Control de la célula de fabricación flexible mediante redes de Petri en lenguaje Java.

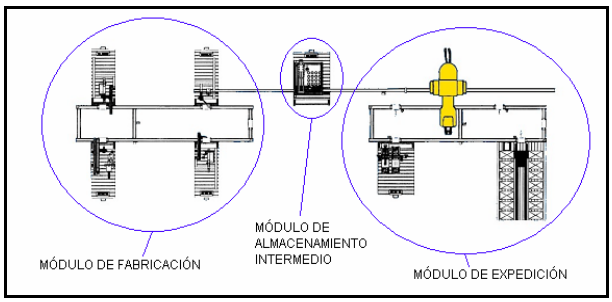
Ramón Piedrafita Moreno
Oscar García Flores
Departamento de Informática e Ingeniería de Sistemas. Universidad de Zaragoza.

El objetivo del presente trabajo es realizar el control de la célula de fabricación flexible del laboratorio del departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza mediante lenguaje Java a través de una red Interbus. Para ello se va a implementar un conjunto de clases que modelen el comportamiento de la célula y permitan su control mediante el uso de redes de Petri, implementadas también en Java. La comunicación con la célula se llevará a cabo mediante una tarjeta controladora de bus Interbus IBS PCI SC/I-T de Phoenix Contact.

1.- La célula de fabricación flexible.

La célula de fabricación flexible está formada por una serie de estaciones que permiten la producción y almacenamiento de cilindros neumáticos con diferentes colores y tapas. La célula está diseñada para funcionar como ayuda para el estudio de un proceso de fabricación real sin tener que recurrir a un proceso industrial a escala completa.

La célula se divide en dos partes: la zona de producción, constituida por las estaciones 1, 2, 3 y 4 y el transporte 1, y la zona de expedición con las estaciones 6, 7, 8 y 9 y el transporte 2. La estación 5 es el módulo de almacenamiento intermedio que actúa como enlace entre los dos módulos.



Se han llevado a cabo numerosos proyectos fin de carrera sobre ella basados en la comunicación de la estaciones mediante una red Fipway para posteriormente, mediante autómatas Premium y Micro de Modicon Telemecanique, controlarlas con ayuda del software PL7 pro.

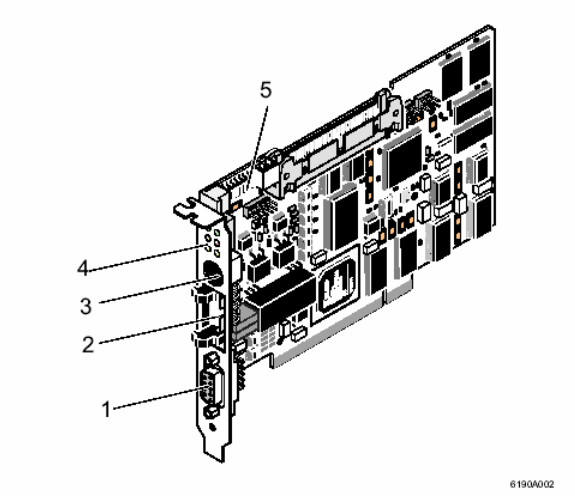
Para el presente proyecto se parte de una base distinta. Se ha construido una red Interbus entre las estaciones 1, 3 y 4 y el transporte 1 (zona de producción). Así, en las estaciones 1 y 4 se han colocado módulos Interbus Inline completamente modularizables mientras que en la estación 3 y el transporte 1 se ha optado por la conexión mediante un módulo TSX Momentum de Schneider Electric de 16 entradas/salidas digitales y 2 entradas/salidas analógicas. Se ha de hallar la forma de controlar las estaciones con esta base. Si se desea más información acerca del funcionamiento concreto de las estaciones se puede recurrir a cualquiera de los muchos proyectos fin de carrera realizados hasta la fecha , al manual para las prácticas de la asignatura de informática industrial de la Escuela Universitaria de Ingeniería Técnica Industrial de Zaragoza, EUITIZ [1], o a la documentación disponible en la página web [2].

2.- La tarjeta controladora de bus IBS PCI SC/I-T. INTERBUS.

