A Case Study in Quantitative Evaluation of Real-Time Software Architectures[1]

José L. Fernández, Bárbara Álvarez, Francisco García,
Ángel Pérez, and Juan A. de la Puente

Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid
ETSI Telecomunicación, Ciudad Universitaria s/n, E-28040 Madrid
Email: jpuente@dit.upm.es

Abstract. Generic architectures for specific domains can provide significant gains in productivity and quality for real-time systems development. In order to choose among different architectural features, a variety of qualitative criteria have been proposed in the literature. However, real-time systems require a more exact characterization based on quantitative evaluation of some architectural features related to timing properties, such as scalability. In this paper we explore a possible way of using Rate Monotonic Analysis to get a measure of scalability between alternative architectures. The technique is illustrated with a case study in a well-known real-time domain, data acquisition systems. The results show clear differences in scalability for different architectures, giving a clear indication of which one is better from this point of view. We believe that the approach can be used on other properties and domain architectures, thus opening new possibilities for quantitative evaluation of software architectures.

1. Introduction

Recent advances in software engineering show that much can be gained from developing generic software architectures for specific application domains that have some "good" properties for a family of related applications in the domain [2,10,20]. In order to find out which architectural patterns suit better for a set of properties, architecture evaluation methods are needed. Although qualitative methods may be appropriate for a large variety of systems, some properties require quantitative measures for the alternative patterns to be properly compared so that the right decision is made.

Real-time systems are special in that timing properties are part of the required capabilities. Specific timing properties, such as response time or deadline guarantees, are clearly dependent on particular system implementations. Timing analysis methods, such as Rate Monotonic Analysis [13], can be used to analyze the

[1] This work has been partially supported by CICYT (projects TAP92-0001-CP and TIC96-0614).

time behavior of a wide class of real-time systems, and to help detailed design and troubleshooting of these. However, if we look at more abstract properties, such as scalability, which can be applied to a range of systems with a common architecture, we find that there are no general methods applicable that can help a system architect choose among different alternative architectural patterns based on their timing properties.

In this paper we explore a possible way of dealing with this problem. We show how Rate Monotonic Analysis can be combined with prototyping in order to get a measure of scalability between alternative architectures. The technique is illustrated with a case study in a well-known real-time domain, laboratory data acquisition systems, which is mature enough for the implemented functionality and the software architectures to be well known [12]. We compare two alternative architectures based on language interface to hardware, because of the flexibility of this approach and the possibility of controlling real-time behavior. The core of both architectures is quite different because the design principles involved are also different. The first one was obtained using the structured design paradigm for real-time systems [15], while the other one is based on the object-oriented paradigm [18].

We developed instances of both architectures, and analyzed them using Rate Monotonic Analysis. This allowed us to examine how their respective timing properties scale up based on quantitative criteria that have not been considered earlier[2]. The results show clear differences in scalability for both of the architectures, which give a clear indication of which one is better from this point of view.

In the next section we describe the general approach to scalability analysis that we propose. Section 3 describes the functionality of the domain of data acquisition systems that we address, and the software architectures that we wish to evaluate. Then in section 4 we show how scalability analysis is carried out, and the main results of the analysis. We conclude by discussing the applicability of the method to other domains and the work that remains to be done for this purpose.

2. General approach

2.1 Rate Monotonic Analysis

Rate Monotonic Analysis (RMA) is a mathematical approach that helps ensuring that a real-time system meets its performance requirements. It does so through a collection of quantitative methods and algorithms that let engineers understand, analyze and predict the timing behavior of their designs, mainly in terms of their response times [16]. The methods are based on preemptive priority scheduling theory [3], which was originated by Liu and Layland in 1973 [14].

---

[2] Sanden [18] showed a way to compare real-time architectures using only indirect scalability criteria, such as the number of concurrent tasks and rendezvous.

Rate Monotonic Analysis is based on an event-response framework, and is carried out through several phases.

