A Client-Server Based Real-Time Control Tool for Complex Distributed Systems

Axenie Cristian
Computer Science Faculty at Dunarea de Jos University
Y Building, Stiintei street, Galati, Romania
haustiq_86@yahoo.com
Pascalin Andrei, Maftei Florentina, Perjeru Marius
Zanoschi Aurelian, Stancu Alexandru
Computer Science Faculty at Dunarea de Jos University
Y Building, Stiintei street, Galati, Romania
andrei_i_pascalin,aurelianz2004,jokerush,flor3nt1na,
stancu_alex@yahoo.com

Abstract

Today's trends in Control Engineering and Computer Engineering are blending gradually into a slightly challenging area, the improvement of Real-Time implementations. Applications should timely deliver synchronized data-sets, minimize latency and jitter in their response and by using an efficient scheduling algorithm to be able to determine various system resources to meet optimal allocation and their performance specifications. As Real-Time systems are increasing their complexity and safety-critical aspects, there is a new trend to raise software development level of abstraction from programming languages to concrete standard models. By accepting the challenge of developing a Real Time application we had been able to design and build a slightly alternative method to implement control algorithms in industrial distributed systems.
Keywords: Real-Time, Linux, RTAI, threads, Hard Real-Time, Soft Real-Time, C++, COMEDI , Java, TCP/IP, Grafcet, distributed systems.

1 Introduction. Architecture overview.

Better system reliability and easier test routines, weight, space, wiring and power requirements reduced, decentralization of control and efficient maintenance. These are the main advantages of using the networked control system pattern in designing efficient solutions for the industry. Following, an architecture overview is given to get familiarized with the specific implementation details of the developed solution. The main application is designed on the client-server pattern because we are handling a homogeneous distributed system from the OS point of view, because each node of the system uses a similar OS but different programming languages for implementation. The main, server node or the controller is a Linux OS based computer configured with the Real-Time patch (the RTAI modules) and it offers the possibility to implement the control algorithm, due to it's direct connection with the controlled industrial process. The second important node in the distributed system is the data and command server, designed as a Linux OS based computer that hosts a second part of the application using a data server to modify the parameters of the control algorithm found in the running application on the main node (identified as the main application server). To complete the distributed architecture we included in our design some clients, Linux OS based computers that are only able to access logs, data charts from the process and other parameters needed for monitoring. The design pattern for developing the application uses a specific architecture, consisting in several compact layers gathered in a "stack" like dependence, based on specific command/control and data flows. The base concrete level is the physical

level of the DAQ card that ensures direct communication with the controlled process. The raw or formatted data gathered from the process are sent to the next level application running on the Linux OS based server. The interface between the DAQ card and the application is ensured by a set of drivers from the Comedi/Comedilib library and the OS specific functions, to maintain and enhance the Real-Time features used in the data acquisition and processing. The second concrete level introduces the C++ application specific. It reveals an OOP design pattern used to enhance productivity, flexibility and compatibility in adjacent classes communication developed inside the application. The center of the C++ application is based on the common use of the RTAI specific methods for Real-Time tasks in data handling and commands and of course the tool for implementing the control algorithm. To maintain the flexibility and portability of the application the Grafcet tool was chosen because it's real advantages in dividing the process in sub-processes. Using this feature we are able to choose a soft Real-Time or Hard Real-Time implementation for the specific events in the process's evolution. Using the Grafcet we would be able to easily modify the specific controlled elements regarding the process components, implementing a good parallel execution tool by describing the evolution of the process and it's actions offering a proven independence over the technology. The third concrete level is the communication level between the C++ application and the Java application running on the data and command server(in a client(Java)-sever(C++) model). The communication channel uses a TCP/IP protocol that offers an efficient and safe method to maintain parameters interchange and command communication. Between the two software applications we managed to create a certain compatibility pattern to easily ensure the communication, so we designed a similar class architecture using a data class, a Grafcet class, a control server class, a sensor class, an actuator class and a system class that all are encapsuling functionality. The last two concrete levels due to their resemblance in implementation are treated together so the Java control and data server application in responsible to send control commands to the Grafcet class and also send / receive data from the process through the C++ application and via TCP/IP, it interacts as the server with the Java clients. The client nodes are based on a data client class that is designed as a Java GUI to permanently interact with the user that monitors the process at a certain time.
We must emphasize the difference between the concrete and the abstract architectural levels, to the extent that some details from the physical DAQ level and the communication specific are lost in the abstraction process. Hence, these two aspects aren't presented as architectural levels but only as interface components in the main application levels.The concrete components desciption was considered just to offer a full depiction of all the aspects involved in this approach.
The developed solution was designed to meet certain accessibility features , including portability, the use of free or low-cost auxiliary software (OS,SDK platforms) and practical use in a large area of industrial control processes.

