The Java based Programmable Logic Controller. New Techniques in Control and Supervision of a Flexible Manufacturing Cell.

Ramón Piedrafita and José Luis Villarroel
Department of Computer Science and Systems Engineering, University of Zaragoza
Spain

1. Introduction

In this chapter, we explain new techniques and technologies applied to the control of a flexible manufacturing cell. We have proposed a new control platform: a Java based Programmable Logic Controller (PLC). The Java PLC comprises several modules where the real time control, the communication with industrial fieldbuses and the supervision via web technologies have been developed. This control architecture (Piedrafita & Villarroel 2006) and development environment is based on Petri Nets (PNs), Sequential Function Charts (SFCs) and the Real Time Java programming language. Our objective is to explore the application of new control techniques of manufacturing systems, and to test the use of the Java programming language as a platform for those techniques. To demonstrate the practical utility of the techniques, we have applied them to the control of a flexible manufacturing cell.
This research follows earlier studies at the University of Zaragoza on the software implementation of PN. In those studies, Ada95 was the implementation language (García & Villarroel 1996). For the current study, Java was chosen for the following reasons:

- The possibility of executing the same code on different platforms.
- To compare the Ada95 and Java implementations using the same concurrent and real-time characteristics.
- Java is a language that is beginning to be used in the development of control and embedded systems.
- Java has a real time extension that allows the necessary time predictability required in these types of applications.

From the perspective of the software implementation of Discrete Event Control Systems, we have translated into the Java language some classic PN implementation techniques such as Enabled Transitions or Representing Places. We have also developed SFC implementation techniques such as the Deferred Transit Evolution Model and Immediate Transit Evolution Model.
In the execution of a Petri net, a task, the coordinator, makes the firing of transitions and updates the marking. We propose a new PN implementation technique called concurrent coordinators, in which centralized and decentralized ideas are merged. The net is decomposed into several subnets following, for example, a functional criterion, and a

Source: Programmable Logic Controller, Book edited by: Luiz Affonso Guedes,
ISBN 978-953-7619-63-3, pp. 170, January 2010, INTECH, Croatia, downloaded from SCIYO.COM

different coordinator implements each subnet in a centralized way. The communication between the different subnets is made using communication places that are implemented by monitors that use synchronized methods for marking and unmarking.
The Java PLC can load PNs in PNML (Billington, Christensen et al. 2003) and can load SFCs in the PLCopen XML format (Open 2005). In the input/output module we have developed libraries for communication with bus Interbus, CANopen and Modbus TCP/IP.
The hardware of the Java PLC is based on a PC equipped with several fieldbus master cards, where the input/output modules of the machines are connected. The software of the control system is based on a real-time control program that has several threads responsible for the concurrent execution of PN or SFC programs. The threads communicate through monitors.
The development of this open platform has involved the development of an open framework of Java classes. This includes classes for the PN and SFC implementation, from the basic classes that model *places*, *transitions*, and the structure of a Petri net to the classes that execute the PN, and classes that allow communication with several fieldbuses. There are also classes for remote supervision via the RMI protocol or web Supervision.
A practical application of the proposed approach, the control of a flexible manufacturing cell, was developed at the Department of Computer Science and Systems Engineering at the University of Zaragoza, Spain.
The remainder of this chapter is organized as follows. In Section 2 we present the characteristics of the Java language relevant to the implementation of control systems and we present the Java PLC structure. In section 3 we review the different techniques for the implementation of PNs and SFCs. The flexible manufacturing cell is described in Section 4. In Section 5 the communication with the controlled system is explained. Section 6 describes the structure of the proposed control architecture and in section 7 we show a new technique: the Execution Time Controller. In Section 8, some details of the software implementation in the Java language are presented. The development environment we have created is described in Section 9 and, in Section 10 we present conclusions and suggest future lines of research.

2. Java based programmable logic controller

This research follows earlier studies at the University of Zaragoza on the software implementation of PN. In those studies, Ada95 was the implementation language (García & Villarroel 1996).
Java has some of the characteristics required for the implementation of control applications, including the concurrent execution of threads; however, Java also has characteristics that impede its use as a programming language for control applications:

- Although there exits compilers, Just in Time compilers, Virtual Machines with onthe-fly recompilation, Java mainly is used like interpreted language.
- Programs written in Java link classes dynamically; thus, the load of remote classes can delay the execution of the program.
- Java uses dynamic memory to create objects as needed, and the time to create an object depends on the memory state.
- A high-priority thread performs the garbage collection, the process of automatically freeing objects that are no longer referenced by the program.

- The Java scheduler works with fixed priorities, but there is no guarantee that the highest-priority thread is running. To avoid starvation, the thread scheduler might choose to run a lower-priority thread. The thread priority affects the scheduling policy for efficiency purposes only.

The first characteristic affects the run-time performance only, but the others imply time unpredictability, which prevents Java from being used in the development of real-time systems. These problems have been solved in the *Real Time Java Specification* (RTJS) (Bollella and Gosling 2000.) which has been implemented in some platforms, such as JamaicaVM (Aicas 2007) which includes the following:

- The Real Time Virtual Machine executes "bytecodes," but programs are also compiled to machine code, which increases the speed of execution.
- New thread classes, RealtimeThread and NoHeapRealtimeThread, which are unaffected or at least less heavily affected by garbage collection activity.
- New memory classes (InmortalMemory and ScopedMemory) that are not under the control of the garbage collector.
- A true fixed-priority pre-emptive scheduler that has an expanded range of priorities.
- The incorporation of classes for the treatment of asynchronous events and to manage the asynchronous transfer of control.
- The ability to work with high-resolution real-time clocks.

These characteristics of real time Java provide the basic tools for the development of control applications. This has enabled the development of a platform for the real time implementation of discrete event control systems based on the Java language. We call this platform the Java PLC. Petri nets are used as a tool for modelling and implementing discrete event systems, and also their programmed implementation: the Sequential Function Charts. The Java PLC should be able to execute PNs or SFCs in real time. To develop the control function, it should be able to communicate with fieldbuses in such a way that it can read the signals of the sensors and write signals to actuators. It should also be able to communicate with the plant operators providing information about the controlled system.

With these objectives, and in order to carry out the development correctly, the Java PLC has been structured in several modules, each one responsible for a function (control, communication, supervision…) and able to exchange information one with another.

The Java PLC comprises several modules (see Fig. 1):

- The control module is in charge of executing the PLC program.
- The input/output module is in charge of communicating with several fieldbuses like Interbus, CANopen or Modbus TCP/IP.
- RMI communication module. This allows communication with other Java virtual machines and for the Web Server to exchange information with the control module.
- Web Server. This module is responsible for providing information about the state of the program and the input / output variables, allowing remote visualisation of the state of the program.

3. The control module

The control module is responsible for executing the PLC programs. These programs can be run using Petri nets in the PNML format or in the Sequential Function Chart language in the PLCopen XML format. The execution in the PLC is carried out in real time as a high-priority task.