1. Describe real-time situations that apply.
2. Measure the execution time of actions.
3. Build the Implementation Table.
4. Build the Techniques Table.
5. Analyze the situation to determine if timing requirements are met.

Readers interested in a full description of RMA should consult the SEI RMA handbook [13].

In order to apply the method to the data acquisition systems that we have studied, we proceeded by first identifying the events which drive the system behavior, and the responses to them. The responses are decomposed into actions which use different resources. We measure the worst case execution times of the actions by means of a "dual loop" technique [6,17]. The measurements take into account the run-time system overheads and the available clock resolution. From this information, summarized in the so-called 'Implementation table', a set of analysis techniques can be applied which give information on the timing properties of the system.

2.2 Scalability analysis

Scalability is a measure of how the performance of a system varies when its size increases. The approach we have used to analyze scalability in different software architectures is based on RMA. What we do is to parameterize the size of a system by means of some quantifiable measure and then obtain estimates of some characteristic real-time properties for different system sizes. In our case, we have used the number of sensors being processed by a data acquisition system as a measure of its size, and deadline missing and response times as the real-time properties that are analyzed. An architecture is more scalable than other if it admits a larger number of sensors to be processed without missing any deadlines. The following sections describe the particular details and results of this approach for two alternative architectures for laboratory data acquisition.

3. Architectures for data acquisition systems

3.1 Data acquisition systems

In this paper we compare two different software architectures for laboratory data acquisition systems built on a simple platform, based on an Intel 486 /MSDOS configuration, with I/O boards directly connected to the computer bus (figure 1). The implementation language is Ada 83 [1].

Data acquisition is related to the reception of data from instruments or sensors. Since data from sensors are analog, I/O boards are used to sample and convert them to digital values. We decided to use software polling, as it implies the lowest level of custom programming, thus increasing the reusability of software.

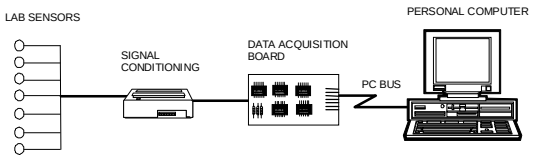

Fig. 1. System description.

Signal ID	Sensor state	Period (ms)	Next_Time_to_Read
T 308	Activated	1000	14 : 05 : 30.600
P204	Activated	500	14 : 05 : 30.100
F203	Deactivated	500	13 : 00 : 30.300
. . .	. . .	. . .	. . .

Fig. 2. Sampling Plan

Conversion to engineering units is typically required before analog data are used by the operator, by other software, or displayed in an alarm message. The software architectures that we analyze perform this conversion in real-time after each data sample is acquired.

Range checking is performed with each data sample before converting it to engineering units. Alarm checking is supported by the comparison of the engineering value of each sample with the alarm limits defined and stored in the system database.

Converted data is stored in disk to permit the engineer or scientist the analysis of data or the processing of them using more sophisticated algorithms not available for real-time processing. During the sample data processing the operator can actuate concurrently, sending commands to the system to activate or deactivate a sensor, change the gain or modify the range or alarm limits.

The core of a data acquisition system is a sampling plan which contains information related to the interval of time for recording the value of each signal. Since the interval of time is usually fixed for each signal, the period of each signal sampling is considered constant. Figure 2 shows an example of a sampling plan containing information about the sensor state, sampling period and time of the next measurement for each signal which is being read.

The sampling plan is not always implemented as a single entity. The software architecture based on the principles of structured design contains a unique sampling plan managed by a control transform that sends reading orders to the drivers connected to the I/O boards. Another approach which is typical of object oriented design, distributes the sampling plan in a collection of objects representing the problem domain and frequently known as sensor objects. The resulting architectures are different and will be described in detail in the next subsection.

