Indice general

Indice general I

Indice de figuras V

0. Antecedentes 1

1. La célula de fabricación. 3

1.1 Tipos de piezas 3
PIEZAS SIN TAPA 4
PIEZAS CON TAPA 5
1.2 Zona de Fabricación 6
1.3 Topología del bus 8

2. Interbus 10

2.1 Características básicas. 10
2.2 Topología 11
2.3 Interfaz de usuario 11
2.4 Resumen de las capacidades de Interbus 12
2.4 La tarjeta controladora Interbus IBS PCI SC/I-T 13
2.5 High-Level Language Interface (HLI) 16
2.6 Configuración de las estaciones mediante CMD 18
Estación 1 19
Estación 3 20
Estación 4 22
Estación 6 23
Transporte 24

3. Lenguaje Java 26

3.1 ¿Qué es Java? 26
3.2 Nociones básicas de Java. 28
3.2.1 Orientación a objetos 28
3.2.2 Concurrencia 29
3.2.3 Métodos nativos. JNI 31
3.3 Proceso para usar Java con Interbus 32
Paso 1: Escribir el código Java. 33
Paso 2: Compilar el código Java. 33
Paso 3: Crear el archivo .h 33
Paso 4: Escribir la implementación del método nativo. 34
Paso 5: Crear una librería compartida. 38
Paso 6: Ejecutar el programa 38

4. Redes de Petri 39

4.1 Introducción 39

4.2 Principios fundamentales ... 40
4.3 Definiciones básicas. ... 41
4.4 Creación de las RdP del proyecto ... 45
4.5 Implementación programada de RdP ... 49
Estado ... 49
Transición ... 51
Red ... 52
Creación de una RdP directamente mediante código. ... 54
Creación de una RdP mediante lectura de un fichero de texto. ... 55
Conflicto ... 56
Coordinador ... 57

5. Código fuente ... 65

Paquete comun ... 65
Estacion.java ... 65
GestorBus.java ... 66
MarcoTest.java y MarcoTest_AcercaDe.java ... 66
Pedido.java ... 66
PruebaTiempoPalets.java ... 67
Test.java ... 67
TextoARed.java ... 67
Variable.java ... 67
VariableAnalogica.java ... 68
VariableBooleana.java ... 69
VariableMemoria.java ... 69
Paquete estacion1 ... 69
Estacion1.java ... 69
Estacion1Swing.java ... 70
MarcoEstacion1.java y MarcoEstacion1_AcercaDe.java ... 71
MonitorEst1 ... 71
Paquete estacion2 ... 72
CoordinadorEst2 ... 72
MonitorEst2 ... 72
Paquete estacion3 ... 72
AplicacionRdPE3.java ... 72
CoordinadorEstacion3.java ... 72
CoordinadorEstacion3PruebaTar.java ... 72
Estacion3.java ... 72
Estacion3Swing.java ... 72
MarcoEst3.java y MarcoEst3_AcercaDe.java ... 73
MarcoEstacion1.java y MarcoEstacion1_AcercaDe.java ... 73
MonitorEst3 ... 73
RedAutomaticoEstacion3 ... 73
RedResetEstacion3 ... 73
Paquete estacion4 ... 73
AplicacionRdPE3.java ... 73
CoordinadorEstacion4.java ... 73
CoordinadorEstacion4PruebaTar.java ... 73
Estacion4.java ... 74

Estacion4Swing.java 74
MarcoEst4.java y MarcoEst4_AcercaDe.java 74
MarcoEstacion1.java y MarcoEstacion1_AcercaDe.java 74
MonitorEst4 74
PruebaAnalogica 74
RedAutomaticoEstacion4 74
Paquete estacion6 74
Estacion6.java 75
Paquete excepciones 75
Paquete finales 75
AplicacionCelula.java 75
CoordinadorBorrar.java 76
CoordinadorCelula.java 76
FrameModoAutomatico.java 76
FrameModoManual 76
TestBorrar 77
Paquete paneles 77
BotonToggle 77
PanelEst1, PanelEst3, PanelEst4 y PanelTransporte 77
PanelEstadoEstacion 78
PanelGestionEstadoEstaciones 79
PanelMuestraPedidos 79
PanelNuevosPedidos 80
PanelSensor 80
Paquete puertoSerie 80
Comunicaciones 80
Palet 81
Trama 82

6. Aplicaciones 83

AplicacionCelula 83
PruebaTiempoPalets 88
Test 89
TextoARed 89
Estacion1Swing 90
AplicacionRdPE3 91
Estacion3Swing 92
AplicacionRdPE4 92
Estacion4Swing 92
PruebaAnalogica 93
TestBorrar 93

Conclusiones y futuras líneas de investigación 95

Bibliografía 96

Bibliografía escrita 96
Bibliografía en Internet 96

Indice de figuras

Figura 1: Disposición espacial de la célula de fabricación ... 3
Figura 2: Tipos de piezas ... 4
Figura 3: Esquema de una pieza ... 4
Figura 4: Tipos de émbolo ... 5
Figura 5: Estación 1 ... 6
Figura 6: Estación 2 ... 7
Figura 7: Estación 3 ... 7
Figura 8: Estación 4 ... 8
Figura 9: TSX Momentum ... 9
Figura 10: Módulos Interbus ... 9
Figura 11: Esquema de la tarjeta IBS PCI SC/I-T ... 14
Figura 12: Tarjeta IBS PCI SC/I-T ... 14
Figura 13: CMD ... 16
Figura 14: Pantalla principal CMD ... 18
Figura 15: Como exportar una configuración ... 19
Figura 16: Configuración estación 1 ... 20
Figura 17: Process data estación 1 ... 20
Figura 18: Configuración estación 3 ... 21
Figura 19: Process data estación 3 ... 21
Figura 20: Configuración estación 4 ... 22
Figura 21: Process data estación 4 ... 23
Figura 22: Configuración estación 6 ... 23
Figura 23: Process data estación 6 ... 24
Figura 24: Configuración transporte ... 24
Figura 25: Process data transporte ... 25
Figura 26: Pasos del JNI ... 32
Figura 27: Nomenclatura de métodos JNI ... 34
Figura 28: Nomenclatura de clases JNI ... 34
Figura 29: Tipos de lugares, arcos y transiciones ... 40
Figura 30: RdP ejemplo ... 42
Figura 31: RdP Conforme ... 43
Figura 32: RdP no binaria ... 44
Figura 33: RdP no viva ... 44
Figura 34: Sacar palets ... 46
Figura 35: RdP para sacar palet de las estaciones ... 47
Figura 36: RdP ejemplo ... 54
Figura 37: Algoritmo coordinador 1 ... 62
Figura 38: Algoritmo coordinador 2 ... 64
Figura 39 BotonToggle activado y desactivado ... 77
Figura 40 PanelTransporte ... 78
Figura 41 PanelEstadoEstacion ... 78
Figura 42 PanelGestionEstadoEstaciones ... 79
Figura 43 Tabla de pedidos ... 79
Figura 44 Panel de nuevos pedidos ... 80
Figura 45 Cuatro instancias de PanelSensor ... 80
Figura 46: Pantalla principal aplicación principal ... 83
Figura 47: Pantalla control manual ... 84
Figura 48: Pantalla control manual 2 ... 85

Figura 49: Cambio de automático a manual 86
Figura 50: Pantalla modo automático 87
Figura 51: Prueba de tiempos de palets 88
Figura 52: Test estaciones 3 y 4 89
Figura 53: TextoARed 90
Figura 54: Estacion1Swing 91
Figura 55: AplicacionRdPE3 92
Figura 56: Prueba de la variable analógica 93
Figura 57: TestBorrar 1 94
Figura 58: TestBorrar 2 94

0. Antecedentes

El presente proyecto se enmarca dentro de una larga serie de PFCs que se han realizado teniendo como objetivo la implementación del control de la célula de fabricación flexible situada en el laboratorio 0.06 del edificio Ada Byron del Campus Río Ebro de la Universidad de Zaragoza. Para ello se han utilizado los autómatas programables conectados a varios PCs mediante un red Fipway, y se han llegado a desarrollar proyectos muy diversos de control de tiempos, control de transportes, control desde terminales Magelis, desarrollo de aplicaciones Scada, etc...

Sin embargo este es el primer proyecto de lo que seguramente será otra larga serie de proyectos dedicados al control de la célula. Lo que diferencia este proyecto de los demás es que es el primer proyecto de control de la célula de fabricación basado en control directo desde un PC mediante Interbus. Para ello se realizó el precableado de las estaciones a los módulos Interbus (como se verá en la parte de descripción física de la célula) y se adquirió la tarjeta controladora de bus.

A pesar de la existencia de software para el control de estos sistemas, dado que la diferencia entre este tipo de control y el realizado mediante el software de control de autómatas PL7Pro sería mínima se decidió crear un tipo de control basado en un programa en un lenguaje de alto nivel. Dadas las características de portabilidad y seguridad se decidió que este lenguaje fuera Java.

Obviamente la tarjeta controladora de bus PCI SC-I/T se presentaba con un software propio de configuración del bus basado en el uso de librerías de comunicación en lenguaje C. Así pues la tarea que había por delante era la creación de una serie de clases en Java que permitieran acceder a las funciones C de control del bus para así, aprovechando las mayores capacidades de Java, implementar un control efectivo.

Para el control en sí mismo era necesario algún tipo de estructura funcional, es decir, algún tipo de herramienta de modelado de sistemas que permitiera crear un modelo de la célula de fabricación y su control. La elección más adecuada es sin duda alguna las redes de Petri, principalmente por dos razones: los principales programas de control de sistemas de las casas fabricantes de autómatas se basan en el Grafcet, que no es más que una simplificación adaptada de las redes de Petri, y la universidad de Zaragoza tiene una notable tradición dentro de la investigación y desarrollo de nuevas teorías y tipos de redes de Petri.

Así pues este proyecto es una mezcla entre automatización y programación. Si bien se parte de varios proyectos anteriores a la hora de automatizar la fabricación dentro de la célula, la creación de una implementación programada de redes de Petri requiere unos conocimientos de temas como concurrencia, creación y comunicación entre interfaces gráficos o documentación correcta de código fuente que se alejan de lo que es un proyecto electrónico para adentrarse en la informática pura y dura.

Así pues, en este proyecto se van a realizar las siguientes tareas:

- Configuración de las estaciones 1, 3, 4 y 6 mediante el software CMD de Phoenix Contact para control directo de las estaciones y configuración conjunta de todas ellas menos la 6 junto con el transporte.

- Usando las librerías HLI para el control de sistemas Interbus mediante la programación en lenguajes de alto nivel, crear un programa de control de cada una de las estaciones.

- Mediante el uso del JNI (Java Native Interface) integrar las funciones en C obtenidas mediante el programa CMD en una serie de clases Java.

- Control del identificador de productos. Desarrollo del protocolo de las tramas y de la comunicación entre el dispositivo y el programa.

- Creación de una serie de clases Java que implementan la estructura de una Red de Petri.

- Creación de aplicaciones sencillas que permiten comprobar si la estación ha sido correctamente configurada. Para ello se crean clases que permitan el desarrollo de la red de Petri y la comunicación entre transiciones/ estados de la red con entradas/ salidas de la célula.

- Modelado de las redes de Petri que permitan el control automático de la producción.

- Creación de una aplicación global de control tanto automático como manual de la célula.

- Crear un entorno gráfico de visualización del desarrollo del programa atractivo para el usuario.

1. La célula de fabricación.

La célula de producción de cilindros en serie está situada en el edificio A del Campus Río Ebro y se compone de dos módulos principales y uno de unión. El módulo I se encarga de la fabricación y verificación de los cilindros, el módulo de unión se encarga de la identificación y clasificación así como del almacenamiento y suministro de pedidos hacia el modulo II, encargado del montaje de estos pedidos y de su almacenamiento final. Todo esto pretende acercar al alumno a un proceso industrial real, con una gran parte de los elementos que están implicados en él.

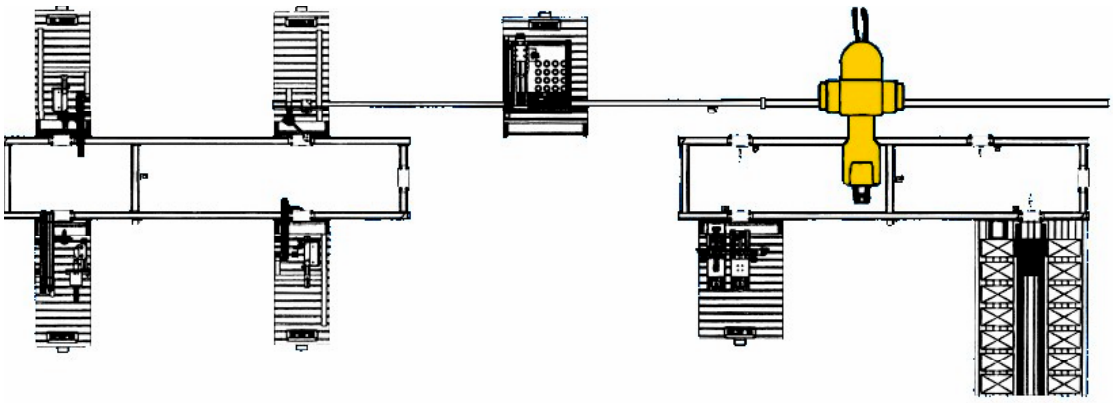


Figura 1: Disposición espacial de la célula de fabricación

Hay que tener siempre presente que se trata de una maqueta, no de máquinas reales, con lo cual se presentarán problemas que no son los de máquinas reales, sino más propios de réplicas a pequeña escala o juguetes.

La célula de fabricación simula un proceso productivo completo. La finalidad de todas las estaciones es la fabricación y expedición de una determinada mercancía constituida por un conjunto de tres cilindros neumáticos sobre una determinada base. Por lo tanto podríamos decir que la célula en conjunto constituye una pequeña fábrica destinada a la producción y expedición de unas determinadas piezas.

1.1 Tipos de piezas

En lo referente a los diferente tipos de piezas que es posible fabricar, podemos dividirlas en dos grupos diferentes. Por un lado están aquellas piezas que figuran ser cilindros neumáticos y por otro las que figuran ser cilindros cerrados (también denominadas piezas con tapa). En el cuadro resumen mostrado a continuación podemos observar la forma de cada una de las piezas.

COLOR DE LA CAMISA	Negra	Roja	Metálica

PIEZAS CON TAPA			
PIEZAS SIN TAPA (cilindros neumáticos)			

Figura 2: Tipos de piezas

Igualmente tal y como podemos ver en la figura anterior, dentro de un mismo tipo de piezas tenemos 3 posibilidades dependiendo del color de la camisa. Tal y como podemos ver en la mencionada figura, hay camisas de color negro, rojo o metálico.

Por lo tanto, existen seis tipos de piezas diferentes. El proceso de fabricación de las piezas dependerá del tipo seleccionado.

PIEZAS SIN TAPA

Ya se ha comentado que las piezas sin tapa simulan cilindros neumáticos de simple efecto. Así cada una de estas estará formada por los siguientes elementos:

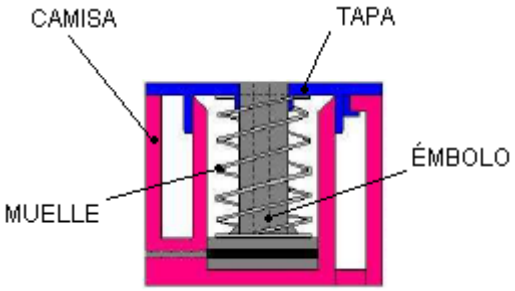


Figura 3: Esquema de una pieza

CAMISA. Constituye la parte exterior o carcasa del cilindro neumático a fabricar. Este será el soporte del resto de las piezas a colocar en el proceso de producción.

ÉMBOLO. Es el encargado de provocar el desplazamiento del eje al inyectar aire comprimido sobre el orificio destinado a tal efecto en la camisa. El diseño de los mismos evitará que el aire se escape a la atmósfera gracias a una junta que posee en su interior.

MUELLE. Dado que los cilindros neumáticos a fabricar son de simple efecto, deberemos de provocar el retorno del émbolo a su posición de origen una vez cortado el flujo de aire comprimido. Así la misión del muelle es la proporcionar esta fuerza de retorno del émbolo para que éste se recoja.

TAPA. La camisa del cilindro neumático deberá de ser cerrada para que las piezas internas no abandonen el interior de la camisa al inyectar aire comprimido al cilindro neumático. Esta función se lleva a cabo por medio de la colocación de la tapa.

Cada uno de los tres tipos de piezas a fabricar tiene unas características que las hacen diferentes entre sí. Así tenemos diferencias en lo referente a los colores y tamaños de los elementos que las forman o constituyen. A continuación vamos a comentar cada una de estas características de cada tipo de pieza.

El diámetro de todas las piezas es el mismo, en cambio la altura ellas no. Debe mencionarse que las camisas de color negro tienen una altura menor que las rojas y las metálicas. Por lo tanto en las piezas negras deberemos de instalar unos émbolos con una longitud más corta que las otras piezas. Los émbolos de menor longitud son de color metálico, mientras que los émbolos largos son de color negro.

Así, para poder realizar la fabricación de ambos tipos de piezas deberemos de disponer de los dos tipos de émbolos para colocárselos a las piezas adecuadas.

En lo referente a los muelles no tenemos ningún tipo de diferencia para cada una de las piezas fabricadas. La diferencia existente en la altura de las diferentes piezas únicamente diferirá en la compresión del muelle dentro de cada una de las camisas.

En la figura siguiente podemos ver la relación de elementos de cada una de las piezas a fabricar.

TIPO DE PIEZA			
CAMISA	Negra	Roja	Metálica
ÉMBOLO			
MUELLE	Estándar		

Figura 4: Tipos de émbolo

PIEZAS CON TAPA

El otro grupo de piezas que se pueden fabricar mediante la célula de fabricación flexible son las denominadas “con tapa”. Estas piezas están formadas únicamente por la camisa y una tapa que no posee orificio para la extensión del émbolo.

En estas piezas no será necesaria la colocación de los elementos mencionados en el apartado anterior, ya que con la camisa se encuentra sólidamente colocada la tapa de la camisa y por lo tanto tenemos una pieza compacta.

El proceso continúa posteriormente con el almacenado intermedio de las piezas para su posterior colocación en palets de 3 piezas tal y como se muestra en la siguiente figura.

La descripción del proceso de almacenaje intermedio y expedición de pedidos no se dará aquí ya que este proyecto sólo controlará la parte de fabricación. Puede encontrarse información sobre ella en cualquiera de los mucho proyectos dedicados a su control.

1.2 Zona de Fabricación

En esta zona tenemos situadas las estaciones necesarias para llevar a cabo el ensamblado de los diferentes elementos que componen la pieza a fabricar en cada momento. Así tenemos las siguientes estaciones:

- Transporte: Encargado del traslado de las piezas de una estación a otra por medio de un trasbordador destinado a tal efecto.

- Estación 1: Encargada de la colocación de la camisa deseada en el palet del trasbordador.

Figura 5: Estación 1

- Estación 2: Por medio de esta estación podremos colocar el émbolo y el muelle en las piezas que así lo requieran. La colocación del émbolo se realiza de forma acorde al tipo de pieza tratada en cada momento.

Figura 6: Estación 2

- Estación 3: Esta estación coloca y rosca la tapa a las piezas que así lo requieran, es decir, en las piezas con tapa no realiza ninguna operación.

Figura 7: Estación 3

- Estación 4: Antes de dar por finalizada la fabricación de una determinada pieza se realiza un test para comprobar que la pieza ha sido fabricada correctamente. Esta estación se encarga de tal función desechando las que son defectuosas y permitiendo el resto de proceso para las piezas fabricadas correctamente.

Figura 8: Estación 4

1.3 Topología del bus

La célula de fabricación se ha controlado a lo largo de varios PFCs mediante una red de comunicación industrial y una serie de autómatas programables. Se ha utilizado una red FIPWAY de la casa Schneider en bus a la que van conectados los 7 autómatas que se encuentran en la célula y 3 PCs por medio de una tarjeta Fipway isa. El bus comienza en la estación 1 (en la caja Fipway situada junto a ella, se coloca un terminador) y acaba en la estación 7. Los ordenadores con tarjeta Fipway ocupan las posiciones 8, 9 y 10 de la red, respectivamente. Ésta es la estructura usual para este tipo de redes, aunque sería recomendable una configuración en anillo para prevenir posibles cortes de la red entre dos estaciones. Podría cerrarse el anillo quitando los terminadores de las cajas Fipway de las estaciones 1 y 7 y uniéndolas con un cable. Además, el autómata 5 dispone de un módulo de conexión Ethernet y tiene su propio IP, con lo que se puede conectar a él a través de esa red o bien conectar con otros autómatas a través del puente Ethernet / Fipway que posee. Este hecho amplía notablemente las posibilidades de comunicación con la célula de fabricación, dado que nos podremos conectar a la misma por medio de la red INTERNET.

Este proyecto parte de la instalación en la célula de fabricación de varios módulos Interbus para el control de la misma. Por tanto el control de la célula se podrá realizar mediante el uso de una tarjeta controladora de bus desde un solo PC en vez del tradicional control basado en autómatas programables. En concreto se han cableado las estaciones 1, 3, 4 y transporte. El control de la estación 2 es tan complejo que su cableado y control fueron pospuestos para otro proyecto. Hay dos tipos de módulos de Interbus presentes en la célula:

- Un módulo TSX Momentum de Schneider Electric de 16 entradas y salidas digitales para el control de las salidas y entradas de la estación 3 y del transporte. Este módulo incluye la cabecera de bus y dos puntos de conexión, uno para entrada y otro para salida.

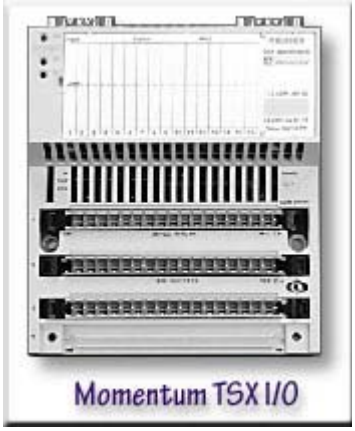


Figura 9: TSX Momentum

- Cabecera de bus IBS IL 24 BK T-U de Phoenix Contact más los módulos de entrada salida necesarios en las estaciones 1 y 4. Este tipo de configuración permite una gran flexibilidad a la hora de cablear los sensores y accionadores de las estaciones.

Figura 10: Módulos Interbus

2. Interbus

2.1 Características básicas.

Interbus es una red de sensores/accionadotes distribuidos para sistemas de fabricación y control de procesos continuos. Es un sistema abierto de altas prestaciones con topología en anillo.

El concepto básico de un bus abierto es permitir un intercambio de informaciones entre dispositivos producidos por diferentes fabricantes. Las informaciones intercambiadas comprenden datos de proceso (entradas / salidas) y parámetros (datos de configuración, programas, datos de monitorización). El formato de las informaciones está definido mediante un perfil estándar con el dispositivo opera. En Interbus se dispone de perfiles estándar para servomotores, encoders, controladores de robot, controladores de posición, paneles de control y operación, entradas / salidas digitales, entradas / salidas analógicas termopares, contadores, variadores de frecuencia, robots, controles de soldadura, sistemas de identificación, etc.…

Interbus no está respaldado por los grandes fabricantes de autómatas. Sin embargo, alrededor de 700 desarrolladores de dispositivos de campo lo soportan. Sacando al mercado continuamente nuevos desarrollos técnicos y productos. En la actualidad hay instalados mas de 1’5 millones de dispositivos de campo.

Un sistema basado en Interbus está compuesto por una tarjeta de control, instalada en un PC industrial o en un autómata programable, que comunica con un conjunto de dispositivos distribuidos de entrada / salida.

El protocolo de transmisión de datos es de alta eficiencia con el objetivo de cumplir los requerimientos de alta velocidad de transmisión en los sistemas de control.

La comunicación entre los dispositivos se realiza mediante el protocolo Interbus, el cual está reflejado en la norma DIN 19258. En Alemania, siguiendo los deseos de los usuarios, las funciones clave de Interbus están estandarizadas por la DKE (Deutsche Elektrrotechnische Kommission: German Elechtrotechnical Comisión for DIN and VDE). En 1993 se publica la norma DIN E 19 258. Esta norma fija los protocolos de transmisión y los servicios necesarios para la transmisión de datos. Las especificaciones de los parámetros de comunicación fueron publicadas en el DIN Report 46 en el año 1995. Todas estas normas fueron homologadas en el ámbito europeo en 1997 en la norma EN 50 254 con el título “High Efficiency Communication Subsystem for Small Data Packages”.

Interbus utiliza el método de comunicación maestro-esclavo, donde el maestro de bus del bus simultáneamente funciona como conexión o interfaz con el bus de nivel superior. El Protocolo Interbus utiliza la tecnología de trama suma: Un único telegrama actualiza simultáneamente todas las entradas y todas las salidas de los dispositivos físicos conectados al bus. Los niveles del modelo OSI que están implementados son: 1 (Físico), 2 (Enlace), y 7 (Aplicación).

2.2 Topología

La topología adoptada de Interbus es el anillo. Todos los dispositivos están conectados en un sistema de comunicación en bucle cerrado. Desde los dispositivos conectados al anillo principal, liderado por el maestro, se pueden conectar subanillos que estructuran el sistema completo. Las conexiones entre los anillos se realizan mediante módulos especiales denominados terminales de bus.

Un detalle que distingue Interbus de otros sistemas en anillo es que ambas líneas, la línea de envío y la línea de retorno de datos, pertenecen al mismo cable que pasa a través de todos los dispositivos físicos conectados al bus. Esto conlleva que tenga la misma apariencia física que una estructura en forma de árbol. El nivel físico de Interbus está basado en el estándar RS 485. La interfaz RS 4885 utiliza el método de transmisión por voltaje diferencial sobre un par trenzado. Debido a la estructura en anillo y a la necesidad de llevar la masa lógica a todos los dispositivos, Interbus requiere la conexión de cinco hilos de señal en el cable que conecta dos dispositivos. La transmisión de datos se puede realizar a una velocidad de 500 KBits a una distancia máxima de 400 m. entre los dispositivos. Como cada dispositivo actúa como repetidor de señal permite que Interbus alcance una distancia superior a los 13 Km. El máximo número de dispositivos conectados al bus es de 512, con un máximo de 4096 puntos de entrada / salida.

La estructura punto a punto de comunicación de Interbus y su división en anillo principal y subanillos se adecua perfectamente a la incorporación de diferentes y modernos sistemas de transmisión como la fibra óptica. El sistema de transmisión se puede convertir en fibra óptica, o en sistemas de transmisión por infrarrojos, o bien a otros sistemas utilizando convertidores estándar disponibles en el mercado. La utilización por medio de fibra óptica supone que la transmisión de datos está libre de interferencias.

La estructura en forma de anillo añade al sistema dos ventajas fundamentales. Primero, en contraste con las redes en línea, el anillo permite el envío y la recepción de datos simultánea. Segundo, la función de autodiagnóstico del bus se ve mejorada con la estructura de anillo. Si se produce un cortocircuito en el bus de comunicación, al disponer de una estructura en anillo es posible la localización del fallo dado que la comunicación sólo se interrumpe en un segmento de bus, lo cual no sucede en una estructura en línea.

Interbus incorpora el diagnóstico centralizado o descentralizado de los dispositivos del bus, el diagnóstico se puede realizar por medio de funciones o por registros de datos. El bus efectúa el reconocimiento automático de los dispositivos participantes y del perfil asociado a ellos. En caso de fallo se produce la desconexión automática del dispositivo donde se localiza el fallo, y se permite el cambio de dispositivos y de la topología del bus online.

2.3 Interfaz de usuario

El protocolo y la topología característica de Interbus aseguran la integración del transporte cíclico de los datos de entrada / salida y de los mensajes no cíclicos en un solo sistema, donde se tienen en cuenta y se

colman las demandas y las necesidades del campo de los sensores y accionadotes industriales. Éstos son los prerrequisitos necesarios para realizar una red de transmisión uniforme en este campo.

También deberá estar garantizado para el usuario un fácil acceso a los datos del bus. En definitiva los datos del proceso deben ser fácilmente accesibles para la aplicación de control en autómata o PC Industrial. Una tarea cíclica en el maestro del bus actualiza continuamente los datos de entrada / salida y los suministra al sistema de control en forma de una memoria de imágenes de entrada / salida.

Los datos de proceso pueden ser utilizados en un programa de autómata de forma idéntica a las entradas / salidas digitales clásicas.

En el caso de programación de aplicaciones de control en plataforma PC, los datos de proceso son accesibles por medio de interfaces de software estándar (DDE, OPC, Open Control).

Cuando se accede a los datos de proceso, el usuario no nota ninguna diferencia entre los datos accedidos vía serie (bus de campo), o los datos de proceso provenientes de un cableado tradicional. El usuario del bus no tiene que estar familiarizado con las complejas formas de comunicación del bus. Se han desarrollado tarjetas maestras de Interbus para autómatas Siemens, Telemecanique, Allen-Bradley, Hitachi, ABB, Bosch, etc.

Entre las herramientas de programación disponibles cabe citar el software CMD, que permite la configuración, monitorización y diagnóstico del bus y el software PCWORX que permite programar aplicaciones de control en plataforma PC conforme a la norma UEC 1131. Además existen drivers para Visual Basic, C++, Delphy.

2.4 Resumen de las capacidades de Interbus

INTERBUS, bus en anillo interno que permite:

• Tratamiento paralelo de la información (diseñado para sensores / actuadores)

• Tiempo de scan determinado (fijo, corto, calculable)

• Desconexión automática de utillajes (Ej.: robots)

• No requiere ajustes de velocidad, direccionamiento de módulo ni atender a las resistencias de cierre

• Preprocesado de señales

• Abierto a diferentes CPU’s de distintos fabricantes

• Alta eficiencia de la transmisión.

• Diagnóstico potente de identificación de fallos y su localización:

– en módulos

– en cableados y conexiones

– en periferia

• Más alta gama de productos y fabricantes del mundo

• Normalizado a nivel mundial (IEC 61158)

INTERBUS es una aproximación de sistemas abiertos a una red de dispositivos distribuidos, basada en anillo de alto rendimiento para fabricación y procesos de control. INTERBUS es un protocolo altamente eficiente para los requerimientos de control de alta velocidad actuales. Un sistema INERBUS consiste en una placa controladora instalada en una computadora (PC, VME, etc...) o autómata programable que se comunica con una variedad de dispositivos de entrada-salida. INTERBUS es operativo con la mayoría de paquetes estándar de software y sistemas operativos. INTERBUS es permitido por mas de 300 fabricantes de dispositivos de todo el mundo.

El protocolo de INTERBUS proporciona el alto rendimiento demandado por los requerimientos de entrada-salida de red actuales. Los datos de E-S se transmiten en marcos que proporcionan actualizaciones simultaneas y predecibles de todos los dispositivos de la red. Las transmisiones seguras son aseguradas mediante la capacidad de comprobación de errores del protocolo CRC. Además, el diagnostico exhaustivo permite localizar las causas y lugares de los errores. Esto proporciona un mayor tiempo de funcionamiento de la red. El protocolo de mensajería incorporado permite enviar parámetros complejos y mensajes de datos a lo largo de la red INTERBUS.

El concepto básico de un sistema de bus abierto es permitir un intercambio similar de información entre dispositivos producidos por diferentes fabricantes. La información incluye comandos y datos de E-S que han sido definidos como un perfil estándar por el cual operan los dispositivos. Los perfiles estándar están disponibles para drivers, encoders, controladores robóticos, válvulas neumáticas, etc. El protocolo INTERBUS, DIN 19258, es el estándar de comunicación para estos perfiles. Es un estándar abierto para redes de E-S en aplicaciones industriales.

La red INTERBUS (IBS) proporciona un enlace serie capaz de transmitir datos de E-S con velocidades en tiempo real. Tiempo real significa aquí que los datos de E-S son actualizados muchas veces más rápido de lo que la aplicación puede resolver la lógica.