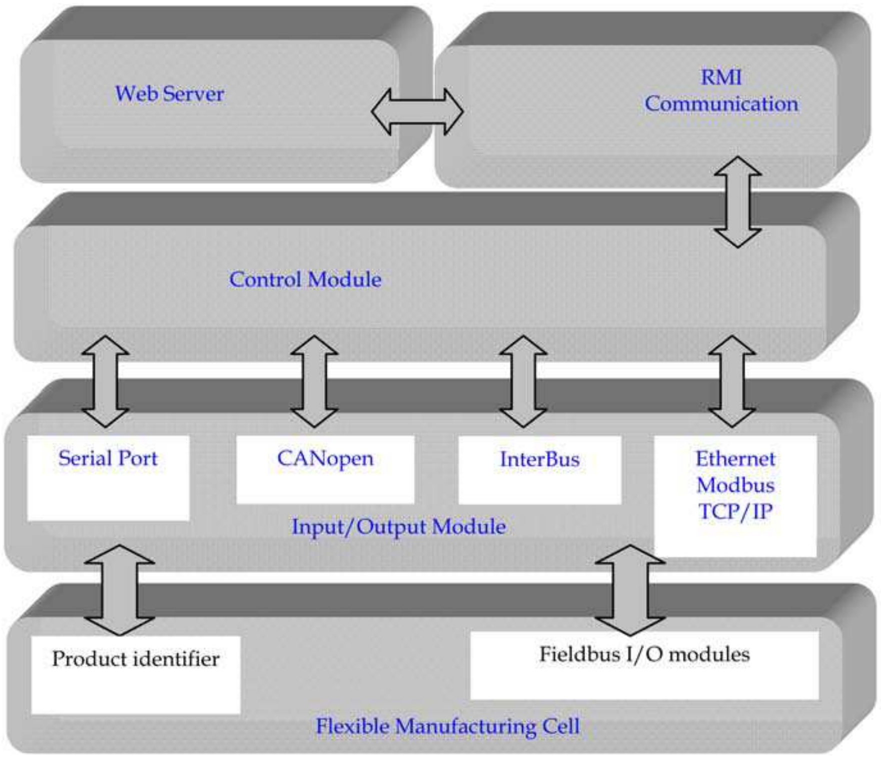


Fig. 1. Java based Programmable Logic Controller.

Currently most industrial PLCs run their programs in an interpreted and centralized manner. The PLC reads the inputs, executes the program (i.e. runs an interpreter of SFCs, also called coordinator in this paper) and writes the outputs.
In the execution of the program it is necessary to determine which transitions can fire, and then fire them so that the state of the SFC (or PN) will evolve. This will also include the actions programmed in the steps. The algorithm to determine which transitions are enabled and can fire is important because it introduces some overhead in the controller execution and the reaction time is affected. In the Java PLC we have implemented several algorithms in which different enabled transition search techniques are developed:

- Brute Force (BF). PN implementation technique.
- Deferred transit evolution model (DTEVM). SFC implementation technique.
- Immediate transit evolution model (ITEVM). SFC implementation technique.
- Static Representing Places (SRP). PN implementation technique.
- Enabled Transitions (ET). PN implementation technique.

In the Brute force algorithm all the transitions are tested for firing. Brute Force algorithms do not try to improve the search of enabled transitions. Works such as (Peng & Zhou 2004) (Uzam & Jones 1996) (Klein, Frey et al. 2003) belong to this implementation class.
The IEC-61131 standard is not very precise in the definition of the SFC execution rules. Different execution models have been proposed to interpret the standard. As with BF, in the

Immediate Transit Evolution Model (ITEVM) algorithm all the transitions are tested for firing (Hellgren, Fabian et al. 2005). However, the Deferred Transit Evolution Model (DTEVM) (Hellgren, Fabian et al. 2005) only performs the testing of the transitions descending from the marked places, improving in this way the Brute Force operation.
In (Lewis 1998) the IEC-61131 standard is interpreted and the following tasks are proposed to run an SFC:

- Determine the set of active steps
- Evaluate all transitions associated with the active steps
- Execute actions with falling edge action flag one last time
- Execute active actions
- Deactivate active steps that precede transition conditions that are true and activate the corresponding succeeding steps
- Update the activity conditions of the actions
- Return to step 1

These tasks are implemented in the DTEVM algorithm. In DTEVM, the transition conditions of all transitions leading from marked places are evaluated first. Then, the transitions found to be fireable are executed one by one. In ITEVM, the transition conditions of all transitions are evaluated one by one. In the case of a transition condition being true, i.e., the corresponding transition being fireable, this transition is fired immediately.
In the Static Representing Places (SRP) algorithm, only the output transitions of some representative marked places are tested (Colom, Silva et al. 1986). Each transition is represented by one of its input places, the Representing Place. The remaining input places are called synchronization places. Only transitions whose Representing places are marked are considered as candidates for firing.

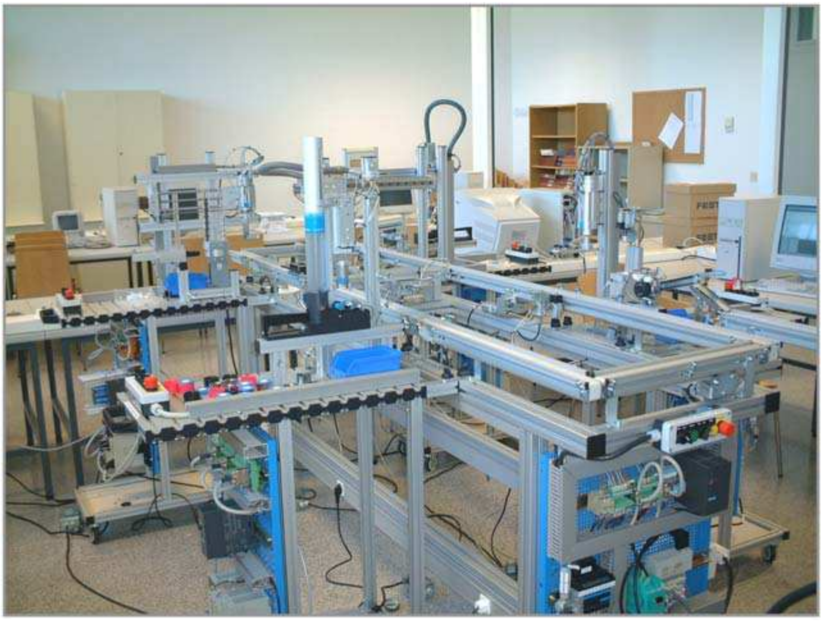


Fig. 2. Flexible manufacturing Cell.

In the Enabled Transitions algorithm, only totally enabled transitions are tested. A characterization of the enabling of transitions, other than marking, is supplied, and only fully enabled transitions are considered. This kind of technique is studied in works such as (Silva & Velilla 1982) (Briz. 1995).
In the implementations developed in the present study, the program loads the net structure from a XML file that is generated by a Petri net editor; thus, the implementation is independent of the net and is an *interpreted* implementation. In the execution of a Petri net, a thread called the "coordinator" is responsible for the firing transitions and updates the state of the net (marking), this being a *centralized* approach. Furthermore, we have introduced centralized techniques into decentralized implementations, thereby creating a new technique called *concurrent coordinators*. The application can run several coordinators simultaneously by executing a sub-net for each subsystem.

