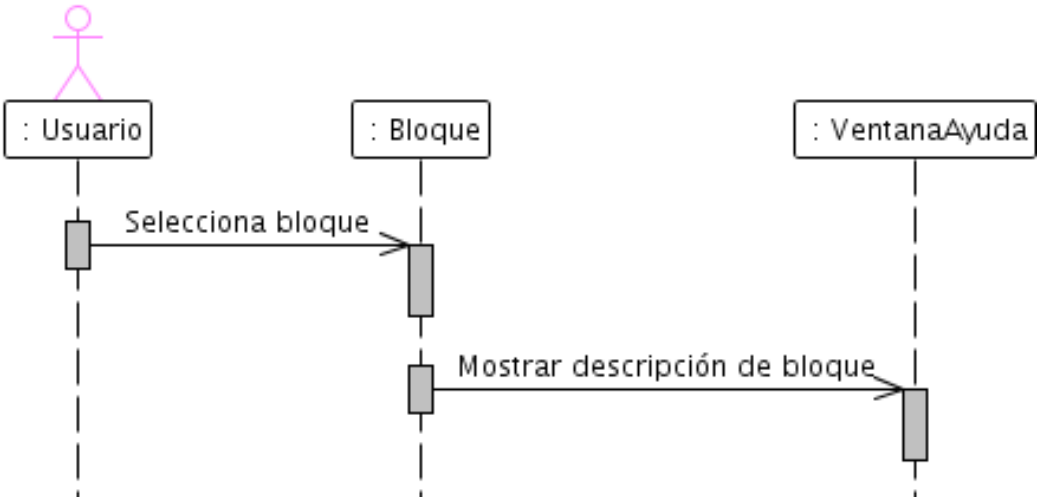

Figura 4.25: Diagrama de secuencia Consultar ayuda.

4.6.9. Diagrama de secuencia consultar ayuda

La Figura 4.25 muestra la secuencia consultar ayuda, donde el usuario tiene que seleccionar un bloque y automáticamente la descripción del bloque debe aparecer en la ventana de ayuda.

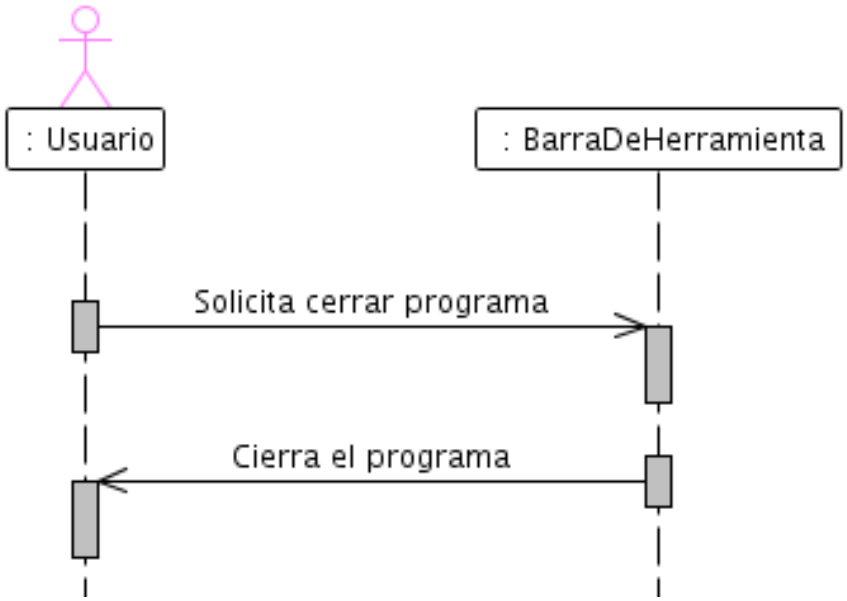

Figura 4.26: Diagrama de secuencia Salir.

4.6.10. Diagrama de secuencia Salir

El proceso de salir del sistema, se muestra en la Figura 4.26, donde el usuario envía su petición de salir del sistema a la clase Barra de herramientas y ésta se encarga de cerrar la aplicación.

4.7. Resumen

En este capítulo se mostró el análisis y diseño del compilador visual. Para ello se utilizó el lenguaje de modelado unificado para realizar la representación de casos de uso, diagrama de paquetes, diagrama detallado de clases y diagrama de secuencias.

En el siguiente capítulo se presentarán las pruebas que se realizaron sobre las unidades lógicas de la aplicación y se muestra un ejemplo de uso de compilador visual programando al robot *Mindstorms NXT* de Lego.

5
Pruebas

En este capítulo se muestra el conjunto de pruebas que se aplicaron al software desarrollado. Estas pruebas son llamadas pruebas unitarias y pruebas de integración.

Las pruebas unitarias son llamadas así porque realizan pruebas sobre unidades lógicas del programa y su objetivo es asegurar el correcto funcionamiento de dichas unidades. Una vez que las unidades lógicas fueron probadas se pasa a las pruebas de integración. En este documento cada unidad lógica es una de las clases descritas en el capítulo anterior.

Ya que en el desarrollo del software para esta tesis se ha utilizado un enfoque orientado a objetos para realizar las pruebas unitarias se utilizará la herramienta CPPUnit, que es un marco de trabajo para realizar pruebas sobre programas escritos en código C++. Las pruebas realizadas en las unidades fueron solo sobre las funciones o métodos principales de cada una de las clases que forman parte del proyecto.

Por último se presenta un ejemplo de programación de un robot *Mindstorms NXT de Lego* por medio del compilador visual desarrollado.

5.1. Pruebas Unitarias

Para todas las unidades o clases se creó una función principal que se encarga de ejecutar todas las pruebas que se encuentran registradas en un apuntador de pruebas llamado suite, como se presenta en las líneas 8 y 12 del código que se muestra a continuación.

main.cc

```
1 #include <cppunit/CompilerOutputter.h>
2 #include <cppunit/extensions/TestFactoryRegistry.h>
3 #include <cppunit/ui/text/TextTestRunner.h>
4
5 int main(void)
6 {
7       // Conjunto de registros
8       CppUnit::Test *suite =
          CppUnit::TestFactoryRegistry::getRegistry().makeTest();
9
10       // Objeto encargado de correr las pruebas
11       CppUnit::TextTestRunner runner;
12       runner.addTest( suite );
13
14       // Configuración de salida en formato estilo compilador
15       runner.setOutputter( new CppUnit::CompilerOutputter(
          &runner.result(), std::cerr ) );
16       // Corre las pruebas
17       bool wasSucessful = runner.run();
18
19       // Regresa codigo de error 1 si la prueba falla
20       return wasSucessful ? 0 : 1;
21 }
```

Cada unidad lógica del proyecto tiene una clase de prueba, por ejemplo, para la clase *Archivo* se creó una clase prueba llamada *ArchivoTest*, para la clase *XmlDoc* se creó la clase de prueba *XmlDocTest* y así sucesivamente para cada una de las clases.

A continuación se muestran el plan de pruebas para cada una de las unidades lógicas del software.

5.1.1. Unidad lógica: Archivo

La clase *Archivo* se encarga de escribir la traducción del programa fuente al programa objeto en un archivo de texto y su código se muestra en el anexo A.

Como se mencionó anteriormente la clase de prueba llamada *ArchivoTest* es la encargada de realizar las pruebas sobre la unidad lógica *Archivo* y su código se muestra en *ArchivoTest.cpp*.

ArchivoTest.cpp

```
1 #include<cppunit/extensions/HelperMacros.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <cstring>
5 #include <stdlib.h>
6 #include "Archivo.h"
7
8 #define ARCHIVO "Hola.txt"
9 #define ARCHIVO_PATRON "HolaPrueba.txt"
10
11 class ArchivoTest : public CppUnit::TestFixture
12 {
13         CPPUNIT_TEST_SUITE( ArchivoTest );
14         CPPUNIT_TEST( prueba );
15         CPPUNIT_TEST_SUITE_END();
16
17 private:
18        Archivo* a;
19        char  *tmpBuffer, *auxBuffer;
20         float tamano;
21 public:
22         void setUp(void);
23         void prueba(void);
24         char* leeArchivo(char *arch);
25 };
26
27 // Recupera datos de archivos
28   char* ArchivoTest::leeArchivo(char *arch)
29   {
30       char*  buffer;
31       struct  stat estado;
32       int  fd, nbytes;
33
34       if((fd=open(arch,O_RDWR, 0666))<0)
35           printf("Fallo al intentar abrir el archivo");
36
37       fstat(fd, &estado);
38       tamano = estado.st_size;
39       printf("Tamaño de %s: %f", arch, tamano);
40
```

```
41       if(lseek(fd,0,SEEK_SET) < 0)
42            printf("Fallo al intentar accesar a la 1ra posicion del archivo");
43
44       buffer = (char*) malloc (1024);
45       if( (nbytes = read(fd, buffer, tamano)) < 0)
46            printf("Fallo al intentar leer archivo");
47
48       return buffer;
49   }
50
51 // Inicilizacion de variable
52   void ArchivoTest::setUp ()
53   {
54       a = new Archivo(ARCHIVO);
55   }
56
57 // Prueba
58   void ArchivoTest::prueba()
59   {
60       a->agregaCadena("Hola ");
61       a->agregaCadena("mundo");
62       a->agregaLinea(";");
63
64       delete(a);
65
66       tmpBuffer = leeArchivo(ARCHIVO);
67       auxBuffer = leeArchivo(ARCHIVO_PATRON);
68
69       // Verifica que apuntadores apunten a algo
70       CPPUNIT_ASSERT(tmpBuffer != NULL);
71           CPPUNIT_ASSERT(auxBuffer != NULL);
72
73       // El contenido de ambos archivos debe ser igual
74          CPPUNIT_ASSERT( 0 == memcmp(tmpBuffer, auxBuffer, tamano) );
75   }
76
77 // Registro de clase en framework cppunit
78 CPPUNIT_TEST_SUITE_REGISTRATION(ArchivoTest);
```

Todas las clases de prueba deben ser derivados de la clase *CppUnit::TestFixture* y es posible sobreescribir los métodos *setUp* y *tearDown* los cuales sirven para inicializar variables u objetos y para liberarlos respectivamente. En el código de arriba sólo se sobreescribió el método *setUp*.

Para comprobar que la clase *Archivo* trabaja como se desea, se creará un archivo llamado *ARCHIVO-PATRON* que contiene los datos contra los que se va a comparar el contenido del *ARCHIVO* generado por nuestra clase, como se muestra en las líneas 8 y 9 de *ArchivoTest.cc*. Si los datos que contienen ambos archivos son iguales la prueba será exitosa, de otro modo existirá una falla.

En el método *prueba* se realizan las operaciones de un objeto *a* de la clase *Archivo* para llenar el *ARCHIVO* con datos, como se muestra en las líneas 60 - 62, y después se realiza la afirmación de que el contenido de ambos archivos es igual, como se muestra en la línea 74.

```
Archivo  Editar  Ver  Terminal  Ayuda
ronnytonche@ronnytonche-laptop:~/Proyectos/CPPUnit$ ./pruebaArchivo
.Tamaño de Hola.txt: 30.000000
Tamaño de HolaPrueba.txt: 30.000000


OK (1)
ronnytonche@ronnytonche-laptop:~/Proyectos/CPPUnit$
```

Figura 5.1: Prueba sobre clase Archivo sin errores.

```
Archivo  Editar  Ver  Terminal  Ayuda
ronnytonche@ronnytonche-laptop:~/Proyectos/CPPUnit$ ./pruebaArchivo
.Tamaño de Hola.txt: 30.000000
Tamaño de HolaPrueba.txt: 31.000000
F

ArchivoTest.cpp:56:Assertion
Test name: ArchivoTest::prueba
assertion failed
- Expression: 0 == memcmp(tmpBuffer, auxBuffer, tamano)

Failures !!!
Run: 1   Failure total: 1   Failures: 1   Errors: 0
ronnytonche@ronnytonche-laptop:~/Proyectos/CPPUnit$
```

Figura 5.2: Prueba sobre clase Archivo con errores.

Después de compilar y ejecutar el programa de *prueba*, si los resultados son favorables,

es decir, si no existió ningún error se obtendrá el mensaje que se muestra en la Figura 5.1, de lo contrario se obtendrá el resultado de la Figura 5.2.

5.1.2. Unidad lógica: XmlDoc

La clase *XmlDoc* es utilizada para guardar el programa fuente del usuario en un documento en formato xml y el código de esta clase y como el de todas las demas unidades lógicas se encuentra en el anexo A.

La clase de prueba para la clase *XmlDoc* es *XmlDocTest* y su código se muestra en *XmlDocTest.cpp*.