2.4 La tarjeta controladora Interbus IBS PCI SC/I-T

La tarjeta IBS PCI SC/I-T es una tarjeta controladora de Interbus de 4ª generación con una interfaz remota para el bus PCI. Mediante el software CMD que viene con ella puede configurarse completamente los parámetros del proceso a controlar. La tarjeta IBS PCI SC/I-T permite mediante una memoria

programable EEPROM el almacenamiento permanente de los datos de parametrización en la propia tarjeta.

Según la página oficial de Interbus la red INTERBUS (IBS) proporciona un enlace serie capaz de transmitir datos de E-S con velocidades en tiempo real. Tiempo real significa aquí que los datos de E-S son actualizados muchas veces más rápido de lo que la aplicación puede resolver la lógica. El concepto básico de un sistema de bus abierto es permitir un intercambio similar de información entre dispositivos producidos por diferentes fabricantes. La información incluye comandos y datos de E-S que han sido definidos como un perfil estándar por el cual operan los dispositivos. Los perfiles estándar están disponibles para drivers, encoders, controladores robóticos, válvulas neumáticas, etc. El protocolo INTERBUS, DIN 19258, es el estándar de comunicación para estos perfiles. Es un estándar abierto para redes de E-S en aplicaciones industriales.

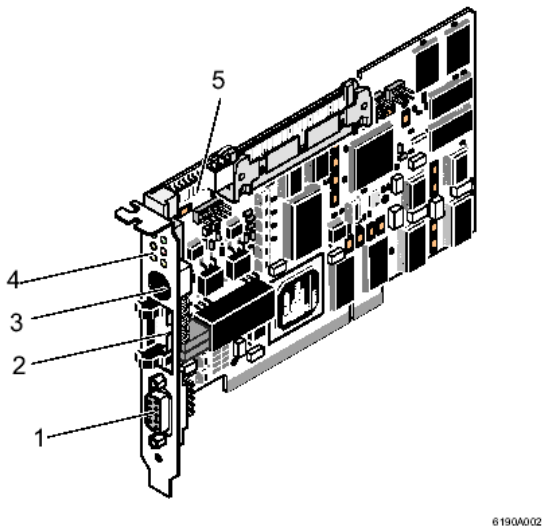


Figura 11: Esquema de la tarjeta IBS PCI SC/I-T

Figura 12: Tarjeta IBS PCI SC/I-T

En la figura puede observarse la configuración de la tarjeta:

1 - Interfaz de bus remoto Interbus.

2 - Conexión directa de entradas y salidas.

3 - Interfaz RS-232.

4 - LEDs de diagnóstico.

5 - Interruptores DIP para el establecimiento del número de la tarjeta.

Para el control de los componentes de la red conectados a la tarjeta Phoenix Contact proporciona el software CMD. Este software permite realiza la configuración del bus de manera rápida y sencilla. Además proporciona una herramienta muy útil: el HLI.

2.5 High-Level Language Interface (HLI)

El HLI es un conjunto de librerías que pueden usarse para desarrollar un programa de control de los componentes del bus mediante un lenguaje de alto nivel. Mediante el software CMD la configuración se realiza de forma directa. El manual de referencia se proporciona junto con la tarjeta.

La configuración del bus puede realizarse de forma manual, añadiendo los componentes que se sabe que están conectados en la red, o bien permitiendo al CMD que autoconfigure el bus leyendo los componentes que se encuentran disponibles. Una vez configurado el bus se tiene la opción de exportar la configuración a un fichero con código en un lenguaje de alto nivel en el que tendremos tres funciones: inicialización del bus, finalización del bus y proceso cíclico.

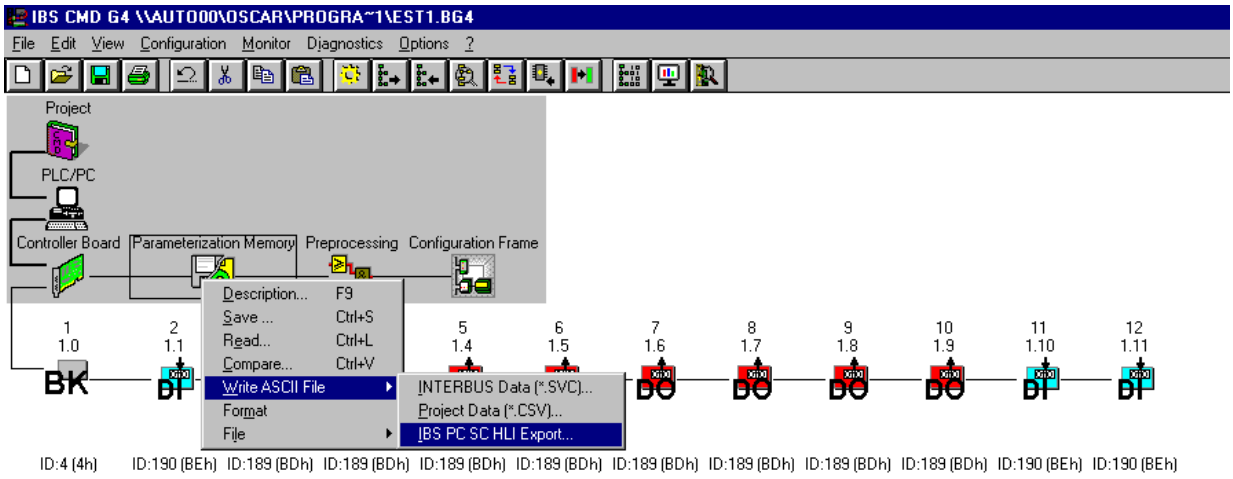


Figura 13: CMD

Dado que el objetivo de este proyecto es trabajar con lenguaje Java usaremos las librerías de lenguaje C, dado que la compatibilidad con Java es mucho más sencilla de implementar, como se explica en el siguiente apartado.

Las ventajas del HLI son:

- Configuración directa mediante el software CMD.
- Acceso a interbus independiente de tanto del sistema operativo como del hardware del sistema.
- Permite muchos lenguajes de programación.
- El uso de nombres de variables permite un intercambio de datos más rápido y fácil.
- Manejo de errores y del bus integrados.
- Acceso idéntico para toda la familia de tarjetas controladoras.

- Establecimiento de la comunicación y monitorización PCP automáticos.

El HLI permite los siguientes lenguajes de programación:

- Microsoft C/C++.
- Borland C/C++ (or compatible).
- Microsoft VB 4.0 (or later).
- Borland Delphi 2.0 (or later).

2.6 Configuración de las estaciones mediante CMD

IBS CMD G4 es un software que permite una configuración interactiva e independiente, la operación y el diagnóstico de todos los dispositivos conectados a la red INTERBUS. HLI ofrece al usuario de Interbus una herramienta de programación simple pero, a la vez, muy eficiente. Se usa para desarrollar aplicaciones de control en lenguajes de alto nivel para las siguientes tarjetas controladoras de Interbus: IBS PC ISA SC/I-T, IBS PCI SC/I-T, IBS PC 104 SC/T, IBS PCCARD SC/I-T, IBS PC ISA SC/486DX/I-T y IBS ETH DSC/I-T.

Esta basado en el Device Driver Interface (DDI) creado por Phoenix Contact y es válido para los sistemas operativos MS-DOS y Microsoft 95/98 y NT 4. Las aplicaciones pueden ser creadas en C y C++. Los sistemas de programación “Borland Delphi” y “Microsoft Visual Basic” solo son soportados para aplicaciones bajo Microsoft Windows 95/98 y NT 4.

Existe una herramienta que exporta la configuración del sistema Interbus creada con IBS CMD G4 a un archivo de programación a través de una generación inteligente de código fuente. La configuración realizada en CMD pasa directamente las variables al lenguaje de programación con el que se trabaje.

El CMD tiene la opción de ser ampliado; podemos coger programas extras o DLLs y añadírselos. Para ello existe la opción “ADD on Programs” y dentro de ella “Activate”.

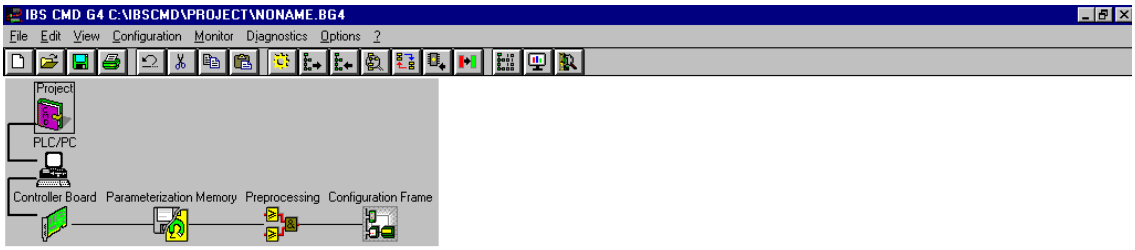


Figura 14: Pantalla principal CMD

Cuando instalamos el HLI aparecen DLLs que pueden activarse. En concreto, hay que buscar la carpeta IBSG4HLI, y dentro de ella el subdirectorio CMDG4EXP. Dentro de él se encuentra el archivo HLIEXPE.DLL que hay que seleccionar y luego “Aceptar” para cargarlo. Hecho esto, dentro del apartado “Memoria de parametrización” se verá que ha aparecido otro subapartado nuevo, dentro de “Write ASCII File”, llamado “IBS PCS HLI Export...”.

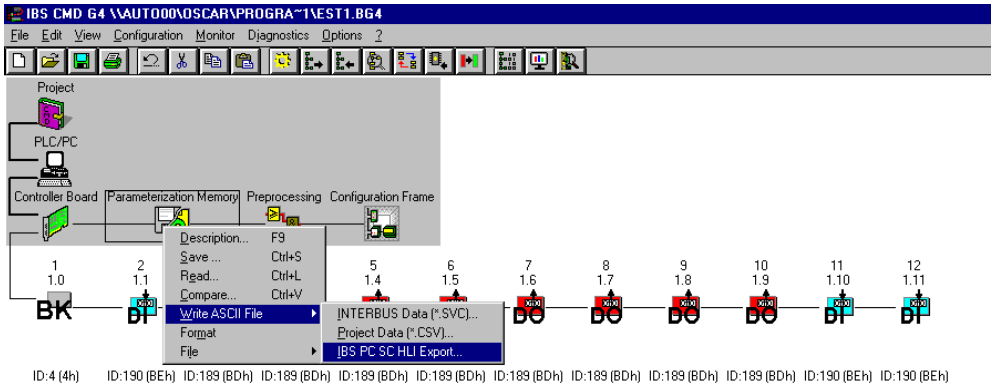


Figura 15: Como exportar una configuración

Ahora podemos crear un proyecto nuevo, uno para cada estación, pero no es necesario realizar asignaciones de memoria, ni direcciones ni links. Lo que importa son los nombres de los datos de proceso. Esos nombres serán luego los de las variables con las que se trabajará dentro del programa de alto nivel.

Existen dos formas de configurar cada estación: la configuración manual, con la que debemos conocer de antemano los módulos que contiene cada estación y la configuración automática, en la que es el propio software el que lee los componentes que están conectados al bus. Dado que en nuestro caso la conexión a la célula no presenta ningún problema se ha optado por esta segunda solución, siendo siempre la más recomendable excepto en el caso de que sea imposible físicamente la conexión del ordenador con el sistema a controlar.

A continuación se muestra la parametrización de cada una de las estaciones.

Estación 1

La estación 1 es la estación de verificación de camisas. Para su control se definen 19 variables digitales de entrada y 8 de salida. La configuración del bus es la que se muestra en la figura, y se compone de una cabecera de bus, un módulo de 8 entradas digitales, seguido de 8 módulos de salidas digitales, y 2 módulos más de 8 entradas digitales, por lo que van a usarse todos los módulos de salidas y van a quedar libres 5 entradas digitales del último módulo.

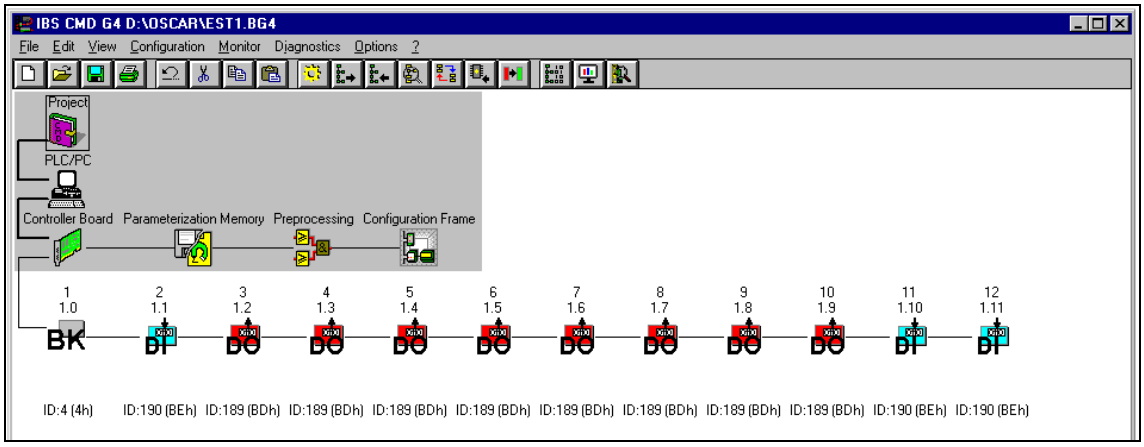


Figura 16: Configuración estación 1

A continuación puede verse una parte de la parametrización de los datos de proceso de la estación.

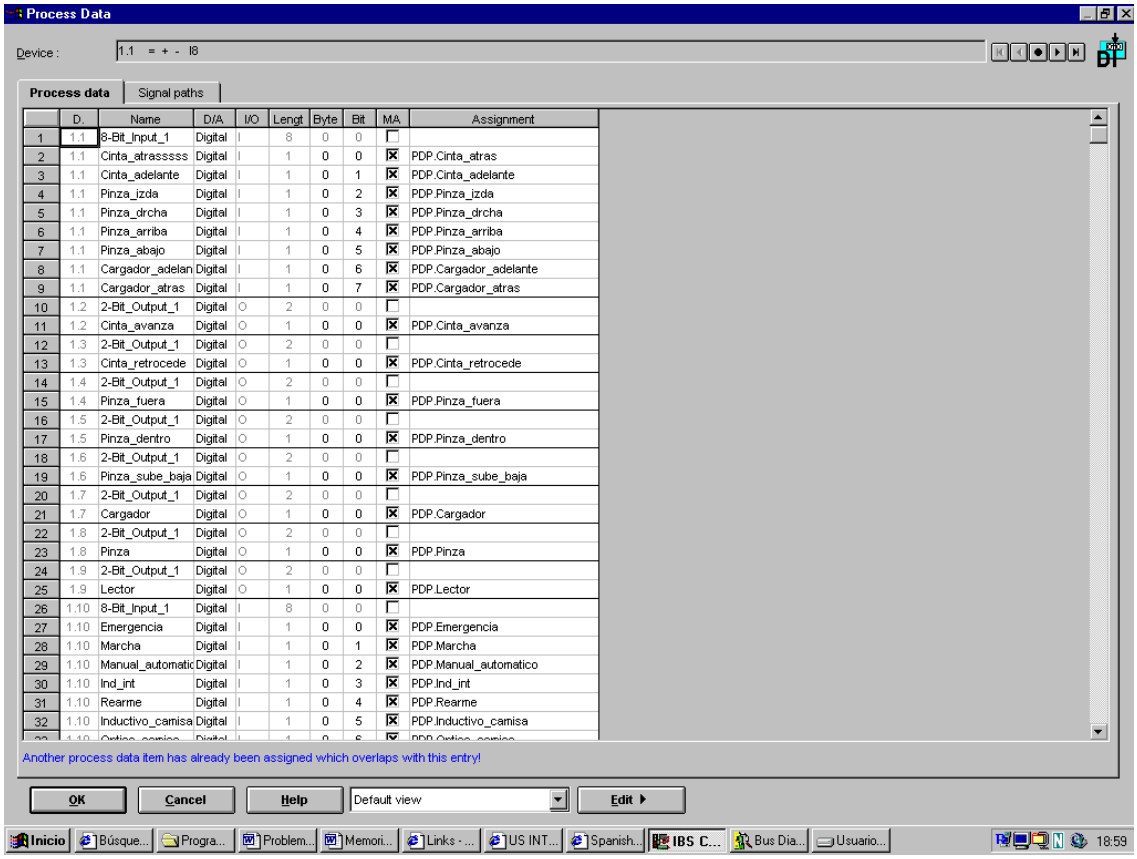


Figura 17: Process data estación 1

Una vez realizados estos procesos puede exportarse la configuración de la estación de la manera que ya ha sido descrita anteriormente.

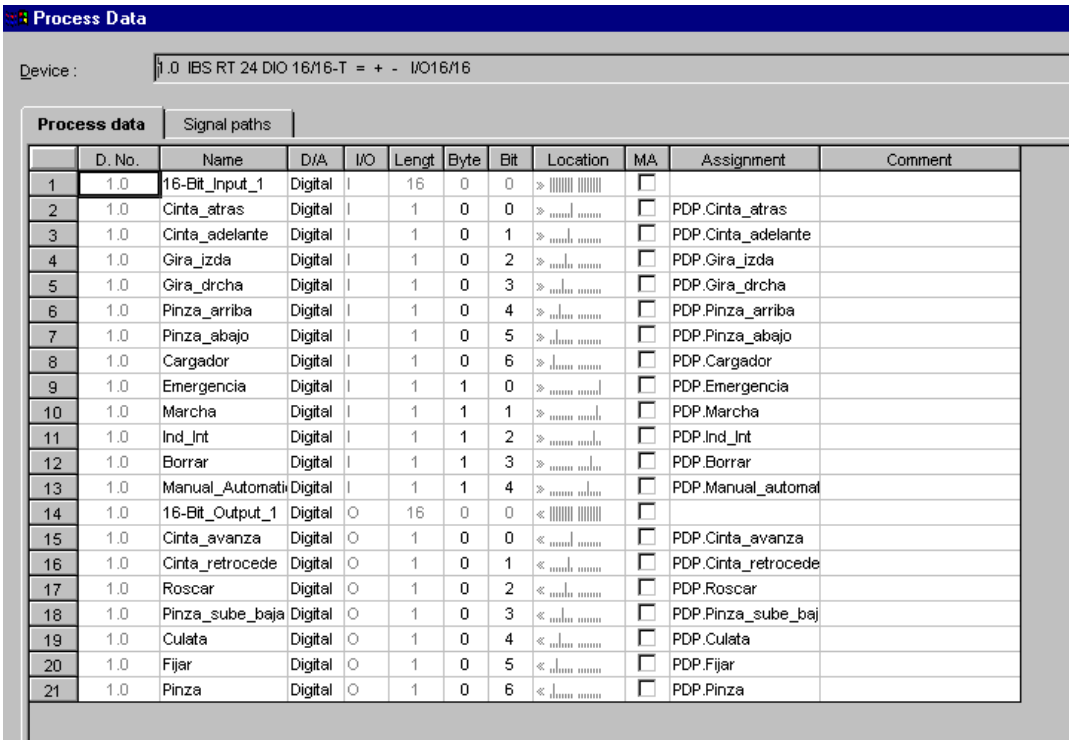
Estación 3

La estación 3 es la estación de montaje de las culatas. Para su control se tienen 13 entradas digitales y 7 salidas digitales. En la siguiente figura puede observarse que para el control mediante Interbus de la estación 3 se dispone de un módulo Momentum de 16 entradas y salidas digitales, que también incluye

una cabecera de bus. Este módulo ofrece un ahorro de espacio respecto a los módulos individuales pero pierde en cuanto a facilidad de mantenimiento, ya que en caso de error es más fácil sustituir el módulo defectuoso. En la imagen puede verse también que el formato en que aparece el dispositivo es diferente de los que aparecen en la figura de la estación 1. Esto es debido a que al ser la primera estación con la que se ha trabajado se realizó su configuración completa: es decir, aparece el dispositivo físico real que está conectado a la red Interbus. A efectos prácticos esto no influye sobre el fichero de programación producido sobre el HLI, pero es de gran utilidad para un control directo mediante el CMD.

Figura 18: Configuración estación 3

De igual forma se pueden observar los datos de proceso de la estación que serán exportados al fichero:



	D. No.	Name	D/A	I/O	Lengt	Byte	Bit	Location	MA	Assignment	Comment
1	1.0	16-Bit_Input_1	Digital	I	16	0	0				
2	1.0	Cinta_atras	Digital	I	1	0	0			PDP.Cinta_atras	
3	1.0	Cinta_adelante	Digital	I	1	0	1			PDP.Cinta_adelante	
4	1.0	Gira_izda	Digital	I	1	0	2			PDP.Gira_izda	
5	1.0	Gira_drcha	Digital	I	1	0	3			PDP.Gira_drcha	
6	1.0	Pinza_arriba	Digital	I	1	0	4			PDP.Pinza_arriba	
7	1.0	Pinza_abajo	Digital	I	1	0	5			PDP.Pinza_abajo	
8	1.0	Cargador	Digital	I	1	0	6			PDP.Cargador	
9	1.0	Emergencia	Digital	I	1	1	0			PDP.Emergencia	
10	1.0	Marcha	Digital	I	1	1	1			PDP.Marcha	
11	1.0	Ind_Int	Digital	I	1	1	2			PDP.Ind_Int	
12	1.0	Borrar	Digital	I	1	1	3			PDP.Borrar	
13	1.0	Manual_Automati	Digital	I	1	1	4			PDP.Manual_automat	
14	1.0	16-Bit_Output_1	Digital	O	16	0	0				
15	1.0	Cinta_avanza	Digital	O	1	0	0			PDP.Cinta_avanza	
16	1.0	Cinta_retrocede	Digital	O	1	0	1			PDP.Cinta_retrocede	
17	1.0	Roscar	Digital	O	1	0	2			PDP.Roscar	
18	1.0	Pinza_sube_baja	Digital	O	1	0	3			PDP.Pinza_sube_baj	
19	1.0	Culata	Digital	O	1	0	4			PDP.Culata	
20	1.0	Fijar	Digital	O	1	0	5			PDP.Fijar	
21	1.0	Pinza	Digital	O	1	0	6			PDP.Pinza	

Figura 19: Process data estación 3

Estación 4

La estación 4 es la de verificación final del producto. Consta de 16 entradas digitales, 11 salidas digitales y una entrada analógica. Al igual que la estación 1 esta estación está formada por módulos independientes, en este caso una cabecera de bus, un módulo de 8 entradas digitales, 8 módulos de salidas digitales, seguidos de otro módulo de 8 entradas y otros 5 módulos de salidas, para finalizar con un módulo de entrada-salida analógico.

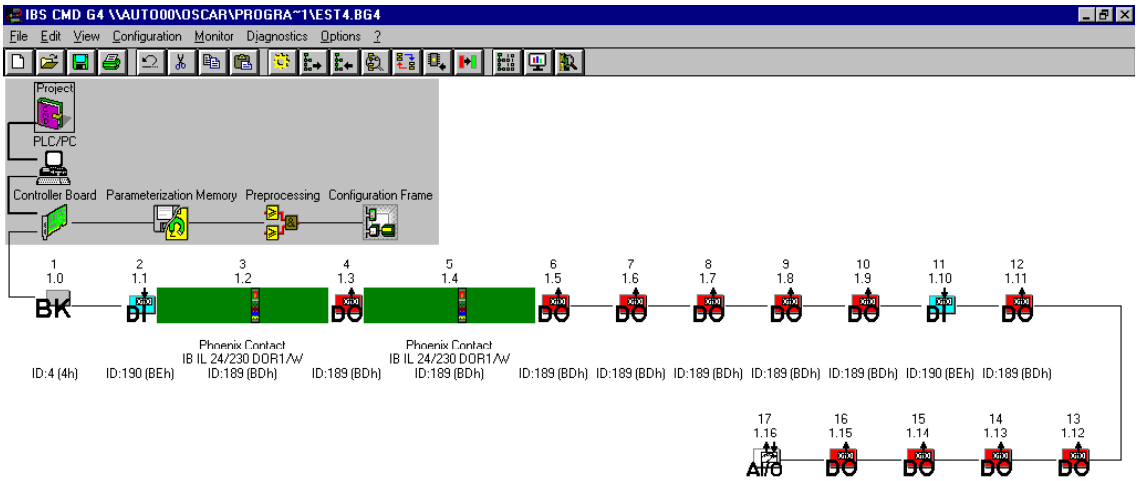


Figura 20: Configuración estación 4

De igual forma se definen los datos del proceso:

Process Data

Device : 1.1 = + - I8

Process data | Signal paths

	D.	Name	D/A	I/O	Lengt	Byte	Bit	Location (Byte/Bit)	MA	Assignment	Comment
13	1.3	Gira_drcha	Digital	O	1	0	0		☒	PDP.Gira_drcha	
14	1.4	2-Bit_Output_1	Digital	O	2	0	0		☐		
15	1.4	Cilindro_sube_ba	Digital	O	1	0	0		☒	PDP.Cilindro_sube_baja	
16	1.5	2-Bit_Output_1	Digital	O	2	0	0		☐		
17	1.5	Verificador_sube	Digital	O	1	0	0		☒	PDP.Verificador_sube_baja	
18	1.6	2-Bit_Output_1	Digital	O	2	0	0		☐		
19	1.6	Inyecta	Digital	O	1	0	0		☒	PDP.Inyecta	
20	1.7	2-Bit_Output_1	Digital	O	2	0	0		☐		
21	1.7	Expulsa	Digital	O	1	0	0		☒	PDP.Expulsa	
22	1.8	2-Bit_Output_1	Digital	O	2	0	0		☐		
23	1.8	Vacio_en_pinza	Digital	O	1	0	0		☒	PDP.Vacio_en_pinza	
24	1.9	2-Bit_Output_1	Digital	O	2	0	0		☐		
25	1.9	Vacio_en_pieza	Digital	O	1	0	0		☒	PDP.Vacio_en_pieza	
26	1.10	8-Bit_Input_1	Digital	I	8	0	0		☐		
27	1.10	Marcha	Digital	I	1	0	0		☒	PDP.Marcha	
28	1.10	Ind_Int	Digital	I	1	0	1		☒	PDP.Ind_int	
29	1.10	Rearme	Digital	I	1	0	2		☒	PDP.Rearme	
30	1.10	Manual_automatic	Digital	I	1	0	3		☒	PDP.Manual_automatico	
31	1.10	Pieza_fuera	Digital	I	1	0	4		☒	PDP.Pieza_fuera	
32	1.10	Sacar_pieza	Digital	I	1	0	5		☒	PDP.Sacar_pieza	
33	1.10	Vacio_pieza	Digital	I	1	0	6		☒	PDP.Vacio_pieza	
34	1.10	Vascula_pieza	Digital	I	1	0	7		☒	PDP.Vascula_pieza	
35	1.11	2-Bit_Output_1	Digital	O	2	0	0		☐		
36	1.11	Saca_pieza	Digital	O	1	0	0		☒	PDP.Saca_pieza	
37	1.12	2-Bit_Output_1	Digital	O	2	0	0		☐		
38	1.12	Expulsar_pieza	Digital	O	1	0	0		☒	PDP.Expulsar_pieza	
39	1.13	2-Bit_Output_1	Digital	O	2	0	0		☐		
40	1.13	Vascular	Digital	O	1	0	0		☒	PDP.Vascular	
41	1.14	2-Bit_Output_1	Digital	O	2	0	0		☐		
42	1.15	2-Bit_Output_1	Digital	O	2	0	0		☐	3.0	
43	1.16	32-Bit_Input_1	Analog	I	32	0	0		☐	2	
44	1.16	32-Bit_Output_1	Analog	O	32	0	0		☐	4	

Another process data item has already been assigned which overlaps with this entry!

OK | Cancel | Help | Additional view | Edit

Figura 21: Process data estación 4

Estación 6

La última estación controlada es la estación 6, que es la encargada de la expedición de bases blancas o negras. Es seguramente la estación más sencilla ya que sólo consta de 16 entradas digitales y 8 salidas digitales. Para su control, al igual que para el de la estación 3, se dispone de un módulo Momentum de 16 entradas y salidas digitales. En la figura se puede apreciar el aspecto que presenta el módulo Momentum antes de ser completamente configurado como en la estación 3.

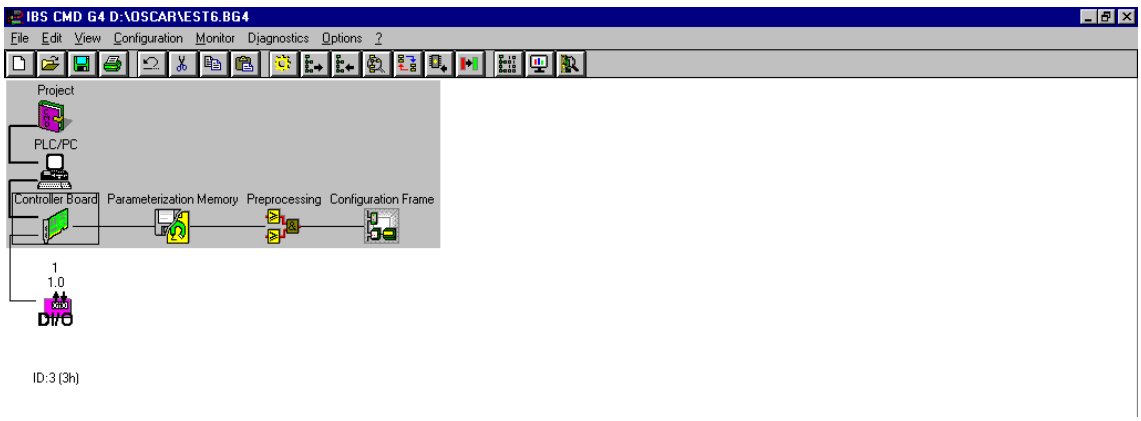
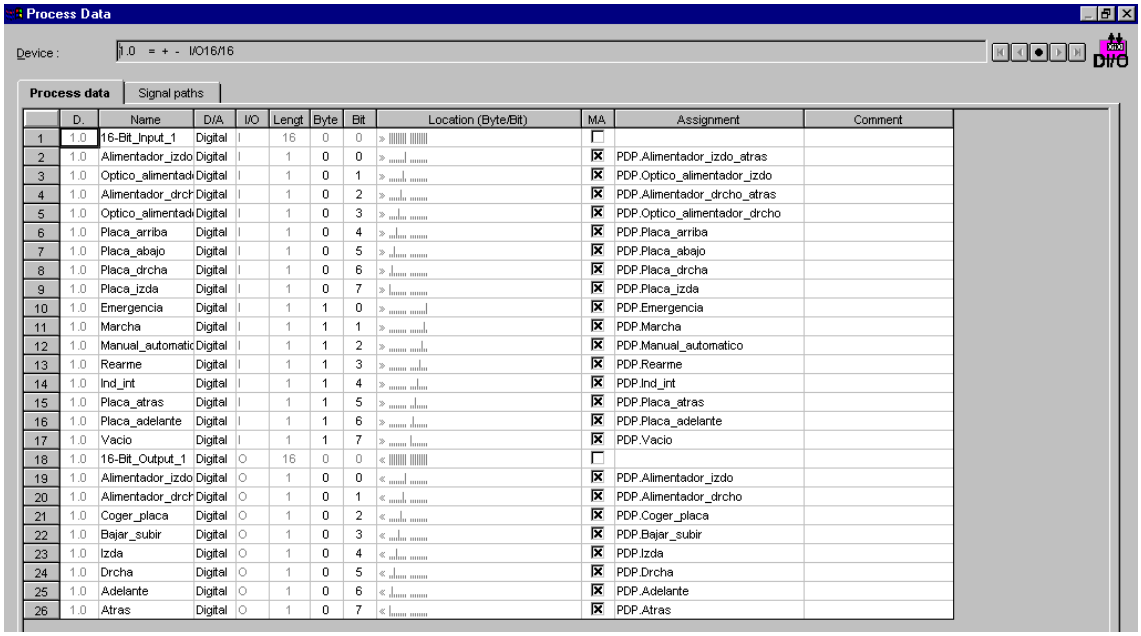


Figura 22: Configuración estación 6

De igual forma que en las anteriores, estos son los datos de proceso a exportar:



	D.	Name	D/A	I/O	Lengt	Byte	Bit	Location (Byte/Bit)	MA	Assignment	Comment
1	1.0	16-Bit_Input_1	Digital	I	16	0	0				
2	1.0	Alimentador_izdo	Digital	I	1	0	0			PDP.Alimentador_izdo_atras	
3	1.0	Optico_alimentadi	Digital	I	1	0	1			PDP.Optico_alimentador_izdo	
4	1.0	Alimentador_drch	Digital	I	1	0	2			PDP.Alimentador_drcho_atras	
5	1.0	Optico_alimentadi	Digital	I	1	0	3			PDP.Optico_alimentador_drcho	
6	1.0	Placa_arriba	Digital	I	1	0	4			PDP.Placa_arriba	
7	1.0	Placa_abajo	Digital	I	1	0	5			PDP.Placa_abajo	
8	1.0	Placa_drcha	Digital	I	1	0	6			PDP.Placa_drcha	
9	1.0	Placa_izda	Digital	I	1	0	7			PDP.Placa_izda	
10	1.0	Emergencia	Digital	I	1	1	0			PDP.Emergencia	
11	1.0	Marcha	Digital	I	1	1	1			PDP.Marcha	
12	1.0	Manual_automatic	Digital	I	1	1	2			PDP.Manual_automatico	
13	1.0	Rearme	Digital	I	1	1	3			PDP.Rearme	
14	1.0	Ind_int	Digital	I	1	1	4			PDP.Ind_int	
15	1.0	Placa_atras	Digital	I	1	1	5			PDP.Placa_atras	
16	1.0	Placa_adelante	Digital	I	1	1	6			PDP.Placa_adelante	
17	1.0	Vacio	Digital	I	1	1	7			PDP.Vacio	
18	1.0	16-Bit_Output_1	Digital	O	16	0	0				
19	1.0	Alimentador_izdo	Digital	O	1	0	0			PDP.Alimentador_izdo	
20	1.0	Alimentador_drch	Digital	O	1	0	1			PDP.Alimentador_drcho	
21	1.0	Coger_placa	Digital	O	1	0	2			PDP.Coger_placa	
22	1.0	Bajar_subir	Digital	O	1	0	3			PDP.Bajar_subir	
23	1.0	Izda	Digital	O	1	0	4			PDP.Izda	
24	1.0	Drcha	Digital	O	1	0	5			PDP.Drcha	
25	1.0	Adelante	Digital	O	1	0	6			PDP.Adelante	
26	1.0	Atras	Digital	O	1	0	7			PDP.Atras	

Figura 23: Process data estación 6

Transporte

Además de las estaciones también controlaremos el módulo que se encarga de gestionar las cintas y enclavamientos de las estaciones. Consta de 11 entradas digitales y 11 salidas digitales. Para su control, al igual que para el de la estación 3, se dispone de un módulo Momentum de 16 entradas y salidas digitales. En la figura se puede apreciar el aspecto que presenta el módulo Momentum antes de ser completamente configurado como en la estación 3.