4. The flexible manufacturing cell.

The practical application of the ideas presented in this paper were tested using a flexible manufacturing cell installed at the Department of Computer Science and Systems Engineering at the University of Zaragoza, Spain, for research and teaching purposes.
The manufacturing cell carries out a complete production process making various types of pneumatic cylinders. The orders are composed by a rectangular base (white or black) and three cylinders produced in the first production ring, being able previously to select both the base and the cylinders. It also has two storage areas, one intermediate store for the manufactured pneumatic cylinders and the other for orders. The term "flexible" means that the cell can manufacture any component type at any moment.
There are two types of base, white and black, and six types of pneumatic cylinders divided into two groups, cylinders with and without a cap.
The cylinders with a cap are made up of a sleeve with a closed cap. These pieces are already manufactured and require no handling apart from identification and storage. The components without a cap are cylinders simple effect pneumatic cylinders made up of a piston, a recoil spring and a head.
There are three types in both groups: black, red and metallic. The cylinders without cap have two sorts of piston, metallic and plastic. The metallic piston is narrower and shorter than the plastic, and is only included in the black pieces, while the plastic piston is mounted in the red or metallic cylinders. The composition of the cylinders without cap is shown in Fig. 3.
The manufacturing cell is composed of a set of stations for the production and storage of pneumatic cylinders and is divided into two zones (Fig. 4): (a) the production zone (stations 1, 2, 3, and 4, and the transport 1), and (b) the product expedition zone (station 6 and storage station 7, two robots, and the transport 2). Station 5 is an intermediate storage area between the two zones.
Transport 1 (left hand side of Fig. 4) is responsible for moving the pallets in the production zone between the stations and is above the pallets where the operations are carried out. This transport connects stations 1, 2, 3 and 4 which carry out the following operations:

- Station 1: identifies and positions the sleeves for cylinders without cap or those with cap and previously manufactured.
- Station 2: assembles the corresponding piston with the sleeve type and next assembles the coil spring.

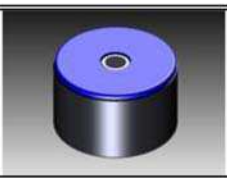
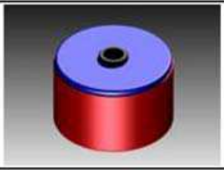
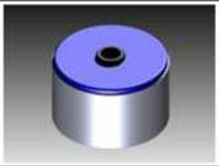
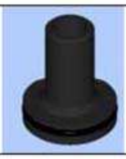
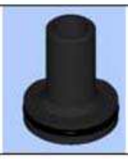
Pneumatic cylinders			
Sleeve	Black	Red	Metallic
Piston			
Coil spring	Standard	Standard	Standard
Cap	Standard	Standard	Standard

Fig. 3. Types of Cylinders.

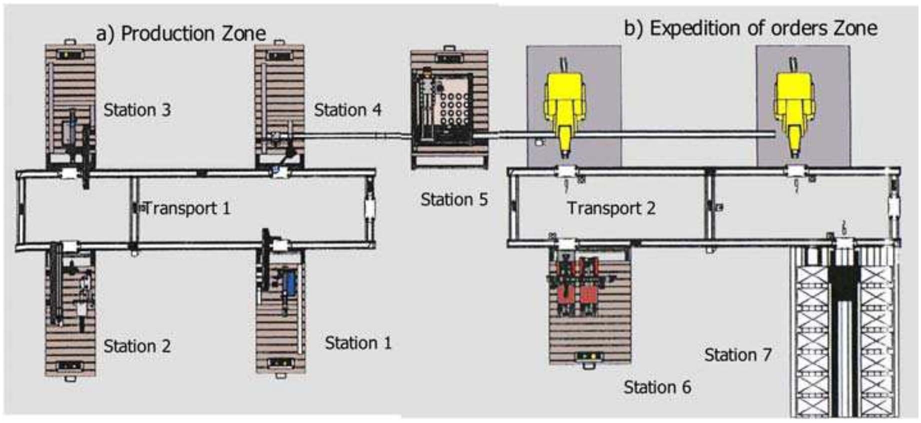


Fig. 4. Flexible manufacturing cell.

- Station 3: attaches the Cap.
- Station 4: checks the components and acts as a link between the production zone and storage area for all the finished components.

Transport 2 (right hand side of Fig. 4) arranges the transport of the cart in the expedition of orders zone between stations 6, 7, 8 and 9 that do the following operations:

- Station 6: is in charge of retrieving the base (black or white) where the three cylinders will be situated.
- Station 7: is a servo controlled storage area that stores the orders in one of its eighty positions.
- Station 8: will grasp the stored cylinders in the station 5 and inserts them in the base according to the order introduced by the user.
- Station 9: removes orders from the cell.

Station 5 links the production and expedition zones, acting as or intermediate storage area of 16 positions. The different pieces (cylinders) are dispatched depending on the production orders and the storage policy of station 5.
The original control system of the cell consisted of a programmable logic controller (PLC) that controlled each station. A real-time industrial net connected all of the PLCs. We then installed a new control system based on a fieldbus. The fieldbus was installed at stations 1, 2, 3, and 4, and at Transport 1 (see Fig. 5). Stations 1 and 4 have Inline modules (Phoenix_Contact 2006), and stations 2, 3 and Transport 1 have Advantys STB modules (Schneider_Electric. 2006).

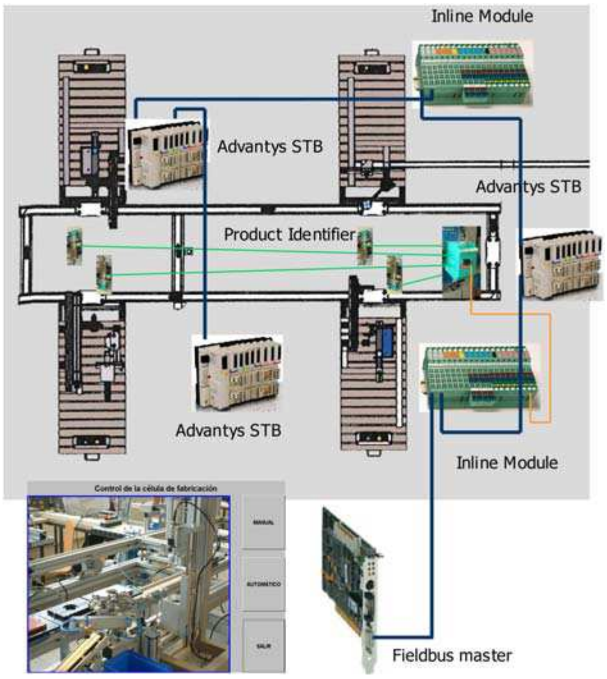


Fig. 5. Fieldbus and Product Identifier

Both Transport 1 and Transport 2 identify the contents of every pallet by means of the read and write heads of the memories attached to the pallets.
At the beginning of an operation at a cell station, to determine the operation that has to be performed, the memory of the pallet that arrives must be read. At the end of the manufacturing operation, the memory of the pallet must be updated.
The product identifier module is a resource shared by all of the stations. Communication with the identifier module is achieved using a serial port inserted in the Inline module of Station 1. Access to the module must be protected from concurrent access.