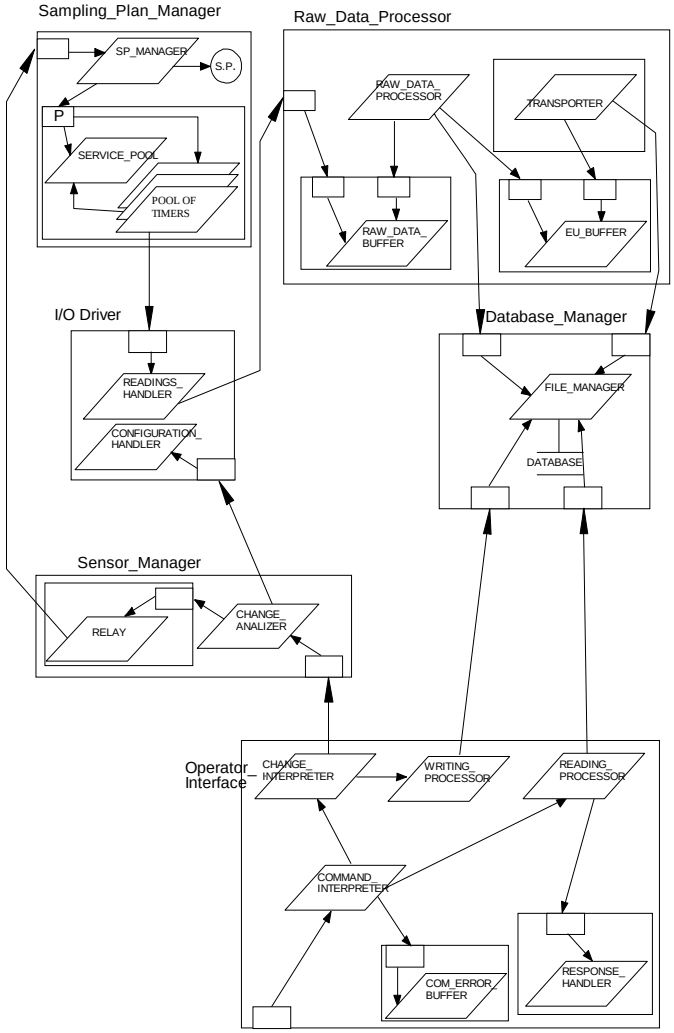

Fig. 3. Centralized software architecture.

3.2 Centralized architecture

The first architecture is based on structured design principles. The sampling plan is managed by a central component that handles the concurrent series of readings for all the sensors and schedules each reading at the appropriate instant. A special-purpose scheduling algorithm is used rather than the native task scheduling of the Ada runtime system, and thus scheduling is handled explicitly by the application.

The architecture is shown on Figure 3. Its components perform the basic functions of the data acquisition system: I/O handling, raw data processing, sensor

management, operator interface, and data base management. There is also a central sampling plan manager. Buffers, transporters, and relays, are used as coordination mechanisms between concurrent activities, implementing synchronization and message passing capabilities [7,9]. Priority assignments are based on the Rate Monotonic Scheduling method [14]. The access protocol for shared resources is the Highest Locker (HL) Protocol [13].

3.3 Distributed architecture

This architecture is based on the object oriented paradigm, so sensors are implemented as independent entities and the implementation of the sampling plan is also distributed.

The architecture is shown on figure 4. As before, its main components are directly related to the data acquisition functions. The sensor objects include all the functions related to data acquisition, including scheduling, for each of the input data signals.

