Developing Component-Based Software for Real-Time Systems

Janusz Zalewski
School of Electrical Engineering & Computer Science
University of Central Florida
Orlando, FL 32816-2450, USA
jza@ece.engr.ucf.edu

Abstract

This paper discusses the principles of developing software components for real-time systems. The procedure is based on the fundamental concept of a real-time architecture rooted in the feedback control paradigm of control engineering. Generic design patterns for real-time software components are presented, valid for all relevant real-time architectures. Finally, a case study of an air traffic control system based on the CORBA framework is discussed. Tool support for component-based design and implementation is presented, including industry-strength commercial off-the-shelf software.

1 Introduction

Developing software components for real-time systems tends to be more difficult than for other domains, because of a unique nature of each individual real-time application. The primary difficulty lies in developing a generic software architecture and abstracting its meaningful components, which are shared among the majority, if not all, models and applications. Due to the uniqueness of real-time problems it is hard to find such distinctive templates and generalize them justifiably to produce reusable real-time software components [2, 6].

One approach to real-time software design is rooted in control engineering and seems to be fruitful for this purpose. It is based on a feedback control paradigm and has been mentioned in several papers, in the last one and a half decade [3]. The principles of this approach have been summarized recently in [12] and are used here to develop real-time software components. It is argued that for all types of real-time systems it is possible to abstract a few representative architectural properties, which are expressive enough to allow the designers base the development on a few structural and behavioral patterns.

The rest of this paper is structured as follows. First, we discuss a generic architecture of real-time software. Then, we develop a pattern for one type of application, a data acquisition system. Next, we present a case study of a sophisticated air traffic control system viewed as a data acquisition system, and finally outline the development of software components with the use of off-the-shelf design and implementation tools.

2 Generic Architecture

Modern real-time systems can be all viewed as specific instances of a general feedback control system presented in Fig. 1. In the most general case, such system includes all of the following elements:

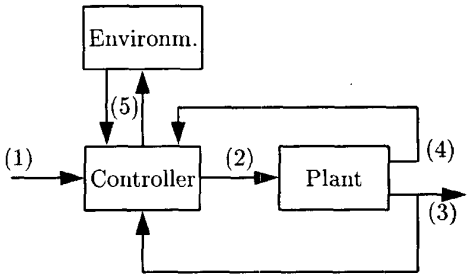

Fig. 1. Illustration of a feedback control system. (1) Desired value; (2) Controller commands; (3) Controlled variables; (4) Other measured variables; (5) Environment interface.

(1) Desired value; a reference for the Controller to make necessary adjustments of controlled variables.

(2) Controller commands; signals applied to the Plant (outputs from the Controller) in order to achieve

1089-6503/01 $10.00 © 2001 IEEE

its desired behavior.

(3) Controlled variables; signals received from the Plant (inputs to the Controller), whose values are being controlled.

(4) Other measured variables; auxiliary signals received from the plant (inputs to the Controller) which are not controlled but used in the determination of the best values of Controller commands.

(5) Environment interfaces – user interface, mass storage interface, and communication link to computer network.

Because of the timing requirements (such as time constants) imposed on the controller dynamics, it always operates in real time. Since nowadays a controller is implemented as a digital processor and its functionality can be extended much beyond that of a regulator function, we can call it a real-time computer.

It is also evident that in addition to a traditional interface a controller has to the process (which includes sensors and actuators), a modern real-time computer interacts with the environment in a number of other ways, including interfaces with a plant operator, mass storage (database), and computer network. A more detailed view of these interfaces is presented in a unified diagram shown in Fig. 2.

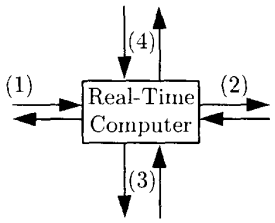

Fig. 2. Real-time computer system. (1) User interface; (2) Process interface; (3) Mass storage interface; (4) Communication link.

In practice, a number of real-time systems exist that do not represent a complete system in a sense of Fig. 1, but nevertheless fit very well into this concept. Respective examples include:

- data acquisition systems, when the connection 2 in Fig. 1 is broken (there is no control signals from the real-time computer)
- programmed controllers, when the feedback connection in Fig. 1, from the plant to the controller, is removed, with connection from the controller to the plant remaining intact
- reduced architecture, if both connections between a plant and a real-time computer are broken (what remains is only interfaces with the operator, the network and the database).

