A real-time systems context for the framework for information systems architecture

by D. J. Schoch
P. A. Laplante

In this paper we review the framework for information systems architecture first introduced by Zachman[1] and show how it can be applied in the context of real-time systems. Discussions are included throughout the paper to convey some of the characteristics unique to real-time systems and to point out areas of special architectural concern.

In the past, when the depth and breadth of computer applications were dramatically limited by hardware technology, system architecture was simple and straightforward. Today, business information systems are driven increasingly by high-level business strategies, instead of smaller-scale functional processes. Rather than just mechanizing a manual procedure, company-wide systems are moving and managing information that is rapidly becoming the actual infrastructure of the business. Furthermore, new technologies and increasing information demands by managers have taken business systems from batch processing and time sharing to the domain of high-performance, real-time systems.

An overall framework to better link systems to businesses and at the same time guide the development of these systems has been recognized by many.[1-3] In other words, a strict discipline of construction—or architecture—must be known and followed. In 1987 John Zachman proposed a framework for information systems architecture,[1] within which all aspects of information systems architecture (ISA) are depicted, from high-level business strategies to system coding. The ISA logical construct serves to define the interfaces and integration of the various system components.

In this paper we:

- Review and examine the Zachman framework within the context of real-time systems, which are increasingly characterizing today's business decision systems
- Interpret this ISA framework so as to apply it specifically to real-time systems architecture (RTSA)
- Present critical areas of architectural concern unique to real-time systems by using several

0018-8670/95/$3.00 © 1995 IBM

Table 1 Architectural representations produced over the process of building a complex engineering project

Representation in Architecture	Nature/Purpose	Generic "View"	Representation in Information Systems	Nature/Purpose
Bubble charts	Basic concepts for building Gross sizing, shape, spatial relationships Architect/owner mutual understanding Initiation of project	Ballpark	Scope/objectives	Strategic direction/focus of the business Product/service focus System concept/white paper Initiation of project
Architect's drawings	Final building as seen by the owner Floor plans, cutaways, pictures Architect/owner agreement on building Establishment of contract	Owner's view	Model of the business	Organizational or functional structure Policies, methods, and procedures for the business processes Requirements document Specifications agreement
Architect's plans	Final building as seen by the designer Translation of owner's view into a product Detailed drawings Basis for negotiation with general contractor	Designer's view	Model of the information system	Translation of business manager's view into an information system design Logical design document Logical representation of system to be built
Contractor's plans	Architect's plans constrained by laws of nature and available technology "How to build" description Direction of construction activities	Builder's view	Technology model	Logical system design constrained by physical technology Depiction of program modules to be written Direction of overall programming activities
Shop plans	Subcontractor's design of a part/section Detailed stand-alone model Specification of what is to be constructed Pattern	Out-of-context representation	Detailed description	Individual program design Program code, database description, networking details Direction of individual programmer activities
	(Not used in architecture, but used frequently in manufacturing where computer-controlled equipment uses this to produce some part of the product)	Machine language representation	Machine language description	Object code Used by computer itself
Building	Physical building	Actual product	Information system	Physical system

Adapted from J. A. Zachman, "A Framework for Information Systems Architecture"[1]

discussions about important aspects of real-time systems
- Provide a cursory analysis of several current products and trends that fit into the RTSA framework
- Draw some conclusions regarding current and future needs within the real-time systems development process

Throughout this paper the term *information systems* refers to those computerized systems designed primarily to support a business or business

Table 2 Different types of descriptions for the same product

	Description I	Description II	Description III
Orientation	Material	Function	Location
Focus	Structure	Transform	Flow/connection
Description	WHAT the item is made of	HOW the item works	WHERE the connections exist
Example	Bill-of-materials	Functional specifications	Engineering drawings
Descriptive model	Part-relationship-part	Input-process-output	Site-link-site
I/S analog	Data model	Process model	Network model
I/S descriptive model	Entity-relationship-entity	Input-process-output	Node-line-node

Adapted from J. A. Zachman, "A Framework for Information Systems Architecture"[1]

process. The most commonly understood of these is the management information system (MIS), which is "a computer-based system that makes information available to managers with similar needs."[2] Usually these systems collect and process data in order to support management decisions. The term *real-time systems* is used herein when referring to computerized systems whose correctness depends not only on logical correctness but on the timeliness of output. Real-time systems appear in virtually every computer application, but especially in avionics, robotics, process control, and simulation, and increasingly in management information systems. In practical situations, the main difference between real-time and non-real-time systems is an emphasis on response time prediction and its reduction.