2 Implementation details and specific functionality.

The developed application is built on a layered modular pattern so that it can support enhancements regarding method implementation. This extensibility feature is very important due to the application main purpose of decentralized control, to the extent that it has to be compliant to a whole range of industrial processes. This extensibility is implemented by a specific mechanism offered by the flexible OOP design pattern and the sequential Grafcet tool, for implementing the control algorithm commands and supporting data flow through the application layers. Next the synthetic and functional description of the architecture is given in figure 1 to emphasize the main clusters in this distributed control problem.

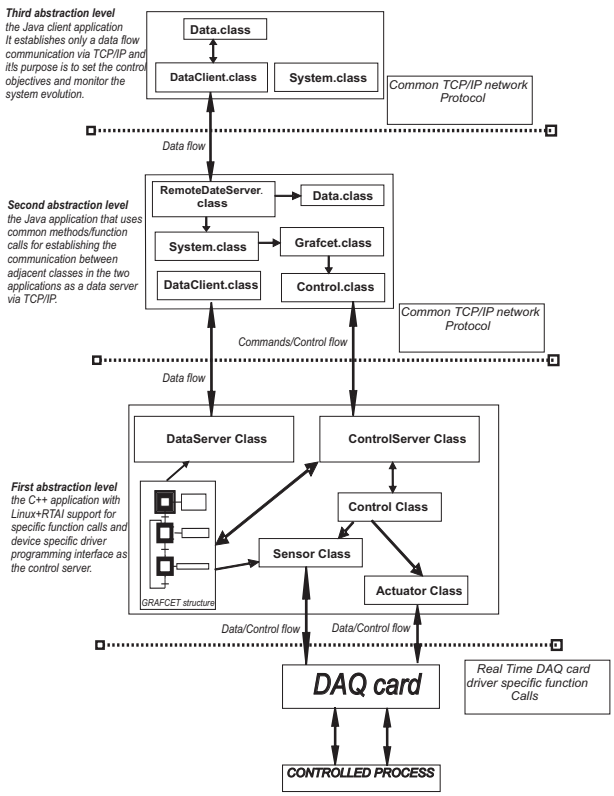

FIGURE 1: *The main application architecture*

2.1 Base level presentation. The Real-Time Linux C++ application.

The first abstraction level is the C++ application under Linux+RTAI that is identified as the control server. To obtain the optimal solution in resource utilization and interfacing under Linux, the Comedi / Comedilib drivers were used. To enhance productivity this variety of drivers was implemented as dynamically loadable kernel modules. So the process scheduler and the virtual file system kernel subsystems virtually are presenting a common application programming interface used to initialize, configure and calibrate Comedi specific devices. The most important element at this level is probably the Kcomedilib module dedicated to the Real-Time applications.

This level's main architecture is now presented in figure 2 and some specifications are given to make a proper mental model for the application specific.

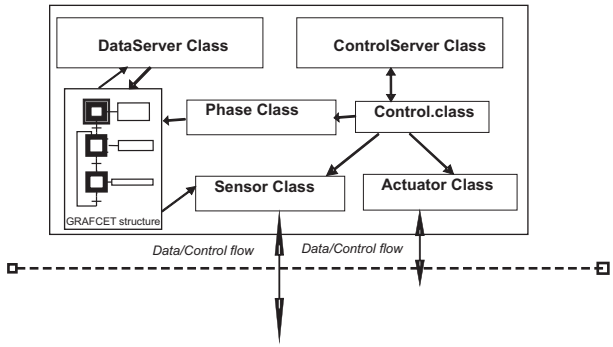

FIGURE 2: *The C++ application architecture*

The communication between the classes is established via public and protected methods using a data/control flow for transmitting parameters and also other OOP specific mechanisms for class communication. It is important to remind that the two applications running are built on a symmetric pattern so that similar tools are used for transmitting data and commands via the TCP/IP network.