While the examples of real-time systems in the first two categories are intuitively clear, existence of the last category may be less obvious. However, taking a closer look at respective dataflows reveals that there are several examples of that kind of real-time systems in practice. Putting emphasis on the distribution and communication, with relatively less interest in GUI and database access, brings us to a typical case of real-time simulation. With a slightly different emphasis, concentrating on the database use and GUI, one has a real-time multimedia system.

3 Real-Time Design Patterns

Once we have an understanding of the nature of a real-time system architecture, we can focus on developing its design and shaping its software architecture. However, thus far, there has been very little guidance on selecting real-time architectures, either in the engineering literature or in practice. When one takes a closer look at Figures 1 and 2, with explanations (1)-(5) in the previous section, there is little doubt that one can and should start designing the controller from the context diagram similar to that in Fig. 3.

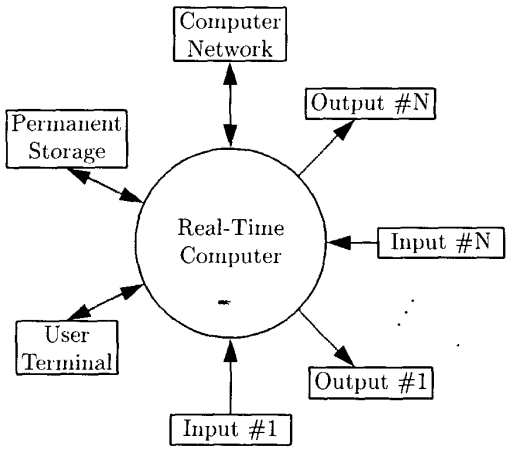

Fig. 3. The top level context diagram.

The role of a context diagram cannot be overestimated. Even though it is a relatively old notational vehicle, it's been well established in real-time software design as the basis for architectural development. It is at the context diagram level, where the interfaces between the software and the external world need to be defined and developed. For this very reason, the concept of a context diagram is indispensable as a starting point in designing a software architecture.

From Figure 3, it becomes immediately clear that the software components must include units responsible for the following interactions with all external elements:

- inputs from and outputs to the plant
- interaction with a user
- possible communication with other controllers/processors
- interaction with storage devices

enhanced by the processing (computational) capability. The time source is also important and can be internal or external depending on circumstances.

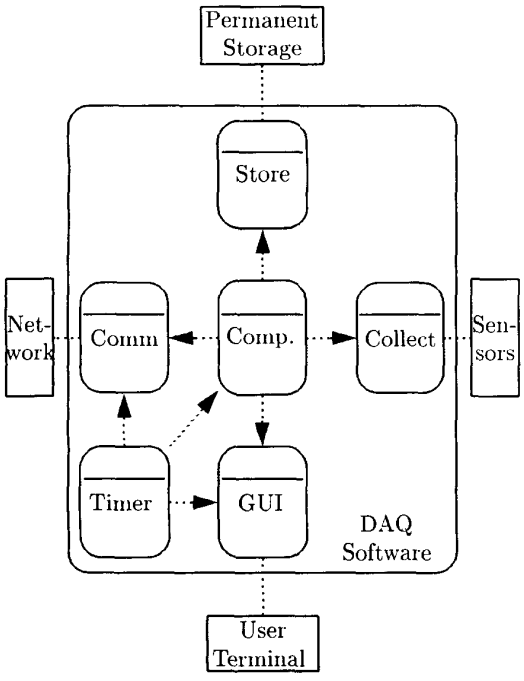

Fig. 4. Outline of a generic data acquisition system.

An example of a corresponding design, which can be considered a basic and generic design pattern for real-time software, is presented in Fig. 4, for a data acquisition application.

Respective software components need to comply with the principle of separation of concerns. They can be considered as sequential modules or individual concurrent tasks, and can run, respectively, on a single processor, on multiple processors, or even on a distributed system or network.

This basic design pattern can be expanded further into more comprehensive architectures, depending on the focus of a particular application, such as a distributed real-time system. Depending on a predominant function of a distributed system, one may produce a variety of its particular architectural instances. One such example (Fig. 5) comes from the area of high-energy physics, where multiple data collection and control facilities are spread over a large area surrounding an elementary particle accelerator [5].