XmlDocTest.cpp

```
1 #include<cppunit/extensions/HelperMacros.h>
2 #include <sys/stat.h> // Estado de archivos
3 #include <fcntl.h> // Operaciones de archivos
4 #include <cstring> // Memcmp
5 #include <stdlib.h> // Malloc
6 #include "XmlDoc.h"
7
8 #define ARCHIVO "Doc.xml"
9 #define ARCHIVO_PATRON "DocPatron.xml"
10
11 class XmlDocTest : public CppUnit::TestFixture {
12       CPPUNIT_TEST_SUITE( XmlDocTest );
13       CPPUNIT_TEST( prueba );
14       CPPUNIT_TEST_SUITE_END();
15 private:
16       XmlDocWriter* xml;
17       char  *tmpBuffer, *auxBuffer;
18       float tamano;
19 public:
20       void setUp(void);
21       void prueba(void);
22       char* leeArchivo(char *arch);
23 };
24
25 // Recupera datos de archivos
26 char* XmlDocTest::leeArchivo(char *arch)
27 {
```

```
28      char*  buffer;
29      struct  stat estado;
30      int  fd, nbytes;
31
32      if((fd=open(arch,O_RDWR, 0666))<0)
33          printf("Fallo al intentar abrir el archivo");
34
35      fstat(fd, &estado);
36      tamano = estado.st_size;
37      printf("Tamaño de %s: %f", arch, tamano);
38
39      if(lseek(fd,0,SEEK_SET) < 0)
40          printf("Fallo al intentar accesar a la 1ra posicion del archivo");
41
42      buffer = (char*) malloc (1024);
43      if( (nbytes = read(fd, buffer, tamano)) < 0)
44          printf("Fallo al intentar leer archivo");
45
46      return buffer;
47 }
48
49 // Inicilizacion de variable
50 void XmlDocTest::setUp ()
51 {
52      xml = new XmlDocWriter(ARCHIVO);//"DocPatron.xml");
53 }
54
55 // Prueba
56 void XmlDocTest::prueba()
57 {
58      delete(xml);
59
60      tmpBuffer = leeArchivo(ARCHIVO);
61      auxBuffer = leeArchivo(ARCHIVO_PATRON);
62
63      // Verifica que apuntadores apunten a algo
64      CPPUNIT_ASSERT(tmpBuffer != NULL);
65      CPPUNIT_ASSERT(auxBuffer != NULL);
66
67      // El contenido de ambos archivos debe ser igual
68      CPPUNIT_ASSERT( 0 == memcmp(tmpBuffer, auxBuffer, tamano) );
69 }
70
71 // Registro de clase en framework cppunit
```

```
72 CPPUNIT_TEST_SUITE_REGISTRATION(XmlDocTest);
```

Al igual que con la clase anterior para comprobar que la clase *XmlDoc* trabaja según lo esperado, se creará un archivo llamado *ARCHIVO-PATRON* que contiene los datos contra los que se va a comparar el contenido del *ARCHIVO* generado por la clase XmlDoc. Si los datos que contienen ambos archivos son iguales la prueba será exitosa.

El método *prueba* trabaja de manera similar al visto anteriormente y aquí nuevamente el punto a resaltar es la afirmación de que el contenido de ambos archivos es igual, como se muestra en la linea 68 de *XmlDocTest.cpp*.

A partir de esta sección del capítulo en adelante se toma como un hecho que las ejecuciones de los códigos de pruebas son satisfactorios y por lo tanto que las clases o unidades lógicas trabajan como se espera.

5.1.3. Unidad lógica: ComandoNXT

Para realizar operaciones de comunicación entre el compilador visual y el CPU del Mindstorms NXT se utilizan comandos con un formato específico y la clase *ComandoNXT* es la encargada de manejar este formato de los comandos.

La clase de prueba para la clase *ComandoNXT* es *ComandoNXTTest* y su código se muestra en *ComandoNXTTest.cpp*.

XmlDocTest.cpp

```
1 #include<cppunit/extensions/HelperMacros.h>
2 #include <cstring> // Memcmp
3 #include "ComandoNXT.h"
4
5 #define TIPO_CMD 1
6 #define BYTE_CMD 1
7
8 class ComandoNXTTest : public CppUnit::TestFixture {
9       CPPUNIT_TEST_SUITE( ComandoNXTTest );
10       CPPUNIT_TEST( prueba );
11       CPPUNIT_TEST_SUITE_END();
12 private:
13       ComandoNXT* cmd;
14       unsigned char bufferPatron[NXT_BUFFER_SIZE];
15 public:
```

```
16      void setUp(void);
17      void tearDown(void);
18      void prueba(void);
19 };
20
21
22 // Inicilizacion de variable
23 void ComandoNXTTest::setUp ()
24 {
25      cmd = new ComandoNXT();
26
27      //Inicializa estructura contra la que se va a comparar
28      memset(&bufferPatron, 0, NXT_BUFFER_SIZE);
29
30      // Se llena buffer
31      bufferPatron[0] = TIPO_CMD;
32      bufferPatron[1] = BYTE_CMD;
33 }
34
35 // Prueba
36 void ComandoNXTTest::prueba()
37 {
38      cmd->llenarFormato(0,0,1,1,NULL);
39
40      int res = memcmp(bufferPatron, cmd->bufferOUT, cmd->len);
41
42      // El contenido de ambos debe ser igual
43      CPPUNIT_ASSERT( 0 == res );
44 }
45
46 // Liberacion de recursos
47 void ComandoNXTTest::tearDown()
48 {
49      delete(cmd);
50 }
51
52 // Registro de clase en framework cppunit
53 CPPUNIT_TEST_SUITE_REGISTRATION(ComandoNXTTest);
```

En el código anterior se crea un objeto *cmd* y se manda a llamar al método *llenarFormato* pasándole cinco parámetros: *tipoEnvio, tamaño, tipoComando, subComando y datos.* Con estos parámetros es llenado un buffer de salida y en nuestra clase de prueba se llena un bufferPatron que servirá para comparar con el buffer de salida del objeto *cmd*

si ambos buffers son iguales la prueba estará completa.

5.1.4. Unidad lógica: Usb-nxt

En la prueba de la clase *Usb-nxt* se creará un objeto *usb* y automáticamente se llama al método *encuentra* que es el método encargado de encontrar algún dispositivo USB conectado a la PC por medio de dos parámetros: vendorID y productID.

Usb-nxtTest.cpp

```
1 // Inicilizacion de variable
2 void Usb_nxtTest::setUp ()
3 {
4       usb = new USB();
5 }
6
7 // Prueba
8 void Usb_nxtTest::prueba()
9 {
10      usb->usb_dispositivo = usb->encuentra(LEGO_VENDOR_ID,
 LEGO_NXT_PRODUCT_ID);
11
12      // Verifica que apuntador apunten a algún dispositivo
13      CPPUNIT_ASSERT(usb->usb_dispositivo != NULL);
14 }
15
16 // Liberacion de recursos
17 void Usb_nxtTest::tearDown()
18 {
19      delete(usb);
20 }
```

En *Usb-nxtTest.cpp* solo se presenta el fragmento de código que muestra los métodos que realizan las pruebas principales de la clase *Usb-nxt*. En esta prueba se llama al método *encuentra()* el cual llena una estructura especial para dispositivos usb y regresa un apuntador(*usb-dispositivo*) a esta estructura(línea 10). El objetivo de la prueba es comprobar que fue encontrado un dispositivo usb conectado a la PC y por lo tanto que *usb-dispositivo* apunte a la estructura o en caso contrario, si no fue encontrado, apuntará a nulo(*NULL*).

5.1.5. Unidad lógica: Bluetooth

La prueba de la clase *Bluetooth* se realiza para comprobar que se ha realizado una conexión exitosa a algún dispositivo bluetooth.

BluetoothTest.cpp

```
1 // Inicilizacion de variable
2 void BluetoothTest::setUp ()
3 {
4       bluetooth = new Bluetooth();
5 }
6
7 void BluetoothTest::prueba()
8 {
9       int res = bluetooth->conectar("00:22:B4:86:08:17");
10
11       // Verifica que el resultado sea exitoso
12       CPPUNIT_ASSERT(res == 0);
13 }
14
15 // Liberacion de recursos
16 void BluetoothTest::tearDown()
17 {
18       delete(bluetooth);
19 }
```

El método *conectar()* regresa un valor entero igual a cero en caso de encontrar al dispositivo bluetooth o menor a cero en caso contrario.

5.1.6. Unidad lógica: Bloque

La prueba unitaria para la clase *Bloque* se realizó con la clase de prueba llamada *BloqueTest*. Dado que la clase *Bloque* es la clase base de varias clases de bloques más especializados, en el código se tomó una de estas clases especializadas llamada *BloqueDISPLAY* y se verificó que los miembros de la clase sean inicializados de manera correcta.

BloqueTest.cpp

```
1 // Inicilizacion de variable
2 void BloqueTest::setUp ()
3 {
4      display = new BloqueDISPLAY();
5 }
6
7 // Prueba
8 void BloqueTest::prueba()
9 {
10      CPPUNIT_ASSERT(display->num == 0);
11      CPPUNIT_ASSERT(display->x == 0);
12      CPPUNIT_ASSERT(display->y == 0);
13      CPPUNIT_ASSERT(display->limpiar == 1);
14
15      CPPUNIT_ASSERT(strcmp("", display->texto) == 0);
16 }
17
18 // Liberacion de recursos
19 void BloqueTest::tearDown()
20 {
21      delete(display);
22 }
```

5.2. Pruebas de Integración

En la sección anterior se probaron las clases del proyecto para verificar que sus funciones no contienen errores de ejecución. En esta sección del capítulo describiremos las pruebas de integración. El motivo de realizar pruebas de integración es porque aunque podemos asegurar que las clases no fallarán de manera individual no podemos suponer que cuando se junten las clases que tengan relación entre si no fallarán de manera conjunta.

5.2.1. Plan de pruebas de integración

A continuación se mostrarán las clases que tiene alguna relación y que por lo tanto se les debe realizar una prueba de integración. Las pruebas de las clases relacionadas han sido divididas en grupos.

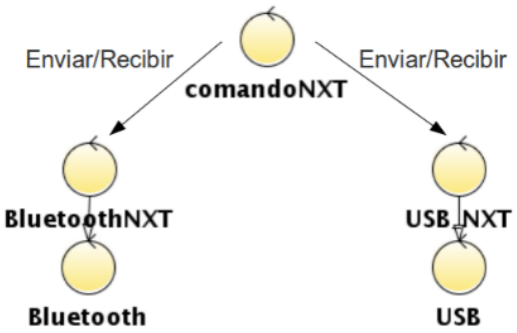

Figura 5.3: Prueba de integración de grupo 1

Grupo 1

El grupo uno se muestra en la Figura 5.3 y en ella se realiza la integración de las clases ComandoNXT, Usb, UsbNXT, Bluetooth y BluetoothNXT; ya que para enviar un comando al robot NXT es necesario crear una cadena de bytes en un formato específico y después puede ser enviado vía usb o bluetooth.

El código que realiza la prueba de integración del grupo 1 se muestra a continuación:

Grupo1Test.cpp

```
1 // Inicilizacion de variable
2 void Grupo1Test::setUp ()
3 {
4      cmd = new ComandoNXT();
5      usb = new USB_NXT();
6 }
7
8 // Prueba
9 void Grupo1Test::prueba()
10 {
11      int enviados = 0;
12      int esperados = 2;
13
```

```
14      // Llenar formato de comando
15      cmd->llenarFormato(TIPOUSB, 0, TIPOCOMANDO, COMANDO, NULL);
16
17      // Enviar comando
18      enviados = cmd->enviar(usb->usb_manejador);
19
20      // Comparar que lo enviado con lo esperado sea igual
21      CPPUNIT_ASSERT( esperados == enviados );
22 }
23
24 // Liberacion de recursos
25 void Grupo1Test::tearDown()
26 {
27      delete(cmd);
28      delete(usb);
29 }
```

Grupo 2

En el grupo dos las clases relacionadas son las de Bloque, Archivo, XmlDoc, XmlDocWriter y XmlDocReader. En el compilador visual se tiene que realizar la traducción de los bloques a un archivo de texto entonces ese es el motivo de la interacción entre la clase Bloque y la clase Archivo, donde cada bloque dentro del programa agrega un porción de código traducido al archivo de texto.

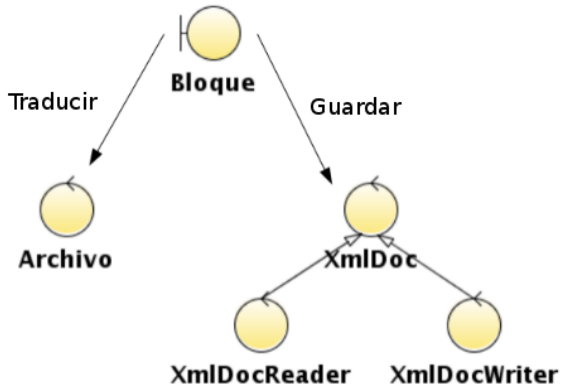

Figura 5.4: Prueba de integración de grupo 2

Para guardar un programa fuente en el compilador visual cada bloque aporta sus datos particulares a un archivo de tipo xml y es por esto que existe una relación entre las clases Bloque y XmlDoc. En la Figura 5.4 se muestra el plan de integración de estas clases y en el código de abajo se muestra la integración de estas clases.