5. The input/output module

Communication with the control system could be established with the coordinators directly accessing the communications network to read or write each time that they need to interact with the cell. This strategy would not be very efficient because even writing or reading only one station variable would involve a complete reading or writing of the bus. This task has therefore been left to the communications layer. The coordinators access the data they need through monitors that guarantee mutual exclusion in accessing variables. In this way the implementation of the control layer is independent of the system used to communicate physically with the cell.

The communication between the control module and the I/O module takes place via the station monitors. The control module reads the input values from the station monitors, executes the PLC program, and writes the output values in the station monitors. The I/O module reads the input signals of the fieldbus and leaves them in the station monitors. It then collects the output values and sends them via the fieldbus to actuators.

The execution of the periodic task of communication with the fieldbuses is run in a real-time thread at a lower priority than the task of executing the PLC program.

We have been developed classes to allow communication with the fieldbuses Interbus, CANopen and Modbus TCP/IP. The communication with the Interbus and CANopen fieldbuses is carried out through libraries in C++. The call to these libraries is made through JNI (Java Native Interface). Java allows fragments of native code to be incorporated in its programs that is code compiled for a specific platform, generally written in C/C++. JNI is the tool Java has to make use of methods run in other programming languages. For this reason it uses the JNI.

Interbus is a network of distributed sensors and actuators for manufacturing systems and control processes. It is an open system with advanced features and a ring topology. The basic concept of an open bus is to provide information exchange between devices produced by different manufacturers. The information exchanged includes processing data (inputs / outputs) and parameters (configuration data, programs, monitoring data). The information format is defined by means of a standard profile for the devices. Interbus has standard profiles for servomotors, encoders, robot controls, positioning controls, control and operation panels, digital inputs / outputs, analogue inputs / outputs, thermocouples, meters, frequency variators, robots, soldering control, identification systems, etc. The INTERBUS protocol, DIN 19258, is the standard interface for these profiles. This is an open standard for E/S networks in industrial applications.

A task has been developed for communicating with Interbus. This is implemented as a periodic task responsible for reading all the input variables and writing the output variables on the bus. This task is run every 10 ms, which is sufficient given the dynamics of the controlled system. The program uses the driver (native functions written in code C accessed through JNI - Java Native Interface) offered by the manufacturer of the bus PCI master card. In fact the reading and writing is not done directly on the bus but on an image of the memory of this card.

Communication with the CANopen bus is carried out through a periodic task responsible for reading the bus inputs and writing the bus output variables and, as with Interbus, this task run every 10 ms.

Communication with the Modbus TCP/IP protocol in ethernet is made directly in Java, providing communication with the input / output modules. The classes have functions for reading and writing the input/ouputs of the modules using the ModBus TCP/IP protocol. It is possible to change the communication interface of each fieldbus module. Interbus, CANopen, and Industrial Ethernet are supported. If Interbus is used, the bus master card will be the Hilscher CIF50-IBM (Hilscher 2007). If CANopen is used, the master card will be the CIF50-COM card and if Industrial Ethernet is chosen, the Ethernet card of the PC acts as the master. The Interbus topology is a ring and inputs cannot be read or outputs written individually. The reading of inputs and the writing of outputs are managed by the bus master in the same operation. Each station of the cell has a reading/writing head of the pallet memory that is connected to an identifying module of the products (Pepperl&Fuchs IVI-KHD2-4HRX) (Pepperl&fuchs 2006)

Finally, a program also has to be developed for providing communication with the product identifier and thus be able to control production. Sun supplies a Java communications library for applications requiring communication with a device through a serial port (Sun 2006).

6. Control architecture.

One of the main objectives of this work is to define a control architecture. From a hierarchical standpoint, our proposal is about the coordination and the local control layers of flexible manufacturing systems. Rather than distribute the local control of each subsystem of a flexible manufacturing system (manufacturing cells, transports, stores) in various PLCs, we have opted to centralize them in a computer. The system inputs/outputs are distributed over several modules connected by a field bus (see Section 5, above). This approach permits an easy way of developing and debugging new control techniques.

The design and implementation of the local control of subsystems are maintained separately. In this way, separate threads running on the central computer control each subsystem. Another thread is responsible for the coordination function of the cell. Synchronization with the controlled system is achieved using an image memory of the inputs and outputs of the subsystems in the control computer, which is periodically updated through the field bus. That is, in each period, the inputs are read from the bus and the outputs are written to the bus. The cell coordinator updates the memory image. Several monitors protect the memory image because the input-output variables can be shared by more than one local controller, the cell coordinator, and some other threads, such as the Human Machine Interface HMI. Each local controller has its own monitor that protects the variables of the controlled subsystem. Fig. 6 shows the proposed software control architecture.

To synchronize the local controller's execution with the reading/writing of data in the field bus, a semaphore for each controller is used. The local controllers are cyclical but, at the beginning of each cycle, they must wait for a signal from the cell coordinator. It is periodic and, in each period, it sends a signal to all of the semaphores, which permits the execution of a new cycle of all of the controllers. Thus, the coordinator and the local controllers are periodic and have the same period. To implement this schema, the following three levels of fixed priorities are needed:

- High priority. This is reserved for the cell coordinator.
- Medium priority. All of the local controllers have the same priority.
- Low priority. This level is associated with threads that do not have real-time requirements, e.g., the HMI.

The proposed concurrent structure and priorities guarantee that the controllers always execute using updated input data and allow real-time analysis of the thread set. Following a rate monotonic approach, all of the local controller's threads run within their period (the control period) if:

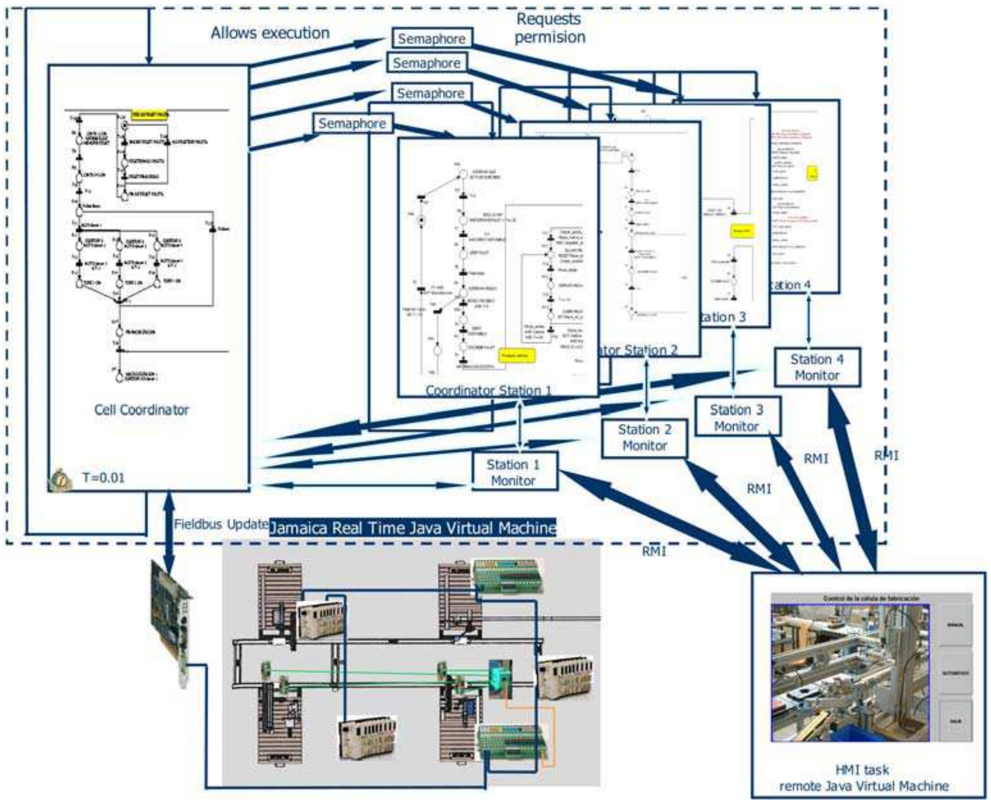


Fig. 6. Control architecture

$$C_{coord} + \sum_i C_i + b \leq T \tag{1}$$

where T is the control period, C_i is the Worst Case Execution Time (WCET) of control thread i, C_{coord} is the WCET of the cell coordinator, and b is the blocking time of the monitors. We have used the immediate ceiling priority protocol, so the blocking time is the largest WCET of all of the services provided by the monitors and called by the lower priority threads, such as the HMI. The expression imposes a restriction on the number and complexity of local controllers running on the computer and on the refresh time of the bus. If the previous condition is fulfilled, the worst-case response time for events in the system can be calculated as:

$$t_r \leq 2T + t_{bus} \tag{2}$$

That is, the response time (t_r) has a bound related to the control period (T) and the readingwriting time of the bus (t_{bus}). An example of the system response time to an incoming event is presented in Fig. 7.

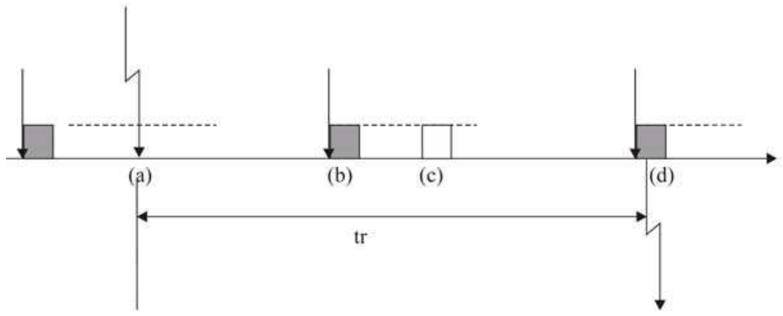


Fig. 7. System Control Time response:
a) An event happens in the system (e.g., a pallet arrives at a station).
b) The event is copied to the memory image of the control system.
c) A local controller reads the event and establishes the reaction as changes on the memory image of outputs.
d) The outputs are established in the system through the fieldbus.

In our application, the control period is 10 ms, sufficient for the dynamics of the controlled system. With Interbus, the read-write time is less than 2 ms; therefore, the response time of the real-time control will be:

$$t_r \leq 2*T + t_{bus} = 2*0.010 + 0.002 = 0.022\ \text{s} \tag{3}$$

In the proposed architecture, only one thread accesses to the communication field bus and establish the refresh frequency; therefore, it is extremely simple to adapt the control application to field buses other than Interbus. At present, it is possible to execute the control over the Interbus, CANopen, and Industrial Ethernet field buses.
In this control architecture, we propose the following:

- The specification of control systems using PN and SFC that allows simulation and system analysis, and the automatic generation of code (see Section 8).
- The use of the JamaicaVM platform to support the concurrence and real-time execution of control programs.
- In addition, we have implemented a graphical task (HMI) that allows real-time visualization of the state of the control system and the PN in Execution (see Fig. 8). In Java Real Time, the graphical task is executed in a remote virtual machine and the communication is made using the *Java Remote Method Invocation System* (RMI) (Sun 2008). In a monitor, the graphical task writes the orders that the operator sends to the control system.

Those aspects are presented below.

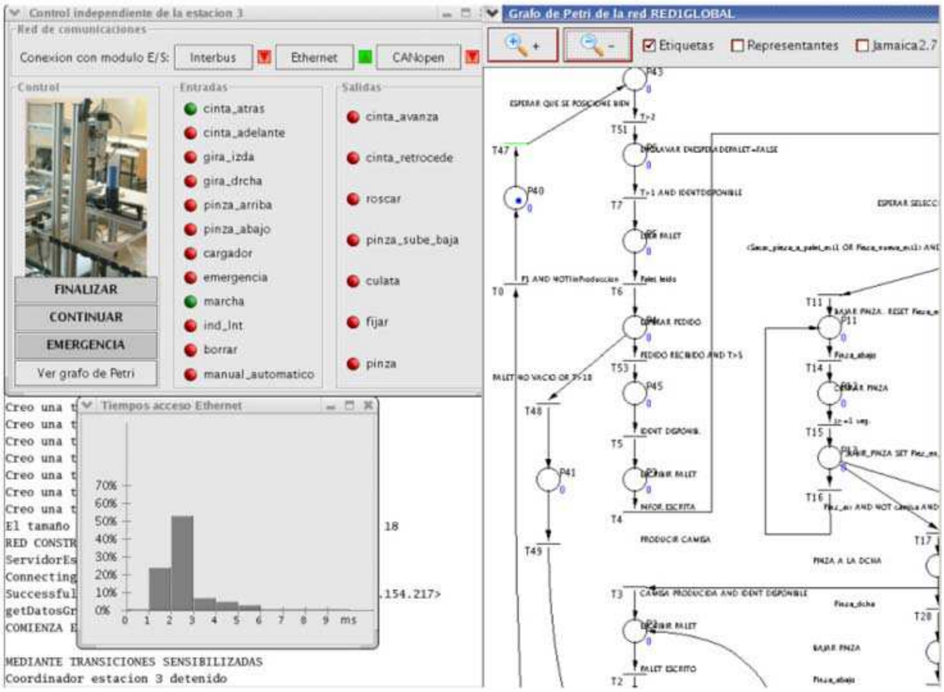
8. The Java implementation

Previous research on deriving Java code from PN specifications (see for example (Conway, Li et al. 2002) or (Buchs, Chachkov et al. 2003)) has focused on prototyping and simulation. Our objective is to generate Java code for Real Time control systems and, thus, we have chosen to adapt classic techniques of PN implementation developed for obtaining efficient and predictable control applications.

Java is an object oriented language. As in any Java application, the code is encapsulated in a series of classes that contain a specific functionality. A set of classes that carry out related tasks are usually grouped in the same package. In the application design it has been attempted to take advantage of the opportunities offered by Java to make a modular and robust application so that future modifications can be made quickly and efficiently.
The packages of the classes implemented are described below. The basic organization is as follows:

- Petri net: a package of classes representing places, transitions and structure of a Petri net.
- SFC: a class that extends the Petri net class and allows the SFC representation.
- Applications: the application package contains the final applications of the project and constitutes the definitive set of classes both for both manual and automatic control of the cell and small manual and automatic control applications of the stations. The main content comprises the programs for the server of the applications and the interfaces required for remote communication with the RMI.
- Input/Output: contains the classes enabling communication with the controlled system through Interbus, CANopen and Ethernet.
- Control: contains the basic classes for the execution of the Petri nets. The execution algorithms are implemented in this package for the Petri nets and SFCs: Enabled Transitions, Representing Places, Brute Force, Deferred Transit and Immediate Transit Evolution Model.
- Centralized control: implements the centralized execution technique of the Petri nets. Among other classes, it contains the coordinator that executes the global Petri net of the cell.
- Decentralized control: implements the decentralized technique for the Petri net control. Among other classes, it contains the coordinators responsible for executing independently and concurrently each of the Petri nets that make up the cell system.
- Stations: monitors that contain the values of the sensors and actuators of each station.
- Serial Port: contains the classes that manage the communication with the product identifier through the PC serial port.