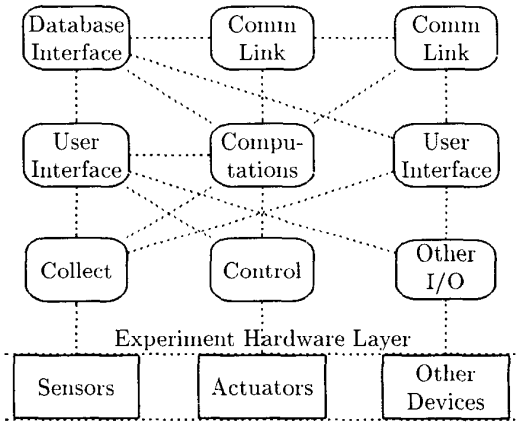

Fig. 5. Generic architecture of a distributed real-time system.

Multiple software components can be created to access various (maybe the same) sources and destinations of data and to exchange information among themselves. Any single component can perform individual functions and communicate with every other component.

Adding a new component or deleting one should have no impact or minimal impact on the operation, in a sense that no degradation of functionality should occur due to such dynamic changes. Communication links can operate individually or be lumped into a mid-

dleware layer [7], with program units communicating partially or exclusively via this layer.

The primary advantage of having such a flexibility is that a number of new components can be created and the architecture expanded during the run of an experiment or operation of a process. One such example is a dynamic GUI creation [8]. If new experiments are conceived which require including additional features to the GUI, or new characteristics are explored which need GUI reorganization, this can be done on-the-fly without jeopardizing the operation of an ongoing experiment or process.

4 Detailed Design Level

For the concept of real-time design patterns, as presented in previous section, to work properly, one must provide enough details at the design level, so that automatic tools could be used. The details mean, in the first place, providing sufficient information about interfaces with the environment. The format involves the following items:

- signal name as a variable (that is, expressed in acceptable textual format)
- its source and direction (input or output)
- signal's detailed function
- its detailed characteristics, including data type (analog, binary, text, etc.), range of valid values, length, shape, frequency, etc. (whatever applicable)
- necessary action in case of invalid value, which effectively means an error handling function.

To illustrate principles of creating detailed designs, starting with such interface description, we will use an example of instrumentation software for testing printed circuit boards (PCB). Let's assume that a PCB stand is interfaced to the following three instruments necessary to operate the board and take respective measurements to test it: power supply, spectrum analyzer and data acquisition box.

A context diagram for the instrumentation software is shown in Fig. 6, limited to the signals exchanged with the above instruments (a plant, in terminology of Fig. 1). A sample list of signal interfaces, exchanged with the plant, is presented in Table 1, following the description format above.

Once the exact description of input and oputput signals is known, a detailed and complete list of actions in terms of operations on all signals has to be developed.

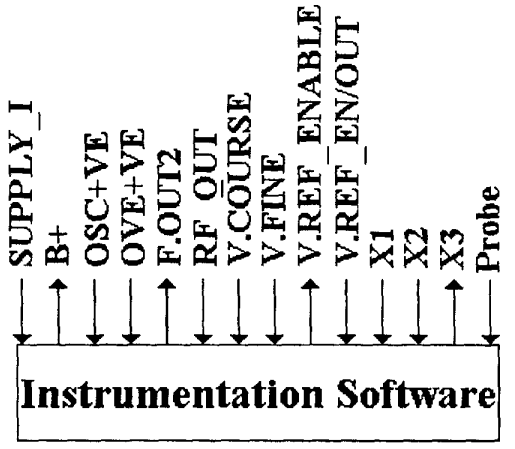

Fig. 6. PSB test system context diagram.

This list can be obtained from the requirements specification document. A sample requirement related to the value of SUPPLY_I is presented below, but the full list of actions had to be omitted from this article due to space restrictions.

Table 1. Detailed signal description (notes are too long to fit in the table).

Signal	Direction/func.	Valid values
B+	OU supply voltage	15 +/- 10mV
OSC+VE	IN regulated supply	11 +/- 0.2V
OVEN+VE	IN regulated supply	12.1 +/-0.2V
F.OUT2	OU from multiplier	10 +/1 1dBm
RF_OUT	IN from oscillator	13 +/- 1dBm
V.COURSE	IN course frequency	2/3 of B+
V.FINE	IN fine frequency	1/3 of B+
V.REF_EN	OU enable RF_OU	On or Off
V.REF_OU	IN check connection	B+ or < 0.4V
X1	IN mid point heater	7.5 +/- 1V
X2	IN bridge amplifier	Note 1.
X3	OU temp sensor	5.5V or 6.5V
SUPPLY_I	IN supply current	Note 2.
Probe	IN monitoring	Note 3.