Grupo2Test.cpp

```
0 // Inicilizacion de variables
1 void Grupo2Test::setUp ()
2 {
3      bloque = new BloqueDisplay();
4      archivo = new Archivo(ARCHIVO);
5      xmlDoc = new XmlDoc(XMLDOC);
6 }
7
8 // Prueba
9 void Grupo2Test::prueba()
10 {
11      // Escribe/Traduce el bloque de código en archivo de texto
12      bloque->codigo(archivo);
13
14      // Guarda bloque en archivo XML
15      bloque->escribirXml(xmlDoc);
16
17      // Leer archivos creados y archivos patrones
18      tmpArchivo = leeArchivo(ARCHIVO);
19      auxArchivo = leeArchivo(ARCHIVO_PATRON);
20
21      tmpXmlDoc = leeArchivo(XMLDOC);
22      auxXmlDoc = leeArchivo(XMLDOC_PATRON);
23
24      // El contenido de ambos archivos debe ser igual
25      CPPUNIT_ASSERT( 0 == memcmp(tmpArchivo, auxArchivo, tamano) );
26      CPPUNIT_ASSERT( 0 == memcmp(tmpXmlDoc, auxXmlDoc, tamano) );
27 }
28
29 // Liberacion de recursos
30 void Grupo2Test::tearDown()
31 {
32      delete(bloque);
33      delete(archivo);
34      delete(xmlDoc);
35 }
```

5.3. Ejemplo de programación del Mindstorms NXT con el compilador visual

Para realizar la programación del CPU del *Mindstorms NXT* se construyó el robot basándonos en el tutorial que se encuentra dentro del compilador visual, seleccionando la opción *Tacto:Reacción*. Después se procedió a crear un programa fuente con sentencias visuales con la aplicación, como se muestra en la sección 5.3.1, y por último se envió el programa compilado a la CPU del robot.

Figura 5.5: Robot Mindstorms NXT con la forma Tacto:Reacción.

En la Figura 5.5 se muestra el robot construido con la forma *Tacto:Reacción*. A continuación se presenta una secuencia de imágenes que muestran el comportamiento que tiene el robot después de ser programado.

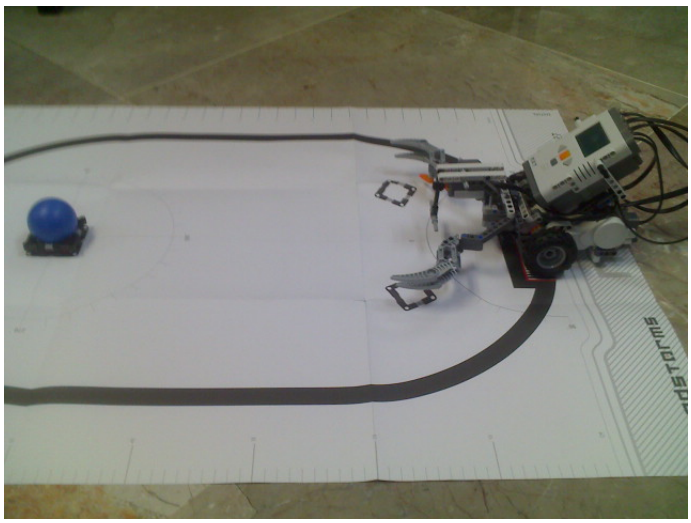

Figura 5.6: Posición inicial del Robot.

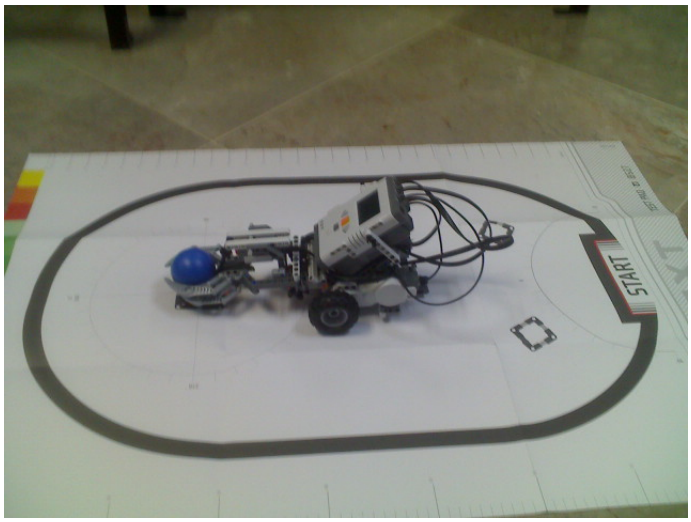

Figura 5.7: Sensor de tacto del robot oprimido y agarre del balón azul.

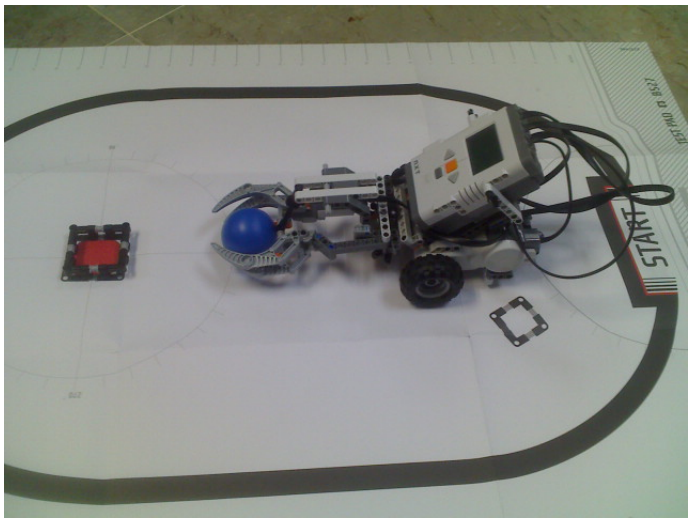

Figura 5.8: Posición final del Robot.

El robot fue programado para iniciar a cierta distancia indeterminada de la esfera (Figura 5.6), los motores inician su marcha en dirección a la esfera y cuando el sensor de tacto es oprimido entonces las garras del robot se cierran para tomar a la esfera (Figura 5.7) y la vez los motores paran. Por último, el robot hace una pausa de diez segundos y los motores vuelven a iniciar la marcha en reversa a una distancia de dos rotaciones de sus ruedas (Figura 5.8).

5.3.1. Programación

La secuencia de bloques que permitieron la programación descrita anteriormente es la que se muestra en la Figura de abajo.

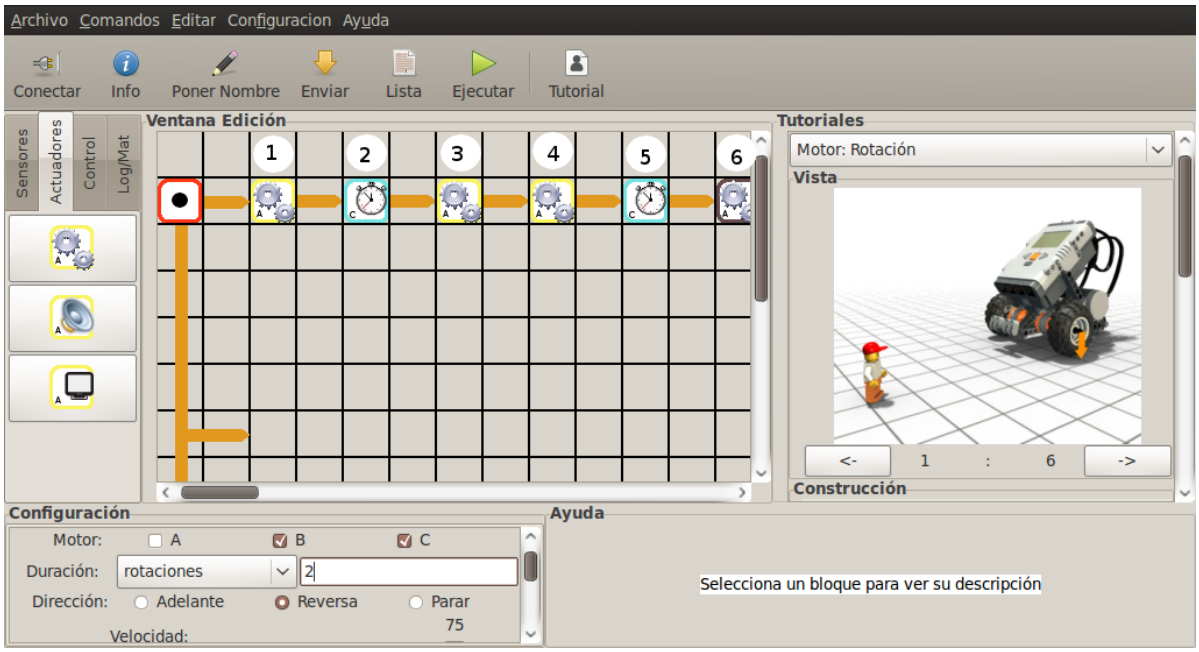

Figura 5.9: Secuencia de bloques en la programación.

En las siguientes Figuras (5.10 - 5.15) se muestran los iconos de cada bloque en la parte izquierda y la configuración es mostrada en la parte derecha de cada una de las imágenes. Los elementos claves de la configuración de cada uno de los bloques se muestran dentro de un cuadro más oscuro.

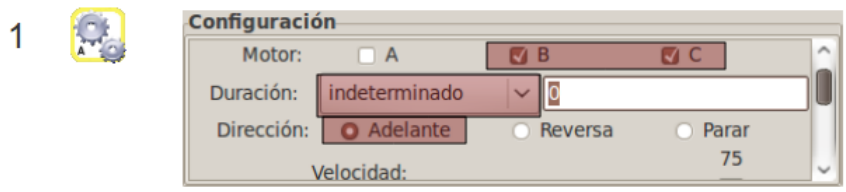

Figura 5.10: Configuración de los motores B y C para avanzar.

En la Figura 5.10 se muestra el primer bloque que se agregó al programa fuente. Este bloque *Motor* se configuró para mover los motores B y C en dirección hacia adelante y

con una duración indeterminada. Esto permitirá que el robot avance hacia adelante en busca de la esfera.

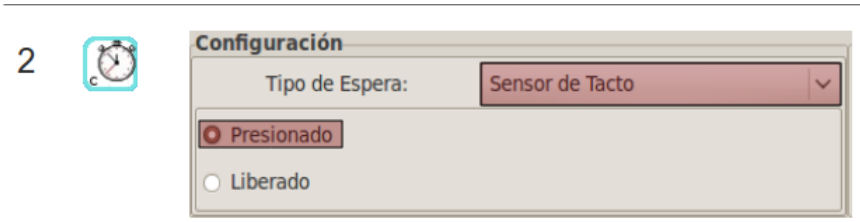

Figura 5.11: Configuración del sensor de Tacto para ser presionado.

La figura 5.11 muestra la configuración del segundo bloque que se agregó al programa. El bloque *Espera* es configurado para esperar a que el sensor de tacto sea presionado, es decir, el bloque que se encuentre después del bloque espera tendrá que esperar a que el sensor de tacto sea presionado para que ejecute su tarea.

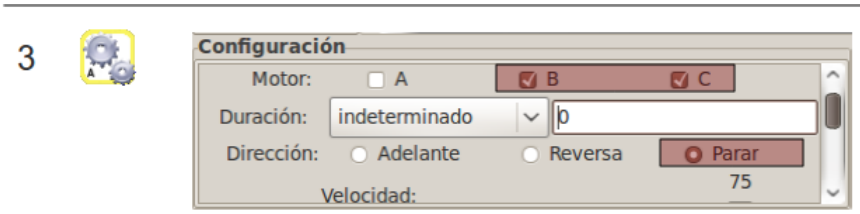

Figura 5.12: Configuración de los motores B y C para parar.

La configuración del tercer bloque, *Bloque Motor*, se muestra en la Figura 5.12. Se modificó nuevamente la configuración de los motores B y C para que después de que el sensor de tacto del paso 2 sea presionado éstos paren su marcha.

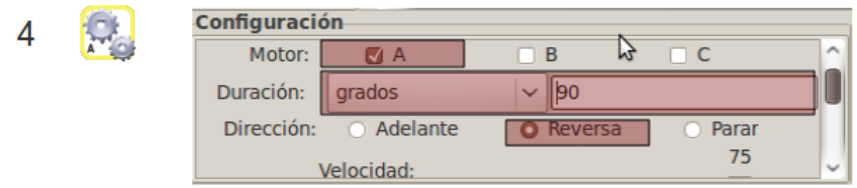

Figura 5.13: Configuración del motor A para cerrar las tenazas.

En el paso 4 el bloque *Motor* se configura para girar 90 grados en dirección de reversa, pero ahora solo accionando al motor A. El motor A esta ligado a las tenazas del robot que se muestran en la Figura 5.5 entonces cuando el motor gira los 90 grados las tenazas se cierran para agarrar a la esfera. La Figura 5.13 muestra ésta configuración.

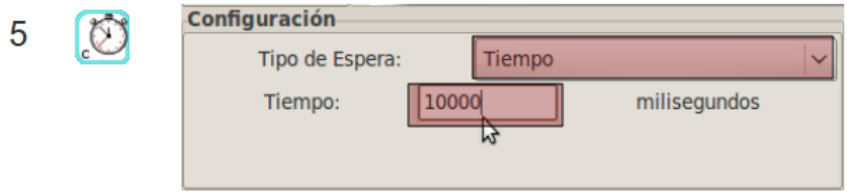

Figura 5.14: Configuración del bloque Espera.

En el paso 5 nuevamente se emplea el bloque *Espera* para que el flujo del programa espere diez segundos para realizar la siguiente acción, como se muestra en la Figura 5.14.

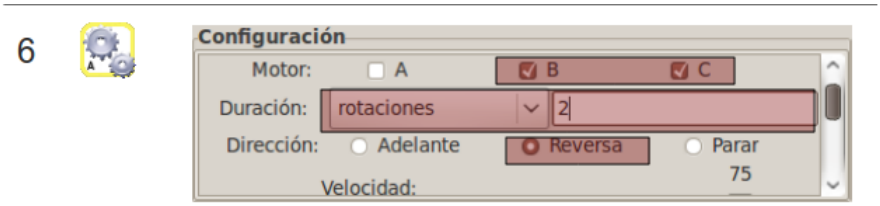

Figura 5.15: Configuración de los motores B y C para girar en reversa.