The first steps in the Java implementation were to develop the basic classes that allow representation of a Petri net. In addition, classes were developed that allow connections between the physical environment and the classes responsible for the execution of the Petri net, as well as classes that allow communication between different threads (monitors).
In the implementations developed in the present work, the program loads the net structure from a text file generated by a Petri net editor. Thus, the implementation is independent of the net and, therefore, is an *interpreted* implementation.
In the execution of a Petri net, a thread, the coordinator, makes the firing of transitions and updates the marking, this being a *centralized* approach. Following the approach presented in the previous section, we propose a new PN implementation technique called concurrent coordinators, in which centralized and decentralized ideas are merged. The concept is very simple; the net is decomposed into several subnets following, for example, a functional criterion, and a different coordinator implements each subnet in a centralized way. The communication between the different subnets is made using communication places that are implemented by monitors that use synchronized methods for marking and unmarking.



```
Creo una t
Creo una t
Creo una t
Creo una t
Creo una t
Creo una t
Creo una t
El tamaño                                  18
RED CONSTR
ServidorEs
Connecting
Successful                                 .154.217>
getDatosGr
COMIENZA E
MEDIANTE TRANSICIONES SENSIBILIZADAS
Coordinador estacion 3 detenido
```

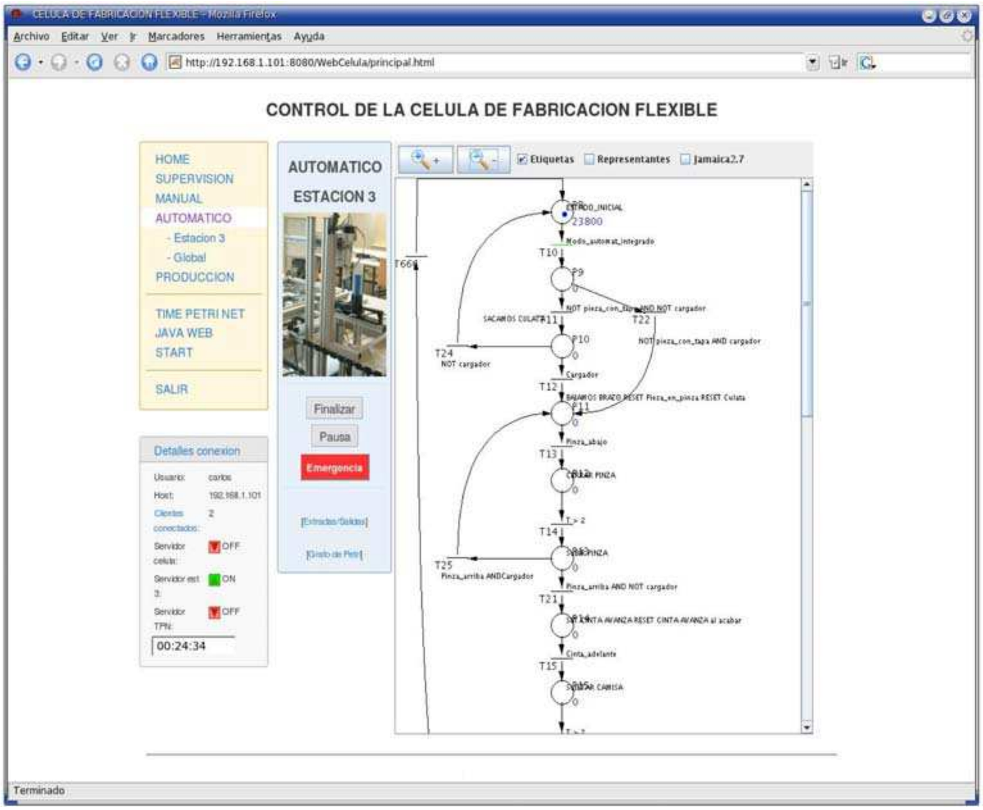
Fig. 8. Human Machine Interface.

In our practical application, the decomposition is straightforward; there are separate subnets for each station. Thus, the local controllers are coordinators that implement these subnets.
For an efficient search of fireable transitions and the update of the data structure that stores them, diverse techniques have been proposed; e.g., enabled transitions, representing and dynamical representing places (Silva 1985) (Colom, Silva et al. 1986) (Villarroel 1990). Also, for the implementation of SFCs: Immediate Transit Evolution Model (ITEVM) (Hellgren, Fabian et al. 2005) algorithm and the Deferred Transit Evolution Model (DTEVM) (Hellgren, Fabian et al. 2005). In the present work, these five techniques were implemented.
As indicated above, the cell coordinator and the local controllers are concurrent threads with real-time requirements. From the perspective of the Real Time Specification of Java, they are RealtimeThreads. Thus, they are scheduled following a static priorities policy with preemption. The cell coordinator is defined as periodic. The Java scheduler is responsible for the periodic activation of that thread. Fig. 9 shows the class hierarchy of our practical application.
The class *coordinator* inherits from the RealtimeThread class of real time Java. Each implementation technique opens a branch in the hierarchy. In the abstract class *Enabledtransitions*, the enabled transitions technique (also called transition driven) is implemented. In the abstract class *representingplaces*, the methods of the technique representing places are implemented. *Dinamicrepresentingplaces* and *staticrepresentingplaces* inherit from the previous one.
The cell coordinator and the local controllers are instances of descendants of those abstract classes. For example, Fig. 9 shows the classes in the control of the manufacturing cell that are

implemented by the enabled transitions method. All of the classes inherit the real-time characteristics.

- java.lang.Thread (implements java.lang.Runnable)
    - javax.realtime.RealtimeThread (implements javax.realtime.Schedulable)
        - control.Coordinator
            - control.EnabledTransitions
                - control.CoordinatorCell
                - control.CoordinatorEst1
                - control.CoordinatorEst2
                - control.CoordinatorEst3
                - control.CoordinatorEst4
            - control.RepresentingPlaces
                - control.StaticRepresentingPlaces
                - control. DynamicepresentingPlaces
            - control.DTEVM
            - control.ITEVM

Fig. 9. Coordinator Class Hierarchy.



Detalles conexion

Usuario: carlos
Host: 192.168.1.101
Clientes conectados: 2
Servidor celula: 🟥 OFF
Servidor est 3: 🟩 ON
Servidor TPN: 🟥 OFF
00:24:34

Fig. 10. Petri Net Visualization in a web browser.