Requirement X.Y.Z. If SUPPLY_I current exceeds the value specified in *Note 2*, then the testing program shall do the following:

- turn the power supply voltage B+ off

- display the value of `SUPPLY_I` in the message window as follows: "`SUPPLY_I too high:` *value*"
- append the value of `SUPPLY_I` to the file `results.txt`, and
- stop the operation;

otherwise it shall continue. *End X.Y.Z*

Even though we are using such a simple example and have deliberately limited interface description to the plant interface only, leaving out information on the remaining three interfaces from Fig. 2, for the sake of brevity, this single requirement contains references both to the GUI (talking about the message window) and to the database (talking about appending value to the file).

At this point, we are ready to define the structure and behavior of the software, in our example, the instrumentation software. Because testing printed circuit boards is a process sequential by nature, the structure of the instrumentation software is also sequential. Following the generic architectural pattern from Fig. 4, we need to distinguish the following sequential modules in our design component: Acquisition and Control, GUI, Storage Handler, and Main Computation. Knowing that there are three separate external devices in the plant connected to the real-time computer, we'll need to create three corresponding instances of the Acquisition and Control module: Power Supply, Spectrum Analyzer, and Data Acquisition Box.

For completeness, we have to mention two things regarding the generic pattern from Fig. 4. There is no need for us to have a Communication Link, because our testing system is stand-alone. We will most likely be using time only locally, so the Timer module is not needed either. As a result, our component will have a relatively simple structure and can be easily represented as a class diagram, with objects listed in the previous paragraph. However, we must be aware that in general, the developer would have to design concurrent modules, according to the generic pattern from Fig. 4, and define precisely respective interfaces among these modules. For a large system, this is likely to become a daunting task and will be illustrated briefly in the next section.

One last thing, which the designer has to do is defining each module's behavior. For a sequential system like our instrumentation software, the behavior can be easily derived from the list of actions produced above in terms of operations on respective signals, and represented as a sequence diagram. For more complex systems with concurrent modules, this has to be done formally using statecharts.

5 Case Study: Air Traffic Control System

Let us consider the design of software for an air traffic control system (ATCS) conceived as a data acquisition system [9]. It is not, in fact, an automatic control system, because there is no direct connection between the real-time computer and the plant. All commands are executed by the pilot who is receiving respective messages from an air-traffic controller (Fig. 7).

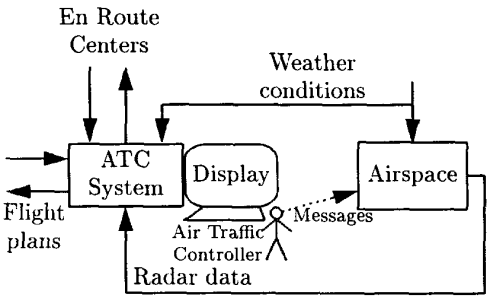

Fig. 7. Air traffic control system as a data acquisition system.

An example of the context diagram for the air traffic control system is presented in Fig. 8. It is fully compatible with the general model (Fig. 3).

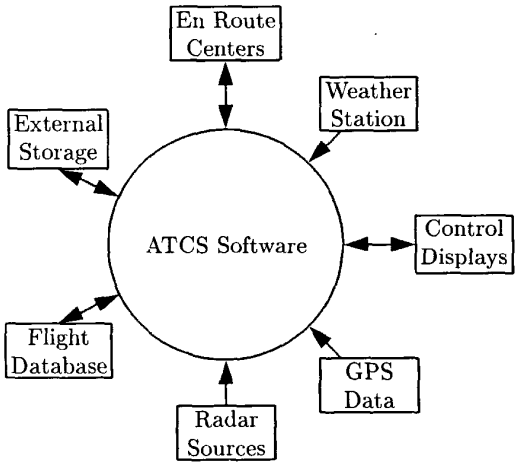

Fig. 8. The top level context diagram for an air traffic control system.

A component-based software architecture derived from this context diagram is presented in Fig. 9. Individual boxes represent software modules interfacing to the external devices and performing respective functions. The architecture is also compatible with the software design pattern shown in Fig. 4 and includes modules responsible for handling:

- three sources of data (radars, GPS and time source; the latter is not shown)
- a user interface to controller displays
- two functionally different mass storage interfaces (for flight plans and event recording)
- two functionally different network interfaces (for *en route* centers and weather services)
- computational function, which in this case is only collision detection but may include altitude warnings, proximity warnings, etc.