Después de los diez segundos esperados en el paso 5 los motores B y C son configurados para que realicen dos rotaciones en dirección de reversa y allí termina la programación del robot.

5.4. Resumen

En este capítulo se mostraron las pruebas que se realizaron sobre las unidades lógicas que forman parte del compilador visual, así como también, las pruebas de integración sobre las clases que tienen relación entre ellas. Por último se presentó un ejemplo del robot *Mindstorms NXT* programado con el compilador visual.

En el siguiente capítulo se darán las conclusiones sobre este trabajo de tesis y se comentarán los planes a futuro para el compilador visual.

6

Conclusiones y trabajo a futuro

En este último capítulo se presentan los comentarios finales sobre el proyecto realizado. Hablaremos sobre los objetivos planeados al inicio de la realización del trabajo y si éstos fueron alcanzados una vez terminado el trabajo.

También se mencionan las aportaciones que puede dar la aplicación en ciertos ámbitos de la educación y el trabajo que se pretende realizar en el futuro para que el compilador visual pueda llegar a ser utilizado por el mayor número de usuarios posibles.

6.1. Revisión de Objetivos

El primero de los objetivos fue crear un lenguaje de programación visual, para lo cual se creó un lenguaje que es utilizado por el compilador para crear sentencias o instrucciones en el programa fuente del usuario, que al final será traducido al lenguaje que reconoce la CPU del robot.

El proyecto es considerado como software libre y fue realizado utilizando otras herramientas libres del universo linux, como son:

- Anjuta. Ambiente de Desarrollo Integrado para desarrollar aplicaciones.
- g++. Compilador del lenguaje c++.
- Gtkmm. Biblioteca de funciones y clases para crear elementos gráficos como: ventanas, botones, cajas de texto , etc.; para desplegar información dentro la aplicación.

- Nbc. Compilador de uno de los lenguajes objetos (NXC) generado por el compilador visual.
- Gpasm. Compilador del lenguaje objeto ensamblador generado por el compilador visual.

Otro de los objetivos era la de crear una interfaz que sirviera para comunicar a la PC con el CPU del robot, por lo cual se crearon módulos de software especiales dentro de la aplicación para tener una comunicación bidireccional entre la máquina anfitrión y el robot. Estos módulos especiales son: módulo USB y módulo Bluetooth.

En la última sección de esta tesis se anexa el manual de operación del compilador visual y dentro de la aplicación se creó una ventana de tutoriales para que los usuarios den sus primeros pasos para crear un robot básico.

Al final se puede concluir que los objetivos fueron cumplidos ya que la aplicación aporta un ambiente para desarrollar programas fuentes, compilarlos y enviarlos al robot para que sean ejecutadas las instrucciones de acuerdo a lo programado por el usuario.

6.2. Aportaciones y trabajo a futuro

En el capítulo 1 se hizo mención sobre las instituciones educativas que tienen la necesidad de utilizar herramientas como apoyo para la enseñanza, por lo cual se puede mencionar que el compilador visual puede ser utilizado por algunas de estas instituciones que deseen tener una herramienta para la enseñanza de la robótica básica.

Otra de las aportaciones que se pueden obtener de la aplicación es que el código es abierto y por tanto cualquier persona puede utilizarlo para mejorarlo o personalizarlo a su conveniencia, o en su defecto extraer ciertas partes del código como por ejemplo la sección que maneja la comunicación por bluetooth o usb.

Además se creó un lenguaje visual de programación por medio de la gramática de disposición de imágenes en la cual se establecen las reglas sintácticas que se deben respetar para realizar un programa fuente dentro del compilador visual.

En el futuro se pretende terminar la parte de la aplicación que permita programar a los conjuntos de construcción que manejen en su unidad de procesamiento al *PIC18F4550* de Microchip, que es la segunda parte del proyecto general de apoyo a la enseñanza de la robótica básica.

También se pensó en realizar una extensión del compilador visual que permitiera ver la acción que tendría el robot de forma simulada y antes de enviarla a la CPU. Es decir, crear una parte del software que permitiera realizar la simulación de la programación del robot en la máquina anfitrión para que el usuario pudiera ver el comportamiento del

robot y comparar si ésta era la acción que se esperaba. Debido al trabajo que implica realizar esta parte de la aplicación se pretende realizar en un futuro como un trabajo independiente.

Además esperamos que el compilador visual pueda ser introducido en los repositorios de software libre para que puedan ser descargados desde cualquier PC con linux y con una conexión a Internet, lo que permitirá que el software madure por medio del intercambio de información entre los usuarios y el programador del software. Por intercambio de información entenderemos a el reporte de errores de la aplicación por parte de los usuarios y las consecutivas mejoras y versiones del compilador por medio del programador.

6.3. Resumen

En este capítulo se dieron los comentarios finales del trabajo realizado durante la elaboración del compilador visual como herramienta para apoyar a la enseñanza de la robótica básica.

Y en la siguiente y ultima parte del documento se encuentran en forma de anexo el código fuente del compilador visual y el manual de usuario de la aplicación.

Referencias

[Ahern] Terence C. Ahern. *The effectiveness of visual programming for model building in middle school.*, IEEE Conferences, Frontiers in Education Conference, 2008. FIE 2008. 38th Annual, Pag. S3D-8 - S3D-13. 2008.

[Aho] Alfred V. Aho, Monica S. Lam, Ravi Sethi y Jeffrey D. Ullman. *Compilers: principles, techniques tools.*, Addison Wesley, Second Edition, Pag. 1 - 2, 2007.

[Boshernitsan] Marat Boshernitsan y Michael Downes. *Visual programming languages: A survey.*, University of California, Berkeley, California 94720, Reporte Técnico, Pag. 6 , Diciembre 2004.

[Chang] Shi-Kuo Chang. *Visual Lenguages and Visual Programming.*, Plenum Press, Pag. vii, 3, 1990.

[Golin] Eric J. Golin, S. P. Reiss. *The specification of visual lenguage syntax.*, Journal of Visual Lenguages and Computing, vol. 1, pag. 141-157, Junio de 1990.

[Kim] Seung Han Kim, Jae Wook Jeon. *Programming LEGO mindstorms NXT with visual programming.*, IEEE Conferences, ICCAS '07. International Conference on Digital Object Identifier: 10.1109/ICCAS.2007.4406778, Pag. 2468 - 2472. 2007.

[Miglino] Orazio Miglino, Henrik Hautop Lund, Maurizio Cardaci. *Robotics as an educational tool.*, Journal of Interactive Learning Research, Vol. 10, No. 1. Pag. 25-26, Abril 1999.

[Mindell] David Mindell. *LEGO Mindstorms. The Structure of an Engineering (R)evolution*, web.mit.edu/6.933/www/Fall2000/LegoMindstorms.pdf, Diciembre 2000.

[Stallman] Richard M. Stallman. *Filosofía del Software Libre*, www.gnu.org/philosophy/free-sw.html, Octubre 2009.

[Thomaz] Sarah Thomaz, Akynara Aglaé, Carla Fernandes, Renata Pitta, Samuel Azevedo, Aquiles Burlamaqui, Alzira Silva, Luiz M. G. Gonçalves. *RoboEduc: A Pedagogical Tool to support Educational Robotics.*, Frontiers in Education Conference, 2009. FIE '09 39th IEEE, Diciembre 2009.

[Zhang] Houxiang Zhang, Weining Zheng, Shengyong Chen, Jianwei Zhang, Wei Wang y Guanghua Zong. *Flexible Educational Robotic System for a Practical Course.*, Integration Tecnology, 2007. ICIT '07. IEEE International Conference on, Agosto 2007.

A

Código fuente

En este anexo se presentan algunos fragmentos de código del proyecto compilador visual. Se eligió poner los archivos de código más sobresalientes del proyecto, por ejemplo se incluyó la clase de comunicación por bluetooth pero se omitió la clase usb. También se agregó la clase ComandoNXT que contiene a los protocolos de alto nivel que reconoce el robot Mindstorms NXT y además la clase Archivo que se utiliza para guardar la traducción del programa fuente. A continuación se muestran las clases seleccionadas que forman parte del compilador visual.

A.1. Archivo

La clase *Archivo* es la encargada de realizar las operaciones de creación, llenado y guardado de un archivo en el sistema que contendrá la traducción del programa fuente, es decir, contendrá al programa objeto en forma de texto, en un lenguaje llamado *NXC(Not eXactly C)*.

Archivo.h

```
#ifndef ARCHIVO_H
#define ARCHIVO_H
```

```
#include <fstream>

class Archivo
{
public:
    std::fstream *io; // Stream de entrada/salida

    Archivo(char*); // Crea, abre e inicializa un archivo
    ~Archivo(); // Finaliza y Cierra un archivo
    int inicializa();
    int finaliza();
    int agregaLinea(char* linea); // Agrega un linea nueva al archivo
    int agregaCadena(char* cad); // Agrega cadena a al archivo
};

#endif /* ARCHIVO_H */
```

Archivo.cc

```
#include <iostream>

#include "Archivo.h"

using namespace std;

int Archivo::inicializa()
{}

Archivo::Archivo(char* arch)
{
    io = new fstream(arch, ios::out);  // Crea o reemplaza un archivo

    if(!io)
        cout << "Error al abrir archivo.\n";

    inicializa();
}

int Archivo::finaliza()
{
*io << "task main() " << endl;
```

```
    *io << "{ " << endl;
*io << "Precedes(tarea1,tarea2,tarea3); " << endl;
    *io << "}" << endl;
}

Archivo::~Archivo()
{
    finaliza();
    io->close();
}

int Archivo::agregaLinea(char* linea)
{
    *io << linea << endl;
}

int Archivo::agregaCadena(char* cad)
{
    *io << cad;
}
```

A.2. ComandoNXT

La clase *ComandoNXT* maneja los comandos que interpreta el robot *Mindstorm NXT*. Los comandos deben tener un formato específico y dichos comandos pueden ser enviados vía USB o Bluetooth.

ComandoNXT.h

```
#include <usb.h>

#define NXT_BUFFER_SIZE 64
#define NXT_DATA_BUFFER_SIZE 256

#define USB_OUT_ENDPOINT  0x01
#define USB_IN_ENDPOINT  0x82
#define USB_TIMEOUT  1000

class ComandoNXT
{
```

```
public:
unsigned char bufferOUT[NXT_BUFFER_SIZE];
unsigned char bufferIN[NXT_BUFFER_SIZE];

int len;

ComandoNXT();
void llenarFormato(int tipo, int size, unsigned int CommandType,
    unsigned int CommandByte, const unsigned char Info[]);
void reset();
int enviar(usb_dev_handle *dev);
int recibir(usb_dev_handle *dev,int sizeCmd);
int enviar (int sockDisp);
int recibir (int sockDisp, int sizeRetorno);
};
```

ComandoNXT.cpp

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include "ComandoNXT.h"
using namespace std;

ComandoNXT::ComandoNXT()
{
    reset ();
}

void ComandoNXT::reset ()
{
    memset(&bufferOUT, 0, NXT_BUFFER_SIZE);
    memset(&bufferIN, 0, NXT_BUFFER_SIZE);
    len = 0;
}

void ComandoNXT::llenarFormato(int tipo, int size, unsigned int CommandType,
    unsigned int CommandByte, const unsigned char Info[])
{
memset(&bufferOUT, 0, NXT_BUFFER_SIZE);
    unsigned short int si;
```

```
    if (tipo)
    { //Bluetooth
        si  =  2+size;
        memcpy(&bufferOUT, &si, 2);

        bufferOUT[2] = CommandType;
        bufferOUT[3] = CommandByte;
        len += 2;
}
    else
    { //Usb
        bufferOUT[0] = CommandType;
        bufferOUT[1] = CommandByte;
    }


for(int i = len; i < size+len; i++)
{
bufferOUT[i+2] = Info[i-len];
}
    len += size + 2;
}


//Enviar via USB
int ComandoNXT::enviar(usb_dev_handle *dev)
{
return usb_bulk_write(dev, USB_OUT_ENDPOINT, (char*)(&bufferOUT[0]),
    len, USB_TIMEOUT);
}

//Recibir via USB
int ComandoNXT::recibir(usb_dev_handle *dev,int sizeRetorno)
{
return usb_bulk_read(dev, USB_IN_ENDPOINT, (char*)(&bufferIN[0]),
    sizeRetorno, USB_TIMEOUT);
}

//Enviar via Bluetooth
int ComandoNXT::enviar (int sockDisp)
{
    return write(sockDisp, &bufferOUT, len);
}
```

```
//Recibir via bluetooth()
int ComandoNXT::recibir(int sockDisp, int sizeRetorno)
{
    ssize_t xr;
    unsigned short int sr;
    struct timeval timeout = {2,0};
    fd_set set;

    FD_ZERO(&set);
    FD_SET(sockDisp, &set);

    int res;
    while  (res = select(sockDisp+1, &set, 0, 0, &timeout) < 0)

    while ((xr = read(sockDisp, &bufferIN, 2)) == -1)

    if (xr != 2) {
    perror("read: ");
    return(-1);
}

    memcpy(&sr, &bufferIN[0], 2);

    while ((xr = read(sockDisp, &bufferIN, (int)sr)) == -1)

    if (xr != (int)sr) { perror("Error al intentar leer, xr != sr\n"); }

    return 0;
}
```

A.3. BluetoothNXT

La clase *BluetoothNXT* al igual que la clase anterior utiliza a la clase *ComandoNXT* para realizar operaciones de intercambio de información. La diferencia es que el envío de la información es a través del protocolo *Bluetooth*.

bluetooth.h

