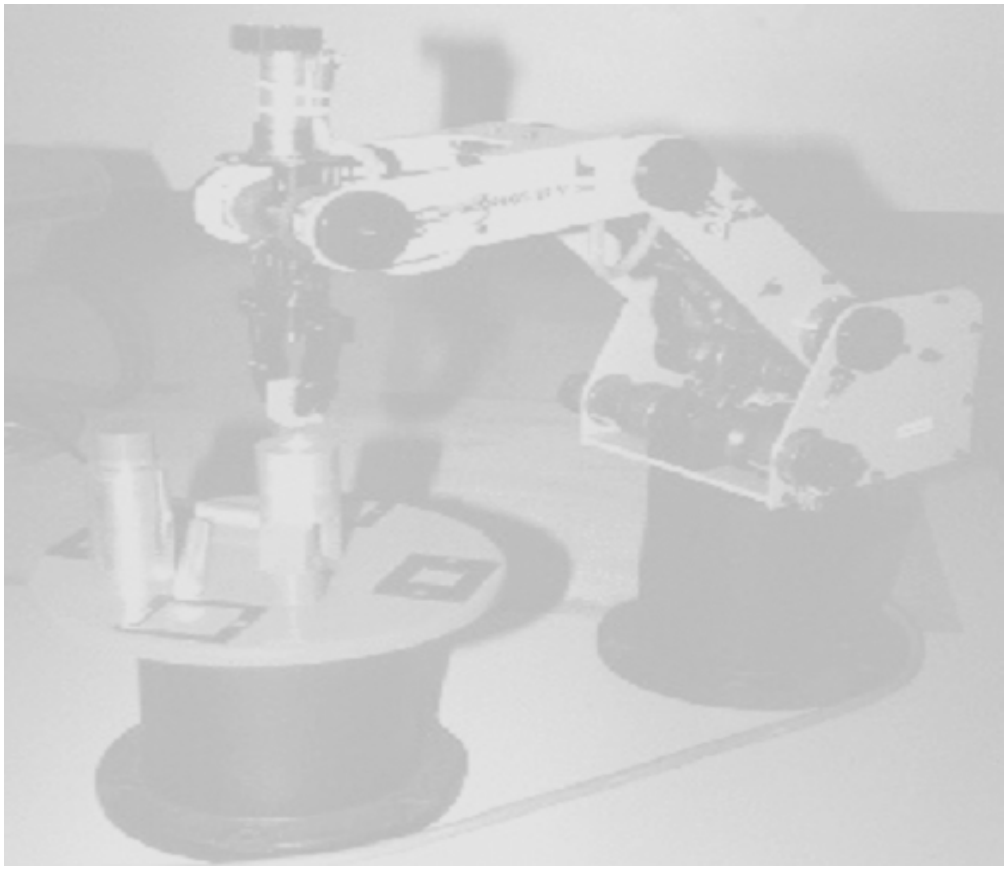


Simulador e intérprete del lenguaje ACL para el robot SCORBOT ER-V

Alejandro Ballesta López-Murcia
DNI nº:77586317

ÍNDICE

1 OBJETIVO DEL PROYECTO

El objetivo de este proyecto es crear una aplicación que refleje lo más fielmente posible el comportamiento del lenguaje ACL implementado en el controlador del manipulador SCORBOT ER-V situado en el Laboratorio de Robótica y Automatización.

Este objetivo ha sido conseguido gracias a una aplicación que consta de tres partes bien diferenciadas: una interfaz gráfica, un intérprete del lenguaje y un modelo gráfico del robot.

2 INTRODUCCIÓN

El robot SCORBOT ER-V es usado para labores didácticas en el Departamento de Sistemas y Automática de la misma Escuela. Debido al aumento del número de alumnos en las titulaciones impartidas, el hecho de realizar prácticas reales con material real se hace cada vez más difícil. Todo esto lleva a que el tiempo que se le puede dedicar a la realización de prácticas se vea reducido, y no sólo el tiempo sino también el aprovechamiento.

En este contexto se pensó en crear un simulador que permita a los alumnos tomar contacto con lo que después se encontrarán en el laboratorio, para que de este modo el nivel de aprovechamiento de las prácticas allí impartidas sea mayor.

Además existen una serie de beneficios añadidos que pueden ser clasificados en dos categorías:

Beneficios para el alumno:

- no tiene que desplazarse hasta el laboratorio para realizar las prácticas de introducción al manejo del robot porque se supone que ya lo conoce en cierta medida.
- puede realizar cualquier tipo de modificación y prueba sobre el robot sin que ponga en peligro el equipo.

Beneficios para el equipo:

- no sufre daños innecesarios debido al entrenamiento previo de los usuarios.
- ayuda a los usuarios a adquirir una disciplina de manejo del lenguaje ACL. Esta disciplina posiblemente contribuirá a la no-proliferación de variables y posiciones sin sentido en cada una de las máquinas controladoras del robot.

Una vez definido el objetivo de este proyecto se realizó una elección que determinaría algunas de las características más importantes de este simulador. La principal elección que debía hacerse era el lenguaje en el que sería programado. El simulador ha sido programado en lenguaje Java debido a varias razones entre las que podemos mencionar:

- facilidad para añadirle nuevas funciones.

- versatilidad a la hora de ser adaptado a otro tipo de robots y lenguajes.
- el compilador e intérprete de Java son gratuitos, por lo tanto están al alcance de cualquier alumno que desee modificar el código.

Además el lenguaje Java será previsiblemente uno de los más usados en el futuro, esto hace que la portabilidad del código sea muy alta y posibles modificaciones futuras sean aceptadas de una forma muy natural.

El simulador funcionará en cualquier máquina que posea una Java Virtual Machine (JVM). En la actualidad la mayoría de los ordenadores personales poseen una implementación de la JVM para su sistema operativo y arquitectura. La portabilidad del código también se extiende a diferentes tipos de máquinas, aunque el PC se ha hecho con el mercado no se puede presuponer nada para el futuro.

Por el contrario, el lenguaje Java tiene la desventaja de que se ejecuta más lentamente que la mayoría de los lenguajes utilizados actualmente (Visual C++, Delphi, etc.). Esto es debido a que el lenguaje Java es interpretado en tiempo de ejecución, por tanto la velocidad de cualquier programa escrito en Java depende en gran medida de la velocidad de implementación de la JVM. En otros lenguajes de programación, tales como Visual C++ o Delphi, el código es compilado en tiempo de diseño a código nativo de la máquina y por tanto la velocidad de ejecución es mucho mayor.

Esto, que en principio podría tomarse como una gran desventaja, se suple con el hecho de que el simulador va a funcionar en cualquier ordenador personal, cosa que no puede ser asegurada por otros lenguajes de programación (hasta la fecha solamente Delphi para Windows en su versión 6 muestra compatibilidad con su versión para Linux, Kylix).

El simulador consta de un entorno de creación de programas en lenguaje ACL y un modelo del robot SCORBOT ER-V. El entorno de creación de programas no es más que un editor de textos con las características añadidas de un entorno de programación.

El modelo del robot responde a los comandos de movimiento que se le den desde los programas. Han sido creadas dos representaciones del robot SCORBOT, una más vistosa pero que requiere más recursos en tres dimensiones; y otra en dos dimensiones que responde más rápido al seguimiento de

trayectorias. Este modelo fue creado a partir de mediciones reales en el robot.

En el documento han sido incluidos algunos apartados que están dedicados al uso general del simulador. Estos apartados pueden ser extraídos para posteriores consultas de los usuarios de este simulador. Este apartado es el número 4, en el que se explica detalladamente el funcionamiento de cada una de las ventanas.

El resto de apartados están dedicados a detallar como se consiguió simular el funcionamiento de la máquina real. Estos apartados son los siguientes:

- Los apartados número 6 y 7. En estos se explica como fueron obtenidos los modelos del robot SCORBOT y los métodos que se usan para la generación de trayectorias y la interacción con los objetos.
- El apartado número 8. En este apartado se detallan las bases para la creación de un lenguaje de programación, el lenguaje ACL, y su funcionamiento conjunto con la máquina en la que es ejecutado. Es decir, son tratadas las estructuras de datos que soportarán los tipos de datos de la máquina real.
- El apartado número 9. Es el apartado más importante de este documentos, en él es tratado el modo en el que ha sido simulado el programa ACL, entendiendo programa como conjunto de instrucciones, posibles saltos y errores.
- El apartado número 10. En este apartado es tratada la forma en la que han sido simulados todos los comandos del lenguaje ACL.

Notación

Dentro de este documento serán usados distintas características del texto que servirán para denotar diferentes informaciones acerca de lo comentado.

tipo de letra	**tipo de texto**
normal	Ninguno de los tipos de texto siguientes.
cursiva normal	referencias al lenguaje Java dentro de párrafos normales
pequeña	código en Java perteneciente al simulador
MAYÚSCULAS normales	referencias al juego de comandos del lenguaje ACL

3 INSTALACIÓN DEL SIMULADOR

Para explicar el proceso de instalación escogeremos la plataforma más común en cuanto a sistemas operativos, esto es Windows.

Es necesario recalcar que el simulador funciona en cualquier máquina que posea una máquina virtual de Java (JVM) instalada. Además esta máquina debe soportar funciones OpenGL para la representación gráfica.

Existía la posibilidad de haber realizado la representación utilizando funciones DirectX propias de las plataformas Windows. Por un lado, esta opción hubiera acelerado la representación gráfica en pantalla y además se hubiera beneficiado de las constantes actualizaciones de dicho software y de las tarjetas gráficas disponibles en el mercado con soporte para dichas funciones. Por otro lado, esta opción hubiera limitado el soporte del simulador a máquinas con el sistema operativo Windows instalado. Dado que OpenGL posee un buen soporte en todas las plataformas y que se puede dejar parte del trabajo a la cpu, no se vio necesario usar las funciones DirectX.

3.1 Instalación del simulador

Para la instalación del simulador disponemos de un archivo ejecutable llamado jscorbot_install.exe. Una vez ejecutado este archivo ya se habrán instalado todos los ficheros necesarios para ejecutar el simulador. Además se habrán instalado las librerías necesarias para la ejecución del simulador con la representación gráfica en 3D.

La parte que se encarga de la representación del modelo usa funciones OpenGL. Estas librerías son instaladas al ejecutar el programa de instalación.

Los requisitos mínimos para la ejecución del simulador son los siguientes: 2 megabytes de espacio libre en la unidad donde se vaya a instalar, 64 megabytes de memoria RAM física libre y una tarjeta gráfica con soporte de funciones OpenGL y capaz de ofrecer resoluciones iguales o superiores a 800x600 píxeles.

3.2 Ejecución del simulador

Existen dos formas de ejecutar el simulador. Para ello disponemos de dos archivos por lotes JSCORBOT2D.BAT y JSCORBOT3D.BAT que arrancarán la versión 2D y la versión 3D

respectivamente. Estos archivos están situados en el directorio JSCORBOT.

Aunque el simulador empiece su ejecución desde una ventana de DOS, es una aplicación Windows y es por ello que estos archivos deben ser ejecutados desde el Explorador de Windows o desde una ventana de DOS, nunca desde el DOS nativo.

El comando que sirve para ejecutar la versión 2D es el siguiente (que coincide con el contenido del archivo JSCORBOT2D.BAT):

```
java -cp ".; lib\java3d-utils-src.jar; lib\j3dutils.jar; lib\j3dcore.jar;
lib\j3daudio.jar; lib\vecmath.jar" SCORBOTSimulator.SimulatorMain -robot2d
```

El de la versión 3D es similar:

```
java -cp ".; lib\java3d-utils-src.jar; lib\j3dutils.jar; lib\j3dcore.jar;
lib\j3daudio.jar; lib\vecmath.jar" SCORBOTSimulator.SimulatorMain
```

Las librerías 3D deben ser cargadas en ambos casos debido a que la distinción de modos de funcionamiento se hace en tiempo de ejecución y por tanto de antemano debe indicarse su carga, aunque en el caso de la versión 2D no se finalizará.

La memoria usada en la versión 2D es de 13.5 megabytes al empezar la ejecución. Y para la versión 3D es de 17 megabytes. Estas cifras irán creciendo y es por ello que es necesario ejecutar el simulador en una máquina con al menos 64 megabytes de memoria RAM.

Una vez que hayamos conseguido arrancar el simulador debemos fijarnos en la respuesta del modelo a los controles de la pistola de programación. Si la respuesta es lenta (debemos tener en cuenta que existe un comando llamado SPEED que puede aumentar la velocidad de ejecución de las trayectorias, pero aquí nos referimos al tiempo desde que se termina la pulsación hasta que comienza la ejecución del movimiento) debemos cerrar alguno de los programas que tengamos abiertos además del simulador. Esto es debido a que Java es muy dependiente de los recursos libres de los que disponga la máquina.

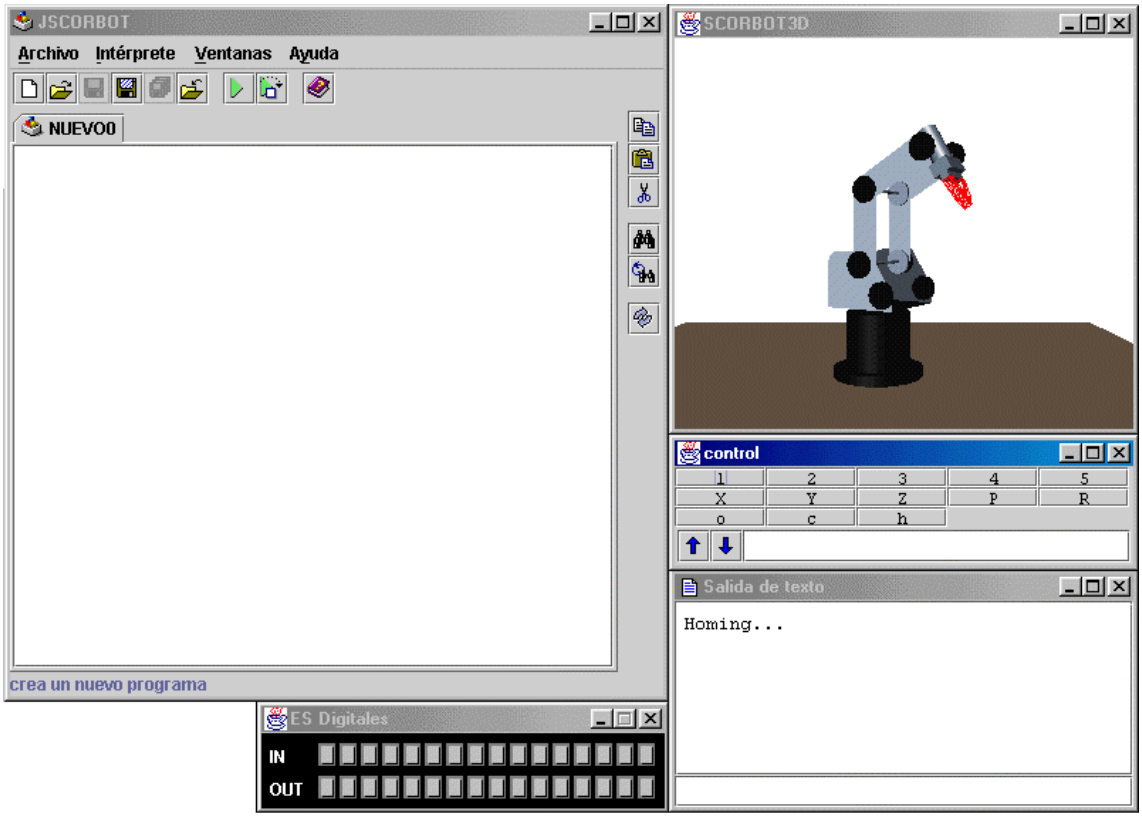
4 VISIÓN GENERAL DEL SIMULADOR

En este capítulo será presentado el simulador en su totalidad, tratando con más detalle las opciones más interesantes de cada una de las ventanas. Este capítulo puede ser usado como guía para el posible estudiante usuario del simulador.

4.1 Vista general

Al terminar la carga del simulador aparecerán una serie de ventanas que nos muestran los comandos y funciones más utilizadas.

La disposición que aquí se nos muestra es la que viene por defecto en el simulador, es decir, es la primera que aparecerá. Esta disposición puede ser cambiada ya que la última disposición de ventanas que usemos será guardada para la próxima vez.

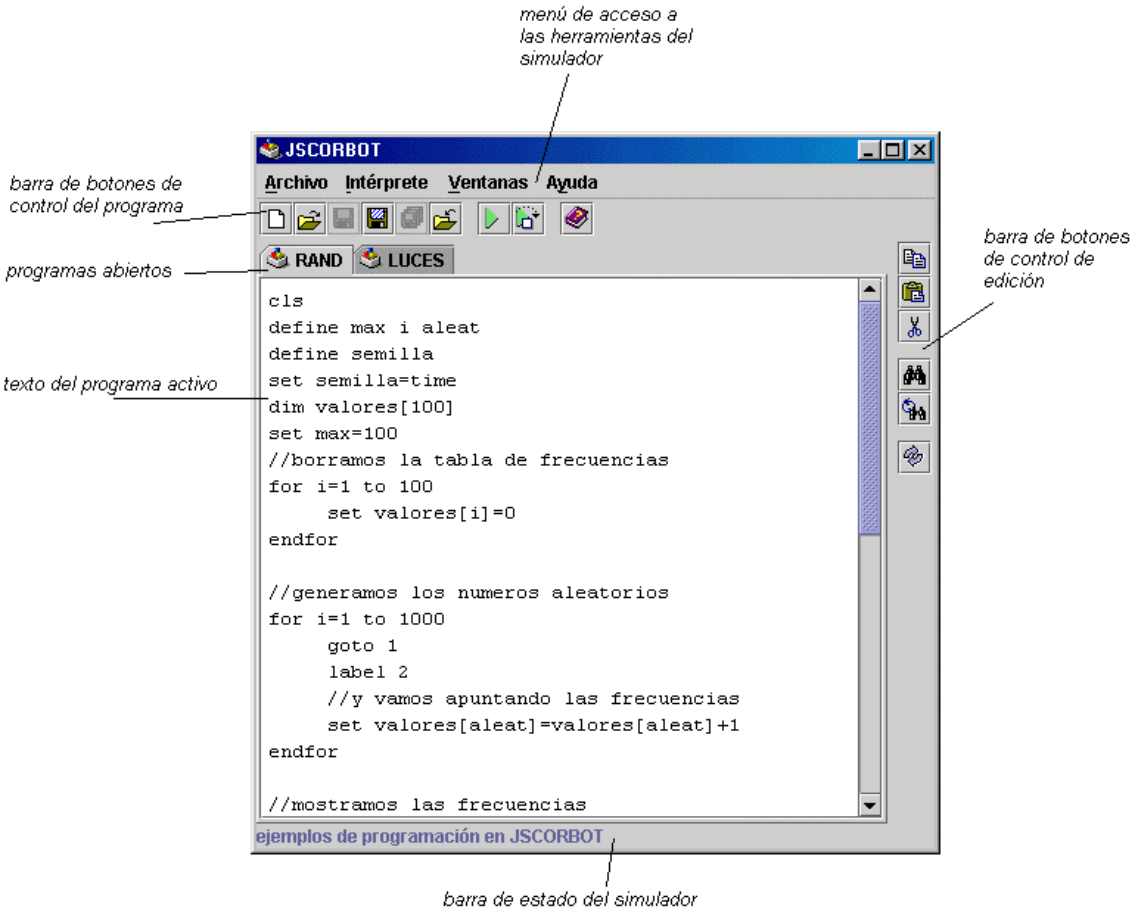


4.2 Ventana principal

En esta ventana podemos encontrar todas las herramientas necesarias para escribir los programas que correrán en el simulador.

Se puede apreciar que posee todas las funcionalidades propias de un editor de texto. Esto es así debido a que básicamente el simulador se compone de tres partes: un editor de texto, el intérprete de comandos y el modelo del robot.

Con este editor de texto se han querido suplir muchas de las deficiencias del equipo real a la hora de escribir los programas, tales como: editor de una sola línea, imposibilidad de cortar y pegar, no existen búsquedas...



Las características básicas de este editor son las siguientes:

- Edición de múltiples ficheros
- Copiar, cortar y pegar textos
- Grabación múltiple
- Búsqueda de cadenas y repetición de búsqueda

Esta ventana es la principal del simulador. Si se cierra esta ventana el simulador finalizará su ejecución. De igual manera para minimizar todas las ventanas visibles del simulador se debe minimizar sólo esta ventana.

Se debe hacer mención aquí a la principal diferencia existente entre el editor del equipo real y el editor del simulador: en el equipo real existen los números de línea. En la realidad se va chequeando la sintaxis de los comandos a la vez que estamos escribiendo las líneas que componen el programa. En el simulador la comprobación se deja hasta el final.

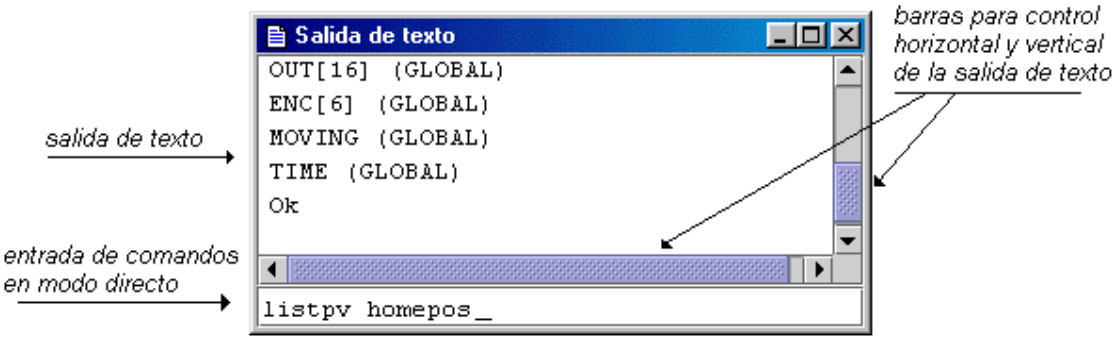
Esta diferencia hace que por ejemplo haya sentencias que no generen líneas de código en un programa como las etiquetas, mientras que en el simulador sí.

Las diferencias entre la máquina real y el simulador serán tratadas en todos los apartados en los que corresponda, aunque han sido recopiladas todas en el apartado número 5.

4.3 Ventana de diálogo

Con este nombre se hace referencia a la ventana que posee los controles de salida de texto de usuario y a entrada de comandos en modo directo.

En el equipo real no se hace la distinción entre edición y ejecución, todo se realiza sobre el mismo cuadro de texto. En el simulador han sido separadas estas partes. La edición se realiza en la ventana anteriormente explicada, mientras que la salida de texto de los programas y comandos se realiza en esta ventana.

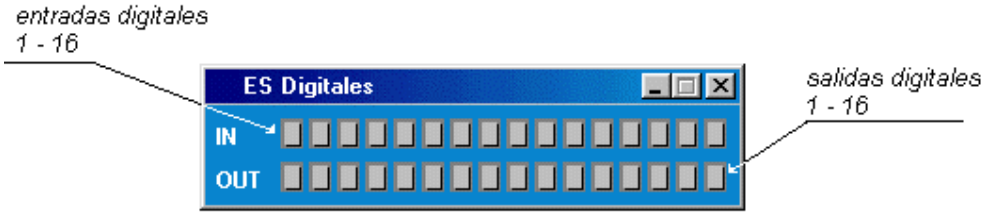


En el control superior se nos muestra la salida de texto que produce el programa en ejecución o el comando introducido en modo directo. Esta salida también incluye los errores que se puedan producir.

En el inferior tenemos un cuadro de texto en el que podemos introducir comandos en modo directo para que sean ejecutados en ese mismo momento. Este cuadro además posee un buffer de comandos. Los comandos que se van ejecutando son guardados en el buffer para poder recuperarlos rápidamente con las flechas de dirección ARRIBA y ABAJO del teclado. Este buffer guarda un total de 10 comandos.

4.4 Ventana de entradas y salidas digitales

Esta ventana nos permite observar la evolución de las entradas y salidas digitales de las que dispone la máquina controladora del robot SCORBOT. Estas E/S están ligadas a las variables IN y OUT (las cuales sólo pueden tomar valores lógicos), por lo que cualquier cambio en estas variables se verá reflejado inmediatamente en esta ventana y viceversa.



En la realidad, y como su propio nombre indica, las entradas digitales no podrían cambiar de valor lógico desde la máquina. Éstas vendrían impuestas por un agente externo (que podría ser otra máquina igual) que impondría el nivel lógico de cualquiera de las entradas de nuestra máquina. Debido a que nos encontramos ante un programa de simulación, la mejor

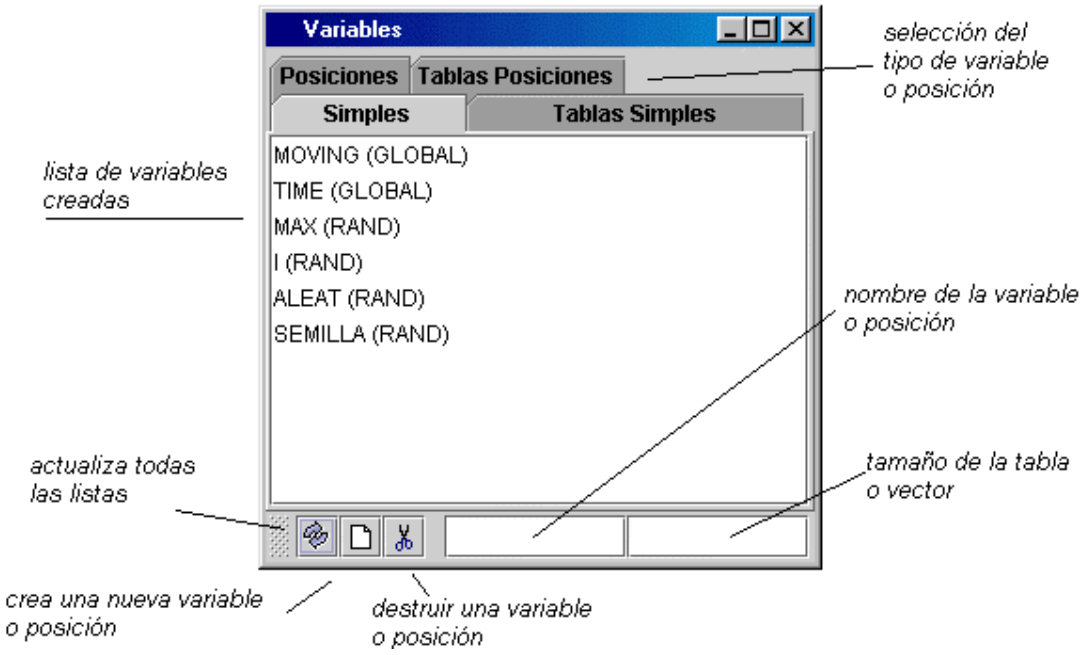
manera de simular ese agente externo es teniendo la posibilidad de cambiar ese valor directamente.

Podemos cambiar el valor de cualquiera de las E/S pulsando sobre su correspondiente cuadro. Si esa variable toma un valor lógico verdadero se iluminará.

4.5 Ventana de variables

Esta ventana nos muestra todas las variables y posiciones que han sido creadas en el simulador. Son mostradas tanto las creadas por el usuario como las del sistema.

Permite la creación y borrado de variables. También nos da la posibilidad de visualizar los valores.

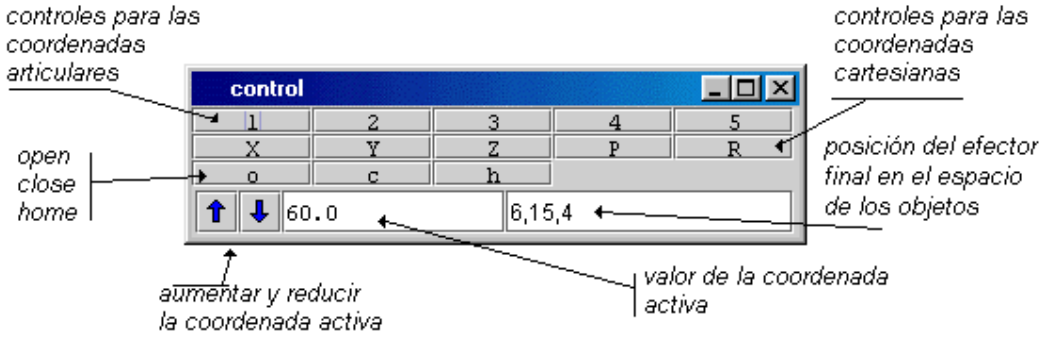


La ventana ha sido dividida en cuatro pestañas diferentes que nos permiten acceder a las listas de variables simples, tablas, posiciones y vectores de posiciones respectivamente.

Para las dos primeras listas obtenemos también el programa responsable de la creación de esa variable. Las posiciones y vectores de posiciones son accedidos de forma global.

4.6 Pistola de programación

Bajo este nombre tenemos la ventana que nos permite un manejo directo del robot. Podemos acceder a cada uno de los ejes del robot para modificar su valor directamente.



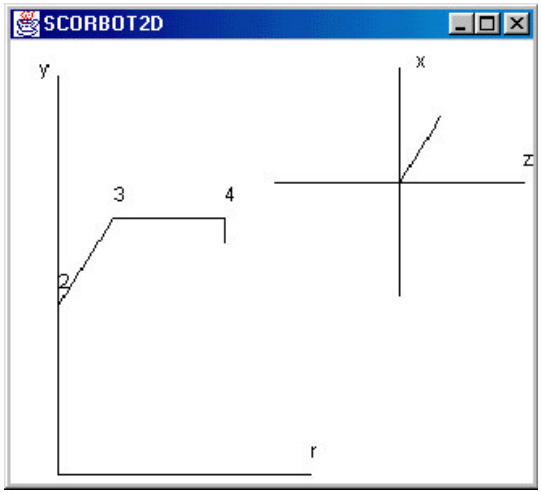
Posee cuatro filas de controles. La primera fila nos permite acceder a los ejes. La segunda fila a la posición del efector final en coordenadas cartesianas. Y por último, la tercera fila posee tres botones para facilitar el acceso a los comandos OPEN, CLOSE y HOME, respectivamente.

Una vez pulsado un botón de cualquiera de las dos primeras filas, se actualizará el valor de dicha variable en coordenadas articulares si accedemos a un eje. Por otra parte será actualizado en coordenadas cartesianas si accedemos a la posición del efector final. Las coordenadas articulares se muestran en grados, mientras que las cartesianas en décimas de milímetro (100 décimas de milímetro es un centímetro).

El valor que se muestra en el cuadro de texto de la izquierda podemos modificarlo directamente introduciendo un nuevo valor, y validándolo pulsando la tecla INTRO. También podemos cambiarlo con los dos botones que se muestran a la izquierda del cuadro de texto. Estos botones permiten aumentar o disminuir la cantidad que se muestra, el incremento que se aplica se puede configurar en la ventana de configuración. Por otra parte, el valor mostrado en el cuadro de texto de la derecha nos muestra la posición del efector final en el espacio de los objetos (para más detalles consultar el apartado número 7.3). Este valor también puede ser cambiado sin más que introducir una nueva terna, separando los coeficientes por comas.

4.7 Ventana del modelo

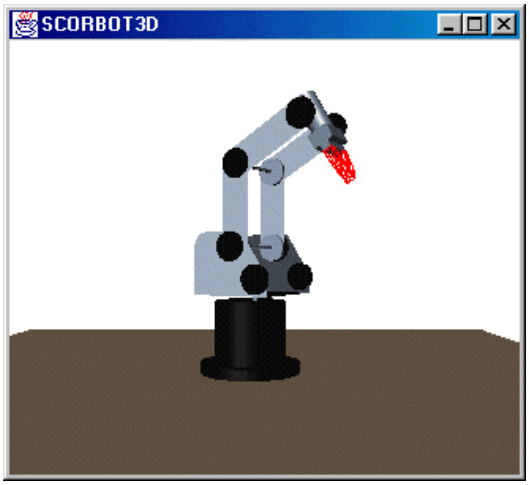
En esta ventana se muestra la representación del modelo del robot. Este modelo responde a los comandos de movimiento. Existen dos representaciones, una en dos dimensiones y otra en tres dimensiones.



La representación en dos dimensiones muestra dos gráficos, el primero con un corte en el eje radial del robot, y el segundo con un corte en plano xz.

En esta representación no se dispone de visualización de objetos con los que se pueda interaccionar, así como tampoco se puede observar la apertura y cierre del efector final.

Pulsando sobre la ventana podemos cambiar la posición del segundo gráfico.



En la representación en tres dimensiones vemos el robot completo sobre una plataforma. En ésta podemos interactuar con objetos.

Si pulsamos sobre la ventana podemos cambiar el punto de vista de la escena.

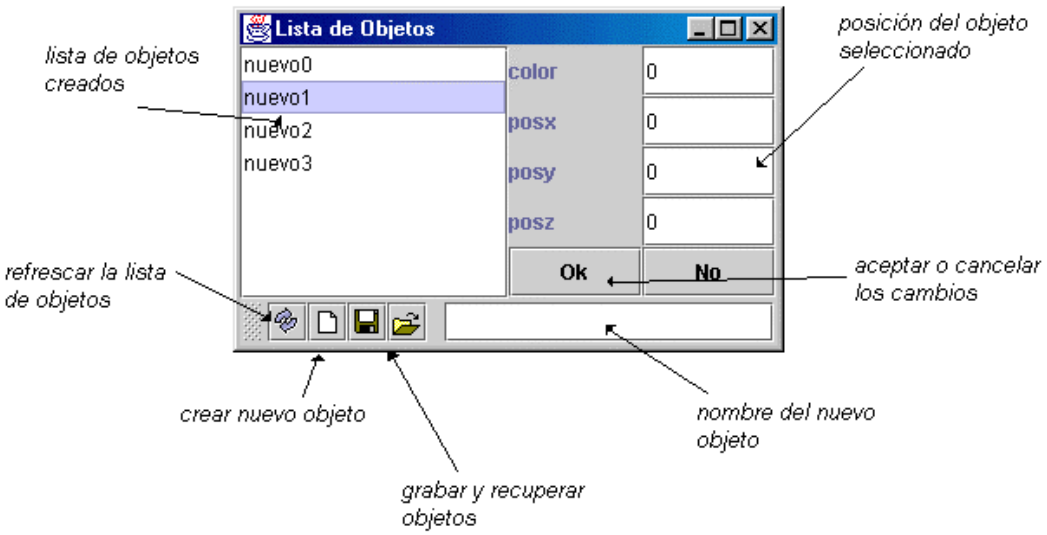
4.8 Ventana para la gestión de objetos

Con esta ventana podemos tener acceso a los objetos que creemos. Estos objetos nos servirán para que el robot SCORBOT pueda interactuar con ellos.

Esta ventana sólo estará activa cuando usemos el modelo 3D del robot, ya que usando la representación gráfica en 2D los objetos no están disponibles.

Los objetos son cubos con cada cara de un color distinto. Aunque dentro de las propiedades de cada objeto también se incluye la de color, esta propiedad ha sido desactivada por lo que todos los objetos son considerados iguales. La posición de los objetos viene dada por una terna de tres coordenadas. Esta terna indica la posición del objeto en un espacio en el que sólo algunas posiciones son posibles. Esto es, el espacio cartesiano ha sido discretizado mediante una cuadrícula de longitud, anchura y altura iguales a la mitad del lado del objeto.

De esta manera, la posición (0,0,0) en el espacio de los objetos equivale a la (0,0,0) en el coordenado; y la (1,0,0) equivale a la (LADOCUBO/2,0,0). Para más detalles acudir al apartado número 7.3.



Al igual que la ventana de variables, aquí se dispone también de la función de guardar y recuperar una lista de objetos. Estas funciones nos pueden servir para crear un escenario compuesto por algunas posiciones definidas, algunos objetos en la escena, y el programa que gestionará los movimientos del robot.

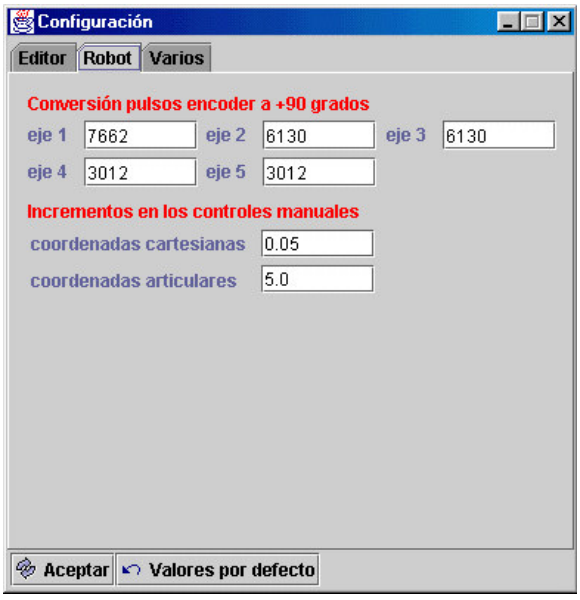
4.9 Ventana de configuración

Esta ventana nos permite cambiar alguna de las características del simulador, de las que podemos destacar color, tipo y tamaño de fuentes, activación de los mensajes de warning, etc.

Está dividida en tres categorías: editor, robot y varios.

En la pestaña de la configuración del editor podemos cambiar los colores de las ventanas, cursores, tipos y tamaños de fuentes.

En la pestaña correspondiente a parámetros del robot podemos cambiar la correspondencia entre las unidades pulsos de los encoders de las articulaciones del robot y grados. Además podemos cambiar el incremento que se aplica al usar los controles de la pistola de programación (apartado nº 4.6).



Y por último, en la tercera pestaña podemos cambiar algunas aspectos del intérprete del lenguaje ACL.

4.10 Notas generales

En este punto trataremos una serie de puntos que no han podido ser clasificados dentro de los otros apartados, pero que sin embargo pueden aclarar ciertos aspectos del simulador.

- ✓ El simulador dispone de una configuración predeterminada para colores, fuentes, posiciones de las ventanas, etc. Esta configuración puede ser cambiada en la ventana de configuración explicada anteriormente. Por otra parte, las posiciones de las distintas ventanas serán guardarán cada vez que se cierre el simulador, de tal manera que al volver a arrancarlo las ventanas recuperarán su posición anterior.

- ✓ El aspecto general del simulador es el de una aplicación ejecutándose bajo entorno Windows. Sin embargo debemos tener en cuenta que nos encontramos ante una aplicación realizada en un lenguaje multiplataforma. Esto quiere decir que éste será el aspecto de la aplicación sea cual sea el sistema operativo y la máquina en la que sea ejecutado el programa. Esto es debido al paquete Swing de Java.

5 DIFERENCIAS ENTRE EL SIMULADOR Y LA MÁQUINA REAL

En este punto han sido recopiladas todas las diferencias existentes entre el simulador y la máquina real. Han sido citadas en orden a que sirvan para posteriores correcciones y también para saber cuáles son las limitaciones del software diseñado en este proyecto.

Diferencias en el modelo

Aquí serán presentadas las características del robot real y del entorno que no han sido transportadas al simulador. También existen algunas características que, habiendo sido transportadas, no dan un resultado similar al real.