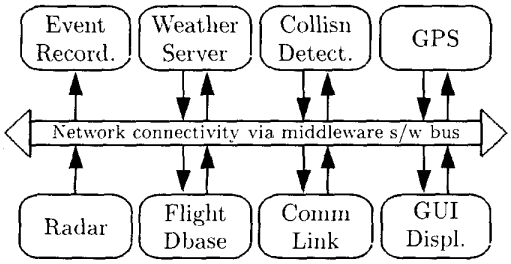

Fig. 9. ATCS software components communicating via middleware.

Since the nature of the system is highly distributed, we use middleware for connecting all the components. This gives the designers the highest flexibility in organizing individual components and shaping their communication structure.

6 Components Development: Design Aspect

Developing software components for the presented ATCS architecture or other contemporary real-time applications, such as the one described in [5] is too complex to be done manually by a single individual. Therefore automatic tools are needed to assist in the development process.

It must be stressed that tools are only a part of a complete design methodology which must include the

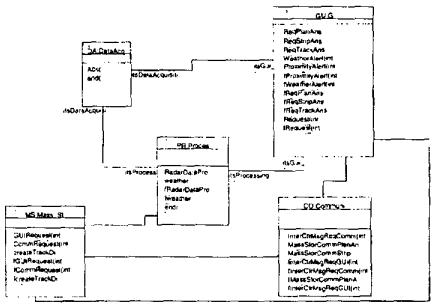

Fig. 10. Sample object diagram in a high-level design tool Rhapsody.

following three elements: method (a graphical notation to express properties of an architecture), techniques (transformations applied to the notation), and software tools supporting the transformation techniques.

To be useful in the development of software components we require the automatic tools to provide support in the following dimensions [12]:

- internal, to express real-time models via the specific notation and respective transformations to create real-time components
- horizontal, related to means of communication with other components and other tools
- vertical, related to the next and previous phases of the development process
- diagonal, related to the use of architectural components in different projects/processes.

In a language of software components this means we have to be able to create, connect, model, and reuse components at the design level.

Several software tools with this concept in mind, which operate in the commercial marketplace, were analyzed [1]. For practical reasons, we focused on those, which have solid methodological foundations, based on OO approaches: ROOM [10] and UML [4].

The selected tools offer extensive services in the internal dimension to model both structural and behavioral patterns of distributed real-time software. The structure and behavior of a component such as the Comm Link from Fig. 9 can be easily expressed as an object diagram and a statechart, respectively (Fig. 10 and 11). In the horizontal dimension, a capability of interacting with externally created components is

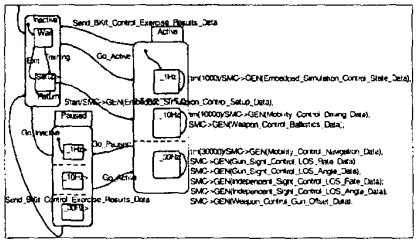

Fig. 11. Statechart for a sample Comm Link module in Rhapsody.

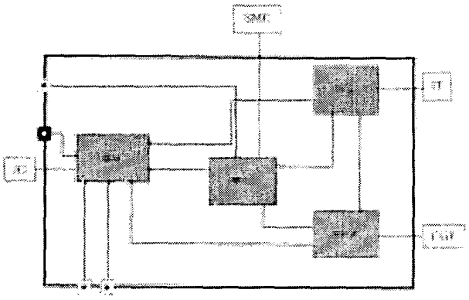

Fig. 12. Modeling communication with external components in ObjecTime.

provided for most tools via standard TCP/IP protocol (Fig. 12).

In the vertical dimension, tools usually have the capability of generating code for specific real-time kernels, however, they lack the capability to perform timing analysis at the design level. They also lack adequate means of importing design components from other tools (diagonal dimension).

7 Components Development: Implementation Aspect

At the implementation level, the design tools should allow timing and scheduling analysis, then correcting the design, and finaly code generation. With current tools, timing analysis is only possible after the code has been generated.

For this reason, when developing the basic ATCS structural components and defining their behaviors, it

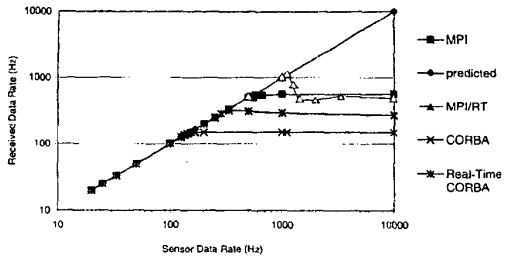

Fig. 13. Handling load stress by MPI, MPI/RT, CORBA and RT-CORBA.

turns out that a much more rigid communication structure is necessary for the whole design, to ensure timeliness and predictability. Traditionally, communication in distributed applications is done via sockets or remote procedure calls (method invocation). This is very inadequate for distributed real-time applications.