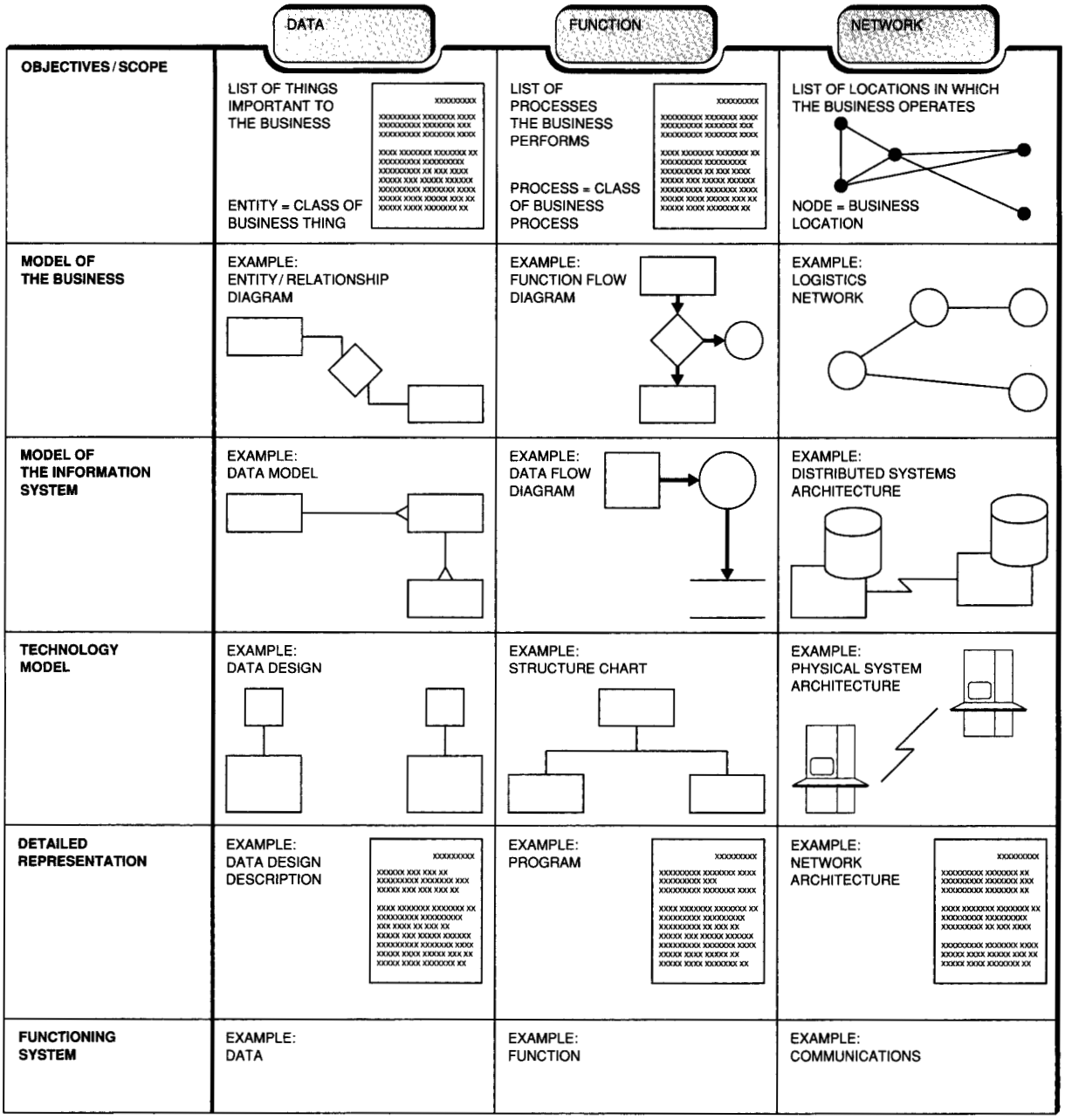
A framework for information systems architecture. In defining an "information systems architecture," Zachman used the field of classical architecture itself as an objective, independent basis upon which to develop a framework for discussion. Using the definition of deliverables within that field led him to the specification of analogous information systems architectural products and, in so doing, helped him to classify concepts that produced this "framework" in which to represent information systems architecture.