- ✓ No existe comprobación de colisiones con los objetos, esto hace que el modelo pueda atravesar los objetos.

- ✓ Los objetos sólo pueden ocupar ciertas posiciones en el espacio cartesiano. Esto quiere decir que un conjunto de posiciones cartesianas dado por un cubo posee la misma imagen en el espacio de los objetos.

- ✓ Los objetos no se ven afectados por la fuerza gravedad. La gravedad no ha sido modelada en el escenario. Sólo ha sido simulada la caída libre de objetos con gravedad nula, es decir, los objetos caerán a velocidad constante.

- ✓ La rutina de comprobación de posible situación de impacto del robot consigo mismo es simple y por ello a veces inexacta. Es por eso que en trayectorias complicadas podría saltar un mensaje a la pantalla de posible impacto. La situación de impacto en la máquina real es más fácil ya que sólo hay que comprobar cuando se dispara el consumo en alguno de los servomotores.

Diferencias en el intérprete

En este punto trataremos las diferentes peculiaridades que definen al simulador y que a la vez diferencian al intérprete de lenguaje ACL de la máquina real del intérprete del simulador.

- ✓ Todos los comandos generan líneas de código. En la máquina real existen ciertos comandos, tales como LABEL y todos los que sirven para crear variables y posiciones, que no generan líneas de código. Esto es debido a que el simulador es un intérprete y por tanto sólo ejecuta una línea cuando es leída y no antes. En la máquina real existe un espacio en memoria asignado a cada programa, y cuando en el modo editor ejecutamos un comando LABEL por ejemplo, la etiqueta se crea directamente para su posterior uso. Algo similar ocurre con las variables y las posiciones. Cuando se crea una variable local puede ser usada en todo el programa, en el simulador sólo puede ser usada en las líneas posteriores del código. Por tanto es aconsejable crear todas las variables al principio de nuestro programa.

- ✓ Todas las variables, posiciones y programas quedan residentes en memoria. En el simulador esto no es así y es por ello que debe hacerse copia siempre del trabajo realizado. Esto quiere decir que si entendemos un programa como un texto con las líneas del programa, sus variables y posiciones, todas deben ser grabadas por separado. Debe evitarse el uso de variables globales creadas desde el modo directo, ya que al recuperar el programa estas no serán creadas, no así las creadas en modo editor.

Diferencias en la máquina

Aquí presentaremos las características del hardware real que no han sido simuladas debido a la complejidad que supone el hecho de copiar una máquina con todos sus detalles a un software con las limitaciones de velocidad y capacidad de proceso.

- ✓ No se puede asegurar una resolución en el temporizador. La máquina real tiene una resolución fija igual a 10 milisegundos. Esta resolución, incluso siendo baja (en el sentido de fácil de alcanzar), es difícil proporcionarla en una máquina virtual de Java. Esto es debido a que la JVM, la cual es multitarea, está corriendo sobre un sistema operativo multitarea. Por tanto, la solución para aumentar la resolución sería manejar el temporizador directamente en el sistema operativo y tratarla como una rutina de interrupción. Esta solución, aunque la más

efectiva, es la que haría que el simulador tuviese que usar código nativo y por tanto se perdería la portabilidad del código Java del simulador.

- ✓ No existe una prioridad máxima para ser asignada a los programas. Según fue explicado en el apartado número 9.3, todos los hilos de ejecución en Java nacen con una prioridad inicial normal. Esto quiere decir que todos los hilos son creados igual y por tanto tendrán el mismo tiempo de ejecución asignado. Al usar la representación gráfica y la gestión de ventanas estamos ocupando tiempo de proceso, y si creamos un hilo con mayor prioridad de la normal este hilo tendrá más prioridad que la representación y la actualización de ventanas. Esto provocaría que no pudiese ser controlada la ejecución de este nuevo hilo y que no actualizase el modelo de acuerdo con los comandos ejecutados. Por consiguiente, se recomienda utilizar las prioridades más bajas, es decir de 1 a 5.

- ✓ Las entradas digitales han sido simuladas desde el mismo programa debido a que el simulador no puede aceptar entradas desde otro dispositivo o programa. Esto hace que tanto las entradas como las salidas digitales funcionen como simples tablas en las que se puede leer y escribir normalmente. La única diferencia que existe con otras tablas creadas, es que éstas además muestran su valor en la pantalla en la ventana de entradas y salidas digitales (ver apartado número 4.4).

- ✓ Aunque la máquina real trabaja con una aritmética entera de 16 bits, el tipo de dato en Java que más se asemeja a éste es el tipo primitivo *int* o su versión más completa *Integer*. Este tipo no tiene la limitación en resolución que tiene la máquina real y es por eso que no se han podido modelar los errores del tipo *overflow*. De todas maneras la estructura para manejarlos existe, aunque no ha sido puesta en funcionamiento puesto que no pudo ser comprobada la forma en que la máquina gestionaba este tipo de errores.

6 MODELO CINEMÁTICO

La obtención del modelo cinemático será dividida en dos puntos: la obtención del modelo directo y del inverso.

El modelo cinemático directo es usado para el cálculo de la posición del efector final a partir de los ángulos que forman las articulaciones.

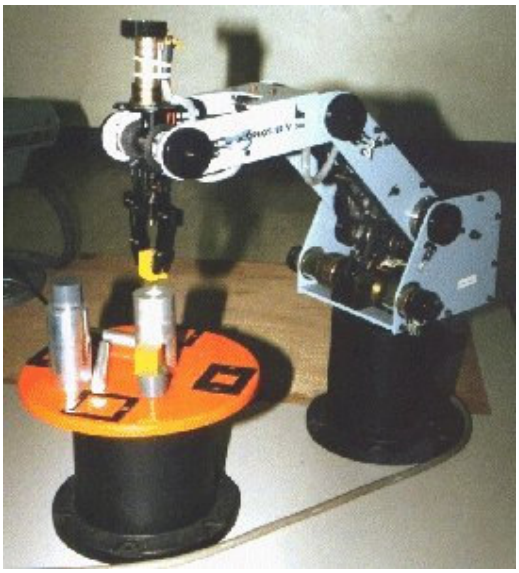
Por otro lado, el modelo cinemático inverso es usado para el cálculo de los ángulos que deben formar las articulaciones a partir de la posición del efector final.

Cuando nos referimos al término posición del efector final hacemos referencia a la quíntupla *x, y, z, pitch y roll*. Estas cinco coordenadas son necesarias para definir una posición en el espacio cartesiano del manipulador.

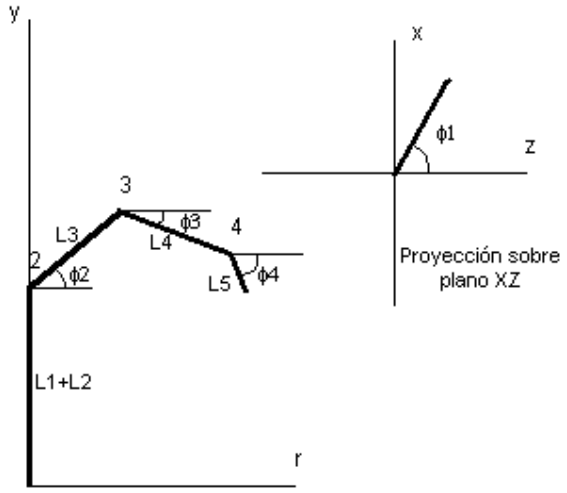
El término *pitch* hace referencia al ángulo que forma el efector final con la vertical. Por tanto, un pitch de 180º quiere decir que el efector final apunta perpendicularmente hacia el suelo, entendiendo por suelo el plano en el que está montado el robot.

El término *roll* hace referencia al ángulo de rotación del efector final. Este ángulo será el que forme la pinza con el eje fijo donde está montada.

Por tanto, tenemos que encontrar unas funciones que nos liguen el espacio cartesiano con el espacio articular.



6.1 Obtención del modelo cinemático directo



Para obtener el modelo debemos tener en cuenta los siguientes puntos:

- la articulación 1ª gira en el plano xz, y su referencia es el eje z.
- las articulaciones 2ª hasta la 5ª giran en el plano yr, siendo r una dirección perpendicular al eje y que viene dada por el ángulo de rotación de la articulación 1. Estos ángulos de rotación tienen como referencia el eje radial.
- la articulación 6 (la que nos dará el roll) no interviene en la determinación de la terna (x, y, z).

Con estas consideraciones podemos empezar a determinar cada una de las cinco coordenadas cartesianas.

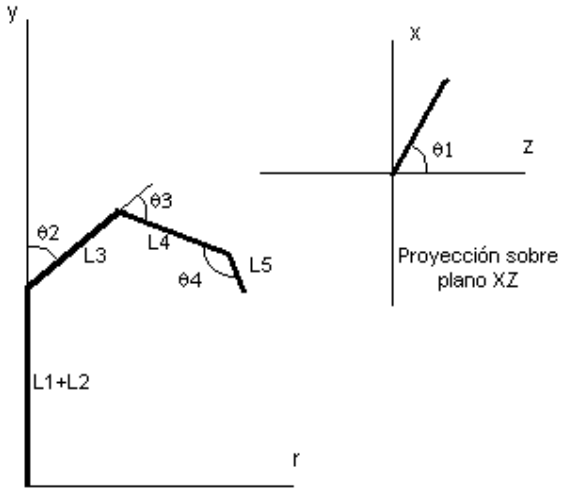
$$y = l_1 + l_2 + l_3 * sen(\phi_2) + l_4 * sen(\phi_3) + l_5 * sen(\phi_4)$$

La proyección del manipulador sobre el eje r es la que nos servirá para encontrar las coordenadas x z.

$$x = \left[l_3 * sen(\phi_2) + l_4 * sen(\phi_3) + l_5 * sen(\phi_4)\right] * sen(\phi_1)$$
$$z = \left[l_3 * sen(\phi_2) + l_4 * sen(\phi_3) + l_5 * sen(\phi_4)\right] * \cos(\phi_1)$$

Para obtener los coordenadas cartesianas *pitch* y *roll* haremos uso de los parámetros de entrada a la hora de calcular el modelo. Estos parámetros de entrada serán los ángulos que realmente serán utilizados tanto para la representación en

pantalla como para calcular las posiciones. La representación es la siguiente:



Por tanto, el cambio de variables que debemos hacer es el siguiente:

$$\phi_1 = \theta_1$$
$$\phi_2 = (\pi/2) - \theta_2$$
$$\phi_3 = (\pi/2) - \theta_3 - \theta_2$$
$$\phi_4 = (\pi/2) - \theta_3 - \theta_2 - (\theta_3 + \pi)$$
$$\phi_5 = \theta_5$$

Entonces, las coordenadas *pitch* y *roll* serán las siguientes:

$$pitch = \theta_2 + \theta_3 + \theta_4 + \pi$$
$$roll = \theta_5$$

6.2 Implementación del modelo cinemático directo

Este código toma como entrada la quíntupla formada por los ángulos de cada una de las articulaciones y como salida la posición del efector final en coordenadas cartesianas.

```
public class ModeloCinematicoDirecto {

  //longitudes de los eslabones
  private static final double[] l={0.23,0.119,2*0.100,2*0.100,0.05};
  private static double[] pxyz=new double[3];
  private static double[] pr=new double[Const.NUMJOINTS];
```

```
  private static double[] pitchroll=new double[2];

  public static void updatePosition(double[] phi) {
    pr[0]=phi[0];
    pr[1]=(Math.PI/2)-phi[1];
    pr[2]=(Math.PI/2)-phi[1]-phi[2];
    pr[3]=(Math.PI/2)-phi[1]-phi[2]-(phi[3]+Math.PI);
    double proyecXZ=l[4]*Math.cos(pr[3]) + l[3]*Math.cos(pr[2]) +
l[2]*Math.cos(pr[1]);
    pxyz[0]=(proyecXZ)*Math.sin(pr[0]);
    pxyz[1]=l[0] + l[1] + l[2]*Math.sin(pr[1]) + l[3]*Math.sin(pr[2]) +
l[4]*Math.sin(pr[3]);
    pxyz[2]=(proyecXZ)*Math.cos(pr[0]);
    //el pitch y el roll van en grados
    pitchroll[0]=Math.toDegrees(phi[1]+phi[2]+phi[3])+180;
    pitchroll[1]=Math.toDegrees(phi[4]);
  }

  public static double[] getXYZ() {return (pxyz);}
  public static double[] getPR() {return (pitchroll);}

  public static double getX() {return pxyz[0];}
  public static double getY() {return pxyz[1];}
  public static double getZ() {return pxyz[2];}
  public static double getP() {return pitchroll[0];}
  public static double getR() {return pitchroll[1];}


}
```

Las posibles mejoras en cuanto a velocidad de ejecución que se podrían realizar sobre este código serían las siguientes:

- Utilizar aritmética de enteros en vez de reales. Esto lo conseguiríamos multiplicando las entradas por un factor (1000 por ejemplo), también podríamos usar grados en vez de radianes, discretizando sus valores en intervalos de 1º.
- También podríamos tabular las funciones trigonométricas seno y coseno. Para ello deberíamos calcular de antemano los valores de la funciones seno y coseno sobre un conjunto finito y discreto de argumentos. Este conjunto finito y discreto podría ser los ángulos desde 0 a 360º con un intervalo de 1º o 0.5º. A partir de esta tabulación tendríamos dos tablas (la de la función seno y la del coseno) de 360 o 720 entradas, dependiendo del intervalo del argumento de entrada.

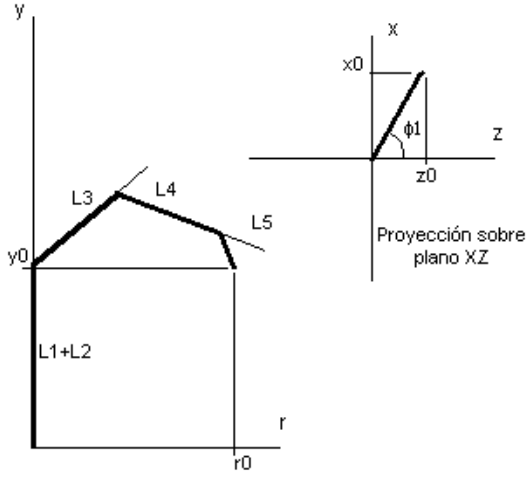
Todas las mejoras propuestas tienen como inconveniente fundamental la pérdida de precisión y el incremento en el tamaño del código. La aritmética de enteros es más rápida pero también más compleja ya que tenemos que escalar los números

reales con los problemas de overflow y pérdida de precisión que podemos generar.

Por otro lado, las funciones tabuladas incrementan la velocidad de cálculo de una manera extraordinaria. Sin embargo, el hecho de tener que generar dos tablas de 360 entradas ocupa también gran cantidad de memoria, aunque este hecho cada vez tiene menos importancia.

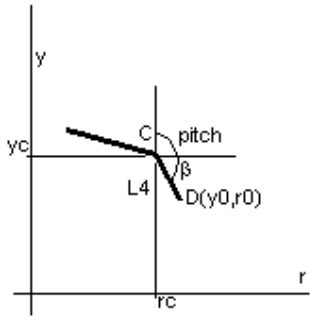
6.3 Obtención del modelo cinemático inverso

El método para obtener las relaciones que liguen el espacio de coordenadas cartesianas con el articular es más complejo. Sin embargo, este método se ve simplificado por el hecho de que el robot SCORBOT ER-V tiene dos de los eslabones de la cadena de igual longitud, es decir L3=L4.



$$\theta_1 = \phi_1 = a\tan\left[x_0 / z_0\right]$$

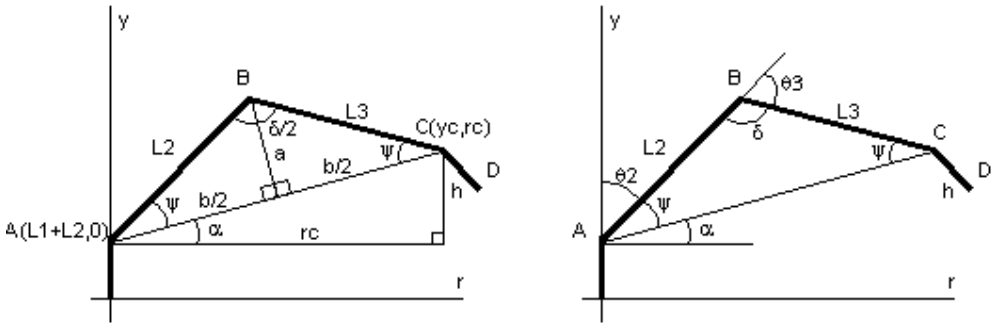
Como parámetro de entrada tenemos ahora la quíntupla (x, y, z, pitch, roll). Y tenemos que actuar en orden inverso, es decir, empezar desde la última articulación hasta la primera.



Si nos centramos en la última articulación:

$$C(y,r) = C(yc,rc)$$
$$yc = y_0 + l_4 * sen(\beta)$$
$$rc = r_0 - l_4 * \cos(\beta)$$
$$r_0 = \sqrt{x_0{}^2 + z_0{}^2}$$
$$\beta = pitch - (\pi/2)$$

Observando ahora las articulaciones 2 y 3 :



Podemos encontrar las siguientes relaciones geométricas

$$2\psi + \delta = \pi$$
$$\theta_2 + \psi + \alpha = \pi/2$$
$$\theta_3 + \delta = \pi$$
$$(rc)^2 + (h)^2 = (b)^2 \rightarrow b = \sqrt{rc^2 + h^2}$$
$$\alpha = a\tan\left[\frac{h}{rc}\right]$$
$$a^2 + (b/2)^2 = L_2{}^2 \rightarrow a = \sqrt{L_2{}^2 - (b/2)^2}$$
$$\psi = a\tan\left[2a/b\right]$$

6.4 Implementación del modelo cinemático inverso

Este código recibe como entrada la quíntupla formada por la posición del efector final (x,y,z) y los ángulos que forma este mismo con la articulación anterior (roll) y con la vertical (pitch).

Debemos recordar que al utilizar funciones inversas como la arcotangente, no podemos asegurar la bondad de la solución obtenida. Por esta razón este código también debe efectuar algunas comparaciones que no eran necesarias a la hora de resolver el modelo cinemático directo.

Para verificar que la solución obtenida es válida sólo debemos comprobar el valor devuelto por la función *Conseguido()*, la cual devolverá *true* si no hubo error.

Debemos recordar que este código no está optimizado en cuanto a velocidad de ejecución. Esto es debido a que el código debe tener las características siguientes: fácilmente modificable y auto explicativo. Este procedimiento está incluido dentro de un programa de simulación, por tanto la velocidad no es el principal objetivo sino la fidelidad con la realidad.

Para una implementación más eficiente que pudiera servir para resolver el modelo en tiempo real para aplicaciones reales deberíamos considerar las siguientes modificaciones:

- Utilizar aritmética de enteros. Las operaciones con enteros son más rápidas que las operaciones con números reales.
- Tabular la función arcotangente. Este método, también conocido como "lookuptable" nos permite ahorrar el tiempo de calcular la función arcotangente cada vez que la usemos. Se basa en guardar en una tabla unidimensional los valores que toma la función arcotangente para un número finito de argumentos. Estos argumentos también deben ser discretizados.

```
public class ModeloCinematicoInverso {

  //longitudes de los eslabones
  private static final double[] l={0.23,0.119,2*0.100,2*0.100,0.05};
  private static double[] q=new double[Const.NUMJOINTS];
  private static double[] qreal=new double[Const.NUMJOINTS];
  private static boolean error=false;

  public static void updateJoints(double[] xyz,double[] pr) {
    error=false;
    //pitch y roll vienen en grados
    q[0]=Math.atan(xyz[0]/xyz[2]);
    q[3]=Math.toRadians(pr[0])-Math.PI/2;
    //punto donde se encuentra la articulación j4 (Mc,yc)
    double yc=xyz[1]+l[4]*Math.sin(q[3]);
    double Mc=Math.sqrt(xyz[0]*xyz[0]+xyz[2]*xyz[2])-l[4]*Math.cos(q[3]);
    //altura de j4 con respecto a j2
    //mc también es la distancia desde j2 hasta j4
```

```
    double h=yc-(l[0]+l[1]);
    //triangulo formado por j2 j3 j4
    double b=Math.sqrt(Mc*Mc+h*h)/2.0; //cateto1
    double a=l[3]*l[3]-b*b; //cateto2
    if (a<0) { //condición para ver si llega al punto
      a=0; error=true;
    }
    else
      a=Math.sqrt(a);
    double theta=Math.atan(a/b); //angulo j3 j2 j4
    double alfa=Math.atan(h/Mc); //angulo j4 j2 horizontal
    q[1]=alfa+theta;
    double d=Math.PI-2*theta; //angulo j2 j3 j4
    q[2]=-(Math.PI-d-q[1]); //los angulos interiores suman 180°

    qreal[0]=q[0];
    qreal[1]=(Math.PI/2)-q[1];
    qreal[2]=(Math.PI/2)-qreal[1]-q[2];
    //qreal[3]=(Math.PI/2)-qreal[1]-qreal[2]-(q[3]+Math.PI);
    qreal[3]=Math.PI-(Math.toRadians(pr[0])-(qreal[1]+qreal[2]));
    qreal[3]=-qreal[3];
    qreal[4]=Math.toRadians(pr[1]);
  }

  public static double[] getJoints() { return(qreal);}
  public static double getJoint(int nj) { return (qreal[nj]);}

  public static boolean Conseguido() {
    return (!error);
  }
}
```

6.5 Conversión de unidades

Existe ahora el problema de la conversión de unidades debido a que la máquina en la que usamos el lenguaje ACL utiliza una aritmética de enteros. Nuestros modelos se basan en cálculos con funciones trigonométricas y por tanto obtenemos resultados dados en números reales. Por tanto debemos tener funciones que nos liguen ambos espacios.

Para las articulaciones el lenguaje ACL utiliza la unidad llamada pulso. Esta unidad indica el número de pulsos del encoder de una de las articulaciones. ACL permite ajustar los pulsos del encoder que son necesarios para realizar un giro de 90°. Este ha sido el mismo criterio utilizado en el simulador. A través de la ventana de configuración podemos acceder a dichas equivalencias que pueden ser modificadas en cualquier momento.

Por otro lado, las coordenadas cartesianas son tratadas en décimas de milímetro. Por tanto, la conversión es bastante simple, sólo hay que escalar hasta hacer que las medidas

tomadas del robot en la realidad sean iguales a las que usamos en el simulador. Las coordenadas pitch y roll son tratadas de una manera especial ya que trabajan en décimas de grado.

```
public class ModeloConv {

  public static int Rad2Pulses(int nj,double rad) {
    rad=rad*Configuracion.getPulsesForJoint(nj)/(Math.PI/2.0);
    return ((int) Math.floor(rad));
  }

  public static int[] Rad2Pulses(double rad[]) {
    int[] p=new int[Const.NUMJOINTS];

    for (int i=0;i<Const.NUMJOINTS;i++)
      p[i]=(int) Math.floor(rad[i] * Configuracion.getPulsesForJoint(i) /
(Math.PI/2.0));

    return (p);
  }

  public static double Pulses2Rad(int nj,int pulses) {
    double aux=pulses*(Math.PI/2.0);
    aux=aux/(double) Configuracion.getPulsesForJoint(nj);

    return (aux);
  }

  public static double[] Pulses2Rad(int[] pulses) {
    double[] aux=new double[Const.NUMJOINTS];
    for (int i=0;i<Const.NUMJOINTS;i++)
      aux[i]=pulses[i]*(Math.PI/2.0)/(double)
Configuracion.getPulsesForJoint(i);

    return (aux);
  }

  public static int Cart2xyz(double x) {
    return ((int) Math.floor(x*Const.ESCALAXYZ));
  }

  public static int[] Cart2xyz(double[] x) {
    int[] q=new int[3];
    q[0]=(int) Math.floor(x[0]*Const.ESCALAXYZ);
    q[1]=(int) Math.floor(x[1]*Const.ESCALAXYZ);
    q[2]=(int) Math.floor(x[2]*Const.ESCALAXYZ);
    return (q);
  }


  public static int Cart2pr(double p) {
    return ((int) Math.floor(p*Const.ESCALAPR));
  }

  public static int[] Cart2pr(double[] p) {
    int[] q=new int[2];
```

```
    q[0]=(int) Math.floor(p[0]*Const.ESCALAPR);
    q[1]=(int) Math.floor(p[1]*Const.ESCALAPR);
    return (q);
  }

  public static double xyz2Cart(int x) {
    return ((double) x/(double) Const.ESCALAXYZ);
  }

  public static double[] xyz2Cart(int[] p) {
    double[] q=new double[3];
    q[0]=((double) p[0]/(double) Const.ESCALAXYZ);
    q[1]=((double) p[1]/(double) Const.ESCALAXYZ);
    q[2]=((double) p[2]/(double) Const.ESCALAXYZ);
    return (q);
  }

  public static double pr2Cart(int p) {
    return ((double) p/(double) Const.ESCALAPR);
  }

  public static double[] pr2Cart(int[] p) {
    double[] q=new double[2];
    q[0]=((double) p[0]/(double) Const.ESCALAPR);
    q[1]=((double) p[1]/(double) Const.ESCALAPR);
    return (q);
  }
}
```

7 MODELOS PARA LA GENERACIÓN DE TRAYECTORIAS

El robot SCORBOT puede describir dos tipos de trayectorias en su movimiento. Éstas trayectorias serán ambas lineales, aunque en espacios diferentes. Para trabajar con el robot SCORBOT podemos usar el espacio articular o el espacio cartesiano.

El espacio articular es el definido por la quíntupla de valores de cada uno de los ejes del robot.

El espacio cartesiano viene dado por otra quíntupla, esta vez definida por la posición del efector final y dos ángulos.

De este modo debemos generar trayectorias lineales tanto en el espacio articular como en el cartesiano ya que existen comandos en el lenguaje ACL que generan trayectorias de ambos tipos.

Por una trayectoria hemos de entender una secuencia de puntos que une un valor inicial con uno final, y además el tiempo que se tardará recorrer dichos puntos.

En el simulador se han separado estas dos partes de la trayectoria. Por un lado se generan las secuencias de puntos, y por otro se controla la duración del movimiento. Estas dos partes se encuentran en el objeto *MovimientoRobot*.

7.1 Generador del movimiento

Para la generación de los movimientos del modelo se utiliza un hilo de ejecución aparte del principal. La función principal de este hilo es mandar las quíntuplas de valores generadas por los generadores de trayectorias a las articulaciones del robot, para de esta forma simular el movimiento del robot.

El hilo va tomando los valores de una tabla bidimensional de 5 columnas y un número de filas dado por una constante llamada *MAXPOINTS*. Cada fila corresponde con una quíntupla de valores para las articulaciones. Existen dos índices para controlar dicha tabla: *ActualIndex* y *NextIndex*. El índice *NextIndex* es el que usarán los generadores de trayectorias para conocer a partir de donde deben empezar a crear las nuevas secuencias de puntos. Por otra parte, *ActualIndex* es utilizado por el hilo generador de la animación para conocer que quíntupla es la que debemos mandar a las articulaciones.

El funcionamiento de la tabla es a modo de cola circular, es decir, cuando uno de los índices llega al final se inicializa otra vez al principio. En el siguiente apartado veremos los problemas que genera el uso de este tipo de colas.

El hilo posee una variable que actúa a modo de semáforo. Esta variable puede tomar dos valores posibles: cero o mayor que cero. Cuando vale cero indica que no existen trayectorias nuevas y por tanto el hilo debe dormir. Cuando vale mayor que cero indica que existen trayectorias por ser ejecutadas y el hilo no dormirá hasta que no termine de ejecutar todas las trayectorias.

Este hilo tiene la categoría de demonio debido a que se estará siempre en ejecución. Esto podría haberse obviado ya que en el código del hilo existe un bucle que hará que siempre se ejecute, pero de esta forma nos aseguramos de que cuando termine el simulador también terminará este hilo.

Posee dos bucles distintos, con el primero se controla la animación del modelo en 3D y con el segundo la del modelo simple en 2D.

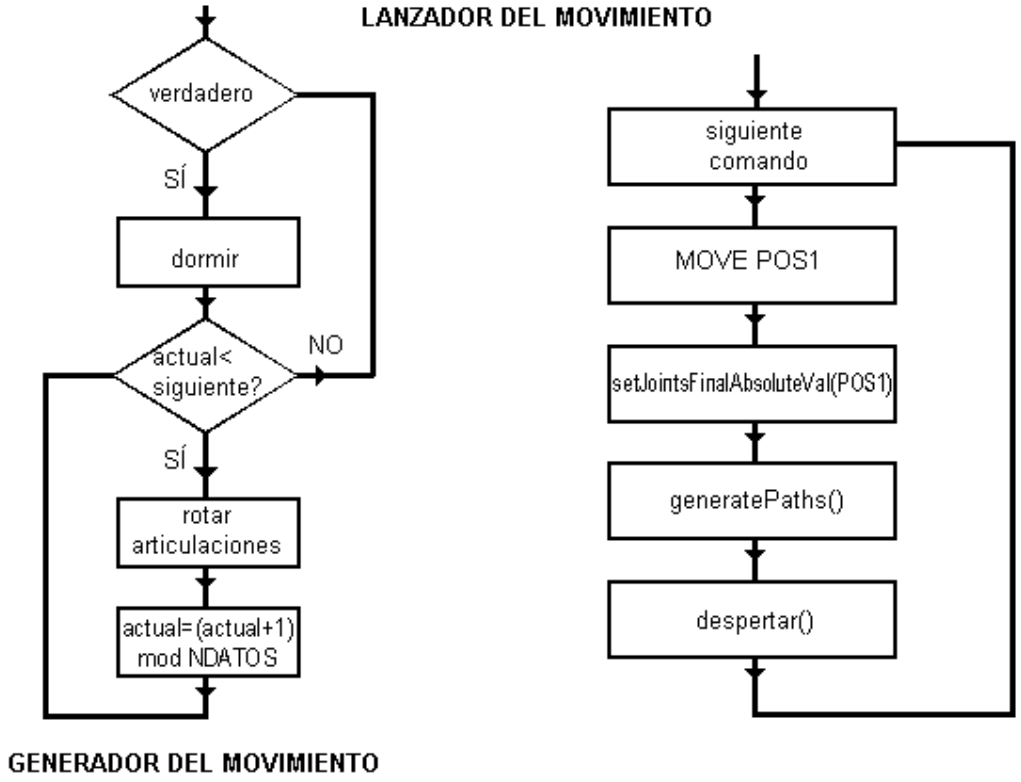
```
private static class Actualizador extends Thread {

  private int semaforo=1;

  public Actualizador() {
    this.setDaemon(true);
    this.setPriority(Thread.NORM_PRIORITY);
  }

  public synchronized void espera() {
    semaforo--;
    if (semaforo==0) {
      moving=false;
      try {wait();}
      catch (InterruptedException e) {}
    }
  }

  public synchronized void despierta() {
    semaforo++;
    if (semaforo==1) notify();
  }

  public void run() {
    double aux;

    if (Const.CARGAROBOT) {
      //movimiento en 3D
      while (true) {
        espera();

        while (ActualIndex!=NextIndex) {
          for (int i=0;i<Const.NUMJOINTS;i++) {
            aux=jointsVector[i][ActualIndex]; JointsActual[i]=aux;
            switch (i) {
              case 0: case 4: JointsRot[i].rotY(aux); break;
```

```
              case 1: case 2: case 3: JointsRot[i].rotX(aux); break;
            }

            if (Const.SERVOSEnabled[i])
              joints[i].setTransform(JointsRot[i]);
          }

          MovimientoObjetos.calculaCUADRICULA(JointsActual,cuadricula);
          MovimientoObjetos.updateObjeto(cuadricula);
          if (ActualIndex%Const.LIMITCHECK==0) checkLimits();

          ActualIndex++;
          if (ActualIndex==Const.MAXPOINTS) ActualIndex=0;

          try {sleep(50);}
          catch (InterruptedException e) {break;}
        } //animando
      } //esperando
    }
    else { //si no se carga el modelo
      //movimiento en 2D
      while (true) {
        espera();

        while (ActualIndex!=NextIndex) {
          //copia el buffer a la articulación
          for (int i=0;i<Const.NUMJOINTS;i++)
            if (Const.SERVOSEnabled[i])
              JointsActual[i]=jointsVector[i][ActualIndex];

          robot2d.updateJ(JointsActual);
          if (ActualIndex%Const.LIMITCHECK==0) checkLimits();
          ActualIndex++;
          if (ActualIndex==Const.MAXPOINTS) ActualIndex=0;

          try {sleep(50);}
          catch (InterruptedException e) {break;}
        } //animando
      }
    }
  }

}
```

De una simple observación de cualquiera de los dos bucles podemos observar que este hilo manda a cada uno de los modelos la secuencia de puntos a un ritmo independiente de la velocidad que queramos para el movimiento. Aunque la velocidad podría haberse controlado aquí sin más que controlar el ritmo de actualización de los valores, esto conlleva muchos problemas. El ritmo se controlará con un comando *sleep()* con un argumento que cambiará dependiendo de la velocidad, es decir, a mayor velocidad menor deberá ser el argumento. Debido a que Java es multitarea nunca se puede asegurar que se espere

la cantidad indicada, lo único que podemos asegurar es que ese tiempo será lo mínimo que se va a esperar.

El esquema general del movimiento en el simulador es el siguiente:



Existe una variable lógica en el simulador llamada MOVING (esta variable es copia de otra variable interna llamada *moving*), la cual indica cuando está siendo ejecutada una trayectoria por el modelo. Esta variable nos servirá para una serie de comandos de movimiento que deben verificar si el movimiento que ordenaron finalizó. Esta variable toma un valor negativo cuando el hilo *Actualizador* duerme. Tomará el valor contrario cuando los procedimientos que generan trayectorias despierten al hilo.

7.2 Generadores de trayectorias

Existen dos procedimientos para la generación de trayectorias: uno para el espacio articular y otra para el cartesiano.

Los comandos de movimiento como MOVE llaman a un procedimiento que sirve de intermediario entre los comandos y los generadores. Este procedimiento copia el valor de la posición final a una variable local al objeto *MovimientoRobot* y se encarga de llamar a uno de los dos generadores dependiendo del caso. El generador llamado usará la posición actual (que será la última que fue generada, que puede o no coincidir con la actual de la representación) y la final para realizar una interpolación lineal para generar la trayectoria.

```
public static void setJointsFinalAbsoluteVal(int[] pval,boolean movel) {
  for (int i=0;i<Const.NUMJOINTS;i++)
    JointsRAD[i]=ModeloConv.Pulses2Rad(i,pval[i]);

  if (auto) {
    if (!movel) generatePaths();
    else generateLinearCartPaths();
  }
}
```

7.2.1 Trayectorias lineales en el espacio articular

El espacio articular es de dimensión 5, por tanto debemos generar 5 trayectorias, una para cada componente. Al hablar de trayectorias nos referimos a secuencia de valores y tiempo. Esta secuencia de valores nos servirá para unir dos puntos mediante una recta. Las cinco secuencias de valores tendrán una duración similar.

Para cada articulación se generará una serie de puntos que unirán el valor inicial con el valor final. La densidad de puntos en el intervalo será inversamente proporcional a la velocidad con la que se ejecutará el movimiento. Es decir, cuanto más rápido queramos que se ejecute la trayectoria, menos puntos se crearán en el intervalo.

Puesto que en el modelo usado en el simulador se controlan las articulaciones, no la posición del efector, sólo se debe generar la secuencia de valores para cada una de las articulaciones.

En el código que genera estas trayectorias podemos observar dos bucles: el exterior nos servirá para seleccionar la articulación sobre la que vamos a actuar, y el interior será el que generará la secuencia para dicha articulación.

El número de puntos que se generan en dicha secuencia viene definido por la variable *rspeed*. Esta variable depende del valor de la velocidad actual. Por ejemplo, si estamos utilizando velocidad máxima (100) sólo se generarán 10 puntos

en la secuencia. Por el contrario, si seleccionamos una velocidad del 10% de la máxima se generarán 100 puntos.

```
  private static void generatePaths() {
    double angi,angf;
    double rel,aux;
    int index=NextIndex;
    int rspeed=110-speed;

    for (int i=0;i<Const.NUMJOINTS;i++) {
      //ángulo inicial y final
      if (NextIndex-1<0) angi=jointsVector[i][Const.MAXPOINTS-1];
      else angi=jointsVector[i][NextIndex-1];
      angf=JointsRAD[i];
      //índice inicial
      index=NextIndex;

      for (int j=0;j<=rspeed;j++) {
        rel=((double) j)/((double) rspeed);
        aux=(angf-angi)*rel+angi;
        //comprobación de límites
        if (aux<Const.limites_minimos[i]) aux=Const.limites_minimos[i];
        if (aux>Const.limites_maximos[i]) aux=Const.limites_maximos[i];
        jointsVector[i][index]=aux;
        index++;
        if (index==Const.MAXPOINTS) index=0;
      }
    }


    //el final
    NextIndex=index;
    //por si ha habido clipping debemos cambiar el final
    //que debería tener la trayectoria
    int indfin=NextIndex-1;
    if (indfin<0) indfin=Const.MAXPOINTS-1;

    for (int i=0;i<Const.NUMJOINTS;i++)
      JointsRAD[i]=jointsVector[i][indfin];


    moving=true;
    GeneradorMov.despierta();
  }
```

Además este generador de trayectorias también se encarga de gestionar los límites de movilidad de las articulaciones. Podemos ver que dada una articulación *i* tiene unos límites máximo y mínimo determinados por las constantes *limite_maximo[i]* y *limite_minimo[i]*, respectivamente. Por tanto, cuando se genera un nuevo valor para el encoder de una articulación se comprueba que esté dentro de los límites

establecidos. Si excediese alguno de los límites tomaría el valor del límite.

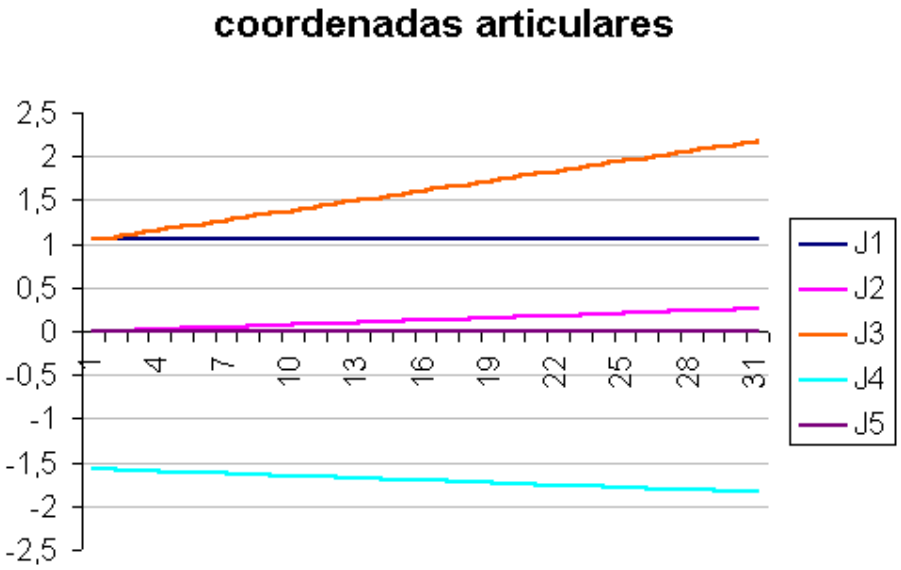
Al generar directamente las comprobaciones de movilidad en el buffer de movimiento estamos facilitando la siguiente labor. Al estar ejecutando una trayectoria si comprobásemos que el encoder va a tomar un valor no válido deberíamos modificar el buffer de movimiento. Esta modificación es debida a que al generar la siguiente trayectoria tenemos que partir de la última posición de la última trayectoria generada, y puede que esta posición no sea válida. Por tanto, la modificación se realiza al generar la trayectoria y no posteriormente al ejecutarla.