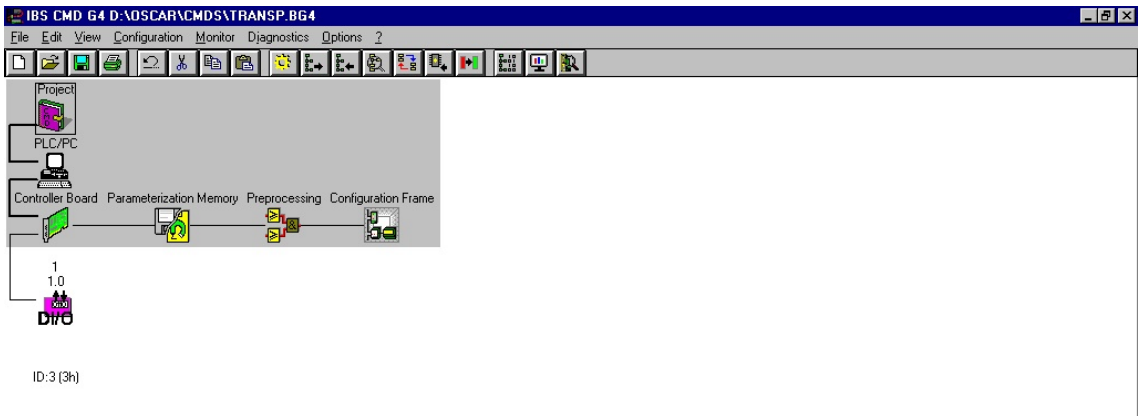
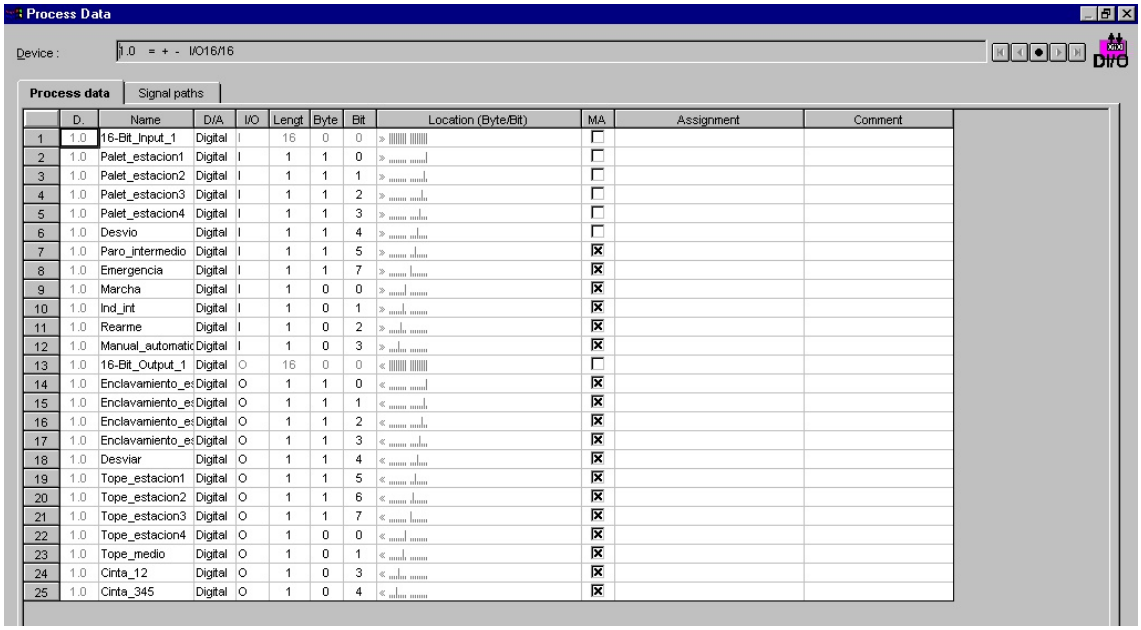


Figura 24: Configuración transporte

De igual forma que en las anteriores, estos son los datos de proceso a exportar:



	D.	Name	D/A	I/O	Lengt	Byte	Bit	Location (Byte/Bit)	MA	Assignment	Comment
1	1.0	16-Bit_Input_1	Digital	I	16	0	0		☐		
2	1.0	Palet_estacion1	Digital	I	1	1	0		☐		
3	1.0	Palet_estacion2	Digital	I	1	1	1		☐		
4	1.0	Palet_estacion3	Digital	I	1	1	2		☐		
5	1.0	Palet_estacion4	Digital	I	1	1	3		☐		
6	1.0	Desvio	Digital	I	1	1	4		☐		
7	1.0	Paro_intermedio	Digital	I	1	1	5		☒		
8	1.0	Emergencia	Digital	I	1	1	7		☒		
9	1.0	Marcha	Digital	I	1	0	0		☒		
10	1.0	Ind_int	Digital	I	1	0	1		☒		
11	1.0	Rearme	Digital	I	1	0	2		☒		
12	1.0	Manual_automatic	Digital	I	1	0	3		☒		
13	1.0	16-Bit_Output_1	Digital	O	16	0	0		☐		
14	1.0	Enclavamiento_e	Digital	O	1	1	0		☒		
15	1.0	Enclavamiento_e	Digital	O	1	1	1		☒		
16	1.0	Enclavamiento_e	Digital	O	1	1	2		☒		
17	1.0	Enclavamiento_e	Digital	O	1	1	3		☒		
18	1.0	Desviar	Digital	O	1	1	4		☒		
19	1.0	Tope_estacion1	Digital	O	1	1	5		☒		
20	1.0	Tope_estacion2	Digital	O	1	1	6		☒		
21	1.0	Tope_estacion3	Digital	O	1	1	7		☒		
22	1.0	Tope_estacion4	Digital	O	1	0	0		☒		
23	1.0	Tope_medio	Digital	O	1	0	1		☒		
24	1.0	Cinta_12	Digital	O	1	0	3		☒		
25	1.0	Cinta_345	Digital	O	1	0	4		☒		

Figura 25: Process data transporte

Quedan así pues listas todas las estaciones para la exportación del programa. Para ello se procederá como ya se ha dicho anteriormente. Para nuestro PFC se va a exportar las configuraciones al lenguaje C, que es con el que más fácilmente podremos trabajar posteriormente con Java (ver siguiente sección). Se crearan de esta forma sendos ficheros.C que contienen cada uno de ellos las definiciones de variables del programa tal y como se han definido con el CMD y 3 funciones pregeneradas: inicialización, finalización y procesamiento del bus. Es labor del programador generar una función principal que use tanto las variables como las funciones pregeneradas.

3. Lenguaje Java

El lenguaje de programación Java surgió en 1991 del trabajo de un pequeño grupo de personas liderados por James Gosling que anticiparon que en el futuro los pequeños electrodomésticos se habrían de conectar entre sí y con un ordenador para crear una red "doméstica". Para programarlos buscaban un lenguaje que permitiera que un mismo código funcionara igual independientemente del dispositivo o plataforma en que se ejecutara. Y aunque el mundo del pequeño electrodoméstico no estaba preparado para este salto de calidad, si lo estaba Internet. Mediante la introducción de la tecnología Java en el navegador Netscape Communicator en 1995 comenzó lo que con el tiempo ha sido uno de los más rápidos desarrollos en la historia de la informática, debido principalmente a las dos principales características de Java: su portabilidad (puede ejecutarse un mismo código en muy diversas plataformas) y su seguridad (Java se concibió como un lenguaje de alto nivel orientado a objetos para aprovechar las características de encapsulamiento y ocultación de información entre objetos inherentes a este tipo de lenguajes).

3.1 ¿Qué es Java?

Java es un lenguaje de programación, un sistema de tiempo de ejecución, un juego de herramientas de desarrollo y una interfaz de programación de aplicaciones (API).

Un desarrollador de software escribe programas en el lenguaje Java que emplean paquetes de software predefinidos de la interfaz para programación de aplicaciones de Java (API). Luego compila sus programas mediante el compilador Java y el resultado de todo ello es lo que se denomina un bytecode compilado. El bytecode está en una forma que puede ser ejecutado en la máquina virtual Java, que es el núcleo del sistema de tiempo de ejecución de Java. Esta máquina virtual puede representarse como un microprocesador implementado en software que funciona utilizando las capacidades que le presta su sistema operativo y el hardware de su computador. Ahora bien, puesto que la máquina virtual Java no es un microprocesador real, el bytecode Java es interpretado más que ejecutado directamente en las instrucciones de la máquina nativa del computador principal. El JRE (Java Runtime Environment) o sistema de tiempo de ejecución en Java se compone pues de la máquina virtual y del software adicional con bibliotecas de enlace dinámico, necesarios para implementar la API de Java.

Las claves de la portabilidad de Java son su sistema de tiempo de ejecución y su API. El primero es sumamente compacto, pues deriva de anteriores esfuerzos de la empresa Sun para crear una plataforma de software para electrodomésticos. Esta plataforma no se diseñó a partir de un microprocesador ya existente, sino que arrancó de cero con el propósito de ser simple y eficaz. La naturaleza simple, eficaz, compacta y de arquitectura neutra del JRE es lo que confiere su gran portabilidad y le hace alcanzar unas prestaciones notables.

Los potentes recursos para el trabajo en ventanas y redes incluidos en el API de Java facilitan a los programadores el desarrollo de un software que resulte a la vez atractivo e independiente de cualquier plataforma. Existen lenguajes de programación, como Ada, que están notablemente estandarizados y son aptos para la mayoría de sistemas operativos. Sin embargo, las aplicaciones de Ada no son precisamente

muy portables porque el lenguaje no cuenta con una API común que admita el trabajo en ventanas y redes en todas las plataformas. Java se diferencia de Ada y de todos los lenguajes de programación en que dispone de una API apta para todos los sistemas operativos más utilizados: Windows, Linux y Solaris, además de contar con extensiones para móviles, bases de datos, etc...

Como ya se ha dicho, Java surgió durante el año 1991 en el seno de Sun Microsystems como un intento de crear una plataforma de software barata e independiente del hardware mediante C++. Por una serie de razones técnicas se decidió C++ y crear un nuevo lenguaje de programación basado en él, llamado Oak, que superara las deficiencias del C++, tales como la herencia múltiple, la conversión automática de tipos, el uso de punteros y la gestión de la memoria. El lenguaje Oak se usó para crear un pequeño dispositivo electrónico llamado *7. Pero el proyecto llevó asimismo a la creación de numerosos elementos que fueron precursores de los componentes de Java. Hubo ciertos intentos de aplicar esta tecnología a cierto número de aplicaciones de consumo, pero lo cierto es que estaba demasiado adelantado a su tiempo y no tuvo la salida esperada.

En 1994 se produjo el nacimiento de Internet; Oak fue rebautizado como Java y sus creadores decidieron utilizarlo como base para un navegador Web, que llamaron WebRunner. A principios de 1995 HotJava, Java y la documentación y código fuente pudieron ser obtenidos vía red en su versión alfa. Inicialmente Java se presentó para Sparc Solaris y, a continuación, para Windows 95 y Linux. En otoño del mismo año se publicó la versión beta 1 de Java en la página de Sun, al tiempo que el navegador Netscape 2.0 introducía la compatibilidad con Java.

En diciembre de 1995 se dio a conocer la versión beta 2 de Java, y Sun y Netscape anunciaron la aparición de JavaScript. El éxito de Java fue ya inevitable cuando a principios de ese mismo mes Microsoft e IBM dieron a conocer su intención de solicitar licencia para aplicar la tecnología de Java. El 23 de enero de 1996 se publicó oficialmente la versión 1.0, que podía obtenerse desde la página web. Actualmente Java está en su versión 5.0 que es la utilizada en este proyecto.

3.2 Nociones básicas de Java.

Esta memoria no puede pretender ser un manual detallado de programación en Java. Sin embargo si que resulta conveniente dar algunas nociones acerca de este lenguaje para clarificar algunas decisiones tomadas a lo largo de la realización del proyecto.

3.2.1 Orientación a objetos

Java es un lenguaje orientado a objetos. El paradigma de la programación orientada a objetos surgió en contraposición a la tradicional programación estructurada. En la programación estructurada primero se examina el problema a resolver, se divide en tantas partes como sea necesario, y se implementa la solución mediante funciones. Lenguajes clásicos como C o Pascal se enmarcan dentro de esta categoría.

Sin embargo los lenguajes de programación orientados a objetos parten de que para resolver un problema lo primero es descomponerlo en los objetos que lo forman. Por ejemplo, si quisiéramos programar un sistema de gestión de un zoológico según el método de programación estructurada cogeríamos todo lo que hay que hacer en un zoo (alimentar a los animales, gestión de las taquillas, etc...) y crearíamos los procedimientos necesarios. En la programación orientada a objetos lo primero que hay que preguntarse es qué objetos componen el sistema. En este caso un posible modelado podría contener a los animales, los empleados, las jaulas, las tiendas, etc... Una vez hecho esto hay que asignar unas funcionalidades a cada objeto y pensar en cómo se comunican entre ellos. Por ejemplo, los empleados tendrán la función de limpiar las jaulas, los animales tendrán una jaula asignada, etc...

La programación orientada a objetos posee ciertas características que hacen de la programación un proceso más eficiente que la programación estructurada: la herencia, el polimorfismo y el ocultamiento de la información. Por ejemplo, para modelar nuestro zoo podríamos crear un objeto animal al que le asignaríamos una jaula a cada uno y una determinada cantidad de alimentos. Después cada animal individual con sus propias características sólo tendría que “sobrescribir” según sus necesidades la cantidad de alimentos. Esto es lo que se llama herencia: aprovechar las funcionalidades de un objeto ya creado (llamado comúnmente padre) para, añadiendo alguna nueva si es necesario, crear un nuevo objeto (llamado hijo). También así un empleado que se encargará de alimentar a los animales no tendría que tener un método específico para alimentar a cada tipo de animal; tendrá un método “alimentaAnimal” que según el animal a alimentar hará una cosa u otra. A esta capacidad se le llama polimorfismo. El ocultamiento de la información consiste en que una clase no tiene por qué conocer la forma en que otra implementa un determinado método: Siguiendo con nuestro ejemplo, el león no necesita saber como es alimentado por el cuidador, mientras este lo alimente correctamente.

El software diseñado con esta metodología se dice que es reutilizable. En programación estructurada se tomaba cada tarea como un todo, por lo que en cuanto surgían problemas particulares se hacía muy difícil el aprovechar código ya existente para un problema similar (por ejemplo, dos zoos con diferentes tipos de animales). Si por algo se ha impuestos la programación orientada a objetos es porque permite más fácilmente la reutilización del código: si quisiéramos hacer otro zoo ya contaríamos con la mayoría del

sistema descrito por los objetos, y en caso de que hubiera animales nuevos en vez de empezar de cero ya contaríamos con la clase animal de la que hacerlos descender. Esto permite una mayor rapidez en el desarrollo del software a la vez que una menor probabilidad de errores de programación, junto con un diseño más eficiente. Características que son las que las empresas de desarrollo de aplicaciones, como cualquier otra empresa, busca finalmente para sus productos.

Aplicando esta metodología en este proyecto lo primero en que se pensó fue en los objetos que habríamos de tener: estados, transiciones, coordinadores, redes, estaciones, variables de entrada y salida, etc... Una vez se tiene esto los métodos de cada uno surgen de manera lógica. Para comprobar la bondad del método basta con echar un vistazo a la implementación programada de RdP en Ada existente y al código creado para este proyecto: de un código casi ilegible, y por tanto difícil de modificar y mantener, se ha pasado a un código perfectamente estructurado en el que cada clase posee una funcionalidad específica que hace que posibles fallos o modificaciones sean más fáciles de localizar y realizar.

3.2.2 Concurrencia

La programación concurrente es aquella en la que están involucrados varios hilos de ejecución funcionando simultáneamente. Esta situación se produce normalmente cuando varios objetos intentan acceder a un recurso común limitado. Para poner un ejemplo de este mismo proyecto, esto sucede cuando las estaciones intentan acceder al identificador de productos para leer o escribir los palets. Dado que el identificador de productos sólo puede atender a una de ellas cada vez debe de implementarse algún sistema que permita el control de quién y cómo accede al recurso.

El lenguaje Java desde sus origines ha dado soporte para la programación multihilo a través de la clase `Thread`. Un `Thread` representa un hilo de ejecución de una aplicación. Una aplicación puede tener un gran número de hilos ejecutándose simultáneamente; por ejemplo, el garbage collector es un hilo que se encarga de borrar de la memoria los objetos que han sido asignados para ello. Hasta la versión 1.4.2 del JDK Java implementaba la concurrencia mediante las llamadas a los métodos `wait()`, `notify()` y `notifyAll()` y la sentencia `synchronized`.

El método `wait()` provoca que un hilo deje de ejecutarse y espere en estado “latente”. El hilo “despertará” cuando otro hilo llame a uno de los métodos `notify()` o `notifyAll()`. La sentencia `synchronized` implica que la porción de código o el método al que afecta no puede ser accedido a la vez por más de un hilo; es decir, cuando un hilo llega a una porción de código o a un método marcado como `synchronized` adquiere la “llave” o lock del objeto donde está el código o método. A partir de este momento y hasta que se libere el lock ningún otro hilo podrá acceder a métodos o variables de ese objeto.

Primitivas de multihilado de bajo nivel, tales como bloques `synchronized`, `Object.wait` y `Object.notify`, son insuficientes para muchas tareas de programación. Como resultado los programadores de aplicaciones se ven forzados a menudo a implementar sus propias funciones de concurrencia. Esto conlleva una enorme duplicación del esfuerzo. Además, estas funciones son difíciles

de conseguir y aún más de optimizar. Las funciones de concurrencia escritas por programadores de aplicaciones son a menudo incorrectas o ineficaces. Ofreciendo un conjunto estandarizado de funciones de concurrencia se puede ayudar en la tarea de escribir una amplia variedad de aplicaciones multihilo y se mejorará la calidad de las aplicaciones que las usen.

Hasta el JDK1.5.0, los desarrolladores sólo podían usar las construcciones de control de concurrencia contenidas en el mismo lenguaje Java. Estas son de demasiado bajo nivel para algunas aplicaciones e incompletas para otras. Por tanto Java sólo proporciona las herramientas y es el programador el que con ellas debe implementar las relaciones entre los hilos de su aplicación. La increíble complejidad de la programación concurrente hace que esto fuera fuente común de errores de programación, comportamientos indeseados, cuelgues misteriosos, etc...

Sin embargo en Java 1.5 se incluyó en el API un paquete que contiene clases diseñadas específicamente para ayudar en la programación concurrente: el `paquete java.util.concurrent`. Según Doug Lea, el coordinador del proyecto, los objetivos de este paquete son similares a los que se pretendían conseguir con el paquete `java.util.collections` contenido en el JDK1.2:

1- Estandarizar un entorno de trabajo simple y desarrollable que organice utilidades usadas comúnmente un paquete lo suficientemente pequeño como para ser rápidamente aprendido por los usuarios y mantenido por los desarrolladores.

2- Proveer algunas implementaciones de alta calidad.

El paquete consiste en interfaces y clases que tienden a ser útiles a lo largo de diversas aplicaciones y estilos de programación. Estas clases incluyen:

- Variables atómicas.
- Monitores (locks) de propósito especial, barreras, semáforos y variables condicionadas.
- Colas y colecciones relacionadas con ellas diseñadas específicamente para su uso con múltiples hilos de ejecución.
- Conjuntos de hilos (thread pools) y frameworks de ejecución prefabricados.

Dentro del presente proyecto existen dos partes claramente condicionadas por la concurrencia: el acceso desde varios coordinadores a las variables comunes y el acceso al identificador de productos.

Para la primera parte, se ha cuidado que el acceso de varios hilos a variables comunes sea seguro frente a accesos concurrentes. Por ello todas estas variables están contenidas no dentro del coordinador, sino dentro de una clase auxiliar llamada monitor. Cualquier variable que necesite uno de los coordinadores o una de las interfaces gráficas debe encontrarse dentro de un monitor, que es el que garantizará la exclusión mutua entre diferentes hilos de ejecución a la hora de acceder a sus variables. Así pues, todos los objetos de la clase `VariableMemoria` que sirven al coordinador para decidir sobre la evolución de la RdP, son declarados como privados, es decir, que sólo se puede acceder a ellos a partir de métodos del propio monitor. Obviamente estos métodos son los típicos `void set(valor)` y `valor get()` para

leer o escribir dichas variables. La sincronización se realiza al declarar estos métodos como `synchronized`, con lo que la exclusión mutua queda garantizada. Para variables del tipo `java.util.Vector` como el objeto `vectorPedidos` que contiene los pedidos realizados en modo automático, no se ha implementado métodos para garantizar la sincronización ya que los propios desarrolladores de Java garantizan que las instancias de esta clase son seguras frente a múltiples accesos. Por tanto estas variables son una excepción a la regla general al ser declaradas como publicas dentro del monitor.

Se consideró también la posibilidad de gestionar todas las variables a utilizar en el proyecto como si fueran un monitor en sí mismas, cada una garantizando su propia exclusión mutua. Se desechó esta opción porque los objetos que quisieran tener acceso a estas variables necesitarían tener una instancia de la misma, lo que en la práctica implica que a la hora de crear, por ejemplo, la interfaz gráfica de la estación 1, habría que pasarle como parámetros, ya fuera en el constructor o en un método, todas las instancias de las variables / monitores, es decir, si necesita para su gestión cuarenta variables, habría que crear una función que le pasara los cuarenta monitores al coordinador. También se implementó una versión de la aplicación de la célula con un solo monitor que contenía el acceso a todas las variables. Al final la opción más acertada en mi opinión, y la implementada en la versión final, consiste en cinco coordinadores independientes, un por estación y otro que gestiona variables consideradas "globales" y el acceso al identificador de productos.

La otra parte condicionada por el acceso exclusivo es el identificador de productos. En efecto, puede darse el caso de que dos o más estaciones intenten leer o escribir sus palets al mismo tiempo. Para gestionar el acceso correcto al identificador de productos hubo que resolver una doble problemática: el acceso de múltiples hilos al identificador y el acceso múltiple desde un mismo hilo. Para resolver este problema se intentó implementar una solución a partir de una instancia de la clase `ReentrantLock`. Sin embargo a la hora de ejecutar el programa se observaron "liberaciones" indeseadas del candado que hacían que se interrumpiera el correcto desarrollo del programa. Por tanto se desechó el uso del `ReentrantLock` y se sustituyó por una estructura de variables booleanas atómicas implementadas a partir de métodos sincronizados. Para una explicación más completa puede verse la sección dedicada al código del identificador de productos y redes de Petri.

3.2.3 Métodos nativos. JNI

El JNI es la herramienta que posee Java para aprovechar métodos realizados en otro lenguaje de programación. Aunque esto pueda parecer un contrasentido es una situación que se da más veces de las que podría suponerse. Por ejemplo, existen algoritmos matemáticos en C que al ser pasados a Java aumentaban extraordinariamente su tiempo de ejecución. O librerías de drivers que sólo están disponibles en determinados lenguajes, como es el caso que nos ocupa.

Dado que Java surgió a partir de C++ la implementación de métodos nativos más sencilla es la realizada en C o C++. El procedimiento a seguir se explicará más detalladamente en el próximo apartado.

3.3 Proceso para usar Java con Interbus

Para este PFC queremos trabajar con el lenguaje de programación Java, que no es uno de los que soporta el HLI. Sin embargo existe una forma de trabajar con las funciones C que se han generado mediante el CMD: el JNI o *Java Native Interface*.

El proceso para utilizar código nativo (que es como designaremos de ahora en adelante al código C) dentro de un programa Java se puede resumir en la siguiente figura, que resume el proceso de creación de la típica aplicación de ejemplo *Hola Mundo*:

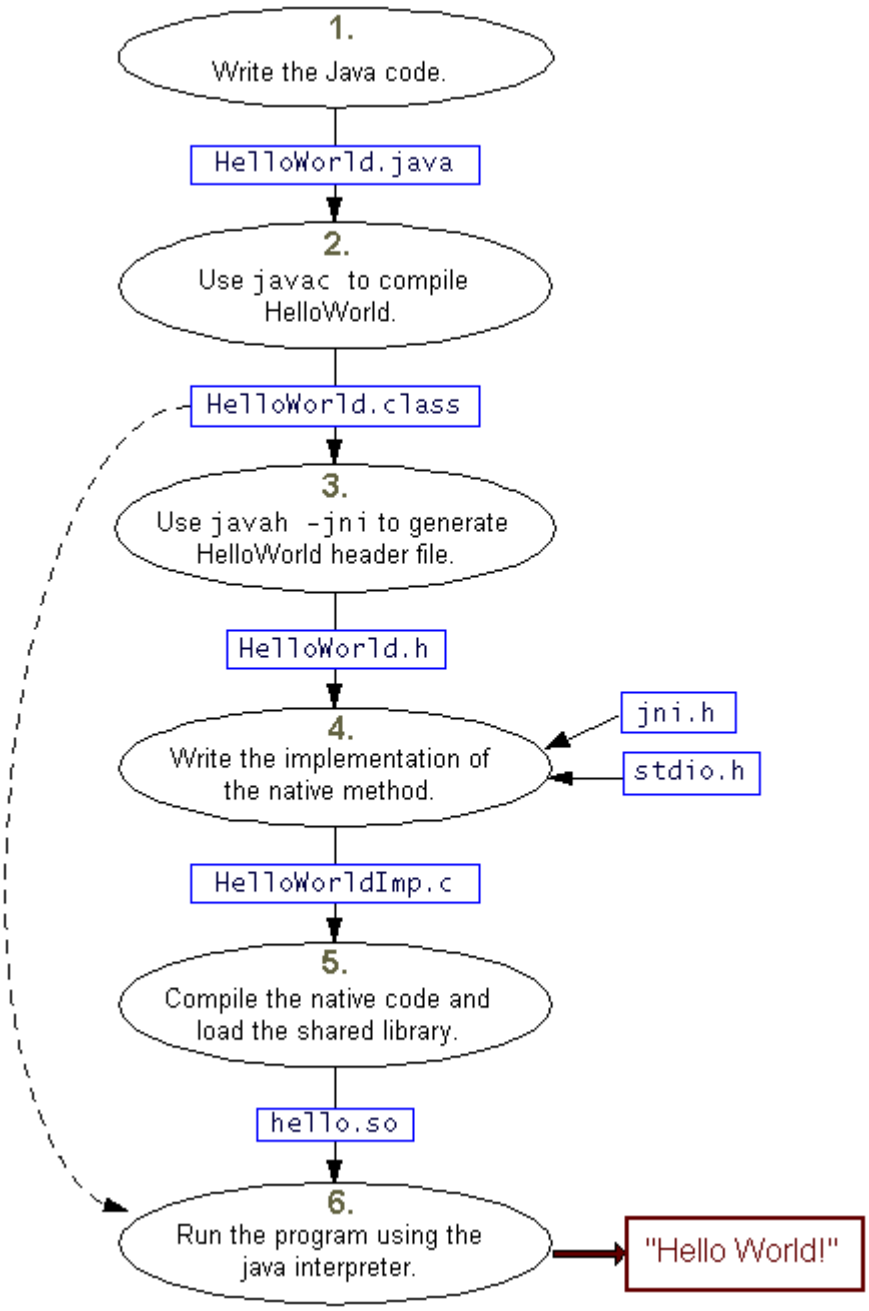


Figura 26: Pasos del JNI

A continuación se resumen los pasos necesarios:

Paso 1: Escribir el código Java.

Tenemos que crear una serie de clases Java que nos permitan controlar las estaciones. Para ello se ha creado una clase por cada estación, en la que se han definido las variables de entrada y salida de la estación como objetos de la clase `VariableBooleana`. Esta clase contiene el valor booleano de la variable así como un número que la identifica dentro del conjunto de variables de la lista. Se ha tratado de conseguir una coherencia interna del programa; así, aunque solo haría falta una mega-clase principal que contuviera todo el código de control he preferido dividir las funciones nativas entre las clases más apropiadas para contenerlas. Así la clase `VariableBooleana` contiene los métodos:

```
public native boolean leeEntrada(int orden);

public native void escribeSalida(boolean salida, int orden);
```

Mientras que cada clase de estación contiene el método:

```
public native void InicializaEstacion();

public native void FinalizaEstacion();
```

La etiqueta **`native`** proporciona al compilador la información acerca de que la implementación de las funciones se define en un archivo en otro lenguaje de programación.

Paso 2: Compilar el código Java.

Para compilar y ejecutar el código Java de este PFC se ha usado JBuilderX, en su versión gratuita. Para compilar las clases Java se usa la aplicación `javac` con lo que resultan los archivos `.class`.