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. [3]

Según la página oficial de Interbus [4] 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.



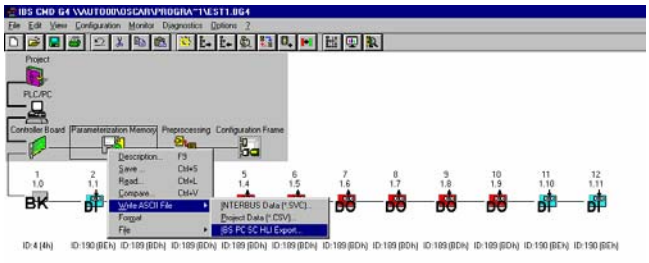
En la figura puede observarse la configuración de la tarjeta:
1 - Interfaz de bus remoto Interbus.
2 - Conexión directa de entradas y salidas.
3 - Interfaz RS-232.
4 - LEDs de diagnóstico.
5 - Interruptores DIP para el establecimiento del número de la tarjeta.

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.

2.1- El HLI (High-level Language Interface).

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 [5] 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.



Por desgracia estas librerías sólo están disponibles en los siguientes lenguajes de programación:
- Microsoft C/C++.
- Borland C/C++ (o compatible).
- Microsoft VB 4.0 (o posterior).
- Borland Delphi 2.0 (o posterior).

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.

3.- De C a Java. JNI.

3.1 - 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) [6].

3.2 – Proceso de conversión.

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 seis pasos:

1 - Escribir el código Java.
2 - Usar javac para compilar el código.

3 - Usar javah –jni para generar un fichero de cabecera.
4 - Escribir la implementación del método nativo.
5 - Compilar el código nativo en una librería compartida y cargarla.
6 - Ejecutar el programa usando el interprete Java.

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 cada estación contiene el método:

```
public native void InicializaEstacion();
public native void FinalizaEstacion();
```

Posteriormente se han creado modificado los métodos nativos para permitir una inicialización o finalización conjunta de todas las estaciones de la célula. 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.

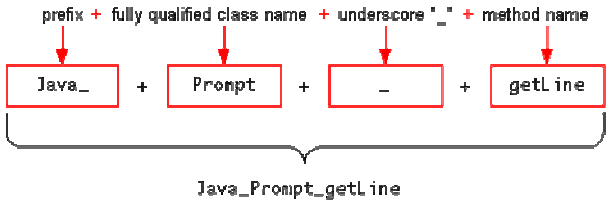
Para escribir el código Java de este PFC se ha usado JbuilderX Foundation [7], en su versión gratuita. Para compilar las clases Java se ha usado el interfaz de comandos de Windows NT mediante la aplicación `javac` con lo que resultan los archivos `.class`.

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:    actualizaVariable
 * Signature: ()V
 */
JNIEXPORT void JNICALL
Java_VariableBooleana_actualizaVariable
  (JNIEnv *, jobject);
```

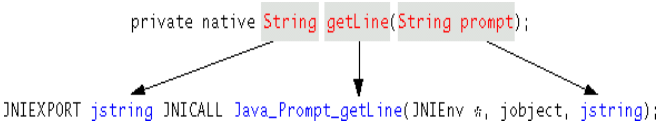
Este proceso es similar para todas las funciones nativas que se quieran utilizar en un programa. El nombre de las funciones se divide en varias partes, según se explica en la figura:



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 es conveniente recurrir a los 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 trabajar con código C.
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:



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) {
  if( PCI1.BusState == HLI_IBS_RUN)
    IBS_HLI_PD_In(PCISC1);
  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;
}
if( PCI1.BusState == HLI_IBS_RUN )
IBS_HLI_PD_Out(PCISC1);
        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);
  /* ......................*/
  /* Resto de registros... */
  /* ......................*/
  /* 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 las etiquetas:

```
#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("Celula");
  }
```

`Celula.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 mismo.
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(Syste
m.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.

La información necesaria acerca del JNI puede hallarse en el tutorial de Java sobre JNI [8].

4.- Redes de Petri.

Una red de Petri (RdP) es un grafo orientado a objetos en el que intervienen dos clases de nudos, los lugares (representados por circunferencias) y las transiciones o lugares (representadas por segmentos rectilíneos), unidos alternativamente por arcos [9].

Hasta ahora la única implementación de redes de Petri disponible en el departamento se realizó para un control de robot en lenguaje Ada. Esta implementación se basaba en la creación de un coordinador que recorría la red en cada iteración disparando las transiciones según un modelo de conflictos que había de ser realizado a ojo durante la creación de la red. Esta implementación obviaba la existencia de los estados basando la estructura de la red en vectores de transiciones.

Dada la potencia que se posee al utilizar Java el uso de un modelo similar para este proyecto se consideró poco aconsejable. En vez de eso se ha creado un modelo nuevo que incluye los estados y cuya estructura se explica a continuación.

4.1 Las clases básicas: Estado y Transición.

Como ya se ha explicado Java es un lenguaje orientado a objetos; por ello el primer paso fue considerar los posibles objetos que intervienen en una RdP. De ahí surgen inmediatamente las clases `Estado` y `Transición`. Dado que en un principio no se iba a crear una interfaz gráfica de creación de redes se decidió no implementar una clase `Arco` que habría complicado la estructura de la RdP, sino basar dicha estructura en los estados o lugares de entrada y/o salida de las transiciones. Así:

```
public class Estado
  int tokens = 0;
  Temporizador temporizador;

public class Transicion
  boolean habilitada = false;
  Vector <Estado> lugaresEntrada;
  Vector <Estado> lugaresSalida;
  int prioridad;
```

Se ha aprovechado la clase `java.util.Vector` en su nuevo formato de la versión 1.5.0 del JDK (Java Development Kit). La estructura de la RdP quedará definida por un conjunto de estados y transiciones siendo estas últimas las que soportan la estructuración de la red. La temporización de estados y la prioridad de transiciones son herramientas auxiliares para el control de un proceso real simulado con esta implementación de RdP. Los métodos que implementan estas clases son:

Transicion:

```
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 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);
```

Estado:

```
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();
```

4.2 La clase Red.

La clase Red es la que contiene la información acerca de la estructura de la RdP. Dado que la clase Transicion no contiene ninguna información acerca de la condición de disparo de la misma para el control de la evolución de la RdP se necesitará un coordinador. La clase Red tiene los siguientes campos:

```
public class Red {
  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;
```

La definición de los términos asociados a la RdP de los que se ha deducido los campos de esta clase se pueden encontrar en [10]. Su significado se puede ver intuitivamente (al contrario que en otras implementaciones de estructura más enrevesada); la RdP se define por sus matrices de incidencia previa y posterior, por su marcado y por su marcado inicial. La matriz de pesos se hace necesaria para implementar la funcionalidad de los arcos. Todos estos parámetros pueden sacarse de la información de los vectores de estados y transiciones o de forma inversa puede construirse la RdP con sus matrices de incidencia y colegir de ellos el conjunto de estados y transiciones que lo componen.

Un conflicto se produce cuando dos o más transiciones simultáneamente sensibilizadas descienden de un mismo lugar y este no dispone de un número de marcas suficientes para dispararlas simultáneamente [11]. Por tanto un conflicto será básicamente un vector de transiciones que cumplen dicha condición. La implementación de la funcionalidad de los conflictos será de utilidad a la hora de implementar el coordinador de la RdP. Además la creación de este vector se hace de forma automática al crearse la RdP, por lo que no hace falta un proceso de estudio de la RdP previo a la realización programada concreta de la misma tal y como se hacía con la implementación previa en Ada.

Loa métodos que esta clase proporciona son:

```
public Red();
public Red(Vector < Estado > estad, Vector <
Transicion > trans);
public void construyeRedMatrizIncidencia(int[ ][
] matrizInc, int[ ] marcado);
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 void imprimeTicks();
public boolean esTransicionConflictiva(Transicion
t);
public Vector <Conflicto>
getConflictosConEstaTransicion (Transicion t);
public Vector < Estado > getEstadosMarcados();
```

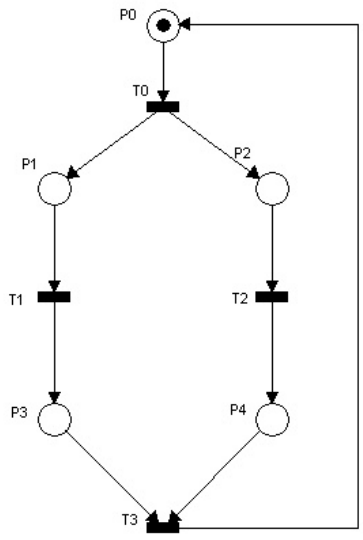
Los métodos que implementan la creación propiamente dicha de la RdP son `construyeRedMatrizIncidencia`, que crea la red a partir de su matriz de incidencia y el marcado inicial y `construyeRedVectores` que crea la RdP a partir de los vectores de estados y transiciones creados al efecto.

4.2.1 Creación de una RdP.

Como se ha explicado anteriormente existen dos maneras de crear una RdP: mediante su matriz de incidencia y su marcado o mediante sus vectores de estados y transiciones.

La primera esta pensada para aprovechar la capacidad del editor de RdPs HPSim [12] de proporcionar los parámetros de una RdP en modo texto. Cogiendo dos de estos parámetros del archivo de texto y pasándolos al método que construye la RdP ya se tiene una RdP correctamente construida.

El otro método está pensado para crear la RdP mediante la creación de una subclase de Red que contenga únicamente los estados y las transiciones correctamente implementados, como se verá a continuación mediante el siguiente ejemplo: sea la siguiente RdP:



La red creada con HPSim proporciona un archivo de texto con el siguiente formato:

```
// 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 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 [13]. Hay que tener en cuenta pues que si se pretende modelar una RdP no pura [14] sólo se podrá realizar con el segundo método.

Para crearla manualmente, se deberá implementar una subclase de Red de la siguiente manera:

```
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);
}
```

De esta forma se pueden construir redes de una forma rápida y limpia. Conforme más compleja sea la RdP más conveniente puede ser el usar la primera forma de crear la red, pero no se debe olvidar que a la hora de aplicar la RdP a un proceso concreto debe implementarse la condición de disparo de cada transición de la misma, además de las posibles acciones asociadas al disparo de la transición o a la entrada, salida o mantenimiento de un estado, por lo que la segunda forma ayuda a evitar confusiones provocadas por la numeración de las transiciones en el primer modo.

4.3 El coordinador.

La creación de una implementación particular de RdP se ha llevado a cabo con la intención de controlar la célula de fabricación. Para ello se dispone de las clases “estacion”: `Estacion1, Estacion3, Estacion4 y Transporte` , y de las clases que implementan la RdP. Para juntar todo necesitaremos una superclase que efectué el control de los disparos de la RdP a partir de las distintas entradas provenientes de los sensores de la célula y que active las salidas consiguientes. A esta clase se le llama `Coordinador`. La implementación de `Coordinador` posee los siguientes campos:

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

Es obvio que el coordinador va a necesitar un objeto Red, que representa el sistema a controlar. El vector transicionesHabilitadas es un subconjunto del vector de transiciones de la RdP que corresponde a todas las transiciones de la red que están habilitadas para disparar,

es decir, que su/s estados/s de entrada poseen las marcas suficientes para que se produzca el disparo si se cumple la condición de disparo. La variable monitor aprovecha otra nueva funcionalidad del JDK 1.5.0: el paquete `java.util.concurrent` que contiene clases que ayudan en la programación concurrente: en este caso la clase `ReentrantLock` hace que sólo un hilo a la vez pueda acceder a los contenidos de la red (para evitar por ejemplo disparos simultáneos de transiciones en conflicto). La variable r nos servirá para elegir una transición u otra en caso de que haya varias de ellas que estén habilitadas, se cumpla su condición de disparo, no estén en conflicto y que tengan la misma prioridad.