Therefore two implementation level standards for distributed real-time communication were studied: MPI/RT and Real-Time CORBA. A simple real-time benchmark, to handle load stress from the sensors, was designed and run under four stardards: MPI (mpich), CORBA (Visibroker), MPI/RT (from Mississippi State) and RT-CORBA/TAO (Washington Univ., St. Louis). Performance results for handling maximum communication load by these tools are presented in Fig. 13 [11] (for a network of Sun Ultra 2's running Solaris 2.6). It is evident that MPI shows general performance superiority over CORBA, however, with much less flexibility for components creation.

An additional study with the same benchmark was done to investigate meeting deadlines. When a task is busy with collecting data from sensors, it may not be able to respond on time to other needs for computation. This situation was simulated by requesting a collecting task work in 5 sec. intervals to gather 100 sensor data items (reasonable for ATCS) and record the following (Table 1):

Table 2. Comparison of deadline behavior.

Platform	SD misses	HD misses
Java sockets	4	1098 ms
C sockets	2	427 ms
CORBA/TAO	0	4 ms
Vgenic CORBA	1	778 ms
CORBA/IIOP	4	494 ms

- the number of times a 5.1 sec. soft deadline was

missed (SD misses)

- the total amount of time a hard 5.0 sec. deadline was missed (HD misses).

All experiments were run in Java under Solaris 2.6 (on Ethernet), except of C sockets, which were run for VxWorks.

In summary, the results of experiments prove that for real-time components to work properly the design phase has to take into consideration the real-time implementation standards (such as MPI/RT and RT-CORBA) for the distributed target platforms. Including implementation related design patterns into design-level description will greatly simplify the process of designing real-time components.

8 Conclusion

We tried to provide evidence that there is a clear template for real-time software architectures and design patterns, historically rooted in control engineering, that allows designers to build software components for complicated real-time systems using commercial off-the-shelf tools.

The development process consists of several steps, including: the precise definition of signals interfacing with the environment, description of required actions on all these signals, and building respective structural and behavioral diagrams, with the use of automatic tools, if necessary.

Applying these concepts and tools to a complicated air traffic control system confirmed this view and revealed the directions of additional work needed to enhance timing analysis at the design level and allow importing design components. Nevertheless, existing principles arm the developers of real-time software components in a set of invariants that they can successfully use in their development practice.

References

[1] A.H.M. AlMazid, *Engineering Analysis of Object-Oriented Software Development Tools for Distributed Real-Time Systems*, M.Sc. Thesis, Univ. of Central Florida, Orlando, Fla., 2000

[2] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley, Reading, Mass., 1998

[3] M. Boasson, Control Systems Software, *IEEE Trans. on Automatic Control*, 38(7):1094–1106, July 1993

[4] B.P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML*, Addison-Wesley, Reading, Mass., 1999

[5] K. Gaspar, B. Franek, J. Schwarz, Architecture of a Distributed Real-Time System to Control Large High-Energy Physics Experiments, *Parallel and Distributed Computing Practices*, 2(1):103–114, March 1999

[6] G.T. Heineman, W.T. Councill (Eds.), *Components-Based Software Engineering*, Addison-Wesley, Boston, Mass., 2001

[7] C. Muñoz, J. Zalewski, Architecture and Performance of Java-Based Distributed Object Models, *Real-Time Systems Journal*, 21(1/2):43–76, July 2001

[8] H. Pedroza, *GUI Builder for Real-Time Distributed Object Models*, M.Sc. Thesis, University of Central Florida, Orlando, Fla., 1999

[9] M.T. Pozesky, M.K. Mann, The US Air Traffic Control System Architecture, *Proc. of the IEEE*, 77(11):1605–1617, November 1989

[10] B. Selic, G. Gullekson, P.T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley and Sons, New York, 1994

[11] S Su, *Benchmarking Distributed Real-Time Applications*, M.Sc. Thesis, University of Central Florida, Orlando, Fla., 2000

[12] J. Zalewski, Real-Time Software Architectures and Design Patterns: Fundamental Concepts and Their Consequences, *Annual Reviews in Control*, 25(1):133–146, July 2001