Paso 3: Crear el archivo .h

Los archivos con la extensión `.h` son archivos de cabecera (en inglés *header*) que proporcionan información al programador acerca de las funciones de un programa independientemente de su implementación. Mediante la aplicación `javah` se crea un archivo que contiene el nombre y los parámetros que debe tener la función nativa para que sea comprensible para el programa Java. Así por ejemplo al ejecutar la aplicación en la clase `VariableBoolena` se obtiene el siguiente resultado:

```
/*
 * Class:     VariableBooleana
 * Method:    actualizaVar
 * Signature: (Z)V
 */
JNIEXPORT void JNICALL Java_VariableBooleana_actualizaVar
  (JNIEnv *, jobject, jboolean);

/*
 * Class:     VariableBooleana
 * Method:    actualizaVariable
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_VariableBooleana_actualizaVariable
```

```
(JNIEnv *, jobject);
```

Este proceso es similar para todas las funciones nativas que se quieran utilizar en un programa. El nombre de las funciones se divide en varias partes, según se explica en la figura:

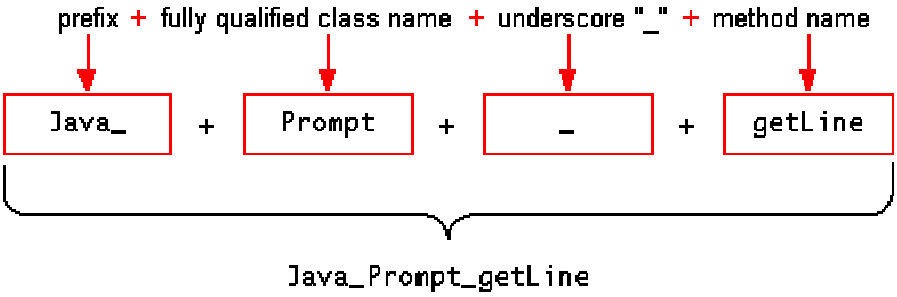


Figura 27: Nomenclatura de métodos JNI

Además cada función nativa, independientemente del número de parámetros que contenga, tendrá también los parámetros `JNIEnv *env, jobject obj` que son parámetros que usa Java y que no tienen ninguna utilidad en este momento.

Paso 4: Escribir la implementación del método nativo.

Para la implementación de los métodos nativos hay que recurrir a los ya casi olvidados a estas alturas ficheros generados por el CMD. Para ello se ha usado el programa Microsoft Visual Studio v. 6.0 que es en realidad un entorno de trabajo para C++ pero que es apto para nuestro PFC.

Lo más importante es que las definiciones del nombre de la función Java coincidan con las de la función C, para lo cual deberemos valernos del archivo `.h` generado en el paso anterior. Así la siguiente figura:

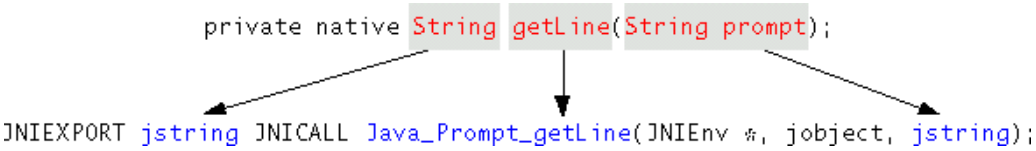


Figura 28: Nomenclatura de clases JNI

muestra como debe hacerse la conversión. La implementación final de las funciones queda como sigue (para la estación 1, y de igual forma para todas las demás):

```
/*
 * Class:     Principal
 * Method:    LeeEntradas
 * Signature: ()V
 */
JNIEXPORT jboolean JNICALL Java_VariableBooleana_leeEntrada
(JNIEnv *env, jobject obj, jint posicion) {
        switch (posicion) {
                case 0: return          Cinta_atras;
                case 1: return          Cinta_adelante;
                case 2: return          Pinza_izda;
```

```
                case 3: return          Pinza_drcha;
                case 4: return          Pinza_arriba;
                case 5: return          Pinza_abajo;
                case 6: return          Cargador_adelante;
                case 7: return          Cargador_atras;
                case 8: return          Emergencia;
                case 9: return          Marcha;
                case 10: return          Manual_automatico;
                case 11: return          Ind_int;
                case 12: return          Rearme;
                case 13: return          Inductivo_camisa;
                case 14: return          Optico_camisa;
                case 16: return          Capacitivo_camisa;
                case 17: return          Lector_adelante;
                case 18: return          Lector_atras;
                case 19: return          Optico_lector;
        }
}

JNIEXPORT void JNICALL Java_VariableBooleana_escribeSalida
(JNIEnv *env, jobject obj, jboolean valor, jint posicion) {
        switch (posicion) {
                case 0:  Cinta_avanza = valor; break;
                case 1:  Cinta_retrocede = valor; break;
                case 2:  Pinza_fuera = valor; break;
                case 3:  Pinza_dentro = valor; break;
                case 4:  Pinza_sube_baja = valor; break;
                case 5:  Cargador = valor; break;
                case 6:  Pinza = valor; break;
                case 7:  Lector = valor; break;
        }
        return;
}
```

Ahora se revela claramente el porque de asignar un número a cada variable. Dicho número sirve para hacer coincidir la variable Java con la variable C (que en realidad no es una variable booleana "tal cual", sino una variable del tipo `T_IBS_BOOL` definido en las librerías HLI). **Por tanto es imprescindible para un correcto funcionamiento del programa la correspondencia total entre la asignación de números en la clase Java y en la función C**. De igual forma se implementan las funciones específicas de la estación:

```
JNIEXPORT void JNICALL
      Java_Estacion1_FinalizaEstacion(JNIEnv *env, jobject obj) {
              IBS_HLI_ResetAllOutputs(PCISC1);
              IBS_HLI_Exit(PCISC1);
              return;
      }
JNIEXPORT void JNICALL
    Java_Estacion1_InicializaEstacion(JNIEnv * env, jobject obj) {
  HLIRET ret;

  /* Call the HLI Initialization function */
  ret = IBS_HLI_Init_CFG(PCISC1, & PCI1, IBS_STANDARD, 12,
PCI1_DeviceList);
  if (ret == HLI_OKAY) {
    /* --- Process data object registration --- */
```

```
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 1,
IBS_PDO_BOOL, 0, 0, 1, & Cinta_atras, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 1,
IBS_PDO_BOOL, 0, 1, 1, & Cinta_adelante, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 1,
IBS_PDO_BOOL, 0, 2, 1, & Pinza_izda, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 1,
IBS_PDO_BOOL, 0, 3, 1, & Pinza_drcha, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 1,
IBS_PDO_BOOL, 0, 4, 1, & Pinza_arriba, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 1,
IBS_PDO_BOOL, 0, 5, 1, & Pinza_abajo, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 1,
IBS_PDO_BOOL, 0, 6, 1, & Cargador_adelante, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 1,
IBS_PDO_BOOL, 0, 7, 1, & Cargador_atras, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_OUTPUT, 1, 2,
IBS_PDO_BOOL, 0, 0, 1, & Cinta_avanza, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_OUTPUT, 1, 3,
IBS_PDO_BOOL, 0, 0, 1, & Cinta_retrocede, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_OUTPUT, 1, 4,
IBS_PDO_BOOL, 0, 0, 1, & Pinza_fuera, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_OUTPUT, 1, 5,
IBS_PDO_BOOL, 0, 0, 1, & Pinza_dentro, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_OUTPUT, 1, 6,
IBS_PDO_BOOL, 0, 0, 1, & Pinza_sube_baja, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_OUTPUT, 1, 7,
IBS_PDO_BOOL, 0, 0, 1, & Cargador, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_OUTPUT, 1, 8,
IBS_PDO_BOOL, 0, 0, 1, & Pinza, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_OUTPUT, 1, 9,
IBS_PDO_BOOL, 0, 0, 1, & Lector, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10,
IBS_PDO_BOOL, 0, 0, 1, & Emergencia, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10,
IBS_PDO_BOOL, 0, 1, 1, & Marcha, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10,
IBS_PDO_BOOL, 0, 2, 1, & Manual_automatico, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10,
IBS_PDO_BOOL, 0, 3, 1, & Ind_int, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10,
IBS_PDO_BOOL, 0, 4, 1, & Rearme, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10,
IBS_PDO_BOOL, 0, 5, 1, & Inductivo_camisa, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10,
IBS_PDO_BOOL, 0, 6, 1, & Optico_camisa, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10,
IBS_PDO_BOOL, 0, 7, 1, & Capacitivo_camisa, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 11,
IBS_PDO_BOOL, 0, 0, 1, & Lector_adelante, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 11,
IBS_PDO_BOOL, 0, 1, 1, & Lector_atras, NULL);
    IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 11,
IBS_PDO_BOOL, 0, 2, 1, & Optico_lector, NULL);
    /* reset all outputs */
    IBS_HLI_ResetAllOutputs(PCISC1);
    /* Start bus now */
    ret = IBS_HLI_StartBus(PCISC1);
  }

  return;
```

```
}
```

Mención aparte merece la función `Java_Estacion1_InicializaEstacion`. La principal llamada dentro de la función es:

```
ret = IBS_HLI_Init_CFG(PCISC1, & PCI1, IBS_STANDARD, 12,
PCI1_DeviceList);
```

Esta función es la que realmente inicializa las comunicaciones entre la tarjeta y los módulos. El modo `IBS_STANDARD` indica que la inicialización se ha realizado en modo normal, en contraposición con el modo `IBS_CONTROLLED` en la que el control de tiempos se ha de realizar directamente desde el programa.

Como la función es la que se encarga de comenzar la comunicación entre la tarjeta controladora de bus y los dispositivos conectados, para ello lo primero que hace es registrar cada variable, es decir, asociar cada variable con una dirección física del dispositivo correspondiente. Por ejemplo la función

```
IBS_HLI_RegisterPDObject(PCISC1, IBS_PDO_INPUT, 1, 10, IBS_PDO_BOOL,
0, 7, 1, & Capacitivo_camisa, NULL);
```

registra dentro del dispositivo PCISC1 una variable de entrada (`IBS_PDO_INPUT`). La variable `Capacitivo_camisa` es una variable de tipo `T_IBS_BOOL` que ha sido definida con de forma global. La función enlaza el valor del sensor capacitivo de la estación 1 con la dirección (uso de punteros con el símbolo `&`) de la variable `Capacitivo_camisa`, que a su vez gracias a la función `leeEntradas` será transmitida cuando sea necesario al programa Java.

Todas las funciones C con el prefijo IBS_ son funciones definidas e implementadas dentro de las librerías HLI, y no es necesario saber nada acerca de su funcionamiento interno. Estas funciones se hacen accesibles a través de la etiqueta:

```
#include <jni.h>
#include "Estacion1.h"
#include "g4hliw32.h"
```

El programa C debe contener al menos estos enlaces: un enlace a las librerías jni, un enlace al fichero de cabecera correspondiente (que hemos conseguido en el paso 3) y el enlace a la librería de funciones de HLI que son la que realizarán todo el trabajo interno de gestión de las comunicaciones entre la tarjeta y la estación.

Las funciones nativas que se usan en el código se basan en las funciones conseguidas gracias al CMD, y la implementación interna que proporciona el HLI es la que gestiona las comunicaciones entre la tarjeta y los módulos

Paso 5: Crear una librería compartida.

Una vez se ha terminado la implementación de las funciones en C se debe compilar el resultado dentro de una librería dinámica, es decir, un archivo `.DLL`. Este archivo es el que se carga en la clase Java al ejecutarse:

```
// Carga de la librería dinámica que contiene la implementación de los
// métodos nativos y de las funciones especificas de control de la
// tarjeta, incluida con el software HLI
  static {
    System.loadLibrary("Estacion1");
  }
```

`Estacion1.dll` es el archivo resultante de la compilación del archivo en que están contenidas las implementaciones de las funciones nativas.

Paso 6: Ejecutar el programa

Para ejecutar el programa se usa la aplicación `java` o `javaw`. Hay que tener cuidado sobre todo con tener muy claro donde tenemos todos los componentes necesarios para que el programa funcione, ya que es muy sencillo que el programa no se ejecute debido a que la máquina virtual Java no encuentra alguno de los archivos o clases necesarios para el buen funcionamiento del programa.

Uno de los errores más comunes que pueden aparecer a la hora de ejecutar el programa es el siguiente, que ocurre cuando el library path de Java no esta bien configurado:

```
java.lang.UnsatisfiedLinkError: no Estacion1 in shared
library path
    at java.lang.Runtime.loadLibrary(Runtime.java)
    at java.lang.System.loadLibrary(System.java)
    at java.lang.Thread.init(Thread.java)
```

Para solucionarlo hay que conseguir que aparezcan en el library path los directorios donde se almacenan las clases java y la librería dinámica.

4. Redes de Petri

4.1 Introducción

Los sistemas de eventos discretos (SED) se componen de varios tipos de nodos que interactúan entres sí. Cada nodo puede ser a la vez un sistema por sí mismo y un componente de un SED. Estos componentes pueden operar concurrentemente, es decir, un componente puede estar realizando una de sus funciones al mismo tiempo que otro. Considérese el caso de un cajero automático conectado a una red de un banco. Varias personas pueden estar realizando operaciones sobre sus cuentas al mismo tiempo sin interferir unas con otras. Pero pongámonos en el caso de una pareja que posee una cuenta conjunta que trata de acceder a su cuenta desde dos cajeros distintos al mismo tiempo, uno de ellos para hacer un ingreso y el otro para sacar dinero. Aparecen entonces problemas de sincronización, al no haber un orden predeterminado para los dos eventos, la salida y el ingreso. El orden en que se realicen puede llevar a resultados diferentes; los bancos han desarrollado unas normas detalladas que tienen en cuenta el tiempo y la secuencia de los ingresos y las salidas. ¿Pero cómo se pueden describir estos problemas de una manera precisa, sin ningún tipo de ambigüedad?

La necesidad de resolver esta pregunta llevó a Carl Adam Petri a introducir en su tesis doctoral en 1962 una clase especial de grafos o redes llamados hoy en día Redes de Petri. Son una herramienta de modelado y análisis especialmente adaptada para el estudio de sistemas de eventos discretos. El uso de redes de Petri lleva a una descripción matemática de la estructura del sistema que puede ser investigada analíticamente.

Este proyecto pretende implementar un sistema de control mediante redes de Petri en lenguaje Java. Por lo tanto el código se presentará como consecuencia directa de las definiciones.

4.2 Principios fundamentales

Dado que una RdP es un tipo especial de grafo, debe comenzarse su explicación mediante unas pocas nociones de teoría de grafos. Un grafo consta de nodos y líneas, y la manera en que están interconectados. Si el par de nodos conectados por una línea está ordenado, entonces la línea es dirigida y se coloca una flecha en su extremo para indicar la dirección. Si todos las líneas de un sistema tienen una orientación, entonces el grafo se denomina grafo orientado. Dos nodos conectados por una línea se llaman nodos adyacentes. La representación de un nodo puede hacerse de diversas maneras: como un circulo, un rectángulo, u otro símbolo conveniente. Cuando un grafo contiene líneas paralelas, por ejemplo, líneas que conectan el mismo par de nodos, y que si son dirigidas tienen la misma dirección, se llama un multigrafo.

Las redes de Petri son multigrafos, considerando solamente redes de Petri ordinarias. En muchas aplicaciones las propiedades como multigrafo de las redes de Petri pueden ser una gran ventaja. Sin embargo, introducen una gran complejidad en notación y otros aspectos que pueden ser tratados mejor mediante extensiones de las redes de Petri ordinarias tales como las redes de Petri coloreadas.

Una segunda característica de las redes de Petri en tanto que grafos es que son grafos bipartidos. Esto significa que hay dos tipos de nodos. Para distinguir entre ellos se usarán diversos símbolos. Por convención, el primer tipo de nodo se denomina estado (o lugar) y se representa mediante un círculo o elipse. El segundo tipo se llama transición y se representa mediante una barra sólida, o un rectángulo. Las líneas de una RdP se llaman arcos y son siempre dirigidas. Los símbolos se muestran en la siguiente figura:

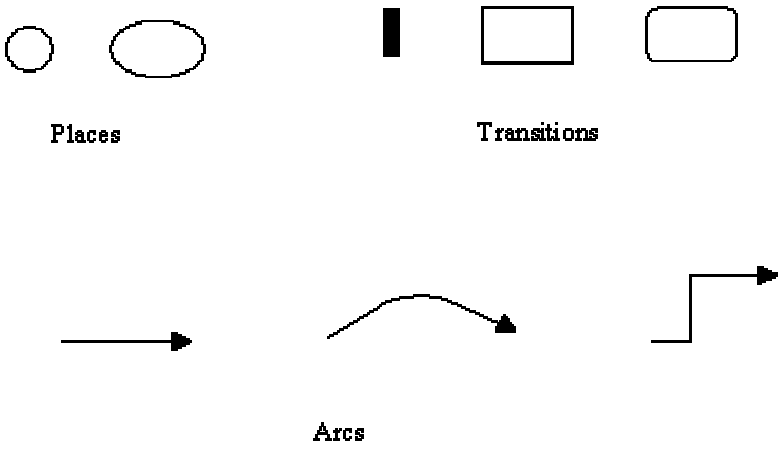


Figura 29: Tipos de lugares, arcos y transiciones

Un grafo bipartido tiene una propiedad especial: una línea sólo puede conectar dos nodos que sean de tipos diferentes; es decir, sólo puede haber un arco desde una transición a un estado o viceversa pero nunca de un estado a otro o de una transición a otra.

4.3 Definiciones básicas.

Una red de Petri (RdP) es un grafo bipartido orientado representado por la cuádrupla R = (P, T, α, β) tal que

- P es un conjunto finito y no vacío de lugares
- T es un conjunto finito y no vacío de transiciones
- $\alpha = P \times T \rightarrow \mathbb{N}$ es la función de incidencia previa.
- $\beta = T \times P \rightarrow \mathbb{N}$ es la función de incidencia posterior.

Una RdP se representa gráficamente por un grafo bipartido orientado. Existe un arco que va del lugar p_i a la transición t_j si $\alpha(p_i, t_j) \neq 0$. Análogamente, existe un arco que va de la transición t_k al lugar p_i si $\beta(t_k, p_i) \neq 0$. Cada arco se etiqueta con un número natural, $\alpha(p, t)$ o $\beta(t, p)$ que se denomina peso del arco. Por convenio, un arco no etiquetado posee un peso unitario. Para facilitar la legibilidad, todo arco cuyo peso sea superior a la unidad se dibuja normalmente con un trazo grueso, o con dos o más trazos paralelos.

Una RdP se representa matricialmente por medio de dos matrices. Sea $|P| = n$ (número de lugares de la red) y sea $|T| = m$ (número de transiciones de la red). Se denomina *matriz de incidencia previa* a la matriz

$$\boldsymbol{C}^- = [\boldsymbol{c}^-_{ij}]_{n \times m},$$

en la que $\boldsymbol{c}^-_{ij} = \alpha(p_i, t_j)$. Se denomina *matriz de incidencia posterior* a la matriz

$$\boldsymbol{C}^+ = [\boldsymbol{c}^+_{ij}]_{n \times m},$$

en la que $\boldsymbol{c}^+_{ij} = \beta(t_k, p_i)$.

Es decir, en las matrices de incidencia los lugares numera las filas (i) y las transiciones las columnas (j), y cada elemento (i, j) expresa la incidencia que el lugar i tiene sobre la transición j.

Una *red es ordinaria* si sus funciones de incidencia sólo pueden tomar los valores 0 y 1, es decir, que todos los arcos de la red tienen peso 1.

$\alpha(p, t) \in \{0, 1\}$

$\beta(t, p) \in \{0, 1\}$

Una red es pura si ninguna transición contiene un lugar que sea simultáneamente de entrada y de salida.

$\forall\, t_j \in T, \forall\, p_i \in P \quad \alpha(p_i, t_j)\, \beta(t_j, p_i) = 0$

La representación matricial de una red pura se simplifica definiendo una única matriz , ***C***, denominada matriz de incidencia.

$\boldsymbol{C} = \boldsymbol{C}^{+} - \boldsymbol{C}^{-} \Rightarrow \boldsymbol{c}_{ij} = \{ \beta (t_j, p_i)$ si es no nula , $-\alpha(p_i, t_j)$ si es no nula , 0 en cualquier otro caso$\}$

De esta forma, en la matriz ***C*** un elemento positivo indica incidencia posterior y uno negativo señala incidencia previa. Un elemento nulo en ***C*** indica que la transición y lugar correspondientes no están conectados directamente a través de un arco.

Pongamos el siguiente ejemplo. Sea la RdP de la figura:

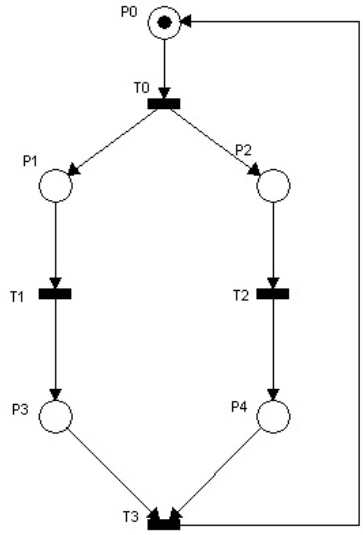


Figura 30: RdP ejemplo

Sus matrices de incidencia serán (dado que es una red pura):

matriz de incidencia previa	matriz de incidencia posterior	matriz de incidencia
0 0 0 -1	1 0 0 0	(1 0 0 -1)
-1 0 0 0	0 1 0 0	(-1 1 0 0)
-1 0 0 0	0 0 1 0	(-1 0 1 0)
0 -1 0 0	0 0 0 1	(0 -1 0 1)
0 0 -1 0	0 0 0 1	(0 0 -1 1)

El *marcado M* de una red ***R*** es una aplicación de *P* en ℕ, es decir, la asignación de un número natural (número de marcas) a cada lugar. En el grafo asociado a ***R***, el marcado ***M*** se representa por una distribución, en los lugares, de objetos denominados *marcas*. Una marca se representa gráficamente por un punto en el interior de la circunferencia que define el lugar que la contiene. Si $| P | = n$ entonces un marcado se representa, en forma matricial, por un vector de *n* elementos: $M(p_i)$ (*vector de marcado*).

Una *red de Petri marcada* es el par (***R, M₀***) en el que ***R*** es una red de Petri y ***M₀*** es un *marcado inicial*. La evolución del marcado le confiere a la RdP marcada un comportamiento dinámico que permite modelar evoluciones simultáneas de sistemas discretos.

Una transición $t \in T$ está *sensibilizada* por el marcado ***M*** si cada uno de sus lugares de entrada posee al menos $\alpha(p, t)$ marcas. Es decir, se exige que para todo lugar p perteneciente al conjunto de lugares de entrada de una transición t $\mathbf{M}(p) \geq \alpha(p, t)$.

Regla de evolución del marcado: *Disparar una transición sensibilizada o habilitada t* es la operación que consiste en eliminar $\alpha(p, t)$ marcas a cada lugar de entrada de t y añadir $\beta(t, p)$ marcas a cada lugar de salida de t. El marcado $\mathbf{M}_i$ evolucionará a otro marcado $\mathbf{M}_j$.

Volviendo a la RdP del ejemplo el marcado inicial puede ser representado por el vector (1 0 0 0 0), por tanto solamente la transición T0 está habilitada para ser disparada. Si se produce el disparo se quitará una marca de P0 y se añadirán una marca a P1 y P2, estados de salida. El nuevo marcado será (0 1 1 0 0) y tanto T1 como T2 estarán habilitadas.

Una *transición es viva*, para un marcado $\mathbf{M}_0$, si para todo marcado ***M*** que se pueda alcanzar a partir del marcado inicial $\mathbf{M}_0$ existe un marcado ***M***' sucesor de ***M*** a partir del cual se puede disparar esa transición. Una *RdP es viva* para un marcado dado si todas sus transiciones son vivas para ese marcado. Si una RdP es no viva para un marcado puede sospecharse que el modelado del sistema objeto de estudio es incorrecto.

Para un marcado inicial dado, una RdP es *binaria* si cualquier marcado alcanzable es tal que ningún lugar posee más de una marca. En una RdP binaria todo lugar estará marcado con una marca o no estará marcado.

Para un marcado inicial dado, se dice que una RdP es conforme si es binaria y viva. Debido a su marcado inicial una RdP puede ser conforme, no binaria o no viva.

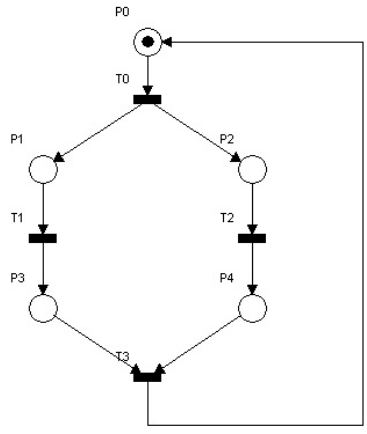


Figura 31: RdP Conforme

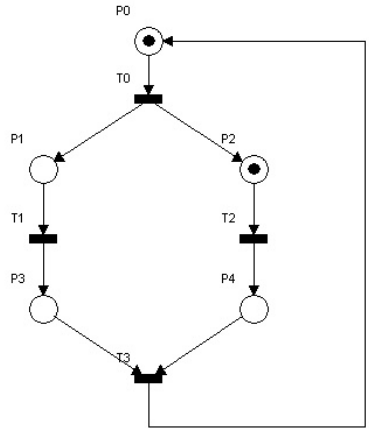


Figura 32: RdP no binaria

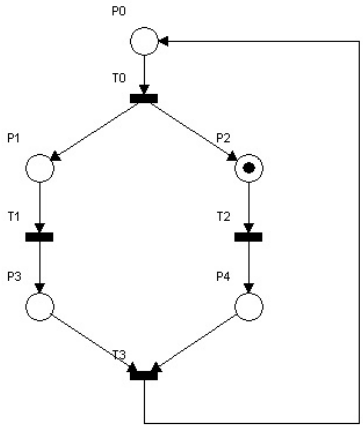


Figura 33: RdP no viva

Dos o más transiciones simultáneamente sensibilizadas están en *conflicto* si descienden de un mismo lugar y éste no dispone de un número de marcas suficientes para dispararlas simultáneamente. Un conflicto se hace *efectivo* si los eventos asociados a las transiciones en conflicto se verifican simultáneamente.

4.4 Creación de las RdP del proyecto

Lo primero que hay que explicar antes de comenzar ninguna explicación es que dada la complejidad en cuestión de programación y establecimiento de una base consistente este proyecto se habría alargado extremadamente si hubiera que haber partido de cero a la hora de crear las RdP de la célula. Sin embargo dado que el lenguaje Grafcet de programación de autómatas es una variante de las redes de Petri las redes modeladas para este proyecto son descendientes adaptadas y optimizadas de estas.

El control de la célula mediante Grafcet en PL7Pro posee una excesiva complejidad, lo que provocó que muchas de ellas resultaran inservibles o redundantes.

Para las redes de control automático de las estaciones se usaron casi directamente la estructura del Grafcet con sólo ligeras modificaciones surgidas de la prueba junto con el coordinador de la misma. Debe tenerse en cuenta que la RdP no puede contener absolutamente toda la información sobre el sistema que define, lo que provoca pequeños problemas. Por ejemplo, un lugar de la RdP puede activar una salida cuando está marcado (acción previa) pero dependiendo de si la salida es del tipo todo o nada (por ejemplo, las salidas pinza-sube-baja) o con dos variables complementarias para una misma salida (por ejemplo, Cinta-adelante y Cinta-atrás) la desactivación de la salida podrá realizarse cuando se desactive el lugar (acción posterior) o en otro lugar de la RdP.

Para la implementación de la red total se usó el siguiente esquema repetido junto con una inicialización previa de los palets:

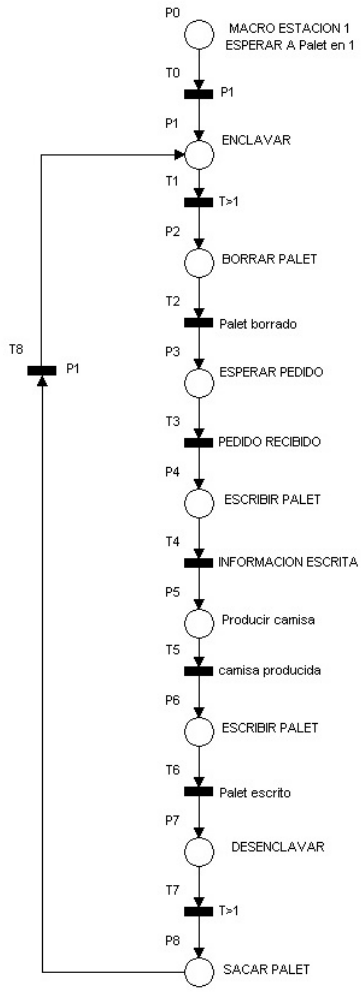


Figura 34: Sacar palets

Este esquema se repite para las cuatro estaciones de la célula con las modificaciones particulares de cada una de ellas; por ejemplo, a pesar de que en el esquema se ve que la estación 1 borra los palets que le llegan esto se modificó más tarde para que fuera la estación 4 la que borrara los palets completados y que la estación 1 reaccionara ante palets ocupados previamente.

La red mostrada es una RdP a un nivel de definición muy básico. Obviamente cada uno de los estados representa a su vez una red completa. Por ejemplo, todos los procesos por los que un palet sale de una estación tienen el mismo esquema:

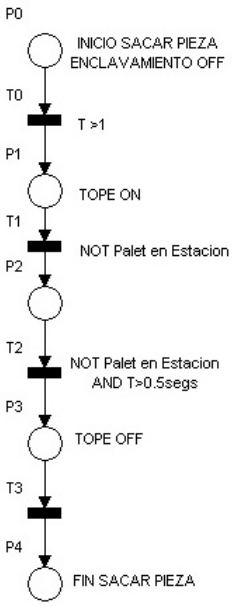


Figura 35: RdP para sacar palet de las estaciones

Quizá el principal problema de la implementación del coordinador sea que no admite el concepto de "subredes"; por tanto, si quisiéramos, como será el caso, controlar la salida de los palets de las estaciones, incluyendo el desvió intermedio, se deberá cortar y pegar el esquema anterior cinco veces. Aunque esto pueda parecer sencillo debe tenerse en cuenta que el HPSim no introduce el concepto de subred, con lo que es imposible a la hora de leer el archivo de texto producido saber si un estado representa una red en sí mismo. Por tanto o se crea la red sin el HPSim, lo que para un proyecto medio es una tarea titánica, o se recurre al socorrido copiar y pegar incluso a la hora de crear el coordinador; en efecto, dado que la red es la misma, las condiciones de disparo de las transiciones y las acciones asociadas a los lugares serán también las mismas, sólo que habrá que copiarlas en el coordinador las veces que aparezcan en la RdP (cada una de ellas tendrá una numeración distinta).

Sin embargo se ha conseguido la funcionalidad de las subredes aplicadas al proceso completo de producción de cada estación mediante el uso de "subcoordinadores". Así pues, a la hora de realizar el borrado de los palets de la célula, el coordinador principal (`CoordinadorCelula`) lanza los timers del coordinador `CoordinadorBorrar` que se ejecuta hasta que alcanza su estado final, señalado mediante una variable booleana pública "finalizado". Al finalizarse, el `CoordinadorBorrar` para sus timers. Obviamente el `CoordinadorBorrar` ha sido implementado de forma que lo primero que hace es cargar su propia subred. Por tanto, el `CoordinadorCelula` no interviene en la evolución de la RdP durante este proceso, sino que deja la responsabilidad de ello al `CoordinadorBorrar` hasta que este acaba. Es importante señalar que el `CoordinadorCelula` no deja de ejecutarse mientras se ejecuta el `CoordinadorBorrar`, lo que pasa es que la condición que debe cumplir para que dispare la transición es precisamente que la variable booleana `finalizado` del `CoordinadorBorrar` marque que el `CoordinadorBorrar` ha acabado su ejecución. Como posible extensión posterior del código se deja la posible implementación de esta funcionalidad de un modo más formal, como por ejemplo la definición de una clase `SubCoordinador` con las características antes definidas. Aunque en las RdP se represente

como un estado con un círculo en su interior el HPSim lo interpreta como un dos estados independientes. Por tanto es importante no confundirnos a la hora de dibujar la RdP y unir el estado que representa el coordinador secundario con sus transiciones.