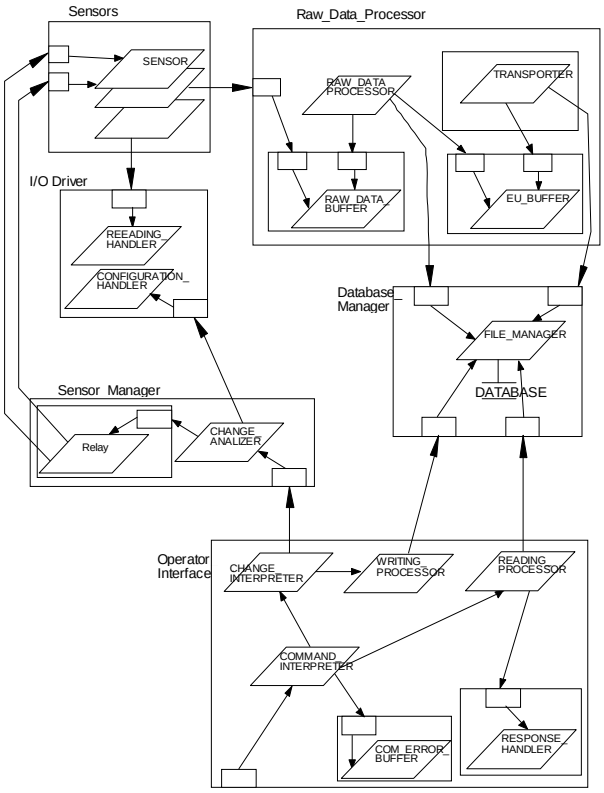

Fig. 4. Distributed software architecture.

The differences between this architecture and the centralized software architecture can be summarized as follows:

- The sampling plan does not exist as a unique entity.
- Periodic readings are managed by a set periodic tasks, each one implementing the corresponding sensor sampling interval.
- The role of each sensor task is to send reading orders, then to receive the raw data from the *I/O_Driver* and add a time tag to each sample. In the centralized solution, some of these capabilities, are implemented in the *I/O_Driver*, making it more complex.

4. Analysis of the architectures

4.1 Main event sequences

The starting point for Rate Monotonic Analysis is identifying the event sequences that act as stimuli for the system, and the responses to them, in terms of actions and used resources [5]. For the sake of brevity we will show only the main sequences related to both software architectures. The sequences having the hardest timing requirements are identified below. Readers interested in the complete description of sequences can consult the technical report describing the analysis results [8]. The event sequences are summarized in tables 1 and 2.

4.2 Rate monotonic analysis of the architectures

The analysis follows the steps described in section 2. It produces the following outputs:

Situation Table
We started building the situation tables by adding action and resource parameters to the above tables. The purpose of this form of the table is to capture the timing requirements.

Implementation Table
We derived the implementation table of the architectures using the action execution times that we obtained by the measurement method described in section 2, as well as other numeric parameters.

Techniques Table
The techniques table is a simplification of the implementation table. Its goal is to have a set of parameters that still describes the architecture but restricts assumptions to conditions where proven mathematical reasoning can be brought to bear. In this tables all the arrival patterns are approximated as periodic arrivals with hard deadlines.

The generation of the techniques table for the centralized architecture was specially difficult, and we consider it is one of the main contributions of this work.

There are several events, the reading orders, with a bursty arrival pattern, i.e. they can arrive arbitrarily on an interval defined by the sampling plan period.

Table 1. Event sequences in the Centralized Software Architecture

Event Name	**Type**	**Arrival Pattern**	**Time Req.**
Sensor_Reading.	Timed	Periodic, Ts	Hard, [Ts]
Timer_Assign.	Internal	Bursty [N,V]	Hard,[Ds]
RD_Processing	Timed	Periodic, Tp	Hard [Tp]
EU_Data_Storage	Timed	Periodic,Tt	Hard, [Tt]
Ch_Sensor_Gain	External	Bounded [L]	Soft [D]
Ch_Sensor_State	External	Bounded [L]	Soft [D]
Ch_Limits	External	Bounded [L]	Soft [D]
DB_Consult	External	Bounded [L]	Soft [D]
Erroneous_Comm.	External	Bounded [L]	Soft [D]

Table 2. Event sequences in the distributed software architecture

Event Name	Type	Arrival Pattern	Time Req.
Sensor_Reading_1-8	Timed	Periodic,T1-T8	Hard [T1-T8]
RD_Processing	Timed	Periodic,Tp	Hard [Tp]
EU_Data_Storage	Timed	Periodic,Tt	Hard [Tt]
Ch_Sensor_Gain	External	Bounded [L]	Soft [D]
Ch_Sensor_State	External	Bounded [L]	Soft [D]
Ch_Limits	External	Bounded [L]	Soft [D]
DB_Consult	External	Bounded [L]	Soft [D]
Erroneous_Comm	External	Bounded [L]	Soft [D]