Podemos observar que la última acción que se realiza al generar la trayectoria es despertar al hilo *Actualizador* para avisarle de que existe otra trayectoria nueva.

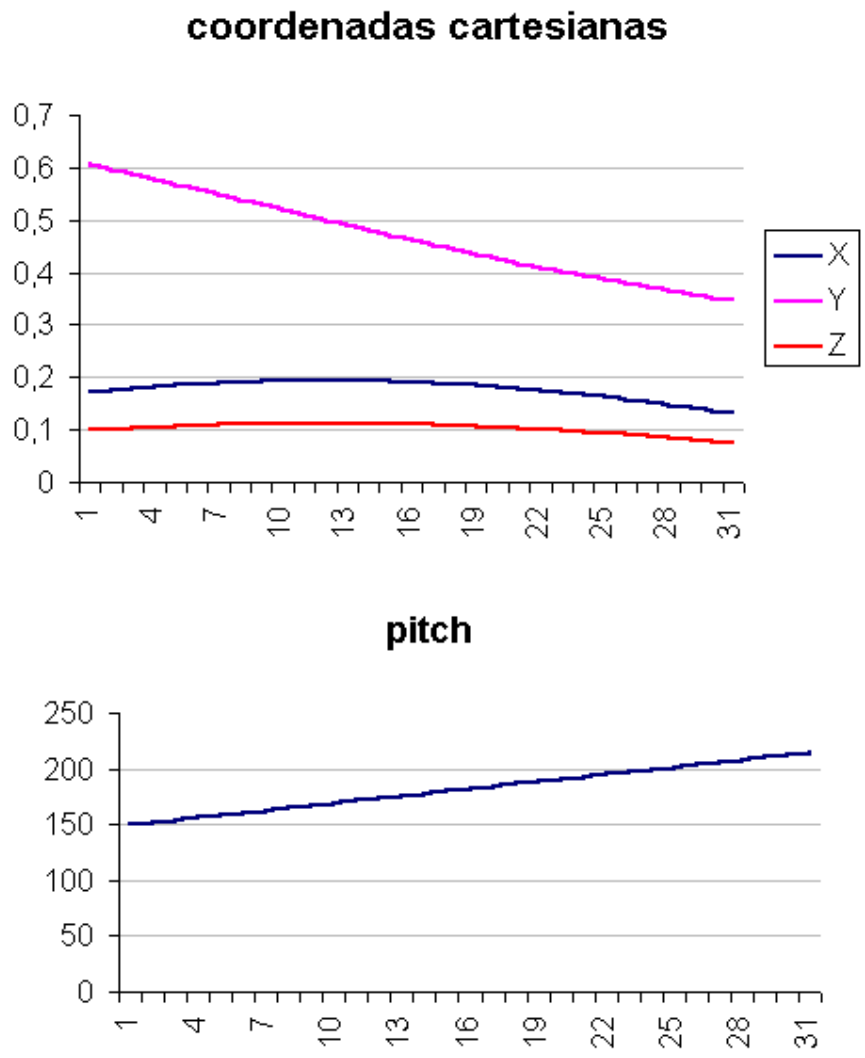
Como ejemplo de trayectoria generada hemos tomado una con los siguientes datos:

- velocidad 75% de la máxima
- posición inicial
- posición final

Las series a partir de las cuales han sido generadas estas gráficas se pueden consultar en la hoja de cálculo incluida en la documentación.



Esta trayectoria ha sido convertida al espacio cartesiano para después poder comparar esta trayectoria con la generada por el siguiente generador.



7.2.2 Trayectorias lineales en el espacio cartesiano

Ahora el argumento de entrada son las coordenadas cartesianas que indicarán el punto final que se desea alcanzar. Estas coordenadas vendrán dadas en coordenadas articulares, entonces lo primero que debemos hacer es convertirlas a coordenadas cartesianas.

Como hemos visto el hilo *Actualizador* necesita coordenadas en el espacio articular, por tanto la secuencia se generará en

el espacio cartesiano pero debemos convertirla al espacio articular antes de ser ejecutada.

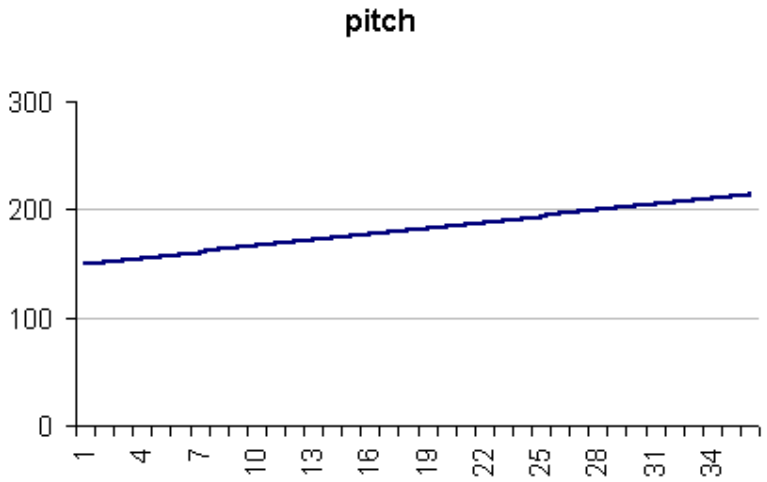
```
  private static void generateLinearCartPaths() {
    double[] xyz;
    double[] pr;
    double[] joints=new double[Const.NUMJOINTS];
    double xi,yi,zi,pi,ri;
    double xf,yf,zf,pf,rf;
    double[] xyzaux=new double[3];
    double[] praux=new double[2];
    double rel;

    int rspeed=110-speed;
    int index;
    if (NextIndex-1<0) index=Const.MAXPOINTS-1;
    else index=NextIndex-1;

    for (int i=0;i<Const.NUMJOINTS;i++)
      joints[i]=jointsVector[i][index];

    //inicial
    ModeloCinematicoDirecto.updatePosition(joints);
    xyz=ModeloCinematicoDirecto.getXYZ();
    pr=ModeloCinematicoDirecto.getPR();
    xi=xyz[0]; yi=xyz[1]; zi=xyz[2]; pi=pr[0]; ri=pr[1];

    //final
    ModeloCinematicoDirecto.updatePosition(JointsRAD);
    xyz=ModeloCinematicoDirecto.getXYZ();
    pr=ModeloCinematicoDirecto.getPR();
    xf=xyz[0]; yf=xyz[1]; zf=xyz[2]; pf=pr[0]; rf=pr[1];

    index=NextIndex;

    //generacion de las trayectorias
    for (int j=0;j<=rspeed;j++) {
      rel=((double) j)/((double) rspeed);
      xyzaux[0]=(xf-xi)*rel+xi;
      xyzaux[1]=(yf-yi)*rel+yi;
      xyzaux[2]=(zf-zi)*rel+zi;
      praux[0]=(pf-pi)*rel+pi; praux[1]=(rf-ri)*rel+ri;
      //ahora encontramos las articulaciones para la nueva posicion
      ModeloCinematicoInverso.updateJoints(xyzaux,praux);
      joints=ModeloCinematicoInverso.getJoints();

      //comprobación del límite de movilidad
      for (int i=0;i<Const.NUMJOINTS;i++) {
        if (joints[i]<Const.limites_minimos[i])
          joints[i]=Const.limites_minimos[i];
        else if (joints[i]>Const.limites_maximos[i])
          joints[i]=Const.limites_maximos[i];
        jointsVector[i][index]=joints[i];
      }

      index++;
      if (index==Const.MAXPOINTS) index=0;
```

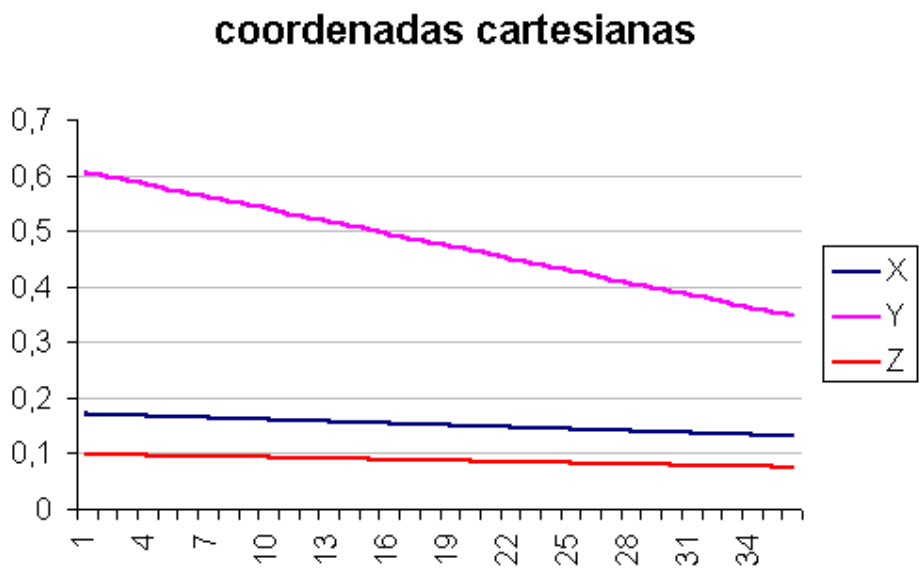
```
    }


    //el final
    NextIndex=index;
    //por si ha habido clipping debemos cambiar el final
    //que debería tener la trayectoria
    int indfin=NextIndex-1;
    if (indfin<0) indfin=Const.MAXPOINTS-1;

    for (int i=0;i<Const.NUMJOINTS;i++)
      JointsRAD[i]=jointsVector[i][indfin];

    moving=true;
    GeneradorMov.despierta();
  }
```

Como en el caso anterior podemos observar que la última acción que se realiza al generar la trayectoria es despertar al hilo *Actualizador* y actualizar la variable *moving*. Además también podemos ver que se realiza la comprobación de los límites de movimiento en la misma función y no al ejecutar la trayectoria.

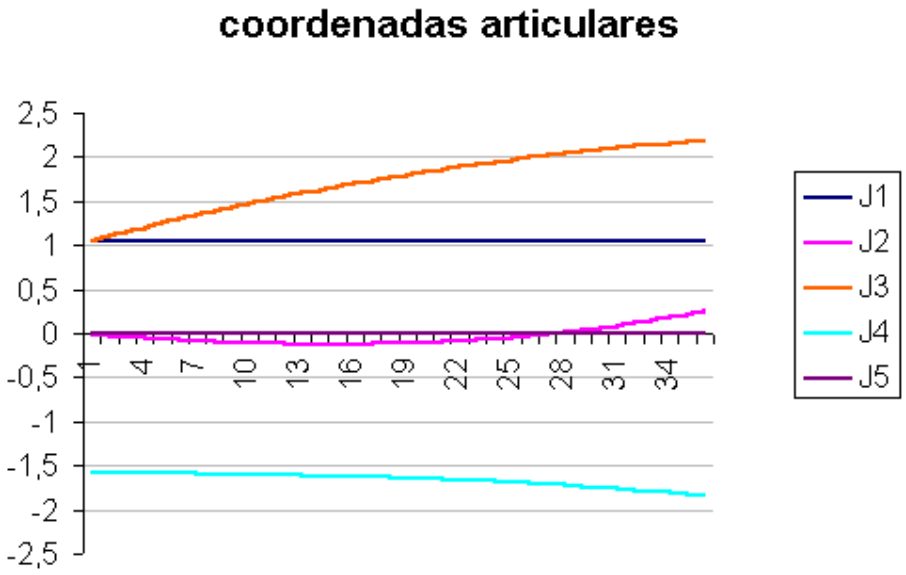
Para el ejemplo de este generador han sido usadas las mismas posiciones iniciales y finales, y como velocidad un 75% de la velocidad máxima.

La trayectoria generada en el espacio cartesiano es la siguiente:





La conversión de dicha trayectoria al espacio articular nos da la siguiente trayectoria:



7.2.3 Gestión de los límites físicos de las articulaciones

Como ya vimos en los apartados anteriores para cada articulación existen dos constantes llamadas *Const.limite_minimo* y *Const.limite_maximo* que hacen referencia al límite físico que posee el movimiento de dicha articulación. El valor de estos límites sólo puede ser cambiado en tiempo de diseño, es decir, antes de la compilación del programa. Este valor no es configurable debido a que en la máquina real tampoco lo es.

En la máquina real la comprobación de trayectorias válidas es más simple que en el simulador. Para cada articulación el robot SCORBOT posee un servomotor con un encoder que realiza la codificación de la posición para la máquina controladora. Estos servomotores están alimentados con corriente continua. En condiciones normales, es decir sin carga ni obstrucción, se puede prever el consumo de cada servomotor (aunque este varía con el tiempo ya que estos motores están sometidos al proceso del envejecimiento). Pero en condiciones anormales, es decir con exceso de carga o alguna obstrucción debido a choques con objetos o cercanía del límite físico del robot, el consumo de estos motores se dispara. Por tanto, la máquina sólo debe vigilar dichos consumos para controlar la movilidad del robot. Este control ya lo realiza porque es la misma máquina la que alimenta los motores, y una sobrecarga de los motores podría dañar la máquina además del motor.

Este tipo de control es fácil en la máquina real, ya que sólo debe cortar la alimentación del servomotor obstruido. Además la máquina también posee un pulsador de "parada de emergencia" porque incluso el simple control explicado puede fallar. Pero simular dicho control en el programa es muy complicado, debido a que deberíamos tomar un modelo del robot que tuviese en cuenta la dinámica del movimiento. En esta dinámica intervendría la gravedad, el peso de cada eslabón del robot, la posible carga que llevase el efector final, etc. Para hallar ese modelo existen herramientas que nos servirían para modelar el comportamiento dinámico del robot. Pero este modelo se aleja de la intención de este simulador que es reflejar el comportamiento del robot SCORBOT con el lenguaje ACL.

De todas formas se ha implementado un pequeño control de los límites físicos del robot. Como fue explicado en los apartados anteriores (7.2.1 y 7.2.2), cuando un valor para una articulación excede alguno de los límites impuestos por las constantes, toma el valor del límite excedido.

Posteriormente, en el hilo *Actualizador*, se realiza la ejecución de la trayectoria. En este hilo existe una llamada a una función denominada *checkLimits()*.

```
private static void checkLimits() {
  for (int i=0;i<Const.NUMJOINTS;i++) {
    if (JointsActual[i]==Const.limites_minimos[i])
      MainFrame_TextOUT.printOutLn("límite inferior en eje "+(i+1));
    else if (JointsActual[i]==Const.limites_maximos[i])
      MainFrame_TextOUT.printOutLn("límite superior en eje "+(i+1));
  }
}
```

Como podemos observar esta función sólo comprueba si el valor que ha tomado alguna de las articulaciones es su valor límite. Si la comprobación da un resultado positivo se muestra un mensaje por pantalla para avisar del hecho.

Dado que esta comprobación realmente sólo sirve para avisar del hecho al usuario, no es necesaria realizarla con cada valor que tome la articulación. Es por eso que la comprobación sólo se realiza una de cada *Const.LIMITCHECK* veces.

7.3 Tratamiento de objetos

Aquí hacemos referencia a los objetos con los que puede interactuar el modelo. Estos objetos han sido simplificados de tal modo que sólo se puede interactuar con cubos.

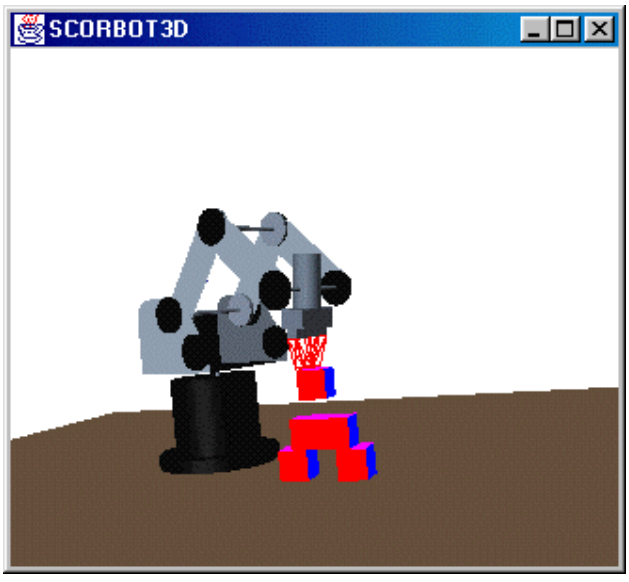
La creación y posicionamiento de los objetos se puede hacer a través de la ventana tratada en el apartado número 4.8. Para un tratamiento más fácil de los objetos se ha discretizado el espacio donde pueden estar presentes. Para ello ha sido dividido el espacio en cubos de lado igual a la mitad del lado de los cubos objeto, para de este modo aumentar la resolución del espacio de los objetos.

Para el manejo de los objetos existe un objeto llamado *NObjeto* que reúne todas los atributos posibles del objeto. Además sirve de enlace entre el objeto creado en el modelo real y la abstracción creada en el espacio discreto.

```
public class NObjeto {

  private String nombre;
  private int tipo=0;
  private int[] xyz={0,0,0};
  private TransformGroup obj;
  private Transform3D pos;
  private Vector3d vpos;
  private boolean yabajado=false;  //ya ha sido bajado una cuadrícula
```

```
  public NObjeto(String nombre) {
    this.nombre=nombre;
    obj=MovimientoObjetos.addObject(xyz);
    pos=new Transform3D();
    vpos=new Vector3d(0,0,0);
  }

  public String toString() {return nombre;}
  public int[] getxyz() {return xyz;}
  public int getx() {return xyz[0];}
  public int gety() {return xyz[1];}
  public int getz() {return xyz[2];}

  //posiciona al objeto en la cuadrícula dada
  public void setxyz(int x,int y,int z) {
    xyz[0]=x; xyz[1]=y; xyz[2]=z;
    vpos.set((xyz[0]*Const.CUBETAM)+Const.CUBETAM,
      (xyz[1]*Const.CUBETAM)+Const.CUBETAM,
      (xyz[2]*Const.CUBETAM)+Const.CUBETAM);
    pos.set(vpos);
    obj.setTransform(pos);
  }

  public boolean y_igual(int y) {
    return (y==xyz[1]);
  }

  public int gettipo() {return tipo;}
  public void settipo(int tipo) {this.tipo=tipo;}
  public String getnombre() {return nombre;}


  public void setyabajado(boolean nc) {yabajado=nc;}
  public boolean isyabajado() {return yabajado;}
}
```

La función más destacada de el objeto *NObjeto* es *setXYZ()*. Esta función permite situar a un objeto dentro del espacio de los objetos.



Para que el robot SCORBOT pueda interactuar con ellos el juego de comandos, el simulador ha sido dotado de una utilidad que transforma la posición actual del efector final al espacio discreto de los objetos. Esta utilidad puede ser consultada en la ventana que equivale a la

pistola de programación (ver apartado número 4.6), la cual proporciona una terna que representa la posición en el espacio de los objetos. Esta terna nos servirá para adecuar nuestra posición a la de los objetos.

De este modo, cuando el robot SCORBOT cierre la pinza del efector final se realizará una comprobación de la cuadrícula en la que el efector está posicionado. Esta cuadrícula será comprobada con todas en las que exista un objeto mediante el algoritmo especificado en el procedimiento *MovimientoRobot.cogerObjeto()*. Si de esta comparación se obtiene algún objeto, el objeto será guardado en la variable *objetoCogido*. Esto hará que dicho objeto actualice su posición con la posición del efector final mediante el procedimiento *MovimientoRobot.updateObjeto()*. De esta forma se conseguirá la impresión de que el objeto está literalmente cogido por el SCORBOT.

Es decir, para que un objeto sea cogido deben darse las siguientes condiciones:

- ✓ Que la posición en el espacio discreto del objeto verifique el algoritmo con la posición en el espacio discreto del efector final.
- ✓ Una vez verificada la anterior condición debe darse también la ejecución del comando CLOSE sobre dicha posición para que el objeto cambie su estado a "cogido".

El valor de la variable *objetoCogido* cambiará a un valor nulo cuando se ejecute el comando OPEN de apertura del efector final, de esta forma su posición dejará de ser actualizada.

Por simplificar el código no han sido introducidas las siguientes características, que pueden ser revisadas para posteriores modificaciones:

- detección de colisiones entre objetos
- detección de colisiones entre el robot SCORBOT y los objetos

Para la gestión de los objetos del tipo *NObjeto* fue creado la clase *MovimientoObjetos*. Este objeto proporciona los procedimientos para añadir y posicionar objetos. Además permite calcular la conversión al espacio discreto y verificar la posición de objetos en cuadrículas. También añade dos procedimientos para guardar y recuperar los objetos creados para un posterior uso.

```
public class MovimientoObjetos {

  public static Vector vObjetos=new Vector();
```

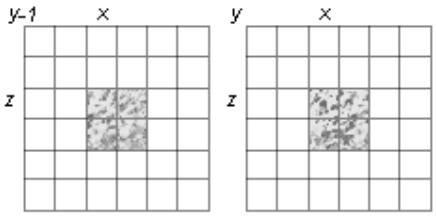
```
  private static RobotObj escena;
  private static Nobjeto objetoCogido=null;

  public static void init(RobotObj obj) {
    escena=obj;
  }

  public static TransformGroup addObject(int[] xyz) {
    double[] dxyz=new double[3];
    dxyz[0]=2*xyz[0]*Const.CUBETAM;
    dxyz[1]=2*xyz[1]*Const.CUBETAM;
    dxyz[2]=2*xyz[2]*Const.CUBETAM;
    return (escena.addCube(dxyz));
  }
```

Las dos funciones siguientes son las encargadas de la conversión de coordenadas del espacio articular al espacio de los objetos la primera, y del espacio de los objetos al cartesiano la segunda.

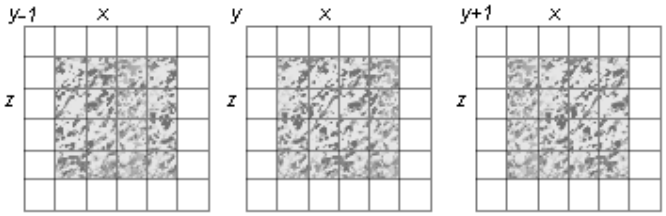
```
  public static void calculaCUADRICULA(double[] jactual,int[] cuad) {
    ModeloCinematicoDirecto.updatePosition(jactual);
    double[] xyz=ModeloCinematicoDirecto.getXYZ();
    cuad[0]=((int) Math.floor(xyz[0]/(Const.CUBETAM)))+1;
    cuad[1]=((int) Math.floor(xyz[1]/(Const.CUBETAM)))-5;
    cuad[2]=((int) Math.floor(xyz[2]/(Const.CUBETAM)))+1;
  }

  public static void calculaCOORD(int[] cuad,double[] xyz) {
    xyz[0]=(double) ((cuad[0]-1)*Const.CUBETAM);
    xyz[1]=(double) ((cuad[1]+5)*Const.CUBETAM);
    xyz[2]=(double) ((cuad[2]-1)*Const.CUBETAM);
  }
```

La función *cuadriculaOCUPADA()* es usada para el posicionamiento de los objetos. Antes de posicionar un objeto se realiza una comprobación para calcular si existe riesgo de solapamiento con otro objeto presente en las cercanías.

Debido a que la retícula está realizada con un tamaño LADOOBJETO/2 debemos realizar las siguientes comprobaciones.



Casillas ocupadas por un objeto posicionado en (x,y,z)



Casillas con posibilidad de colisionar con las ocupadas por un objeto en (x,y,z)

corresponden a objetos en:
(x-1,y-1,z-1), (x,y-1,z-1), (x+1,y-1,z-1), (x-1,y-1,z), (x,y-1,z), (x+1,y-1,z), (x-1,y-1,z+1), (x,y-1,z+1), (x+1,y-1,z+1), (x-1,y,z-1), (x,y,z-1), (x+1,y,z-1), (x-1,y,z), (x,y,z), (x+1,y,z), (x-1,y,z+1), (x,y,z+1), (x+1,y,z+1), (x-1,y+1,z-1), (x,y+1,z-1), (x+1,y+1,z-1), (x-1,y+1,z), (x,y+1,z), (x+1,y+1,z), (x-1,y+1,z+1), (x,y+1,z+1), (x+1,y+1,z+1)

```
public static boolean cuadriculaOCUPADA(int[] cuad) {
  int[] aux;
  int j;

  for (int i=0;i<vObjetos.size();i++) {
    aux=((NObjeto) vObjetos.get(i)).getxyz();
    for (j=-1;j<2;j++) { //j=-1,0,1
      if (aux[1]==cuad[1]+j) {
        if ((aux[0]==cuad[0]-1 && aux[2]==cuad[2]-1) ||
          (aux[0]==cuad[0] && aux[2]==cuad[2]-1) ||
          (aux[0]==cuad[0]+1 && aux[2]==cuad[2]-1) ||
          (aux[0]==cuad[0]-1 && aux[2]==cuad[2]) ||
          (aux[0]==cuad[0] && aux[2]==cuad[2]) ||
          (aux[0]==cuad[0]+1 && aux[2]==cuad[2]) ||
          (aux[0]==cuad[0]-1 && aux[2]==cuad[2]+1) ||
          (aux[0]==cuad[0] && aux[2]==cuad[2]+1) ||
          (aux[0]==cuad[0]+1 && aux[2]==cuad[2]+1))
            return (true);
      }
    }
  }

  return (false);
}
```

El procedimiento que sigue es el que se ejecuta cuando se cierra la pinza del efector final con un comando CLOSE. Este procedimiento devuelve *true* si fue cogido un objeto.

Para simular el hecho de coger un objeto se pensó en algo más realista que solamente la acción de comprobar que la pinza estuviese posicionada en la misma cuadrícula que el objeto y por ello fue realizado el siguiente algoritmo.

El primer punto es calcular la "distancia" definida en el espacio de los objetos que existe entre la posición del

efector final y todos los objetos que existen en la escena. A la vez que es calculada la distancia se va escogiendo el objeto cuya distancia sea mínima. Para finalizar sólo debemos comprobar que dicha distancia sea menor que la constante *LIMITCOGER*, la cual nos dará el rango de actuación para el efector final.

Cuando tengamos seleccionado dicho objeto será cambiada su posición para darle más realidad al hecho de ser cogido, y por otra parte debemos cambiar el valor de la variable *objetoCogido* para que apunte al objeto. De esta forma la posición del objeto cogido será actualizada con el movimiento del robot.

```
public static boolean cogerObjeto(int[] cuad) {
  int[] aux;
  int dist,mindist=Integer.MAX_VALUE;
  NObjeto objaux,objmin=null;

  for (int i=0;i<vObjetos.size();i++) {
    objaux=(NObjeto) vObjetos.get(i);
    aux=objaux.getxyz();
    dist=(aux[0]-cuad[0])*(aux[0]-cuad[0])+
      (aux[1]-cuad[1])*(aux[1]-cuad[1])+
      (aux[2]-cuad[2])*(aux[2]-cuad[2]);
    if (dist<mindist) {
      mindist=dist; objmin=objaux;
    }
  }

  if (mindist<Const.LIMITCOGER && objmin!=null) {
    objmin.setxyz(cuad[0],cuad[1],cuad[2]);
    objetoCogido=objmin;
    return (true);
  }

  return (false);
}
```

La operación de soltar un objeto es más simple que la contraria. Ahora se da la circunstancia de que sólo existe un objeto que cumpla la condición. Por tanto sólo debemos poner un valor nulo en la variable *objetoCogido*.

```
public static void soltarObjeto() {
  objetoCogido=null;
}
```

Para actualizar la posición del objeto cogido sólo debemos comprobar cuál es dicho objeto y hacer su posición igual a la del efector final dada por la terna *cuad*. Este procedimiento es llamado por el hilo *Actualizador* explicado en el apartado

número 7.1 para que el objeto sea actualizado a la vez que el modelo.

```
public static void updateObjeto(int[] cuad) {
  if (objetoCogido!=null)
    objetoCogido.setxyz(cuad[0],cuad[1],cuad[2]);
}
```

La caída de los objetos también ha sido simulada. Verdaderamente no ha sido simulada la condición de gravedad, ya que eso haría cambiar también los modelos usados del robot SCORBOT (ver apartado número 5).

Se ha supuesto que todos los objetos caen a velocidad constante, es decir el valor de la gravedad en el espacio del robot es 0 m/s^2. Al igual que en procedimiento *updateObjeto()*, es el hilo actualizador quien se encarga de llamar a esta función *caidaLibre()* que es la que simula la caída de los objetos.

Este procedimiento debe primero identificar que objetos debe mover. Los objetos que no deben caer son los que ya están en el suelo (nivel y=0) y el objeto que está cogido por el robot. Todos estos objetos tomarán un valor *true* en su atributo *yabajado*. Por tanto, todos los que tengan el valor *false* deben bajar un nivel cada vez que se ejecute el procedimiento *caidaLibre()*.

Una vez señalados los objetos que deben caer debemos empezar primero desde los niveles inferiores, esto es "y" igual a 1, hasta el nivel superior marcado por el objeto con "y" más alta.

Antes de bajar un objeto debemos comprobar que puede bajar para ello debemos realizar las mismas comprobaciones señaladas en el procedimiento *cuadriculaOCUPADA()*, pero sólo dos niveles por debajo.

El objeto ya bajado es marcado con *true* en su atributo *yabajado* para que no vuelva a descender en la misma llamada.

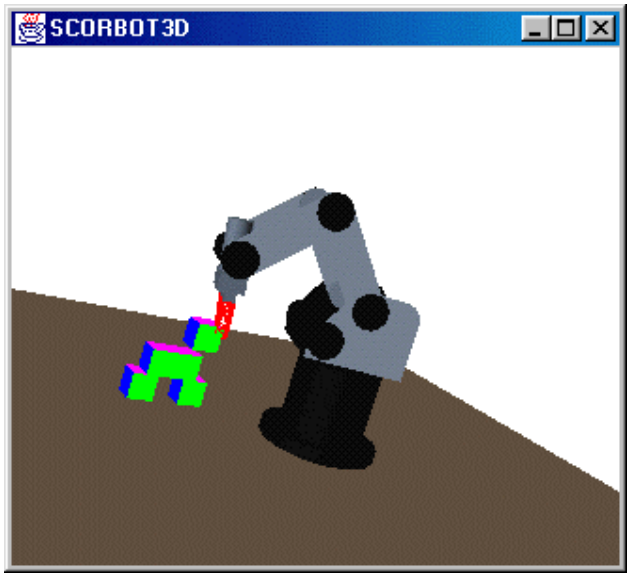
Para un uso adecuado de los objetos debemos proceder de la siguiente manera:

- ✓ crear una posición (POSA) en la cual serán recogidos los objetos
- ✓ posicionar el robot en POSA
- ✓ obtener la cuadrícula correspondiente a POSA
- ✓ crear un objeto en dicha cuadrícula
- ✓ crear una posición (POSB) en la cual serán depositados los objetos

Una vez realizados estos pasos la secuencia a seguir será la siguiente:

- ✓ abrir la pinza del efector final
- ✓ posicionar al robot en POSA
- ✓ cerrar la pinza del efector final
- ✓ posicionar el robot en POSB
- ✓ abrir la pinza del efector final
- ✓ volver a la posición inicial

Con esta secuencia de órdenes habremos conseguido mover un objeto desde POSA hasta POSB.



8 SOPORTE DE PROGRAMA

Bajo este apartado han sido incluidas las bases sobre las que se fundamenta el simulador del lenguaje ACL. El lenguaje ACL proporciona las herramientas básicas que posee un lenguaje de programación cualquiera: tipos de datos y comandos. Además este lenguaje sirve de intermediario entre el usuario y la máquina en la que se ejecuta el lenguaje. Esta máquina es la que controla el movimiento del robot, se encarga de gestionar las entradas y salidas digitales, etc. Por tanto, nuestro simulador debe ser capaz de realizar todas esas operaciones.

Para realizar las operaciones propias del lenguaje ACL ejecutándose en la máquina que controla el SCORBOT, el simulador se ayuda de las herramientas que serán mostradas a continuación.

Todo el código que sirve para simular los tipos de datos existentes en ACL está englobado dentro del paquete *Tipos*.

8.1 Variables

El lenguaje ACL maneja sólo tipos enteros, es decir, números enteros positivos y negativos. La máquina real posee un rango de 16 bits para los enteros. Si se utiliza la codificación en complemento a 2, los enteros estarán entre el -32768 y 32767. Esta restricción también ha sido simulada pero no se utiliza.

Las variables se han dividido en dos tipos distintos: simples y tablas. Para cada una ha sido creada una clase para su manejo.

Una variable del tipo simple sólo puede guardar un entero. Por el contrario, una variable tabla puede guardar un número de enteros dado por el tamaño de la tabla en su creación.

Para el manejo de todo el conjunto de variables simples (todos los objetos *Simple* creados) existe la clase *Simples*. De la misma manera para los objetos de la clase *Tabla* existe la clase *Tablas*.

A una variable se le asocian una serie de atributos comunes como son: *nombre*, *padre* y *valor*. El atributo *nombre* hace referencia al nombre de la variable. El atributo *padre* será el programa que ha sido responsable de crear la variable.

Pueden existir variables con el mismo atributo *nombre*, pero no pueden coincidir también en el atributo *padre*.

8.1.1 Simples

Para el manejo de las variables simples tenemos una clase denominada *Simple* con los siguientes procedimientos:

```
package SCORBOTSimulator.Tipos;

public class Simple {

  private String nombre;
  private int valor;
  private String parent;

  public Simple(String nombre,int valor,String parent) {
    this.nombre=nombre.toUpperCase();
    this.valor=valor;
    this.parent=parent.toUpperCase();
  }

  public String getNombre() {return(nombre);}
  public String getParent() {return (parent);}
  public synchronized int getValor() {return(valor);}

  //set valor con overflow
  public synchronized void setValor(int valor) {
    if (valor>Const.MAXINT) valor=Const.MAXINT;
    else if (valor<Const.MININT) valor=Const.MININT;
    this.valor=valor;
  }

  public String toString() {return (nombre+" ("+parent+")");}
}
```

Los procedimientos de la clase permiten acceder a los campos internos de la clase. Es necesario mencionar que se ha tenido en cuenta que estamos emulando una máquina que es multitarea. Por tanto el acceso al valor de la variable, tanto en escritura como en lectura, debe ser sincronizado. En este contexto la palabra sincronizado adquiere el significado de que solamente puede ejecutarse un procedimiento dentro del objeto *Simple*. Esto quiere decir que cuando dos hilos de ejecución accedan al valor de una variable simple lo harán de forma secuencial, es decir, uno primero y otro después.

El valor de la variable es guardado en el campo *valor*, y los programas accederán a él a través de los procedimientos *getValor()* y *setValor()*.

La clase que se encargará de dar acceso a los comandos a los objetos *Simple* será la clase *Simples*. Esta clase permite la creación, modificación y destrucción de objetos *Simple*.

```
package SCORBOTSimulator.Tipos;
```

```
public class Simples {

  private static Vector tablaVariables=new Vector();
  private static long starttime;

  public static void init() {
    try {
      add("MOVING",Const.FALSE,"GLOBAL");
      add("TIME",0,"GLOBAL");
    }
    catch (Exception err) {
      System.err.println("no se ha podido crear la variable MOVING");
      System.exit(-1);
   }

   starttime=System.currentTimeMillis()/10;
  }

  public static int size() {return (tablaVariables.size());}
  public static Vector listAll() {return (tablaVariables);}
```

Esta clase se encarga de crear y destruir los objetos *Simple*, con los procedimientos *add()* y *erase()* y de dar acceso al valor de las variables con *get()* y *set()*.

```
  public static synchronized void add(String nombreVariable,int
valor,String parent) throws Exception {
    nombreVariable=nombreVariable.toUpperCase();
    CommonTools.isValidName(nombreVariable);
    if (contains(nombreVariable,parent) || //esta ya definida localmente?
      Tablas.contains(nombreVariable,parent)) //es una tabla definida
localm?
      throw (Errores.VariableYaCreada);

    tablaVariables.add(new Simple(nombreVariable,valor,parent));
  }

  public static synchronized void erase(String nombreVariable,String
parent) throws Exception {
    int index=indexOf(nombreVariable,parent);

    if (index==-1)
      throw (Errores.VariableNoExiste);

    if (nombreVariable.equals("MOVING")||nombreVariable.equals("TIME"))
      throw (Errores.VariableNoBorrable);

    tablaVariables.removeElementAt(index);
  }
```

Para acceder a los objetos *Simple* tenemos los procedimientos *set()* y *get()*, que permiten el acceso en

escritura y en lectura respectivamente. Para que el acceso tenga éxito debemos tener visibilidad de la variable. Este concepto de visibilidad será explicado posteriormente.

```
  public static synchronized void set(String nombreVariable,int
valor,String pparent) throws Exception {
    int index=indexOf(nombreVariable,pparent);
    if (index==-1) index=indexOf(nombreVariable,"GLOBAL");
    if (index==-1) throw (Errores.VariableNoExiste);

    if (nombreVariable.equals("MOVING")||nombreVariable.equals("TIME"))
      throw (Errores.VariableSoloLectura);

    ((Simple) tablaVariables.get(index)).setValor(valor);
  }

  public static synchronized int get(String nombreVariable,String pparent)
throws Exception {
    int index=indexOf(nombreVariable,pparent);
    if (index==-1) index=indexOf(nombreVariable,"GLOBAL");

    if (index==-1)
      throw (Errores.VariableNoExiste);

    if (nombreVariable.equals("MOVING"))
      return (MovimientoRobot.isMoving()?Const.TRUE:Const.FALSE);

    if (nombreVariable.equals("TIME"))
      return ((int) ((System.currentTimeMillis()/10)-starttime));

    return (((Simple) tablaVariables.get(index)).getValor());
  }
```

También posee un procedimiento que destruye todos los objetos *Simple* que poseen un atributo *padre* dado. Esto permitirá la destrucción de variables locales una vez haya terminado el programa que las creó.

```
  public static synchronized void eliminaSimples(String parent) {
    Simple aux;
    int i=0;

    while (i<tablaVariables.size()) {
      aux=(Simple) tablaVariables.get(i);
      if (aux.getParent().equals(parent)) tablaVariables.remove(i);
      else i++;
    }

  }
}
```

Los procedimientos que necesitan también de un parámetro de entrada *parent* (ver apartado número 10), son aquellos que

antes de hacer algún cambio sobre un objeto *Simple* deben comprobar si ese parámetro es igual al del objeto que deben modificar.

