EXPERIENCES WITH A TOOL TO BUILD SUPERVISORY APPLICATIONS

Silvia V. Benítez sbenitez@vnet.ibm.com
Juan J. Seoane seoane@vnet.ibm.com
Gabriel A. Wainer gabrielw@dc.uba.ar
Roberto J.G.Bevilacqua robevi@dc.uba.ar

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Pabellón I - Ciudad Universitaria
(1428) Buenos Aires - Argentina

ABSTRACT

This work is devoted to present the results obtained when using a tool to build Supervisory Systems called IGNATIUS. An architectural decomposition the Supervisory Systems in three service levels allows to improve development times and security. IGNATIUS encapsulates the Basic Service routines, providing a development environment that can be easily used by inexperienced programmers. This approach allowed to build complex Supervisory applications, experiences whose results are discussed with detail.

1. INTRODUCTION

At present the advances and cost reduction in hardware have increased the number of computers used to control physical systems automatically. In several cases, exception conditions in the controlled system occur, and the automatic control of the desired process fails. To prevent these conditions and to insure correct system response, Supervisory Systems can be used [1].

Supervisors can be built tailored to the application, or using any of the existing general purpose supervisory tools (also called SCADA, Supervisory Control And Data Acquisition). SCADA systems coordinate, monitor and service system components. They handle communication with the controlled processes, schedule the execution flow, assess priorities between application programs and carry out housekeeping functions. They also process interrupts and deal with error and emergency conditions. They must be designed to coordinate the functions of the system under varying loads. They also must provide the managers and plant engineers with a snapshot of the process status.

This work was partially supported by the SECYT (Science and Technique Secretary) of the Universidad de Buenos Aires. Research Project EX-033, "Concurrence in Operating Systems".

Both approaches are not flexible enough and have a very limited range of use. The use of SCADAs for simple applications can be difficult to manage (as configuration tasks are usually complex). Supervisors specially designed for an application can be difficult to maintain (or to scale up if the system complexity increases). To avoid these problems, the design and implementation of a tool to build SCADA applications was faced [2]. Three *Service Levels* common to all SCADA systems were identified. The tool (called IGNATIUS) was built to avoid the development of the basic services, reducing and easing the process cycle.

This *Basic Service Level (BSL)* encapsulates all the routines to provide basic supervision services. It includes, for instance, point internal representation, alarm detection routines, historic storage, user command interpretation, high level task scheduling, etc. The *Interface Service Level (ISL)* includes the routines to provide man-machine interaction (displays, mimics, alarms, screen messages, etc.). Finally, the *Particular Service Level (PSL)* implement the particular requirements for a specific SCADA (for instance, alarm handling routines, event managing, user command execution, etc.).

In this work we analyze the results obtained with the use of this tool when developing Supervisory applications. Special attention is paid to complexity reduction and improvements in flexibility and development times.

2. TOOL DESCRIPTION

IGNATIUS is not end-user oriented, but intended to be used by programmers that need to develop specific supervisory applications with minimum effort. It was implemented as a Client/Server class library. IGNATIUS acts as a server, and the ISL and PSL as the clients. These service levels may vary for each particular application. Therefore, IGNATIUS can be seen as the same BSL server providing services to different ISL and PSL clients.

The tool was designed to be coupled with most existing semiformal real-time design techniques, but specially adapted to MASCOT [3]. It was also necessary to count with a stable development environment, easy to use by inexperienced developers. Another goal is to improve security, maintainability and correctness of the developed applications. With this purpose several operating systems were analyzed, and OS/2 was selected. This operating system provides multitasking with priorities preemptive scheduling, that can be used to emulate most existing real-time scheduling algorithms.

In OS/2, the smallest execution unit is the *thread*, consisting of instructions, a set of CPU registers values, and a stack. Each *process* consists, at least, of one execution thread, and can have several threads running concurrently. The process consists of data, code and other resources (file handlers, semaphores, pipes, etc.). The multithreading facilities allow efficient execution of new tasks with little overhead. This feature is useful for certain supervisory tasks, such as detection of exception conditions. Finally, a *session* consists of one or more processes, each running in a virtual console (a virtual screen and the keyboard). The operating system supports up to 255 concurrent sessions and 4095 processes (each with a variable number of threads).

In this case, IGNATIUS runs as a process, providing the instructions and data, as other resources. Each of the running task will execute in a different thread. The supervisory applications runs in the session level, providing the virtual console for the application.