The base class for the C++ class hierarchy is the Thread class that encapsulates specific pthread and pthread_mutex_t OS data structure and implements methods that will be overridden in the derived classes. This class maintains permanent communication with the OS.

The Thread class has multiple realizations that are used to abstract, in a down to top manner the details of the hardware architecture.

The main purpose of the C++ application is to issue direct commands to the actuator, so it is identified as the control server in the application, but also it is responsible to retrieve data from the sensors and implement decision making through the specific control algorithm. All that is dpossible by using specific means to maintain a bi-directional communication in Real-Time with the DAQ card.

The class hierarchy abstracts the physical elements comprised in the control system. So we designed a DataServer class to continuously interchange data, transmitting and restoring, during the phase execution with a potential client node connected to the homologous Java application.

This communication is built on specific method calls such solveRequests(), transferData(), emptyBuffer() or debug(). Another class with the same functionality is the ControlServer class that offers an interface to the upper layers (such the Java server or a Java client) to easily issue commands to the control system hardware like setControl(), serveRequest(), getIPControl() or connect() .

The two presented classes are implementing the communication functionality between the two synchro-

nized running applications and to the extent that are modifying the control system hardware status. The concrete tool for implementing control algorithms is the Grafcet so we had to abstract all it's specific components so that we can benefit of it's main advantages. In this context we managed to design a Stage class and a Transition class.
The two classes are implementing the specific Grafcet elements such as stages, phases, action, cycles, and transitions. Methods like start(), stop(), execute() are used to activate and so control the execution of the stages and the associate phases and transitions. Another kind of methods are used to maintain a consistent and common interface such as add(), reset(), setName(), getName() that give specific access to describe the sequential or parallel execution. Hence, building the Phase class gave us the possibility to suitable group all possible events in the system evolution in phases that are later executed.
Other Grafcet specific elements were abstracted in classes like the Action class and Cycle class that are fully described by specific methods like start(), stop(), execute(), setPeriod(), initialize(), in order to set the specific actions to execute, if some condition variables in the transition evolution are met and to iterate some specific cycles.
These elements are very important because these are the main mean to implement the hard Real-Time blending OS specific pthread methods to the RTAI RT_TASK specific methods.
The phases are composed of stages related through transitions. The Grafcet tool describes the process evolution mapping in a coherent way the control objectives by easy adapting to the process and maintaining technological independence.
The C++ application relies on the Grafcet tool to support a proper process separation in smaller processes that are easier to implement on a single processor architecture with a proper task management and parallelism.
The two classes responsible with the direct communication with the controlled process are abstracted as Sensor class and Actuator class and they're role is obvious. But it is interesting to study they're specific. We can start the description only based on the Comedi specific in/out methods. The Sensor class and Actuator class specific methods like enable(), control(), configure(), get(), set() are containing Comedi function calls.
The drivers from Comedi are organizing the hardware in a specific hierarchy starting with the channel as the low-level component responsible of the properties of a single data transfer channel; sub-device that is a set of identically functional channels implemented on the same interface and the device that is a set of sub-devices physically implemented through a generic interface. Each supported device has his own set of specific parameters and Comedi validates/calibrates hardware types by specific methods like comedi_dio_config().
The two classes are implemented in a slightly practical way with the possibility of supporting both the acquisition or sending of analog and digital signals through the Comedi comedi _data _read()and comedi _data _write() methods.
Hence, we have an AnalogSensor class that inherits all the fields of the Sensor class, like the state, frequency or value, but it also introduces his own fields like activeChannel or voltage range. The most important methods implemented in this class is the physicalDataToVolt() method that returns the converted data from the input into volts and the get() method, that reads a sample from the DAQ input channel and then returns an analogue value in volts using also the conversion method.
Obviously we also designed a DigitalSensor class that is generating objects containing the identifier for the acquisition channel, a pointer to the Comedi device and having an interface composed of a function call to comedi_dio_config() and a get() method that returns a digital value from the digital sensor object.
The Actuator class implements a similar functionality to both analog and digital domains. There we have the AnalogActuator class and the DigitalActuator class that are inheriting the Actuator class and are implementing also a field containing a measure of the maximum voltage. We now introduce a voltsToPhysicalData() method used to convert voltage into data trough an implementation specific formula given by the measurement scale and extreme (minimum and maximum) values. The active part of these classes is the set() method that sends an analogue/digital value to one of the DAQ's analog/digital output channels through the Comedi specific comedi_data_write() method.
The connecting class of the C++ architecture is the Control class because it mediates access to the phase execution and to the sensors and actuators specific data. This class objects are encapsuling data like the phase number, the parameters tables, the phases, the sensors and actuators and also a RT_TASK object to implement Real-Time in the derived classes.
Methods like executePhase(), setParameters(), getParameters(), getSensors(), setActuators(), setPhases() and stop() are used to fully extend it's power to control the desired evolution in the control system actions. It's Real-Time character is used and extended to it's dependent/derived classes by a specific mechanism introduced by the RTAI patch installed over the Linux kernel and configured to use