El atributo *padre* de los objetos *Simple* nos permitirá gestionar la visibilidad de las variables. Esta capacidad será mostrada con un ejemplo.

ejemplo ———————————————————————————————

Si una variable es creada dentro de un programa PROG con el comando: "DEFINE VAR", entonces se creará un objeto *Simple("VAR",0,"PROG")*.

Cuando el programa encuentre una línea como esta: SET VAR=2, se realizará una llamada a *set("VAR",2,"PROG")*, y puesto que coinciden el nombre del programa con el atributo *padre* del objeto *Simple*, podrá ser modificado su campo *valor*.

Si por el contrario desde otro programa PROG2 realizamos el mismo comando *set("VAR",2,"PROG2")*, no nos dejará terminar la acción ya que no coinciden los atributos.

———————————————————————————————————————

Los procedimientos *set()* y *get()* de la clase *Simples* realizan una comprobación de existencia del objeto *Simple* del siguiente modo: primero miran si existe uno que coincida en *nombre* y *padre*, y si no hubo éxito entonces se comprobaría la existencia de uno con el mismo *nombre* y *padre="GLOBAL"*. Esto quiere decir que primero se comprueba si existe una variable local con dicho nombre, y después si existiese una global.

De esta manera se da prioridad a las variables locales sobre las globales, en el caso de que existiesen dos variables con el mismo *nombre*.

Entonces, si desde el modo directo ejecutamos SET VAR=2, se realizará la siguiente acción *set("VAR",2,"GLOBAL")*. La acción tampoco tendrá éxito porque no existe el objeto *Simple("VAR","GLOBAL")*.

Si desde el modo directo ejecutamos GLOBAL VAR, se creará un objeto *Simple("VAR",0,"GLOBAL")*. Ahora desde el programa PROG realizando SET VAR=2, sí tendremos éxito puesto que estaremos usando *set("VAR",2,"PROG")* y no encontrará la variable con el atributo *padre="PROG"*, pero sí *padre="GLOBAL"*.

Con todo esto conseguimos los siguientes objetivos:

- ✓ la creación de variables locales y globales.
- ✓ la coexistencia de variables locales y globales con igual nombre, ya que lo que las diferencia es el atributo *padre*.

- ✓ la visibilidad de variables locales sólo desde donde fueron creadas, verificando el atributo *padre* cada vez que se intenta acceder a ellas.
- ✓ y por último, la visibilidad de variables globales desde todos los programas, incluso desde el modo directo.

Hay procedimientos que verifican la visibilidad de una variable, si esta no se cumple lanzan una excepción que será manejada por el procedimiento que llamó al primero.

Las excepciones posibles son las siguientes:

- la variable no es visible
- la variable no existe

El procedimiento *init()* de la clase *Simples* realiza la creación de dos variables del sistema: TIME y MOVING. Estas variables son gestionadas de una manera especial, puesto que su valor no es guardado en el objeto *Simple* sino que sirven para leer datos del sistema. Por tanto serán tratadas como variables de sólo lectura.

La variable TIME nos expresa el tiempo en centésimas de segundo que lleva activo el simulador. Y la variable MOVING nos indica si está siendo ejecutado algún movimiento por el robot SCORBOT.

El procedimiento *eliminaSimples()* nos servirá para eliminar todas las variables locales de un programa una vez haya concluido su ejecución.

La sincronización también debe hacerse a este nivel, pero ahora la razón es otra. En este caso el problema podría venir al acceder al campo *valor* de una variable que puede ser destruida (procedimiento *erase()*) sin haber terminado de realizar el procedimiento *get()*, por ejemplo.

No es frecuente que el problema expuesto se presento, pero podría ocurrir debido a que el recurso compartido (la variable *Vector tablaVariables*) es visible para cualquier procedimiento de la clase. Por eso sólo podemos dejar que un procedimiento acceda a la variable a la vez.

Esta sincronización se repite en las otras clases que manejan tanto variables como posiciones para el simulador.

8.1.2 Tablas

Al igual que para las variables tipo simple tenemos dos clases para manejar este tipo de variables. Existe una clase *Tabla* que es la que se encargará de guardar los valores y otra clase *Tablas* que manejará todos los objetos tipo *Tabla*.

```
package SCORBOTSimulator.Tipos;

public class Tabla {

  private String nombre;
  private int[] valores;
  private String parent;

  public Tabla(String nombre,int tam,String parent) throws Exception
  public String getNombre()
  public int getSize()
  public synchronized void setValor(int pos,int valor) throws Exception
  public synchronized int getValor(int pos) throws Exception
  public String getParent()
  public String toString()
}
```

El constructor del objeto *Tabla* necesita también del tamaño de la tabla a crear, ya que el campo *valores[]* es creado al ser creada la variable (en realidad el objeto *Tabla*) y es aquí donde serán guardados los elementos de la tabla.

Este clase posee tres miembros privados que guardan los atributos de la variable: el nombre, el padre y la verdadera tabla de valores. Los procedimientos dan acceso a esos atributos.

En este tipo de variables tenemos además un procedimiento para obtener el tamaño de la tabla creada, el procedimiento *getSize()*.

Hay que mencionar el hecho de la sincronización en el acceso a los valores de la tabla por el problema antes explicado. Además existe un nuevo mensaje de error dado por este objeto ya que debido a que tenemos que hacer una comprobación de la posición de la tabla a la que se está accediendo, estos procedimientos pueden lanzar una excepción del tipo *IndiceFueraDeRango*.

Como curiosidad citar el hecho de que en el lenguaje ACL en una tabla de N elementos, el primer elemento es el número 1 y el último es el N. Sin embargo en Java ocurre que el primero es el elemento número 0 y el último es el N-1. Por tanto, los procedimientos *setValor()* y *getValor()* realizan un decremento en una unidad en el índice antes de acceder a un elemento guardado en la tabla *valores[]*.

Para el acceso a los objetos *Tabla* por parte de los comandos ACL, ha sido creada la clase *Tablas*.

```
package SCORBOTSimulator.Tipos;

public class Tablas {

  private static Vector tablaArrays;
```

```
  public static void init()
  public static int size()
  public static Vector listAll()
  public static boolean contains(String nombreTabla,String parent)
  public static synchronized void add(String nombreTabla,int tam,String
parent) throws Exception
  public static synchronized int get(String nombreTabla,int pos,String
parent) throws Exception
  public static synchronized int getSize(String nombreTabla,String parent)
throws Exception
  public static synchronized void set(String nombreTabla,int pos,int
valor,String parent) throws Exception
  public static synchronized void erase(String nombreTabla,String parent)
throws Exception
  public static synchronized void eliminaTablas(String parent)
}
```

El acceso a los objetos *Tabla* se hace de igual forma que el de los objetos tipo *Simple*. La única diferencia es que para acceder a un valor debemos dar también la posición del elemento de la tabla al que queremos acceder.

La comprobación de rango se hace en el propio objeto *Tabla*. Existe además otro procedimiento para acceder al tamaño de la tabla.

El procedimiento *init()* crea tres tablas del sistema que serán IN, OUT y ENC. Las dos primeras nos permiten acceder a los valores de las entradas y salidas digitales. Y la tabla ENC nos da los valores actuales de los encoders del SCORBOT.

Estas tres variables tabla, debido a que son propias del sistema, no son tratadas de igual forma. La tabla ENC es de sólo lectura ya que no guarda los valores, sólo sirve para leer el modelo. Las tablas IN y OUT guardan realmente los valores, pero además cuando un valor de alguna de las tablas es cambiado también se modifica el estado en la ventana de Entradas y Salidas digitales (apartado 4.4). Además el valor de alguna posición de estas tablas no puede cambiar si antes se ha ejecutado un comando DISABLE (ver apartado número 10.2.7) sobre alguna de esas posiciones.

Antes de ser creada una variable tipo tabla se comprueba si existe una variable tabla (es decir su objeto *Tabla* correspondiente) con iguales atributos *nombre* y *padre* que la que se quiere crear. Si no existe se comprueba además si existe alguna variable simple (es decir su objeto *Simple* correspondiente) de igual *nombre* y *padre*. Y si tampoco existe entonces se puede crear la variable tabla.

También se realiza una comprobación del campo *nombre* de la variable. Sólo serán aceptados nombres válidos para las nuevas variables.

Un nombre para una variable se considera inválido cuando se cumple alguna de estas condiciones:

- Es una palabra reservada. Los comandos y los operadores son considerados palabras reservadas.
- El primer carácter del nombre de la variable es un número.
- Contiene algún carácter no válido. Caracteres válidos son las letras (A-Z) y los números (0-9), todos los demás son considerados inválidos. Además no se distingue entre mayúsculas y minúsculas.

8.2 Posiciones

Para la gestión de las posiciones en el simulador se han creado dos tipos de objetos: el objeto tipo *Posicion* y el objeto tipo *TPosicion*.

El objeto *Posicion* guardará los atributos propios de una posición. Puesto que las posiciones pueden actuar tanto en coordenadas articulares como cartesianas, poseen ambos atributos. Además también tienen los atributos *nombre*, *tipo* y *relativa*.

El objeto *TPosicion* será el encargado de simular los vectores de posiciones del lenguaje ACL.

El sistema que se usará será convertir siempre las posiciones, sea cual sea su tipo de coordenadas, a posiciones en coordenadas articulares.

Tanto para las posiciones como para los vectores de posiciones no existe el concepto de visibilidad ya que son tratadas siempre como globales. De esta manera, ningún tipo de dato posición posee el atributo *padre*, que poseían los objetos *Simple* y *Tabla*.

8.2.1 Posición

Una posición es un tipo de dato estructurado. Sus campos guardan dos tipos de coordenadas.

El primer tipo de coordenadas son las correspondientes a los valores que tomarían al alcanzar una posición cada uno de las encoders de las articulaciones del robot SCORBOT. Estos valores formarán las coordenadas en el espacio articular y son guardados en el campo *joints[]*. Por otra parte estos valores definen una posición en el espacio cartesiano que serán el segundo tipo de coordenadas. Para guardar esta posición en coordenadas cartesianas existen 5 campos más dados por los campos *xyz[]* y *pr[]*.

Por tanto el objeto *Posicion* debe dar acceso a cada uno de estos campos y ,además del *nombre* y el *tipo*, debe indicarnos si la posición es absoluta o relativa a otra. Si fuese

relativa debe darnos la posibilidad de saber a qué posición es relativa.

```
package SCORBOTSimulator.Tipos;

public class Posicion {

  private String nombre;
  private int[] xyz=new int[3];
  private int[] pr=new int[2];
  private int[] joints=new int[Const.NUMJOINTS];
  private int status=Const.ST_PABS;
  private String relativa;

  public Posicion(String nombre)
  public Posicion(String nombre,String relativa)
  public void setStatus(int status)
  public void setRelativa(String relativa)
  public void setAbsoluta()
  public synchronized void setXYZ(int x,int y,int z)
  public synchronized void setXYZ(int[] nxyz)
  public synchronized void setX(int x)
  public synchronized void setY(int y)
  public synchronized void setZ(int z)
  public synchronized void setPitch(int p)
  public synchronized void setRoll(int r)
  public synchronized void setPR(int p,int r)
  public synchronized void setPR(int[] npr)
  public synchronized void setJoint(int nj,int val) throws Exception

  public synchronized void setJoints(int[] jointsval) throws Exception
  public String getNombre()
  public int getStatus()
  public String getRelativa()
  public boolean isRelativa()
  public boolean isAbsoluta()
  public int[] getXYZ()
  public synchronized int getX()
  public synchronized int getY()
  public synchronized int getZ()
  public synchronized int getPitch()
  public synchronized int getRoll()
  public int[] getPR()
  public int[] getJoints()
  public synchronized int getJoint(int nj) throws Exception
  public String toString()

}
```

El acceso a las coordenadas será sincronizado siempre que se quiera acceder a los valores propios de una posición, es decir a las tablas *joints*, *xyz* y *pr*.

Los procedimientos que acceden a las coordenadas articulares lanzan excepciones del tipo *AxisInvalid* cuando se accede a un eje no válido. Este método será uno de los que permita la extensión de este simulador a otros tipos de

robots. Los ejes válidos vienen dados desde 1 hasta el número dado por la constante *Const.NUMJOINTS*.

Las coordenadas cartesianas serán X, Y, Z, P y R. Siendo P el pitch y R el roll medidos en grados.

Existen distintos tipos de posiciones definidos todos en el lenguaje ACL. Estos tipos son:

- posición no definida, *Const.ST_PUNDEF=0*;
- posición absoluta, *Const.ST_PABS=1*;
- posición relativa en coordenadas articulares a otra posición, *Const.ST_PREL_JOTRA=2*;
- posición relativa en coordenadas cartesianas a otra posición, *Const.ST_PREL_COTRA=3*;
- posición relativa en coordenadas articulares a la posición actual, *Const.ST_PREL_JACTU=12*;
- posición relativa en coordenadas cartesianas a la posición actual, *Const.ST_PREL_CACTU=13*.

El tipo de la posición será asignado en función del comando que se haya usado para definirla. El tipo es guardado en el campo *status*. Al crear una posición el tipo inicial será *Const.ST_PUNDEF* debido a que sus campos todavía no guardan valores correspondientes a una posición definida (en realidad sí, pero la posición es la dada por todos los campos a un valor nulo a la vez en coordenadas articulares y cartesianas, hecho que no es posible debido al modelo que liga ambos espacios). El atributo *status* es devuelto cuando se usa un comando del tipo SET <var>=PSTATUS <pos> en el lenguaje ACL.

Al igual que para los objetos *Simple* y *Tabla* existe una clase que manejará los objetos *Posicion*, ésta será la clase *Posiciones*.

La clase *Posiciones* se encargará de la creación, destrucción y acceso a los atributos de las posiciones creadas con el comando DEFP (ver apartado número 10.2.2).

```
package SCORBOTSimulator.Tipos;

public class Posiciones {

  private static Vector tablaPosiciones;

  public static void init()
  public static int size()
  public static Vector listAll()

  public static boolean contains(String nombrePosicion)
```

```
  public static synchronized void add(String nombrePosicion) throws
Exception
  public static synchronized void erase(String nombrePosicion) throws
Exception
  public static synchronized void setXYZ(String nombrePosicion, int x, int
y, int z) throws Exception
  public static synchronized void setPR(String nombrePosicion, int p, int
r) throws Exception
  public static synchronized void setJoints(String nombrePosicion, int[]
jointval) throws Exception
  public static synchronized void setJoint(String nombrePosicion, int nj,
int val) throws Exception
  public static synchronized void setParam(String nombrePosicion , String
param, int val) throws Exception
  public static synchronized void setAbsoluta(String nombrePosicion) throws
Exception
  public static synchronized void setRelativa(String nombrePosicion, String
relativaA) throws Exception
  public static synchronized void setSTATUS(String nombrePosicion, int
tipo) throws Exception

  public static synchronized int getParam(String nombrePosicion, String
param) throws Exception
  public static synchronized int[] getXYZ(String nombrePosicion) throws
Exception
  public static synchronized int[] getPR(String nombrePosicion) throws
Exception
  public static synchronized int[] getPRRelativa(String nombrePosicion)
throws Exception
  public static synchronized int[] getJoints(String nombrePosicion) throws
Exception
  public static synchronized boolean isRelativa(String nombrePosicion)
throws Exception
  public static boolean isAbsoluta(String nombrePosicion) throws Exception
  public static synchronized String getRelativa(String nombrePosicion)
throws Exception
  public static synchronized int[] getXYZRelativa(String nombrePosicion)
throws Exception
  public static synchronized int[] getJointsRelativa(String nombrePosicion)
throws Exception

  public static synchronized int getSTATUS(String nombrePosicion) throws
Exception

  public static synchronized void updateXYZPRfromJoints(String
nombrePosicion) throws Exception
  public static synchronized void updateJointsFromXYZPR(String
nombrePosicion) throws Exception
}
```

Todos estos procedimientos permiten el acceso a los atributos de cada una de las posiciones creadas durante la ejecución del simulador.

Merecen ser destacados los procedimientos *setParam()* y *getParam()*. Estos procedimientos permiten el acceso a uno sola

de las coordenadas, ya sean articulares o cartesianas, y modificarla.

Debido a que se hacen operaciones de índole articular y cartesiana sobre las posiciones, y también a que para la generación de movimientos sólo se tienen en cuenta las coordenadas articulares, fueron creados los procedimientos *updateXYZPRfromJoints()* y *updateJointsFromXYZPR()*. Estos procedimientos hacen uso de los modelos cinemático directo e inverso (ver apartados número 6.2 y 6.4) para generar el juego de coordenadas contrarias a las que se han modificado.

ejemplo ———————————————————————————

Creamos una posición POS1 con el comando DEFP POS1. Actualizamos su valor con la actual, HERE POS1. Ahora realizamos un desplazamiento en la coordenada y de 1000 sobre dicha posición, SHIFTC POS1 BY Y 1000.

El conjunto de acciones realizadas internamente son las siguientes, citadas en orden secuencial:

- *Posiciones.add("POS1")*
- Se copian todos los atributos de la posición ACTUAL a POS1. Realmente HERE <pos> equivale a hacer SETP <pos>=ACTUAL
- Se modifica la coordenada cartesiana de la siguiente manera: *setParam("POS1", "Y", getParam("POS1", "Y")+1000)*
- Y después se ejecuta el procedimiento *updateJointsFromXYZPR("POS1")*, que actualizará las coordenadas articulares a partir de la nueva posición en coordenadas cartesianas.

De igual modo se realizaría la operación si modificásemos una coordenada articular con el comando SHIFT.

———————————————————————————————————

Los procedimientos que efectúen cambios sobre alguno de los atributos de una posición lanzarán excepciones del tipo *PosicionNoExiste* y *ParametroNoExiste*, si la posición no existe o si la coordenada a la que se accede no existe respectivamente.

El procedimiento *init()* crea tres posiciones del sistema: HOMEPOS, ZEROPOS y ACTUAL. Sólo la posición HOMEPOS puede ser modificada. HOMEPOS es la posición a la que se va cuando se realiza un comando HOME.

ZEROPOS es una posición con todos sus atributos de coordenadas a cero. Sus atributos no se pueden modificar. Esta posición es usada a la hora de crear nuevas posiciones.

ACTUAL es la única que es tratada de un modo especial, no se guardan sus valores en el objeto *Posicion* sino que son leídos directamente del modelo. Por tanto, también es de sólo lectura.

8.2.2 Posiciones relativas

Las posiciones relativas son simuladas con el atributo *relativa* de un objeto *Posicion*. Este atributo hace referencia solamente al nombre de una posición absoluta, es decir una posición que tenga su atributo *status=Const.ST_PABS*.

Los atributos *relativa* y *tipo* de un objeto *Posicion* determinan el tipo de posición que estamos manejando.

Del manejo de las coordenadas relativas se encarga la clase que maneja los objetos *Posicion*, es decir la clase *Posiciones*.

Una posición relativa es tratada de igual manera que una absoluta, solamente cuando ha de ser usada para realizar un movimiento (cualquier comando derivado de MOVE) se usará el hecho de que es una posición relativa.

Cualquier procedimiento de acceso a los valores de una posición relativa nos dará como resultado el acceso directo a esos atributos de la posición. Es decir, si tenemos una posición POS1 definida 1000 unidades por encima de la actual, cualquier acceso, que no sea un comando MOVE, a POS1 nos dará como resultado que esa posición es una posición con X:0, Y:1000, Z:0, P:0 y R:0 (obviamos las coordenadas articulares).

Sólo cuando usemos los procedimiento *getXYZrelativa()* y *getJointsRelativa()*, serán calculadas las nuevas coordenadas sumándole las correspondientes a la posición actual.

8.2.3 Vectores de posiciones

Los vectores de posiciones son una tablas cuyos elementos son posiciones. De esta forma han sido simulados. Se creó una clase *TPosicion* que guardara en su interior N objetos del tipo *Posicion*. Cada uno de estos objetos puede ser tratado como si realmente se tratase de una posición.

Estos elementos *Posicion* sólo tienen en común el nombre, que es el mismo del vector de posición. Esto sólo ha sido una conveniencia, pues realmente no se puede acceder a ese atributo a través del lenguaje ACL.

Como en los otros casos, también se creó una clase que manejase los objetos *TPosicion*. Esta clase es *TPosiciones*.

```
public class TPosicion {

  private String nombre;
  private Posicion[] valores;

  public TPosicion(String nombre,int tam) throws Exception
  public String getNombre()
  public int getSize()
  public int[] getXYZ(int pos) throws Exception
  public int[] getPR(int pos) throws Exception
  public int getX(int pos) throws Exception
  public int getY(int pos) throws Exception
  public int getZ(int pos) throws Exception
  public int getPitch(int pos) throws Exception
  public int getRoll(int pos) throws Exception
  public int[] getJoints(int pos) throws Exception
  public int getJoint(int pos,int nj) throws Exception
  public String getRelativa(int pos) throws Exception
  public int getStatus(int pos) throws Exception
  public boolean isRelativa(int pos) throws Exception
  public boolean isAbsoluta(int pos) throws Exception
  public void setXYZ(int pos,int x,int y,int z) throws Exception
  public void setXYZ(int pos,int[] xyz) throws Exception
  public void setX(int pos,int x) throws Exception
  public void setY(int pos,int y) throws Exception
  public void setZ(int pos,int z) throws Exception
  public void setPitch(int pos,int p) throws Exception
  public void setRoll(int pos,int r) throws Exception
  public void setPR(int pos,int p,int r) throws Exception
  public void setJoint(int pos,int nj,int val) throws Exception
  public void setJoints(int pos,int[] jointsval) throws Exception
  public void setRelativa(int pos,String relativa) throws Exception
  public void setAbsoluta(int pos) throws Exception
  public void setStatus(int pos,int status) throws Exception
  public String toString()
}
```

Todos los procedimientos funcionan de la misma manera que los de la clase *Posicion*, sólo que ahora requieren también del número del elemento al que se quiere acceder. Este mismo hecho ocurría con los objetos *Simple* y los objetos *Tabla*, es decir el código que representa el funcionamiento de la tabla se complica al tener que indexar los elementos de dicha tabla.

Existe también un procedimiento para obtener el tamaño del vector de posiciones denominado *getSize()*, al igual que ocurría en el objeto *Tabla*.

El funcionamiento interno de estos vectores no es el más efectivo, pero sí el más simple. Lo otra opción que haría más rápido el tratamiento de vectores sería que el objeto *TPosicion* no fuese un simple acceso a cada uno de los objetos *Posicion* de la tabla creada, sino que ya se crearán las tablas con los atributos dentro de la misma clase.

Esta otra alternativa no fue escogida debido al hecho de que realmente no se usan con mucha frecuencia este tipo de datos. Para ser más exacto sólo es usado de una forma común cuando se usa el comando MOVES.

Para el manejo de los vectores de posición creados existe la clase *TPosiciones*. Esta clase como las otras del mismo tipo, hace las funciones de intermediario entre los comandos del lenguaje ACL y los objetos *TPosicion*.

La clase *TPosiciones* es muy similar a la clase *Posiciones*. La única diferencia apreciable es que los procedimientos requieren también del número del elemento del vector a tratar. Además añade otro procedimiento para obtener el tamaño del vector de posiciones denominado *getSize()*.

8.3 Parsers

Se han llamado genéricamente parsers a las clases que se encargan de extraer de una línea de texto los parámetros de los comandos, de obtener el resultado de operaciones matemáticas, etc. En definitiva, los parsers serán el núcleo del intérprete.

Han sido creados tres parsers distintos cada uno con una finalidad definida:

- ✓ *SimpleParser* se encarga de tratar una línea de texto y obtener el valor de la variable a la que se está haciendo referencia en esa línea.
- ✓ *CompParser* se encarga de obtener el resultado de operaciones lógicas o booleanas entre dos variables.
- ✓ *ParamParser* se encarga de separar una línea de texto en trozos que puedan ser interpretados por los procedimientos que simulan los comandos del lenguaje ACL.

Es necesario recalcar el hecho de que estas clases necesitan como parámetro de entrada una línea de texto. Una línea de texto, incluso teniendo el mismo significado, puede estar escrita de muchas maneras. El problema radica entonces en que los parsers puedan "entender" todas esas formas y dar siempre el mismo resultado. Para explicar este hecho mostraremos un ejemplo.

ejemplo

Ahora veremos distintas maneras de escribir una misma operación lógica entre dos variables:

"x[i]>y[j]", "x [i]>y [j]", "x [i] > y [j]", etc...

Todas estas formas de escribir la operación lógica tienen el mismo significado, podemos ir desde la más compacta hasta extendernos sobre una línea completa. La forma de escribir depende del usuario, por tanto no podemos obligar al usuario a que se adapte a la sintaxis del lenguaje, pero sí podemos adaptar la sintaxis al usuario dentro de ciertos límites.

8.3.1 SimpleParser

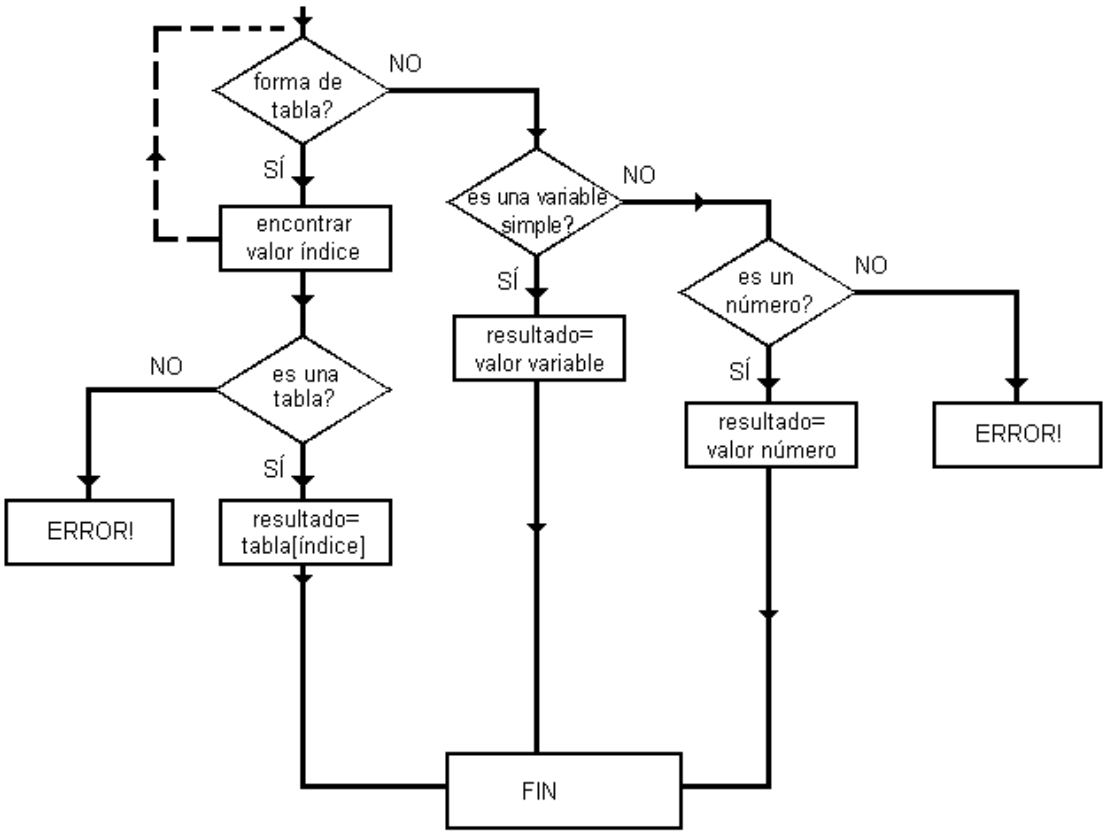
Esta clase se encarga de obtener el valor de la variable a la que se hace referencia en la línea de texto que recibe como

parámetro de entrada dentro de una sentencia ACL. Esta texto sólo puede hacer referencia a una variable o a una constante.

La variable puede ser una variable simple o una tabla. Hay que tener en cuenta que la tabla requiere un índice, y este índice puede ser otra variable o una constante.

También necesita como parámetro de entrada el nombre del programa que está consultando el valor de las variables, para que las clases más interiores (*Simples* y *Tablas*) se encarguen de gestionar la visibilidad.

El algoritmo que sigue es el siguiente:



La visibilidad de las variables es gestionada a la hora de verificar la existencia de una variable con ese nombre, ya que la existencia implica la igualdad con el nombre y la igualdad también con el padre de dicha variable. Si no se diese una de las dos condiciones se produciría un error al no encontrar la variable.

```
package SCORBOTSimulator.parser;

import SCORBOTSimulator.Tipos.*;
```

```
import SCORBOTSimulator.Errores;
import SCORBOTSimulator.CommonTools;

public class SimpleParser {

  public static int eval(String elemento,String parent) throws Exception {
    int res;
    String[] param=CommonTools.getNameAndIndex(elemento);

    if (param[0]==null) throw (Errores.EnteroNoValido);
    else {
      if (param[1]==null) { //es una variable simple o un entero
        try {
          res=Simples.get(param[0],parent);
        } catch (Exception err) { //no existe la variable
          try {
            res=Integer.parseInt(param[0]);
          } catch (NumberFormatException err2) {
            throw (Errores.EnteroNoValido);
          }
        }
      }
      else { //es una tabla,debemos iterar para calcular el indice
        res=Tablas.get(param[0],eval(param[1],parent),parent);
      }
    }

    return (res);
  }
}
```

La mayoría de las clases que se usan se han hecho estáticas. Esto fue decidido así para evitar la creación de objetos innecesarios y el consiguiente gasto de memoria. Además la creación y destrucción dinámica de objetos consume tiempo. El inconveniente es que este objeto estará siempre ocupando un trozo de memoria, por otro lado esta clase va a ser usada con mucha frecuencia, por lo que no compensa crear un objeto *SimpleParser* cada vez que queramos realizar una operación.

Este mismo razonamiento ha sido realizado con muchas de las clases usadas en el proyecto.

El manejo de variables y posiciones se apoya en el procedimiento *getNameAndIndex()* de la clase *CommonTools*. Este procedimiento devuelve una tabla con dos cadenas de texto. En la primera devuelve el nombre de la posible variable, y en la segunda el índice. Aunque realmente es muy simple, es a la vez muy potente ya que permite separar de una línea de texto las dos partes de una variable, verificar la existencia de corchetes y gestionar los errores.

8.3.2 CompParser

El funcionamiento de esta clase es bastante simple. En primer lugar busca el operador lógico, y una vez encontrado, separa en dos la línea de texto.

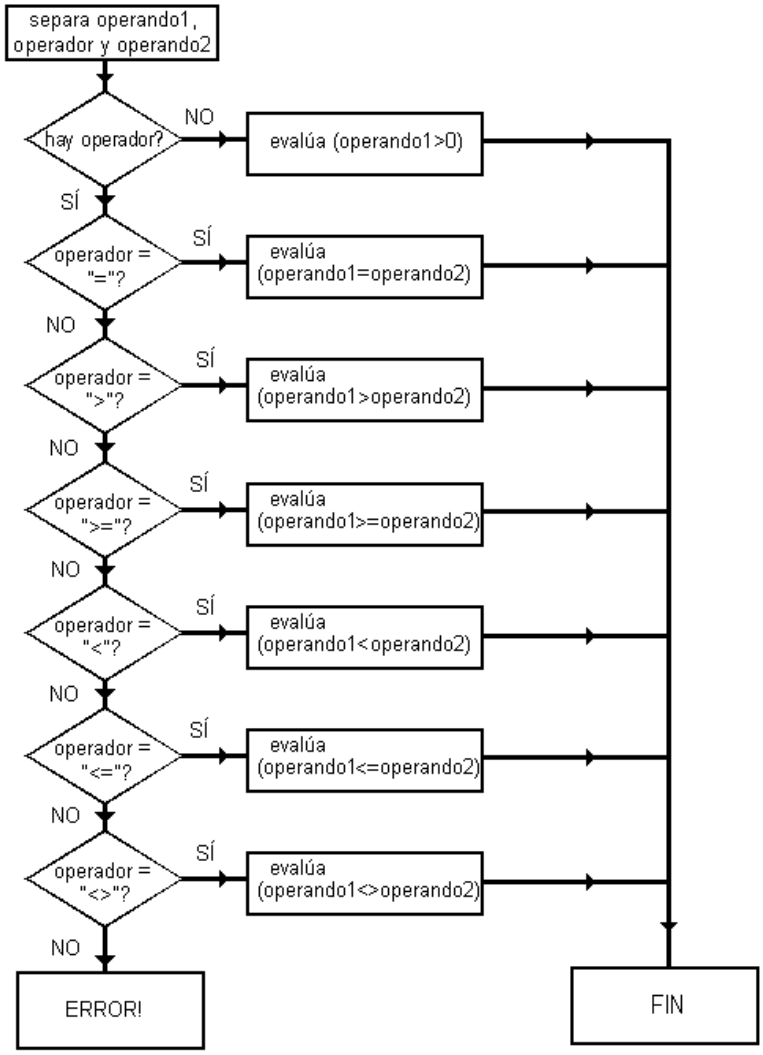
Estas dos partes deberán ser cada una variables o constantes, por tanto sólo hay que encontrar sus valores y realizar la operación lógica con esos dos variables (los operandos).

Puesto que el lenguaje ACL también permite operaciones de un solo operando, si no se encuentra ningún operando se realizará esa operación, es decir, si el valor es mayor que cero el resultado será 1 y en caso contrario 0.

Para encontrar el valor de las dos variables, los operandos, esta clase hace uso de la anterior *SimpleParser*.
También necesita como parámetro de entrada el nombre del programa que está consultando el valor de las variables, para que la clase *SimpleParser* se encargue de gestionar la visibilidad.

El diagrama de flujo que sigue este parser para encontrar el resultado de la condición introducida es el que se presenta en la página siguiente.

Este parser es solamente utilizado por las órdenes IF y WAIT (ver apartados número 10.2.5 y 10.2.4, respectivamente). Estas órdenes realizan una comparación y entregan un resultado lógico de esa comparación.



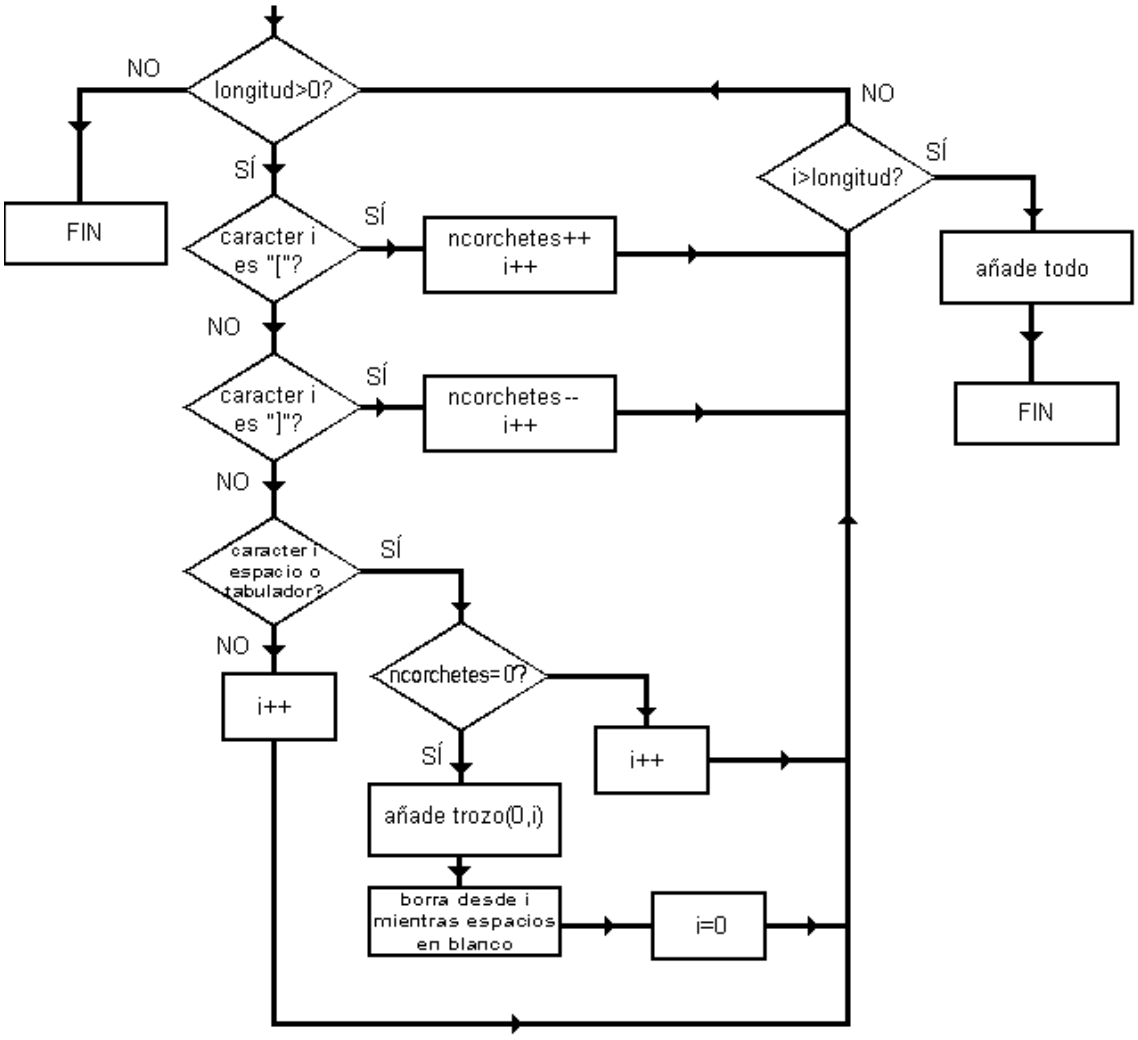
8.3.3 ParamParser

Se puede decir que esta es una de las clases más importantes en las que se apoya el intérprete de comandos del simulador.

Esta clase requiere como entrada una línea de texto, y como salida nos dará los trozos de los que está compuesta. La potencia de esta clase radica en el hecho de que permite al usuario escribir los comandos con cierta libertad.

La separación en partes de una línea de texto ha sido dividida en dos partes. En la primera parte son extraídas las partes separadas por espacios y las encerradas por corchetes. Para la búsqueda de corchetes se ha seguido la idea siguiente: se debe cerrar siempre lo que encontremos abierto.

La primera parte sigue el siguiente diagrama de flujo:



De esta forma conseguimos en una línea como la siguiente "SET ELEM[4]=ELEM[ELEM[3]]+50" esta separación de elementos (el método que utiliza el comando SET para evaluar el resultado de la expresión puede ser consultado en el apartado número 10.2.1):

- primer grupo: "ELEM","[4]"
- segundo grupo: "ELEM","[ELEM[3]]"
- tercer grupo: "50"

En la segunda parte del algoritmo de obtención de elementos de una línea, se realiza una búsqueda sobre la lista de grupos generada y se actúa de la forma siguiente: cuando nos encontremos un elemento encerrado por corchetes (que debemos

entender que será el índice de una tabla) debe unirse con el elemento que le antecede.

De esta forma los grupos del ejemplo quedan como siguen:

- primer grupo: "ELEM[4]"
- segundo grupo: "ELEM[ELEM[3]]"
- tercer grupo: "50"

Por tanto sólo resta que el comando SET encuentre el resultado de la suma del segundo grupo con el tercero y lo asigne al primero.