In the coordinator class, we have defined the abstract methods related to the Petri net interpretation:

```
protected abstract void ContinuosAction();
protected abstract void Postaction();
protected abstract void Preaction();
Protected abstract boolean fireCondition ();
```

These methods are application-dependent and must be implemented by the developer. Also we have defined the methods for the implementation of SFCs actions. Table 1 shows the actions that can be programmed in a SFC. In a PLC cycle, the following must be executed:

- Actions which depend on the state of a step: action qualifiers N, L, D, P, P0, P1.
- In the step where is programmed the storage of the stored actions (S, SL, SD, DS) and their cancellation (R).
- The stored actions (S, SL, SD, DS)

The state of the action depends on the action flag Q and the activation flag A. The activation and action flags are activated when the action is activated; the activation flag also remains active in the cycle that turns off the action flag. The action types and qualifiers are the standard ones of the IEC 61131 (ISO/IEC 2001).

Qualifier	Description	#cycles set	
		Q	A
N	Non-stored, executes while step is active.	≥ 1	≥ 2
L	Limited, executes only a limited time while step is active.	≥ 1	≥ 2
D	Delayed, starts executing after the step has been active.	≥ 1	≥ 2
S	Stored, starts executing when the step is activated until reset.	≥ 1	≥ 2
R	Reset stored action.		
SL	Stored and limited	≥ 1	≥ 2
SD	Stored and delayed	≥ 1	≥ 2
DS	Delayed and stored	≥ 1	≥ 2
P	Pulse, executes when the step is activated.	1	2
P1	Pulse, positive flank, executes once when the step is activated.	0	1
P0	Pulse, negative flank, executes once when the step is deactivated.	0	1

Table 1. SFC action qualifiers.

The methods for the implementation of SFCs actions are described in (Piedrafita & Villarroel 2008).

The application of control consists of three separate modules. In the abstract coordinators, the control algorithms are implemented. In the non-abstract descending coordinators, the actions of the places and the conditions of firing of the transitions are implemented. The last module is responsible for the input/output. To communicate with Interbus and CANopen, the C libraries provided by the manufacturer were used. For their use in Java, several functions in JNI were developed. In the case of Ethernet, the communication was developed entirely in Java.

We used the JamaicaVM for our implementation. JamaicaVM's real-time garbage collector does not interrupt application threads; therefore, RealtimeThreads do not have to run in special memory areas, such as LTMemory or ImmortalMemory, which are not under the control of the garbage collector. The garbage collector is pre-empted by any real-time thread.

9. Development platform

We have set up a development environment for the control architecture based on PN and the Java language (see Fig. 11). The basic Java classes were extended to allow the graphical representation and animation of PN that control the system in real time. To describe the PN, we used the *Petri Net Markup Language* (PNML) (Billington, Christensen et al. 2003), which, among other advances, contains graphical information of the net elements (its graphical coordinates) thus allowing the graphical representation of PN and its evolution.

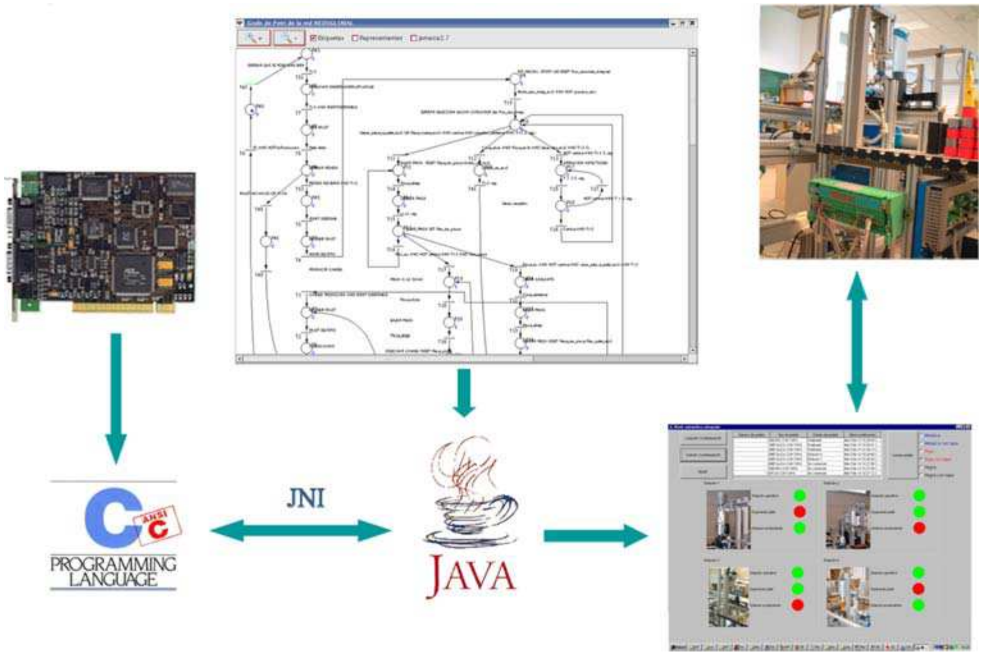


Fig. 11. Development of control aplications

Petri nets were created using Pipe Editor (Bloom, Clark et al.), which allows the Petri net structure to be defined. The transitions predicates and the actions to execute are implemented directly in Java and are associated with the corresponding place and transition objects. The Petri net in PNML format is loaded by the application using parser XML, which creates a tree structure. From that tree, an object of the class Petri Net is created. Later, a thread from the coordinator class is instantiated and the Petri net object is sent as an argument.

We have developed a graphical module that allows the load, representation, and run time animation of the Petri net models. The Java Real time platform does not support the graphical classes of Java. The JamaicaVM platform has a small set of classes that allows the

drawing of points, lines and rectangles only. We decided to keep the graphics section separate from the real time control application. The communication between the two applications is performed using RMI, which allows an object that is executing in a Java virtual machine to call methods of objects or threads running on another virtual machine.
RMI applications have two separate programs: a server and a client. In our case, the server is the real time control application and the remote client is the HMI graphic interface. The full application is composed of the following (see Fig. 12):

- The real time control. In this application, the real-time execution of the Petri net occurs. It acts as a server in the RMI protocol. There is a set of class methods to allow the RMI clients to consult the state of the Petri net in execution.
- The graphical Java client. In this application, the necessary classes for the graphical visualization created in Classic Java are defined. It uses the RMI to consult the data about the PN in execution and allows the visualization.
- Web server. The Web server is programmed in Java and uses the RMI to consult the data of the real-time control.

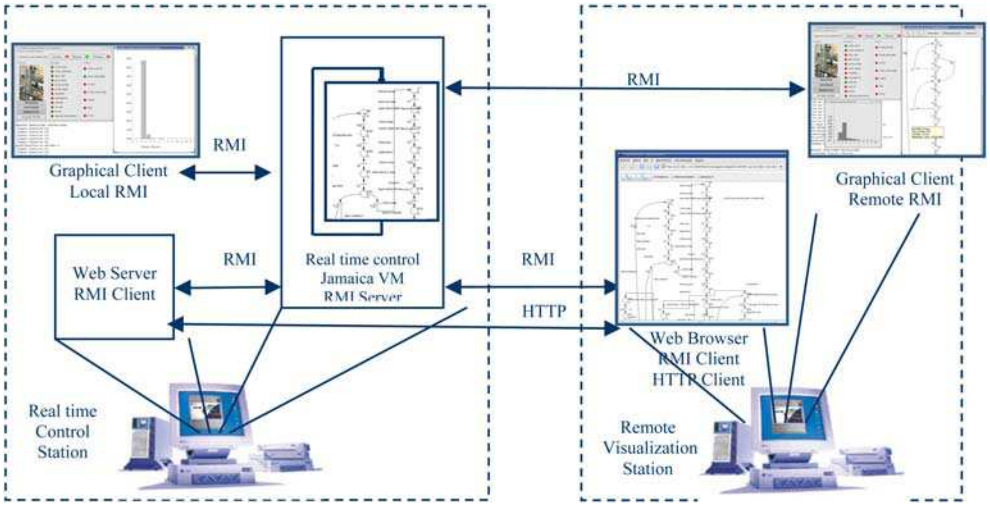


Fig. 12. Graphical interface and communication RMI

When a Web browser connects to the server, the communication is produced by means of the HTTP protocol, but when the real time visualisation of the Petri net execution applet is executed in the browser the communication with the control application takes place in the RMI protocol directly with the server.