The framework he developed in this process addresses the different views of the various participants involved in each stage (e.g., owner's view, architect's or designer's view, builder's view), along with the discovery that the same product can—and must—be described in different ways (e.g., material: what the item is made of; function: how the item works; and location: where the flows or connections exist). The resulting framework is a two-dimensional matrix that presents different architectural representations of the product.

In Zachman's architectural framework, the *rows* represent the perspectives of the different participants in the architectural representations, as depicted in Table 1. These views and representations of the same product by the various players in the process are different (in nature, content, semantics, and so forth), not merely a set of representations varying in detail. Each of these representations is perfectly valid and necessary for the development of the product, with the key being the transformations from one representation to the next. The fourth column of Table 1, which we have added to Zachman's earlier work for comparison, presents the analogous architectural representations for information systems.

A second idea needs to be presented for the *columns* of the framework matrix—different types of descriptions exist for the same product. For example, in manufacturing, a bill-of-materials describes what a product is made of, the functional specifications describe how the product works, and engineering drawings show where the connections exist. Each description is required: looking at a list of parts tells nothing about what the part does or how it relates to other parts. Similarly, functional specifications say nothing explicit about the parts that make up the product or how it is constructed. Table 2 shows how different descriptions can be used for different things, including the analogs for information systems. Notice how each description has been prepared for a different reason and purpose. Each one stands alone and is distinct from the others, even though all the descriptions pertain to the same object and therefore are directly related to one another.

Figure 1 Zachman's framework for information systems architecture



Based on Figure 1 in Sowa and Zachman, *IBM Systems Journal* **31**, No. 3, 1992.

The information systems architecture. By combining the notions of different descriptions and different participants from the architectural framework and applying the method to information systems, Zachman formed the information systems architecture framework, shown in Figure 1. Notice how this figure is essentially Table 1 but drawn using information systems analogs, and

that the columns "Data," "Function," and "Network" represent different descriptions of the system.

Interpreting the Zachman framework in the context of real-time systems

Like information systems, real-time systems are increasingly complex and difficult to build and maintain. Zachman's framework for information systems architecture offers an opportunity to structure the development of real-time systems. Before proceeding to this development, we need to discuss a few unique aspects of real-time systems.

An overview of real-time systems. Consider a software system in which inputs represent digital data from hardware devices or other software systems, and outputs are digital data that control external hardware. The time between the presentation of a set of inputs and the appearance of all the associated outputs is called the *response time*. In hard real-time systems, response times must be explicitly bounded, or the system is considered to have failed. Notice that response times of, for example, microseconds are not needed to characterize a real-time system; it simply must have response times that are constrained and thus predictable. Most of the literature also mentions soft real-time systems—those in which performance is degraded but not destroyed when response time constraints are not met—and even firm real-time systems in which a few missed deadlines can be tolerated. (In real-time systems, one type of fault tolerance includes design choices that transform hard real-time deadlines into firm or soft ones.) Throughout this paper, however, we use the term "real time" to mean "hard real time."

Real-time systems are often reactive or embedded systems, or both. Reactive systems are those that have ongoing interaction with their environment, such as a fire-control system that reacts to a pilot's commands. Embedded systems are those used to control specialized hardware, which completely encapsulate the software. For example, an automatic teller machine is embedded and reactive.