the class facilities (mutexes and condition variables introduced within the POSIX 1003.1c standard).
Hence, we managed to use and develop the architecture facilities stored in the RTAI_SCHED_MODULE like the rt_task_init(), rt_task_make_periodic(), rt_task_make_periodic() and
rt_task_task_delete() or in the LXRT_MODULE like rt_make_hard_real_time() and rt_make_soft_real_time() to fully aggregate the use of Real-Time especially in the Cycle class.
There we have designed methods that initialize the Real-Time agent and creates a Real-Time task with multiple facilities to set/return the execution period. This was done by evaluating the finalization conditions of the execution and the management of the loop control tasks, that is a specific behavior for the Real-Time applications.
Next to easily pass the description to the Java data and control server application we have to focus on the communication problem, emphasizing important aspects and implementation details in using the TCP/IP network protocol.

2.2 The communication interface between levels. Describing TCP/IP specific use.

TCP/IP was chosen to fulfill the communication interface between the C++ and the Java applications. The Linux network system provides two transport protocols with differing communication models and quality of service. We chose the reliable streamed TCP/IP over the unreliable message based UDP. The application communication activities take place in the background and are supported over a specific range of methods that are used to transfer parameters to the other entities of the application and other data regarding the process evolution.
The main application is comprised of a Real-Time control server (the C++ application), a control and data server (the main Java application, that is a client in the direct relation with the C++ application) and a variable number of clients (simple Java GUI applications, clients in the direct relation with the Java control and data server) that completes the distributed architecture. The C++ and Java entities communicate through unidimensional arrays. The two applications are built on compatible class structure so that the communication is established on compatible modules in the level. Data are formatted before sending it to the C++ control server that is responsible of executing the control algorithm with the received parameters. There is the possibility that Java clients connect they're self to the Java control and data server but only with limited access, that gives them the possibility only to gather information about the system evolution through data/message logs and graphs. To implement the communication objectives the C++ application uses the DataServer and ControlServer classes and the client-server connection is based on sockets, using an IP address and a port number. The ControlServer class communicates bi-directional with the Control class in the Java server application to transfer the parameters and the commands for the execution of the control algorithm. Data is stored in matrices that can be full or empty and we can insert/extract data. This feature is offered by the DataServer C++ class that implements specific formatting.
The Java control and data server is using the TCP/IP interface through the Control class and the DataClient class to communicate with the compatible classes in the C++ application and the RemoteDataServer class to communicate with the Java clients. The Control class contains specific methods to initialize, connect and disconnect clients and it's main purpose is to mediate the communication between the Real-Time C++ application, that communicates directly with the process and the Java clients, that are executing a continuous monitoring of the process. The RemoteDataServer class is responsible for the efficient data transfer between the C++ classes and the Java client classes. It also serves the Grafcet passing requests, parameters transfer and it processes event notifications from the process evolution.