La implementación completa del control de la estación mediante una RdP posee actualmente 140 estados y 157 transiciones. Estas se han dividido hasta conseguir 6 redes controladas por sus correspondientes coordinadores. Las redes más representativas se pueden ver en el Anexo 2.

Esta versión utiliza cuatro subcoordinadores, uno para cada estación, que se ejecutan a la vez hasta que se llega al final de la producción. Esto, que a primera vista podría parecer que mejoraría el rendimiento del programa, lo que consigue en realidad es una ralentización del mismo. Además el coordinador principal debe seguir ejecutándose mientras se ejecutan los subcoordinadores, lo que resulta en otra pequeña pérdida de eficacia. El precio que impone Java a garantizar el buen funcionamiento de programas concurrentes es una menor eficiencia en los mismos. Para más información acerca de este tema se puede recurrir a alguno de los interesantes artículos web recogidos en la bibliografía.

La pregunta obvia es: si el uso de múltiples coordinadores ralentiza el programa, ¿por qué no dejar un solo `CoordinadorCelula`? Pues porque el tener cuatro coordinadores, uno para cada estación, permite una mayor facilidad de cambios y modificaciones en el programa, además de abrir camino a otras posibles mejoras, como un control mucho más estricto sobre las emergencias.

4.5 Implementación programada de RdP

Una vez revisados los conceptos básicos acerca de redes de Petri se puede comenzar a explicar el código implementado para la aplicación. Obviamente lo primero que debemos conocer acerca de una RdP son sus estados, transiciones y conexión entre ambos mediante arcos. La primera decisión importante que se tomó fue decidir si se implementaban los arcos o no, con lo que habría sido conveniente la creación de una superclase "`Nodo`" de la que descendieran `Estado, Transición y Arco`. Dado que los arcos introducirían una complejidad innecesaria dado que no se planteó la creación de un entorno gráfico de edición de RdP, se decidió crear la funcionalidad de los arcos mediante el atributo de la clase `Red matrizDePesos`, que examinaremos al estudiar la clase `Red`.

Estado

La clase `Estado` es la primera clase necesaria para la implementación programada de una RdP. Como ya se ha explicado, en estado o lugar representa, junto con el marcado, el estado en que se encuentra una RdP. He preferido el nombre `Estado` en vez de `Lugar` porque resulta más intuitivo y consecuente con la nomenclatura de sistemas de eventos discretos, basadas en el binomio estado-transición. La clase `Estado` posee los siguientes atributos:

```
int tokens = 0;
int ticks = 0;
Temporizador temporizador;
String nombre;
```

Obviamente según la definición un lugar o estado de una RdP se caracteriza por el número de marcas. He usado el término inglés tokens por comodidad. Aquí, al igual que se verá en la clase `Transicion`, no se ha creado un campo que determine el código a ejecutar durante la activación, desactivación y mantenimiento del estado; está responsabilidad se dejará a la clase `Coordinador`. El nombre es un añadido que permite diferenciar los estados. La clase `Temporizador` tiene como función implementar un control de tiempo de los estados. Aunque a priori no sería necesario implementar una funcionalidad para controlar el tiempo, la aplicación de esta implementación de RdP al control de un proceso industrial concreto (en este caso la célula de fabricación, aunque podría extenderse a otros controles) planteaba la necesidad de un cierto espacio de tiempo entre el establecimiento de varias salidas, sobre todo teniendo en mente la seguridad de los operarios y de las diversas máquinas a utilizar, o también a la espera de un determinado intervalo de tiempo en espera de producirse un determinado evento. El que controlará el establecimiento de esta temporización será el coordinador de la red. Los atributos y métodos que implementa la clase temporizador son:

```
int ticks;
Timer timer;
```

La clase `Timer` corresponde a la clase `java.util.Timer` y no a `javax.swing.Timer`. La diferencia entre ambas es que la segunda está más orientada hacia la programación de interfaces de usuario y la primera posee más propiedades. La clase `Temporizador` hace que se ejecuten a la vez una

gran cantidad de hilos, uno por cada estado activo. Si este número fuera muy alto esto podría llegar a interferir con el correcto funcionamiento de la pantalla gráfica. La clase Temporizador posee los métodos para parar, resetear y lanzar la temporización así como para seleccionar la unidad de tiempo a medir:

```
  public void comienzaContar(int periodo)
  public void paraContar()
  public void reset()
  public int getTicks()
```

Para conseguir que el temporizador cumpla su función debemos crear una subclase de `TimerTask` que será la que contenga el código que debe realizar el temporizador:

```
  class TareaTemporizador
      extends TimerTask {
    /**
     * Método run de la clase TimerTask.
     */
    public void run() {
      ticks++;
    }
  }

public class Temporizador {
...
  public void comienzaContar(int periodo) {
    timer = new Timer();
    if (periodo > 0) timer.scheduleAtFixedRate(new
TareaTemporizador(), 0, periodo);
    else timer.scheduleAtFixedRate(new TareaTemporizador(), 0,
DECIMAS);
  }
...
}
```

Por defecto el temporizador actualiza el valor de los ticks cada décima de segundo. Por ejemplo, si queremos saber si un `Estado` lleva activo más de un segundo tendríamos que poner una instrucción del tipo: `Estado.getTicks() > 10`.

La clase `Estado` posee los siguientes métodos implementados:

```
  public Estado(int toks)
  public void setTokens(int t) throws TokensFueraDeRangoException
  public int getTokens()
  public boolean isMarcado()
  public void startTemporizacion(int periodo)
  public void stopTemporizacion()
  public void resetTemporizador()
  public int getTiempoMarcado()
  public void setNombre(String nom) {
  public String getNombre() {
```

Todos los métodos se explican por sí mismos y no poseen complejidad alguna. Los métodos que influyen en el temporizador sólo encapsulan llamadas a los métodos homólogos de la clase Temporizador. La función `isMarcado()` verdadero cuando el estado tiene una o más marcas.

Transición

La otra clase fundamental en la que se basa una RdP es la clase `Transicion`. En este modelo las transiciones representan condiciones que se cumplen (o no). La principal dificultad a la hora de implementar la clase `Transicion` es dilucidar que funcionalidades debe tener y cuales son propias de la `Red`. Por ejemplo, una de las decisiones que se tomaron fue la de no implementar ningún campo del tipo "`condicionDeDisparo`", ya que eso obligaría a que cada Transición tuviera que ser creada con un código específico. Sin embargo, si se considerara la extensión de este código mediante la creación de un editor gráfico de RdP en el que las condiciones de disparo de la transición pudieran integrarse dentro del editor mediante el uso de menús, esta funcionalidad debería ser implementada para evitar la larga extensión de los métodos de la clase `Coordinador` encargados de determinar si una transición está habilitada.

Otra de las dificultades a las que tuvo que enfrentarse el diseño de la estructura fue la de cómo implementar la "unión" entre los diferentes componentes de la RdP. Como ya se ha explicado, la funcionalidad de los arcos puede fácilmente traspasarse creado un atributo de la clase `Red` que fuera la matriz de pesos de la RdP. Sin embargo a la hora de crear una RdP, y antes de implementar la función de lectura de un archivo de texto, se llegó a la conclusión de que la forma más fácil de crear una RdP era crear un conjunto de objetos `Estado` y `Transicion` y una vez ordenados en un `Vector` deducir de ellos las correspondientes matrices de incidencia, conflictos, etc... Para ello bien las transiciones deberían conocer sus estados de entrada y salida o bien los estados deberían conocer sus transiciones de entrada y salida. La primera opción se consideró más intuitiva además de que facilita la implementación de métodos que determinen la habilitación y permitan el disparo de la transición, y por eso fue la finalmente adoptada. Así pues, los atributos de la clase `Transicion` son:

```
boolean habilitada = false;
Vector < Estado > lugaresEntrada;
Vector < Estado > lugaresSalida;
int prioridad;
String nombre;
```

Una transición puede estar habilitada o no habilitada dependiendo de si sus lugares de entrada están marcados o no. El campo prioridad fue un añadido posterior que permite al coordinador determinar en caso de conflicto que transición debe ser disparada; cuanto más alta sea la prioridad, más probabilidad tendrá la transición de ser la que se dispara. Por defecto las transiciones se crean con prioridad media (valor de 50) y se recomienda el uso de los modificadores de la clase:

```
public static final int PRIORIDAD_MAXIMA = 100;
public static final int PRIORIDAD_ALTA = 75;
public static final int PRIORIDAD_NORMAL = 50;
public static final int PRIORIDAD_BAJA = 25;
public static final int PRIORIDAD_MINIMA = 0;
```

Los métodos que implementa la clase son:

```
public Transicion(Vector < Estado > lEnt, Vector < Estado > lSal)
public Transicion(Estado estEnt, Estado estSal)
public int getNumLugaresEntrada()
public int getNumLugaresSalida()
public Estado getLugarEntrada(int orden) throws EstacionesException
public Estado getLugarSalida(int orden) throws EstacionesException
public Vector<Estado> getLugaresEntrada()
public Vector<Estado> getLugaresSalida()
public boolean estaHabilitada()
public void setHabilitacion(boolean valor)
public int getPrioridad()
public void setPrioridad(int priorid)
public void setNombre(String nom) {
public String getNombre() {
```

Se ha dotado a la clase de dos constructores, uno para la definición clásica (varios lugares de entrada y/o salida) y otro para el caso de que la transición sólo tenga un único lugar de entrada y otro de salida. El método `estaHabilitada()` sólo devuelve el valor de la variable `habilitada`, no incluye código que determine si lo está o no; esto se deja para el coordinador.

Red

Una vez que tenemos la implementación tanto de lugares como de transiciones puede procederse a la implementación de la clase `Red`.

En primer lugar, y antes de pasar al código, hay que aclarar el sentido de "red" con el que se trabajó. La RdP o red con la que vamos a trabajar es la representación gráfica y analítica de un sistema. Sin embargo a la hora de realizar el proyecto surge la necesidad de diferenciar entre la implementación de la RdP y la implementación del control del sistema modelado por la misma. Por poner un ejemplo concreto, para controlar la célula de fabricación su RdP va a ser pura, es decir, no va a encontrarse ningún lugar que sea a la vez entrada y salida de una misma transición. Sin embargo la implementación de la clase `Red` que veremos a continuación permite este tipo de estructuras. Por eso se llegó a la conclusión de que para separar las funcionalidades de la RdP de las de su aplicación al control de un sistema debería crearse un binomio: la clase `Red` se encarga de la parte de "estructura" mientras que la clase `Coordinador` se encargará de la evolución de la RdP y del control del sistema. Es por eso que es el coordinador el que se encarga de disparar las transiciones actualizando los valores de las marcas de los estados, vector de transiciones habilitadas, etc... Toda la funcionalidad ausente pues en la implementación de la clase `Red` se encontrará en la clase `Coordinador`.

La clase `Red` posee los siguientes atributos:

```
public int[][] matrizIncidenciaPrevia;
public int[][] matrizIncidenciaPosterior;
public int[] marcado;
public int[] marcadoInicial;
public int[][] matrizPesos;
public Vector < Conflicto > conflictos;
public Vector < Estado > estados;
public Vector < Transicion > transiciones;
```

Como puede verse la implementación de la clase sigue al pie de la letra la definición teórica de lo que una RdP debe ser. Todos los atributos tienen su razón de ser según la teoría. Dado que se ha procurado ser consecuente con la nomenclatura, la explicación de lo que representa cada atributo se puede encontrar en la parte de explicación de los fundamentos de una RdP.

```
  public Red() {
  public Red(Vector < Estado > estad, Vector < Transicion > trans) {
  public void construyeRedMatrizIncidencia(int[][] matrizInc, int[]
marcado, Vector<String> vectorNombresEstados, Vector<String>
vectorNombresTransiciones) {
  public void construyeRedVectores(Vector < Estado > estad,
                                   Vector < Transicion > trans) {
  private void iniciaRed(int lugares, int transiciones) {
  private void inicializaConflictos() {
  public void volverMarcadoInicial() {
  public void setMarcado() {
  public void setMarcado(int[] vectorMarcado) {
  public void setMarcadoInicial(int[] vectorMarcado) {
  public void setEstados(Vector < Estado > e) {
  public void setTransiciones(Vector < Transicion > t) {
  public Transicion getTransicion(int index) {
  public Estado getEstado(int index) {
  public Estado getEstado(String nombre) {
  public void imprimeTicks() {
  public boolean esTransicionConflictiva(Transicion t) {
  public Vector <Conflicto> getConflictosConEstaTransicion(Transicion
t) {
  public Vector <Estado> getEstadosMarcados() {
```

Esta es una clase atípica porque los verdaderos constructores son los métodos `construyeXxx`. En efecto, el constructor sin parámetros está vacío mientras que el segundo constructor es una llamada directa al método `construyeRedVectores()`. La mayoría de los métodos se explican por su nombre y parámetros. El método `inicializaRed(int lugares, int transiciones)` lo que hace es inicializar los valores de las matrices y vectores con todos sus componentes a 0. Los métodos en los que intervienen conflictos se explicarán más adelante, ya que son derivados de los métodos de la clase `Conflicto`.

Actualmente se permite la implementación de una RdP de dos maneras. Para explicar esto lo mejor es acudir a un ejemplo. Sea la RdP:

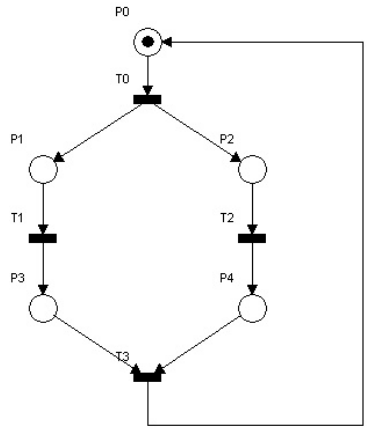


Figura 36: RdP ejemplo

Creación de una RdP directamente mediante código.

Para crear una RdP mediante código deberemos conocer sus lugares, transiciones y marcado inicial. En el caso de la RdP que se pone como ejemplo:

```
public class RedEjemplo extends Red {
  public RedEjemplo() {
    Vector <Estado> estados =
           new Vector <Estado> ();
    Estado lugar0 = new Estado(1);
    estados.add(lugar0);
    Estado lugar1 = new Estado(0);
    estados.add(lugar1);
    Estado lugar2 = new Estado(0);
    estados.add(lugar2);
    Estado lugar3 = new Estado(0);
    estados.add(lugar3);
    Vector < Transicion > transiciones =
           new Vector < Transicion > ();
//Transicion con más de un lugar de entrada o salida:
    Vector<Estado> vEnt = new Vector<Estado>();
    vEnt.add(lugar0);
    Vector<Estado> vSal = new Vector<Estado>();
    vSal.add(lugar1);
    vSal.add(lugar2);
    Transicion trans0 =
         new Transicion(vEnt, vSal);
    transiciones.add(trans0);
//Transicion simple: 1 lugar de entrada y salida:
    Transicion trans1 =
         new Transicion(lugar1, lugar3);
    transiciones.add(trans2);
    Transicion trans2 =
         new Transicion(lugar2, lugar4);
    transiciones.add(trans2);
    Transicion trans3 =
         new Transicion(lugar1, lugar13);
    transiciones.add(trans3);
    vEnt = new Vector<Estado>();
    vEnt.add(lugar3);
    vEnt.add(lugar4);
    vSal = new Vector<Estado>();
```

```
    vSal.add(lugar0);
    Transicion trans4 =
         new Transicion(vEnt, vSal);
    transiciones.add(trans0);

    construyeRedVectores(estados,transiciones);
}
```

Con este código se ha creado de una manera fácil y rápida una RdP. Lo primero es hacer que nuestra red descienda de la superclase `Red` para poder aprovechar sus métodos. Luego se definen los lugares junto con su marcado inicial. Después, y una vez creado el vector de Estados se crea de igual forma el vector de transiciones, teniendo en cuenta que a la hora de construir una `Transicion` se deben de especificar sus lugares de entrada y salida. Por último se llama al método `construyeRedVectores` que es el que se encarga de actualizar los campos del objeto `Red`. La implementación es complicada ya que debe actualizarse a partir de los estados y las transiciones el valor de las matrices de incidencia previa y posterior, el vector de marcado inicial (que al ser construida coincidirá con el vector de marcado) y el vector de conflictos.

Creación de una RdP mediante lectura de un fichero de texto.

El método anteriormente presentado presenta el importante inconveniente de que para cada modificación de la RdP se deba cambiar el código, recompilar, etc... Esto podría solucionarse mediante el uso de un editor gráfico que permitiera exportar una RdP a un formato que pudiera ser aceptable por la aplicación.

Las RdP de este proyecto se han desarrollado a partir de un programa libre de edición y análisis de RdP: el HPSim. Este programa dentro de sus opciones tiene la de exportar una RdP en modo texto. Por ejemplo, la red del ejemplo produce un archivo de texto con el siguiente contenido:

```
// Transition Name Vector:
(T0 ;T1 ;T2 ;T3 ;)
// Position Name Vector:
(P0;P1;P2;P3;P4;)
// Inzidenz Matrix:
{
( 1  0  0 -1 )
(-1  1  0  0 )
(-1  0  1  0 )
( 0 -1  0  1 )
( 0  0 -1  1 )
}
// Marking Vector:
(1 0 0 0 0 )
// Arc Type Matrix:
// Code:0 = None; 1 = Normal; 2 = Inhibitor; 3 = Test
{
(1 0 0 1 )
(1 1 0 0 )
(1 0 1 0 )
(0 1 0 1 )
(0 0 1 1 )
}
// Transition Time Model Vektor:
```

```
// Code:1 = Immidiate; 2= Delay;3 = Exponential; 4 = Equal
Distibution;
(1 ;1 ;1 ;1 ;
```

La idea es la depuración de este fichero de texto para extraer de él los atributos de la RdP necesarios: la matriz de incidencia y el vector de marcado. Como puede observarse el HPSim también proporciona herramientas para realizar RdP extendidas, es decir, con varios tipos de arcos (normales, inhibidores, de test) y de transiciones (inmediatas, con espera, exponenciales). Las RdP de este proyecto van a tener todos sus arcos del tipo 0 (normales) y todas sus transiciones del tipo 1 (inmediatas). Por tanto no necesitamos esta información. También hay que señalar que el HPSim no permite la construcción de RdP no puras, aunque la implementación si que lo permita. Esto se observa en que la RdP se exporta con una sólo matriz de incidencia, lo que quiere decir que la red es pura (si no lo fuera necesitaríamos dos matrices, previa y posterior).

La clase `TextoARed` proporciona un interfaz gráfico para seleccionar el archivo de texto creado por el HPSim y convertir la información en un objeto de la clase `Red`. La actual implementación de la misma recorre el archivo para sacar la matriz de incidencia y el vector de marcado. Una vez se tienen estos parámetros el método `construyeRedMatrizIncidencia()` crea la RdP transformando la matriz de incidencia en las dos matrices de incidencia previa y posterior para posteriormente crear un conjunto de estados y transiciones a partir de los datos contenidos en las mismas y del marcado. Estos estados y transiciones reciben su nombre del número asignado, por lo que la búsqueda de estados y/o transiciones concretas dentro de los vectores de estados y transiciones puede realizarse a partir de su nombre además de por su posición dentro del vector, como se verá al tratar la clase `Coordinador`.

Conflicto

Para ayudar al coordinador a gestionar la evolución de la RdP se hace necesaria la implementación de la clase `Conflicto`, que es simplemente un encapsulamiento adecuado para un vector de transiciones: las transiciones que estén en conflicto entre sí. La definición rigurosa de lo que es un conflicto se encuentra en la sección de definiciones.

```
public class Conflicto {
  Vector<Transicion> transEnConflicto;
  ...
}
```

Una misma transición puede pertenecer a más de un conflicto. Los métodos que implementa esta clase son:

```
  public Conflicto(Vector<Transicion> trans) {
  public Transicion getTransicionConflictiva(int index) {
  public int tamaño() {return transEnConflicto.size();}
```

Aquí convienen explicar el procedimiento que se usa para inicializar el vector de conflictos de la clase `Red`. Este es inicializado mediante el método `inicializaConflictos()`, que debe ser llamado cuando todos los demás componentes de la RdP están correctamente inicializados, y nunca antes.

Como se ha visto en la parte teórica, un conflicto se produce cuando dos o más transiciones tienen un mismo lugar de entrada. Por tanto basta con recorrer la matriz de incidencia previa para comprobar si un estado tiene dos o más transiciones de salida. Si esto ocurre entonces se crea un nuevo objeto `Conflicto` a partir de todas las transiciones de salida de dicho lugar.

Volviendo a los métodos de la clase `Red` que hemos dejado momentáneamente de lado se ve que están pensados para ser llamados uno detrás de otro por el coordinador de la RdP. Primero se comprueba si una determinada transición está dentro de algún conflicto mediante el método `esTransicionConflictiva(Transicion t)` e inmediatamente después podemos llamar al método `Vector <Conflicto> getConflictosConEstaTransicion(Transicion t)` para tener disponibles todos los conflictos en los que interviene dicha transición y actuar de acuerdo a la metodología específica que adopte el coordinador. En definitiva, la RdP es la que se encarga de determinar sus posibles conflictos, pero será el coordinador el que decida como enfrentarse a ellos.

Coordinador

La clase `Coordinador` es la clase principal dentro de la estructura de la aplicación: es la que se encarga de gestionar la evolución de la RdP mediante el disparo de las transiciones. Sus campos son:

```
public abstract class Coordinador {
  public Red red;
  public Vector < Transicion > transicionesHabilitadas;
  protected Random r = new Random();
  private ReentrantLock monitor = new ReentrantLock();
```

Obviamente el coordinador lo va a ser de una RdP concreta, por lo que debe tener acceso a modificar los campos de la misma. La clase `Coordinador` se define como abstracta porque no tiene sentido un coordinador que no implemente los métodos para averiguar si una condición se cumple para una transición habilitada y que sea capaz de ejecutar un código asociado al establecimiento, mantenimiento y finalización de un estado. El vector de transiciones habilitadas va a contener todas las transiciones que estén habilitadas dentro de la red en un momento dado. Una transición habilitada es aquella que está en disposición de ser disparada según el número de marcas de sus estados de entrada; por tanto este vector contendrá todas las transiciones que cumplan esto independientemente de si la condición asociada a cada transición se verifica o no. Obviamente el coordinador deberá asegurar que dicha condición se verifica. La variable `random` proporciona números aleatorios que nos serán de utilidad a la hora de escoger una transición a disparar.

La clase `ReentrantLock` proporciona la funcionalidad de la sincronización entre hilos de ejecución. Se puede observar que aunque luego vayan a usarse monitores para gestionar la comunicación con la célula la clase `Coordinador` sólo hace referencia a la parte de evolución de la RdP.

La verdadera complejidad de esta clase está en sus métodos. Estos son:

```
  public Coordinador(Red r) {
  protected native void inicializaComunicacion();
```

```
public int getRandomTrans() {
public void setTransicionesHabilitadas() {
public Transicion getTransicionADisparar() {
public void disparaTransicion(Transicion t) throws
    TransicionNoHabilitadaException, TokensFueraDeRangoException {
public Transicion setHabilitacion(Transicion t) {
```

Obviamente el constructor del coordinador necesita de una RdP a la que coordinar y de la cual sacar el vector de transiciones habilitadas. Puede observarse que ninguno de los métodos hace referencia a objetos específicos de la célula, por lo que puede decirse que esta clase es independiente del sistema que se quiera controlar.

El método `getTransicionADisparar()` devuelve la transición del vector de transiciones habilitadas con una mayor prioridad o en caso de que hubiera varias transiciones con la misma prioridad devuelve una de ellas de manera aleatoria. El método `setTransicionesHabilitadas()` recorre el vector de transiciones de la RdP y va comprobando una por una si están habilitadas, actualizando el vector de transiciones habilitadas. El método `disparaTransicion(Transición t)` comprueba que la transición t pertenece a la RdP, realiza una comprobación de si la transición sigue habilitada y sigue cumpliéndose su condición de disparo y realiza el proceso del disparo: quitar marcas de los estados de entrada de t, poner marcas a los estados de salida de t y actualizar los vectores implicados. Hay que resaltar el hecho de que no se implementa ningún tipo de algoritmo para decidir cual va a ser la tarea periódica del coordinador; es decir, en la superclase `Coordinador` se dan métodos para que las subclases `CoordinadorXxxx` puedan implementar un comportamiento concreto frente a un determinado sistema, pero no ofrece una solución específica por sí mismo.

Como ejemplo vamos a observar la implementación de un Coordinador concreto: el que controla el modo automático de la estación 3 de la célula de fabricación:

```
public class CoordinadorEst3
    extends Coordinador {

  Timer timerGlobal, timerEmergencia;
  public volatile long counter = 0;
  Date inicio, fin;
  Vector <Estado> estadosMarcados, estadosAntes, estadosDespues;
  MonitorEst3 monitor;
  MonitorExtra monitorExtra;
  public boolean finalizado = false;
  public boolean empezado = false;
  private static int delayGlobal = 10, delayEmergencia = 5;
```

En un primer momento se pensó que las subclases de `Coordinador`, aparte de descender de la clase `Coordinador` como es lógico, también deberían implementar el interfaz `Runnable`. El interfaz `Runnable` es la forma que tiene Java de hacer que una clase tenga la funcionalidad de la clase `Thread` sin descender directamente de ella, ya que Java no soporta la herencia múltiple. Una clase que implemente el interfaz `Runnable` debe implementar obligatoriamente el método `run()`. Con todo esto lo que se pretende es que la ejecución del coordinador se realice en un hilo de ejecución aparte del hilo principal. Independientemente de esto el coordinador puede tener varios hilos de ejecución lanzados por

él mismo, como se verá mas adelante. Esto hizo que directamente se implementarán los métodos para lanzar y parar estos procesos del coordinador sin crear un `Thread` directamente.

Obviamente si vamos a controlar la estación 3 necesitaremos acceso a las salidas y entradas de la misma a través de un objeto de la clase `Estacion3`. Para ello necesitaremos crear previamente una instancia de la clase `Estacion3` y un `Monitor` que contenga a dicha estación. Dicho monitor tiene existencia independiente a la del coordinador, por lo que se le deberá pasar a la hora de construirlo. Las variables `timerGlobal` y `timerEmergencia` son instancias de la clase `java.util.`. Cada una de ellas representa un hilo de ejecución independiente con una función específica.

En un principio se implementaron estos timers como instancias de la clase `javax.swing.Timer`. Sin embargo una lectura detallada del API de la clase reveló, ya con una versión definitiva de la `AplicacionCelula`, que internamente todos los timers de esta clase se ejecutan en un solo hilo de ejecución, que además es el que se encarga también de los eventos gráficos; esto hacía que el diseño concurrente no fuera necesario. Por tanto para demostrar que la aplicación podía funcionar con múltiples hilos de ejecución se cambió todos los timers de swing por timers de `java.util`, que si que utilizan un hilo para cada tarea (`TimerTask`) asignada. Es un orgullo decir que el paso de uno a otro se realizó sin problemas, lo que demuestra que el análisis de la problemática de la concurrencia dentro del proyecto se realizó de forma acertada.

Los vectores de estados `estadosMarcados`, `estadosAntes` y `estadosDespues` son vectores que ayudan en el proceso de disparo de las transiciones.

El control del modo automático de la célula es un sistema relativamente sencillo, pero como se verá cuando se trate el coordinador global de toda la célula su complejidad puede aumentar extraordinariamente. Esta es otra de las razones por las que la estructura con un supercoordinador abstracto y subclases de él es la más adecuada. Bastará con añadir la funcionalidad necesaria a la clase `CoordinadorCelula`.

Los métodos que implementa `CoordinadorEstacion3` son:

```
public CoordinadorEstacion3(Red r, MonitorExtra mon, MonitorEst3 m3)
public void run()
public void relanzaTimers()
public void paraTimers()
public void finCoordinador()
public void finalizar()
public void devuelvePaletActualizado(Palet paletAActualizar)
public boolean condicionDisparo(Transicion t)
public void accionPrevia(Estado e)
public void accionPosterior(Estado e)
public void accionContinua(Estado e)
```

El constructor del coordinador tiene como parámetros la RdP a controlar, que ha sido creada previamente por la aplicación principal, y el monitor o monitores necesarios donde volcar o leer información. Dentro del monitor de la clase `MonitorEst3` se encuentra una instancia de la clase `Estacion3`, mientras que

el acceso al identificador de productos se hace a través del monitor global de la clase `MonitorExtra`. El coordinador no tiene por que saber si la RdP que se le pasa está correctamente construida; él solo actuará de acuerdo con ella.

Los métodos `run(), relanzaTimers() y paraTimers()` corresponden, respectivamente, a la inicialización, pausa y restablecimiento de los timers asociados al coordinador. La estructura es la misma que la utilizada de forma mucho más esquemática a la hora de crear los temporizadores para los estados. El método `finCoordinador()` encapsula el final de la ejecución del coordinador: para los timers, resetea las salidas y variables de memoria y devuelve el marcado de la red a su valor inicial para una posible futura reinicialización. El método `finalizar()` simplemente muestra por pantalla el valor de distintas variables de monitorización de la ejecución del coordinador y puede prescindirse de su uso en posteriores versiones. Estos métodos resultan imprescindibles para controlar el proceso dentro de la aplicación principal; no sería coherente que mientras utilizáramos el modo manual se siguieran ejecutando los coordinadores del modo automático.

Dentro del constructor se realiza la declaración de los timers. Los dos tienen una estructura idéntica, aunque obviamente la complejidad de `timerGlobal`, que representa la tarea de coordinación propiamente dicha, es mucho mayor que la de `timerEmergencia`, que controla que la seta de emergencia de la estación no esté pulsada. La estructura de la clase `Timer` necesita de un `TimerTask` que tenga el código a ejecutar por la tarea. Este es el código para crear un timer:

```
class tareaLaQueSea extends TimerTask {
  public void run() {
    //hacer cosas aquí...
  }
};
```

A pesar de que parece una llamada muy complicada como lo que realmente hay que implementar es el cuerpo del método no debemos preocuparnos por ella. Una vez creada la tarea sólo queda asignarla al `Timer`:

```
int delayElQueSea = 70;
timerEjemplo = new Timer();
timerEjemplo.schedule(tareaLaQueSea, 0, delayElQueSea);
```

Este código viene a decir que la tarea se ejecutará cada 70 milisegundos..

En una primera aproximación al problema se implementó una lectura directa desde el Coordinador mediante un timer especifico para la tarea. Después, al aumentar la complejidad de la aplicación mediante el uso de múltiples monitores y coordinadores se decidió crear una clase auxiliar que fuera la encargada exclusiva de leer y escribir el bus, y que actualizara los valores de los monitores. Así en vez de tener varios coordinadores leyendo y escribiendo del bus se tiene un solo agente lector / escritor y los coordinadores acceden a los datos que necesitan a través de monitores. También a lo largo del proyecto se modificó la estructura del programa para que la escritura del bus no fuera directa: en las primeras versiones un cambio en la variable Java (instancia de la clase `VariableBooleana` contenido en una

instancia de la clase `EstacionX`, descendiente de `Estacion`) se transmitía hasta el bus inmediatamente, al igual que una petición de lectura de una de estas mismas instancias implicaba una lectura completa del bus, aunque fuera para una sólo variable. En la versión final es la clase auxiliar la que se encarga de cada cierto tiempo leer y escribir el bus, mientras que los coordinadores y demás clases sólo modifican las variables Java y las de la librería dinámica. Si bien esto para la lectura puede considerarse beneficioso mi opinión personal es que, dado que las operaciones de escritura (activación / desactivación de salidas del bus) no van a ser frecuentes, el hecho de dejar los valores en una especie de estado "latente" hasta que se escriba el bus, y dado que la concurrencia estaba garantizada, es una pérdida de eficacia. Los detalles de esta clase pueden examinarse en la sección de código en la clase `GestorBus`.