An important concept in real-time systems is the notion of an event, that is, any occurrence that results in a change in the sequential flow of program execution. Events can be divided into two categories: synchronous and asynchronous. Synchronous events occur at predictable times such as execution of a branch instruction or hardware trap. Asynchronous events occur at unpredictable points in the flow-of-control and are usually caused by external sources such as a clock signal. Both types of events can be signaled to the CPU by hardware interrupts.

There is an inherent delay between when an interrupt occurs and when the CPU begins reacting to it, called the *interrupt latency*. Interrupt latency is caused by both hardware and software factors. Interrupts may occur periodically (at fixed rates), aperiodically, or both. Tasks driven by interrupts that occur aperiodically are called *sporadic tasks*. Systems in which interrupts occur only at fixed frequencies are called *fixed-rate systems* and those with interrupts occurring sporadically are called *sporadic systems*. In round-robin systems, each task is assigned a fixed-time quantum in which to execute. A clock is used to initiate an interrupt at a rate corresponding to the time quantum. Each task executes until it completes or its time quantum expires as indicated by the clock interrupt. When the time quantum of a task expires, a snapshot of the machine must be saved so that the task can be resumed later. A higher-priority task is said to preempt a lower-priority task if it interrupts the lower-priority task; that is, a lower-priority task is running when the higher-priority task signals that it is about to begin. As with the round-robin system, a snapshot of the machine must be saved so that the lower-priority task can be resumed when the higher-priority task has finished. Systems that use preemption schemes instead of round-robin or first-come-first-served scheduling are called *preemptive priority systems*. The priorities assigned to each interrupt are based on the urgency of the task associated with that interrupt. Preemptive priority schemes have the associated problem of hogging of resources by higher-priority tasks. In this case, the lower-priority tasks are said to be facing starvation. There are other, nonpreemptive, priority scheduling schemes, but these are of less interest to us.

Prioritized interrupts can be either "fixed priority" or "dynamic priority." Fixed priority systems are less flexible in that the task priorities cannot be changed once the system is implemented. Dynamic priority systems can allow the priorities of tasks to change during program execution—a feature that is particularly important

in threat management systems. In a special class of fixed-rate, preemptive priority, interrupt-driven systems called *rate-monotonic systems*, priorities are assigned so that the higher the execution frequency, the higher the priority. This scheme is common in embedded applications, particularly avionics systems.

A term often used as a measurement of real-time system performance is *time-loading*, or CPU utilization, and is a measure of the percentage of nonidle processing. A system is said to be "time-overloaded" if it is 100 percent or more time-loaded.

Special difficulties in real-time systems development. At least five considerations are crucial in the development of real-time systems. We now examine them. They are:

- Temporal behavior
- Multitasking
- Intertask communication and synchronization
- Object code efficiency and performance
- System verification

Each of these considerations relates to *cell(s)* in the ISA framework. We need to reinterpret the framework, then, in order to accommodate these considerations.

Temporal behavior. In real-time systems, bounding response times, and thus predicting them, are the most important considerations. Although response times are not often perceived as a major issue in information systems, modern applications such as program trading are highly time-critical because they support devices such as network interfaces, Quotron boxes, or high-speed modems. Moreover, most managers would argue that there is an intangible valuation function associated with the temporal "freshness" of any piece of information involved in a business decision.

Since all hard real-time systems have scheduling, timing, response time, and deadline concerns, the design, depiction, and implementation of this behavior is mandatory in any architectural structure guiding the life cycle of the real-time system. Indeed, the ability to predict timing behaviors for systems under development is considered by many to be one of the most critical issues facing real-time systems today.[4,5] Such prediction and verification is known as "schedulability analysis."[6]

The basic idea presented by Zachman regarding time (a "when" description column) was that it need not be included because the processes could be described via "snapshots" in time and because of the lack of formalisms needed to depict the cells in such a column. Although it is even more difficult for real-time systems to specify temporal behavior (and there is no agreement on how to do it anyway), this "Time" (when) column cannot be omitted since the correctness of the real-time system is based on its satisfaction of explicit temporal behavior.