```
#include <bluetooth/bluetooth.h>
```

```
#include <bluetooth/rfcomm.h>
#include "ComandoNXT.h"

#define NXT_BUFFER_SIZE 64
#define NXT_DIRECT_COMMAND      0x00
#define NXT_SYSTEM_COMMAND  0x01
#define GET_DEVICE_INFO 0x9B
#define NXT_SET_NAME 0x98
#define NXT_START_PROGRAM 0x00
#define NXT_OPEN_WRITE 0x81
#define NXT_WRITE 0x83
#define NXT_CLOSE 0x84
#define NXT_DELETE 0x85
#define NXT_FIND_FIRST 0x86
#define NXT_FIND_NEXT 0x87

struct listaDisp
{
char name[20];
char addr[19];
int canal;
};

class Bluetooth
{
protected:
int sockDisp;
struct sockaddr_rc addr;
public:
~Bluetooth();
int buscarDispositivos(int servicio);
int conectar(char *btaddr);

struct listaDisp disp[30]; // Lista de dispositivos
};

class BluetoothNXT: public Bluetooth
{
public:
char nombreNXT[20];
char dirBluetooth[20];
char memoriaNXT[10];
char nomProg[20][20]; //Lista de programas
```

```
int startPrograma(const char *nombre);
int info();
int setName(const char *nombre);
int upPrograma(char* fname);
int listaProgs();
int chekar_conexion();

int enviarArchivo(char *);
int buscarArchivo(char *);
int borrarArchivo(char *);
};
```

bluetooth.cc

```
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <libgen.h>
#include <fcntl.h>
#include <sys/socket.h>

#include <gtkmm/messagedialog.h>

// Librerias para comunicacion bluetooth por rfcomm
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

// Librerias para buscar dispositivos
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>

// Librerias para buscar servicios
#include <bluetooth/sdp.h>
#include <bluetooth/sdp_lib.h>

#include "bluetooth.h"

using namespace std;

Bluetooth::~Bluetooth(){
if(sockDisp) close(sockDisp);
}
```

```
// Busca dispositivo bluetooth y un servicio especifco
int Bluetooth::buscarDispositivos(int servicio)
{
    inquiry_info *ii = NULL; // Apuntador a informacion de investigacion
    int max_rsp, num_rsp; // Maximo de respuestas y numero de resp.
    int dev_id, sock; // ID de dispositivo y sockect
    int len, flags; //
    int i, cont = 0;
    char addr[19] = { 0 }; // Direccion bluetooth
    char name[20] = { 0 }; // Nombre amigable

    cout << "Dentro de buscarDispositivos" << endl;

    dev_id = hci_get_route(NULL); // Número del primer adaptador bt disponible
    sock = hci_open_dev( dev_id ); // Abre el dispositivo bt recuperado
    if (dev_id < 0 || sock < 0)
    {
        perror("Abriendo socket");
        return -1;
    }

    len  = 4; // 1 Segundo(s) * 1.28 de busqueda
    max_rsp = 32; // Maximo de respuestas
// Limpia el cache anterior de dispositivos encontrados
    flags = IREQ_CACHE_FLUSH;
// Reserva de recursos para 255 disp foraneos
ii = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));

// Realiza una búsqueda de dispositivos Bluetooth y
//devuelve una lista de dispositivos detectados
    num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL, &ii, flags);
    if( num_rsp < 0 )
    perror("hci_inquiry");
    else
        if(num_rsp == 0)
            cout << "No se encontraron dispositivos" << endl;


    for (i = 0; i < num_rsp; i++)  // Por cada disp encontrado
    {
        ba2str(&(ii+i)->bdaddr, addr); // Dir blue de 6 bytes a texto
        memset(name, 0, sizeof(name));
// Va por el nombre amigable del disp actual
```

```
        if (hci_read_remote_name(sock, &(ii+i)->bdaddr, sizeof(name),
            name, 0) < 0)
        strcpy(name, "[desconocido]"); // Si falla le pone desconocido
        printf("%s  %s\n", addr, name);

// Buscar servicio spp de dispositivo
char serv_class_ID[]="0x1101"; /* Perfil Puerto Serial (SPP)*/

uuid_t svc_uuid;
  int err;
sdp_list_t *lista_resp = NULL, *lista_busc, *lista_atrib;
  sdp_session_t *sesion = 0;

uint32_t clase = 0;
  int num;

// Convierte cadena class id a int
sscanf(serv_class_ID + 2, "%X", &num);
clase = num;
cout << "Service Class" << clase << endl;

if (clase)
{
    uint16_t clase16 = clase & 0xffff;
    sdp_uuid16_create(&svc_uuid, clase16);
    cout << "Creado un uuid de 16 bits" << endl;
}

// Se conecta al servidor SDP del dispositivo foraneo
cout << "sdp_connect" << endl;
  sesion = sdp_connect( BDADDR_ANY, &(ii+i)->bdaddr, SDP_RETRY_IF_BUSY );
if(sesion == (sdp_session_t *)NULL)
{
cout << "Fallo sesion" << endl;
}
else
{
  // Especificamos el UUID (servicio) que estamos buscando
cout << "sdp_list_append" << endl;
  lista_busc = sdp_list_append( NULL, &svc_uuid );

  // Especificamos que queremos una lista de todos los atributos
// de apliacion que encuentre
  uint32_t rango = 0x0000ffff;
```

```
cout << "list_append_2!!" << endl;
  lista_atrib = sdp_list_append( NULL, &rango );

  // Consigue una lista de registros de servicio
cout << "sdp_service_serach_attr_req" << endl;
  err = sdp_service_search_attr_req( sesion, lista_busc, \
  SDP_ATTR_REQ_RANGE, lista_atrib, &lista_resp);

sdp_list_t *r = lista_resp;

  // Para cada registro de servicio encontrado
cout << "Para cada servicio encontrado" << endl;
  for (; r; r = r->next )
{
sdp_record_t *rec = (sdp_record_t*) r->data;
  sdp_record_print(rec);
  cout << "RecHandle de servicio: " << rec->handle << endl;
  sdp_list_t *lista_proto;

  // Consigue lista de protocolos
if( sdp_get_access_protos( rec, &lista_proto ) == 0 )
{
  sdp_list_t *p = lista_proto;

  // Para cada secuencia de protocolo encontrado
  for( ; p ; p = p->next )
{
  sdp_list_t *pds = (sdp_list_t*)p->data;

  // Para cada lista de protocolo de secuencia especifica
  for( ; pds ; pds = pds->next )
{

  // Atributos de protocolo
  sdp_data_t *d = (sdp_data_t*)pds->data;
  int proto = 0;
  for( ; d; d = d->next )
{
  switch( d->dtd )
{
  case SDP_UUID16:
  case SDP_UUID32:
  case SDP_UUID128: proto = sdp_uuid_to_proto( &d->val.uuid ); break;
  case SDP_UINT8: if( proto == RFCOMM_UUID )
```

```
printf("Canal rfcomm: %d\n",d->val.int8);

                  disp[cont].canal = (int)d->val.int8;
                  memcpy(&disp[cont].addr, &addr[0], 19);
                  memcpy(&disp[cont].name, &name[0], 20);
                  cont++;
  break;
  }
  }
  }
  sdp_list_free( (sdp_list_t*)p->data, 0 );
  }
  sdp_list_free( lista_proto, 0 );
  }

  // libera registros
  sdp_record_free( rec );
  }

  sdp_close(sesion);
}
    }

    free( ii );
    close( sock );
    return cont;
 }


int Bluetooth::conectar(char *btaddr)
{
    //Obtiene direccion del NXT
    char dest[18];
    memcpy(&dest, btaddr, 18);

    // Asigna un socket
    sockDisp = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

    // Parametros de conexión
    addr.rc_family = AF_BLUETOOTH;
    addr.rc_channel = (uint8_t) 1;
    str2ba( dest, &addr.rc_bdaddr );

    int status = connect(sockDisp, (struct sockaddr *)&addr, sizeof(addr));
```

```
    if( status == 0 )
   cout<< "Conectado" << endl;
else
    {
    cout<< "Falló conexión" << endl;
        return -1;
    }

    int res = fcntl(sockDisp, F_SETFL, O_NONBLOCK);
    if (res == 0)
   cout<< "nonblock listo"<< endl;
else {
    cout<< "Falló nonblock: "<< endl;
    return -2;
}

    return 0;
}

int BluetoothNXT::info()
{
unsigned int tipoComando;
unsigned int comando;
unsigned char info[NXT_BUFFER_SIZE-2];

ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = GET_DEVICE_INFO;
ComandoNXT1.llenarFormato(1/*bluetooth*/,0,tipoComando, comando, info);

cout << "\nSolicitando información del NXT..." << endl;;
int res = 0;
res = ComandoNXT1.enviar(sockDisp);

if(res < 0)
cout <<"Error enviando comando" << endl;
else
        cout <<"Exito enviando comando: " << res << endl;

res = ComandoNXT1.recibir(sockDisp,32);

if(res < 0)
cout <<"Error recibiendo datos" << endl;
```

```
else
        cout <<"Exito recibiendo comando" << endl;

cout << "Datos de Retorno: " << endl;

char addr[4];
for(int i=0; i<24; i++)
{
if(i >=0 && i<3) printf("<%02X>", ComandoNXT1.bufferIN[i]);
if(i >=18 && i<=23)
        {
            sprintf (addr, "%02X:", ComandoNXT1.bufferIN[i]);
            strcat(dirBluetooth, addr);
        }
}
memcpy(&nombreNXT, &ComandoNXT1.bufferIN[3], 15);
dirBluetooth[17] = '\0';

int memFlash;
memcpy(&memFlash, &ComandoNXT1.bufferIN[29], 4);
sprintf (memoriaNXT, "%d", memFlash);
}

int BluetoothNXT::setName(const char *nombre)
{
unsigned char tipoComando;
unsigned char comando;
char info[15];
int res = 0;

ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_SET_NAME;
strcpy(info, nombre);
ComandoNXT1.llenarFormato(1/*bluetooth*/, 9, tipoComando, comando,
    (unsigned char*)(&info[0]));

cout <<"\nCambiando nombre del ladrillo NXT..." << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando comando" << endl;

res = ComandoNXT1.recibir(sockDisp,5);
```

```
if(res < 0)
cout <<"Error recibiendo comando" << endl;

cout <<"Datos de Retorno: " << endl;
for(int i=0; i<4; i++)
{
if(i >=0 && i<3) printf("<%02X>", ComandoNXT1.bufferIN[i]);
}
cout << endl;
}


int BluetoothNXT::startPrograma(const char *nombre)
{
unsigned char tipoComando;
unsigned char comando;
char info[15];
int res = 0;

ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_DIRECT_COMMAND;
comando = NXT_START_PROGRAM;

memset(&info, 0, 15);
memcpy(&info[0], &nombre[0], strlen(nombre));
ComandoNXT1.llenarFormato(1/*bluetooth*/, strlen(nombre)+1,
    tipoComando, comando, (unsigned char*)(&info[0]));

cout <<"\nIniciando programa " << nombre << " en NXT..." << endl;
res = ComandoNXT1.enviar(sockDisp);
cout << "Enviados: " << res << endl;

if(res < 0)
cout <<"Error enviando comando" << endl;

res = ComandoNXT1.recibir(sockDisp,5);
if(res < 0)
cout <<"Error recibiendo comando" << endl;

cout << "Datos de Retorno: ";
for(int i=0; i<4; i++)
{
if(i >=0 && i<3) printf("<%02X>", ComandoNXT1.bufferIN[i]);
```

```
}
cout << endl;
}


// Cheka la conexion del dispositivo
int BluetoothNXT::chekar_conexion()
{
unsigned char tipoComando;
unsigned char comando;
char info[15];
int res = 0;

ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_DIRECT_COMMAND;
comando = 0x0D; // Keep Alive

ComandoNXT1.llenarFormato(1 /*bluetooth*/,0,tipoComando,
    comando, (unsigned char*)(&info[0]));

cout <<"\nCheka estatus NXT..." << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando comando" << endl;

res = ComandoNXT1.recibir(sockDisp,32);
cout <<"Res: " << res << endl;
if(res < 0)
return 0;

return 1;
}

int BluetoothNXT::enviarArchivo(char *fname)
{
unsigned char tipoComando;
unsigned char comando;
char info[25];
int res = 0;

//Abre archivo para lectura
FILE* inf = fopen(fname,"r");
int fno = fileno(inf);
```

```
//Separa ruta y nombre de archivo
char *dirc, *basec, *bname, *dname;
dirc = strdup(fname);
basec = strdup(fname);
dname = dirname(dirc);
bname = basename(basec);

cout << "En upPrograma" << endl;
cout << "Dir: " << dname << endl;
cout << "File: " << bname << endl;

struct stat infoFile;
stat(fname, &infoFile);

int fsize = (int)infoFile.st_size;
cout << "File Size: " << fsize << endl;

ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_OPEN_WRITE; // Open write
memset(&info, 0, 25);
memcpy(info, bname, strlen(bname));

memset(&info[20], 0, 5);
memcpy(&info[20],&fsize,4);