The worst-case response time for an aperiodic event occurs just after the polling task has checked for the event arrival when all reading orders arrive at the same time. In this case, the last event has to wait for the preceding events to be processed. The first event would have to wait one polling period for its processing to begin. To model this situation we propose a scheme , where the timer tasks are considered as polling tasks which process sensor reading orders and send them to the *I/O Driver*. For the purpose of the analysis, we represent this part of the system as completely periodic, and therefore we can analyze it as if were a collection of periodic tasks. It is important to emphasize the presence of blocking due to the time interval spent by each timer task until it sends a sensor reading order to the *I/O Driver*. We estimated the blocking time considering the worst case, which is reading all the sensors in 500 ms (the sampling plan period).

The generation of the techniques table for the architecture with the distributed sampling plan is easier than the previous one, as the sensor reading tasks are periodic and there are no blocking delays.

Situation Analysis
The next step is applying analysis techniques to the event sequences using the information of the techniques table. In our case, all event responses can be modeled as periodic tasks with varying priorities, which can be analyzed using a well known technique [11,13]. The result of this analysis is the worst case response time for each event.

4.3 Results of the analysis

The main purpose of our analysis was to determine quantitatively what software design features contribute to the limitations in the performance of the real-time laboratory automation system, comparing the behavior of both design solutions. For this purpose, we start analyzing the response times for a system with 8 sensors with different frequencies, and then we scale up both software architectures by incrementing the number of sensors in groups of 8, until some of the sequences fail to meet the timing requirements.

It is important to emphasize that the timing requirements of both architectures are apparently different. In the distributed software architecture, each sensor is managed separately, implemented by a different component with an execution thread with a period and deadline determined by the sensor sampling requirements. The resulting behavior is typically periodic. The situation is quite different in the centralized software architecture, where all sensors are managed in the same way despite their sampling requirements . There is one object implementing the sampling plan, the *SP Manager*, containing a task that generates reading orders for all sensors. Due to the fact that different sensor readings are not distinguished and can occur during a very short time interval, the sequence behavior is considered bursty.

Dealing with different sensor reading requirements
Bursty arrival patterns are characterized by an event density. An event density consists of a bursty interval, the length of time over which the burst restriction applies, and a burst size, the number of events which can occur during that time interval. In the centralized software architecture, the burst interval is 500 milliseconds and the burst size 8, so the system is required to process 16 reading orders per second in the worst case. A distributed software architecture with identical sensor sampling periods as the centralized solution only requires 8 readings per second in the average. The above considerations determine the main differences in the results obtained.

For 8 sensors, the CPU total utilization is 15.41% for the centralized software architecture versus 7.86% for the distributed software architecture. This is due to the fact that the average readings of the centralized architecture for identical sensor sampling requirements duplicate the average readings of the distributed software architecture. With this number of sensors, there are no missed deadlines in either architecture.

When scaling up the centralized architecture, the first sequence to miss its deadline is *EU_Data Storage,* when the number of 48 sensors is reached.