Multitasking. Although multitasking is implicitly supported in mainframe computers, minicomputers, and even personal computers, for many embedded and reactive real-time applications, temporal performance prediction is impossible or unacceptably imprecise. Hence, in many time-critical information systems applications, streamlined and predictable multitasking systems need to be constructed. There are three types of multitasking:

1. Cooperative schemes that do not require interrupts (such as polled loops, cooperative multitasking, and state-driven code)
2. Preemptive priority multitasking
3. Round-robin multitasking

There are also hybrids of these three. We talk only about the second type, since the others can be modeled as special cases of it. Typically, preemptive priority multitasking allows higher-priority tasks to preempt lower-priority tasks and permits low-priority or background tasks to easily "slip in between" regularly scheduled or interrupt-driven high-priority tasks and execute. These foreground-background systems are the most common solution for embedded applications. They are an improvement over the interrupt-only systems in that the jump-to-self is replaced by non-time-critical code, called the background. The interrupt-driven processes are called the foreground. The background task is fully preemptable by any foreground task and, in a sense, represents the lowest-priority task in the system.

It is common to increment a counter in the background to provide a measure of time-loading or

to detect whether any foreground process has locked. For example, a counter is provided for each of the foreground processes and is reset by its respective process. If the background detects one of the counters as not being reset, the corresponding task is assumed to be locked, and a failure can be indicated. Certain types of low-priority self-testing can also be performed in the background. Other potential background tasks include low-priority display updates, logging to printers, and other actions that interface to slow devices.

Basically, preemptive priority multitasking provides for interrupt-driven systems to be written and numerous tasks to be run concurrently. A "Time" column is needed to depict this multitasking facet of real-time systems. In addition, "Data" cells play a major role in managing these data structures so that the necessary bookkeeping can be performed by the dispatcher of the operating system. At the same time, the various functions of the dispatcher would be identified from a "Function" column. A conceptual view (Objectives/Scope) will probably address some of these needs, but as development moves into the design and construction of the system, depiction of this view becomes crucial, further justifying the usefulness of all three columns ("Data," "Function," "Time"). Also, since the actual machine language representation is necessary for the successful implementation of a reliable and predictable interrupt scheme, a "Machine Representation" view is inserted into the RTSA framework. Indeed, recent work with compiler optimization stresses the need to rearrange the object code in order to improve and monitor timing behavior.[7]

Intertask communication and synchronization. The communication of data between processes and the synchronization of tasks are two more areas critical to real-time systems. Several techniques can help implement reliable methods to handle these issues, such as ring buffers, semaphores, test-and-set instructions, mailboxes, and event flags. Some of these techniques (e.g., test-and-set instructions) may relate to the machine language and hardware architecture of the target computer, again supporting the need to include "Machine Representation" in the RTSA framework.

Intertask communication can be implemented using a variety of methods. A more detailed discussion of these approaches will be helpful in order to address these critical real-time systems architectural concerns. For polled-loop systems, where the polling and event processing code run in mutual exclusion, intertask communication and synchronization services are not obligatory. With coroutines, synchronization and communication are built into the code, although protection of shared resources is not. But for foreground-background systems and operating systems based on the task control block model, these services are essential. Intertask communication and synchronization concerns generally arise during the logical design and coding implementation of the system. The "Time" cells residing in the "designer's" view and especially in the "builder's" view should address these issues. The "owner's" specifications may address some of these issues, but as the system is broken down into program modules and the intertask relationships are defined, these communication and synchronization aspects become more apparent. (Note that these communications are between software modules and usually unrelated to geographic or geometric configuration, which are more of a concern with geographically dispersed information systems and the physical manufacturing of products.)

The communication of data between processes and the synchronization of tasks are critical to real-time systems.

Another method of handling synchronization constraints is via an off-line pre-run-time scheduling algorithm, an approach that can significantly reduce the resources needed for run-time scheduling and context switching.[8] This approach also fits into the "Time" cells, but at the "builder's" or "subcontractor's" view. Interestingly, recent work utilizing an extended entity relationship database to support an object-oriented programming and execution system addresses the related issue of concurrency at a higher level.[9] This ap-

proach quite unexpectedly tends to bring the issues of synchronization and concurrency into the "Data" column in the RTSA framework as well.