Puede que choque que este Coordinador no contenga ninguna "estación" a controlar; esto es porque lo que se pretende con esta clase, al igual que se planteó para la clase Red, es crear una clase padre para la implementación de distintos coordinadores específicos que desciendan de ella. Los métodos que define son:

```
public Coordinador(Red r);
protected native void inicializaComunicacion();
protected void inicializaEstacion(Estacion
estacion);
protected void finalizaEstacion(Estacion
estacion);
public int getRandomTrans();
public void setTransicionesHabilitadas();
public Transicion getTransicionADisparar();
public void disparaTransicion(Transicion t);
```

Los métodos de inicialización y finalización son específicos del proyecto (ya que todos los coordinadores que se van a construir van a ser para controlar la célula), mientras que los otros cuatro métodos implementan la funcionalidad básica de todo coordinador de RdP: encontrar las posibles transiciones a disparar, implementar el disparo (desmarcado de lugares de entrada, marcado de lugares de salida y código asociado a la entrada o salida de estados) y disparo efectivo de transiciones[15]. Dado que el presente proyecto trata acerca de automatizar un proceso industrial real (aunque a escala más pequeña) se ha tenido en cuenta a la hora de la implementación los programas de las diferentes casas fabricantes de autómatas programables basados en RdPs. Sobre todo, dado que los autómatas del laboratorio pertenecen a Modicon Telemecanique, se ha tenido en cuenta la estructura de Grafcet del PL7Pro en particular a la hora de diseñar la estructura de acciones asociadas a la entrada, salida o mantenimiento de un estado, así como a la creación de temporizadores para conocer el tiempo de marcado de un estado [16].

Para la implementación de un coordinador específico de un proceso concreto con una RdP asociada se muestra el coordinador de la estación 4 de la célula:

```
public class CoordinadorEstacion4
    extends Coordinador
    implements Runnable {
  Estacion4 est4;
  Timer timerLectura, timerGlobal;
  public volatile long counter = 0;
  Date inicio, fin;
  Vector <Estado> estadosMarcados, estadosAntes,
estadosDespues;
  boolean piezaBuena = false;
```

Lo primero sobre lo que conviene llamar la atención es que se ha considerado conveniente que el coordinador específico además de descender de `Coordinador` implemente la interfaz `Runnable`, es decir, que pueda ser lanzado como un hilo de ejecución independiente. Para ello esta clase deberá definir un método `run ()` en el que se inicializarán los timers asociados al mismo.

Dado el carácter cíclico del control lo conveniente es hacer que se realice cada cierto tiempo. Esto en Java conduce inevitablemente al uso de la clase `Timer`. Una vez tenido esto en cuenta, la decisión a tomar pasa a ser si se usa la clase `java.util.Timer` o la `javax.swing.Timer`. Dado que la mayoría de procesos a controlar van a necesitar de una interfaz gráfica de interacción con el usuario en este caso se ha utilizado la segunda opción.

Dado que se necesita activar o desactivar salidas de las estaciones de la célula a partir de las entradas se necesitará un medio para gestionar estas acciones. Esto se hará mediante los métodos:

```
public boolean condicionDisparo(Transicion t);
public void accionPrevia(Estado e);
public void accionPosterior(Estado e);
public void accionContinua(Estado e);
```

Basta echar un vistazo al código para comprender el funcionamiento del método:

```
public boolean condicionDisparo(Transicion t) {
    int i = super.red.transiciones.indexOf(t);
    switch (i) {
      case 0:
        return est4.Marcha.getValor();
        //return true;
      case 1:
        return est4.Cilindro_abajo.getValor();
      case 2:
        return est4.Vacio_pinza.getValor();
//Resto de las transiciones…
  default:
        return false;
    }
  }
```

Se puede observar claramente la dependencia total de la RdP con las condiciones físicas impuestas por el correcto funcionamiento de la célula; la RdP no puede progresar si no se cumplen unas ciertas condiciones provenientes de las estaciones, lo cual justifica la decisión de crear una estructura basada en un Controlador abstracto y coordinadores específicos del sistema a controlar.

4.3.1 Timers

El uso de timers y threads en aplicaciones gráficas está muy documentado y puede encontrarse mucha información al respecto en la página del tutorial de Java [17][18].