El código que nos sirve para conseguir dicho objetivo queda reflejado en el objeto estático *ParamParser*.

También existen los procedimientos *ToolJoin()* y *ToolJoinEx()* sirven para unir trozos de un vector de cadenas en una cadena. Fueron añadidos para prevenir el hecho de que se hubieran separado trozos de una línea de texto que no hacía falta separar.

ejemplo

En la línea "FOR I = 1 TO 20", la función *eval()* de *ParamParser* nos devolvería una variable *Vector* con los siguientes elementos del siguiente modo:

"FOR", "I", "=", "1", "TO", "20"

Esta separación es debida solamente al hecho de que el usuario lo escribió así. Si por el contrario hubiera escrito "FOR I=1 TO 20", el resultado hubiera sido:

"FOR", "I=1", "TO", "20"

La función que interpreta órdenes FOR requiere que la asignación inicial esté en una misma cadena.

Por tanto, usaremos la función *ToolJoinEx(resultado, 1, "TO")* que en cualquiera de los dos casos nos devolverá una cadena de la forma "I=1".

9 EL PROGRAMA

El programa será una clase que tras adecuar el listado de órdenes para ser interpretado, irá ejecutando una a una cada línea del programa. Además se tiene que encargar de realizar los saltos (condicionales e incondicionales) y esperas.

Debido a que el lenguaje ACL se construye sobre una máquina multitarea, cada uno de los objetos del tipo programa funcionará como un hilo de ejecución independiente del resto de hilos.

Los objetos programa son instancias de una clase llamada *ProgramThread*. Para el manejo de todo el conjunto de objetos *ProgramThread* existe otra clase llamada *ProgramGroups*. Los primeros se crean dinámicamente, mientras que el segundo será un objeto estático.

El manejo de errores se hace a través de las excepciones. Las excepciones son objetos que al ser lanzados (en Java se realiza con el comando *throw*) rompen la ejecución normal del programa. En general, las excepciones serán lanzadas cuando los procedimientos que ejecutan cada una de las acciones encuentren un error. Estas excepciones serán recogidas en el lugar desde donde fueron ejecutados los comandos, es decir la instancia del objeto *ProgramThread* que será un hilo.

9.1 El objeto ProgramThread

Este objeto se crea cada vez que se ejecuta un comando RUN <prog> del lenguaje ACL. El objeto necesita como entrada los siguientes parámetros: el texto y el nombre del programa, y tres botones que servirán para controlar su ejecución.

El nombre del programa será el que nos servirá para que este programa pueda ir creando sus variables locales y poder comprobar la visibilidad al acceder a las mismas u otras. También este nombre nos servirá para que otros programas puedan controlar su ejecución a través de los comandos SUSPEND, CONTINUE, GOSUB, STOP...

Antes de empezar la ejecución es necesario realizar una serie de comprobaciones y arreglos en el texto del listado del programa. Estas comprobaciones son (en orden secuencial):

- ✓ deben quitarse los posibles comentarios.
- ✓ contar el número de comandos IF que debe coincidir con el número de comandos ENDIF.
- ✓ realizar la misma cuenta con los comandos FOR y ENDFOR.

- ✓ modificar algunos comandos para que puedan ser tratados de forma correcta.

Los primeros tres puntos son simples comprobaciones de la integridad o consistencia del programa. Se realiza el chequeo antes de empezar a ejecutar para evitar crear objetos innecesariamente. También intenta emular de alguna manera al lenguaje ACL, éste realiza la comprobación al escribir el comando END en el modo editor.

Si al realizar la cuenta anterior existiese alguna disparidad en el número de comandos, el programa no llegaría a ejecutarse y se daría un error del tipo "programa no consistente" en la pantalla del usuario (ver apartado nº 4.3).

El más interesante es el último punto. En éste se cambian algunas órdenes para que pueda ser simulado el comportamiento real de esas órdenes correctamente.

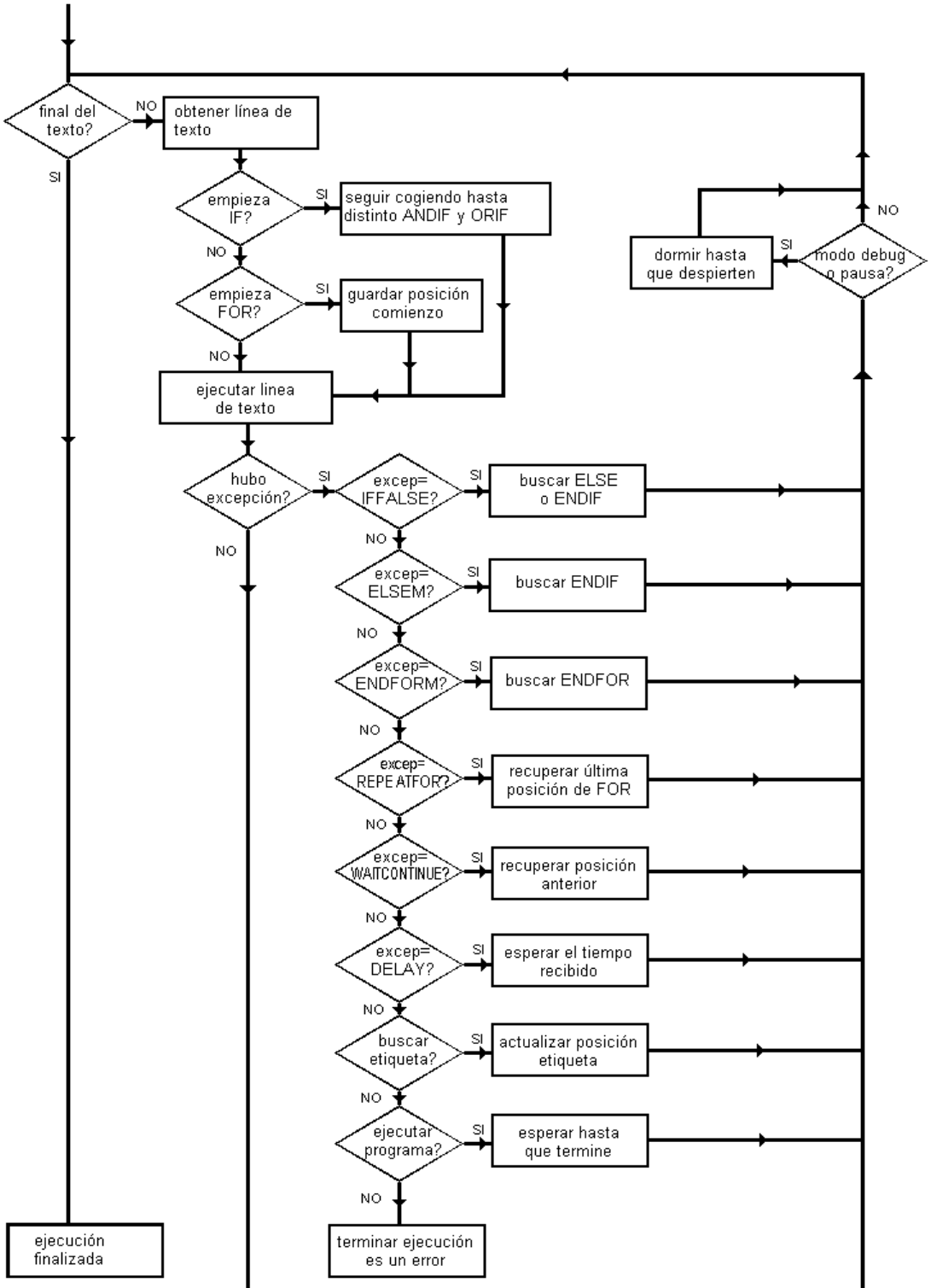
Las órdenes que deben ser cambiadas son las siguientes: todos los comandos de movimiento terminados en 'D' (MOVED, MOVESD, MOVELD, etc., aquellos que detienen la ejecución del programa hasta que concluye la ejecución del movimiento), y los bucles del tipo for.

Para los comandos MOVExxD se añade una línea de texto con la sentencia "WAIT MOVING=0". Este comando será el que realice la espera activa hasta la finalización del movimiento. Debemos recordar que la variable MOVING no existe en el lenguaje ACL como tal, por tanto debe evitarse su uso en la medida de lo posible.

El funcionamiento de los bucles for es el siguiente: cuando es encontrada una instrucción FOR se guarda la posición de esa línea en una pila. Esta posición nos servirá para volver cuando encontremos el comando ENDFOR correspondiente.

La instrucción "FOR <var>=<var1> TO <var2>" realizará el bucle sobre la variable <var> desde <var>=<var1> hasta <var2>. Por tanto, debemos saber cuando es la primera vez que se entra en el bucle para inicializar la variable al valor dado por <var1>. De este modo, y suponiendo que <var1> es menor que <var2> (sólo para continuar de forma simple la explicación), se añade antes de la instrucción FOR la línea "SET <var>=<var1>-1". Entonces, el comando FOR sólo debe incrementar en una unidad la variable y realizar la comprobación. Si <var> es mayor que <var2> dejamos de iterar y buscamos ENDFOR. Del mismo modo se actuaría si <var1> fuese mayor que <var2>.

El diagrama de interpretación del programa es el siguiente:


final del texto?
NO
obtener línea de texto
SI
empieza IF?
SI
seguir cogiendo hasta distinto ANDIF y ORIF
NO
empieza FOR?
SI
guardar posición comienzo
NO
ejecutar linea de texto
hubo excepción?
SI
excep= IFFALSE?
SI
buscar ELSE o ENDIF
NO
excep= ELSEM?
SI
buscar ENDIF
NO
excep= ENDFORM?
SI
buscar ENDFOR
NO
excep= REPEATFOR?
SI
recuperar última posición de FOR
NO
excep= WAITCONTINUE?
SI
recuperar posición anterior
NO
excep= DELAY?
SI
esperar el tiempo recibido
NO
buscar etiqueta?
SI
actualizar posición etiqueta
NO
ejecutar programa?
SI
esperar hasta que termine
NO
terminar ejecución es un error
NO
modo debug o pausa?
SI
dormir hasta que despierten
ejecución finalizada

Para simplificar el anterior diagrama no ha sido incluido el lanzamiento de excepciones en las búsquedas de etiquetas y comandos ENDIF, ELSE y ENDFOR (ver apartado número 9.1.1.4). Si no se encontrase la etiqueta o el comando correspondiente se lanzaría una excepción de error y se daría por terminada la ejecución del programa.

En el diagrama se incluyen las palabras dormir y despertar. Estos verbos hacen referencia a los métodos *wait()* y *notify()* de Java para la sincronización de threads. Estos métodos hacen que el thread que se encarga de interpretar nuestro programa (la instancia del objeto *ProgramThread*) pase del estado "en ejecución" al estado "bloqueado". En el estado "bloqueado" el thread no consume tiempo de ejecución y por tanto permite que otros threads pasen a ejecución.

Tampoco ha sido incluido la condición de salida por interrupción del programa. Nuestro programa debe interrumpirse cuando se ejecute un comando ABORT desde modo directo o STOP desde modo editor. La simulación de estos comandos ha sido realizada con fidelidad, ya que estos comandos interrumpen la ejecución cuando finalice la ejecución del comando activo en ese momento. Por esta razón, el programa además de comprobar que todavía exista texto por ejecutar, también comprueba que no ha habido una interrupción del programa. Esta es la única manera de realmente terminar la ejecución del último comando.

El comando DELAY, como veremos más adelante en el apartado número 10.2.4, lanza siempre una excepción unas veces de error y otras de gestión. Si es de gestión es tratada por el hilo que hizo la llamada al comando DELAY. Esta excepción ("*$DELAY-NNNNN*", donde NNNN es el tiempo de espera) provoca un comando *sleep(NNNN)* en Java, el cual también cambia el estado de ejecución del thread a "bloqueado".

En este punto debemos recordar que se están manejando dos tipos de excepciones: los errores y las de gestión. Debido a que son instancias de objetos del mismo tipo, debemos realizar la comprobación del tipo de la excepción. De este modo se comprueban antes las de gestión, las más frecuentes y las menos numerosas, y si no concuerda con ninguna de las esperadas se tratará como un error.

Los errores, como muestra el diagrama, terminan la ejecución del programa. Para la gestión del error se toma la excepción con su mensaje de error, generada normalmente por alguno de los comandos, y se le suma el texto de la línea que

provocó el error para clarificarlo. Todo este texto es mostrado en la pantalla del usuario.

9.1.1 La implementación del objeto ProgramThread

Ha sido incluido aquí el código de éste objeto debido a que es el pilar en el que se apoya el simulador en la parte que trata de la ejecución de programas. Este es el código que hace que se ejecuten los programas, que no son simplemente secuencias de órdenes. Este código además permite que se realicen saltos condicionales e incondicionales, bucles y esperas activas.

Para la creación de objetos *ProgramThread* necesitamos los siguientes parámetros de entrada: el texto del programa, un nombre para el programa, un indicador de modo *debug* y los botones con los que controlar la ejecución del programa.

```
public class ProgramThread extends Thread {

  private String nombre;      //nombre del programa
  private String texto;       //listado del programa
  private int offset,lastoffset;    //indices para contador de programa
  private Stack offsetStack; //pilas para anidamientos
  private Stack forStack;
  private boolean interrumpido=false;     //variables para ejecución
  private boolean huboerror=false;
  private boolean debug=false;
  private boolean pausa=false;
  private boolean acabado=false;
  private BotonPlay botonplay;      //referencias a los botones
  private BotonPause botonpause;
  private BotonStop botonstop;
  private JToolBar padreBotones;
  private int prioridad=5;    //prioridad inicial

  //estados
  private static final int DELAY=0;
  private static final int PEND=1;
  private static final int SUSPENDED=2;
  private static final int RUNNING=3;
  private int estado=SUSPENDED;     //estado actual

  public ProgramThread(String nombre,String listado,boolean debug,BotonPlay
bplay,BotonPause bpause,BotonStop bstop) {
    this.setName(nombre.toUpperCase());
    this.nombre=nombre.toUpperCase();
    texto=listado;
    //adecuacion del listado del programa
    texto=ProgramUtils.quitaComentarios(texto);
    adecuaOrdenes();
    compruebaConsistencia();
```

```
    //para moverse por el programa
    offset=-1;
    lastoffset=offset;
    offsetStack=new Stack();
    forStack=new Stack();
    //control del flujo del programa
    this.debug=debug;
    acabado=false;

    //botones de control
    botonplay=bplay; botonpause=bpause; botonstop=bstop;
    padreBotones=(JToolBar) bplay.getParent();
    //programa añadido al gestor
    setPriority(Thread.NORM_PRIORITY);
    ProgramGroups.addThread(this);
  }

  public String getNombre() {return (this.nombre);}
  public int getPrioridad() {return (prioridad);}
  public void setPrioridad(int p) {prioridad=p;}
  ...
```

Durante la ejecución del programa éste va modificando su estado de ejecución. Estos estados están definidos en el manual del lenguaje ACL y son consultados por un comando llamado STAT. Este comando presentará por pantalla una lista de los programas en ejecución junto con su estado (ver apartado número 9.2). Es por esto que se añadió una variable local al programa llamada *estado*, la cual se va actualizando en consonancia con los estados que sean alcanzados. El comando STAT realizará una llamada al procedimiento *getEstado()* de cada *ProgramThread* para consultar su estado.

```
  ...
  public String getEstado() {
    switch (estado) {
      case DELAY: return ("DELAY");
      case PEND: return ("PEND");
      case SUSPENDED: return ("SUSPENDED");
      case RUNNING: return ("RUNNING");
      default: return ("DONT KNOW");
    }
  }
```

Las funciones que siguen son las que son llamadas por los comandos STOP, ABORT, SUSPEND y CONTINUE. STOP y ABORT sólo cambian el valor de la variable lógica *interrumpido* para que al finalizar la ejecución del comando actual en el hilo se finalice la ejecución del programa. Los comandos SUSPEND y CONTINUE hacen llamadas a los comandos *wait()* y *notify()* del hilo en ejecución.

```
public void interrumpe() {interrumpido=true;}
public boolean huboError() {return (huboerror);}

public synchronized void espera() {
  try {wait();}
  catch (InterruptedException e) {}
}

public synchronized void despierta() {
  notify();
}

public void parar() {
  pausa=true;
}
...
```

9.1.1.1 Eliminación de comentarios, comprobación de consistencia y adecuación de órdenes.

Antes de empezar la ejecución de un programa se realizan una serie de comprobaciones y cambios en el listado para poder ejecutarlo sin fallos y con la mayor eficacia posible.

Previa a las comprobaciones se deben retirar los comentarios del texto. Esto es debido a que los comentarios no son interpretados sino que previamente se eliminan para hacer más rápida la interpretación de las líneas de texto que contienen comandos del lenguaje ACL.

La función *quitaComentarios()* da como resultado un programa equivalente al que se le dio como parámetro de entrada, en cuanto a secuencia de comandos ACL. La única diferencia que existe entre ellos es que el programa de salida no tiene comentarios.

```
public static String quitaComentarios(String texto) {
  int remindex=texto.indexOf("//");
  int nextbr;

  while (remindex>=0) {
    nextbr=texto.indexOf('\n',remindex);
    if (nextbr!=-1)
      texto=texto.substring(0, remindex) + texto.substring(nextbr);
    else
      texto=texto.substring(0,remindex);

    remindex=texto.indexOf("//");
  }

  return (texto);
}
```

La primera comprobación tras eliminar los comentarios es examinar la consistencia del programa. Por consistencia se entiende la bondad de la estructura del programa. Un programa puede ser inconsistente por alguno de los siguientes fallos (o por una mezcla de ellos):

- Sentencias IF no terminadas con ENDIF
- Bucles FOR no terminados con ENDFOR
- Sentencias IF...ELSE...ENDIF mal construidas.

La comprobación que se realiza es bastante simple. Se realiza una pasada sobre todas las líneas del programa con dos índices que nos servirán para contar el número de sentencias IF (índice *nifs*) y el número de bucles FOR (índice *nfors*), inicializados a cero. Cuando nos encontremos un comando IF sólo debemos incrementar *nifs*, y cuando nos encontremos el esperado ENDIF debemos disminuirlo en una unidad también. Por tanto, al finalizar un programa consistente se debe dar la condición *nifs* igual a cero. El mismo proceso se repite con los bucles FOR y con el índice *nfors*.

El método utilizado para las sentencias IF...ELSE...ENDIF es distinto, ya que el comando ELSE no es necesario que aparezca. Para comprobar la consistencia de este tipo de sentencias sólo debemos comprobar si *nifs* es mayor que cero al encontrar un comando ELSE, esto nos dirá si nos encontramos dentro de una sentencia IF.

```
...
private void compruebaConsistencia() {
  offset=-1;
  int nifs=0;
  int nfors=0;
  String word;

  do {
    word=getFirstWord(getNextLine());
    if (word.equals("IF")) nifs++;
    else if (word.equals("ELSE")) {
      if (nifs==0) {nifs++; break;}
    }
    else if (word.equals("ENDIF")) nifs--;
    else if (word.equals("FOR")) nfors++;
    else if (word.equals("ENDFOR")) nfors--;

  } while (offset!=-1 && nfors>=0 && nifs>=0);

  if (nfors!=0 || nifs!=0) huboerror=true;

}
...
```

Una vez realizada la comprobación de la consistencia del listado del programa se deben realizar algunos cambios sobre dicho listado.

El primer cambio que se realiza es el siguiente: por cada comando de movimiento que implique parar la ejecución del programa hasta que se complete el movimiento, debemos añadir una línea que realice una espera activa para que la ejecución se detenga.

Dichos comandos de movimiento son: MOVED, MOVELD, MOVESD y HOME. El hecho de que se deba insertar la nueva línea es debido a que la espera activa no se realiza en el comando como lo realiza la máquina real, sino que la implementación de tales comandos ha hecho que se espere en el siguiente comando. El resultado final es el mismo y además esta modificación no tiene lugar sobre el texto original (sobre la ventana principal del simulador) del programa, es decir, el usuario del simulador nunca se entera de dicha modificación.

```
  ...
  private void adecuaOrdenes() {
    int lastoffset;
    offset=-1;
    String comando;
    String linea;
    do {
      lastoffset=offset;
      linea=getNextLine();
      comando=getFirstWord(linea); //cogemos la línea
      if (comando.equals("MOVED") || comando.equals("MOVESD") ||
        comando.equals("HOME") || comando.equals("MOVELD")) {
        texto=texto.substring(0,offset+1) + "WAIT MOVING=0\n" +
texto.substring(offset+1);
      }
      else if (comando.equals("FOR")) {
        String resultado=ProgramUtils.tratarFOR(linea);
        if (resultado==null) huboerror=true;
        texto=texto.substring(0,lastoffset+1) + resultado +
texto.substring(lastoffset+1);
        offset=lastoffset;
        for (int i=0;i<6;i++)
          getNextLine(); //para saltarnos las insertadas
      }
    } while (offset!=-1);
  }
  ...
```

También son modificadas los bucles FOR para que puedan ser usados normalmente. Estos bucles FOR...ENDFOR junto con las sentencias IF...ELSE...ENDIF serán objeto de un estudio posterior.

9.1.1.2 El procedimiento run()

Este procedimiento es el primero que se llama una vez el hilo pase a estado de ejecución. Es el encargado de poner en marcha el proceso de interpretación de líneas y de gestionar los errores al principio y al final de la ejecución.

Los errores al principio son marcados con un cambio en la variable *huboerror*. Esta variable tomará un valor lógico afirmativo si se produjo algún error de consistencia en el texto.

La interpretación del programa comienza con la llamada al procedimiento *runProgram()*. Tras la finalización de este procedimiento pueden darse dos tipos de resultados: o una excepción o nada. En este punto siempre que se produzca una excepción se tratará de un error del programa ACL, y por tanto sólo debemos recoger el mensaje que porta dicha excepción e imprimirlo en la pantalla del usuario para que sea corregido. En cualquier caso, al finalizar deben vaciarse las pilas para que no ocupen memoria.

```
...
public void run() {

  try {
    if (!huboerror) {
      actualizaBotones(false,true,true);    //actualizamos los botones
      estado=RUNNING;
      runProgram(); // y ejecutamos el programa
    } else {
      throw (Errores.ProgramNoValido);
    }
  }
  catch (Exception err) {
    huboerror=true;
    StringBuffer q=new StringBuffer();
    q.append("\n****************************** error\n");
    q.append(err.getMessage());
    MainFrame_TextOUT.printOutLn(q.toString());
    if (Const.DEBUG) err.printStackTrace();
  }
  finally {
    //vaciamos las pilas
    while (!offsetStack.empty()) offsetStack.pop();
    while (!forStack.empty()) forStack.pop();
    offsetStack=null; //ayuda al GC
    forStack=null;
  }

  if (interrumpido)
    MainFrame_TextOUT.printOutLn("programa "+nombre+" interrumpido");

  //eliminamos las variables locales
  if (Configuracion.isDestroyLocalOn())
    ProgramUtils.EliminaVariablesLocales(nombre);
```

```
    //borramos los botones
    padreBotones.remove(botonplay);
    padreBotones.remove(botonpause);
    padreBotones.remove(botonstop);

    try {
      ProgramGroups.deleteThread(nombre);
    }
    catch (Exception err) {
      System.out.println(nombre+" no se ha podido eliminar a si mismo");
      System.exit(-1);
    }

    try {
      this.finalize();
    }
    catch (Throwable err) {
      System.out.println("error de tipo throwable no esperado");
    }
  }
  ...
```

Aquí debemos citar las dos funciones que nos ayudan a obtener la línea de texto siguiente en el programa (que será la encargada de actualizar el contador de programa) , y la que nos ayuda a identificar algunos comandos tratados de forma especial.

```
  ...
  private String getNextLine() {
    int last=offset+1;
    offset=texto.indexOf('\n',last);
    if (offset==-1) {
      acabado=true;
      return (texto.substring(last));
    }
    else return (texto.substring(last,offset));
  }
```

La función *getFirstWord()* tomará como entrada la línea obtenida desde la función *getNextLine()* y devolverá la primera palabra que encuentre en dicha línea. Para encontrar la palabra debe buscar dónde encuentra el primer carácter en blanco, es decir un espacio o un tabulador. También existen sentencias ACL compuestas por un sólo comando, por tanto también se tiene en cuenta este caso.

```
  private String getFirstWord(String line) {
    line=line.trim();
    int espacio=line.indexOf(' '); //para buscar espacio primero
    if (espacio==-1)
```

```
      espacio=line.indexOf('\t'); //y después tabulador

    //si no hay nada es que es una sola palabra puede que vacia
    if (espacio==-1)
      return (line.toUpperCase());
    else
      return (line.substring(0,espacio).toUpperCase());
  }
  ...
```

9.1.1.3 El procedimiento runProgram()

Este procedimiento es la pieza fundamental del programa, se encargará de mandar al procedimiento de interpretación cada línea del texto del programa y posteriormente gestionará los posibles errores y mensajes que provengan de dicha interpretación.

La primera parte se encarga de obtener la línea de texto y de realizar unas comprobaciones. Dichas comprobaciones consisten en verificar qué comando se encuentra en la línea para ver si es un comando del tipo IF o del tipo FOR, ya que estos comandos son tratados de forma especial.

Con los comandos IF debemos actuar de la siguiente manera: mirar las siguientes líneas del programa para extraer las posibles condiciones ANDIF y ORIF posteriores. Esto es debido a que el comando IF es tratado en su conjunto y no línea a línea. Para más detalles ver el comando IF en el apartado 10.2.5.

Por otro lado, cada vez que nos encontremos con una sentencia del tipo FOR debemos guardar la posición del contador de programa en la pila dedicada a los bucles FOR. Esto es debido a que el comando ENDFOR hará uso de esta pila para realizar el bucle.

```
  ...
  private void runProgram() throws Exception {
    String comando;
    boolean salto=false;
    boolean forc=false;
    String JumpLabel=null;
    String mensaje;

    do {
      lastoffset=offset; //el principio de la línea
      comando=getNextLine().trim(); //cogemos la línea
```

```
//comprobación del comando que va a ser ejecutado
if (getFirstWord(comando).equals("FOR"))
  forStack.push(new Integer(lastoffset));
else if (getFirstWord(comando).equals("IF")) { //mirar ANDIF y ORIF
  String aux;
  lastoffset=offset;
  aux=getNextLine().trim().toUpperCase();
  while (getFirstWord(aux).equals("ANDIF") ||
   getFirstWord(aux).equals("ORIF")) {
    comando=comando+"$"+aux;
    lastoffset=offset;
    aux=getNextLine().trim().toUpperCase();
  };
  offset=lastoffset; //para dejarlo antes
}
else if (comando.toUpperCase().equals("WAIT MOVING=0")) estado=PEND;
else if (getFirstWord(comando).equals("WAIT")) estado=DELAY;
else estado=RUNNING;

//ejecutamos el comando
try {
  Command.doAction(comando,nombre,Const.EDITOR);
}
catch (Exception exc) {
  mensaje=exc.getMessage();
  //distintos mensajes que nos pueden venir
  if (exc==Errores.IFFALSE) SearchELSEorENDIF(); //if no cumplido
  else if (exc==Errores.ELSEM) SearchENDIF();
  else if (exc==Errores.ENDFORM) {
    forStack.pop();
    SearchENDFOR();
  }
  else if (exc==Errores.REPEATFOR) forc=true;
  else if (exc==Errores.WAITCONTINUE) {
    //sirve para repetir la última instrucción
    offset=lastoffset; //repetimos el wait
    try { sleep(50);}
    catch (InterruptedException err) {}
    yield(); //dejamos a los demás participar
  }
  else if (mensaje.startsWith("$DELAY-")) {
    estado=DELAY;
    String valor=mensaje.substring(mensaje.indexOf('-')+1);
    int espera=Integer.parseInt(valor);
    try {sleep(espera);}
    catch (InterruptedException err) {}
    estado=RUNNING;
  }
  else if (mensaje.charAt(0)==':') { //salto incondicional
    salto=true; JumpLabel=mensaje.substring(1);
  }
  else if (mensaje.charAt(0)=='@') { //esperar hasta que termine
    String name=mensaje.substring(1);
    do {
      sleep(100);
    } while (!ProgramGroups.hasFinishedThread(name));
  }
  else {
```

```
          throw new Exception("en línea: ->"+comando+"<-"+"\n"+mensaje);
        }
      }

      if (forc) {
        if (forStack.empty()) throw (Errores.ENDFORnoFOR);
        offset=((Integer) forStack.pop()).intValue();
        forc=false;
      }
      else if (salto) {
        SearchLABEL(Integer.parseInt(JumpLabel));
        salto=false;
      }

      //si estamos en modo debug debemos esperar a que nos despierten
      //para continuar, la pausa sólo se hace una vez, el debug siempre
      if (debug||pausa) {
        //actualizamos los botones
        actualizaBotones(true,false,true);
        estado=SUSPENDED;
        espera();
        pausa=false;
        actualizaBotones(false,true,true);
        estado=RUNNING;
      }

    } while (!acabado && !interrumpido);

    //la condición de salida es que termine de leer el texto
    //o que el programa sea interrumpido con interrumpe();
  }

  private void actualizaBotones(boolean b1,boolean b2,boolean b3) {
    botonplay.setEnabled(b1);
    botonpause.setEnabled(b2);
    botonstop.setEnabled(b3);
    padreBotones.repaint();
  }
  ...
```

El diagrama de flujo que sigue el procedimiento *runProgram()* es el que viene dado por el gráfico presentado al principio del apartado 9.1.

Al finalizar la ejecución de una línea de comando pueden darse dos tipos de resultados: una excepción o nada. Las excepciones las hemos tratado de dos maneras distintas: como mensajes o como errores. Las excepciones del tipo error son lanzadas a un nivel superior para que sean presentadas por pantalla, éstas son recogidas por el procedimiento *run()*.

Por otra parte, las excepciones del tipo mensaje que podemos obtener al finalizar la interpretación de un comando son los siguientes:

- ✓ *"IFFALSE"*: este tipo de mensajes provienen de un comando IF. Nos indica que la condición incluida en la sentencia IF, y posibles ANDIF y ORIF, no se ha cumplido. De esta manera no se deben ejecutar los comandos interiores de la sentencia y se debe buscar el final del grupo interior. El final puede ser de dos tipos ya que podemos encontrarnos con sentencias IF con ELSE y sin ELSE. Por ello debemos actualizar el contador de programa con la posición del primer comando ELSE o ENDIF que nos encontremos, y que pertenezca a esta sentencia IF. Para más detalles consultar el apartado 9.1.1.4.

- ✓ *"ELSEM"*: este mensaje es enviado siempre y en cualquier condición por el comando ELSE. Indica que ya se ha finalizado la ejecución del primer conjunto de sentencias definidas por la estructura IF...ELSE...ENDIF y que debe saltarse el segundo conjunto. Para ello debe buscarse el final de la estructura IF, marcado por el comando ENDIF.

- ✓ *"ENDFORM"* es enviado por el comando FOR y le indica al programa que ya ha finalizado la iteración dentro del bucle FOR. Por tanto, se debe actualizar la posición marcada por el contador de programa con la posición siguiente al comando ENDFOR que corresponda.

- ✓ *"REPEATFOR"* es enviado por el comando ENDFOR bajo cualquier condición. De esta forma le indica al programa que debe recuperar la posición del principio del bucle FOR de la pila *forstack* correspondiente para volver a iterar.

- ✓ *"WAITCONTINUE"* es un mensaje enviado por el comando WAIT en caso de que la condición que le acompaña no se cumpla. Con este mensaje le indica al programa que debe volver a comprobar la condición y que por tanto no debe actualizar el contador de programa.

- ✓ *"$DELAY-nnnn"* le indica al hilo del programa que debe ejecutar un comando *delay(nnnn)* para retrasar la ejecución de la siguiente instrucción. Este mensaje

sólo es enviado por el comando DELAY (ver apartado número 10.2.4).

- ✓ *":nnnn"* es enviado por el comando GOTO (ver apartado 10.2.5) para indicar al programa que debe lanzar una búsqueda de la etiqueta dada por el número *nnnn*.

- ✓ *"@nombreprog"* es enviado únicamente por el comando GOSUB (ver apartado 10.2.5) y le indica al programa que debe lanzar la ejecución de un programa llamado *nombreprog* y esperar hasta que finalice su ejecución.

9.1.1.4 Búsqueda de etiquetas y actualización del contador de programa

Aquí se engloban una serie de procedimientos que permiten modificar el contador de programa y actualizarlo con la posición de alguna etiqueta o comando. Estas actualizaciones nos servirán para los siguientes casos:

- Ejecución de sentencias IF...ELSE...ENDIF
- Bucles FOR
- Saltos con GOTO

Para el caso de sentencias del tipo IF...ELSE...ENDIF podemos definir dos conjuntos de sentencias. Un primer conjunto que será ejecutado en el caso de que la condición sea cumplida, y un segundo conjunto que será ejecutado en caso contrario.

Cuando la condición no se cumple debemos ejecutar el segundo conjunto. Este conjunto debe ser buscado a partir del comando ELSE. Sin embargo debemos tener en cuenta el hecho de que el lenguaje ACL es interpretado por el simulador, esto quiere decir que cuando ejecutamos una sentencia del tipo IF no sabemos si existe un segundo conjunto de sentencias para ejecutar. Esto quiere decir que en el caso de que no se cumpla la condición debemos buscar o el comando ELSE, el cual nos daría el principio del segundo conjunto de sentencias, o el comando ENDIF, el cual nos marcaría el final de la sentencia IF.

Por tanto, el procedimiento *SearchELSEorENDIF()* se ejecuta a partir de la línea posterior a la condición de la sentencia IF. Este código permite además la anidación de sentencias IF, es decir sólo buscará el comando ELSE o ENDIF que le corresponda al IF inicial.

```
//busca a partir de offset la ocurrencia del endif correspondiente
//deja el offset después del endif si no hay error

private void SearchELSEorENDIF() throws Exception {
  int nifs=1;
  String firstword;

  while (nifs!=0 && offset!=-1) {
    firstword=getFirstWord(getNextLine());
    if (firstword.equals("IF")) nifs++;
    else if (firstword.equals("ELSE")) {
      if (nifs==1) nifs--;
    }
    else if (firstword.equals("ENDIF")) nifs--;
  }

  if (nifs!=0)
    throw (Errores.ENDIFnotfound);
}
```

Por otro lado, cuando debe ejecutarse el primer conjunto de sentencias no debe realizarse ningún salto ya que la condición se cumplió y este primer conjunto tiene su comienzo en la línea siguiente. Sin embargo, al finalizar la ejecución de este conjunto debemos saltarnos el posible segundo conjunto. Surgen aquí las siguientes preguntas: ¿cómo saber cuando acaba el primer conjunto? ¿hasta dónde debemos saltar para no ejecutar el segundo conjunto?

El final del primer conjunto de sentencias viene marcado por el comando ELSE, si la sentencia IF tiene dos partes, y por el comando ENDIF, si sólo tiene una. Es por eso que el comando ELSE sólo lanza un tipo de aviso al hilo *ProgramThread* padre, la excepción *Errores.ELSEM*. Con esta excepción le avisa de que llegó el final del primer conjunto de sentencias. Si el final viniese marcado por el comando ENDIF no debemos hacer ninguna búsqueda porque el segundo conjunto de sentencias no existe.

```
private void SearchENDIF() throws Exception {
  String firstword;
  int nifs=1;

  while (nifs!=0 && offset!=-1) {
    firstword=getFirstWord(getNextLine());
    if (firstword.equals("ENDIF")) nifs--;
    else if (firstword.equals("IF")) nifs++;
```

```
    }

    if (nifs!=0)
      throw (Errores.ENDIFnotfound);
  }
```

En el caso de los bucles del tipo FOR...ENDFOR debemos realizar la búsqueda del comando ENDFOR cuando el bucle haya finalizado su iteración sobre la variable índice. Esto se realiza así porque el bucle no termina con el comando ENDFOR, sino con la comparación del índice al principio. La búsqueda del comando ENDFOR se realiza en el sentido de incremento del contador de programa.

La forma en la que se busca el comando ENDFOR correspondiente permite también la posible anidación de bucles FOR. Es decir, para buscar el comando ENDFOR correspondiente debemos saltarnos los comandos ENDFOR interiores que correspondan a otros bucles FOR.

```
  private void SearchENDFOR() throws Exception {
    int nfors=1;
    String firstword;

    while (nfors!=0 && offset!=-1) {
      firstword=getFirstWord(getNextLine());
      if (firstword.equals("FOR")) nfors++;
      else if (firstword.equals("ENDFOR")) nfors--;
    }

    if (nfors!=0) throw (Errores.ENDFORnotfound);
  }
```

Por último debemos comentar la actualización del contador de programa debido a sentencias GOTO. Debido a que este tipo de sentencias permite un salto incondicional a cualquier punto del programa, debemos empezar la búsqueda de la etiqueta desde el principio del programa. Para ello se interpretan todas las etiquetas encontradas y son comparadas con la etiqueta a la que debemos saltar. Tras una comparación afirmativa la búsqueda finaliza y se actualiza el contador de programa con la posición del siguiente comando a dicha etiqueta.

Si llegásemos al final del texto y no hubiese sido encontrada la etiqueta, se lanzaría una excepción del tipo *EtiquetaNoEncontrada*.

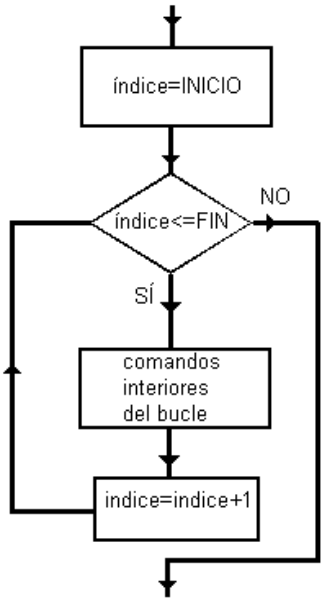
```
  private void SearchLABEL(int numero) throws Exception {
    offset=-1;
    boolean encontrada=false;
```

```
    String linea;
    int i;

    do {
      linea=getNextLine();
      if (getFirstWord(linea).equals("LABEL")) {
        linea=linea.trim();
        i=linea.indexOf(' ');
        if (i==-1) throw (Errores.LABELformato);
        else {
          try {
            encontrada=
              (Integer.parseInt(linea.substring(i).trim())==numero);
          }
          catch (NumberFormatException err) {
            throw (Errores.LABELformato);
          }
        }
      }
    } while (!encontrada && offset!=-1);

    if (!encontrada)
      throw (Errores.EtiquetaNoEncontrada);
  }

}
```

9.1.1.5 Sentencias especiales

Los bucles FOR...ENDFOR son tratados de una forma especial. El funcionamiento general de estos bucles es verificar que la variable que sirve de índice está dentro del rango y ejecutar el grupo de sentencias interiores. Posteriormente se incrementa la variable índice y repetir. Aquí hemos explicado sólo el caso en el que el valor inicial es menor que el valor final para el rango del índice, en el caso contrario sólo hay que disminuir la variable índice.

El diagrama de flujo simplificado de un bucle del tipo FOR puede ser como el mostrado a la derecha.