2.3 The second architectural level. The control and data server Java application.

The second abstraction layer is the Java data and control server level that mediates and maintains an efficient communication between the Real-Time component of the main architecture and the passive component that only accesses data from the process and has also a local access for modifying the system structure. The main conceptual architecture of this level is now presented. Hence, we designed a Control class that is the client side for the C++ control server, communicating directly with the C++ application ControlServer class. Another convention, that we considered, provides a clear look on the specific relations between the main components of the application. So the server is identified with the C++ application and it's unique client is the Java control and data server that simultaneously is identified as the local server for the Java clients. Hence, the Java control and data server application plays a double

role in the main architecture, as a client and it the same time as a server, this is an important feature offered by the OOP paradigm specific mechanisms.
The Control class is highly dependent of the Data class that is used to store data arrays to easy restore it by the data clients. The Control class is responsible to command the phase execution or to access the sensor's and actuator's specific data through methods like Sensor.get(), Actuator.set(), executePhase(). It also depends on the Grafcet class that stores the Grafcet structure parameter values of the process and it offers functionality to dynamically modify phases and change the process parameters through Grafcet.setControl(), Grafcet.forward(), Grafcet.current(), Grafcet.write() or Grafcet.end(). The Grafcet class is higly dependent on an virtual Action class that offers the mechanisms to maintain the structure's flexibility and dynamics. It is lately extended in classes like the Phase class, that handles the phase specific execution, commands and I/O operations, the Parameters class, that is implementing methods to set and get parameters and write/read them from the data buffer to/from dedicated files.
The execution specific methods behind the Action class are then implemented in the toDO class that stores Action objects and brings up a method interface to modify the number of iterations, or the control operations done at each execution step. The base architecture of this level is depicted here, in figure 3, to emphasize the inheritance relations and dependencies.

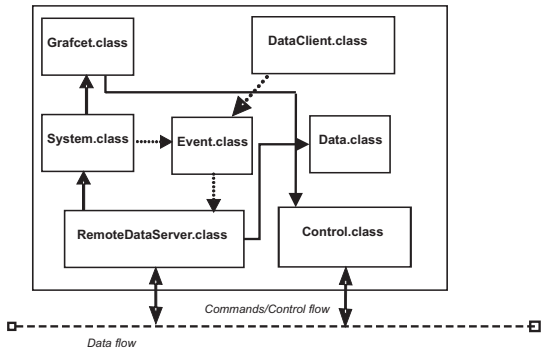

FIGURE 3: *The Java control and data server application*

As we can see, in figure 3, there is a System class that permits the execution of the implemented algorithm and it can mediate storing/retrieving of data regarding the controlled process. All operations in the system control process are done using signals implemented in the virtual Event class supporting both the System's specific methods like System.setGrafcet(), System.setEvent(), System.start(), System.stop(), System.save() or System.restore() and also specific communication actions like notifying and returning the active port on which will occur an event or solving client requests to set/get data or set/get port.
The latest specifications are implementing and describing functionality ofor the RemoteDataServer class, that mainly serves Grafcet passing requests and data or event handling. Another event dependent class is the DataClient class that receives data from the C++ control server application and stores it internally in a buffer, with the facility of notifying if there are new data across one event. It implements methods like setServer() or getServer(), setEvent() or getEvent(), setBuffer() or getBuffer(), connect() or run() and finally getPort() and start(). The purpose of these methods is obvious and there is an extensibility feature to adjust they're behavior.
Following a simple example of execution is presented to understand the main pattern to design process specific application behavior. Hence, we instantiate a Control and a System object and then by specific Control.setControl() , start() and stop() methods we describe the execution steps.
We call the run() method, overridden in mostly all active classes, because it represents the specific Thread class mechanism to implement critical sections for executing Real-Time tasks. Then through a serializable interface to Grafcet class we create two Action objects that shall be executed in execute() and ended with end(). On the other hand we use the Control class through a serializable interface to set/get parameters and execute and finalize phases. Hence, when we first call the start() method a thread is launched that will make the Grafcet object to reference actions and make them execute sequentially. Next is a depiction of two possible implementations for the Grafcet, a serial approach and a parallel one. The serial approach presented in figure 4 presents it's advantages in proper understanding systems's dynamics and evolution.