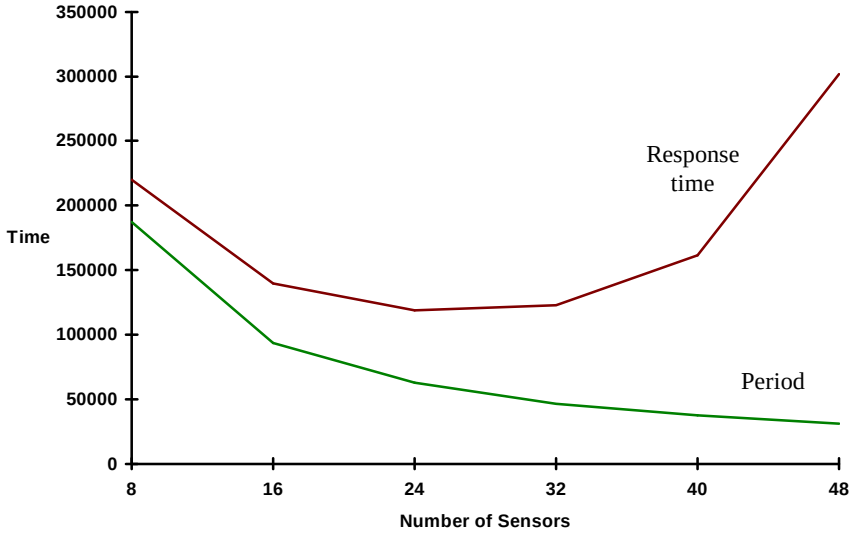

Fig. 5. Behavior of the Timer_Assignment sequence (Centralized Software Architecture)

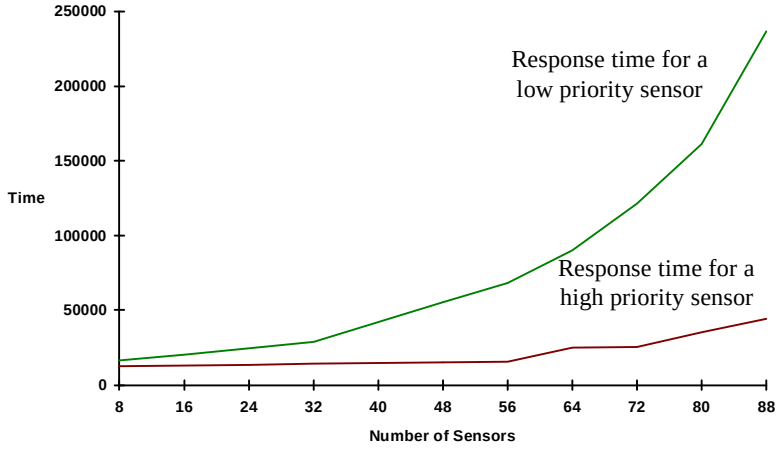

Fig. 6. Behavior of the Sensor_Readings sequence (Distributed Software Architecture**)**

For the distributed architecture, the first sequence missing its deadline is also *EU_Data Storage*, but this happens only when we arrive at a number of 88 sensors. The results are pessimistic for the centralized software architecture due to the bursty nature of its arrival patterns. In the analysis we are considering the worst-case when all the signals are acquired during a 500 milliseconds time interval.

Architectures Behavior

It is also interesting to describe the behavior of the sequences related to sensor readings in both architectures. Figure 5 shows the timing behavior of the *Timer Assignment* sequence for the centralized software architecture. The response time has an optimum value when 24 sensors are processed (122746 microseconds). There are two reasons to explain this behavior. When the number of sensors is lower than 24, the response time is influenced by the idle time of the timer task. When the number of sensors is greater than 24 we have to consider the increase in the processing requirements of the *Raw_Data_Processor* task (see figure 3).

Figure 6 shows the timing behavior of the *sensor reading* sequences for the distributed software architecture. We distinguish between sensors with high sampling frequency and low sampling frequency. The response time increases as the number of sensors increases. For high sampling frequency (high priority) sensors the behavior is almost linear. In contrast, the response time for low priority sensors increases abruptly as the number of sensors grow. Thus the distributed software architecture handles sensors differently with respect to their sampling time requirements, as opposed to the centralized architecture, in which all the sensors are handled in the same way in spite of their sampling time requirements.

5. Conclusions

Performance engineering using RMA allows quantitative comparison of alternative architectures, giving illustrative results about system behavior and scalability properties that cannot discovered by testing.

RMA is best suited to periodic arrival patterns, but in this study we had to deal with a bursty arrival pattern, that of the internal events generated by the *SP_Manager* task. We found a solution considering the timer tasks as polling tasks processing the internal events generated by the *SP_Manager* task and queued by the *Service_Pool* task. This modeling solution allowed us to apply seamlessly RMA.