Los últimos métodos son los que se encargan de decir si una transición cumple su condición de disparo y el código a ejecutar durante las diferentes etapas de un estado. En particular, si queremos que la RdP evolucione, el método El método `condicionDisparo(Transicion t)` debe ser implementado ya que una transición necesita para ser disparada que esté habilitada y que se cumpla su condición de disparo. El método `condicionDisparo(Transicion t)` devuelve si se cumple dicha condición, que puede ser de muy variada índole: una entrada o combinación de ellas, una temporización de un estado, etc... Los métodos `accionXxxx(Estado e)` no necesitan ser implementados para la evolución de la RdP, pero obviamente se necesitará que el sistema a controlar cambie sus salidas de acuerdo con la RdP para obtener un control práctico. Un estado, por ejemplo, puede tener solamente una acción asignada cuando pasa de inactivo a activo, y no en las demás situaciones.

```
public boolean condicionDisparo(Transicion t) {
  String nom = t.getNombre();
  if (nom.equals("T9")) {
    return true;
  }
  else if (nom.equals("T10")) {
    return (red.getEstado("P9").getTiempoMarcado() > 40);
  }

  else if (nom.equals("T11")) {
    return (!monitor.est3.Cinta_adelante.getValor());
  }
  ...
}
public void accionPrevia(Estado e) {
  String nom = e.getNombre();
  if (nom.equals("P2")) {
    monitor.est3.Culata.setValor(true);
    return;
  }
  else if (nom.equals("P3")) {
    monitor.est3.Pinza_sube_baja.setValor(true);
    monitor.est3.Culata.setValor(false);
    return;
  }
  ...
}
```

Los métodos `accionContinua()` y `accionPosterior()` tienen una estructura idéntica a la de `accionPrevia()`. En un principio se realizó la implementación de estos métodos según el índice dentro del vector de la red. Sin embargo esto sólo resulta práctico en el caso de que la RdP se hubiera creado de forma manual con código, ya que no hay forma de saber en que orden coloca el HPSim los lugares y transiciones. Por tanto se hizo necesario cambiar la estructura de `switch/case` dependiente del índice dentro del vector por una de `if/else/if` dependiente del nombre. Esto facilita además las posibles modificaciones que se le puedan hacer al sistema de control. Si hacemos desaparecer un estado o transición y añadimos uno o varios a la red sólo tendremos que cambiar la cláusula `if/else/if` previamente creada; si no tendríamos que saber exactamente en que lugar del vector se ha quitado y en cual o cuales se han añadido los nuevos, aumentando las probabilidades de equivocarnos y provocar una evolución incorrecta de la red, lo que conlleva un comportamiento indeterminado del sistema.

Una vez explicados todos los componentes podemos explicar la tarea global del coordinador. Una primera aproximación al problema es:

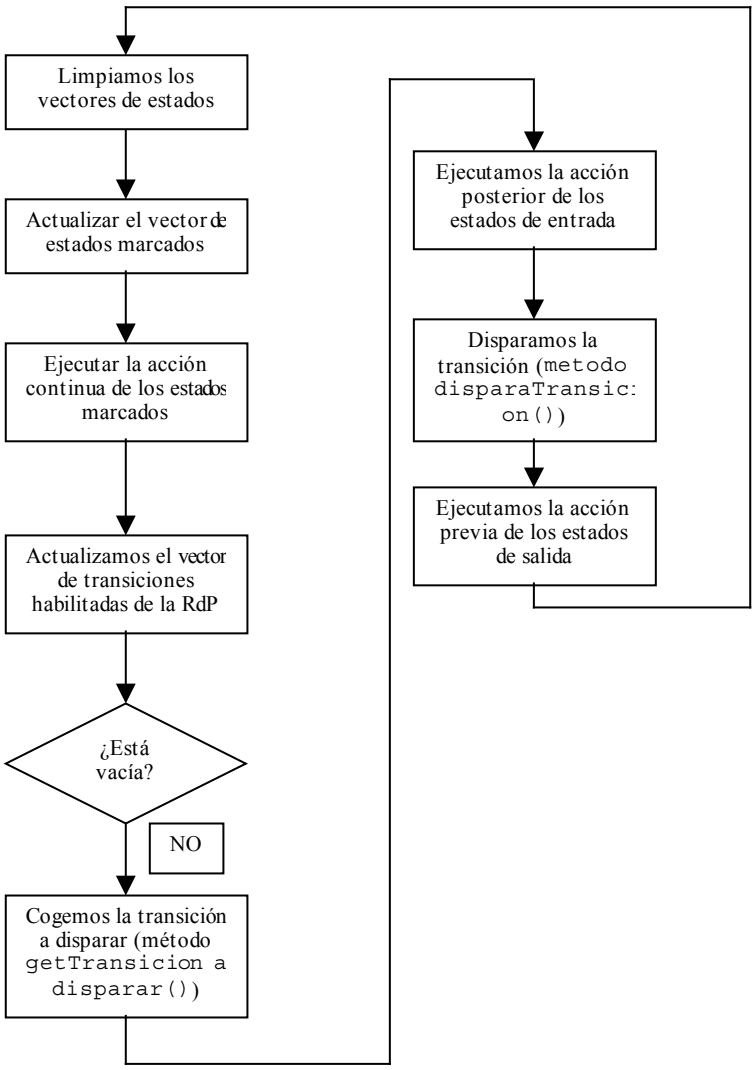


Figura 37: Algoritmo coordinador 1

Esta implementación implica una serie de ventajas y desventajas:

- Al disparar una sola transición en cada ciclo del coordinador no hace falta tener en cuenta posibles conflictos; el método `getTransicionADisparar()` de la clase `Coordinador` nos devuelve la transición a disparar teniendo en cuenta la prioridad y la RdP se actualiza (los vectores de transiciones y de estados y el marcado) dentro del método `disparaTransicion()`.

- Sin embargo, debido precisamente a esto puede darse la situación de que dentro de una serie de transiciones habilitadas siempre haya una con mayor prioridad que las demás pero que no se cumpla su condición de disparo; la red quedaría pues bloqueada.

- También puede darse el caso de que habiendo un gran número de transiciones habilitadas todas ellas con prioridades similares una transición habilitada puede quedar sin dispararse durante un largo periodo (indeterminado) de tiempo, debido a que en cada ciclo del coordinador tendrá la misma probabilidad de ser disparada que el resto de transiciones del vector de transiciones habilitadas, teniendo pues una rama de la RdP sin evolucionar durante mucho tiempo.

- Estas dos últimas desventajas no son de gran importancia a la hora de implementar el control de la célula de fabricación, dado que el número de transiciones habilitadas será normalmente limitado (casi siempre cuatro, una por estación) y todas las transiciones tienen la misma prioridad (exceptuando las de salida de palets de las estaciones, que siempre cumplirán la condición de salida). Sin embardo para sistemas más complicados pueden resultar desastrosas.

Por tanto este algoritmo está sujeto a mejoras y modificaciones que ayudándose de los métodos implementados en la clase Coordinador pueden resultar más eficientes. La versión final implementada consiste en que en cada ciclo del coordinador en vez de dispararse una sola transición se dispararan todas las transiciones del vector de transiciones habilitadas que cumplan su condición de disparo. Esto conduce a un control más estricto de los posibles conflictos efectivos que se puedan dar en la RdP. En la siguiente figura puede observarse el algoritmo utilizado:

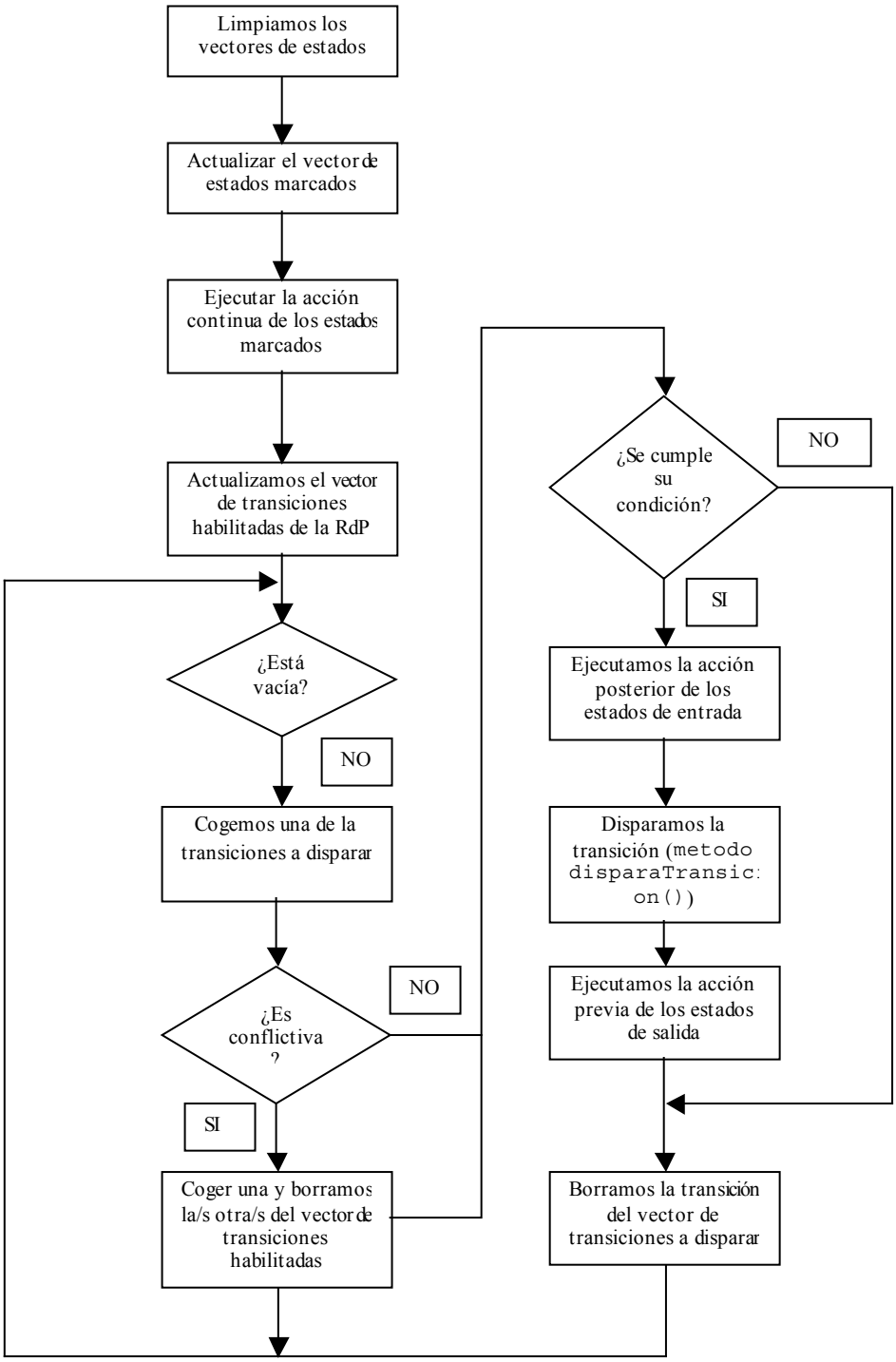


Figura 38: Algoritmo coordinador 2

Este algoritmo resuelve los problemas del anterior, pero también posee su propia problemática. Ahora no se puede decir nada a priori del tiempo que puede costar la ejecución de un ciclo del coordinador porque dependerá del número de transiciones habilitadas y de los posibles conflictos entre ellas. Además introduce la problemática del tratamiento efectivo de los conflictos en los que se cumple más de una condición de disparo, es decir, además de los ya implementados conflictos estructurales se debe detectar conflictos efectivos y tratarlos convenientemente.

5. Código fuente

En esta sección se va a hacer un repaso de las clases implementadas y utilizadas durante este proyecto. La organización básica que se ha seguido ha sido la siguiente:

- Paquete comun: clases básicas de control que van a ser necesarias en la mayor parte de aplicaciones del proyecto.
- Paquete estacion1: clase `Estacion1` y pequeñas aplicaciones de control manual y automático de la misma, incluyendo el coordinador específico.
- Paquete estacion2: clase `MonitorEst2` y `CoordinadorEst2`.
- Paquete estacion3: clase `Estacion3` y pequeñas aplicaciones de control manual y automático de la misma, incluyendo el coordinador específico.
- Paquete estacion4: clase `Estacion1` y pequeñas aplicaciones de control manual y automático de la misma, incluyendo el coordinador específico.
- Paquete estacion6: clase `Estacion6`. Dado que posteriormente se eliminó el modulo Interbus de esta estación para colocarlo en el transporte no se realizó ninguna clase más en este paquete.
- Paquete transporte: clase `Transporte` y pequeñas aplicaciones de control manual y automático del transporte de la célula, incluyendo el coordinador específico.
- Paquete puertoSerie: contiene las clases que gestionan la comunicación con el identificador de productos desde el puerto serie del PC.
- Paquete paneles: clases de contenido gráfico de uso común para las aplicaciones del proyecto.
- Código C de la librería dinámica.

Paquete comun

El paquete comun contiene las clases básicas del proyecto. Las clases relativas a las redes de Petri se han explicado ya en la sección correspondiente, por lo que aquí sólo se describen las clases relativas a la gestión de las comunicaciones que no se han explicado hasta ahora.

Estacion.java

Esta clase sólo implementa el interfaz para las clases `EstacionX` que se verán posteriormente (lo cual hace pensar que quizás debería haber sido implementada como interfaz en vez de cómo clase abstracta). Define los siguientes métodos:

```
public abstract class Estacion {
```

```
  public abstract void InicializaEstacion();
  public abstract void FinalizaEstacion();
  public abstract void leeEntradas();
  public abstract void escribeSalidas();
}
```

GestorBus.java

Esta clase encapsula la inicialización, finalización, lectura y escritura del bus. Sus métodos son los que corresponden a estas funciones, encapsulados a su vez en la clase `LeeYEscribeBus`, que no es más que un `Timer` de la clase `java.util.Timer` que se encarga de leer y escribir los valores contenidos en la librería dinámica en o del bus.

```
  public native void InicializaCelula();
  public native void FinalizaCelula ();
```

Dentro del timer se ejecutarán cada cierto tiempo las tareas de lectura y escritura del bus:

```
  public native void leeEntradas();
  public native void escribeSalidas();
```

Estos métodos leen el valor de las variables contenidas en la librería dinámica y lo vuelcan en el bus (en caso de escritura) o efectúan la actualización de dichas variables a partir de los valores de los sensores del bus.

MarcoTest.java y MarcoTest_AcercaDe.java

Clases creadas por JBuilder para implementar la aplicación `comun.Test`. En un principio dada la gran cantidad de entornos gráficos de usuario que se iban a implementar se utilizó el JBuilder de Borland para crear las pantallas necesarias. Sin embargo pronto se vio que el código generado era absolutamente ilegible cuando las aplicaciones comenzaban a tener una magnitud media - alta. Por ello las aplicaciones creadas en una fase temprana del proyecto suelen tener una clase `MarcoXxxx` creada por JBuilder mientras que las más recientes están creadas manualmente y su código está mucho mejor estructurado, si bien la carga de trabajo extra generado ha sido considerable.

En concreto estas dos clases están creadas para ser usadas junto con la aplicación `Test.java`.

Pedido.java

La clase `comun.Pedido` implementa un pedido de una pieza de la célula. Es una clase sencilla que sin embargo posee una gran importancia dado que va a ser de ella de la que se compongan los vectores de pedidos que deben gestionarse. Los campos que posee son consecuentes con lo explicado:

```
  Palet palet;
  int orden;
  int estado;
  GregorianCalendar fechaCreacion;
```

y los métodos que implementa son los de establecimiento y lectura (`set` y `get`) de los mismos, exceptuando el de la fecha de creación que sólo posee un método `get`, ya que el establecimiento se realiza automáticamente cuando se crea el pedido. La clase `Pedido` en sí misma no tiene ningún control sobre el valor del orden; este debe ser controlado externamente antes de llamar al constructor de la clase.

PruebaTiempoPalets.java

Para una explicación de esta clase ver el apartado **aplicaciones**.

Test.java

Para una explicación de esta clase ver el apartado **aplicaciones**.

TextoARed.java

Para una explicación de la aplicación generada por esta clase ver el apartado **aplicaciones**. La clase TextoARed gestiona el paso de un archivo de texto generado por el HPSim a una red correctamente creada utilizable por los coordinadores. Su método principal es

```
  public Red construyeRedDeArchivo(File archivoEntrada) {
```

que es el que realiza la conversión. Para la implementación de la clase se utilizó la clase `StringTokenizer` que implementa la división de un archivo de texto (más concretamente un flujo, tal y como Java usa los archivos) en partes más pequeñas o tokens. El código resultante es complicado de leer y engorroso de mantener si se produce un cambio en el HPSim; sin embargo todas las pruebas realizadas dan fe de que actualmente la clase funciona perfectamente.

Variable.java

La clase `Variable` es una de las clases vitales para el desarrollo del proyecto. Es la clase abstracta de la que dependerán todas las subclases que se usarán para implementar las variables (entradas y salidas) de las distintas estaciones. Sus campos son los siguientes:

```
  public int tipo;
  public int tamaño;
  public int orden;
  public int estacion;
  public String nombre;
```

además de los siguientes métodos:

```
  public Variable(int tipo) {
  public Variable(int tip, int tam)
  public Variable(int tip, int tam, String nom, int orden, int
estacion)
  public synchronized void setTipo(int tip) {
  public synchronized int getTipo()
  public synchronized void setTamaño(int tamaño)
  public synchronized int getTamaño() {
```

```
  public synchronized void setOrden(int ord) {
  public synchronized int getOrden() {
  public synchronized void setEstacion(int est) {
  public synchronized int getEstacion() {
  public synchronized String getNombre() {
  public synchronized void setNombre(String n) {
  public abstract void actualizaVariable();
```

Las variables `orden` y `estacion` deben ser creadas de acuerdo con la variable que vayan a controlar, mientras que `tipo`, `tamaño` y `nombre` son indicativas.

Todos los métodos se declaran `synchronized` para evitar posibles problemas derivados del acceso simultaneo de varios coordinadores a una variable. El método `actualizaVariable` debe ser implementado por cada subclase y consistirá en la lectura de la variable de la estación si es de entrada o en el establecimiento de la salida.

VariableAnalogica.java

La clase `VariableAnalogica` implementa la funcionalidad de una variable de tipo analógico, es decir, con un amplio rango de valores.

```
public class VariableAnalogica extends Variable {
  public double valor;
```

Como puede verse la clase presenta un campo llamado `valor` que es el que almacena el valor analógico a leer. Los métodos que implementa son:

```
  public VariableAnalogica(int tipo, int tamaño, String nom, int est,
int ord) {
  public void actualizaVariable() {
  public void setValor(double v) {
  public double getValor() {
  private native int leeEntrada(int estacion, int orden);
  public double leeEntradaAnalogica(int estacion, int orden) {
  public native void escribeSalida(double valor, int estacion, int
orden);
```

El método `actualizaVariable` debe ser implementado por todas las subclases de `Variable`. Los métodos `leeEntrada` y `escribeSalida` son declarados nativos, es decir, que su implementación está realizada en C. El método leeEntradaAnalogica surge de la necesidad de filtrado de la señal. En efecto, la señal leída necesita de un filtrado para adquirir un valor coherente. El filtrado se ha realizado mediante la siguiente ecuación:

$$VF = \alpha\, VF + (1-\alpha)\, VSF$$

donde VF = Valor filtrado y VSF = Valor Sin Filtrar.

El código usa un α de 0.995, es decir, un filtrado muy fuerte. Dado que la lectura del valor se va a realizar según el delay del `TimerTask` del Coordinador encargado de la misma, el tiempo de respuesta para que la variable tenga un valor correcto es:

$$Tr = (1 / delay) * \alpha$$

Es decir, que para un delay de 50 ms. el tiempo de respuesta aproximado de la variable es de...

VariableBooleana.java

Todas las entradas y salidas de la célula tienen un valor de todo o nada, es decir, booleano. La clase `VariableBoolena` implementa la funcionalidad de este tipo de variables, tanto de entrada como de salida.

```
public class VariableBooleana extends Variable {
  public boolean valor;
```

Los métodos implementados por esta clase son:

```
  public VariableBooleana(int tipo, String nom, int est, int ord)
  public void setValor(int val)
  public void setValor(boolean val)
  public boolean getValor()
  public native boolean leeEntrada(int estacion, int orden);
  public native void escribeSalida(boolean salida, int estacion, int
orden);
  public void actualizaVariable()
```

Al igual que pasaba con `VariableAnalogica` esta clase posee dos métodos nativos: `escribeSalida` y `leeEntrada`, que son implementados en C para aprovechar las librerías del HLI del fabricante de la tarjeta controladora de bus. Los métodos se explican por sí solos.

El método correcto para actualizar el valor de una variable de salida es el método `setValor()`, mientras que para las de entrada conviene usar `actualizaVariable()`; lo único que hacen ambas es encapsular llamadas a los métodos nativos dependiendo del tipo de variable.

VariableMemoria.java

Esta clase encapsula un valor booleano para ser usado por el coordinador para almacenar determinados valores o estados necesarios para el correcto funcionamiento de la célula. Sus campos son:

```
  boolean valor;
  String nombre;
```

Mientras que sus únicos métodos son los constructores y los métodos `set` y `get` de las variables.

Paquete estacion1

El paquete estacion1 contiene las clases relacionadas con la estación 1 de la célula

Estacion1.java

La clase estacion1 contiene la información de registro de las variables de la estación 1. Para más información ver la parte de descripción física de la célula de fabricación.

En primer lugar se coloca el número que corresponde a la estación. En este caso el número que ocupa la estación 1 es el número 2:

```
public class Estacion1 {
  private static final int NUMERO_ESTACION = 2;
```

Una vez hecho esto se inicializan las variables de entrada y salida de la siguiente forma:

```
  //Definición de variables de entrada.
  public static VariableBooleana Cinta_atras = new
VariableBooleana(Variable.
      BOOLEANA_ENTRADA,
      "Cinta atras", NUMERO_ESTACION, 0);
  .....
  //Definición de variables de salida.
  public static VariableBooleana Cinta_avanza = new
VariableBooleana(Variable.
      BOOLEANA_SALIDA,
      "Cinta avanza", NUMERO_ESTACION, 0);
```

Puede verse la estructura de creación de las variables de la estación: se crea un objeto de la clase VariableBooleana, se le asigna el tipo correspondiente, un nombre y finalmente el número de la estación y un número de orden dentro de la misma. Este número de orden es **imprescindible** que coincida con el número dado a la variable en la función nativa por el CMD. En caso contrario se producirá la escritura o lectura de una variable indeseada, con imprevisibles consecuencias. Después de ser probado, puede asegurarse que salvo cambios en el cableado las variables contenidas en este código coinciden con sus homólogas de la función C.

Los métodos implementados por esta clase son:

```
  public synchronized native void InicializaEstacion();
  public synchronized native void FinalizaEstacion();
  public synchronized void leeEntradas() {
  public synchronized void escribeSalidas() {
  public void resetSalidas() {
  public synchronized native void MuestraEntradas();
```

No debe olvidarse de “conectar” esta clase con la librería compartida mediante la sentencia:

```
  static {
    System.loadLibrary("CELULA");
  }
```

Esta llamada puede hacerse desde cualquier clase de la que se cree una instancia antes de llamar a cualquier método nativo. Por ejemplo, en la versión final de `AplicacionCelula` esta llamada se hace desde la clase `GestorBus`.

Estacion1Swing.java

Para una explicación de esta clase ver el apartado **aplicaciones**.

MarcoEstacion1.java y MarcoEstacion1_AcercaDe.java

Para una explicación de estas clases ver el apartado **aplicaciones**.

MonitorEst1

Como ya se ha explicado en el apartado referente a monitores, la clase MonitorEst1 encapsula el acceso a las variables de la estación 1, ya sea mediante variables de memoria o variables de entrada o salida de la estación.

```
public class MonitorEst1
    extends Monitor {

  public Estacion1 est1;

  private VariableMemoria est1Comunicando;
  private VariableMemoria est1Funcionando;
  private VariableMemoria est1Produciendo;
  private VariableMemoria sacarPiezaAPalet;
  private VariableMemoria tirarPieza;
  private VariableMemoria retirarPiezaDePalet;
  private VariableMemoria piezaCorrectaEst1;
  private VariableMemoria camisaProducida;

  private Pedido pedidoEnEst1;
  private Palet paletEnEst1;
```

Aquí puede verse la estructura que se ha seguido para gestionar los monitores. Se componen en primer lugar de una `Estacion`, que se declara pública porque la sincronización se lleva a cabo por los métodos de la propia clase `EstacionX`. Los demás campos de la clase se declaran como privados para que sólo se pueda acceder a ellos mediante los métodos implementados en la propia clase, obviamente todos ellos declarados como `synchronized` para garantizar la exclusión mutua entre hilos intentando acceder a las variables del mismo. Los métodos de la clase Estacion1 son los de set y get de las variables, que se han implementado siguiendo la regla de que el método de lectura de la variable se llamará como la misma con la palabra `get` delante, respetando la separación de palabras mediante mayúsculas. De igual forma se procede con los métodos `set` de establecimiento.

```
  public synchronized boolean getEst1Comunicando() {
    return est1Comunicando.getValor();
  }

  public synchronized void setEst1Comunicando(boolean valor) {
    est1Comunicando.setValor(valor);
  }
```

Para garantizar la lectura y escritura de las variables de la estación, para acceder a la misma se han implementado los siguientes métodos:

```
  public synchronized void leeEntradas() {est1.leeEntradas();}
  public synchronized void escribeSalidas() {est1.escribeSalidas();}
  public synchronized void resetSalidas() {est1.resetSalidas();}
```

Paquete estacion2

CoordinadorEst2

Coordinador que controla el proceso de la estación 2, pero que depende de `CoordinadorCelula`.

MonitorEst2

El monitor de la estación 2. Contiene solamente variables de memoria dado que la estación 2 de la célula no se ha conectado a Interbus.

Paquete estacion3

El paquete estacion3 contiene las clases relacionadas con la estación 3 de la célula

AplicacionRdPE3.java

Para una explicación de esta clase ver el apartado **aplicaciones**.

CoordinadorEstacion3.java

Para una explicación de esta clase ver el apartado **Redes de Petri**.

CoordinadorEstacion3PruebaTar.java

Esta clase se creó como ejemplo para el modelado de un coordinador a partir de la lectura de una RdP desde un archivo de texto, en contraposición a la clase `CoordinadorEstacion3` que usaba la red implementada manualmente en el archivo `RedAutomaticoEstacion3.java`. Su estructura es análoga a la de `CoordinadorEstacion3` excepto en el detalle ya comentado, así que para una explicación más detallada acudir al apartado referente a **Redes de Petri**.

Estacion3.java

La clase `Estacion3` contiene la información de registro de las variables de la estación 3. Para más información ver la parte de descripción física de la célula de fabricación.

En primer lugar se coloca el número que corresponde a la estación. En este caso el número que ocupa la estación 3 es el número.

La estructura es análoga a la de la clase `Estacion1` y los métodos implementados son los mismos, por lo que no se añade nada nuevo a lo ya explicado para la clase `Estacion1`.

Estacion3Swing.java

Para una explicación de esta clase ver el apartado **aplicaciones**.

MarcoEst3.java y MarcoEst3_AcercaDe.java

Clases de entorno gráfico de usuario para la aplicación `AplicacionRdPE3`. Para una explicación de estas clases ver el apartado **aplicaciones**.

MarcoEstacion1.java y MarcoEstacion1_AcercaDe.java

Clases de entorno gráfico de usuario para la aplicación `Estacion3Swing`. Para una explicación de estas clases ver el apartado **aplicaciones**.

MonitorEst3

Similar a `MonitorEst1` pero para variables de la estación 3 de la célula.

RedAutomaticoEstacion3

Clase que implementa la creación de una RdP para el control automático de la estación. Para saber más acerca de la creación manual de RdP ver el apartado **Redes de Petri** y después el subapartado **Creación de las redes de Petri del proyecto.**

RedResetEstacion3

Clase que implementa la creación de una RdP para el control de la inicialización de la estación 3. Para saber más acerca de la creación manual de RdP ver el apartado **Redes de Petri** y después el subapartado **Creación de las redes de Petri del proyecto.**

Paquete estacion4

El paquete estacion4 contiene las clases relacionadas con la estación 4 de la célula

AplicacionRdPE3.java

Para una explicación de esta clase ver el apartado **aplicaciones**.

CoordinadorEstacion4.java

Para una explicación de esta clase ver el apartado **Redes de Petri**.

CoordinadorEstacion4PruebaTar.java

Esta clase se creó como ejemplo para el modelado de un coordinador a partir de la lectura de una RdP desde un archivo de texto, en contraposición a la clase `CoordinadorEstacion4` que usaba la red implementada manualmente en el archivo `RedAutomaticoEstacion4.java`. Su estructura es análoga a la de `CoordinadorEstacion4` excepto en el detalle ya comentado, así que para una explicación más detallada acudir al apartado referente a **Redes de Petri**.

Estacion4.java

La clase `Estacion4` contiene la información de registro de las variables de la estación 4. Para más información ver la parte de descripción física de la célula de fabricación.

En primer lugar se coloca el número que corresponde a la estación. En este caso el número que ocupa la estación 4 es el número.

La estructura es análoga a la de la clase `Estacion1` y los métodos implementados son los mismos, por lo que no se añade nada nuevo a lo ya explicado para la clase `Estacion1`.

Estacion4Swing.java

Para una explicación de esta clase ver el apartado **aplicaciones**.

MarcoEst4.java y MarcoEst4_AcercaDe.java

Clases de entorno gráfico de usuario para la aplicación `AplicacionRdPE4`. Para una explicación de estas clases ver el apartado **aplicaciones**.

MarcoEstacion1.java y MarcoEstacion1_AcercaDe.java

Clases de entorno gráfico de usuario para la aplicación `Estacion4Swing`. Para una explicación de estas clases ver el apartado **aplicaciones**.

MonitorEst4

Similar a `MonitorEst1` pero para variables de la estación 4 de la célula.

PruebaAnalogica

Para una explicación de esta clase ver el apartado **aplicaciones**.

RedAutomaticoEstacion4

Clase que implementa la creación de una RdP para el control automático de la estación. Para saber más acerca de la creación manual de RdP ver el apartado **Redes de Petri** y después el subapartado **Creación de las redes de Petri del proyecto.**

Paquete estacion6

El paquete estacion6 contiene únicamente la clase Estacion6. Posteriormente se desmontó su módulo de Interbus por lo que no se implementaron más clases en este paquete.

Estacion6.java

La clase `Estacion6` contiene la información de registro de las variables de la estación 6. Para más información ver la parte de descripción física de la célula de fabricación.

Como la estación 6 no llegó a formar parte de la red Interbus de la célula su número es el 0, ya que era la única.

La estructura es análoga a la de la clase `Estacion1` y los métodos implementados son los mismos, por lo que no se añade nada nuevo a lo ya explicado para la clase `Estacion1`.

Paquete excepciones

Este paquete contiene las excepciones implementadas para el proyecto. Dado que se ha puesto mucho más énfasis en el correcto control de la célula y en la aplicación de las RdPs el tratamiento de posibles errores se realiza a un nivel básico. Las clases contenidas en el paquete son:

```
class EstacionesException extends Exception
class TipoDatoIncorrectoException extends EstacionesException
class TokensFueraDeRangoException extends EstacionesException
class EstacionNullPointerException extends NullPointerException
class TransicionNoHabilitadaException extends EstacionesException
```

Paquete finales

El paquete finales contiene las aplicaciones finales del proyecto y constituyen el conjunto de clases definitivas, por ahora, tanto para el control manual como automático de la célula.