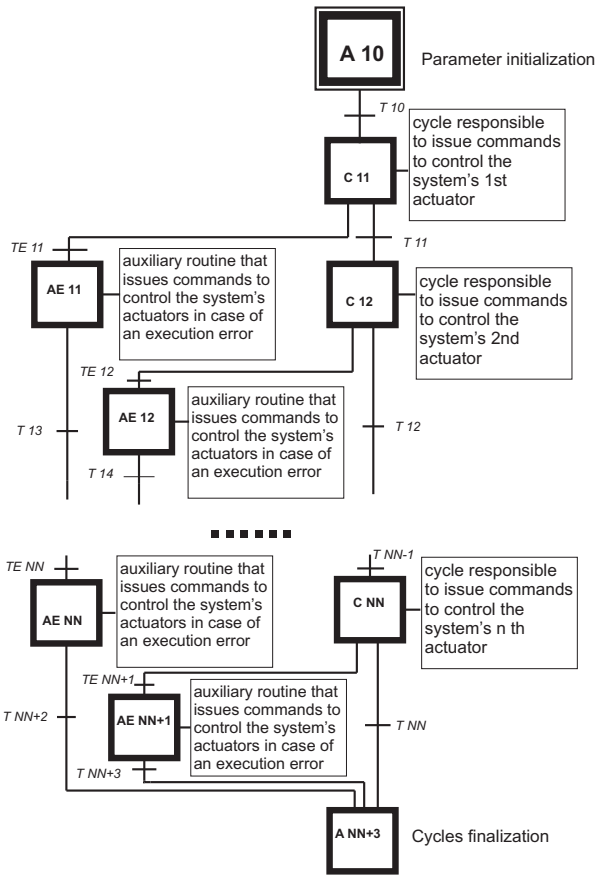

FIGURE 4: *The serial phase execution Grafcet concept structure*

On the other hand the parallel Grafcet approach presented in figure 5 is suitable for reaching optimal concurrent execution with the possibility to divide a main process in multiple more simple sub-processes, and an easy debug routine implementation.

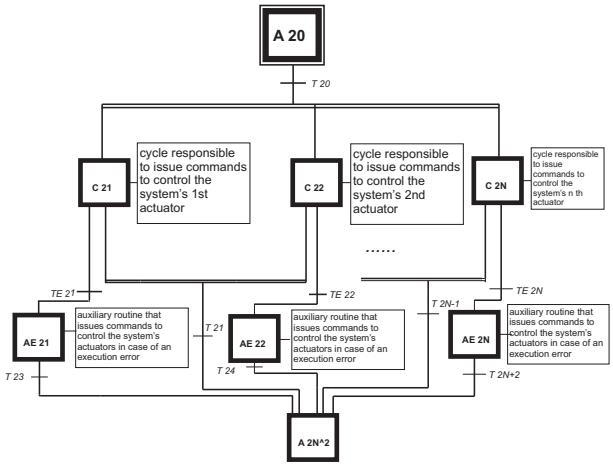

FIGURE 5: *The parallel phase execution Grafcet concept structure*

The System class controls the Grafcet execution through Event objects executing asynchronous actions. Then, finally, the RemoteDataServer permits to the Java client applications to access a Data object and a System object to consult the results and the notifications. So that we can summarize, we have a System object that store/restores a Grafcet object containing the Action structure of the system that accesses the Control. On the other hand, phase execution data is stored in a DataClient object and if there are requests, data and event notifications are sent to a Java client application via the specific RemoteDataServer methods.

2.4 The third application level. The Java client architecture.

Being the highest level of the application it's degree of low level details abstraction is high, so that an user will be absolved of any knowledge of implementation and specific structure of the lower levels in the architecture. Hence, it's architecture, shown in figure 6, is quite simple .