A grandes rasgos podemos decir que un objeto Timer del paquete `javax.swing` necesita como parámetro un `ActionListener`, para después ejecutar el código asociado al mismo. Por ejemplo, en nuestra clase CoordinadorEstacion4 el código del timer encargado de leer periódicamente las entradas de la estación, que se encuentra en el constructor, es el siguiente:

```
ActionListener tareaLectura = new
ActionListener() {
  public void actionPerformed(ActionEvent evt) {
        est4.leeEntradas();
        counter++;
        fin = new Date();
      }
    };
int delayLectura = 50;
timerLectura = new Timer(delayLectura,
tareaLectura);
```

Esto hace que cada 50 milisegundos (según el método `Object.wait()`) se ejecute el código asociado al timer, que en este caso solo lee las entradas de la estación, actualizando su valor, y cambia unos determinados campos cuya única función es la monitorización del proceso para ver si realmente se cumple la especificación temporal de 50 ms. Obviamente el proceso para la creación del otro timer, `timerGlobal`, será similar, con la diferencia que el código asociado a este será el encargado de gestionar los posibles disparos de la red, es decir, el auténtico coordinador.

El método run () sólo tendrá que lanzar los timers:

```
public void run() {
    this.inicializaEstacion();
    this.timerLectura.start();
    this.timerGlobal.start();
    inicio = new Date();
  }
```

Para gestionar la RdP se ha escogido un método que no tiene en cuenta los posibles conflictos estructurales que puede tener la red. El proceso que se sigue es:

1- Se realiza la acción continua de los estados marcados.
2- Se crea un vector con todas las transiciones habilitadas.
3- Se escoge una de ellas según su prioridad, o en caso de igual prioridad, aleatoriamente.
4- Se comprueba que su condición de disparo se cumpla.
5- Se dispara la transición.
6- Se ejecutan el código asociado a la salida de los estados de entrada y al marcado de los estados de salida.

Obviamente este algoritmo no tiene en cuenta conceptos tales como el tiempo real, imposible de controlar al estar trabajando en un sistema con Windows NT. Una posible mejora sería hacer que el algoritmo se ejecutara hasta que no quedara ninguna transición habilitada que cumpliera su condición de salida.

5.- El paquete puertoSerie. Identificador de productos.

Como ya se ha comentado el módulo de fabricación de la célula de fabricación consta de cuatro estaciones por las que va pasando un palet con la pieza producida hasta que se produce su salida al almacén intermedio. Pero para el control global del proceso de producción es necesario saber que tipo de pieza se va a realizar y cual es exactamente el contenido de cada palet. Es por esto que la célula dispone de un identificador de productos modelo IVI-KHD2-4HRX [19]. Este identificador permite la colocación de hasta cuatro cabezales de lectura/escritura, que se colocan en cada estación. Según el proceso realizado en cada una de ellas se escribirá una determinada información en el disco magnético situado en la parte inferior del palet que podrá ser utilizada por las siguientes estaciones cuando el palet llegue a ellas para actuar de forma coherente con el contenido del mismo.

Dado que este identificador aún no está conectado a la red Interbús del laboratorio la única opción es su control a través del puerto serie mediante protocolo RS232. Para ello se han creado cuatro clases que se encargaran de gestionar las comunicaciones con el identificador de productos:

- Comunicaciones
- Palet
- Trama
- IdentificadorProductos

5.1 Comunicación por puerto serie.

Aunque el JDK no incluye paquetes específicos para el control de los puertos del ordenador, Sun Microsystems si que ha creado un paquete específico dirigido a desarrolladores que deseen crear aplicaciones que deban intercambiar información por un puerto del ordenador: el paquete `javax.comm`. Este paquete contiene clases específicas para la detección y el control de puertos del PC bajo un sistema Windows o Solaris x86 [20]. Dada la escasez de información proporcionada en el API lo más cómodo y efectivo es acudir a alguno de los ejemplos que vienen con el paquete para asimilar el funcionamiento de las aplicaciones.

5.2 Protocolo de comunicaciones.

Para la comunicación entre el PC y el identificador se deberá usar un protocolo definido en el manual de usuario del identificador [21]. El IVI-KHD2-4HRX permite la conexión directa mediante protocolo RS232 a través de un cable de conexión de 9 pines. Dado que la lectura y escritura de palets va a ser un proceso que se realizará pocas veces en comparación con la lectura o escritura de entradas o salidas de las estaciones seleccionamos el modo de funcionamiento más sencillo: el basic read/write operating mode. En este modo se pueden enviar órdenes al identificador de dos tipos: lectura/escritura o comandos del sistema.