AplicacionCelula.java

Es la aplicación definitiva compendio de todo lo realizado en el proyecto. Para una explicación de esta clase ver el apartado **aplicaciones**.

Dentro del archivo `AplicacionCelula.java` se crearon más clases de interfaz gráfico de usuario de gran utilidad a la hora de decidir el diseño final de la pantalla. Estas son `BotonToggle` y `PanelSensor`. `BotonToggle` implementa el botón con fondo rojo usado para las variables de salida de las estaciones mientras que `PanelSensor` implementa un mini-panel formado por una etiqueta con el nombre de una variable de entrada y un dibujo, ya sea un punto rojo o verde para indicar el estado de la entrada (activado o desactivado). En un principió se crearon aquí estas clases porque se pensó que ambas podrían ser utilizadas tanto en la pantalla de modo automático como en la de manual, pero al final se han usado solamente en la de modo manual.

CoordinadorBorrar.java

Coordinador específico que gestiona el control del borrado de los palets de la célula de fabricación. Para una explicación de esta clase ver el apartado **aplicaciones**, especialmente `TestBorrar`, también el apartado **Redes de Petri**.

CoordinadorCelula.java

Coordinador específico que gestiona el control completo de la célula de fabricación. Para una explicación de esta clase ver el apartado **aplicaciones**, especialmente `TestBorrar`, también el apartado **Redes de Petri**.

FrameModoAutomatico.java

La clase `FrameModoAutomatico.java` se encarga de dibujar la pantalla de control automático de la célula. Dentro de este archivo se encuentran las pantallas gráficas que pueden observarse en el apartado dedicado a la aplicación `AplicacionCelula` dentro de **aplicaciones**. Para la creación de esta pantalla se han implementado también una serie de clases, la mayoría implementaciones de JPanel, para una mayor comodidad a la hora de implementar la interfaz gráfica. Estas clases son:

```
class PanelNuevosPedidos
class PanelEstadoEstacion
class PanelGestionPedidos
class ModeloTablaPedido
class PanelMuestraPedidos
```

Cada una de ellas implementa una parte de la pantalla, excepto `ModeloTablaPedido` que es un modelo que sirve para que la clase `JTable` dibuje la tabla central donde se ve el desarrollo de los pedidos. Para mayor comodidad a la hora de crear nuevas aplicaciones se han sacado estas clases de aquí y se han ordenado todas en el paquete `paneles`.

FrameModoManual

Exactamente como la anterior, sólo que dedicada a la pantalla de explotación manual. Para más información debe verse el apartado de **aplicaciones** y más concretamente `AplicacionCelula`. Las clases complementarias implementadas dentro del archivo FrameModoManual.java son:

```
class PanelEst1
class PanelEst2
class PanelEst3
class PanelEst4
class PanelTransporte
```

Cada una de ellas crea y controla manualmente una de las estaciones. Como en el de modo automático, estas clases también se han colocado dentro del paquete `paneles`.

TestBorrar

Para una explicación de esta clase ver el apartado **aplicaciones**.

Paquete paneles

Contiene clases de contenido gráfico usadas para la creación de las aplicaciones. Pueden servir como base para futuras ampliaciones o mejoras del aspecto visual de la aplicación

BotonToggle

Esta clase desciende directamente de la clase `JToggleButton`, añadiéndole la funcionalidad de cambiar el fondo de gris a rojo cuando el botón está activado. Se ha empleado esta clase para crear los botones de salidas del control manual de las estaciones y para los botones de pausa y emergencia del modo automático de control.

Figura 39 BotonToggle activado y desactivado

PanelEst1, PanelEst3, PanelEst4 y PanelTransporte

Cada una de estas clases representan la pantalla de control manual de cada estación y del transporte, de tal forma que encapsula en una sola pantalla las entradas y salidas de la estación, siendo las entradas instancias de `BotonToggle` y las salidas de `PanelSensor`. Obviamente estas clases van a tener un timer de actualización de las variables, y al igual que con los coordinadores, a la hora de construirlas se les deberá pasar como parámetro los monitores respectivos. Al ser una aplicación gráfica aquí los timers si que van a ser instancias de `javax.swing.Timer` y las acciones relativas a la actualización de la imagen se llevarán a cabo dentro del thread de eventos del AWT. Como ejemplo muestro el resultado de la creación de un `PanelTransporte`:

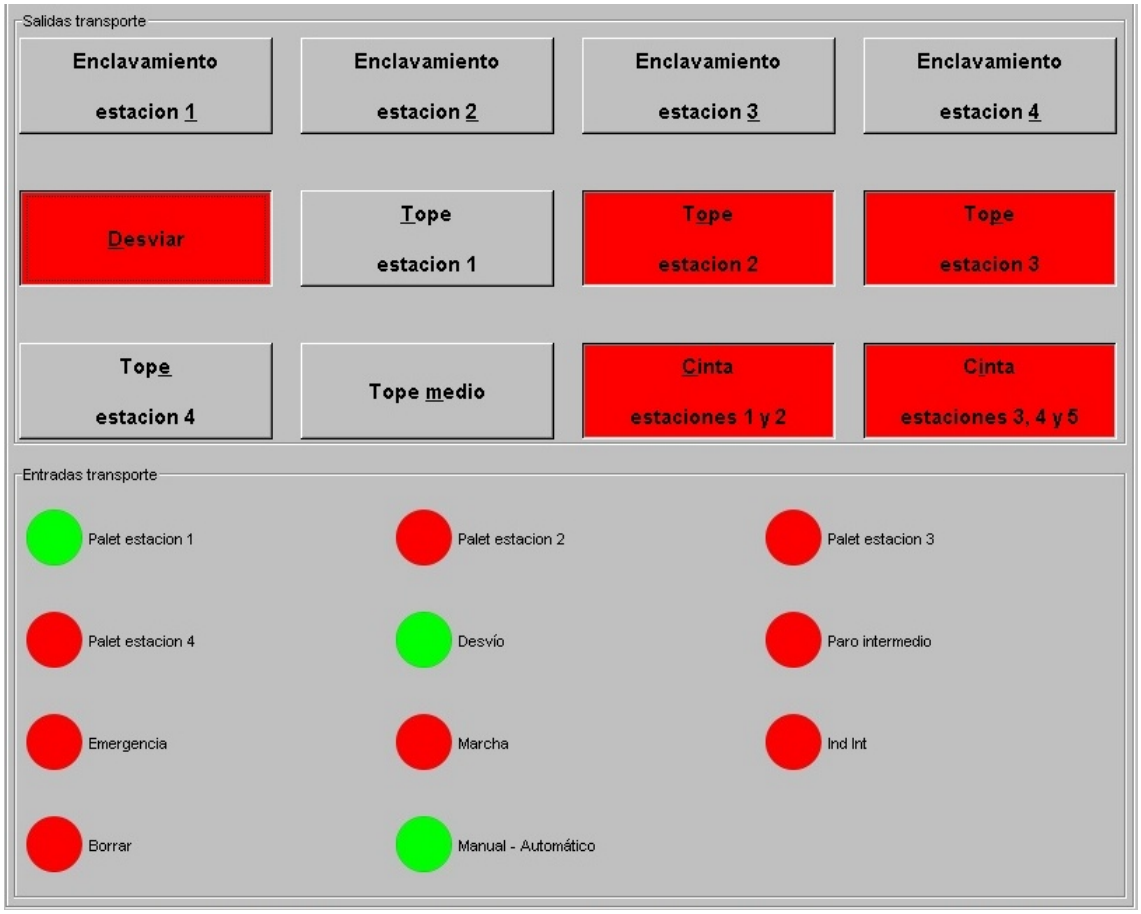


Figura 40 PanelTransporte

PanelEstadoEstacion

Este panel dibuja el estado de las estaciones dentro del modo automático de fabricación. Por tanto también se le deberán pasar los monitores apropiados e implementar timers de actualización.

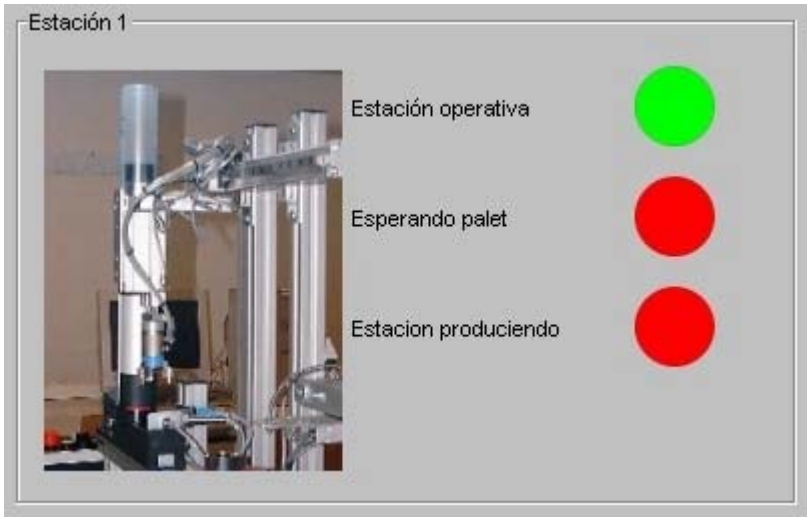


Figura 41 PanelEstadoEstacion

PanelGestionEstadoEstaciones

Encapsula dentro de un mismo panel cuatro instancias de la clase PanelEstadoEstacion relativas a las cuatro estaciones de la célula:

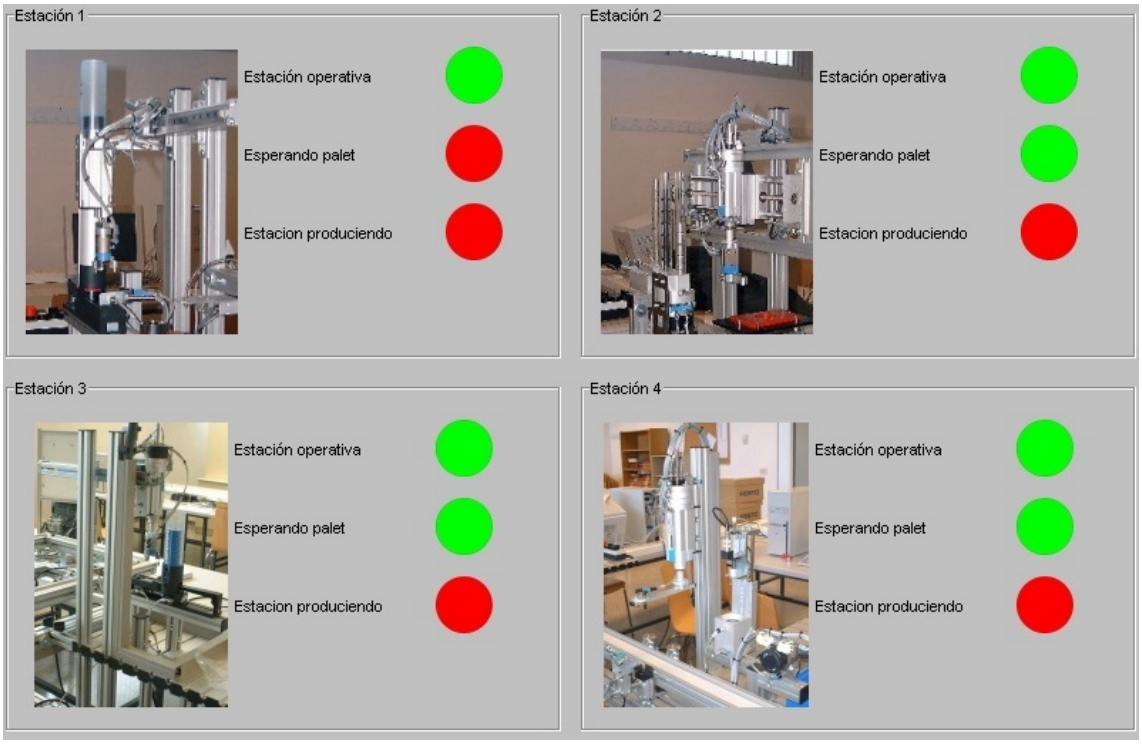


Figura 42 PanelGestionEstadoEstaciones

PanelMuestraPedidos

Esta clase muestra una tabla con el contenido del vector de pedidos e información acerca de los mismos. Para ello se ha creado una clase auxiliar, la clase `ModeloTablaPedido` que desciende de `JTableModel`, lo que es imprescindible para dibujar una tabla en Swing. Como esta tabla va a acceder al vector de pedidos se le deberá pasar el monitor que lo contenga. La implementación de esta clase se ha hecho a partir de una muy similar proveniente del tutorial de Java sobre edición de tablas.

Número de pedido	Tipo de pedido	Estado del pedido	Última modificación
0	NEGRA CON TAPA	Finalizado	Mon Feb 14 10:28:08 C...
1	METALICA CON TAPA	Finalizado	Mon Feb 14 10:28:41 C...
2	METALICA CON TAPA	Finalizado	Mon Feb 14 10:29:15 C...
3	METALICA CON TAPA	Estacion 4	Mon Feb 14 10:29:48 C...
4	METALICA CON TAPA	Estacion 1	Mon Feb 14 10:26:38 C...
8	METALICA CON TAPA	Sin comenzar	Mon Feb 14 10:27:06 C...
9	NEGRA CON TAPA	Sin comenzar	Mon Feb 14 10:27:09 C...
10	ROJA CON TAPA	Sin comenzar	Mon Feb 14 10:27:12 C...

Figura 43 Tabla de pedidos

PanelNuevosPedidos

Este panel permite lanzar nuevos pedidos, incluyéndolos en el vector de pedidos. Por tanto también necesitará, al igual que todos, que se le pase el monitor o monitores correspondientes.

Figura 44 Panel de nuevos pedidos

PanelSensor

Este panel implementa una imagen, que puede ser un punto verde o rojo, para indicar si una entrada de la célula está activa o no dentro del modo manual, aunque podría usarse en cualquier modo de funcionamiento.

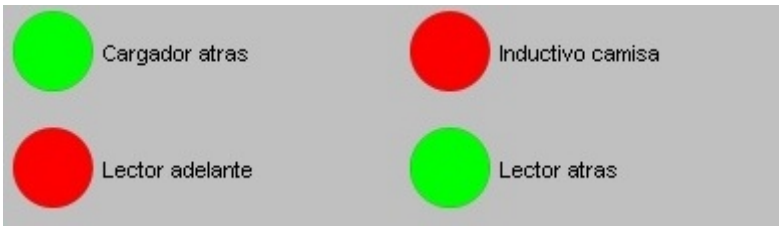


Figura 45 Cuatro instancias de PanelSensor

Paquete puertoSerie

Este paquete contiene las clases que gestionan la comunicación entre la aplicación y el identificador de productos. Aunque estas clases se han pensado para comunicar con el identificador de productos a través del puerto serie una mejora inmediata de la aplicación es gestionar el identificador como un dispositivo PCP a través de Interbus. A la hora de redactar esta memoria el identificador se ha cableado como el último módulo de la estación 1.

Comunicaciones

Esta clase se ha creado para guardar una serie de instancias de Trama que van a ser usadas continuamente. Por eso se definen como estáticas y finales, lo que en otros lenguajes de programación equivale a constantes. Las tramas implementadas son:

```
  public static Trama leer1, leer2, leer3, leer4, borrar1, borrar2,
borrar3, borrar4;
  public static Trama salir, restart, test;
```

Como a la hora de escribir una trama tendremos que incluir datos acerca del palet a escribir no podemos crear aquí una trama que sea `escribirX`.

Palet

Clase que modela los datos que debe contener los datos de un palet de la célula de fabricación, tales como pieza, color, tapa, etc... Está clase también provee métodos necesarios para pasar dicha información a una trama enviable al identificador de productos y viceversa. Por tanto sus campos serán los necesarios para la identificación del palet:

```
public class Palet {
  public static final byte SINDEFINIR = 0x00;
  public static final byte NEGRA = 0x10;
  public static final byte NEGRACONTAPA = 0x20;
  public static final byte ROJA = 0x30;
  public static final byte ROJACONTAPA = 0x40;
  public static final byte METALICA = 0x50;
  public static final byte METALICACONTAPA = 0x60;

  GregorianCalendar tiempo;
  int tamañoEnBytes;

  int segundo;
  int minuto;
  int hora;
  int dia;
  int mes;
  int año;
  byte tipoPieza;
  boolean camisa;
  boolean embolo;
  boolean muelle;
  boolean culata;
  boolean piezaConTapa;
  boolean piezaEnPalet;
```

Los métodos que implementa son los de lectura y establecimiento de todos los campos mediante llamadas a `getNombreDeLaVariable` o `setNombreDeLaVariable`.

Además de estos, la clase `Palet` también implementa los siguientes métodos:

```
public byte[] pasaAscii(int digito)
public boolean equals(Palet p)
public byte[] toBytes()
public int getEnteroDeAscii(byte [] b)
public int aDatos(Trama t)
public void printPalet()
```

Todos estos métodos han sido implementados pensando en la necesidad de que los datos del palet tienen que ser pasados mediante una cadena de bytes (`Trama`) y también realizar el proceso inverso de leer de una `Trama` un `Palet`.

Trama

La clase Trama encapsula la cadena de bytes en protocolo RS232 que hay que mandar al identificador de productos vía puerto serie. Para explicar esta clase conviene conocer algunas nociones acerca del funcionamiento del identificador de productos, por lo que es necesario haber entendido la sección sobre el identificador de productos en la parte de hardware.

Esta clase en realidad es muy sencilla, ya que únicamente contiene la cadena de bytes que se va a mandar al identificador y un flag que indica si esta terminada o no. La verdadera dificultad reside en la complejidad de creación de las tramas

6. Aplicaciones

Aunque la aplicación principal del proyecto sea `finales.AplicacionCelula` para su realización ha sido necesario implementar una serie de pequeñas aplicaciones, ya sea como prototipo para una aplicación más compleja o para probar funcionalidades problemáticas de la célula, tales como el identificador de productos o la variable analógica de la estación 4. Muchas de ellas poseen un cierto interés por lo que se va a presentar un breve resumen de ellas a continuación.

AplicacionCelula

La clase `AplicacionCelula` es la clase principal del proyecto. Para ejecutarla se usa la aplicación `java` o `javaw`. En ella se han recogido toda la experiencia y progresos anteriores. Nada más ejecutarla aparece la siguiente pantalla principal:

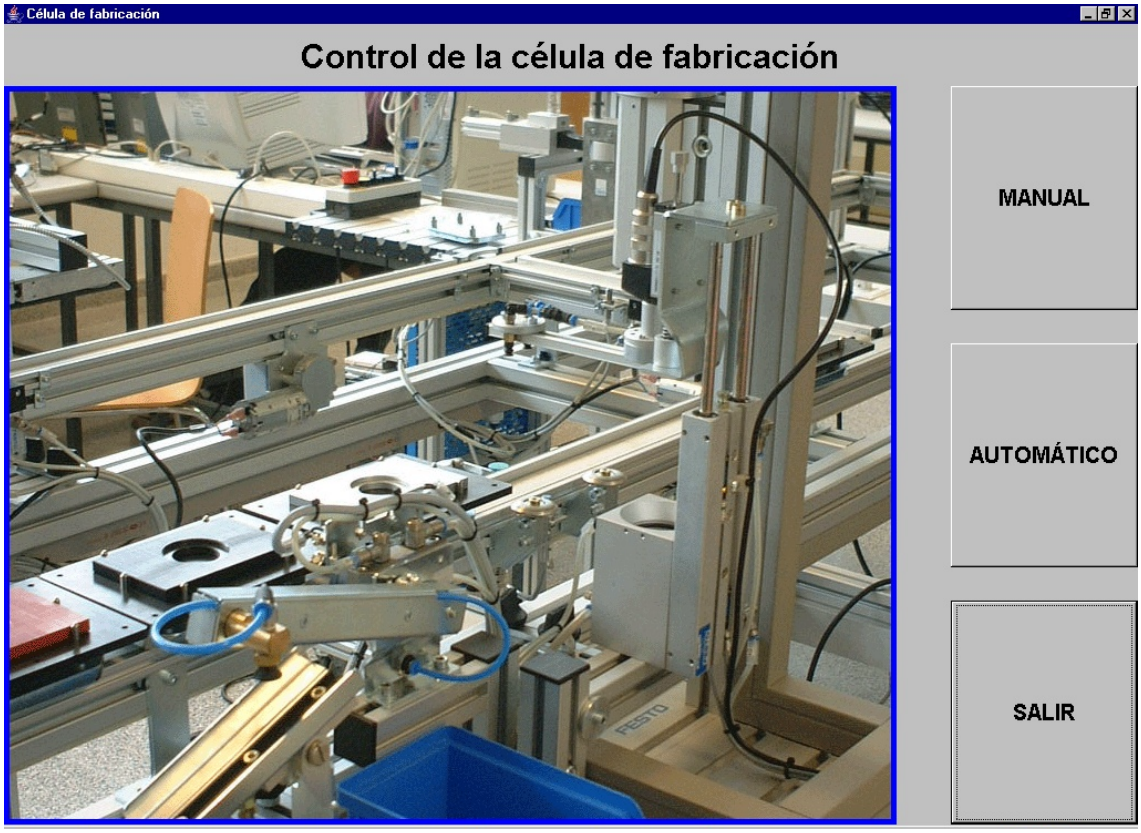


Figura 46: Pantalla principal aplicación principal

Obviamente vamos a poder controlar la célula de dos maneras: producción automática y producción manual. Según la manera escogida pulsaremos uno u otro menú. Dado que los dos tipos de control no pueden darse simultáneamente al pulsar un botón éste queda desactivado para posteriores pulsaciones a no ser que se cierre la ventana que aparezca.

En el modo manual de control de la célula podremos controlar todas las salidas de las estaciones y el transporte a la vez que se tiene un panel indicador del estado de las entradas que va cambiando en tiempo

real. Esto se consiguió leyendo periódicamente el estado de las entradas de la célula al igual que se hizo a la hora de implementar el coordinador.

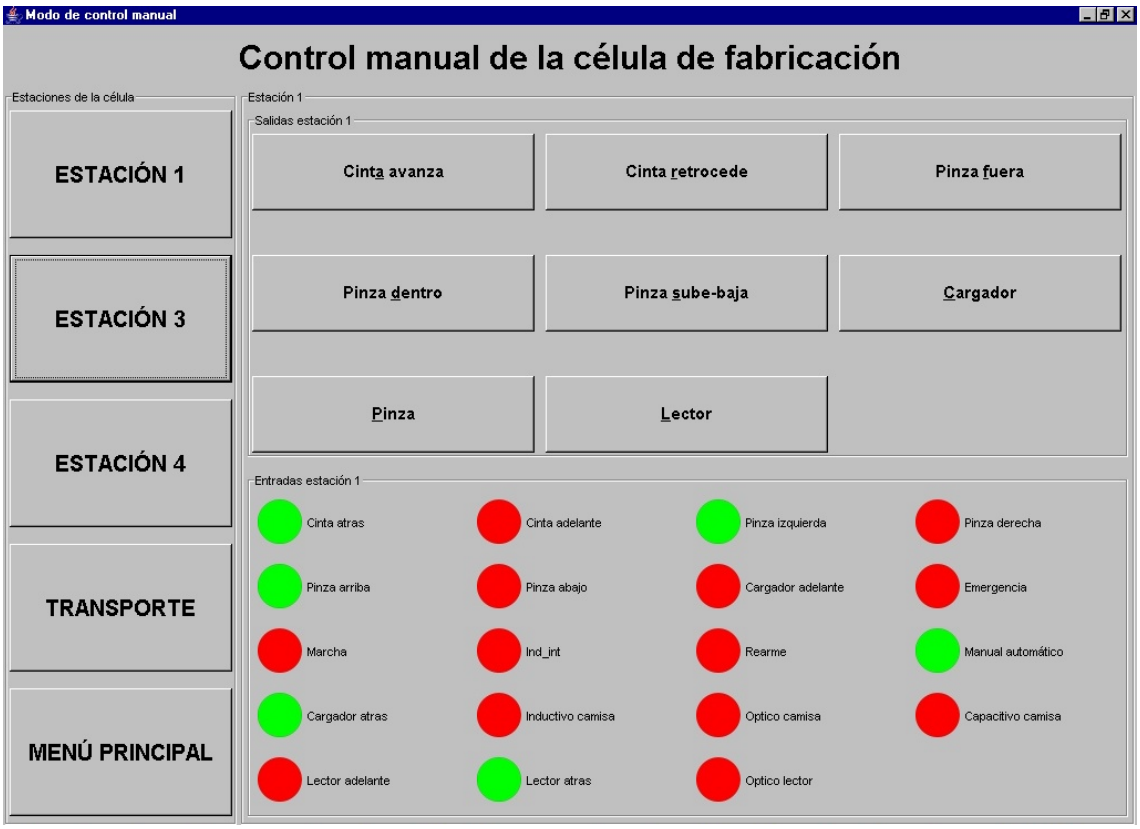


Figura 47: Pantalla control manual

En la figura puede verse concretamente la pantalla de la estación 1, que es la que aparece por defecto. Si se desea controlar las otras estaciones sólo hay que navegar mediante los botones de la parte izquierda. La pantalla se divide en dos mitades: la parte superior muestra las salidas de la estación, mediante un botón de dos estados. La parte de abajo muestra las entradas de la estación actualizadas automáticamente (en realidad mediante un `Timer` de 20 ms.). en la siguiente figura puede verse el panel del transporte estando activadas las salidas de las dos cintas transportadoras, el desvío intermedio y los topes 2 y 3:

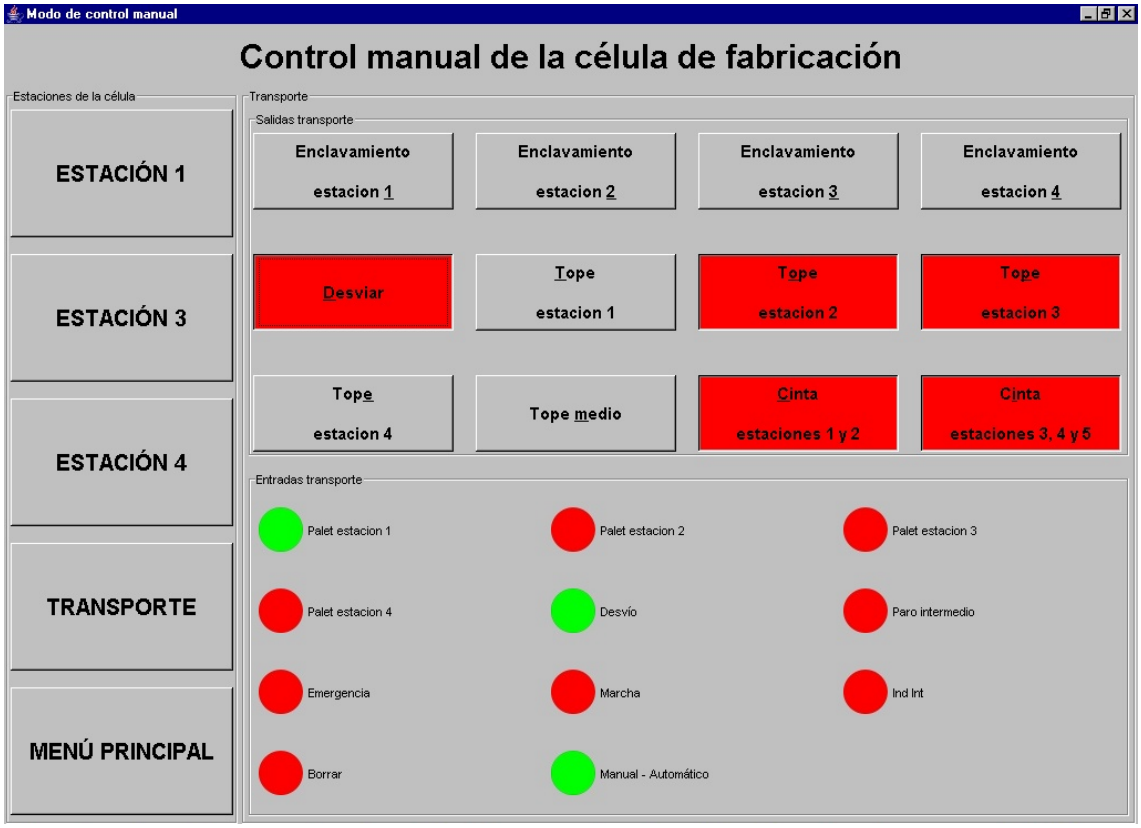


Figura 48: Pantalla control manual 2

Si quisiéramos pasar del modo automático al modo manual o viceversa podrían producirse resultados indeseados. Por ello una pantalla informa al usuario de esta posibilidad permitiendo cancelar su selección:

Figura 49: Cambio de automático a manual

El modo automático es el modo en el cual se reúnen todos los progresos realizados en el proyecto: control de las estaciones mediante la estructura de una red de Petri. La pantalla principal del modo automático es la siguiente:

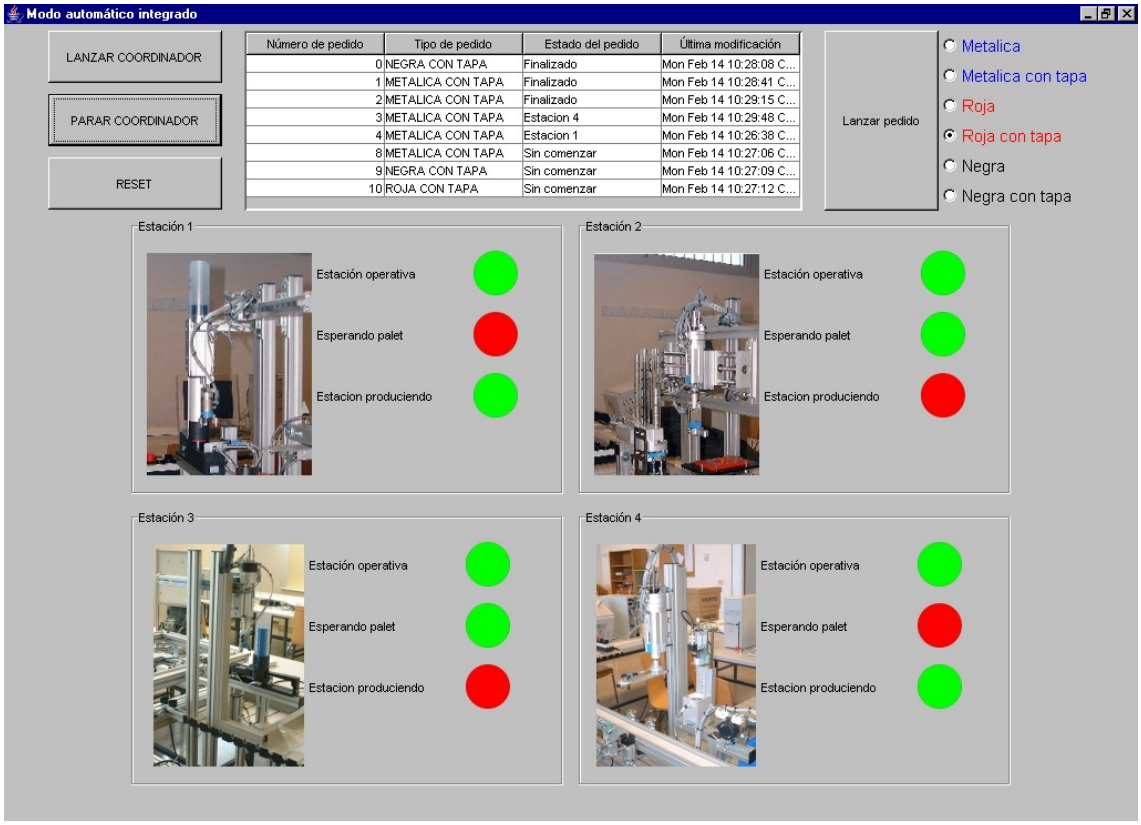


Figura 50: Pantalla modo automático

En la imagen puede observarse la composición de la pantalla. En primer lugar arriba a la izquierda están los tres botones que controlan el coordinador de la célula: lanzar, parar y reset. Los dos primeros lanzan al coordinador o lo paran si está ejecutándose respectivamente. La combinación de estos botones no anula las salidas existentes de la célula, es decir, podemos parar el coordinador y relanzarlo después sin que el proceso de control se vea afectado. El tercer botón provoca la anulación de las salidas de la estación, el establecimiento de la RdP a su valor inicial y el borrado del vector de pedidos. Obviamente esto provocará que la aplicación comience a ejecutar el proceso de borrado de palets, ya que no tiene sentido hacer un reset cuando hay palets con piezas por en medio. Una vez realizado el borrado la aplicación esperará a que el usuario de la orden de lanzar coordinador mediante el primer botón.