The class library has been built using C++, and Vispro/C++ was used as graphical front-end. To obtain further information about these decisions, see [2,4].

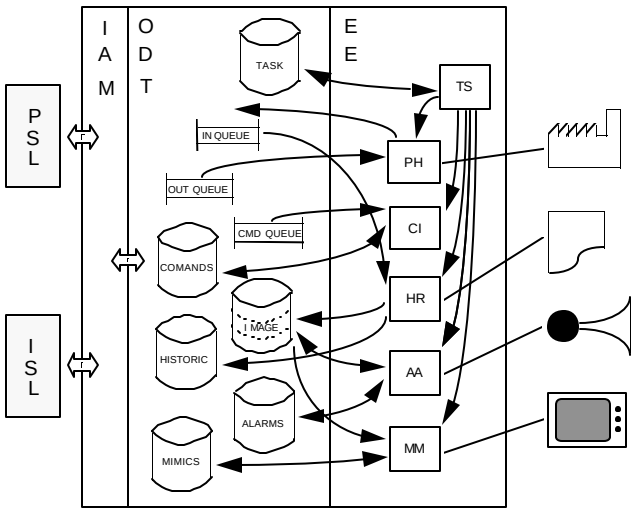

Figure 1. IGNATIUS component interaction.

The main services provided by IGNATIUS are shown in the figure 1, and they can be described as follows.

Object Data Tables (ODT)
This is a set of data structures used to store information about the different real world objects. The following classes are included:

Image: image points for the physical system (divided into three virtual segments representing integer, analog and digital values).

Alarm: values of the alarm conditions that must be checked to detect alarm scenarios, and the alarm conditions methods as well. It also links alarm conditions with real world image points.

Queue: messages sent from the SCADA to the physical devices (In Queue) and the messages sent from the physical devices to the SCADA (Out Queue).

Historic: stores historical information of the image points based on an update parameter externally specified.

Task: synchronizes the execution of the different system components by using system semaphores. It is used to perform high level task scheduling.

Process: basic information for a physical process (relating mimics, displays, alarms, command interpreters, etc.). It provides information that can be used by the ISL to display process data.

Message: messages sent by the class Mimic to the SCADA. Each message contains information about an image point and its attributes on the screen: position, color, etc., used by the ISL to display the process' image points.

Pipe: communication channel between the Mimic class and the SCADA for the message delivery. The Mimic class writes Messages in the Pipe, while the ISL read this messages to display the image point information on the screen.

Mimic: links the physical processes with image points and their screen attributes: position, foreground color, background color, etc.

Port: I/O port handling, used to connect the computer to the physical devices. The class allows to link physical devices with image points letting the user to define multiple relations between them.

Command: a mechanism to define command interpreters. It also validates the commands and parameters entered to determine if they are correct. Command definition, handling

and validation are separated from the command execution (command assistance routines must be implemented in the PSL).

Information Access Methods (IAM)
These methods provide a way to use the data stored in the ODT. They are implemented as class methods that enable the user to store, read, update and delete the ODT information.

Execution Engine (EE)
This set of routines included to ease the construction of complex applications. They have been built using the ODT (accessed through the IAM), and can be changed or excluded if necessary. The main components include a High Level Task Scheduler (TS), an Alarm Analyzer (AA), a Historic Recorder (HR), a Mimic Manager (MM), a Command Interpreter (CI) and a Port Handler (PH).

4. BUILDING SUPERVISORY APPLICATIONS

IGNATIUS has been used as an educational tool, and this section will present two different supervisor application developed by students. The goal of this phase was to test the tool and show its usage by building different supervisor systems. In both examples the requirements of a drying oven process application [5] has been considered. First, the requirements of this control application were considered and a specification document was constructed.

In this case, a drying oven is considered. The oven consists of three different areas: preheating, drying and cooling. The material to be dried is placed on a conveyor belt used to transport the material through the oven. The normal velocity of the conveyor belt is 80 to 100 feet/minute. If the speed falls below those values, an alarm must be issued. The temperature is maintained by using gas heaters. Their values are automatically controlled using a PID algorithm, and the present value must be displayed in the operator console. If the temperature differs more than 5% from the set-point, another signal alarm must be issued.