10. Conclusions

In this paper, we have proposed a control architecture and a development environment based on Petri Nets, SFCs and the Java language. The architecture is related with the coordination and the local control layers of flexible manufacturing systems. A centralized approach permits the easy development and debugging of new control techniques. One of the mains contributions of this paper is that the proposed architecture allows the verification of the real time constraints of control systems and the use of formal tools as PNs and SFCs.

The hardware is based on a central computer that is connected to the controlled system through a fieldbus. The software architecture is based on several periodic threads. One of the threads, the cell coordinator, plays the role of the coordination layer, and the others are the local controllers of each subsystem. Synchronization with the controlled system is achieved using an image memory of inputs and outputs that is periodically updated through the fieldbus. A set of monitors protects that memory image.
The execution of the software structure over a fixed-priority scheduler allows real-time analysis of the system and a bound for the worst-case response time for events in the system can be established.
Java has been chosen as the implementation language to evaluate them in control systems. An open framework of Java classes for the PN and SFCs implementation has been developed, from the basic classes to the classes that execute the PN, and the classes that allow communication with several fieldbuses. Java lacks the real-time characteristics needed to execute the proposed architecture; therefore, the Real Time Java Specification, which provides the necessary real-time behaviour, has been adopted. The non real-time functionalities, such as the graphic interface, were implemented in classic Java because the development of graphic environments and remote web clients is easier. In this way the controlled system can be supervised remotely through the web.
The coordination function and the local controllers have been specified using Petri nets that allow simulation, systems analysis, and automatic code generation. A new PN implementation technique called concurrent coordinators, involving the use of techniques centralized in decentralized implementations has been developed. This technique is perfectly suited to the control architecture.
All of the techniques and technologies presented in this paper have been evaluated in a practical application, the control of a flexible manufacturing cell composed of manufacturing stations, transports, robots, and stores. The control system of the cell is currently running without problems. This work concludes that the Real Time Java Specification for the development of control systems based (or not) on PN is entirely valid.
The work sets the basis for more detailed research in the fields of programmed implementation of PN and SFC, languages, and execution and real-time platforms. Future work will examine the following:

- The migration to real time operative systems.
- Simultaneous control in several fieldbuses.
- Discrete event control systems decentralized and distributed implementations.

11. References

Aicas, G. (2007). JamaicaVM Realtime Java Technology. http://www.aicas.com/.

Billington, J., S. Christensen, et al. (2003). "The Petri net markup language: Concepts, technology, and tools." Lecture Notes in Computer Science: 483-506.

Bloom, J., C. Clark, et al. PIPE Platform Independent Petri Net Editor, Technischer Bericht, Department of Computing, Imperial College London, 2003.

Bollella, G. and J. Gosling (2000.). "The Real-time Specification for Java." Computer 33(6): 47-54.

Briz., J. L. (1995). "Técnicas de implementación de redes de Petri. ." PhD thesis, Univ. Zaragoza.

Buchs, D., S. Chachkov, et al. (2003). "Modelling a secure, mobile, and transactional system with CO-OPN." Application of Concurrency to System Design, 2003. Proceedings. Third International Conference on: 82-91.

Colom, J. M., M. Silva, et al. (1986). "On software implementation of Petri nets and colored Petri nets using high-level concurrent languages." Seventh European Workshop on applications and theory of Petri nets, Oxford, July 86: 207-241.

Conway, C., C. H. Li, et al. (2002). Pencil: A Petri net specification language for Java. Math. Dept., Macquarie Univ., Sydney, Australia.

García, F. J. and J. L. Villarroel (1996). "Modelling and Ada Implementation of Real-Time Systems using Time Petri Nets." Proc. of the 21st IFAC/IFIP Workshop on Real-Time Programming. Gramado-RS, Brazil. November.

Hellgren, A., M. Fabian, et al. (2005). "On the execution of sequential function charts." Control Engineering Practice 13(10): 1283-1293.

Hilscher. (2007). "PC cards. http://www.hilscher.com." ISO/IEC (2001). "International standard IEC 61131-3 (2nd ed.). Programmable logic controllers—Part 3. ISO/IEC (final draft). ."

Klein, S., G. Frey, et al. (2003). PLC Programming with Signal Interpreted Petri Nets. ICATPN 2003, Eindhoven, Srpinger Verlag.

Lewis, R. W. (1998). "Programming industrial control systems using IEC 1131-3." IEEE control engineering series 50. Open, P. L. C. (2005). XML Formats for IEC 61131-3.

Peng, S. S. and M. C. Zhou (2004). "Ladder diagram and Petri-net-based discrete-event control design methods." Systems, Man and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 34(4): 523 – 531.

Pepperl&fuchs. (2006). "IVI-KHD2-4HRX DataSheet. http://www.pepperl-fuchs.com." 2006.

Phoenix_Contact. (2006). "Inline Modules. www.phoenixcontact.com. ."

Piedrafita, R. and J. L. Villarroel (2006). Petri Nets and Java. Real-Time Control of a flexible manufacturing cell. Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on: 1246-1253.

Piedrafita, R. and J. L. Villarroel (2008). Evaluation of Sequential Function Charts Execution Techniques. The Active Steps Algorithm. Emerging Technologies and Factory Automation,. IEEE Conference on. Hamburg, Germany.

Schneider_Electric. (2006). "Advantys STB. http://www.telemecanique.com."

Silva, M. (1985). Las Redes de Petri en la Automatica y la Informatica. Editorial AC, Madrid, 1985, Spanish.

Silva, M. and S. Velilla (1982). "Programmable logic controllers and Petri nets: A comparative study." IFAC/IFIP Symposium on Software for Computer Control, Madrid, Spain, October 1982: 83–88.

Sun. (2006). "Java serial communication http://java.sun.com/products/javacomm/." 2006.

Sun. (2008). "Java Remote Method Invocation.http://java.sun.com/."

Uzam, M. and A. H. Jones (1996). Towards a Unified Methodology for Converting Coloured Petri Net Controllers into Ladder Logic Using TPLL: Part I - Methodology. International Workshop on Discrete Event Systems - WODES'96. Edinburgh, UK: 178 - 183.

Villarroel, J. L. (1990). Integración Informática del Control de Sistemas Flexibles de Fabricación. Tesis Doctoral. Ingeniería Eléctrica e Informática, Universidad de Zaragoza.