Diverse causes of inaccuracy can be identified in the analysis techniques (Bailey 1995):

- Execution times are always assumed to be at the maximum.
- Sporadic inter-arrival intervals are always assumed to be the minimum.
- Compiler optimization has been prohibited to allow the calculation of execution times.
- Overheads associated to the Ada runtime system are simplified.

These inaccuracies imply pessimistic results in the evaluation of the architectures. Therefore, we think that the method is more suitable to compare diverse solutions as we did, than to analyze single designs at the architectural level of description, since the inaccuracies impact evenly in the evaluation of the diverse architectures.

References

1. Reference Manual for the Ada Programming Language (1983). ANSI/MIL-STD-1815A-1983; ISO/8652:1987.
2. A. Alonso, B. Álvarez, J.A. Pastor, J.A. de la Puente, A. Iborra (1997). "Software Architecture for a Robot Teleoperation System." Proc. IFAC Symposium on Algorithms and Architectures for Real-Time Control. Elsevier Science, 1997.
3. N.C. Audsley, A. Burns, R.I Davis, K. Tindell, and A.J. Wellings (1995). "Fixed Priority Pre-emptive Scheduling: An Historical Perspective." Real-Time Systems, vol. **8**, no. 2/3, pp. 173-198.
4. C.M. Bailey, A. Burns,. A.J. Wellings, and C.H. Forsyth (1995). "A Performance Analysis of a Hard Real-Time System." Control Engineering Practice. Vol. **3** No 4: 447-464.
5. R.J.A. Buhr, (1990). Practical Visual Techniques in System Design. Prentice Hall.
6. R.M. Clapp and T. Mudge (1990). "The Time Problem." ACM Ada Letters, Vol. **X**, No. 3: 20-28.
7. J.L Fernández (1993). "A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis." Technical Report CMU/SEI-93-TR-34, Software Engineering Institute, Pittsburgh, PA.
8. J.L. Fernández, A. Pérez, B. Álvarez and F. García (1995). "Performance Engineering of Real-Time Laboratory Automation Software Architectures." Technical Report DIT/UPM 1995/01.
9 J.L Fernández (1997). "A Taxonomy of Coordination Mechanisms Used by Real-Time Processes." ACM Ada Letters, vol. **XVII**, no. 2: 29-54.
10. D. Garlan and M. Shaw (1996). Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall.
11. M.González-Harbour, M.H. Klein and J.P. Lehoczky (1991). "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority." Proceedings of the IEEE Real-Time Systems Symposium. Los Alamitos, C.A. pp.116-128.
12. R. House (1995). "Choosing the Right Software for Data Acquisition." IEEE Spectrum. May 1995: 24-39.
13. M.H. Klein, T. Ralya, B. Pollak, R. Obenza and M. Gonzalez-Harbour. (1993). A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers.
14. C.L. Liu and J.W. Layland (1973). "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment." ACM Journal. **20**,1: 40-61.
15. K. Nielsen and K. Shumate (1988). Designing Large Real-Time Systems with Ada. McGraw-Hill.
16. R. Obenza (1993). "Rate Monotonic Analysis for Real-Time Systems." IEEE Computer. Vol. **26**, No 3: 73-74.
17. D. Roy (1990). "PIWG Measurement Methodology." ACM Ada Letters. Vol **X** No 3: 72-90.
18. B. Sanden (1989). "Entity-Life Modeling and Structured Analysis in Real-Time Software Design. A comparison." Communications ACM . Vol. **32** No 12: 1458-1466.
19. L. Sha and J.B. Goodenough (1990). "Real Time Scheduling Theory and Ada." IEEE Computer. Vol. **23,** No 4: 53-62.
20. B. Witt, T. Baker, and E. Merrit (1994). Software Architecture and Design. Principles, Models and Methods. Van Nostrand Reinhold.