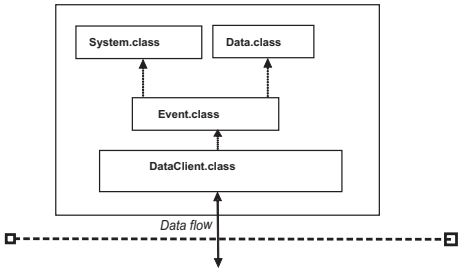

FIGURE 6: *The Java client application architecture*

Based on the direct communication with the data and control Java server through a DataClient class it works based on request and notification signals from the Event objects. So, it can implement an efficient monitoring system through DataClient.serveRequests() and DataClient.serveNotifications() methods and set through a limited access the structural details of the system. The System class is a local copy of the control and data server Java application System class. It is an active class that uses start(), stop() and run() methods to activate/stop the execution and maintain continuous state information about the process. Finally the Data class stores the data transferred via the TCP/IP network from the control and data Java server.

Throughout the figures it had been used a specific way to mark data flow and command/control flow that's because these entities are representing in fact

the numerical values of some parameters or the parameter passing actions for some methods. They're main purpose was to emphasize the dependency between the adjacent classes in one application or between the three application components.

3 Conclusions and future work.

Given the developed architecture, we shall concentrate on extending the application by adding new characteristics that will enhance functionality of the abstracted elements. We intend to obtain a flexible and efficient solution that adapts itself to the control specifications and the needs of an industrial process. It is proposed to raise the steak by building a true platform based on this application. So our first concern is to develop a specific Linux kernel equipped with al the necessary modules for supporting all kinds of DAQ devices and especially to support the Real-Time facilities, used in the application execution, all that in a consistent conceptual architected kernel and a flexible concrete kernel.
At this moment with the first version of the application we managed to control slowly variable systems but also systems that require Real-Time facilities in they're control. Each application level will maintain it's initial purpose but the component classes will be extended by adding new more efficient methods so that it will ensure an easy modification of the Grafcet structure for the controlled systems, efficient and safe parameter transfer between associate methods and of course more flexible inter-application communication features.
Another approach for the future work would be in the fault tolerant control area with the possibility of implementing specific mechanisms to detect and isolate the faults occurred in complex systems, as specific actions for the phase execution in the Grafcet algorithm, both on the valid transitions and also error transitions.
The main advantages of this application are residing in the intelligent use of the RTAI function calls and data structures combined with the Comedi/ Comedilib drivers, that are most likely extensible and of course the flexibility and diversity of the OS specific features to support Real-Time.
Another aspect that emphasizes the efficiency of the developed application resides in the facility to easily change the control strategy and monitor the process evolution in Hard Real-Time in a distributed system architecture.
Finally we must emphasize that the costs were minimized by using open source, free software like the Linux kernel (using RedHat9 and Ubuntu6.10 distros), the C++ and Java compilers for Linux, the RTAI patch and the Comedi drivers, this supports our focus in the enhancement of the current application version.

References

[1] ,*K Computing. Embedded and Real-Time Linux development*

[2] ,*Kevin Dankwardt. What is real time? and benchmarks on real time Linux - Part 1,2,3*

[3] ,*Distributed Operating Systems, Doreen L. Galli, 2000, Springer*

[4] ,*Burns A, Wellings A:Real-Time Systems ans Programming Languages,AddWesley,California 1996*

[5] ,*Real-Time Control Systems, K.E.Arzen course Automatic Control Department IT Lund*

[6] ,*Charles Curley. Open source software for Real-Time solutions. Linux Journal*

[7] ,*Dipartimento di Ingegneria Aerospaziale Politecnico di Milano, RTAI Development www.aero.polimi.it/rtai/about/index.htm*

[8] ,*Marcus Goncalves. Is Real-Time Linux for real? Technical report, Real-Time Magazine*

[9] ,*James Norton and Clark Roundy. Real-Time Linux - where is it now? Technical report, Real-Time Magazine*

[10] ,*Charles Curley. Open source software for solutions. Linux Journal, 66:1*

[11] ,*COMEDI,http://www.comedi.org/doc/index.html*

[12] ,*Grafcet,http://www.lurpa.ens-cachan.fr/grafcet.html*