ComandoNXT1.llenarFormato(1/*bluetooth*/, 25, tipoComando,
    comando, (unsigned char*)(&info[0]));

cout << "\nOpen write" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando Open write" << endl;

res = ComandoNXT1.recibir(sockDisp,32);
if(res < 0)
cout <<"Error recibiendo Open write" << endl;

cout <<"Datos de Retorno: ";
for(int i=0; i<4; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);
```

```
unsigned char fhandler = ComandoNXT1.bufferIN[3];
cout << endl;

int maxpiece = 24;
unsigned char status, buff[maxpiece];

int nbb;

int berr=false;
do
{
nbb = read(fno, &buff[0], maxpiece);
cout << endl << "Bytes leidos: " << nbb << endl;
if (nbb)
{
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_WRITE; //Write command
memset(&info, 0, 25);

info[0] = fhandler;
memcpy(&info[1], &buff[0], nbb);
ComandoNXT1.llenarFormato(1/*bluetooth*/, nbb+1, tipoComando,
    comando, (unsigned char*)(&info[0]));

res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando Open write" << endl;

res = ComandoNXT1.recibir(sockDisp,32);
if(res < 0)
cout <<"Error recibiendo Open write" << endl;

cout <<"Datos de Retorno: " << endl;
for(int i=0; i<3; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

status = ComandoNXT1.bufferIN[2];
}
}while (nbb && !status);

fclose(inf);
```

```
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_CLOSE; //Close command
memset(&info, 0, 20);
info[0] = fhandler;

ComandoNXT1.llenarFormato(1/*bluetooth*/, 1, tipoComando, comando,
    (unsigned char*)(&info[0]));

cout <<"\nClose command\n" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando Open write" << endl;

res = ComandoNXT1.recibir(sockDisp,32);
if(res < 0)
cout <<"Error recibiendo Open write" << endl;

cout << "Datos de Retorno: ";
for(int i=0; i<3; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

if (ComandoNXT1.bufferIN[2] == 0)
return 1;
else
return 0;
}

int BluetoothNXT::borrarArchivo(char *archivo)
{
unsigned char tipoComando;
unsigned char comando;
char info[25]; //, arch[20];
int res = 0;

ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_DELETE;

memset(&info, 0, 25);
memcpy(info, archivo, strlen(archivo));

ComandoNXT1.llenarFormato(1/*bluetooth*/, 20, tipoComando,
```

```
    comando, (unsigned char*)(&info[0]));

cout <<"\nDelete" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando delete" << endl;

res = ComandoNXT1.recibir(sockDisp,28);
if(res < 0)
cout <<"Error recibiendo delete" << endl;

cout <<"Datos de Retorno: ";
for(int i=0; i<28; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

if(ComandoNXT1.bufferIN[2] == 0)
return 1;
else
return 0;
}

int BluetoothNXT::buscarArchivo(char *archivo)
{
unsigned char tipoComando;
unsigned char comando;
char info[25]; //, arch[20];
int res = 0;

// Cheka si existe el archivo en NXT ------------
ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_FIND_FIRST;
memset(&info, 0, 25);
memcpy(info, archivo, strlen(archivo));

ComandoNXT1.llenarFormato(1/*bluetooth*/, 20, tipoComando,
    comando, (unsigned char*)(&info[0]));

cout <<"\nFind first" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando find first" << endl;
```

```
res = ComandoNXT1.recibir(sockDisp,28);
if(res < 0)
cout <<"Error recibiendo find first" << endl;

cout <<"Datos de Retorno: " << endl;
for(int i=0; i<28; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

unsigned char res_find = ComandoNXT1.bufferIN[2];

// Si exitoso regresa un 1
if(res_find == 0)
return 1;
else
return 0;
}

int BluetoothNXT::listaProgs()
{
int cont=0;
unsigned char tipoComando;
unsigned char comando;
char info[25], arch[20];
int res = 0;

ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_FIND_FIRST;   // FIND FIRST
memset(&info, 0, 20);
memcpy(info, "*.rxe", 5);

    ComandoNXT1.llenarFormato(1/*bluetooth*/, 20, tipoComando,
       comando, (unsigned char*)(&info[0]));

cout <<"\nFind first" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando find first" << endl;

res = ComandoNXT1.recibir(sockDisp,28);
if(res < 0)
cout <<"Error recibiendo find first" << endl;
```

```
cout <<"Datos de Retorno: ";
for(int i=0; i<28; i++)
{
printf("<%02X>", ComandoNXT1.bufferIN[i]);
}

unsigned char fhandler = ComandoNXT1.bufferIN[3];
memcpy(&nomProg[cont], &ComandoNXT1.bufferIN[4], 20);
cout << nomProg[cont] << endl;

char status;

do
{
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_FIND_NEXT;  // FIND NEXT FILE
memset(&info, 0, 25);

info[0] = fhandler;
ComandoNXT1.llenarFormato(1/*bluetooth*/, 1, tipoComando,
    comando, (unsigned char*)(&info[0]));

res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando Open write" << endl;

res = ComandoNXT1.recibir(sockDisp,28);
if(res < 0)
cout <<"Error recibiendo Open write" << endl;

cout << "Datos de Retorno: ";
for(int i=0; i<28; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

cont++;
memcpy(&nomProg[cont], &ComandoNXT1.bufferIN[4], 20);
printf("%s \n", nomProg[cont]);

status = ComandoNXT1.bufferIN[2];
fhandler = ComandoNXT1.bufferIN[3];
}while (status == 0);

return --cont;
```

```
}


int BluetoothNXT::upPrograma(char* fname)
{
unsigned char tipoComando;
unsigned char comando;
char info[25];
int res = 0;

//Abre archivo para lectura
FILE* inf = fopen(fname,"r");
int fno = fileno(inf);

//Separa ruta y nombre de archivo
char *dirc, *basec, *bname, *dname;
dirc = strdup(fname);
basec = strdup(fname);
dname = dirname(dirc);
bname = basename(basec);

cout << "En upPrograma" << endl;
cout << "Dir: " << dname << endl;
cout << "File: " << bname << endl;

// Cheka si ya existe el archivo en NXT --------------------
ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_FIND_FIRST; //0x86;    // FIND FIRST
memset(&info, 0, 25);
memcpy(info, bname, strlen(bname));

ComandoNXT1.llenarFormato(1/*bluetooth*/, 20, tipoComando,
    comando, (unsigned char*)(&info[0]));

cout <<"\nFind first" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando find first" << endl;

res = ComandoNXT1.recibir(sockDisp,28);
if(res < 0)
cout <<"Error recibiendo find first" << endl;
```

```
cout <<"Datos de Retorno: ";
for(int i=0; i<28; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

unsigned char res_find = ComandoNXT1.bufferIN[2];

// Si existe borra el archivo
if(res_find == 0)
{
//ComandoNXT ComandoNXT1;
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_DELETE;

ComandoNXT1.llenarFormato(1/*bluetooth*/, 20, tipoComando,
    comando, (unsigned char*)(&info[0]));

cout <<"\nDelete" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando delete" << endl;

res = ComandoNXT1.recibir(sockDisp,28);
if(res < 0)
cout <<"Error recibiendo delete" << endl;

cout <<"Datos de Retorno: ";
for(int i=0; i<28; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

if(ComandoNXT1.bufferIN[2] == 0)
 cout <<"Borrardo!!" << endl;
}

//-------------------------------------------------------------
// Y luego envia el archivo nuevo
//-------------------------------------------------------------
struct stat infoFile;
stat(fname, &infoFile);

int fsize = (int)infoFile.st_size;
cout << "File Size: " << fsize << endl;
```

```
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_OPEN_WRITE; //0x81;  // Open write
memset(&info, 0, 25);
memcpy(info, bname, strlen(bname));

 memset(&info[20], 0, 5);
memcpy(&info[20],&fsize,4);

ComandoNXT1.llenarFormato(1/*bluetooth*/, 25, tipoComando,
    comando, (unsigned char*)(&info[0]));

cout <<"\nOpen write" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando Open write" << endl;

res = ComandoNXT1.recibir(sockDisp,32);
if(res < 0)
cout <<"Error recibiendo Open write" << endl;

cout <<"Datos de Retorno: ";
for(int i=0; i<4; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

unsigned char fhandler = ComandoNXT1.bufferIN[3];
printf("\n");

int maxpiece = 24;
char status, buff[maxpiece];

int nbb;

int berr=false;
do
{
nbb = read(fno, &buff[0], maxpiece);
cout << endl << "Bytes leidos: " << nbb << endl;
if (nbb)
{
ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_WRITE; //0x83;   //Write command
memset(&info, 0, 25);
```

```
info[0] = fhandler;
memcpy(&info[1], &buff[0], nbb);
ComandoNXT1.llenarFormato(1/*bluetooth*/, nbb+1,
    tipoComando, comando, (unsigned char*)(&info[0]));

res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando Open write" << endl;

res = ComandoNXT1.recibir(sockDisp,32);
if(res < 0)
cout <<"Error recibiendo Open write" << endl;

cout <<"Datos de Retorno: ";
for(int i=0; i<3; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

status = ComandoNXT1.bufferIN[2];
}
}while (nbb && !status);

fclose(inf);


ComandoNXT1.reset ();
tipoComando =  NXT_SYSTEM_COMMAND;
comando = NXT_CLOSE; //0x84; //Close command
memset(&info, 0, 20);
info[0] = fhandler;

ComandoNXT1.llenarFormato(1/*bluetooth*/, 1, tipoComando,
    comando, (unsigned char*)(&info[0]));

cout <<"\nClose command" << endl;
res = ComandoNXT1.enviar(sockDisp);
if(res < 0)
cout <<"Error enviando Open write" << endl;

res = ComandoNXT1.recibir(sockDisp,32);
if(res < 0)
cout <<"Error recibiendo Open write" << endl;

cout <<"Datos de Retorno: ";
```

```
for(int i=0; i<3; i++)
printf("<%02X>", ComandoNXT1.bufferIN[i]);

Gtk::MessageDialog dialogo1("");
if (ComandoNXT1.bufferIN[2] == 0)
dialogo1.set_message("Archivo enviado!");
else
dialogo1.set_message("Error al enviar archivo!");