As a second stage, the specification document was used to build general and detailed design documents using MASCOT methodology. Finally, the supervisor was implemented following the detailed design specification, using IGNATIUS and the visual programming tools stated earlier.

The first supervisor developed is a simple application specially tailored to the controlled system. It was built to experience the usage of the tool and to determine the minimum effort required to build a single purpose application. In this case, the resources are statically defined (the operator has no capability to dynamically configure them).

The control applications were not built, but simulated. Two different simulation approaches were considered, allowing to test different facilities provided by the tool. The first one was centralized, running the control simulated application as another thread in the system. The simulation is executed by including the corresponding task in the system task queue to be scheduled by the high level Task Scheduler. Every sensor, actuator and controller was emulated with an object in this class. The activation rates for each control task were established during the supervisor construction. In this way we also tested how to develop simulation facilities using the tool.

After that, a simple distributed (hierarchical) configuration was considered (i.e., the control application running in a PLC or an industrial PC communicating with the supervisor through RS-232 ports). In this case, the control parameters were simulated in a different processor. The Port Handler reads the input messages, and places them in the input queue, letting the supervisor to act as if an external message would have arrived.

The main work was devoted to build the ISL and PSL routines. The ISL for this application consists of a simple mimic, a command interpreter and a historic recorder. The mimic displays a unique bit-map for the process, with values associated for each of the monitored variables. The command interpreter allows to input user commands.

The available commands are strongly related with the PSL routines. In this case there are commands to show or print historical records, and to refresh the screen. Other commands are related with the drying oven operation. The oven can be started or stopped, the conveyor belt can be halted, and the whole system can be shut down in an orderly fashion.

Four different alarms are included to accomplish with the specification. Three of them are related with the temperature in the different oven areas, and the fourth is used to monitor the conveyor belt speed (the main screens for this system have been included in the Appendix).

At present, this example system has been slightly changed to be tailored to a real application for a cotton plant. In this case a cotton drying control system has been implemented using digital control, and the modification of the presented application provides supervisory functions.

The second example developed allowed to test the utility of the tool to build complex customizable SCADA systems. It also served to determine the effort required to build a SCADA using all the facilities of the tool. This application

lets the operator dynamically configure the resources: define image points, alarm conditions, processes, physical devices, screen representations, update the image point values, alarm conditions, etc. The flexibility given by the dynamic configuration, allows us to use of the SCADA for any particular purpose implementation.

In this example also the main work was devoted to define the ISL and BSL routines. The ISL was built to allow simple configuration and operation, including pop-up menus and icons bars representing the existing operations in a friendly fashion. Commands can also be manually issued. Alarms can be recognized by issuing a command, or by clicking buttons on the screen. Different alarm levels are displayed and can be recognized. Each process can use different mimics that can be defined with any existing graphic tool, and are stored as bitmaps (the main screens for this SCADA are also included in the Appendix).

Each of PSL functions permit to define or use different functions. In this case, the particular services allow different flexible operations, including:

- Image point definition and update;
- Physical process basic definition and association with mimics;
- Mimics definition and their association with image points;
- Alarm condition definition and update, allowing its association with user defined attention routines;
- Definition of relationships between alarms and image points;
- Definition of input/output ports and their initialization;
- Port association with image points;
- Historic data recording conditions;
- Historic data visualization;
- Execution motor start/stop;
- Command input and interpretation.

5. EXPERIMENTAL RESULTS

The tool was thoroughly tested. First, unit test was carried out, analyzing each of the methods in the IAM, independently of the integrated behavior. After that, the integration test allowed to check the behavior of the integrated work for each of the functions in the IAM. The main attention was paid to the interface testing and the communication between modules. Finally, the examples presented in the previous section also served as a system test workbench. In all the mentioned cases, the chosen tools allowed to improve the error detection.

The use of object-oriented programming made easy the error detection. The stable environment provided by the operating system avoided system crashes, even when the errors hung individual tasks. Building the applications following the specifications and the design also helped to detect subtle mistakes (mainly those generated by concurrent task's synchronization). Finally, the division in three levels also helped to ease the testing phase: errors are easily associated with the service level in which they are detected.

Both examples were tested while the CPU was shared with other applications. The response times resulted between the specified parameters (with an activation rate of, at most, one second). The SCADA execution mode was also changed to background, and response times were respected. Therefore, the user can execute other tasks concurrently with the SCADA, without letting the supervisor system off line. For instance, he can analyze historical records, start new processes, add alarm conditions, monitor mimics, etc.