La parte derecha de la aplicación corresponde al panel de lanzamiento de pedidos. En él el usuario introduce el tipo de pieza que desea fabricar, pidiéndosele después una confirmación para lanzar un nuevo pedido. Todos los pedidos lanzados pasan pues a formar parte de un vector de pedidos que el coordinador gestionará hasta su finalización. La parte central de la pantalla la ocupa la tabla con la información acerca de los pedidos. En ella puede seguirse el proceso que lleva cada pedido desde que es lanzado hasta que finaliza la producción en la estación 4.

Como guarda ante posibles fallos provocados por la manipulación indebida de los palets se ha implementado el coordinador de la célula de modo que cuando un palet llega a una estación lo primero que se hace es comprobar si ese palet es el que *debería* haber llegado a la estación. En caso de que no lo sea el coordinador automáticamente creará un pedido nuevo basándose en la información leída del palet.

En caso de que el palet indeseado esté vacío la tabla no mostrará información sobre él, aunque internamente permanecerá en el vector de pedidos.

El código del `CoordinadorCelula` incluye un número de variables de memoria que se encargan de almacenar el valor de determinados parámetros del proceso de fabricación. Estas variables de memoria son instancias de la clase `VariableMemoria` y tienen la misma funcionalidad que los objetos de tipo %Mxx en un lenguaje de programación de autómatas. La mayoría de estas variables son condiciones de disparo de alguna transición de la RdP. La red final de control de la célula se encuentra en el archivo RedFinal.hps y su traducción a texto en RedFinal.txt.

Estas variables de memoria se han colocado en los monitores correspondientes, y el coordinador correspondiente es el que modifica su valor. Las distintas clases gráficas leen de estos monitores para actualizar sus valores.

PruebaTiempoPalets

Esta pequeña aplicación surgió de la necesidad de saber el tiempo exacto que tarda cada palet en salir y entrar de las estaciones. En efecto, si dos palets están pegados uno al otro el tiempo entre que el tope de la estación al subir no golpea al palet saliente y el palet siguiente aún puede ser parado al levantarse el palet es muy reducido. Si levantamos el tope antes de tiempo, golpeamos o incluso evitamos que el primer palet salga. Si tardamos demasiado, el segundo palet aún no habrá podido alcanzar la posición en que se realiza el correcto enclavamiento de la misma. Esta segunda opción es aun más peligrosa que la primera ya que implica que la lectura y escritura del palet ya no van a ser correctas.

Por tanto se creo esta pequeña clase que mide el tiempo exacto que transcurre entre que se baja el tope de la estación y el segundo palet activa de nuevo la entrada que detecta si hay un palet en la estación. No se consideró siquiera necesario la creación de un entorno gráfico. A continuación se muestra el resultado de uno de los ensayos:

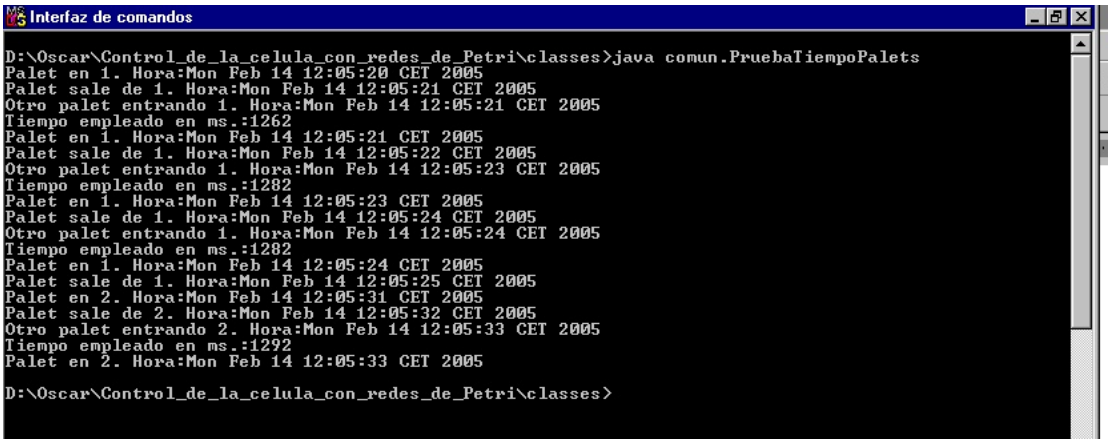


Figura 51: Prueba de tiempos de palets

Se comprobó que el tiempo aproximado entre estos dos sucesos es de 1’3 segundos. Este dato se empleó a la hora de realizar la RdP de salida de palets de las estaciones.

Test

La clase Test fue la primera prueba sobre el uso conjunto simultáneo de dos coordinadores sobre tres redes distintas. La implementación de esta aplicación se realizó directamente juntando las aplicaciones de control automático de las estaciones 3 y 4.

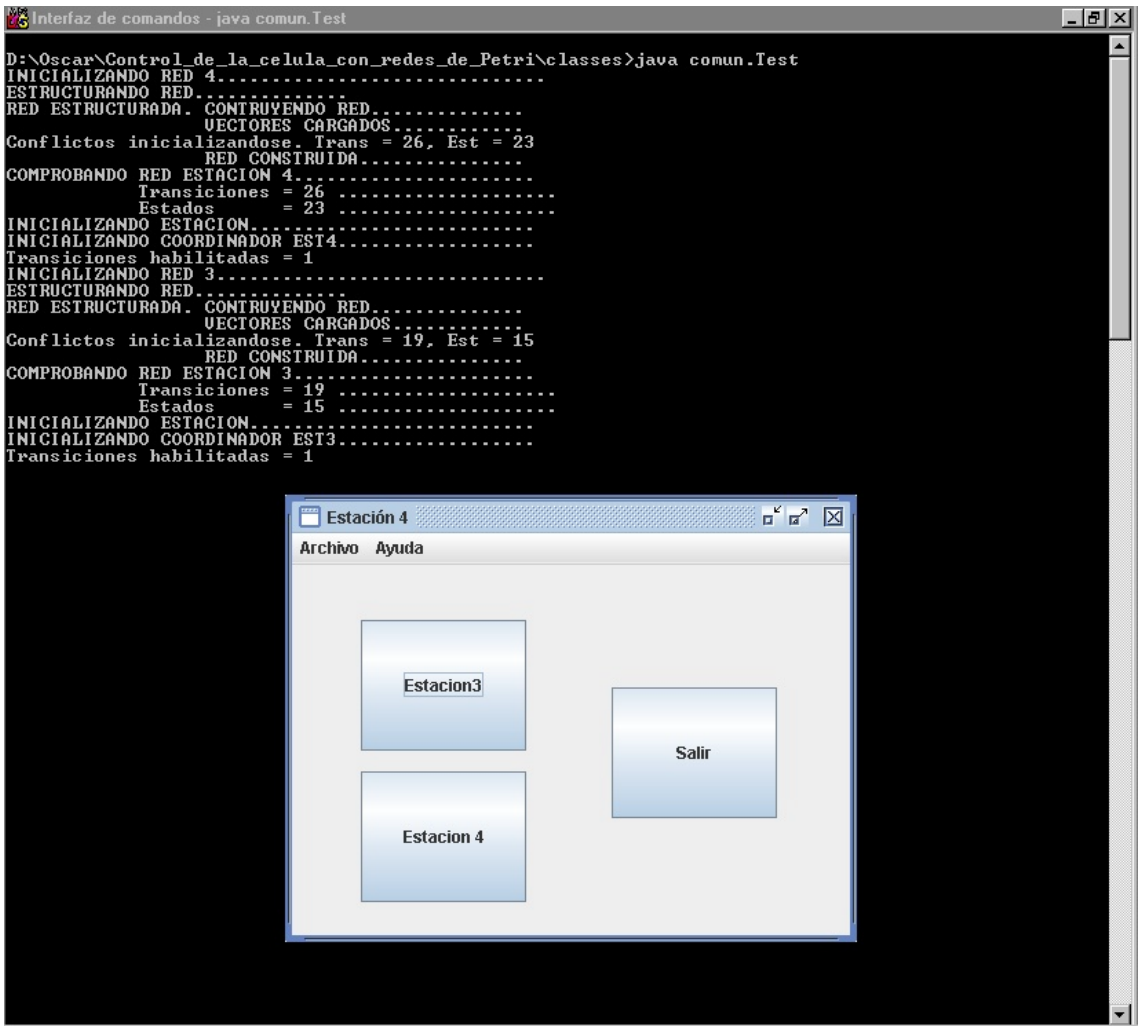


Figura 52: Test estaciones 3 y 4

Según puede verse en la pantalla anterior, la mayoría de las aplicaciones del proyecto generan una serie de salidas en la ventana de comandos que permiten comprobar la correcta ejecución de la aplicación; en este caso puede verse el proceso de creación de las redes de Petri y del lanzamiento del coordinador. Se pudo comprobar que los coordinadores se provocaron ninguna interferencia entre sí.

TextoARed

La aplicación `TextoARed` incluida dentro de la propia clase se usó para comprobar la corrección del proceso de conversión de un archivo de texto generado por el HPSim a un objeto correctamente creado de la clase `Red` utilizable por la aplicación de control a través del coordinador. Consiste básicamente en un selector de ficheros que permite escoger un archivo que contenga una red y un panel de texto que permite cotejar con la red si el proceso de conversión se ha realizado de forma satisfactoria.

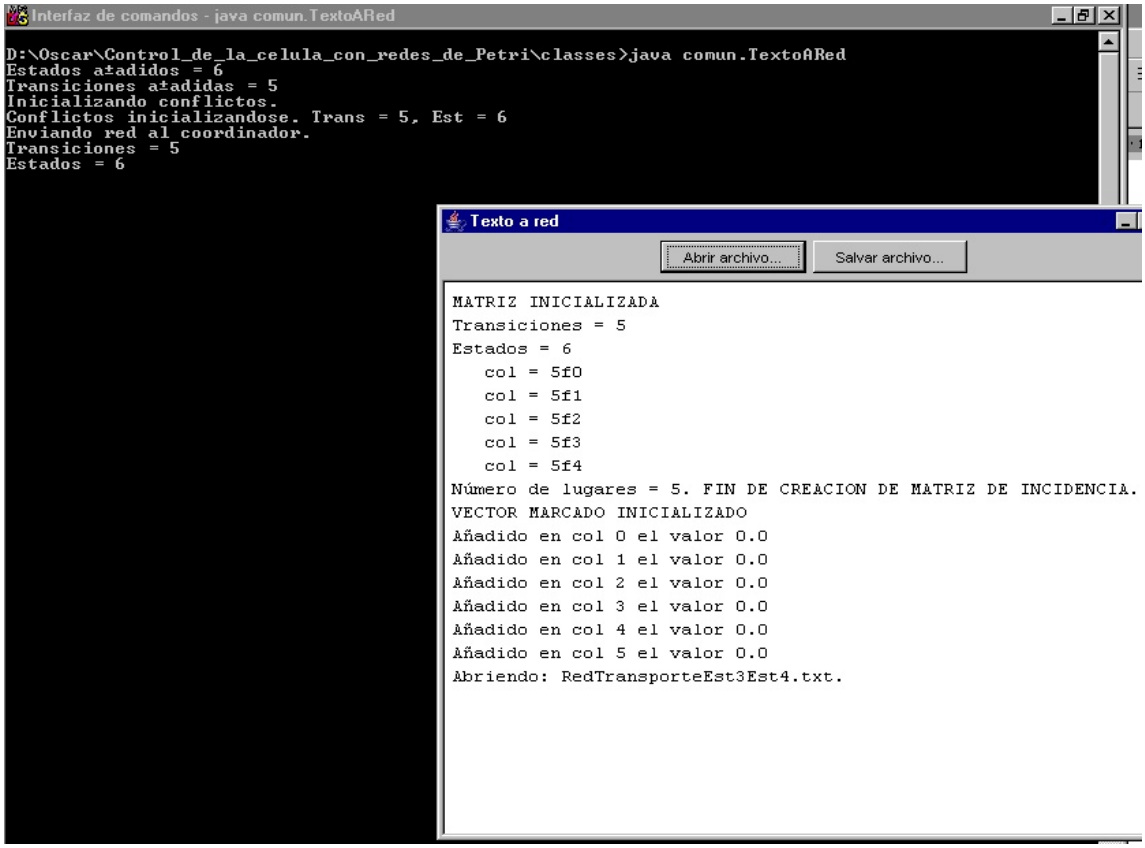


Figura 53: TextoARed

Estacion1Swing

Esta aplicación se creo en una fase temprana del proyecto, por lo que también está controlada por dos clases creadas por el JBuilder: `MarcoEstacion1` y `MarcoEstacion1_AcercaDe`. Su función fue la de comprobar la relación entre las variables Java (instancias de la clase VariableBooleana) y sus correspondientes variables C.

Gracias a ella se pudo comprobar fallos provenientes de una numeración incorrecta de la estación 1 ya fuera en la parte C del código como en la parte Java.

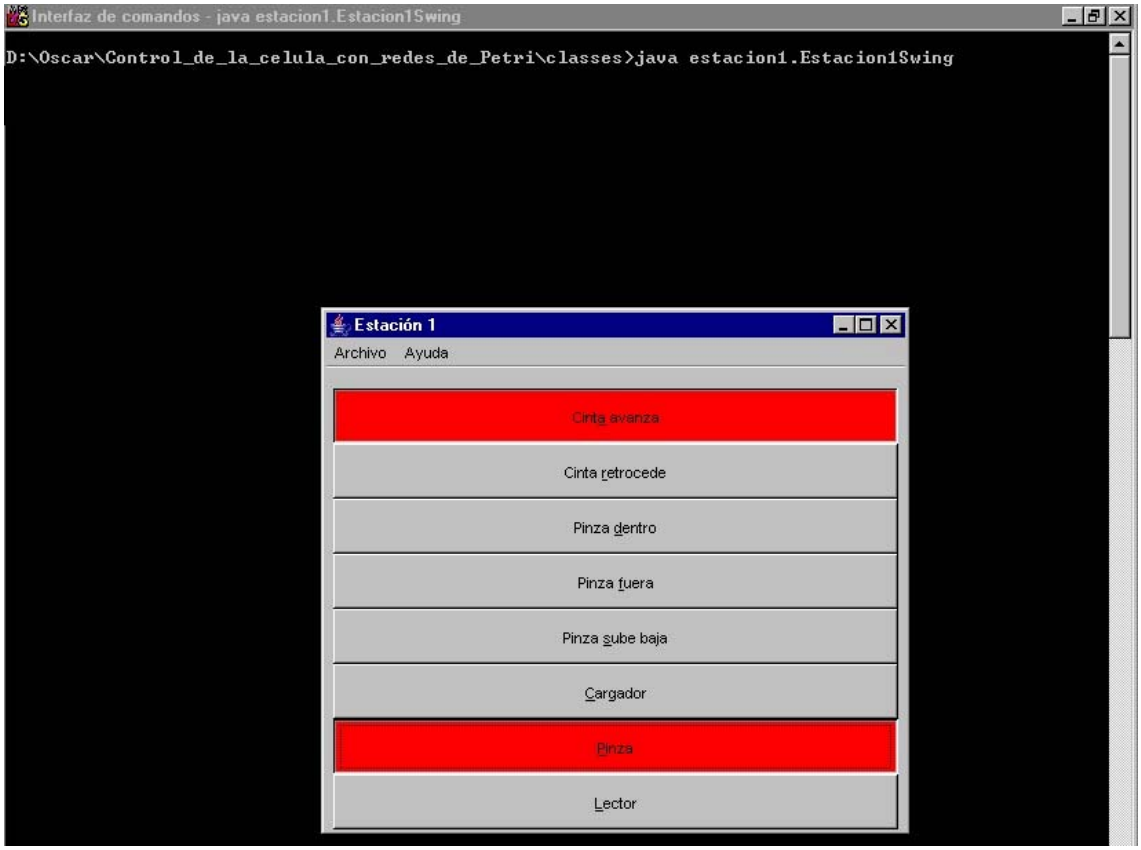


Figura 54: Estacion1Swing

Puede verse en la imagen que esta clase sirvió como primera prueba de lo que posteriormente se convertiría en la clase `BotonToggle` dentro del archivo `FrameModoManual.java`, que sirvió para implementar las pantallas de control de modo manual de la aplicación final.

AplicacionRdPE3

Esta aplicación fue la primera mini-aplicación en la que se utilizaba un `Coordinador` para controlar una estación mediante una RdP, en este caso creada manualmente. El código asociado a la gestión gráfica e inicialización está también distribuido por las clases `MarcoEst3` y `MarcoEst3_AcercaDe`.

Consiste únicamente de dos botones, uno que lanza el coordinador y otro que finaliza la aplicación.

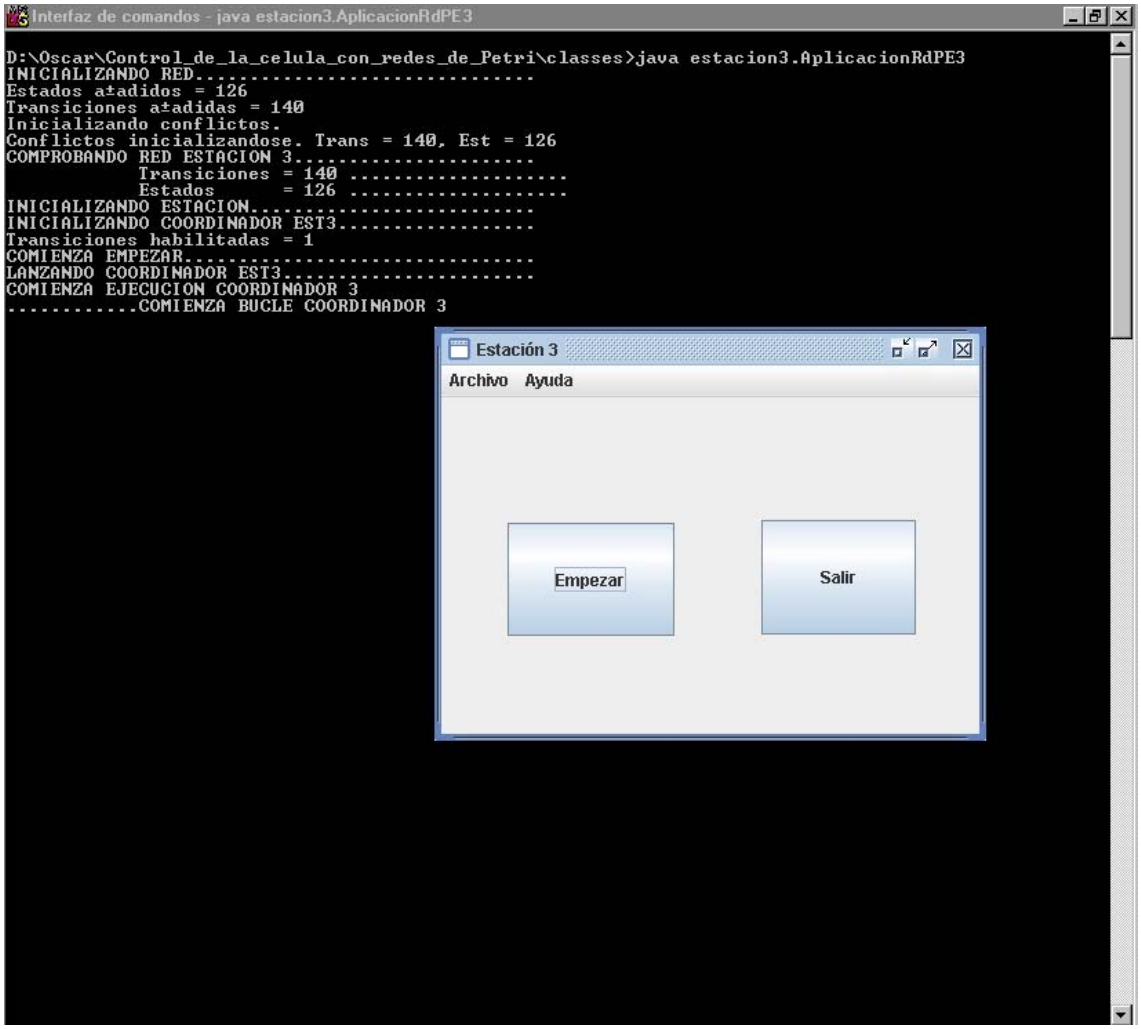


Figura 55: AplicacionRdPE3

Estacion3Swing

Esta aplicación es absolutamente similar en forma y estructura a `Estacion1Swing`, por lo que no se comentará nada más acerca de ella.

AplicacionRdPE4

Esta aplicación es similar en contenido y estructura a `AplicacionRdPE3`. El código asociado a la gestión gráfica e inicialización está también distribuido por las clases `MarcoEst4` y `MarcoEst4_AcercaDe`.

Consiste únicamente de dos botones, uno que lanza el coordinador y otro que finaliza la aplicación.

Estacion4Swing

Esta aplicación es absolutamente similar en forma y estructura a `Estacion1Swing`, por lo que no se comentará nada más acerca de ella.

PruebaAnalogica

La aplicación `PruebaAnalogica` se creó para comprobar el proceso de lectura y filtrado del valor analógico del verificador de piezas de la estación 4.

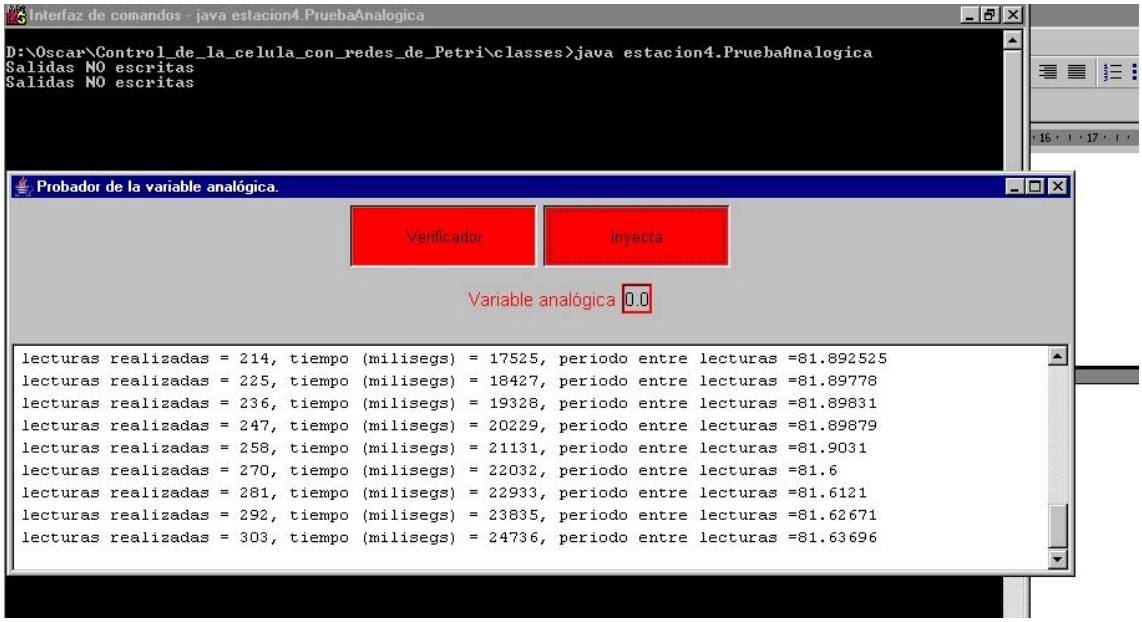


Figura 56: Prueba de la variable analógica

Consiste básicamente en dos botones que controlan dos salidas de la estación 4: la de subida y bajada del verificador y la de inyección de aire para comprobar la presión. Una etiqueta en el medio nos muestra la evolución del valor de la variable, mientras que la mayor parte de la ventana la ocupa un panel de texto en el que podemos ir viendo la evolución del programa.

Como ya se ha visto la variable analógica de la estacion 4 está representada por un objeto de la clase `VariableAnalogica`, en la que ya está implementado el filtrado. Esta aplicación sirve pues para jugar con los valores de alfa según la ecuación del filtrado y con los tiempos de ejecución de la tarea de lectura. La aplicación lanzará dos tareas periódicas, una de lectura y otra de actualización de pantallas, contadores, etc...

Sin embargo los resultados obtenidos por esta aplicación fueron que la variable analógica daba unos valores aparentemente aleatorios, por lo que no se puede distinguir entre piezas buenas y malas.

TestBorrar

Está aplicación surgió como necesidad a probar el coordinador `CoordinadorBorrar` en conjunción con la RdP contenida en `BorrarPalets.hps` y exportada a texto como `RedBorrar.txt`. Además permitió la primera implementación y prueba del método `restart()` antés de aplicarlo a la estructura del coordinador de la célula.

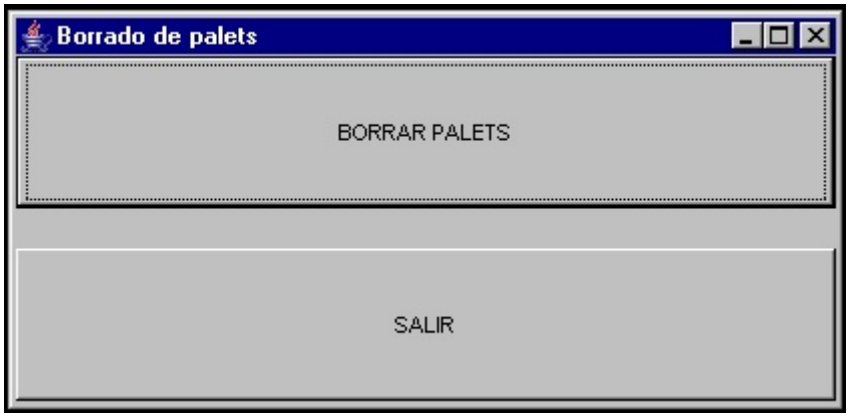


Figura 57: TestBorrar 1

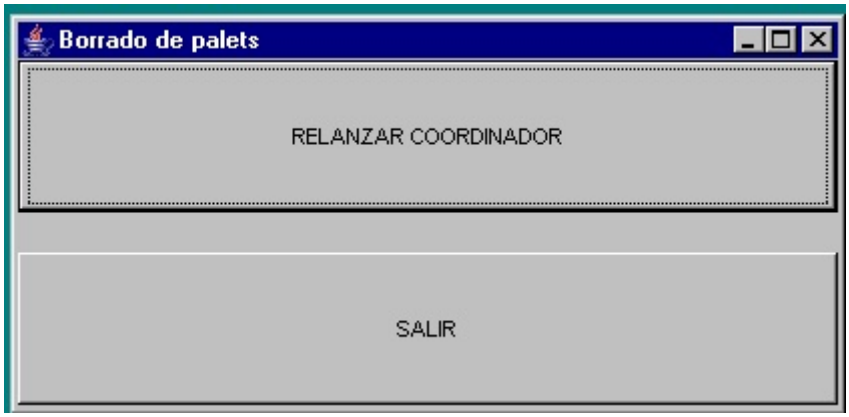


Figura 58: TestBorrar 2

Su estructura no puede ser más sencilla: dos botones, uno de salida y otro que lanza o provoca un reset del coordinador, inicializando a los valores iniciales. Como discusión posterior queda la pregunta de si un reset debe realizar una anulación de las salidas, con la problemática que conlleva el no conocer el estado del sistema en caso de realizarse (accidentes), o no hacerlo, con lo cual el coordinador necesita un sistema que devuélvale sistema a su valor inicial (inicialización).

Conclusiones y futuras líneas de investigación

Al finalizar este proyecto puede decirse que se han cumplido todos los objetivos que habían sido marcados en un principio. Se han creado una serie de clases Java que implementan la comunicación entre un PC y una serie de accionadores y captadores a través de bus Interbus, se ha realizado la implementación de un conjunto de clases que modelan una RdP y juntando ambas implementaciones se han creado una serie de aplicaciones de control sobre la célula de fabricación.

Para hacerse una idea de la complejidad del proyecto baste decir que anteriores proyectos fin de carrera se han ocupado de la implementación del modo automático de la célula a partir del PL7Pro de Telemecánica: es decir, aquí se ha hecho lo mismo pero *implementando nuestro propio software, es decir, partiendo casi de cero*, en vez de usar una herramienta creada por los profesionales de una de las más potentes casas de fabricantes de autómatas específicamente para la tarea. Pretender por tanto que la funcionalidad de las aplicaciones sea siquiera similar sería utópico. Lo que aquí se ha creado es una amplia base para posteriormente ir añadiendo funcionalidades que permitan, sino la equiparación con un software profesional, sí una serie de aplicaciones de gran utilidad y potencia.

Quizás lo más importante de este proyecto sea la increíble cantidad de posibilidades que ofrece de continuar progresando. Estas son las que podrían considerarse más importantes:

- Ante todo, creación de un editor gráfico de RdPs que incluya la funcionalidad de que cada estado y transición contengan información acerca de su condición de disparo y acciones asociadas respectivamente. Esto implicaría que los coordinadores no tendría que contener toda la información, que se podría implementar el concepto de subred, e implicaría la creación de una herramienta didáctica realmente útil.

- Expansión del cableado de Interbus a toda la célula de fabricación, para su posterior control.

- Revisión del código para su posible expansión a sistemas de tiempo real.

- Añadir funcionalidad a las clases relativas a la RdP: por ejemplo, funciones que detecten la corrección de la RdP, extensión a RdP coloreadas, etc.

- Añadir rigurosidad al código de gestión de excepciones, y más concretamente mejorar las posibilidades de detección de fallos en el bus, lo que implica la implementación de más y mejores métodos nativos y un manejo de excepciones más riguroso.

Bibliografía

Bibliografía escrita

Prácticas de Informática Industrial. Célula de fabricación flexible. EUITIZ, 2002.

INTERBUS User Manual. User Interface Version 2.x for High-Level Language Programming of INTERBUS Generation 4 Standard Controller Boards. Phoenix Contact, IBS PC SC HLI UM E.

Manuel Silva. "Las redes de Petri: en la Automática y la Informática." Editorial AC. ISBN 84-7288-045-1. 2002.

Schneider Electric. Manual de PL/7 Pro.

Jaworski, Jamie; "Java. Guía de desarrollo." Editorial Prentice Hall, Madrid, 1997. ISBN 1-57521-069-X

Bibliografía en Internet

Página web de la célula de fabricación. http://automata.cps.unizar.es/celula.html.

IBS PCI SC/I-T Data Sheet 6039B. www.phoenixcontact.com.

Interbus Club. http://www.Interbusclub.com/.

About java technology. http://www.java.com/en/about/java_technology.jsp/.

Jbuilder X Foundation. www.borland.com.

The Java Tutorial: Java Native Interface. http://java.sun.com/docs/books/tutorial/native1.1/index.html.

Editor HPSim. Copyright (C) 1999 - 2001 Henryk Anschuetz.
http://www.winpesim.de/petrinet/e/hpsim_e.htm.

How to use Threads. http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html.

How to use Swing Timers. http://java.sun.com/docs/books/tutorial/uiswing/misc/timer.html

IVI-KHD2-4HRX DataSheet. http://www.pepperl-fuchs.com

javax.comm page. http://java.sun.com/products/javacomm/