dialogo1.run();
}
```

B

Manual de usuario

El compilador visual, cvMindstorms, es una aplicación hecha para crear programas que permitan programar el comportamiento del conjunto de construcción Mindstorms de Lego y también al conjunto de construcción resultado de la tesis 'Diseño de un sistema de desarrollo para la enseñanza de la robótica', creado en el Centro de Investigación en Computación del Instituto Politécnico Nacional. cvMindstorms está creada para ser instalada en plataformas Linux.

Uno de los objetivos de cvMindstorms es crear una interfaz similar al programa que viene con el Mindstorms, que es una aplicación creada para la plataforma Windows. La facilidad de uso y el permitir programar al Mindstorms sin tener conocimiento previos de robótica, por parte de los usuarios, es el principal motivo para presentar una interfaz similar.

En este documento se presenta cada una de las características del cvMindstorms y las diferentes tareas que pueden realizar los usuarios desde guardar y recuperar los programas fuentes que creen, hasta el envío del programa al CPU del robot.

B.1. Requerimientos mínimos e Instalación del compilador

Para realizar la instalación de la aplicación es necesario tener en la máquina anfitrión el archivo cvMindstorms-0.1-i386.deb y a Ubuntu 10.04 como sistema operativo. Luego hacer doble click en el archivo deb.

Figura B.1: Ventana de instalación de cvMindstorms.

En la Figura B.1 se muestra la ventana de instalación que surge después de dar doble click sobre el archivo deb. Después se debe hacer click sobre el botón instalar paquete y esperar el mensaje de "Instalación finalizada".

Los requerimientos mínimos que se necesitan para realizar la instalación de cvMindstorms son los siguientes:

- Procesador: 300 MHz x86
- Disco duro:10 MB.
- Memoria Ram: 512 MB.
- USB 2.0.
- Bluetooth (Opcional).

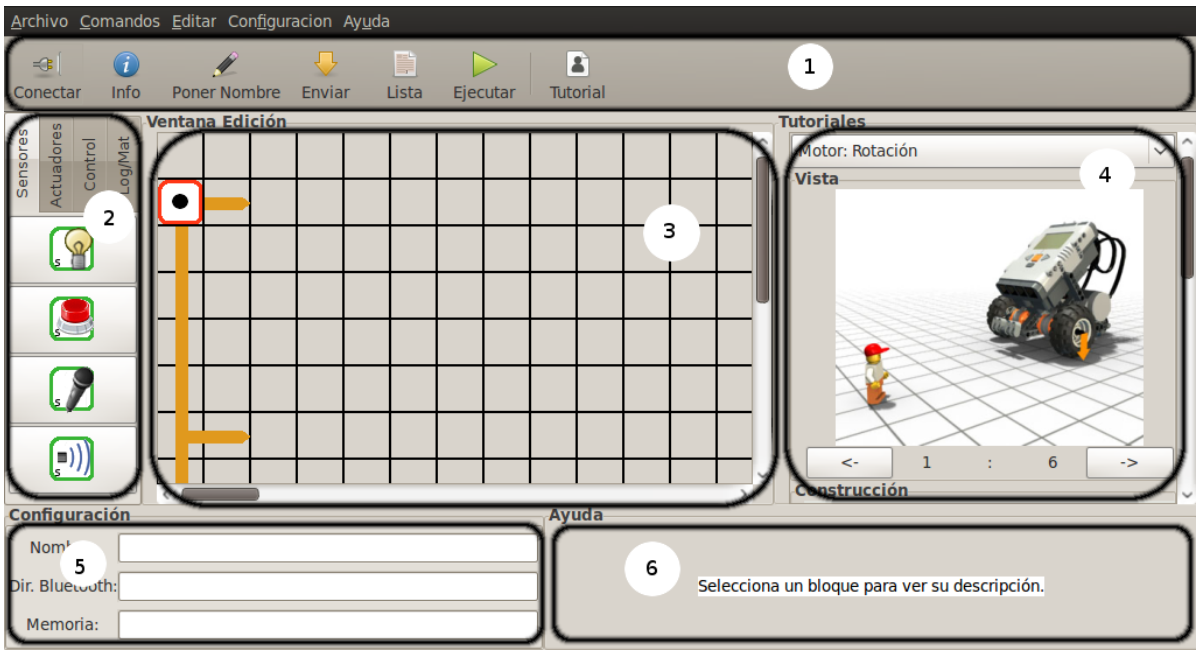

Figura B.2: Interfaz del cvMindstorms.

B.2. Aspecto de la Interfaz del Compilador Visual

El compilador visual está formado por una ventana principal que es el ambiente en el que los usuarios pueden realizar sus programas visuales. La ventana principal, que se muestra en la Figura B.2, a su vez es dividida en 6 secciones:

Barra de Herramientas(1): contiene las herramientas básicas para la comunicación con el CPU del Robot por ejemplo, cargar programa al Robot, recuperar información, etc.

Paleta de Bloques(2): permite a los usuarios seleccionar los bloques para programar el comportamiento que tendrá el Robot.

Ventana de Edición(3): es el espacio de trabajo para crear el programa por medio de los bloques.

Tutorial(4): presenta una serie de imágenes que ayudan al usuario a crear sus primeros proyectos de programación visual.

Ventana de Configuración(5): permitirá ver y cambiar la configuración de algún bloque que haya sido seleccionado por el usuario en la ventana de edición.

Ventana de Ayuda(6): muestra una descripción del bloque que sea seleccionado por el usuario en la ventana de edición.

Los bloques son elementos del programa que definirán el comportamiento del robot una vez que éste sea cargado en su CPU. Y existen 4 tipos diferentes de bloques: sensores, actuadores, de control y lógicos - matemáticos.

Con todas estas ventanas y herramientas el usuario debe ser capaz de escribir un programa que defina el comportamiento del robot y una vez que el usuario haya terminado de realizar el programa en el compilador, éste tendrá que enviar o cargar el programa en la CPU del Robot. Cuando el usuario presiona el botón de 'cargar programa' la aplicación tiene que traducir el programa del lenguaje visual (o de alto nivel) a un lenguaje de bajo nivel (lenguaje máquina) que será entendido por el robot y que será enviado por medio de un cable USB o conexión Bluetooth. Después de esto el usuario puede probar el programa que ha creado en el conjunto de construcción del robot y posteriormente realizar cambios si así lo desea.

B.3. Crear un programa

En el menú de la barra de tareas ir a Aplicaciones > Programación > cvMindstorms. Después de dar click en cvMindstorms se abrirá la aplicación y mostrará una ventana similar a la mostrada en la Figura B.2.

En cuanto la aplicación es abierta la ventana de edición está lista para comenzar a escribir un programa por parte del usuario. O si ya se tiene un programa y se desea crear otro entonces en el menú de cvMindstorms ir a Archivo > Cerrar y el usuario puede empezar a escribir un nuevo programa.

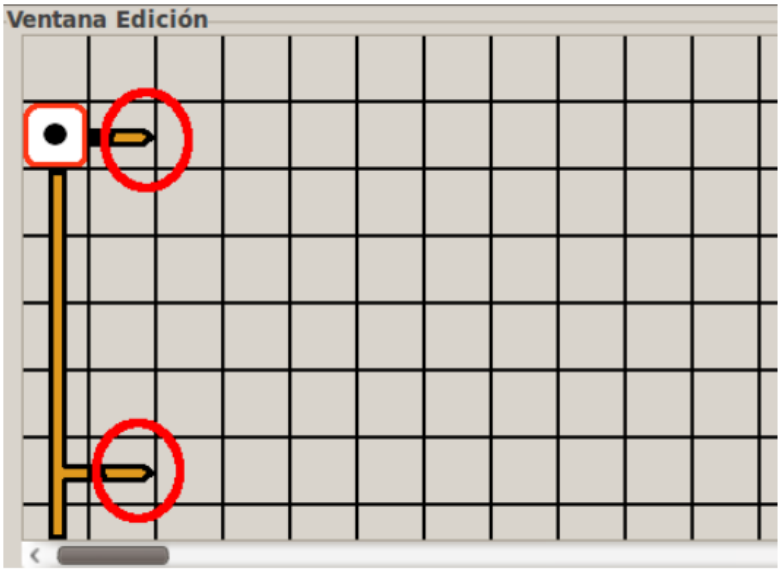

Figura B.3: Flechas de secuencia.

Dentro de la ventana de edición se muestran unas flechas que marcan la secuencia que tendrá el programa que realice el usuario, es decir, los bloques se deben poner a la derecha de alguna flecha de secuencia libre o sobre una flecha de secuencia (entre dos bloques programables). En la Figura B.3 se muestran las flechas de secuencia (dentro de círculos) que aparecen al iniciar el programa.

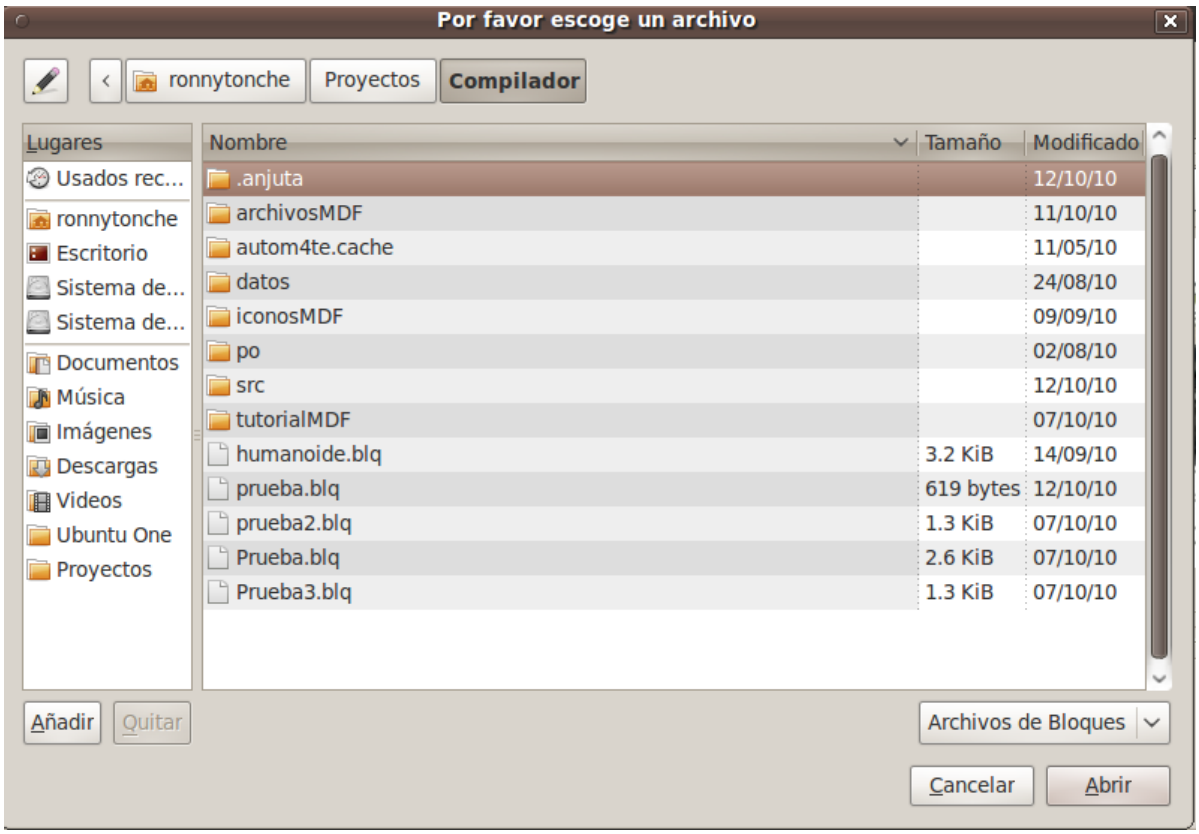

Figura B.4: Dialogo Abrir programa.

B.4. Abrir un programa

En el menu de cvMindstorms ir a Archivo > Abrir lo cual mostrará una ventana de dialogo abrir (Figura B.4) en la cual el usuario tiene que seleccionar un archivo .blq y presionar el botón abrir. Los archivos guardados previamente con cvMindstorms son guardados con la extensión .blq, previamente mencionada.

B.5. Enviar programa al robot

Una vez que el usuario haya terminado de realizar un programa, éste tiene la opción de enviar el programa a la CPU del robot y el usuario puede elegir de entre dos opciones para el modo de la conexión que son: usb y bluetooth.

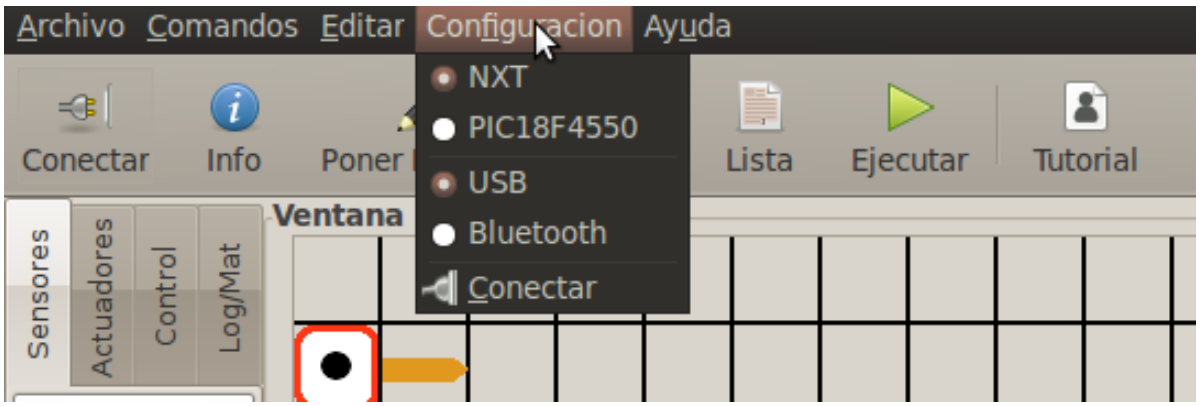

Figura B.5: Menú Configuración.

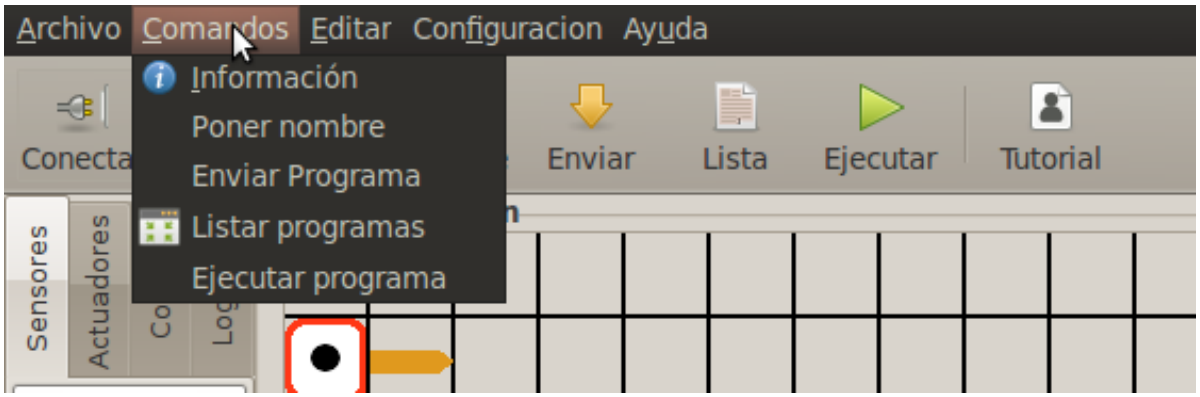

Figura B.6: Menú Comandos.

En la Figura B.5 se muestra el menú Configuración donde el usuario puede elegir la conexión con el robot por usb o bluetooth. Después en el mismo menú el usuario debe dar click en la opción conectar para realizar el enlace con el robot y finalmente en el menú Comandos (Figura B.6) elegir la opción Enviar programa para realizar la transferencia del programa compilado al CPU del Mindstorms.

B.6. Descripción de los bloques

Dentro de la aplicación existe 4 diferentes tipos de bloques programables: sensores, actuadores, de control y lógicos – matemáticos.

A continuación se muestra cada uno de los bloques que pueden ser utilizados para programar el comportamiento del robot divididos en los diferentes tipos mencionados anteriormente.

B.6.1. Sensores

Los sensores son los encargados de realizar una alimentación de ciertas condiciones del ambiente hacia el robot. Todos los bloques tienen ciertas opciones de configuración que pueden ser modificados para alterar (configurar) el comportamiento de dicho sensor.

Sensor de Luz.

Este sensor es el encargado de detectar la intensidad de la luz ambiental. Además tiene la opción de activar una pequeña fuente de luz que el sensor tiene integrada. Esta pequeña fuente de luz puede ser útil en ambientes cerrados donde la luz es muy baja.

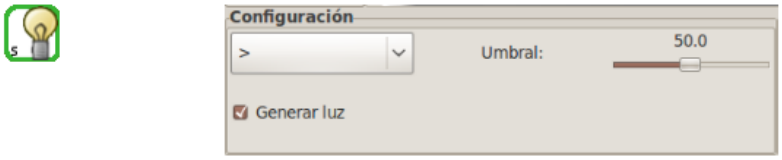

Figura B.7: Icono y configuración del bloque Sensor de Luz.

Las opciones de configuración del sensor de luz son las siguientes:

- Umbral. Es un valor específico dentro del rango de números que se pueden sensar.
- Operador de comparación. Es el operador para determinar si el resultado del sensado esta por abajo o arriba del valor umbral.
- Generar luz. Permite activar o desactivar la fuente de luz que trae el sensor.

En la Figura B.7 se muestra el icono y la ventana de configuración del bloque Sensor de Luz.

Sensor de Sonido.

Este bloque representa al sensor que detecta los niveles de sonido del ambiente. Las opciones de configuración del sensor de sonido son las siguientes:

- Umbral. Es un valor específico dentro del rango de números que se pueden sensar.
- Operador de comparación. Es el operador para determinar si el resultado del sensado está por abajo o arriba del valor umbral.

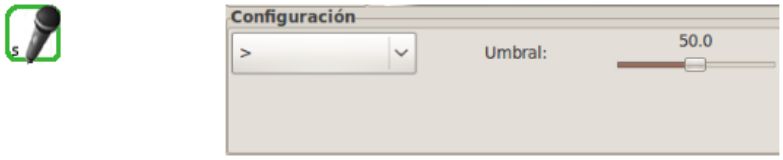

Figura B.8: Icono y configuración del bloque Sensor de Sonido.