The timing requirements were accomplished by the high level scheduler. Here, the priorities preemptive scheduling algorithm was combined with the Rate Monotonic algorithm [6]. This combination avoided the excessive use of the CPU by processes that could delay the execution of the EE, and reduced the interference of low priorities tasks.

To provide more precise activation rates, the system was tested with standard system loads and the average minimal periods were obtained. The tests were repeated with the system running in a stand-alone fashion (see Table 1). These tests were done in an 50 MHz Intel 80486 CPU with 4Mb RAM.

EE routine	(A)	(B)
Alarm Analyzer (AA)	0.15	0.05
Historic Recorder (HR)	0.45	0.20
Mimic Manager (MM)	0.35	0.15
Command Interpreter (CI)	0.65	0.50
Port Handler (PH)	0.50	0.05

Table 1. EE Minimal Intervals (in seconds). (A): average system execution times when running concurrently with standard loads; (B) Stand-alone execution mode.

The use of the tool allowed to reduce the time spent to design, code and test the examples. A total of 24 man hours was used for the first example, while only 120 man hours were used to develop the second example. These values clearly reflect the tool usefulness and the reduction in the development cycle time, mainly considering that both systems were developed by undergraduates. The complexity is highly reduced, letting the designer to concentrate on high level design aspects.

The development effort is proportional to the complexity of the particular problem environment. As we could see in our

examples, the effort is minimum for medium complexity environments.

6. CONCLUSION AND FUTURE DIRECTIONS

This work presented the results obtained when using a tool for integrated and flexible development of SCADA applications.

A three level classification of the supervisory tasks allowed to reduce the development times for SCADA applications. The Basic Service Level is encapsulated in the tool, reducing the programming tasks and allowing flexible development of general and particular purpose SCADAs.

The utility of the tool and the effort required to built SCADAs with different complexity requirements were tested. Performance bounds were determined by checking the response time constrains for different task intervals.

The use of Object Oriented programming and the provision of a stable environment also helped to reduce the development times. These factors also made easier the debugging and testing phases (the most expensive ones for this kind of systems). Therefore, the tool also can be used as a base for rapid prototyping or incremental design.

The proposed decomposition in three service levels and the provision of the basic service level reduces any SCADA development. Only the ISL and PSL must be built, shortening the development cycle and reducing the complexity. The implementation of these services is dependent of the software platform and the particular application requirements. The implementation time for the ISL can be substantially reduced by using any of the existing visual programming tools.

The development cost of the particular services is proportional to the environment complexity, improving the flexibility of the developed systems. The tool also lets the user concentrate on high level design aspects, providing a complete development environment that can be easily used by inexperienced programmers. It can help the programmers to maintain the systems, make changes, and develop correct applications. This could be checked by using the tool with educational purposes in undergraduate programs.

IGNATIUS is now being integrated in several projects where another tool [7] is being used. Several structural changes have been faced, including the migration of the tool to other operating systems, and its use in a distributed Client/Server architecture. In this case the BSL will be implemented as a separated server providing services to different supervisory systems (the clients). It is also planned the addition of interfaces to different industrial PCs and PLCs and its testing in rough application environments.

Finally, the development of other supervisory applications (electronic worksheets, statistic generators, etc.) will be faced. Most of these improvements will be carried out by undergraduates without explicit knowledge about the tool. In this way, we would test the tool's utility when used by inexperienced programmers.

At present the tool is public domain and can be obtained at: http://www.dc.uba.ar/people/materias/str.

7. REFERENCES

[1] G. Wainer. *Real-Time Systems: concepts and applications.* (in Spanish). Nueva Librería. 1997.

[2] S. Benítez; J. Seoane, G. Wainer and R. Bevilacqua. "IGNATIUS: a tool to develop Supervisory Systems". *Technical Report 96-006, Departamento de Computación, FCEN/UBA.http://www.dc.uba.ar/people/proyinv/tr.html.* 1996.

[3] M. Jackson. *The official handbook of MASCOT, version 3.1*. Computing Division, RSRE, Malvern. 1987.

[4] S. Benítez and J. Seoane. *Development and implementation of a tool to build supervisory systems.* (in Spanish). M.Sc. Thesis. Computer Sciences Department. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1996.