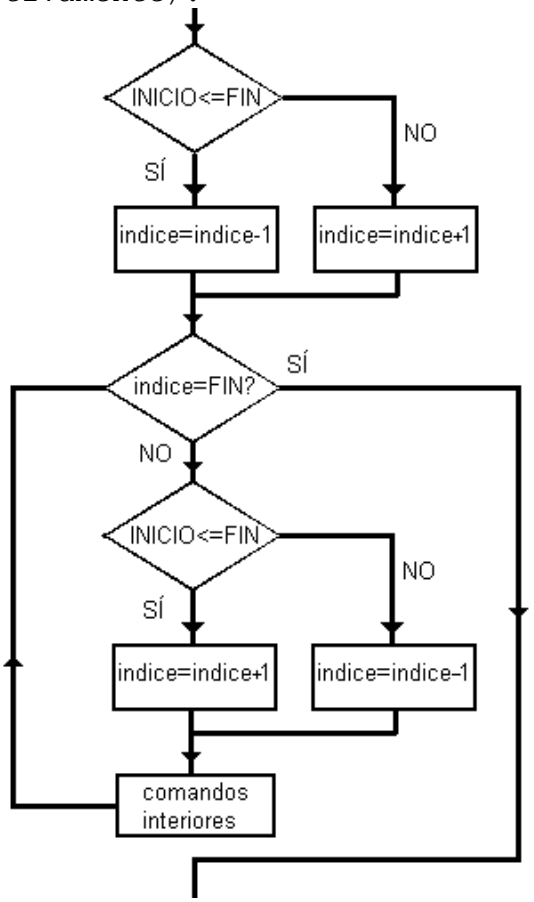
A la hora de implementar este tipo de bucles se encontró el siguiente problema: ¿dónde incrementar la variable índice? La implementación directa hubiera sido al final, es decir cuando se encuentra el comando que apunta el final del bucle ENDFOR. Esto quiere

decir que a la hora de ejecutar el comando ENDFOR deberíamos tener alguna información de que variable actúa como índice del bucle.

Otro problema existente es cómo diferenciar entre la primera vez que entramos en el bucle y las siguientes. La primera vez debemos inicializar la variable índice con el valor inicial y comprobar el rango. Las siguientes sólo debemos comprobar el rango.

La solución elegida fue la siguiente: hacer las comprobaciones y el incremento de la variable índice al principio del bucle. Pero debemos inicializar la variable índice con el valor inicial fuera del bucle para que sólo se inicialice una vez. Para comprobar el funcionamiento de los comandos FOR y ENDFOR podemos consultar el apartado número 10.2.5.

En este diagrama se muestra el flujo de instrucciones causadas por un bucle del tipo FOR para ambos tipos, es decir, recorridos en sentido directo e inverso (cuando INICIO<=FIN y INICIO>FIN respectivamente).



Para realizar dichas acciones el conjunto FOR...ENDFOR debe modificarse (según la manera explicada en el apartado número 9.1.1.1) de la forma que indica este procedimiento.

```
public static String tratarFOR(String linea) {
  Vector q=ParamParser.eval(linea.toUpperCase());
  String inicioT=ParamParser.ToolJoinEx(q,1,"TO");
  if (inicioT==null) return (null);
  String fin=ParamParser.ToolJoinEx(q,"TO",q.size());
  if (fin==null) return (null);

  int eqindex=inicioT.indexOf('=');
  if (eqindex==-1) return (null);

  String inicio=inicioT.substring(eqindex+1);
  String variable=inicioT.substring(0,eqindex);

  String resultado="IF "+inicio+"<="+fin+"\n";
  resultado=resultado+"SET "+variable+"="+inicio+"-1\n";
  resultado=resultado+"ELSE\n";
  resultado=resultado+"SET "+variable+"="+inicio+"+1\n";
  resultado=resultado+"ENDIF\n";

  return (resultado);
}
```

Con esto se consigue introducir una sentencia IF antes del bucle FOR que inicializa la variable índice al valor inicial disminuyéndola en una unidad en el caso en el que el valor inicial sea menor que el final, o incrementándola en caso contrario. Este incremento es debido a que la forma en la que se ha implementado el bucle FOR hace que se ejecuten una vez más los comandos interiores del bucle.

Otro hecho importante que debe ser tratado con cuidado es el anidamiento de sentencias. Para los bucles del tipo FOR esto ha sido implementado gracias a una pila llamada *forstack*. Esta pila es una variable local al hilo dado por la instancia del objeto *ProgramThread*.

El método usado es el siguiente: cuando empiece la ejecución de un bucle FOR se anotará la dirección de comienzo de este bucle. De este modo, cuando lleguemos al comando ENDFOR sólo debemos recuperar la dirección del bucle correspondiente. El hecho de que funcione bien la anidación es gracias al funcionamiento de la pila.

La posición que debe ser introducida en la pila es la dada por el comando FOR, por tanto esta posición debe ser introducida antes de ejecutar el comando ya que una vez ejecutado el contador de programa se incrementa. Por tanto, existe una rutina en el procedimiento *runProgram()* (ver

apartado número 9.1.1.3) que se encarga de comprobar que comando va a ser ejecutado por *Command.doAction()* (apartado número 10) antes de que sea mandado a ejecución.

ejemplo

Supongamos el siguiente programa:

```
DEFINE I J      //definición de las variables locales
FOR I=1 TO 5    //bucle exterior
FOR J=1 TO 5    //bucle interior
ENDFOR          //fin bucle interior
ENDFOR          //fin bucle exterior
```

Para el ejemplo también supondremos que cada línea está numerada en orden creciente. El simulador no funciona con números de línea como la máquina real, sino con posición dentro de un texto.

Al principio la pila está vacía, pero al llegar al comienzo del bucle exterior meteríamos en la pila esta posición. Cuando lleguemos al comienzo del bucle interior también sería introducida esa posición.

Por tanto, cuando recuperemos la posición de la pila con el primer comando ENDFOR, la posición recuperada sería la del bucle FOR interior. Cuando este bucle se haya repetido las cinco veces, el siguiente comando ENDFOR recuperaría la posición del bucle más exterior.

Este mecanismo funciona debido a que la posición que se recupera de la pila es la dada por el comando FOR. Por lo tanto, al recuperar la posición de la pila, volverá a ser introducida para que el posterior comando ENDFOR cumpla su cometido.

9.2 La clase ProgramGroups

Esta clase se encargará de gestionar el acceso a los programas que se están ejecutando. Los comandos GOSUB, CONTINUE, SUSPEND, ABORT y STOP hacen uso de esta clase. A grandes rasgos funciona de la misma manera que las clases *Simples*, *Tablas*, etc., es decir, sirve de intermediario entre los hilos de ejecución de programas y los comandos.

Los procedimientos de esta clase hacen de interlocutores entre los comandos y las instancias de objetos *ProgramThread*. Por ejemplo, la función *addThread()* crea un nuevo objeto *ProgramThread* y ejecuta su método *run()*. Las funciones

suspendThread() y *resumeThread()* llaman a los comandos *dormir()* y *despertar()* de un *ProgramThread* dado, respectivamente.

Del mismo modo, existen también algunos métodos para verificar el estado de ejecución.

```
public class ProgramGroups {

  private static Vector grupo=new Vector();
  private static MainFrame parent;

  public static void init(MainFrame q) {
    parent=q;
  }
```

La forma que tienen los comandos de acceder a un programa es mediante su nombre. Por tanto debemos tener alguna función que indexe los hilos en cuanto a su nombre, ésta es la función de *getThreadIndex()*.

```
  private static synchronized int getThreadIndex(String name) {}
```

Las siguientes funciones son las encargadas de crear y destruir y ejecutar hilos de ejecución, es decir instancias de objetos *ProgramThread*.

```
  public static synchronized void addThread(ProgramThread q) {
    grupo.add(q);
  }

  public static synchronized void deleteThread(String name) throws
Exception {
    int index=getThreadIndex(name);
    if (index!=-1) grupo.remove(index);
    else throw (Errores.ProgramNoExiste);
  }


  public static synchronized void executeThread(String name) throws
Exception {
    int index=getThreadIndex(name);
    if (index!=-1) {
      ProgramThread q=(ProgramThread) grupo.get(index);
      q.start();
    }
    else {
      parent.addRunningProgram(name,false); //la añadimos a la barra
      index=getThreadIndex(name);
      ProgramThread q=(ProgramThread) grupo.get(index);
      q.start();
    }
  }
```

```
  public static synchronized void createNewProgram(String name,String
program) throws Exception {
    parent.addProgram(name,program,false);
    executeThread(name);
  }
```

Los comandos SUSPEND, CONTINUE y ABORT o STOP, llaman respectivamente a las funciones *suspendThread()*, *resumeThread()* y *breakThread()*.

```
  public static synchronized void suspendThread(String name) throws
Exception {
    int index=getThreadIndex(name);
    if (index!=-1) {
      ProgramThread q=(ProgramThread) grupo.get(index);
      q.parar();
    }
    else throw (Errores.ProgramNoExiste);
  }

  public static synchronized void resumeThread(String name) throws
Exception {
    int index=getThreadIndex(name);
    if (index!=-1) {
      ProgramThread q=(ProgramThread) grupo.get(index);
      q.despierta();
    }
    else throw (Errores.ProgramNoExiste);
  }

  public static synchronized void breakThread(String name) throws Exception
  {
    int index=getThreadIndex(name);
    if (index!=-1) {
      ProgramThread q=(ProgramThread) grupo.get(index);
      q.despierta();
      q.interrumpe();
    }
    else throw (Errores.ProgramNoExiste);
  }

  public static synchronized void breakAll() {
    ProgramThread q;
    for (int i=0;i<grupo.size();i++) {
      q=(ProgramThread) grupo.get(i);
      q.despierta();
      q.interrumpe();
    }
  }
```

También disponemos de funciones que permiten consultar el estado de los hilos.

```
  public static synchronized boolean isAlive(String name) throws Exception
  {
```

```
    int index=getThreadIndex(name);
    if (index!=-1) {
      ProgramThread q=(ProgramThread) grupo.get(index);
      return (q.isAlive());
    }

    throw (Errores.ProgramNoExiste);
  }

  public static synchronized boolean hasFinishedThread(String name) {
    int index=getThreadIndex(name);
    if (index==-1) return(true);
    return(false);
  }

  public static synchronized String getJobs() {
    if (grupo.size()==0) return("no jobs");

    StringBuffer q=new StringBuffer();
    ProgramThread pg;

    for (int i=0;i<grupo.size();i++) {
      pg=(ProgramThread) grupo.get(i);
      q.append(pg.getNombre());
      q.append(" prioridad="); q.append(pg.getPrioridad());
      q.append(" estado="); q.append(pg.getEstado());
      q.append('\n');
    }

    return (q.toString());
  }
```

Con la función *setPriorityThread()* podemos cambiar la prioridad asignada a un hilo. De esta forma conseguiremos que la cpu le asigne mayor o menor tiempo de ejecución a nuestro programa ACL. Para más detalles sobre prioridades consultar el apartado número 9.3.

```
  public static synchronized void setPriorityThread(String name,int
prioridad) throws Exception {
    int index=getThreadIndex(name);
    if (index!=-1) {
      ProgramThread q=(ProgramThread) grupo.get(index);
      if (prioridad>10) prioridad=10;
      if (prioridad<1) prioridad=1;
      //10 será maximo y 1 minimo
      //en java será 10 Thread.NORM normal y 1 minima
      q.setPrioridad(prioridad);
      double a=Thread.NORM_PRIORITY-Thread.MIN_PRIORITY;
      a=a/9.0;
      double b=10*Thread.MIN_PRIORITY-Thread.NORM_PRIORITY;
      b=b/9.0;
      double res=a*prioridad+b;
      int p=(int) Math.floor(res);
      q.setPriority(p);
    }
    else throw (Errores.ProgramNoExiste);
```

```
  }
}
```

Cuando el simulador es arrancado se crean en primer lugar todos los objetos estáticos para que vayan siendo accesibles. Debido a que el estado de ejecución de los threads debe reflejarse en la ventana principal, necesita de la referencia de la ventana *MainFrame*. Con esta referencia podrá actualizar dicho estado en la pantalla.

Es necesario hacer mención aquí a la gestión multitarea de Java y como ha servido para simular la máquina en la que se ejecuta el lenguaje ACL.

9.3 Multitarea en Java

Para explicar como ha sido transportado el carácter multitarea de la máquina real en la que se ejecuta el lenguaje ACL al lenguaje Java supondremos que el programa está siendo ejecutado en una máquina con una única cpu. Esto no nos hace perder generalidad en nuestro comentario ya que, hasta la fecha, las máquinas para ámbito doméstico todavía poseen una sola cpu.

Los threads en Java se ejecutan generalmente por timeslots, es decir intervalos de tiempo. Cuando a un thread se le acaba su timeslot o pasa a estado "bloqueado", cede su timeslot al siguiente thread programado en la cola. Esto se repetirá hasta que ya no queden más threads, en ese momento se volverá al primero de la cola. A este algoritmo se le conoce como round-robin.

Si a este algoritmo le añadimos las prioridades tenemos la siguiente modificación al algoritmo round-robin. Se realizará el algoritmo round-robin sobre los threads de prioridad más alta. Cuando ya no queden threads de esa prioridad por finalizar su ejecución, o estén todos bloqueados, se realizará round-robin sobre los de prioridad de un nivel más bajo. Es decir, no se ejecutará ningún thread de prioridad baja mientras exista un thread de mayor prioridad listo para ejecución.

Si realizando round-robin sobre un grupo de igual prioridad aparece un thread de mayor prioridad, el thread en ejecución soltará su timeslot en beneficio del nuevo.

En Java existen tres constantes definidas para las prioridades: *Thread.NORM_PRIORITY*, *Thread.MAX_PRIORITY* y *Thread.MIN_PRIORITY*. Estas constantes deben usarse llamándolas

así y no por su valor, puesto que pueden depender de la implementación particular de la JVM.

Una vez explicado como funciona la gestión multitarea en Java veremos como se aplica al simulador, y como nos ayudará a copiar el método en que se gestionan los programas en el lenguaje ACL.

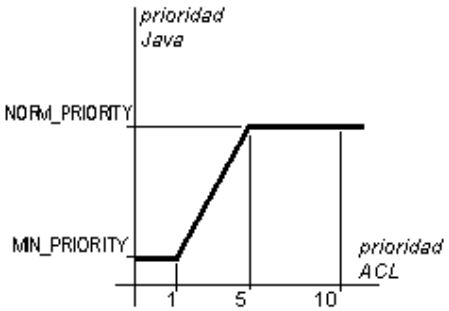
En el lenguaje ACL se define para cada programa una prioridad. Esta prioridad será un entero en el rango 1 a 10, siendo 5 la prioridad por defecto. Aquel programa que tenga prioridad mayor se ejecutará antes que los que tengan prioridad menor. Aquí deben entenderse los calificativos mayor y menor de modo numérico, es decir 5 es menor que 10 y 10 es mayor que 5.

Para programas con igual prioridad el lenguaje ACL no especifica que hacer con ellos.

Por tanto, se realizó una ley lineal del siguiente modo: se asigna a la prioridad 1 la prioridad *Thread.MIN_PRIORITY* y a la 10 la constante *Thread.NORM_PRIORITY*. Estas asignaciones sólo se ven internamente, el usuario en ningún momento da cuenta de ellas.

No se asignó *Thread.MAX_PRIORITY* a la prioridad 10 ya que al asignar esa prioridad a un thread se ejecuta en tiempo real en la JVM y, por tanto, no dejaría ejecutarse a ningún otro thread hasta que finalizase. Esto no sería problemático si no fuese que tampoco deja ejecutarse a los objetos *Swing*, *AWT* y demás, así como tampoco se volvería a dibujar la pantalla en la que se muestra el modelo.

Para la conversión de prioridades se usó una relación como la que muestra el diagrama siguiente:



Esta conversión está implementada en el procedimiento *setPriorityThread()* en la clase *ProgramGroup*.

10 INTÉRPRETE DE COMANDOS

En este punto trataremos el procedimiento general de ejecución de comandos. En resumen el proceso es el siguiente: la clase *ProgramThread* busca un comando por línea, y este comando será mandado como parámetro a un procedimiento que será el encargado de generar la secuencia de comandos apropiada.

El procedimiento que se encarga de buscar el comando y mandarlo al procedimiento adecuado es *doAction()*, el cual está en la clase *Command*. Éste debe extraer el supuesto comando de una línea de texto y comprobar que de verdad es un comando. Para ello dispone de una lista de todos los comandos que soporta el lenguaje ACL (que es consultada también por los creadores de variables y posiciones, ver apartados número 8.1 y 8.2, para comprobar la validez de los nombres) con la que compara el comando extraído. Debemos recalcar el hecho de que no todos los comandos del lenguaje ACL han sido simulados en este software, es por ello que si un comando no se encuentra en la lista total se lanzará una excepción del tipo *Errores.NotACommand*, y si un comando está en la lista pero no ha sido simulado se lanzaría *Errores.CommandnotAvailable*.

Este procedimiento está declarado como estático ya que es usado por todos los programas en ejecución, así como por el modo directo de ejecución de comandos. Si fuera declarado como dinámico debería crearse una instancia de él cada vez que fuese creado un programa, cosa que ralentizaría mucho la creación de hilos *ProgramThread*.

```
package SCORBOTSimulator.Ordenes;


import SCORBOTSimulator.CommonTools;
import SCORBOTSimulator.Errores;

public class Command {

  public static void doAction(String line,String parent,int modo) throws
Exception {
    if (line.length()==0) return; //no es un error
    int separacion=line.indexOf(' ');
    if (separacion==-1) separacion=line.indexOf('\t');
    String p,r;

    if (separacion==-1) {
      p=line; r="";
    }
```

```
else {
  p=line.substring(0,separacion);
  r=line.substring(separacion+1).trim();
}

//comprobamos que sea una palabra reservada
if (!CommonTools.isReservada(palabra))
  throw (Errores.NotACommand);

p=p.toUpperCase();
//realizamos la accion
//creacion y manejo de variables simples y tablas simples
if (p.equals("DEFINE")) Asignacion.doDEFINE(r,p,m);
else if (p.equals("GLOBAL")) Asignacion.doGLOBAL(r,p,m);
else if (p.equals("SET")) Asignacion.doSET(r,parent,modo);
else if (p.equals("DELVAR")) Asignacion.doDELVAR(r,parent,modo);
else if (p.equals("DIM")) Asignacion.doDIM(r,parent,modo);
else if (p.equals("DIMG")) Asignacion.doDIMG(r,parent,modo);
//creacion y manejo de posiciones y vectores de posiciones
else if (p.equals("DEFP")) OpPosiciones.doDEFP(r,parent,modo);
else if (p.equals("DELP")) OpPosiciones.doDELP(r,parent,modo);
else if (p.equals("SETP")) OpPosiciones.doSETP(r,parent,modo);
else if (p.equals("SETPV")) OpPosiciones.doSETPV(r,parent,modo);
else if (p.equals("SETPVC")) OpPosiciones.doSETPVC(r,parent,modo);
else if (p.equals("SHIFT")) OpPosiciones.doSHIFT(r,parent,modo);
else if (p.equals("SHIFTC")) OpPosiciones.doSHIFTC(r,parent,modo);
else if (p.equals("DIMP")) OpPosiciones.doDIMP(r,parent,modo);
else if (p.equals("HERE")) OpPosiciones.doHERE(r,parent,modo);
else if (p.equals("HERER")) OpPosiciones.doHERER(r,parent,modo);
else if (p.equals("TEACH")) OpPosiciones.doTEACH(r,parent,modo);
else if (p.equals("TEACHR")) OpPosiciones.doTEACHR(r,parent,modo);
else if (p.equals("UNDEF")) OpPosiciones.doUNDEF(r,parent,modo);
//entrada y salida
else if (p.equals("PRINT")) EntradaSalida.doPRINT(r,parent,modo);
else if (p.equals("PRINTLN")) EntradaSalida.doPRINTLN(r,parent,modo);
else if (p.equals("CLS")) EntradaSalida.doCLS(r);
else if (p.equals("READ")) EntradaSalida.doREAD(r,parent,modo);
else if (p.equals("LISTP")) EntradaSalida.doLISTP(r,parent,modo);
else if (p.equals("LISTPV")) EntradaSalida.doLISTPV(r,parent,modo);
else if (p.equals("LISTVAR")) EntradaSalida.doLISTVAR(r,parent,modo);
else if (p.equals("SHOW")) EntradaSalida.doSHOW(r,parent,modo);
else if (p.equals("VER")) EntradaSalida.doVER(r,parent,modo);
else if (p.equals("STAT")) EntradaSalida.doSTAT(r,parent,modo);
else if (p.equals("HELP")) EntradaSalida.doHELP(r,parent,modo);
//condiciones y estructuras
else if (p.equals("IF")) Condicion.doIF(r,parent,modo);
else if (p.equals("ANDIF")) Condicion.doANDIF(r,parent,modo);
else if (p.equals("ORIF")) Condicion.doORIF(r,parent,modo);
else if (p.equals("ELSE")) Condicion.doELSE(r,parent,modo);
else if (p.equals("ENDIF")) {}
else if (p.equals("FOR")) Condicion.doFOR(r,parent,modo);
else if (p.equals("ENDFOR")) Condicion.doENDFOR(r,parent,modo);
//movimiento
else if (p.equals("OPEN")) Control.doOPEN(r,parent,modo);
else if (p.equals("CLOSE")) Control.doCLOSE(r,parent,modo);
else if (p.equals("MOVE")) Control.doMOVE(r,parent,modo);
else if (p.equals("MOVED")) Control.doMOVED(r,parent,modo);
else if (p.equals("MOVEL")) Control.doMOVEL(r,parent,modo);
```

```
    else if (p.equals("MOVELD")) Control.doMOVELD(r,parent,modo);
    else if (p.equals("MOVES")) Control.doMOVES(r,parent,modo);
    else if (p.equals("MOVESD")) Control.doMOVESD(r,parent,modo);
    else if (p.equals("HOME")) Control.doHOME(r,parent,modo);
    else if (p.equals("SPEED")) Control.doSPEED(r,parent,modo);
    //rotura del hilo de ejecucion
    else if (p.equals("LABEL")) Saltos.doLABEL(r,parent,modo);
    else if (p.equals("GOTO")) Saltos.doGOTO(r,parent,modo);
    else if (p.equals("GOSUB")) Saltos.doGOSUB(r,parent,modo);
    //sincronizacion
    else if (p.equals("DELAY")) Sincronizacion.doDELAY(r,parent,modo);
    else if (p.equals("PRIORITY")) Sincronizacion.doPRIORITY(r, parent,
modo);
    else if (p.equals("WAIT")) Sincronizacion.doWAIT(r,parent,modo);
    else if (p.equals("RUN")) Sincronizacion.doRUN(r,parent,modo);
    else if (p.equals("ABORT")||p.equals("A")) Sincronizacion.doABORT(r,
parent, modo);
    else if (p.equals("STOP")) Sincronizacion.doSTOP(r,parent,modo);
    else if (p.equals("SUSPEND")) Sincronizacion.doSUSPEND(r,parent,modo);
    else if (p.equals("CONTINUE")) Sincronizacion.doCONTINUE(r, parent,
modo);
    else if (p.equals("TRIGGER")) Sincronizacion.doTRIGGER(r,parent,modo);
    else if (p.equals("POST")) Sincronizacion.doPOST(r,parent,modo);
    else if (p.equals("PEND")) Sincronizacion.doPEND(r,parent,modo);
    else if (p.equals("QPOST")) Sincronizacion.doQPOST(r,parent,modo);
    else if (p.equals("QPEND")) Sincronizacion.doQPEND(r,parent,modo);

    //varios
    else if (p.equals("DISABLE")) Varios.doDISABLE(r,parent,modo);
    else if (p.equals("ENABLE")) Varios.doENABLE(r,parent,modo);
    else if (p.equals("CON")) Varios.doCON(r,parent,modo);
    else if (p.equals("COFF")) Varios.doCOFF(r,parent,modo);
    //error
    else throw (Errores.COMMANDnotAvailable);
  }

}
```

Como se puede comprobar tras una simple observación del código, se trata de un procedimiento que busca la primera palabra de la línea y compara con todos los comandos posibles.

Todos los comandos tienen una sintaxis general a la hora de aceptar los comandos: línea a ejecutar sin el comando, programa que mandó el comando y modo en el que se está ejecutando.

ejemplo

Si fuese mandado al procedimiento *doAction()* una línea como la siguiente "MOVELD POS1" desde un programa llamado PROG1, se actuaría de la siguiente manera.

- se separa el comando del resto de la línea: "MOVELD POS1"="MOVELD"+"POS1"

- se compara el supuesto comando extraído de la línea "MOVELD" con la lista de comandos y se ejecuta el procedimiento correspondiente si la comparación tuvo éxito
- se ejecuta el procedimiento *Control.doMOVELD("POS1", "PROG1", Const.EDITOR)*. De este modo le indicamos al procedimiento que estamos ejecutando un comando en modo editor y le enviamos el nombre del programa por si tuviese que comprobar la visibilidad de alguna variable. En este caso no tendría que comprobar dicha visibilidad ya que las posiciones son tratadas de forma global.

Si este comando fuese ejecutado en modo directo la secuencia habría sido similar con la única diferencia que la llamada sería de la siguiente forma: *Control.doMOVELD("POS1", "GLOBAL", Const.DIRECTO)*.

La línea a ejecutar es la misma que manda el programa pero ya se le ha extraído el comando. Esto es simplemente para acelerar un poco el proceso posterior, ya que realmente no haría falta.

El segundo parámetro necesario es el nombre del programa que pide la ejecución de un comando. Este es necesario para que en las operaciones internas que realicen aquellos comprueben la visibilidad de las variables. También existen algunos comandos que sólo pueden ejecutarse sobre variables globales como veremos más adelante.

El tercer parámetro es el modo. Este parámetro necesita una explicación más detenida ya que se usa para simular otro aspecto del lenguaje ACL: el modo de ejecución.

10.1 Modos de ejecución

En el lenguaje ACL existen dos modos distintos de ejecución: el modo directo y el modo editor.

El modo directo es el que se nos presenta nada más arrancar el programa monitor del SCORBOT. Este modo permite la ejecución directa de comandos del lenguaje ACL. Como veremos más adelante no todos los comandos se pueden ejecutar en este modo.

Este modo se caracteriza porque cualquier comando que se ejecute lo hace de forma global. Por ejemplo, si creamos una variable será una variable global. Además los comandos son ejecutados de uno en uno, y no se devuelve el control al usuario hasta que no haya terminado la ejecución.

En el simulador se ejecutan los comandos en modo directo a partir de la ventana de usuario, en el recuadro al efecto (ver apartado número 4.3).

El modo editor es el modo en el que son ejecutados los programas. Este modo se pone en funcionamiento en el mismo momento en el que hacemos correr el programa que hemos escrito.

En el simulador ambos modos son idénticos, ya que el modo directo llama directamente al intérprete *Command.doAction()* y el modo editor llama a través de *ProgramThread* al mismo procedimiento. La única diferencia es el parámetro que se le da como entrada a este procedimiento que en un caso será *Const.DIRECTO* y en el otro *Const.EDITOR*. De esta manera cada comando, antes de realizar propiamente su acción, comprobará que el modo desde el cual se hace la llamada es el apropiado. Si el modo no es correcto será lanzada una excepción del tipo "modo no válido" y será presentado un mensaje por pantalla para avisar al usuario del simulador.

10.2 Procedimientos para la simulación de los comandos

Ahora realizaremos una breve alusión a todos los comandos simulados del lenguaje ACL.

Debemos recordar el hecho de que la máquina en la que corre el lenguaje ACL es una máquina multitarea. Este hecho hizo que tuviésemos que considerar la sincronización/secuenciación en el acceso a los valores de las variables y otros objetos que eran visibles por todos los procedimientos.

Los procedimientos que simulan los comandos no necesitan de sincronización. Esto es debido a que, aún siendo procedimientos estáticos visibles desde todos los restantes procesos y ejecutables por cualquier thread activo, cuando un thread ejecuta uno de estos procedimientos se crean todas sus variables dinámicamente.

De este modo se restringe la sincronización sólo en algunas partes del simulador, y gracias a esto y la declaración estática de los procedimientos que simulan los comandos se acelera la interpretación de los programas.

Han sido divididos los comandos en varias categorías:

- Manejo de variables
- Manejo de posiciones
- Entrada y salida

- Sincronización
- Bucles, saltos y condiciones
- Movimiento
- Otros comandos

Dentro de cada categoría veremos cada comando por separado, explicando el formato que requiere a su entrada y como ha sido simulado.

El procedimiento general que siguen todos los comandos es el siguiente:

```
public static void doCOMANDO(parametros,programa,modo) throws Exception {
    compruebaModo();
    compruebaParametros();
    realizaFunciones();
    si (hubo error) lanzaExcepcion();
}
```

El código que realizará la acción del comando simulado ha sido a veces omitido para simplificar la lectura y debido a la similitud de la forma de implementar algunos comandos. Esto es, sólo han sido incluidos los comandos que pueden ofrecer algún detalle añadido al proyecto.

Los comandos que se pueden ejecutar en ambos modos de ejecución no realizan la comprobación de modo, así como los comandos que sólo necesitan un parámetro de entrada no realizan tampoco la comprobación del número de estos.

10.2.1 Comandos para el manejo de variables

Estos comandos son: DEFINE, GLOBAL, SET, DELVAR, DIM y DIMG. Todos estos comandos se limitan a usar los procedimientos de las clases *Simples* y *Tablas* (ver apartados número 8.1.1 y 8.1.2), los cuales nos ayudarán a crear, acceder y destruir variables.

DEFINE
formato: *DEFINE <var1> ... <var8>*
modo: editor

Este comando se encarga de la creación de variables simples de modo local. Como entrada sólo requiere los nombres de las ocho posibles variables para ser creadas de una sola vez.

Crea las variables dadas llamando al procedimiento *add()* de la clase *Simples*.

```
  public static void doDEFINE(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);

    Vector q=ParamParser.eval(line.toUpperCase());

    if (q.size()>8 || q.size()==0) throw (Errores.DEFINEformat);

    String nombre;
    //solo se crea la variable si no existe
    //si existe y esta warning activado se da el mensaje
    for (int i=0;i<q.size();i++) {
      nombre=(String) q.get(i);
      if (!Simples.contains(nombre,parent)) Simples.add(nombre,0,parent);
      else if (Configuracion.isWarningOn())
        MainFrame_TextOUT.printOutLn("W: la variable \""+nombre+"\" ya ha
sido creada");
    }
  }
```

Posibles errores: modo no válido, exceso o falta de parámetros y variable ya existente.

GLOBAL
formato: *GLOBAL <var1> ... <var8>*
modo: todos

Este comando se encarga de la creación de variables de forma global. Ha sido simulado a través del comando DEFINE pero cambiando el parámetro *parent* a "GLOBAL".

Posibles errores: los mismos que el comando DEFINE.

DELVAR
formato: *DELVAR <var>*
modo: todos

Sirve para borrar variables creadas con DEFINE o GLOBAL. Sólo admite un nombre cada vez. También puede borrar variables tipo tabla. Este comando llama al procedimiento *erase()* de la clase *Simples* o *Tablas*, según se trate de una variable simple o una tabla. Para saber a qué procedimiento llamar comprueba antes la existencia de la variable de forma local y después de forma global.

```
  public static void doDELVAR(String line,String parent,int modo) throws
Exception {
```

```
    line=line.toUpperCase();

    //simpleslocales, tablaslocales, simplesglobales y tablasglobales
    if (Simples.contains(line,parent)) Simples.erase(line,parent);
    else if (Tablas.contains(line,parent)) Tablas.erase(line,parent);
    else if (Simples.contains(line,"GLOBAL")) Simples.erase(line,"GLOBAL");
    else if (Tablas.contains(line,"GLOBAL")) Tablas.erase(line,"GLOBAL");
    else throw (Errores.DELVARformat);
  }
```

Debemos recalcar el hecho de que aquí también se ha hecho la simulación de visibilidad de variables. A la hora de borrar busca en este orden: entre las simples con el mismo padre, las tablas con el mismo padre, las simples globales y las tablas globales.

Posibles errores: formato no válido (que equivale a que no ha sido encontrada ninguna variable con el nombre indicado en <var>).

DIM
formato: *DIM <var>[tam]*
modo: editor

Crea una variable tipo tabla del tamaño indicado. La crea de manera local, por tanto sólo es válida en modo editor. Este comando llama al procedimiento *add()* de la clase *Tablas*.

```
  public static void doDIM(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);

    String[] nombre=CommonTools.getNameAndIndex(line.toUpperCase());

    if (nombre[1]==null) throw (Errores.DIMformat);

    int size;
    try {
      size=Integer.parseInt(nombre[1]);
    } catch (NumberFormatException err) {
      throw (Errores.EnteroNoValido);
    }

    if (!Tablas.contains(nombre[0],parent))
      Tablas.add(nombre[0],size,parent);
    else if (Configuracion.isWarningOn())
      MainFrame_TextOUT.printOutLn("W: la variable \""+nombre[0]+"\" ya ha
sido creada");
  }
```

Posibles errores: formato no válido y variable ya creada.

DIMG
formato: *DIMG <var>[tam]*
modo: todos

Este comando crea una variable tipo tabla de forma global. Para ello, al igual que GLOBAL hacía uso de DEFINE, DIMG hace uso de DIM. Por tanto, sólo realiza una llamada al procedimiento que simula DIM con el nombre del programa igual a *GLOBAL*

SET
formato:
SET <var>=<var> OPER <var>, SET <var>=PVAL <pos> <coord>
SET <var>=PVALC <pos> <coord>, SET <var>=PSTATUS <pos>
modo: todos

Este comando será uno de los más usados junto con los de movimiento. Con SET se realizan las operaciones numéricas básicas sobre las variables. Además permite acceder a las campos que representan las coordenadas de una posición.

El procedimiento para realizar las operaciones se basa en primero separar la operación en operandos, operador y destino. El destino será la variable en la que se guardará el resultado de la operación. Básicamente se trata de realizar la operación indicada una vez obtengamos los valores de los correspondientes operandos.

```
  public static void doSET(String line,String parent,int modo) throws
Exception {
    line=line.toUpperCase();
    int eqindex=line.indexOf('=');
    if (eqindex==-1) throw (Errores.SETformat);

    String[] left=CommonTools.getNameAndIndex(line.substring(0,eqindex));
    String right=line.substring(eqindex+1).trim();

    if (right.length()==0 || left[0]==null)
      throw (Errores.SETformat);

    int resultado=0;
    char q=right.charAt(0);
    int i=0;
    //primero vamos a ver si es una operacion
```

Para realizar la operación en primer lugar se busca el operador. Se realiza una búsqueda sobre los más fáciles: suma, resta, multiplicación y división. Estos operadores requieren dos operandos.

```
    while (q!='+' && q!='-' && q!='/' && q!='*') {
      i++;
      if (i==right.length()) break;
      q=right.charAt(i);
    }

    if (i!=right.length()) { //dos operandos
      int op1=SimpleParser.eval(right.substring(0,i),parent);
      int op2=SimpleParser.eval(right.substring(i+1),parent);

      switch (q) {
        case '+': resultado=op1+op2; break;
        case '-': resultado=op1-op2; break;
        case '*': resultado=op1*op2; break;
        case '/':
          if (op2==0) throw (Errores.DivideByZero);
          resultado=op1/op2;
          break;
      }
    }
    else { //puede ser una operacion compuesta
```

Si no se encuentra ninguno se buscarán los operadores compuestos por palabras. Para ello debe extraer las partes de la operación y comprobar su número. Sólo son válidas las operaciones con 1, 2 ó 3 parámetros.

```
      Vector oper=ParamParser.eval(right);
      String operador;

      switch (oper.size()) {
        case 0: throw (Errores.SETformat);
        case 1: resultado=SimpleParser.eval((String) oper.get(0),parent);
break;
        case 2:
          operador=(String) oper.get(0);
          if (operador.equals("PSTATUS")) {
            String[] posic=CommonTools.getNameAndIndex((String)
oper.get(1));
            if (posic[1]==null) resultado=Posiciones.getSTATUS(posic[0]);
            else resultado=TPosiciones.getSTATUS(posic[0],
                    SimpleParser.eval(posic[1],parent));
          }
```

Las operaciones de 1 operando deben limitarse a evaluarlo. Las de dos operandos son las que tienen el operador primero: ABS, NOT y COMPLEMENT.

```
          else if (operador.equals("NOT"))
            resultado=(SimpleParser.eval((String) oper.get(1),
parent)>0)?Const.TRUE:Const.FALSE;
          else if (operador.equals("COMPLEMENT"))
            resultado=~SimpleParser.eval((String) oper.get(1),parent);
          else if (operador.equals("ABS")) {
            int aux=SimpleParser.eval((String) oper.get(1),parent);
            resultado=(aux>0)?aux:-aux;
```

```
          }
          else throw (Errores.SETformat);
          break;
```

Las de tres partes son las que tienen el operador entre los dos operandos: SIN, COS, TAN, LOG, ATAN, MOD... Estos requieren también dos operandos que son tratados de forma especial puesto que el primero actúa como factor de escala del resultado. Esto se debe al hecho de que la máquina en la que corre el lenguaje es una máquina que funciona con aritmética entera.

```
        case 3:
          operador=(String) oper.get(1);
          if (operador.equals("COS")) {
            int op1=SimpleParser.eval((String) oper.get(0),parent);
            int op2=SimpleParser.eval((String) oper.get(2),parent);
            resultado=(int) Math.floor(op1*Math.cos(Math.toRadians(op2)));
          }
          else if (operador.equals("SIN")) {
            int op1=SimpleParser.eval((String) oper.get(0),parent);
            int op2=SimpleParser.eval((String) oper.get(2),parent);
            resultado=(int) Math.floor(op1*Math.sin(Math.toRadians(op2)));
          }
          else if (operador.equals("TAN")) {
            int op1=SimpleParser.eval((String) oper.get(0),parent);
            int op2=SimpleParser.eval((String) oper.get(2),parent);
            resultado=(int) Math.floor(op1*Math.tan(Math.toRadians(op2)));
          }
          else if (operador.equals("ATAN")) {
            int op1=SimpleParser.eval((String) oper.get(0),parent);
            int op2=SimpleParser.eval((String) oper.get(2),parent);
            resultado=(int) Math.floor(op1*Math.atan(((double)
op2)/10000.0));
          }
          else if (operador.equals("EXP")) {
            int op1=SimpleParser.eval((String) oper.get(0),parent);
            int op2=SimpleParser.eval((String) oper.get(2),parent);
            double res=op1*Math.exp(((double) op2)/10000.0);
            resultado=(int) Math.floor(res);
          }
          else if (operador.equals("LOG")) {
            int op1=SimpleParser.eval((String) oper.get(0),parent);
            int op2=SimpleParser.eval((String) oper.get(2),parent);
            double res=op1*Math.log(((double) op2)/10000.0);
            resultado=(int) Math.floor(res);
          }
          else if (operador.equals("MOD")) {
            int op2=SimpleParser.eval((String) oper.get(2),parent);
            if (op2==0) throw (Errores.DivideByZero);
            resultado=SimpleParser.eval((String) oper.get(0),parent)%op2;
          }
          else if (operador.equals("AND")) {
            resultado=SimpleParser.eval((String) oper.get(0),parent) &
              SimpleParser.eval((String) oper.get(2),parent);
          }
```

```
          else if (operador.equals("OR")) {
            resultado=SimpleParser.eval((String) oper.get(0),parent) |
              SimpleParser.eval((String) oper.get(2),parent);
          }
          else {
```

Si no encontrase ninguno de estos operadores buscará los que actúan sobre posiciones.

```
            operador=(String) oper.get(0);
            String[] posic=CommonTools.getNameAndIndex((String)
oper.get(1));
            String param=(String) oper.get(2);
            if (operador.equals("PVAL")) {
              int j;
              try {
                j=Integer.parseInt(param);
              } catch (NumberFormatException err) {
                throw (Errores.SETformat);
              }
              if (posic[1]==null) {
                resultado=Posiciones.getParam(posic[0],""+j);
              } else {
                resultado=TPosiciones.getParam(posic[0],
                  SimpleParser.eval(posic[1],parent),""+j);
              }
            }
            else if (operador.equals("PVALC")) {
              if (param.equals("X") || param.equals("Y") ||
param.equals("Z") ||
                param.equals("P") || param.equals("R")) {
                if (posic[1]==null) resultado=Posiciones.getParam(posic[0],
param);
                else resultado=TPosiciones.getParam(posic[0],
                    SimpleParser.eval(posic[1],parent),param);
              }
              else throw (Errores.SETformat);
            }
            else throw (Errores.SETformat);
          }

          break;
      }

    }

    //asignación del resultado a la variable
    if (left[1]==null) //es simple no una tabla
      Simples.set(left[0],resultado,parent);
    else //es una tabla
      Tablas.set(left[0], SimpleParser.eval(left[1], parent), resultado,
parent);
  }
```

Posibles errores: todos los propios de acceso a los valores de las variables (visibilidad, no existencia, etc...), y los de no encontrar ningún operador válido, o falta de parámetros.