La Figura B.8 muestra al icono y la ventana de configuración del bloque Sensor de Sonido.

Sensor de Tacto.

Este sensor esta formado por un dispositivo que censa el presionado o liberación de un botón. Las opciones de configuración del sensor de sonido son las siguientes:

- Presionado. Es uno de los dos valores posibles del censado de este bloque y es verdadero cuando el botón está siendo presionado por algún objeto externo.
- Liberado. Es lo contrario a la opción de presionado. Es verdad si el botón no está siendo presionado.

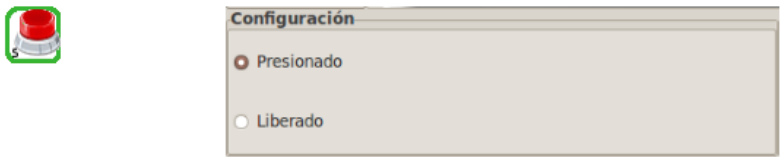

Figura B.9: Icono y configuración del bloque Sensor de Tacto.

El icono y la ventana de configuración del bloque Sensor de Tacto se muestra en la Figura B.9.

Sensor Ultrasónico.

Este sensor es el encargado de detectar a qué distancia de dicho sensor se encuentra un obstáculo como una pared o cualquier otro objeto que se encuentre en frente del sensor. Las opciones de configuración del sensor de ultrasónico son las siguientes:

- Umbral. Es un valor específico dentro del rango de números que se pueden sensar.

- Operador de comparación. Es el operador para determinar si el resultado del sensado esta por abajo o arriba del valor umbral.

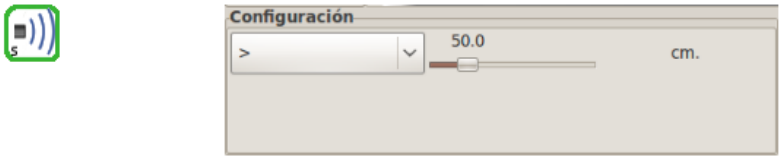

Figura B.10: Icono y configuración del bloque Sensor Ultrasónico.

El icono y la ventana de configuración del bloque Sensor Ultrasónico se muestra en la Figura B.10.

B.6.2. Actuadores

Los bloque actuadores, en el compilador visual, representan a dispositivos que tienen la función de emitir un sonido, fuerza o algún tipo de señal hacia el exterior del robot.

Motor

Este actuador es el encargado de realizar rotaciones para realizar trabajos que necesiten este tipo de fuerzas. Las opciones de configuración del actuador motor son las siguientes:

- Motores. Esta opción permite seleccionar cuáles de los tres motores (A, B y C) estarán en funcionamiento, pudiendo tener diferentes combinaciones de motores como son: A, B, C, AB, AC, BC y ABC.
- Duración. Permite configurar la duración de las rotaciones eligiendo entre grados, número de rotaciones, tiempo en milisegundos o de duración indeterminado.
- Dirección. Especifica la dirección que pueden tomar los motores en el sentido hacia adelante, reversa y parar.
- Velocidad. Es el valor que determina la intensidad y fuerza de las rotaciones.
- Desviación. Valor que configura la desviación que tendrán dos motores cuando funcionen de manera coordinada. Eligiendo entre izquierda o derecha.

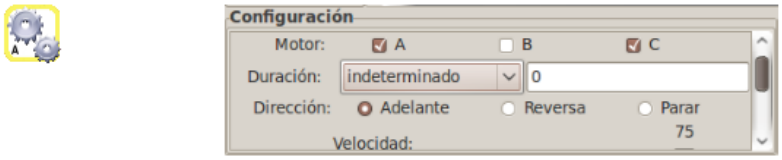

Figura B.11: Icono y configuración del bloque Actuador Motor.

La Figura B.11 muestra al icono y la ventana de configuración del bloque Actuador de Motor.

Bocina

Bloque que representa a una bocina que permite reproducir archivos de sonidos. Las opciones de configuración del actuador bocina son las siguientes:

- Archivo. Permite elegir uno de entre varios archivos de sonido.
- Volumen. Valor que permite configurar el nivel del volumen.
- Duración. Permite elegir entre Esperar reproducción completa, Repetir reproducción o Parar reproducción.

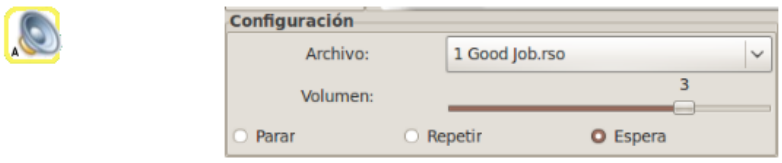

Figura B.12: Icono y configuración del bloque Actuador Bocina.

La Figura B.12 muestra al icono y la ventana de configuración del bloque Actuador de Bocina.

Pantalla

Dispositivo que permite mostrar imágenes o texto. Las opciones de configuración del actuador Pantalla son las siguientes:

- Despliega. Permite elegir entre desplegar un texto o imágenes.

- Texto. Si se elige desplegar texto el usuario puede introducir el texto que desea que aparezca en el display del NXT.
- X y Y. Representan a las coordenadas del punto (x,y) donde iniciará el despliegue del texto o imagen.
- Limpiar. Permite especificar si se desea limpiar el display totalmente antes de desplegar el texto o imagen actual.

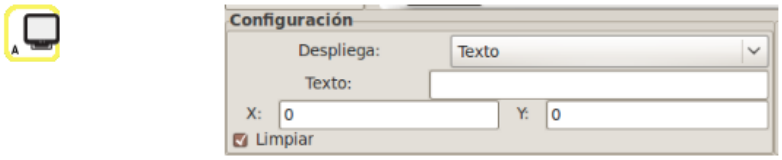

Figura B.13: Icono y configuración del bloque Actuador Pantalla.

La Figura B.13 muestra al icono y la ventana de configuración del bloque Actuador de Pantalla.

B.6.3. De control

Los bloques de control representan a sentencias que, como su nombre lo dice, permiten controlar la secuencia o flujo que tendrá el programa una vez que esté dentro de la CPU del robot.

Ciclo

El bloque de ciclo permite repetir una secuencia de bloques para que realicen ciertas operaciones un número determinado o indeterminado de veces. La configuración principal del bloque ciclo se puede realizar de cuatro diferentes formas: infinito, sensor, tiempo y conteo. Las características de las diferentes opciones de configuración del bloque de control Ciclo son las siguientes:

- Infinito. Repetición de la secuencia un número indeterminado de veces.
- Sensor. Repetición de la secuencia un número indeterminado de veces hasta que se cumpla la condición de censado de un sensor.
- Tiempo. Repetición de la secuencia por un tiempo determinado en milisegundos.
- Conteo. Repetición de la secuencia un número determinado de veces.

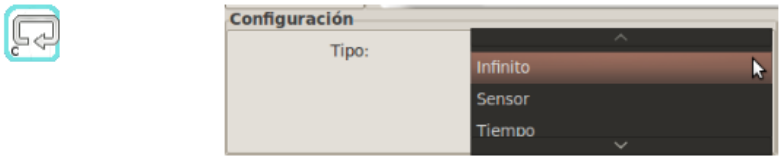

Figura B.14: Icono y configuración del bloque de Control Ciclo.

El icono y la ventana de configuración del bloque de Control Ciclo se muestra en la Figura B.14.

Según caso

Este bloque se utiliza para poder elegir de entre dos caminos distintos que pudiera tomar la secuencia o flujo del programa. Las dos diferentes secuencias están determinadas por el valor por debajo o arriba de un umbral de algún sensor o por un valor lógico introducido al bloque de manera dinámica en tiempo de ejecución. Las opciones de configuración del bloque Según caso son:

- Sensor. Permite seleccionar uno de los diferentes sensores y especificar qué secuencia tomará el flujo del programa según el valor de censado en conjunto con el valor de umbral.
- Valor lógico. Esta opción permite especificar la secuencia del programa en tiempo de la ejecución tomando el dato desde otro bloque.

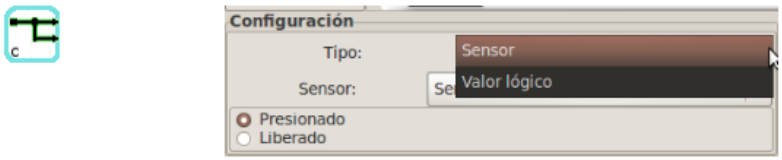

Figura B.15: Icono y configuración del bloque de Control Según caso.

El icono y la ventana de configuración del bloque de Control Según Caso se muestra en la Figura B.15.

Espera

Bloque que permite esperar un tiempo determinado o permite esperar la condición del censado de un bloque sensor. Las características de configuración son las siguientes:

- Tiempo. Para el flujo del programa un determinado tiempo en milisegundos.
- Sensor. Espera la condición de censado de algún sensor para continuar con la secuencia del programa.

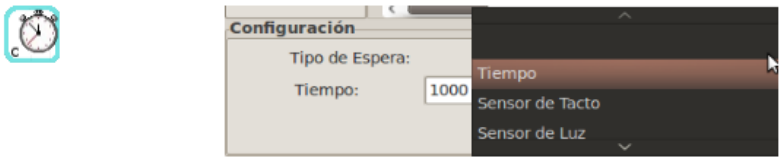

Figura B.16: Icono y configuración del bloque de Control Espera.

El icono y la ventana de configuración del bloque de Control Espera se muestra en la Figura B.16.

B.6.4. Lógicos – matemáticos

Los bloques lógicos – matemáticos nos proporcionan bloques que sirven para realizar operaciones aritméticas, lógicas y funciones más especificas como obtener valores aleatorios o ver si un valor está dentro de cierto rango.

Operadores Aritméticos.

Permite realizar operaciones de suma, resta, división y multiplicación de dos operadores de tipo entero o flotante. Sus opciones de configuración son:

- Suma. Permite realizar la operación aritmética suma.
- Resta. Permite realizar la operación aritmética resta.
- División. Permite realizar la operación aritmética división.
- Multiplicación. Permite realizar la operación aritmética multiplicación.

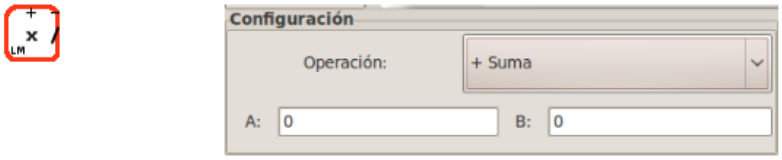

Figura B.17: Icono y configuración del bloque Aritmético.

En la Figura B.17 se muestra el icono y la ventana de configuración del bloque Operadores Aritméticos.

Operadores Lógicos

Se utiliza para realizar operaciones de AND, OR y NOT sobre dos operadores booleanos. Sus opciones de configuración son:

- And. Permite realizar la operación lógica AND (y).
- Or. Permite realizar la operación lógica OR (o).
- Not. Permite realizar la operación lógica NOT (negación).

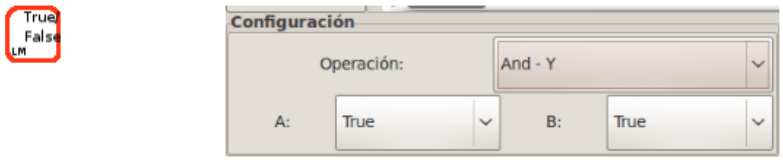

Figura B.18: Icono y configuración del bloque Lógico.

En la Figura B.18 se muestra el icono y la ventana de configuración del bloque Operadores Lógicos.