Todas las órdenes que se le envían al identificador deben tener una estructura fija: por ejemplo, para leer un número de bytes del cabezal 3 desde la posición 8 la instrucción a enviar sería:

w<HdNo><StAdrH><BytesH><CHCK><ETX>

y la respuesta en caso de que no haya habido problemas será del tipo:

w<Status><DB><CHCK><ETX>

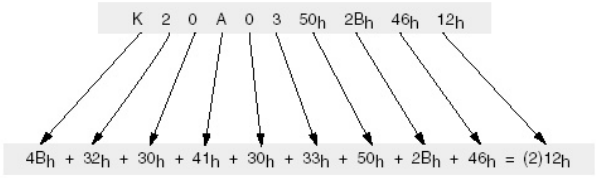
Para tratar esto en Java se ha creado la clase Trama. Esta clase básicamente es un vector de bytes correctamente formados según el protocolo del identificador. Por desgracia esto que es tan fácil de decir es problemático a la hora de programar, dados los estrictos requisitos a la hora de crear estas tramas. Por ejemplo, la dirección de inicio de lectura o escritura (StAdrH) debe ser de dos cifras en formado hexadecimal y pasado luego como dos bytes a la trama. Así que el primer paso fue crear un amplio conjunto de constantes para que después el proceso de construcción de la trama se realizara por el sencillo proceso de ir añadiendo bytes al vector de la clase `Trama`. En este caso un ejemplo sacado de la implementación de `Trama` vale más que cualquier explicación:

```
  public static final byte ETX = 0x03;
  public static final byte CABEZAL1 = 0x31; //'1'
  public static final byte CABEZAL2 = 0x32; //'2'
  public static final byte LEER = 0x77; //w
  public static final byte ESCRIBIR = 0x6B; //k

//Para las direcciones de inicio y bytes a leer o
//escribir:
//Método -> por ejemplo 12 decimal
//  12d = 0Ch --> '0' = 30h, 'C' = 43h -->
//  DOCE = {0x30,0x43}

  public static final byte[] CERO = {
      0x30, 0x30};
  public static final byte[] UNO = {
      0x30, 0x31};
  public static final byte[] DOS = {
      0x30, 0x32};
```

Una vez se tienen estas constantes basta con la creación de métodos para añadir éstas a la trama:

```
public void añade(byte[] b);
public void añade(byte b);
```

Mención aparte merece el cálculo del CHECKSUM. El CHCK se calcula como la suma de todos los bytes de la trama excepto el carácter de fin de trama ETX. Si el valor final de esta suma es mayor de tres dígitos (hexadecimales) se trunca el más significativo[22]. Asi:



Para gestionar esto se han implementado los métodos:

```
public byte trunca(int num);
public byte hazChecksum();
public void finalizaTrama();
```

Una vez conseguido juntar todo de manera satisfactoria sólo resta crear unas cuantas tramas de uso común para su utilización de forma directa; esto se ha hecho mediante la clase soporte `Comunicaciones` cuya única función es contener estas tramas. Ejemplos:

```
salir = new Trama();
salir.añade(Trama.QUIT); //comando
salir.finalizaTrama(); //checksum y ETX

leer2 = new Trama();
leer2.añade(Trama.LEER); //comando
leer2.añade(Trama.CABEZAL2); //cabeza 2
leer2.añade(Trama.CERO); //dirección de inicio
leer2.añade(Trama.VEINTIUNO); //bytes a leer
leer2.finalizaTrama(); //checksum y ETX

escribir1 = new Trama();
escribir1.añade(Trama.ESCRIBIR); //comando
escribir1.añade(Trama.CABEZAL1); //cabeza 1
escribir1.añade(Trama.CERO);//dirección de inicio
escribir1.añade(new byte[44]);//bytes a escribir
escribir1.finalizaTrama();  //checksum y ETX
```

5.3 Pasando tramas a datos.

Para pasar las tramas recibidas a una serie de campos que podamos leer con el programa necesitamos una clase que encapsule la información necesaria: la clase Palet:

```
public class Palet {
  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;
```