10.2.2 Comandos para el manejo de posiciones

Estos comandos están englobados en la clase *OpPosiciones*: HERE, SHIFT, DEFP, etc... Son operandos para manejar posiciones, pueden acceder a las coordenadas articulares y cartesianas, también nos ayudarán a manejar posiciones relativas y a unir el modelo del robot con el intérprete de comandos.

DEFP
formato: *DEFP <pos>*
modo: todos

Este comando nos ayudará a crear posiciones. Se limita a hacer las funciones de intermediario entre el programa y la clase *Posiciones*, a través del procedimiento *add()*.

```
  public static void doDEFP(String line,String parent,int modo) throws
Exception {
    if (line.length()==0)
      throw (Errores.DEFPformato);

    line=line.toUpperCase();

    if (!Posiciones.contains(line)) Posiciones.add(line);
    else if (Configuracion.isWarningOn())
      MainFrame_TextOUT.printOutLn("W: la posicion \""+line+"\" ya ha sido
creada");
  }
```

Posibles errores: posición ya creada, nombre no válido para una posición, etc...

DIMP
formato: *DIMP <pvect>[tam]*
modo: todos

Este realiza las funciones de intermediario entre el programa y la clase *TPosiciones*. Funciona de la misma manera que el comando DEFP, sólo que aquí también se debe encontrar el tamaño del vector de posiciones.

Posibles errores: nombre no válido, vector de posiciones ya creado, tamaño no válido, etc...

DELP
formato:
DELP <pos>, DELP <pvect>
modo: todos

Sirve para borrar una posición o vector de posiciones. Si deseamos borrar un vector de posiciones no hace falta incluir el tamaño de éste. Para realizar esta acción usa el procedimiento *erase()* de la clase *Posiciones* o *TPosiciones*.

```
  public static void doDELP(String line,String parent,int modo) throws
Exception {

    Vector q=ParamParser.eval(line.toUpperCase());
    if (q.size()!=1)
      throw (Errores.DELPformato);

    String pos=(String) q.get(0);

    if (Posiciones.contains(pos)) Posiciones.erase(pos);
    else if (TPosiciones.contains(pos)) TPosiciones.erase(pos);
    else throw (Errores.DELPformato);
  }
```

Posibles errores: posición o vector de posiciones no existente, número de parámetros no válido.

HERE
formato: *HERE <pos>*
modo: todos

Este comando es uno de los más importantes en el manejo real del SCORBOT. Su función es copiar la posición actual del robot a una variable del tipo posición. Internamente copia los campos de las coordenadas cartesianas y articulares de la posición ACTUAL sobre la que nosotros demos como parámetro a través del comando SETP. Esta es una de las razones por las que la posición ACTUAL es considerada del sistema y no se puede borrar.

El uso normal de este comando será el de definición de nuevas posiciones cuando estemos usando los controles de la pistola de programación.

Este comando no crea la posición, por tanto la posición dada como argumento debe estar creada previamente. También cambia el status de una posición a "absoluta".

```
  public static void doHERE(String line,String parent,int modo) throws
Exception {
```

```
    Vector q=ParamParser.eval(line.toUpperCase());
    if (q.size()!=1)
      throw (Errores.HEREformato);

    String[] posicion=CommonTools.getNameAndIndex((String) q.get(0));
    if (posicion[0]==null)
      throw (Errores.HEREformato);

    if (posicion[1]==null) {
      try {
        doSETP(posicion[0]+"=ACTUAL",parent,modo);
      } catch (Exception err) {
        if (err==Errores.SETPformato) throw (Errores.HEREformato);
        else throw (err);
      }
      Posiciones.setSTATUS(posicion[0],Const.ST_PABS);
    }
    else {
      try {
        doSETP(posicion[0]+"["+posicion[1]+"]=ACTUAL",parent,modo);
      } catch (Exception err) {
        if (err==Errores.SETPformato) throw (Errores.HEREformato);
        else throw (err);
      }

      int pos=SimpleParser.eval(posicion[1],parent);
      TPosiciones.setSTATUS(posicion[0],pos,Const.ST_PABS);
    }
  }
```

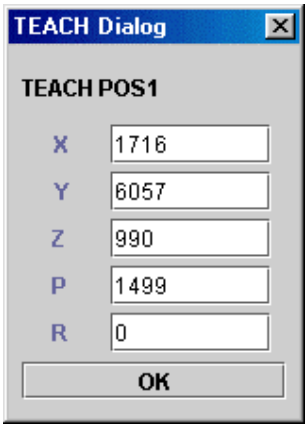
Posibles errores: formato no válido, posición no definida, etc...

TEACH

formato: *TEACH <pos>*

modo: directo

Este comando nos permitirá actualizar cada uno de los campos de coordenadas cartesianas de una posición. Abre una ventana de diálogo en la pantalla en la que podemos introducir cada una de las coordenadas cartesianas.



En esta ventana serán introducidos los nuevos valores para las coordenadas cartesianas, los que aparecerán serán los antiguos. Los nuevos se grabarán tras pulsar sobre el botón OK. Si fue introducido algún valor no válido para alguna de las coordenadas, éste tomará un color rojo y no será válido hasta que no sea cambiado.

Cuando aparece esta ventana es de obligado cumplimiento, es decir sólo se puede terminar la actualización pulsando sobre el botón OK o cerrándola. Ninguna de las otras ventanas aceptará comandos hasta que no haya acabado la actualización.

Este procedimiento sólo se limita a abrir dicha ventana, después al pulsar sobre el botón OK serán copiados todos los valores a los campos de la posición dada como entrada. La actualización de dichos campos se hará a través del procedimiento *setParam()* de la clase *Posiciones* o *TPosiciones*.

Posibles errores: formato de entrada incorrecto y los que pueda dar la entrada de valores en el cuadro de diálogo.

SETP

formato: *SETP <pos1>=<pos2>*
modo: todos

Este comando copia todos los campos de una posición <pos2> a otra posición <pos1>. También copia el status de una posición. Realmente se hace una copia de una variable posición en otra, son idénticas salvo el nombre.

Este comando es utilizado por el comando HERE, el cual realiza una llamada al comando SETP con la posición a copiar la ACTUAL.

Para realizar la copia de los campos de la posición origen a la destino, se hace uso de las funciones *getParam()* y *setParam()* de la clase *Posiciones* o *TPosiciones*.

```
  public static void doSETP(String line,String parent,int modo) throws
Exception {
    line=line.toUpperCase();

    int equal=line.indexOf('=');
    if (equal==-1)
      throw (Errores.SETPformato);

    Vector ql=ParamParser.eval(line.substring(0,equal));
    Vector qr=ParamParser.eval(line.substring(equal+1));

    if (ql.size()!=1||qr.size()!=1)
      throw (Errores.SETPformato);

    String[] left=CommonTools.getNameAndIndex((String) ql.get(0));
    String[] right=CommonTools.getNameAndIndex((String) qr.get(0));

    if (left[0]==null || right[0]==null)
      throw (Errores.SETPformato);

    int[] xyz;
    int pitch,roll;
```

```
    int[] joints;
    int tipo;

    if (right[1]==null) {
      xyz=Posiciones.getXYZ(right[0]);
      pitch=Posiciones.getParam(right[0],"P");
      roll=Posiciones.getParam(right[0],"R");
      joints=Posiciones.getJoints(right[0]);
      tipo=Posiciones.getSTATUS(right[0]);
    }
    else {
      int pos=SimpleParser.eval(right[1],parent);
      xyz=TPosiciones.getXYZ(right[0],pos);
      pitch=TPosiciones.getParam(right[0],pos,"P");
      roll=TPosiciones.getParam(right[0],pos,"R");
      joints=TPosiciones.getJoints(right[0],pos);
      tipo=TPosiciones.getSTATUS(right[0],pos);
    }

    if (left[1]==null) {
      Posiciones.setXYZ(left[0],xyz[0],xyz[1],xyz[2]);
      Posiciones.setPR(left[0],pitch,roll);
      Posiciones.setJoints(left[0],joints);
      Posiciones.setSTATUS(left[0],tipo);
    }
    else {
      int pos=SimpleParser.eval(left[1],parent);
      TPosiciones.setXYZ(left[0],pos,xyz[0],xyz[1],xyz[2]);
      TPosiciones.setPR(left[0],pos,pitch,roll);
      TPosiciones.setJoints(left[0],pos,joints);
      TPosiciones.setSTATUS(left[0],pos,tipo);
    }

  }
```

Posibles errores: formato de entrada incorrecto, posiciones no existentes, etc...

SHIFT
formato: *SHIFT <pos> BY <eje> <var>*
modo: todos

Realiza un desplazamiento aritmético sobre alguna de las coordenadas articulares de una posición, es decir, suma una cantidad <var> al valor de la coordenada articular <eje> de la posición <pos>.

Como se puede observar al final del código, cada vez que se modifica alguna de las coordenadas articulares se recalculan todas las coordenadas cartesianas a partir de la nueva posición dada por el conjunto de articulares. Para ello usamos el procedimiento *updateXYZPRfromJoints()* de la clase *Posiciones*. Cuando tratemos el comando SHIFTC veremos como se llamará al finalizar al comando *updateJointsFromXYZPR()* para actualizar el juego contrario de coordenadas.

```
  public static void doSHIFT(String line,String parent,int modo) throws
Exception {

    Vector q=ParamParser.eval(line.toUpperCase());

    if (q.size()!=4 || !q.get(1).equals("BY"))
      throw (Errores.SHIFTformato);

    //obtenemos el nombre de la variable y el posible indice
    String[] nombre=CommonTools.getNameAndIndex((String) q.get(0));
    if (nombre[0]==null)
      throw (Errores.SHIFTformato);

    String coord=(String) q.get(2);
    String expr=(String) q.get(3);

    try { //comprobación de si es un número
      int eje=Integer.parseInt(coord);
    } catch (NumberFormatException err) {
      throw (Errores.AxisInvalid);
    }

    int cantidad=SimpleParser.eval(expr,parent);

    if (nombre[1]==null) {
      int anterior=Posiciones.getParam(nombre[0],coord);
      Posiciones.setParam(nombre[0],coord,anterior+cantidad);
      if (Const.DEBUG)
        System.out.println("SHIFT "+nombre[0]+" by "+expr+" "+coord);
      //actualizamos la posicion definida por las articulaciones
      Posiciones.updateXYZPRfromJoints(nombre[0]);
    }
    else {
      int indice=SimpleParser.eval(nombre[1],parent);
      int anterior=TPosiciones.getParam(nombre[0],indice,coord);
      TPosiciones.setParam(nombre[0],indice,coord,anterior+cantidad);
      if (Const.DEBUG)
        System.out.println("SHIFT "+nombre[0]+"["+nombre[1]+"]"+" by
"+expr+" "+coord);
      //actualizamos la posicion definida por las articulaciones

      TPosiciones.updateXYZPRfromJoints(nombre[0],indice);
    }
  }
```

Posibles errores: articulación no válida, posición no existente, etc...

SHIFTC

formato: *SHIFTC <pos> BY <artic> <var>*

modo: todos

Trabaja del mismo modo que el comando SHIFT salvo que sirve para modificar las coordenadas cartesianas x, y, z, p ó r.

Como se puede observar al final del código, cada vez que se realiza una modificación sobre alguna de las coordenadas cartesianas se recalcula el valor de las coordenadas articulares, para actualizarlas a la nueva posición a través de la función *updateJointsFromXYZPR()*.

El código de este comando ha sido omitido por su similitud con el del comando SHIFT.

Posibles errores: coordenada no válida, posición no existe, etc...

UNDEF

formato: *UNDEF <pos>*
modo: directo

Este comando normalmente no se utiliza durante una sesión real con el SCORBOT, ha sido incluido para total compatibilidad con el juego de comandos del lenguaje ACL.

Realiza la función de cambiar el status de una posición a no definido. Además borra todos los campos de la posición, es como si creáramos de nuevo la posición.

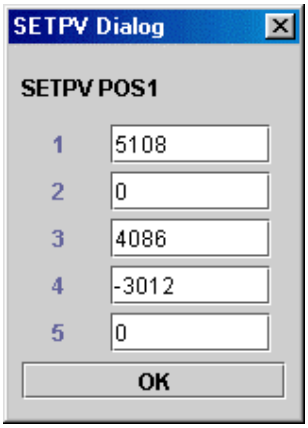
El código cambia el estado de la posición o vector de posiciones a ST_PUNDEF e iguala a la posición ZEROPOS.

SETPV

formato:
SETPV <pos> <cartic> <var>
SETPV <pos>
modo: todos

Comando que nos permite asignar un valor a una coordenada articular de una posición. Difiere del comando SHIFT en el hecho de que no se realiza un desplazamiento sobre el valor sino una sobrescritura.

Si no incluimos la coordenada articular que queremos cambiar aparecerá una pantalla de diálogo que nos permitirá modificar todos los valores de la posición. En esta pantalla aparecerán los valores antiguos de las coordenadas articulares. Los nuevos se grabarán tras pulsar sobre el botón OK.




```
  public static void doSETPV(String line,String parent,int modo) throws
Exception {
```

```
    //<pos> <ejes> <val>
    Vector q=ParamParser.eval(line.toUpperCase());
    if (q.size()!=3 && q.size()!=1)
      throw (Errores.SETPVformato);

    //obtenemos el nombre de la variable y el posible indice
    //<pos>
    String[] nombre=CommonTools.getNameAndIndex((String) q.get(0));
    if (nombre[0]==null)
      throw (Errores.SETPVformato);

    if (q.size()==1) {
      CommonTools.checkModo(modo,Const.DIRECTO);
      SETPVDialogFrame dlg = new SETPVDialogFrame(nombre[0],nombre[1]);
      dlg.setLocation(30,30);
      dlg.setModal(true);
      dlg.show();
      return;
    }

    if (q.size()==3) {

      String coord=(String) q.get(1); //<eje>
      String expr=(String) q.get(2); //<val>

      try { //comprobacion de si es un numero
        int eje=Integer.parseInt(coord);
      } catch (NumberFormatException err) {
        throw (Errores.AxisInvalid);
      }

      int cantidad=SimpleParser.eval(expr,parent);

      if (nombre[1]==null) {
        Posiciones.setParam(nombre[0],coord,cantidad);
        if (Const.DEBUG)
          System.out.println("SETPV "+nombre[0]+" "+coord+" "+expr);
        //actualizamos la posici¾n definida por las articulaciones
        Posiciones.updateXYZPRfromJoints(nombre[0]);
      }
      else {
        int indice=SimpleParser.eval(nombre[1],parent);
        TPosiciones.setParam(nombre[0],indice,coord,cantidad);
        if (Const.DEBUG)
          System.out.println("SETPV "+nombre[0]+"["+nombre[1]+"]"+"
"+coord+" "+expr);
        //actualizamos la posici¾n definida por las articulaciones
        TPosiciones.updateXYZPRfromJoints(nombre[0],indice);
      }
    }
  }
```

Debido a que este comando puede funcionar en los dos modos y funciona de forma diferente en cada uno, el código debe comprobar en que modo fue realizada la llamada para así

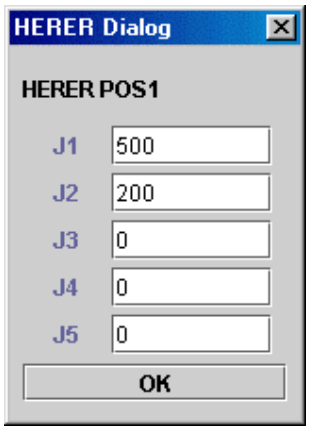
proceder con el funcionamiento adecuado. Sólo el modo directo debe lanzar la actualización con ventana de diálogo.

SETPVC
formato: *SETPVC <pos> <ccart> <var>*
modo: todos

Comando que nos proporciona un modo de asignar valores absolutos a las coordenadas cartesianas de una posición. Funciona del mismo modo que el comando SHIFTC con la diferencia de que este sobrescribe los valores. Hace uso de la función *setParam()* para actualizar los campos indicados en <ccart>.

Debido a la similitud con ese comando ha sido omitido de la documentación el código.

HERER
formato:
HERER <pos>
HERER <pos1> <pos2>
modo: depende



Comando que nos permite hacer una posición <pos1> relativa a otra <pos2>. Si no se incluye <pos2> se hará <pos1> relativa a la actual. Este última forma sólo es soportada en modo directo puesto que muestra una ventana de diálogo que nos permite introducir los desplazamientos relativos a la posición actual en coordenadas articulares.

Llegado a este punto es necesaria una aclaración acerca de las posiciones relativas a la posición actual. La posición actual hace referencia a la que ha sido alcanzada por el robot, es decir la que marca la posición ACTUAL en el momento de su lectura.

En esta pantalla aparecerán los valores antiguos de las articulares. Los nuevos se grabarán tras pulsar sobre el botón OK.

```
  public static void doHERER(String line,String parent,int modo) throws
Exception {

    Vector q=ParamParser.eval(line.toUpperCase());

    //hay dos casos
```

```
    if (q.size()==1) {
      //esto solo soportado en modo directo
      CommonTools.checkModo(modo,Const.DIRECTO);
      String[] left=CommonTools.getNameAndIndex((String) q.get(0));
      HERERDialogFrame dlg = new HERERDialogFrame(left[0],left[1]);
      dlg.setLocation(30,30);
      dlg.setModal(true);
      dlg.show();
    }
    else if (q.size()==2) { //este en los dos
      //CommonTools.checkModo(modo,Const.TODOS);

      String[] left=CommonTools.getNameAndIndex((String) q.get(0));
      String right=(String) q.get(1);
      if (left[1]==null) {
        Posiciones.setRelativa(left[0],right);
        if (right.equals("ACTUAL"))
          Posiciones.setSTATUS(left[0],Const.ST_PREL_JACTU);
        else
          Posiciones.setSTATUS(left[0],Const.ST_PREL_JOTRA);
      }
      else {
        int i=SimpleParser.eval(left[1],parent);
        TPosiciones.setRelativa(left[0],i,right);
        if (right.equals("ACTUAL"))
          TPosiciones.setSTATUS(left[0],i,Const.ST_PREL_JACTU);
        else
          TPosiciones.setSTATUS(left[0],i,Const.ST_PREL_JOTRA);
      }
    }
    else throw (Errores.HERERformato);
  }
```

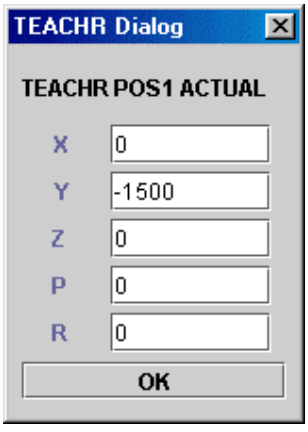
TEACHR

formato:

TEACHR <pos1>

TEACHR <pos1> <pos2>

modo: directo



Este comando nos permite, al igual que HERER, hacer que la posición <pos1> sea relativa a <pos2> en términos de coordenadas cartesianas. Si no se especifica <pos2> se tomará como la posición actual.

TEACHR sólo es soportado en modo directo, ya que ambas formas de utilizarlo hacen que aparezca una pantalla de diálogo en la que introducir los términos en los que es relativa.

En esta pantalla aparecerán los valores antiguos de las coordenadas articulares. Los nuevos se grabarán tras pulsar sobre el botón OK.

```
  public static void doTEACHR(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.DIRECTO);

    Vector q=ParamParser.eval(line.toUpperCase());

    //hay dos casos
    String[] left;
    String right;
    if (q.size()==1) {
      left=CommonTools.getNameAndIndex((String) q.get(0));
      right="ACTUAL";
    }
    else if (q.size()==2) {
      left=CommonTools.getNameAndIndex((String) q.get(0));
      right=(String) q.get(1);
    }
    else throw (Errores.TEACHRformato);

    TEACHRDialogFrame dlg = new TEACHRDialogFrame(left[0],left[1],right);
    dlg.setLocation(30,30);
    dlg.setModal(true);
    dlg.show();
  }

}
```

10.2.3 Comandos para la gestión de entrada y salida

Estos comandos permiten la gestión de la impresión de caracteres en la salida de texto de usuario y la entrada de datos por teclado del usuario.

PRINT y PRINTLN
formato: *PRINT/PRINTLN cad1 var1 cad2 var2 ...*
modo: editor

Estos comandos permiten la impresión en la ventana de salida de texto de cadenas de caracteres y valores de variables. Por razones de compatibilidad con la máquina original sólo admite como máximo 4 argumentos, los cuales pueden ser indistintamente cadenas o variables.

PRINT y PRINTLN son tratados de la misma manera ya que lo único que los diferencia es que el comando PRINTLN imprime al final un carácter de retorno de carro.

La implementación de los comandos ha sido realizada con una función única a la que llaman ambos, la función *doPrintBoth()*:

```
  private static void doPrintBoth(String linea,String parent,boolean
retorno) throws Exception {
    int i1=0,i2=0;
    int n=0;

    StringBuffer q=new StringBuffer();

    while (linea.length()>0) {
      i1=linea.indexOf('"',0);

      if (i1!=0) {
        if (i1==-1) i1=linea.length();
        Vector
params=ParamParser.eval(linea.substring(0,i1).toUpperCase());
        for (int i=0;i<params.size();i++) {
          q.append(SimpleParser.eval((String) params.get(i),parent));
          n++;
          if (n>4) throw (Errores.PRINTformat);
        }
        //quitamos lo que ya hemos tratado
        linea=linea.substring(i1).trim();
      }
      else {
        i2=linea.indexOf('"',1);
        if (i2==-1) throw (Errores.PRINTformat);
        else {
          q.append(linea.substring(1,i2));
          linea=linea.substring(i2+1).trim();
          n++;
          if (n>4) throw (Errores.PRINTformat);
        }
      }
    }

    if (retorno) q.append('\n');
    MainFrame_TextOUT.printOut(q.toString());
  }
```

Ambos comandos llaman a la función anterior, la única diferencia radica en el último parámetro con el cual le indicamos que debe incluir un retorno de carro al finalizar la impresión.

CLS
formato: *CLS*
modos: todos

Este comando no existe en la máquina original, ha sido introducido como una facilidad para el usuario del simulador.

CLS realiza una limpieza de la ventana de texto, borrando toda la información que en ella se muestra.

El código no será mostrado debido a que no introduce ninguna novedad sobre la forma de implementar las órdenes en el simulador.

LISTP
formato: *LISTP*
modo: directo

Muestra una lista de todas las posiciones y vectores de posición creados en el simulador. Con este comando no se pueden observar los campos de las posiciones, se limita a mostrar los nombres de dichas posiciones, y los vectores de posición con su tamaño. Ello es conseguido a través de las funciones *listAll()* de *Posiciones* y *TPosiciones*.

LISTVAR
formato: *LISTVAR*
modo: directo

Este comando muestra una lista de todas las variables, tanto simples como tablas, creadas en el simulador. Muestra también la visibilidad de dichas variables.

De la misma manera que LISTP, también consulta las funciones *listAll()* de *Simples* y *Tablas*.

LISTPV
formato: *LISTPV <pos>*
modo: directo

Muestra todos los campos de la posición indicada como argumento. El argumento puede ser una posición o un elemento de un vector de posiciones.

Este comando se limita a obtener los campos de la posición indicada en <pos> con la función *getParam()*, o *getJoints()* en el caso de los valores de los ejes.

SHOW
formato:
SHOW DIN,
SHOW DOUT,
SHOW ENCO,
SHOW SPEED
modo: directo

En los tres primeros casos nos muestra una visualización de los valores de las tablas IN, OUT y ENC. En el último caso muestra la velocidad actual.

Para obtener dichos valores hace uso de la función *get()* presente en la clase *Tablas* sobre todos los elementos de las tablas.

```
  public static void doSHOW(String linea,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.DIRECTO);
    linea=linea.toUpperCase();

    if (linea.equals("DIN")) {
      StringBuffer q=new StringBuffer();

      q.append("1 -> "); q.append(Const.NUMES); q.append(": ");

      for (int i=1;i<=Const.NUMES;i++) {
        q.append(Tablas.get("IN",i,parent));
        q.append(' ');
      }

      MainFrame_TextOUT.printOutLn(q.toString());
    }
    else if (linea.equals("DOUT")) {
      StringBuffer q=new StringBuffer();

      q.append("1 -> "); q.append(Const.NUMES); q.append(": ");

      for (int i=1;i<=Const.NUMES;i++) {
        q.append(Tablas.get("OUT",i,parent));
        q.append(' ');
      }

      MainFrame_TextOUT.printOutLn(q.toString());
    }
    else if (linea.equals("ENCO")) {
      StringBuffer q=new StringBuffer();

      for (int i=1;i<=Const.NUMJOINTS+1;i++) {
        q.append("enc"); q.append(i); q.append(": ");
        q.append(Tablas.get("ENC",i,parent)); q.append('\n');
      }

      MainFrame_TextOUT.printOutLn(q.toString());
    }
    else if (linea.equals("SPEED")) {
      MainFrame_TextOUT.printOutLn("GROUP A SPEED IS:" +
MovimientoRobot.getSpeed());
    }
    else throw (Errores.SHOWformat);
  }
```

VER
formato: *VER*
modo: directo

Muestra la versión del simulador. Se limita a mostrar el valor de algunas constantes por pantalla y es por eso que ha sido omitido el código.

STAT
formato: *STAT*
modo: directo

Muestra una lista de los programas que se están ejecutando en el simulador, además de los nombres nos muestra su estado. Para obtener el estado de todos los programas activos hace uso de la función *getJobs()* presente en la clase *ProgramGroups*. Esta función devolvía ya una cadena puesto que fue creada expresamente para ayudar a simular este comando.

Para más detalles sobre la consulta de estados de los programas consultar el apartado número 9.2.

```
  public static void doSTAT(String linea,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.DIRECTO);
    if (linea.length()>0) throw (Errores.STATformat);
    MainFrame_TextOUT.printOut(ProgramGroups.getJobs());
  }
```

HELP
formato: *HELP*
modo: directo

Este comando ha sido introducido por razones de compatibilidad ya que a la ayuda sobre los comandos se accede a partir del menú de la ventana principal, o pulsando la tecla F1 sobre el comando que se precisa la ayuda.

Se limita a presentar un mensaje advirtiendo del hecho.

READ
formato: *READ cad1 var1 cad2 var2 ...*
modo: editor

Permite asignar valores a variables que sean introducidos por el usuario a través del teclado. Como el comando PRINT, READ permite un máximo de cuatro argumentos.

Los argumentos pueden ser cadenas o variables. Se asignarán los valores introducidos por teclado a las variables.

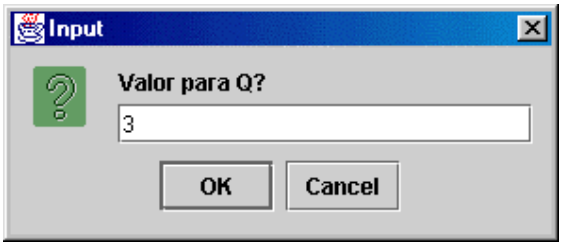
Se distingue entre cadena y variable ya que las cadenas deben ser mostradas, mientras que para las variables se pedirá un valor por pantalla. Para pedir dicho valor se usa la función *pideUsuario()* que será mostrada posteriormente.

```
  public static void doREAD(String linea,String parent,int modo) throws
Exception {
    int i1=0,i2=0;
    int n=0;

    StringBuffer q=new StringBuffer();

    while (linea.length()>0) {
      i1=linea.indexOf('"',0);

      if (i1!=0) {
        if (i1==-1) i1=linea.length();
        Vector params=ParamParser.eval(linea.substring(0,
i1).toUpperCase());
        for (int i=0;i<params.size();i++) {
          pideUsuario(q.toString(),(String) params.get(i),parent);
          n++;
          if (n>4) throw (Errores.READformat);
        }
        //quitamos lo que ya hemos tratado
        linea=linea.substring(i1).trim();
        q.setLength(0);
      }
      else {
        i2=linea.indexOf('"',1);
        if (i2==-1) throw (Errores.READformat);
        else {
          q.append(linea.substring(1,i2));
          linea=linea.substring(i2+1).trim();
          n++;
          if (n>4) throw (Errores.READformat);
        }
      }
    }
  }
```

Esta función presenta una ventana de diálogo en la cual se pide un valor. Este valor será asignado a la variable dada por el parámetro de entrada variable. Además muestra un texto a modo de título que debe servir para aclarar el motivo por el cual fue pedido un valor.

```
  private static void pideUsuario(String texto,String variable,String
parent) throws Exception {
    String resultado=JOptionPane.showInputDialog(texto+"?");
    int res=0;
    if (resultado!=null && resultado.length()>0) {
```

```
      try {
        res=Integer.parseInt(resultado);
      } catch (NumberFormatException err) {
        throw (Errores.EnteroNoValido);
      }
    }
    //asignamos el resultado a la variable
    String[] vari=CommonTools.getNameAndIndex(variable);
    if (vari[1]==null)
      Simples.set(vari[0],res,parent);
    else
      Tablas.set(vari[0],SimpleParser.eval(vari[1],parent),res,parent);
  }
```



Por ejemplo, una sentencia como la siguiente "READ "valor para Q" Q (se supone la existencia de una variable global llamada Q), producirá el siguiente resultado:

10.2.4 Procedimientos para simular los comandos de sincronización

Estos procedimientos simularán los comandos DELAY, RUN, WAIT, ABORT, QPEND, QPOST... Son utilizados para las labores de sincronización de eventos dentro de un programa.

La simulación de estos comandos a veces se ha realizado con bucles de espera activa y otras con semáforos y herramientas para la sincronización propias de Java.

Debido a que los procedimientos son declarados como *static* son visibles por todos los programas que sean ejecutados. Esto quiere decir que a la hora de usar, por ejemplo, una técnica de espera activa no se debe realizar dentro del procedimiento sino dentro del hilo que llamó a este procedimiento. Para ello fueron utilizadas excepciones para mandar mensajes al hilo del programa en ejecución (ver apartado número 9.1.1.3).

DELAY
formato: *DELAY <cant>*
modo: editor

Lanza una excepción de espera *"$DELAY-NNNN"* al hilo principal. Esta excepción es recogida y se ejecuta un comando *delay(NNNN)* de Java con *NNNN* en milisegundos por eso debe ser multiplicada por 10 ya que en ACL se utiliza la centésima de segundo como unidad.

```
  public static void doDELAY(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    int tiempo=SimpleParser.eval(line.toUpperCase(),parent);
    if (tiempo<0) throw (Errores.DELAYformato);

    throw new Exception("$DELAY-"+(tiempo*10));
  }
```

PRIORITY

formato: *PRIORITY <prog> <var>*
modo: editor

Ajusta la prioridad de ejecución de un programa. Sólo sirve con programas que ya estén en ejecución (aunque pueden estar bloqueados). Hace uso de la función *setPriorityThread()* definida en la clase *ProgramGroups*.

```
  public static void doPRIORITY(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    Vector q=ParamParser.eval(line.toUpperCase());
    if (q.size()!=2)
      throw (Errores.PRIORITYformato);

    ProgramGroups.setPriorityThread((String) q.get(0),
      SimpleParser.eval((String) q.get(1),parent));
  }
```

WAIT

formato: *WAIT <var1> <cond> <var2>*
modo: editor

Realiza la comprobación de la condición incorporada en el comando. La simulación ha sido realizada haciendo que el procedimiento lance una excepción si no se cumple la condición y no haga nada en caso contrario.

Este comando debe simular una espera activa, aunque como ya explicamos antes la espera activa no debe realizarse en este procedimiento puesto que de ese modo ningún otro programa podría acceder a él. En el caso de que hubiese sido simulada la espera dentro del procedimiento, cualquier otro programa que realizase alguna llamada a este procedimiento debería esperar a que finalizase la espera anterior. Si la espera del primero dependía de algún evento externo provocado por el segundo programa se provocaría una situación conocida como inanición.

Si la condición no se cumple se lanza una excepción que es recogida por el programa principal el cual hace que el contador de programa, que indica la posición de la ejecución del programa, quede invariable. Con esto se consigue que se vuelva a repetir la última instrucción y simular la espera activa.

Por el contrario, si la condición se cumple el comando no dará ningún resultado. Esto conseguirá que la ejecución del programa siga su curso normal, es decir, se incremente el contador de programa.

```
  public static void doWAIT(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    boolean res=CompParser.eval(line.toUpperCase(),parent);
    if (!res) throw (Errores.WAITCONTINUE);
  }
```

RUN

formato: *RUN <prog> [<prioridad>]*
modos: todos

Este comando permite la ejecución de programas asignándoles una prioridad desde el principio. Si esta prioridad es omitida se le asignará una por defecto, prioridad 5.

Antes de lanzar a ejecución el nuevo programa se realiza una pequeña comprobación para ver si existe una copia de él en ejecución mediante la función *hasFinishedThread()*.

Dos programas que se llamen igual aunque estén guardados en distintos lugares son considerados iguales. Esto quiere decir que no es conveniente tener programas abiertos con nombres iguales. Aunque podría considerarse un error, no lo es ya que la máquina real no permite tener dos programas en memoria con el mismo nombre. Esto ha sido simulado de forma que puedan ser abiertos pero no ejecutados simultáneamente.

Para crear el nuevo thread usa la función *executeThread()* y después le será asignada la prioridad con *setPriorityThread()*, funciones definidas en la clase *ProgramGroups*.

```
  public static void doRUN(String line,String parent,int modo) throws
Exception {
    Vector q=ParamParser.eval(line.toUpperCase());
    if (q.size()!=1 && q.size()!=2)
      throw (Errores.RUNformato);

    if (q.size()==1) {
      if (ProgramGroups.hasFinishedThread((String) q.get(0)))
        ProgramGroups.executeThread((String) q.get(0));
    }
```

```
    else if (q.size()==2) {
      int p=SimpleParser.eval((String) q.get(1),parent);
      if (ProgramGroups.hasFinishedThread((String) q.get(0))) {
        ProgramGroups.executeThread((String) q.get(0));
        ProgramGroups.setPriorityThread((String) q.get(0),p);
      }
    }
  }
```

STOP
formato: *STOP <prog>*
modo: editor

Este comando termina la ejecución el programa indicado en el argumento <prog>. Para ello se ayuda de una variable llave propia de cada hilo de ejecución, que al activarse hace que dicho programa no continue en ejecución. De esta forma se consigue que un programa no concluya sin haber terminado de ejecutar la instrucción en curso.

La variable que se modifica es *interrumpido* (local a una instancia de *ProgramThread*), a través de la función *breakThread()* de la clase *ProgramGroups*. Para más detalles consultar el apartado número 9.1.1.3.

SUSPEND
formato: *SUSPEND <prog>*
modo: todos

El comando SUSPEND trabaja de igual manera que STOP. La principal diferencia es que la variable llave no está ahora en el bucle principal (el bucle *do..while* de *runProgram()*), que haría que finalizase la ejecución, sino que es comprobada antes para detener la ejecución. De igual manera la ejecución es detenida una vez haya finalizado el último comando.

Para más detalles consultar el apartado número 9.1.1.3.

```
  public static void doSUSPEND(String line,String parent,int modo) throws
Exception {
    ProgramGroups.suspendThread(line.toUpperCase());
  }
```

CONTINUE
formato: *CONTINUE <prog>*
modo: todos

Una vez haya sido detenida la ejecución del programa indicado con el comando SUSPEND, el comando CONTINUE sólo debe

cambiar el valor de la variable llave y la ejecución se reanudará desde el punto donde fue detenida.

```
  public static void doCONTINUE(String line,String parent,int modo) throws
Exception {
    ProgramGroups.resumeThread(line.toUpperCase());
  }
```

ABORT
formato: *ABORT [<prog>]*
modo: directo

Este comando es un claro ejemplo de que existen comandos que pueden funcionar en ambos modos pero por razones de compatibilidad ha sido copiado y renombrado para los dos modos de ejecución.

Realmente el código es el mismo que el del comando STOP, sólo se diferencian en que se hace una comprobación para ver si se ha incluido algún argumento. En caso afirmativo actúa igual que STOP, en el contrario detiene todos los programas en ejecución a través de la función *breakAll()* de *ProgramGroups*.

```
  public static void doABORT(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.DIRECTO);
    if (line.length()>0) ProgramGroups.breakThread(line.toUpperCase());
    else ProgramGroups.breakAll();
  }
```

TRIGGER
formato: *TRIGGER <prog> BY IN/OUT <n> {<estado>}*
modo: editor

El comando TRIGGER fue uno de los más complicados de simular. En la máquina real este comando crea una entrada en una "tabla de vectores de interrupción" haciendo que un cambio en una de las entradas y salidas lógicas lance la ejecución de un programa.

Debido a que estas E/S lógicas han sido simuladas mediante software, la manera más inmediata para simular las interrupciones fue creando un programa que vigilase los cambios. Estos cambios, una vez detectados, harían que fuese lanzado un evento de ejecución.

Esta manera es muy poco eficiente, pero debemos recordar que estamos ante un programa de simulación. El fin primordial es el de conocer el manejo de los comandos y el de investigar el funcionamiento general de la máquina, por tanto la eficiencia queda relegada a un segundo plano.