[5] S. Bennett. *Real-Time computer control: an introduction*. Prentice-Hall International. 2nd. Edition, 1993.

[6] J.P. Lehoczky; L. Sha; Y. Ding. "The Rate Monotonic Scheduling algorithm - exact characterization and average case behavior". *Proceedings of the IEEE Real-Time Systems Symposium.* CS Press, Los Alamitos, Calif. 1986.pp. 166-171.

[7] G. Wainer. "SSDT: a tool to develop Real Time Supervisory Systems " (in Spanish). *Proceedings of the Jornadas Chilenas de Ciencias de la Computación*, La Serena, Chile. October 1993. pp. 44-52.

APPENDIX - Graphical interfaces

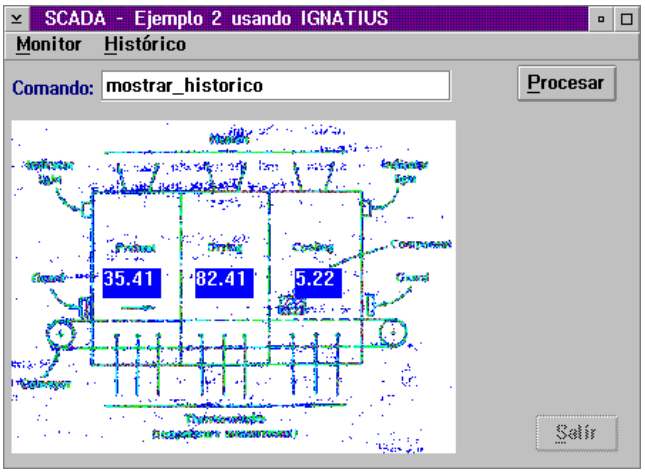

Figure 2. Example 1 - Main Window.

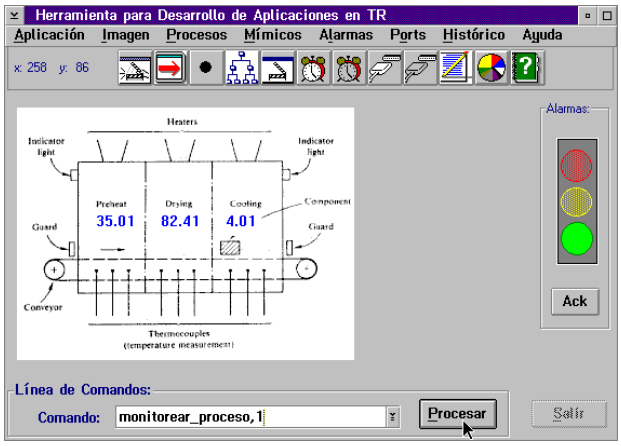

Figure 3. Example 2 - Main Window.

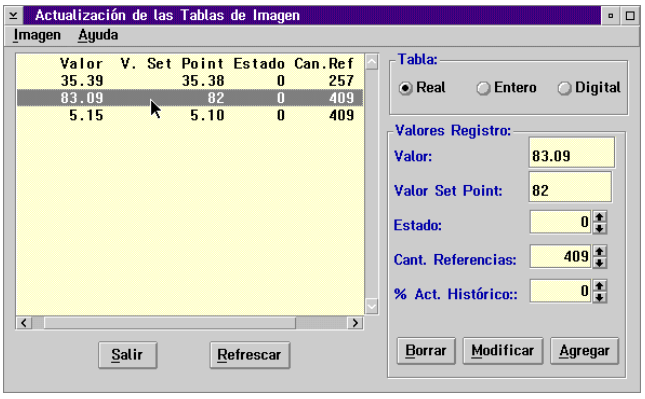

Figure 4. Example 2 - Image point definition.

Figure 5 - Process information.

Figure 6 - Mimics information.

Figure 7 - Alarm information.

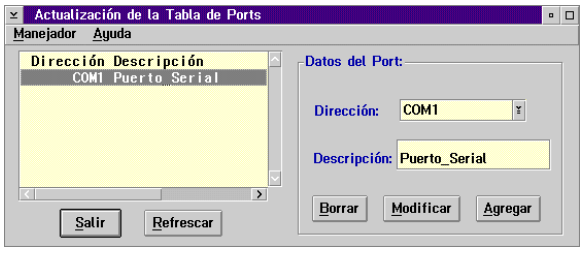

Figure 8 - Port information.