Estos son los datos básicos a almacenar en el palet: la última fecha de modificación, ya sea de lectura o de escritura y el tipo de pieza a fabricar, si hay una camisa en el palet, etc... El tipo de pieza viene determinado por las siguientes constantes:

```
  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;
```

La clase Palet también proporcionará métodos para pasar los datos a una cadena de bytes, que podrá ser añadida después a una `Trama`, y para leer los datos de una cadena de bytes.

```
public byte[] pasaAscii(int digito);
public int getEnteroDeAscii(byte [] b);

public byte[] toBytes();
public int aDatos(Trama t);
```

Al método aDatos se le da como parámetro un objeto Trama, ya que la cadena de datos que queremos decodificar estará contenida siempre dentro de una Trama, que nos habrá sido enviada desde el identificador después de una petición de lectura.

Además proporciona los métodos para modificar los campos de la manera usual:

```
public int setTipoPieza(byte tip);
public byte getTipoPieza();
public boolean tieneCamisa();
public void setCamisa(boolean valor);
public boolean tieneEmbolo();
public void setEmbolo(boolean valor);
public boolean tieneMuelle();
public void setMuelle(boolean valor);
public boolean tieneCulata();
public void setCulata(boolean valor);
public boolean esPiezaConTapa();
public void setPiezaConTapa(boolean valor);
public boolean hayPiezaEnPalet();
public void setPiezaEnPalet(boolean valor);
```

6.- Conclusiones y futuras lineas de investigación.

En el momento de escritura de este informe el proyecto está en su fase final de implementación, habiéndose probado con éxito la implementación de RdP de modo automático sobre cada estación independientemente de las demás. Falta pues el paso final de crear la RdP que controla la célula y la creación de su coordinador.

Este proyecto sienta las bases para posteriores proyectos. Como ejemplos se puede citar la extensión del control a toda la célula, el control del identificador de productos como dispositivo PCP del bus Interbus, estudio de tiempos de producción de la célula con las nuevas redes implementadas, etc... También se está llevando a cabo paralelamente a la finalización de este proyecto un intento de aplicar la estructura de RdP a un control en tiempo real mediante RT-Linux a cargo de Alberto Gran Tejero, dirigido también por Ramón Piedrafita Moreno, director e impulsor del presente proyecto.

Referencias:

[1] Prácticas de Informática Industrial. Célula de fabricación flexible. EUITIZ, 2002.

[2] Página web de la célula de fabricación. http://automata.cps.unizar.es/celula.html
En particular, "Modelo, control de tiempos y obtención de prestaciones de una Célula Flexible de Fabricación". (Proyecto fin de Carrera). Pablo Zorraquino Guallar. Universidad de Zaragoza. 2004.

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

[4] Interbus Club. http://www.Interbusclub.com/.

[5] 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. www.phoenixcontact.com.

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

[7] Jbuilder X Foundation. www.borland.com.

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

[9] Manuel Silva. "Las redes de Petri: en la Automática y la Informática." Editorial AC. ISBN 84-7288-045-1. 2002. p. 16.

[10] Manuel Silva. "Las redes de Petri: en la Automática y la Informática." Editorial AC. ISBN 84-7288-045-1. 2002. pp. 30-34.

[11] Manuel Silva. "Las redes de Petri: en la Automática y la Informática." Editorial AC. ISBN 84-7288-045-1. 2002. p. 22.

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

[13] Manuel Silva. "Las redes de Petri: en la Automática y la Informática." Editorial AC. ISBN 84-7288-045-1. 2002. p. 30.

[14] Manuel Silva. "Las redes de Petri: en la Automática y la Informática." Editorial AC. ISBN 84-7288-045-1. 2002. p. 31,.

[15] Manuel Silva. "Las redes de Petri: en la Automática y la Informática." Editorial AC. ISBN 84-7288-045-1. 2002. p. 33,.

[16] Schneider Electric. Manual de PL/7 Pro.

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

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

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

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

[21] IVI-KHD2-4HRX Manual. p. 12. http://www.pepperl-fuchs.com

[22] IVI-KHD2-4HRX Manual. p. 27. http://www.pepperl-fuchs.com