El código se limita a crear un programa y añadirlo a la cola de ejecución. La vigilancia de los cambios de estado ha sido simulada mediante el comando WAIT.

```
  public static void doTRIGGER(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    Vector q=ParamParser.eval(line.toUpperCase());

    if (q.size()!=4 && q.size()!=5)
      throw (Errores.TRIGGERformato);
    if (!q.get(1).equals("BY"))
      throw (Errores.TRIGGERformato);
    if (!q.get(2).equals("IN") && !q.get(2).equals("OUT"))
      throw (Errores.TRIGGERformato);

    String prog=(String) q.get(0);
    String tabla=(String) q.get(2);
    int n=SimpleParser.eval((String) q.get(3),parent);

    if (q.size()==5) { //cuando tenga un estado concreto
      int estado=SimpleParser.eval((String) q.get(4),parent);
      String nprog="TRIGGER_"+tabla+"_"+n+"_"+estado;
      String texto="WAIT "+tabla+"["+n+"]="+estado+"\n";
      texto=texto+"RUN "+prog+"\n";
      ProgramGroups.createNewProgram(nprog,texto);
    }
    else if (q.size()==4) { //cuando cambie
      String nprog="TRIGGER_"+tabla+"_"+n+"_CAMBIO";
      String texto="IF "+tabla+"["+n+"]=1\n";
      texto=texto+"WAIT "+tabla+"["+n+"]=0\n";
      texto=texto+"ELSE\n";
      texto=texto+"WAIT "+tabla+"["+n+"]=1\n";
      texto=texto+"ENDIF\n";
      texto=texto+"RUN "+prog+"\n";
      ProgramGroups.createNewProgram(nprog,texto);
    }
  }
```

Existen dos formas de vigilar el cambio de estado de una variable en el comando TRIGGER.

Si se incluye el estado, el programa que vigila el estado indicado de la variable IN o OUT es el siguiente:

```
WAIT IN/OUT[n]=<estado>
RUN PROG
```

El programa creado tomará el nombre siguiente: TRIGGER_<IN>/<OUT>_<n>_<estado>.

Si no se incluye estado, el programa que vigila el cambio de estado de la variable IN o OUT es el siguiente:

```
IF IN/OUT[n]=1
```

```
WAIT IN/OUT[n]=0
ELSE
WAIT IN/OUT[n]=1
ENDIF
RUN PROG
```

El programa creado tomará el nombre siguiente: TRIGGER_<IN>/<OUT>_<n>_CAMBIO.

PEND

formato: *PEND <var1> FROM <var2>*
modo: editor

El comando PEND realiza una espera activa sobre el valor de la variable <var2>. La espera concluye una vez recibido un valor distinto de cero en <var2>, este valor se copia a <var1>.

La espera activa ha sido simulada de igual manera que el comando WAIT, es decir, al finalizar la ejecución de la orden no se actualiza el contador de programa mientras no venga un valor distinto de cero en <var2>.

Debemos recordar aquí como se realiza la espera activa. La espera se consigue solamente lanzando al final del procedimiento una excepción WAITCONTINUE, la cual hace que no se actualice al contador de programa. Para más detalles de la excepción ver apartado número 9.1.1.3.

```
  public static void doPEND(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    Vector q=ParamParser.eval(line.toUpperCase());

    if (q.size()!=3) throw (Errores.PENDformato);
    if (!q.get(1).equals("FROM")) throw (Errores.PENDformato);

    String var2=(String) q.get(2);
    String var1=(String) q.get(0);

    int val2=SimpleParser.eval(var2,parent);
    //si val2==0 esperaremos
    //si es distinto de cero lo mandamos a var1
    if (val2!=0) {
      //Asignacion.doSET(var1+"="+val2,parent,modo);
      String[] left=CommonTools.getNameAndIndex(var1);
      if (left[1]==null) Simples.set(left[0],val2,parent);
      else
Tablas.set(left[0],SimpleParser.eval(left[1],parent),val2,parent);
    }
    else //para repetir la instrucción hasta que se de la contraria
      throw (Errores.WAITCONTINUE);
  }
```

POST
formato: *POST <var3> TO <var2>*
modo: editor

Este comando ha sido simulado como si se tratase de una operación con el comando SET. Debe ser usado en conjunción con el comando PEND, ya que éste queda en espera del cambio de valor de la variable <var2>. El comando POST realiza ese cambio, asigna el valor de <var3> a <var2>, siendo <var2> la variable sobre la que se realiza la espera del comando PEND.

Realmente a sido simulado con una forma reducida del comando SET, equivale a hacer SET <var2>=<var3>.

```
  public static void doPOST(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    Vector q=ParamParser.eval(line.toUpperCase());

    if (q.size()!=3) throw (Errores.POSTformato);
    if (!q.get(1).equals("TO")) throw (Errores.POSTformato);

    String var2=(String) q.get(2);
    String var1=(String) q.get(0);

    int val=SimpleParser.eval(var1,parent);

    //Asignacion.doSET(var2+"="+var1,parent,modo);
    String[] left=CommonTools.getNameAndIndex(var2);
    if (left[1]==null)
      Simples.set(left[0],val,parent);
    else
      Tablas.set(left[0],SimpleParser.eval(left[1],parent),val,parent);
  }
```

QPEND
formato: *QPEND <var1> FROM <var2>*
modo: editor

Los dos comandos siguientes funcionan de igual manera que PEND y POST, la diferencia radica en que <var2> no es una variable simple sino una tabla. QPEND realiza una espera sobre <var2> esperando la llegada de un valor distinto de cero.

El vector <var2> funciona a modo de cola FIFO. Este funcionamiento se consiguió haciendo que siempre se lea el vector desde el principio y cada vez que se retire un elemento del vector se mueven todos los elementos del vector una posición hacia el comienzo. De esta manera se consigue que los valores entrantes vayan al final y siempre se lean los valores más antiguos.

Este comando realiza una espera activa sobre el vector, de manera que si no existiesen valores distintos de cero en el vector el programa se detiene. Para realizar la comprobación nos limitamos a comprobar la primera posición, ya que la última vez que fue retirado un valor de la cola fueron copiados todos los elementos una posición hacia el comienzo.

La espera activa se realiza de igual manera que el comando WAIT, es decir lanzando una excepción del tipo WAITCONTINUE.

```
  public static synchronized void doQPEND(String line,String parent,int
modo) throws Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    Vector q=ParamParser.eval(line.toUpperCase());

    if (q.size()!=3) throw (Errores.QPENDformato);
    if (!q.get(1).equals("FROM")) throw (Errores.QPENDformato);

    String var2=(String) q.get(2);

    if (!Tablas.contains(var2,"GLOBAL"))
      throw (Errores.NeedsGLOBAL);

    String var1=(String) q.get(0);

    if (Tablas.get(var2,1,"GLOBAL")==0) {
      throw (Errores.WAITCONTINUE);
    }
    else {
      int val=Tablas.get(var2,1,"GLOBAL");
      int tam=Tablas.getSize(var2,"GLOBAL");
      //Asignacion.doSET(var1+"="+val,parent,modo);
      String[] left=CommonTools.getNameAndIndex(var1);
      if (left[1]==null) Simples.set(left[0],val,parent);
      else
Tablas.set(left[0],SimpleParser.eval(left[1],parent),val,parent);

      Tablas.set(var2,tam,0,"GLOBAL");
      for (int i=2;i<=tam;i++)
        Tablas.set(var2,i-1,Tablas.get(var2,i,"GLOBAL"),"GLOBAL");
    }
  }
```

QPOST

formato: *QPOST <var3> TO <var2>*
modo: editor

Este comando debe introducir el valor de <var3> en la cola dada por el vector <var2>. Si no existen posiciones libres en la cola debe realizar una espera sobre la cola hasta que queda alguna.

Para introducir un valor sólo debemos recorrer la tabla hasta que lleguemos a una posición con un valor igual a cero, lugar donde colocaremos el nuevo valor.

```
  public static synchronized void doQPOST(String line,String parent,int
modo) throws Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    Vector q=ParamParser.eval(line.toUpperCase());

    if (q.size()!=3) throw (Errores.QPOSTformato);
    if (!q.get(1).equals("TO")) throw (Errores.QPOSTformato);

    String var2=(String) q.get(2);
    if (!Tablas.contains(var2,"GLOBAL"))
      throw (Errores.NeedsGLOBAL);

    String var1=(String) q.get(0);

    int tam=Tablas.getSize(var2,"GLOBAL");

    if (Tablas.get(var2,tam,"GLOBAL")!=0) //si esta llena debemos esperar
      throw (Errores.WAITCONTINUE); //repite la última instrucción
    else {
      int pos=1;
      while (pos<=tam && Tablas.get(var2,pos,"GLOBAL")!=0) {
        pos++;
      }

      Tablas.set(var2,pos,SimpleParser.eval(var1,parent),"GLOBAL");
    }
  }
```

10.2.5 Bucles, saltos y condiciones

En este apartado podemos considerar los comandos del tipo IF, GOTO, LABEL, y FOR. Estos comandos son los responsables de que los programas en ACL no sean simplemente una lista secuencial de órdenes que se ejecutan, sino que permiten realizar saltos condicionados, saltos incondicionales, bucles, etc...

Estos comandos sólo funcionan en modo editor, ya que en modo directo no existen los números de línea, es decir, no existe un apuntador (número de línea) que indique la dirección en la que se ejecutan los comandos.

IF/ANDIF/ORIF ... ELSE ... ENDIF

formato:

IF <cond1>
ANDIF/ORIF <cond2>...
conjunto1 de comandos
ELSE
conjunto2 de comandos

ENDIF
modo: editor

Esta estructura fundamenta su funcionamiento en la clase *CompParser*. La clase *CompParser* es la encargada de realizar las operaciones en las que intervengan operadores relacionales o de comparación, y dará dos posibles resultados: verdadero o falso. Para más detalles sobre *CompParser* consultar el apartado número 8.3.2.

En esta estructura se definen dos conjuntos posibles de instrucciones, el primero de los cuales se ejecutará si se cumplen las condiciones dadas por los comandos IF y ANDIF/ORIF. Si el conjunto de operaciones no diese un resultado verdadero se ejecutaría el segundo conjunto.

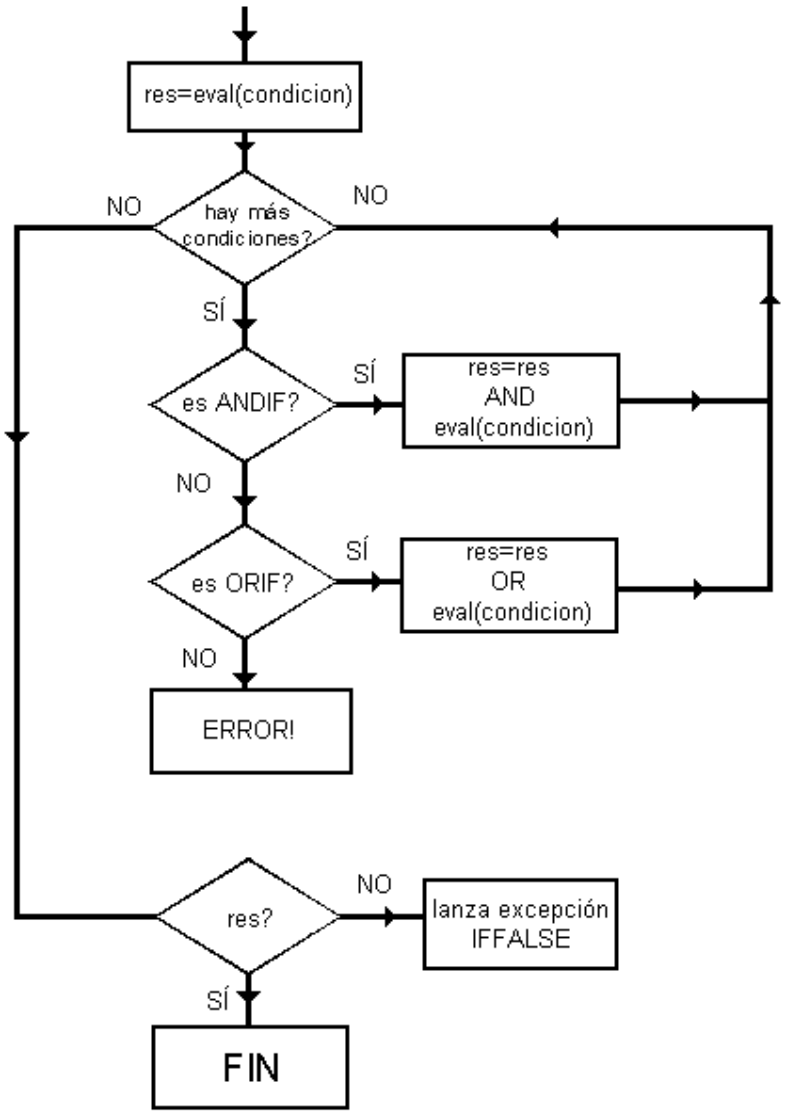
Para la comprobación de las condiciones, el procedimiento necesita que le sea dado como entrada el conjunto total de líneas que componen las condiciones, es decir el comando IF y los siguientes. De una forma más clara se tomará el conjunto de operaciones de comparación como una sola.

Por ejemplo una sentencia como la siguiente: IF A>0, ANDIF B>0, ORIF C>0. Es mandada a este comando del modo siguiente "A>0$ANDIF B>0$ORIF C>0". De realizar este cambio se encarga el procedimiento *runProgram()* explicado en el apartado número 9.1.1.3.

Al finalizar las operaciones podemos tener dos resultados posibles: lanzar una excepción del tipo IFFALSE o no hacer nada. Si se lanza una excepción IFFALSE quiere decir que no se ha cumplido el conjunto de condiciones, por tanto debemos buscar el segundo conjunto de comandos o el final. Para buscarlo sólo debemos ir al siguiente comando ELSE o ENDIF, mediante la función *searchELSEorENDIF()* (ver apartado nº 9.1.1.4).

El otro resultado posible es no dar ningún resultado. Esto trae como consecuencia que el programa siga su curso normal ejecutando el primer conjunto de operaciones.

El diagrama de flujo que sigue este código para interpretar las sentencias IF es el siguiente:




```
  public static void doIF(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);

    line=line.toUpperCase()+"$";
    int sep=line.indexOf('$');

    String cond=line.substring(0,sep); //el primero es solo condicion
    String op;
    boolean encontrado=false;
    boolean resultado=CompParser.eval(cond,parent);
    line=line.substring(sep+1); //quitamos lo ya hecho

    while (!encontrado) {
      sep=line.indexOf('$');
      if (sep==-1) encontrado=true; //ya no hay mas
      else {
```

```
        cond=line.substring(0,sep);
        int spindex=0;
        while (spindex<cond.length() &&
          !Character.isWhitespace(cond.charAt(spindex))) spindex++;

        if (spindex==cond.length()) {
          if (cond.startsWith("ANDIF")) throw (Errores.ANDIFformat);
          else throw (Errores.ORIFformat);
        }

        op=cond.substring(0,spindex); //el comando

        if (op.equals("ANDIF"))
          resultado=resultado &&
(CompParser.eval(cond.substring(spindex+1).trim(), parent));
        else //es ORIF
          resultado=resultado ||
(CompParser.eval(cond.substring(spindex+1).trim(), parent));

        line=line.substring(sep+1); //quitamos lo ya hecho
      }
    }

    //si la condicion se cumple no le decimos nada para que el programa
    //siga su flujo normal, si no se cumple debe buscar ENDIF
    if (!resultado) throw (Errores.IFFALSE);
  }
```

Debido a que no podemos encontrarnos comandos ANDIF y ORIF sin su correspondiente IF, si se encontrase alguno se debe generar un error. Debemos recordar que si se ha interpretado completamente una sentencia del tipo IF con sus correspondientes ANDIF y ORIF, no se puede encontrar un comando de ese tipo suelto ya que el conjunto IF/ANDIF/ORIF es interpretado como un solo comando.

Si se ejecutó el primer conjunto de operaciones, podemos encontrarnos con dos posibles instrucciones finales: ELSE o ENDIF. El segundo caso es de fácil tratamiento ya que no debemos hacer nada. Este caso también nos lo encontraremos si se ejecutó el segundo conjunto de instrucciones.

Por otro lado, si la última instrucción es ELSE debemos saltarnos el segundo conjunto de instrucciones y buscar el comando ENDIF para salir de la estructura IF...ELSE...ENDIF. Para realizar esto se lanza una excepción del tipo ELSEM que lanzará una búsqueda de ENDIF mediante la función *searchENDIF()*.

```
  public static void doELSE(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    throw (Errores.ELSEM);
```

```
  }
```

En este caso no puede darse el error de que nos encontremos un comando ELSE fuera de su correspondiente sentencia IF..ENDIF ya que antes de comenzar la ejecución del programa se realiza la comprobación de consistencia explicada en el apartado número 9.1.1.1.

FOR ... ENDFOR

formato:

FOR <var1>=<n1> TO <n2>
...conjunto de comandos...
ENDFOR

modo: editor

Los bucles FOR son tratados de una forma especial como ya ha sido explicado en el apartado número 9.1.1.5. Por otra parte aquí será tratada la función que realizan los comandos FOR y ENDFOR para conseguir el resultado final del bucle.

El comando FOR realiza dos acciones principalmente, actualizar el valor de la variable índice y comprobar que todavía ese valor se encuentra entre los valores inicial y final.

Este código permite las dos posibilidades en los argumentos <n1> y <n2>, es decir, <n1> menor que <n2> en cuyo caso será incrementada la variable índice, y <n1> mayor que <n2> en el que será disminuida.

Debido a que al hacer el cambio en la variable índice justo al principio del bucle ejecutamos una vez más el bucle de la deseada, la variable índice toma el valor inicial menos uno (en el caso <n1> menor que <n2>) antes de entrar en el bucle FOR (esto se realiza incluyendo automáticamente unas sentencias al comenzar la ejecución del programa, ver apartado número 9.1.1.5). Entonces al ejecutar el comando FOR sólo debemos comparar que la variable índice sea distinta del valor final dado por <n2>.

En caso de que se haya verificado la condición <var1> igual a <n2> debemos finalizar el bucle, para ello se lanza una excepción del tipo ENDFORM. Esta excepción será entendida como un mensaje al hilo *ProgramThread* correspondiente y provocará una actualización del contador de programa con la posición del comando ENDFOR correspondiente. Para realizar esta actualización del contador se hará uso de la función *searchENDFOR()*.

```
  public static void doFOR(String linea,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
```

```
    Vector q=ParamParser.eval(linea.toUpperCase());

    String ini=ParamParser.ToolJoinEx(q,0,"TO");
    String end=ParamParser.ToolJoinEx(q,"TO",q.size());

    if (ini==null||end==null) throw (Errores.FORformat);

    int eqindex=ini.indexOf('=');

    if (eqindex==-1) throw (Errores.FORformat);

    String vari=ini.substring(0,eqindex);
    String begin=ini.substring(eqindex+1);

    int i1=SimpleParser.eval(begin,parent);
    int i2=SimpleParser.eval(end,parent);
    int valor=SimpleParser.eval(vari,parent);

    if (valor!=i2) {
      String[] left=CommonTools.getNameAndIndex(vari);
      if (i1<=i2) {
        //Asignacion.doSET(vari+"="+vari+"+1",parent,modo);
        if (left[1]==null) Simples.set(left[0],valor+1,parent);
        else Tablas.set(left[0], SimpleParser.eval(left[1],parent),
valor+1, parent);
      }
      else {
        //Asignacion.doSET(vari+"="+vari+"-1",parent,modo);
        if (left[1]==null) Simples.set(left[0],valor-1,parent);
        else Tablas.set(left[0], SimpleParser.eval(left[1],parent), valor-
1, parent);
      }
    }
    else throw (Errores.ENDFORM);
  }
```

Por otra parte, el comando ENDFOR sólo lanza una excepción del tipo REPEATFOR. Esta excepción provocará que se actualice el contador de programa con el último valor almacenado en la pila de índices para los comandos FOR. La pila dada por la variable *forstack* local a una instancia de *ProgramThread*. Con esta actualización conseguiremos volver al principio del bucle FOR correspondiente.

```
  public static void doENDFOR(String linea,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    throw (Errores.REPEATFOR);
  }
```

GOTO
formato: *GOTO <nnnn>*
modo: editor

Este comando realiza una comprobación sobre la validez de la etiqueta, si es correcta lanza una excepción que será recogida por el objeto *ProgramThread* correspondiente. Esta excepción hará que el contador de programa se actualice con la posición de la etiqueta definida con el comando LABEL, para lo cual será lanzada una búsqueda de esa etiqueta mediante la función *searchLABEL()* (ver apartado número 9.1.1.4).

```
  public static void doGOTO(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    int nlabel;

    try {
      nlabel=Integer.parseInt(line);
    } catch (NumberFormatException err) {
      throw (Errores.GOTOformato);
    }

    if (nlabel<0 || nlabel>9999)
      throw (Errores.GOTOformato);

    throw new Exception(":"+nlabel);
  }
```

LABEL
formato: *LABEL <nnnn>*
modo: editor

Este comando no realiza ninguna función. Realmente en el lenguaje ACL no genera ninguna línea de comando, sólo genera en la memoria destinada a cada programa una entrada que asocia a una posición de memoria un número (a un número de línea le asignará la etiqueta que servirá para desplazar el contador de programa a dicha posición de memoria).

Debido a que en el simulador no existen números de línea y que el programa se interpreta a medida que se ejecuta, las etiquetas definidas con el comando LABEL sí generan una línea de programa. En la máquina real las etiquetas son anotadas en una tabla de etiquetas en tiempo de diseño del programa.

Este comando sólo realiza una comprobación de la validez del número de etiqueta, es decir la etiqueta debe estar comprendida entre los números 0 y 9999.

Para más detalles sobre el funcionamiento de las etiquetas consultar el apartado número 9.1.1.4.

GOSUB
formato: *GOSUB <prog>*
modo: editor

Este comando hace que se ejecute un programa ya cargado, mientras este programa se está ejecutando el programa que generó la llamada con el comando GOSUB está detenido.

Para conseguir este resultado desde este procedimiento se realiza la llamada al programa dado por <prog> para lanzar su ejecución y también se lanza una excepción. Esta excepción nos servirá para indicar al objeto *ProgramThread* correspondiente al programa en curso que debe detener su ejecución hasta que termine el programa llamado.

```
  public static void doGOSUB(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    String name=line.toUpperCase();
    Sincronizacion.doRUN(name,parent,modo);
    //le decimos que tiene que esperar hasta que termine el programa
    throw new Exception("@"+name);
  }
```

El hilo *ProgramThread* que realizó la sentencia GOSUB detiene su ejecución de la siguiente manera:

```
do {
  sleep(100);
} while (!ProgramGroups.hasFinishedThread(prog));
```

Esto es, se realiza una espera activa sobre el estado de ejecución del programa dado por el parámetro <prog>. Realmente no es una espera activa, la diferencia estriba en el comando interior del bucle *sleep(100)*. Este comando provoca que el hilo duerma durante 100 milisegundos para después comprobar otra vez el estado del programa. De este modo dejamos a otros programas, entre ellos el que hemos lanzado con GOSUB, que se ejecuten con tiempo de CPU disponible.

10.2.6 Comandos para generar movimientos del modelo del robot

Los comandos que permiten al modelo realizar movimientos son simples intermediarios entre los programas, las instancias de *ProgramThread*, y la clase que se encarga de realizar los movimientos del robot, la clase *MovimientoRobot*.

OPEN/CLOSE

formato:

OPEN

CLOSE

modos: todos

Los procedimientos que sirven para gestionar los comandos OPEN y CLOSE sólo realizan una llamada al correspondiente procedimiento *openEfector()* y *closeEfector()* del objeto *MovimientoRobot*. Para más detalles sobre el objeto *MovimientoRobot* consultar el apartado número 7.

```
  public static void doOPEN(String resto,String parent,int modo) throws
Exception {
    if (resto.length()>0) throw (Errores.OPENformat);
    MovimientoRobot.openEfector();
  }

  public static void doCLOSE(String resto,String parent,int modo) throws
Exception {
    if (resto.length()>0) throw (Errores.CLOSEformat);
    MovimientoRobot.closeEfector();
  }
```

MOVE/MOVEL

formato:

MOVE <pos> [<tiempo>]

MOVEL <pos> [<tiempo>]

modos: todos

Los comandos de generación de trayectorias (MOVE, MOVES, MOVEL, ...) hacen uso de un procedimiento llamado *doMOVEboth()*. Este proceso es el que se encarga de generar las llamadas a la clase *MovimientoRobot*. Acepta además un argumento que nos sirve para identificar si se tiene que generar una trayectoria lineal en coordenadas cartesianas o articulares.

Si incluimos el parámetro de entrada <tiempo> en los comandos de movimiento, el programa continuará la ejecución sólo cuando haya transcurrido el tiempo indicado tras la ejecución de la instrucción de movimiento. Este parámetro sólo es aceptado en modo editor, ya que no se pueden realizar esperas en modo directo.

```
  public static void doMOVE(String resto,String parent,int modo) throws
Exception {
    doMOVEboth(resto,parent,modo,false);
  }
```

Por otro lado, el comando MOVEL realiza una llamada al mismo procedimiento con la única diferencia del último argumento. Con este argumento le indicamos que debe realizar la llamada al generador de trayectorias lineales en el espacio cartesiano y no en el articular como en el contrario.

```
  public static void doMOVEL(String resto,String parent,int modo) throws
Exception {
    try {
      doMOVEboth(resto,parent,modo,true);
    } catch (Exception err) {
      if (err==Errores.MOVEformat) throw (Errores.MOVELformat);
      else throw (err);
    }
  }
```

El procedimiento *doMOVEboth()* es el encargado de llevar la posición destino al generador de trayectorias. Aquí se diferencia la generación de trayectorias lineales en el espacio articular del espacio cartesiano con el parámetro de entrada *lineal*.

Además debemos diferenciar los distintos tipos de posiciones que existen (ver apartado número 8.2.1) ya que los generadores sólo entienden posiciones dadas en coordenadas articulares. Por tanto, las posiciones dadas en coordenadas cartesianas, sean o no relativas a otras, debemos convertirlas a coordenadas articulares.

Directamente o mediante conversión las coordenadas finales serán guardadas en la variable *joints*, la cual será entregada al objeto *MovimientoRobot* a través del procedimiento *MovimientoRobot.setJointsFinalAbsoluteVal(joints,lineal)*. Este será el procedimiento encargado de llamar directamente al generador.

```
  private static void doMOVEboth(String resto,String parent,int
modo,boolean lineal) throws Exception {
    Vector q=ParamParser.eval(resto.toUpperCase());
    int tiempo=0;

    if (q.size()!=1 && q.size()!=2)
      throw (Errores.MOVEformat);

    String[] posicion=CommonTools.getNameAndIndex((String) q.get(0));

    if (q.size()==2) { //tiempo de espera sólo modo editor
      CommonTools.checkModo(modo,Const.EDITOR);
      try {
        tiempo=Integer.parseInt((String) q.get(1));
      } catch (NumberFormatException err) {
        throw (Errores.MOVEformat);
      }
```

```
      if (tiempo<0)
        throw (Errores.MOVEformat);
    }

    int[] xyz=null;
    int[] joints=null;
    int[] pr=null;
    int tipo;

    if (posicion[1]==null) {
      tipo=Posiciones.getSTATUS(posicion[0]);
      switch (tipo) {
        case Const.ST_PABS:
          joints=Posiciones.getJoints(posicion[0]);
          break;
        case Const.ST_PUNDEF:
          throw (Errores.UNDEFposicion);
        case Const.ST_PREL_JACTU: case Const.ST_PREL_JOTRA:
          joints=Posiciones.getJointsRelativa(posicion[0]);
          break;
        case Const.ST_PREL_CACTU: case Const.ST_PREL_COTRA:
          xyz=Posiciones.getXYZRelativa(posicion[0]);
          pr=Posiciones.getPRRelativa(posicion[0]);
          break;
      }
    }
    else {
      int pos=SimpleParser.eval(posicion[1],parent);
      tipo=TPosiciones.getSTATUS(posicion[0],pos);
      switch (tipo) {
        case Const.ST_PABS:
          joints=TPosiciones.getJoints(posicion[0],pos);
          break;
        case Const.ST_PUNDEF:
          throw (Errores.UNDEFposicion);
        case Const.ST_PREL_JACTU: case Const.ST_PREL_JOTRA:
          joints=TPosiciones.getJointsRelativa(posicion[0],pos);
          break;
        case Const.ST_PREL_CACTU: case Const.ST_PREL_COTRA:
          xyz=TPosiciones.getXYZRelativa(posicion[0],pos);
          pr=TPosiciones.getPRRelativa(posicion[0],pos);
          break;
      }
    }

    if (tipo==Const.ST_PREL_CACTU||tipo==Const.ST_PREL_COTRA) {
      //es relativa en coordenadas cartesianas
      //debemos coger las coordenadas absolutas xyz
      //y convertirlas a articulares absolutas
      double[] pxyz=ModeloConv.xyz2Cart(xyz);
      double[] ppr=ModeloConv.pr2Cart(pr);

      ModeloCinematicoInverso.updateJoints(pxyz,ppr);
      joints=ModeloConv.Rad2Pulses(ModeloCinematicoInverso.getJoints());
    }

    MovimientoRobot.setJointsFinalAbsoluteVal(joints,lineal);
    if (tiempo>0)
```

```
      throw new Exception("$DELAY-"+(tiempo*10));
  }
```

MOVED/MOVELD

formato:

MOVED <pos> [<tiempo>]

MOVELD <pos> [<tiempo>]

modo: editor

Los comandos de movimiento terminados en la letra 'D', además de generar la correspondiente trayectoria de movimiento, deben realizar una espera hasta que se alcance la posición final. Esta espera no se realiza en el procedimiento al que se llama. Podemos observar que realmente no existe diferencia entre el comando MOVED y el comando MOVE. Esto es así debido a que la diferencia no se encuentra aquí, sino que antes de ejecutar el programa se introduce un comando de forma automática entre el comando MOVED y el siguiente. Este comando no puede ser otro que el adecuado para realizar esperas activas, es decir el comando WAIT.

La espera activa se realiza sobre la variable MOVING, la cual toma un valor igual a 1 cuando el robot está en movimiento y 0 en caso contrario.

Por otro lado debemos observar como son manejados los errores de forma que sea transparente al usuario el hecho de que realmente está haciendo uso del comando MOVE. Si se produjese alguna excepción esta sería recogida aquí y cambiada por la correspondiente al comando MOVED.

```
  public static void doMOVED(String resto,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.EDITOR);
    try {
      doMOVEboth(resto,parent,modo,false);
    } catch (Exception err) {
      if (err==Errores.MOVEformat) throw (Errores.MOVEDformat);
      else throw (err);
    }
  }
```

MOVES

formato: *MOVES <pvect> <ini> <fin> [<tiempo>]*

modos: todos

El comando MOVES genera una secuencia de movimientos dada por el vector de posiciones <pvect>. Esta secuencia tendrá como posición inicial la dada por <pvect>[<ini>] y finalizará con la posición <pvect>[<fin>].

De forma similar al comando MOVE lo que realiza es una serie de llamadas al generador de trayectorias con las nuevas posiciones que deben ser alcanzadas. La complicación viene de que hay que distinguir los casos <ini> menor que <fin>, e <ini> mayor que <fin>. Estos casos harán que la secuencia de movimientos se ejecute de manera directa, en el primer caso, e inversa en el segundo. Además se debe realizar una serie de comprobaciones para verificar que los índices se encuentran dentro de la tabla.

```
  public static void doMOVES(String resto,String parent,int modo) throws
Exception {
    Vector q=ParamParser.eval(resto.toUpperCase());
    int tiempo=0;

    if (q.size()!=3 && q.size()!=4)
      throw (Errores.MOVESformat);

    if (q.size()==4) { //sólo para modo editor
      CommonTools.checkModo(modo,Const.EDITOR);
      try {
        tiempo=Integer.parseInt((String) q.get(3));
      } catch (NumberFormatException err) {
        throw (Errores.MOVESformat);
      }
      if (tiempo<0)
        throw (Errores.MOVESformat);
    }
    String vector=(String) q.get(0);
    String expre1=(String) q.get(1);
    String expre2=(String) q.get(2);

    int index1=SimpleParser.eval(expre1,parent);
    int index2=SimpleParser.eval(expre2,parent);

    //comprobación de índices
    if (index1<1 || index2<1 || index2>TPosiciones.getSize(vector) ||
index1>TPosiciones.getSize(vector))
      throw (Errores.IndiceFueraRango);

    int[] joints;
    int[] xyz;

    if (index1<=index2) {
      for (int i=index1;i<=index2;i++) {
        //leemos la posición
        if (TPosiciones.isAbsoluta(vector,i)) {
          xyz=TPosiciones.getXYZ(vector,i);
          joints=TPosiciones.getJoints(vector,i);
        }
        else {
          xyz=TPosiciones.getXYZRelativa(vector,i);
          joints=TPosiciones.getJointsRelativa(vector,i);
        }
        //y la mandamos a la cola de generación
```

```
        MovimientoRobot.setJointsFinalAbsoluteVal(joints,false);
      }
    }
    else { //hay que ejecutarla al revés index1>index2
      for (int i=index1;i>=index2;i--) {
        //leemos la posición
        if (TPosiciones.isAbsoluta(vector,i)) {
          xyz=TPosiciones.getXYZ(vector,i);
          joints=TPosiciones.getJoints(vector,i);
        }
        else {
          xyz=TPosiciones.getXYZRelativa(vector,i);
          joints=TPosiciones.getJointsRelativa(vector,i);
        }
        //y la mandamos a la cola de generación
        MovimientoRobot.setJointsFinalAbsoluteVal(joints,false);
      }
    }


    if (tiempo>0) //tiempo de espera
      throw new Exception("$DELAY-"+(tiempo*10));
  }
```

Posibles errores: número de parámetros incorrecto, índice fuera de rango en el vector, vector inexistente.

MOVESD

formato: *MOVESD <pvect> <ini> <fin> [<tiempo>]*
modo: editor

Este comando, al igual que MOVED y MOVELD, es tratado antes de la ejecución del programa. Solamente realiza una llamada al comando MOVES y realiza un tratamiento de los errores al finalizar la ejecución del comando. El objeto *ProgramThread* se encargará de detener la ejecución del programa mientras la variable MOVING valga 1.

HOME

formato: *HOME*
modos: todos

En la máquina real este comando sirve para llevar al robot a la posición inicial. Además elimina todos los errores que se hayan podido provocar al realizar los movimientos, como los errores del tipo "AxisLimit reached".

Aquí se ha optado por llevar al robot a una posición conocida dada por la posición HOMEPOS. Dicha posición es creada al iniciar el simulador. Por tanto, la implementación ha sido simple, sólo hay que realizar una llamada al comando

de movimiento con esa posición. Es decir, el comando HOME equivale a hacer MOVE HOMEPOS.

Además ha sido incluido un mensaje que indica que no es un movimiento como otro cualquiera, sino que es un movimiento de inicialización. Dicho mensaje también existe en la máquina real.

Por otro lado, el comando HOME es tratado de forma distinta en cada modo. En el modo directo se limita a ejecutar la acción. Sin embargo, en modo editor, cuando el objeto *ProgramThread* se encuentra una línea con la sentencia HOME la ejecución se detiene hasta que haya finalizado el movimiento desde la posición actual hasta HOMEPOS. Esta pausa en la ejecución es realizada de la misma forma que los comandos MOVED, MOVELD, etc.

SPEED
formato: *SPEED <n>*
modos: todos

Este comando sirve de intermediario entre el usuario y la variable *speed*, local al objeto *MovimientoRobot*. Esta variable permite ajustar el número de pasos que se darán para llegar a la próxima posición, de esta forma será simulada la velocidad. Es decir, a menor número de pasos mayor velocidad.

Para más detalles sobre el funcionamiento de la variable speed consultar los generadores de trayectorias en el apartado 7.2.

10.2.7 Otros comandos

En esta categoría están los comandos que no han podido ser introducidos en ninguna de las anteriores. Estos comandos son ENABLE y DISABLE.

ENABLE/DISABLE
formato:
ENABLE IN/OUT <num>
DISABLE IN/OUT <num>
modo: directo

Estos comandos permiten habilitar las entradas y salidas digitales. En el caso de una entrada, esto quiere decir que si no estuviera habilitada, no reflejaría los cambios que se produzcan.

Sólo será presentado el código del comando ENABLE puesto que el de DISABLE es muy similar. La única diferencia radica en el valor lógico que asignan a la variable que controla la disponibilidad de las entradas y salidas digitales.

```
  public static void doENABLE(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.DIRECTO);

    Vector q=ParamParser.eval(line.toUpperCase());
    if (q.size()!=2)
      throw (Errores.ENABLEformato);

    String tabla=(String) q.get(0);
    String numstr=(String) q.get(1);

    int num;
    try {
      num=Integer.parseInt(numstr);
    } catch (NumberFormatException err) {
      throw (Errores.EnteroNoValido);
    }

    if (num>=1 && num<=Const.NUMES) {
      if (tabla.equals("IN")) Const.INsDisabled[num-1]=false;
      else if (tabla.equals("OUT")) Const.OUTsDisabled[num-1]=false;
      else throw (Errores.ENABLEformato);
    }
    else throw (Errores.ENABLEformato);
  }
```

CON/COFF

formato:

CON [<num>]

COFF [<num>]

modo: directo

Estos comandos permiten habilitar los motores de cada una de las articulaciones. Esto quiere decir que los valores almacenados en el buffer de movimiento no serán llevados al modelo del robot, en el caso de no estar habilitados.

Para ello modifica el valor de una variable lógica asociada a cada buffer de movimiento de cada articulación. Sólo será mostrado el código de CON puesto que el de COFF es similar, la única diferencia es el valor lógico asignado a la variable.

```
  public static void doCON(String line,String parent,int modo) throws
Exception {
    CommonTools.checkModo(modo,Const.DIRECTO);
    Vector q=ParamParser.eval(line.toUpperCase());

    if (q.size()==0) { //activar todos
```

```
      for (int i=0;i<Const.NUMJOINTS;i++)
        Const.SERVOSEnabled[i]=true;
    }
    else if (q.size()==1) { //activar sólo uno
      String numstr=(String) q.get(0);
      int num;
      try {
        num=Integer.parseInt(numstr);
      } catch (NumberFormatException err) {
        throw (Errores.EnteroNoValido);
      }

      if (num>=1 && num<=Const.NUMJOINTS) Const.SERVOSEnabled[num-1]=true;
      else throw (Errores.CONformato);
    }
    else throw (Errores.CONformato);
  }
```

11 BIBLIOGRAFÍA Y MATERIAL COMPLEMENTARIO

Para la realización de este proyecto ha sido utilizado el siguiente material:

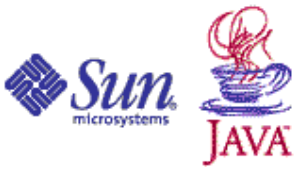
- ✓ "Programación concurrente en Java. Principios y patrones de diseño". Doug Lea (681.3 d.3 j lea)

- ✓ ACL. Advanced Control Language. Reference Guide (third edition).

- ✓ "Data structures and algorithms in Java (second edition)". Michael T. Goodrich & Roberto Tamassia (681.3 d.3 j goo)

Recursos en Internet:

- ✓ Java 2 SDK 1.3. http://java.sun.com



- ✓ The Java 3D Community. http://www.j3d.org