Object code efficiency and performance. Because of the critical time-related demands of real-time systems, the quality of the object code generated by the compiler is a concern from at least two perspectives:

- Efficiency, with respect to size, speed, and overall performance
- Ability to predict and monitor execution time

Although there are formal methods for the determination of such performance factors as response time and time-loading, these methods generally are applicable in extremely restricted situations or in theoretical studies only. In most settings, performance analysis is done using logic analyzers, simulators, or "back-of-the-envelope" calculations.

The best method for measuring the execution time of any piece of code is to use a logic analyzer. One advantage of its use is that hardware latencies and other delays not due simply to instruction execution times are taken into account. The drawback of the logic analyzer is that the system must be completely (or partially) coded and the target hardware available. Hence, the logic analyzer is usually only employed in the late stages of the coding phase, in the testing phase, and especially during system integration. When a logic analyzer is not available, the code execution time can be estimated by examining the compiler output and counting macroinstructions. This technique also requires that the code be written, that an approximation of the final code exists, or that similar systems are available for analysis. The approach simply involves tracing the worst case path through the code, counting the macroinstructions along the way, and adding their execution times. These times can be found in the manufacturer's specifications or through measurement with a logic analyzer.

Another accurate method of code execution timing uses the system clock, which is read before and after executing code. The time difference can then be measured to determine the actual time of execution. This technique, however, is only viable when the code to be timed is large relative to the code that reads the clock.

Time-loading requirements are specific design goals because they affect hardware selection and overall system performance. Several methods can be used to predict or measure the code execution times that are needed in the calculation of time-loading. These techniques can also be used to calculate the context switch or software scheduling time for any interrupt handler.

Time-loading estimates are measures that are meaningful primarily in cyclic real-time systems. In polled loops, the figure is the relative percentage of time spent processing an event compared to the time spent checking the flag. In state-driven or cooperative multitasking systems, the measure is the time spent in the dispatcher when no processes need to run. In interrupt-driven systems, calculation of time-loading from measured data cannot be accurately computed for any type of system.

Reducing execution times. Identifying wasteful computation is crucial in reducing response times and time-loading. Many approaches used in compiler optimization can be used (see Reference 10 for a summary of these), but other methods have evolved that are specifically oriented toward real-time systems, and we discuss those methods here. For example, in most computers, integer operations are faster than floating point operations. We can exploit this fact by converting floating point algorithms into scaled integer algorithms. In such a scheme, the least significant bit (LSB) of an integer variable is assigned a real number scale factor. Scaled numbers can be added and subtracted together and multiplied and divided by a constant (but not another scaled number). The results are converted to floating point output only at the last step—a process that can save considerable time.

These concerns relating to object code efficiency and performance analysis resulted in the insertion of a "Machine Representation" view into the RTSA framework—how the computer hardware architecture "views" the product. In other words, what the real-time system looks like when the computer is actually executing the code. The needs for this view were noted above with respect to the other function-specific "difficulties" discussed, but the quality of the object code—from both the efficiency and measurability perspectives—must be included as a unique area of the architecture.

System verification. In order to increase system reliability, rigorous testing of the real-time system is required. Since testing can only detect the presence of errors and not the absence of them, the goal of testing must be to ensure that the software meets its requirements.[11] To this end, system verification answers the question, "Are we building the system right?" Because of the importance of system verification in connection with the development of real-time systems, a new column, titled "Verification," has been added to the RTSA framework. Interestingly, system validation (see DeMarco[12]) answers the question, "Are we building the right system?" This validation falls into a "why" column, addressing the concept of motivation as presented in Zachman's second paper on his framework[13] (with Sowa). Using the building construction analog, system verification and testing is equivalent to the building codes established to ensure the safety and reliability of the building. Also equivalent are quality assurance inspections, designed to verify that the manner in which the building is being built adheres to established methodologies and techniques. The building inspector checks to see whether the building is "up to code" at various stages during the construction, meeting all applicable requirements. Similarly, a comprehensive system test plan verifies that the deliverable by each participant meets the original requirement. The "rules of the framework" presented by Sowa and Zachman[13] are an appropriate test to ensure that this new column adheres to the construct of the original framework:

Rule 1. The columns have no order. The placement or priority of the "Verification" column is irrelevant to the other columns. A full and complete test plan could be developed either before or after the work on the other columns has been completed.
Rule 2. Each column has a simple, basic model. The simple, basic model for the "Verification" column could be: performance-verification-feedback; therefore, this rule is also met.
Rule 3. The basic model of each column must be unique. Again, the entity and connector in the basic, columnar model for this new column, performance and verification, are not repeated from another column. And the basic model itself (performance-verification-feedback) is unique as well. Due to the limited scope of the extensions suggested here for the RTSA framework, the following remaining rules continue to be true.
Rule 4. Each row represents a distinct, unique perspective.
Rule 5. Each cell is unique.
Rule 6. The composite or integration of all cell models in one row constitutes a complete model from the perspective of that row.
Rule 7. The logic is recursive.

A real-time systems context for the framework for information systems architecture

In his paper, Zachman points out that since his "descriptions for the same product" answer the three questions "what," "how," and "where," it is only logical that there must be at least "who," "when," and "why" descriptions also. It is the "when" dimension that is critical to the architectural framework of real-time systems—and in fact more important than the "where," at least with respect to real-time systems. In the realm of real-time systems, any geographically distant communication is generally handled at a higher level within the operating system and is probably transparent to the real-time system application software. As noted above, Zachman's "Network" column relates to the question of "where" in the sense of a geographic dimension, not with respect to intertask communications inherent in all real-time systems. This intertask communication is more concerned with event timings and task synchronization than with communications protocols and session management. In simpler terms, real-time systems are focused on brief, time-bound "handshakes" rather than with prolonged "conversations" which must take place irrespective of geographic proximity. Of course, as it relates to the geographic and certainly the geometric description of the hardware used in the real-time system, this dimension remains important; however, this discussion is left for future work.

One fundamental modification to the ISA framework in the development of the RTSA framework became necessary, that is, the addition of a new column. Because of the extreme criticality at every point in the development of the real-time system, "Verification" is introduced as an additional column in the RTSA framework and as an addi-

Figure 2 A framework for real-time systems architecture

	DATA • ENTITY • RELATIONSHIP	FUNCTION • FUNCTION • ARG	TIME • TIME • CYCLE	VERIFICATION • PERFORMANCE • VERIFICATION
1 OBJECTIVES/ SCOPE	LIST OF THINGS IMPORTANT TO THE PHYSICAL SYSTEM	LIST OF PROCESSES THE PHYSICAL SYSTEM PERFORMS	LIST OF TIME-CRITICAL BEHAVIORS IN THE PHYSICAL SYSTEM	LIST OF KEY TESTS TO DETERMINE IF WE ARE BUILDING THE SYSTEM RIGHT
2 MODEL OF THE PHYSICAL SYSTEM	EXAMPLE: ENTITY/ RELATIONSHIP DIAGRAM	EXAMPLE: FUNCTION FLOW DIAGRAM	EXAMPLE: ENGLISH DESCRIPTION OF THE TIMING AND SYNCHRONIZATION ASPECTS OF THE PHYSICAL PROCESS	EXAMPLE: RESPONSE TIME DEFINITIONS
3 MODEL OF THE REAL-TIME SYSTEM	EXAMPLE: DATA MODEL	EXAMPLE: DATA FLOW DIAGRAM	EXAMPLE: HIGH-LEVEL STATE TRANSITIONS, TIME-DEPENDENCY DIAGRAMS, PSEUDOCODE, PROGRAMMING DESIGN LANGUAGES	EXAMPLE: FORMAL VERIFICATION OF METHODS/TOOLS, PROTOTYPING, BLACK BOX TEST PLAN
4 TECHNOLOGY MODEL	EXAMPLE: DATA DESIGN	EXAMPLE: STRUCTURE CHART	EXAMPLE: FINITE STATES AUTOMATA, STATE CHARTS, PETRI NETS	EXAMPLE: TECHNICAL TEST PLAN, WHITE BOX TEST PLAN
5 DETAILED REPRESENTATION	EXAMPLE: DATA DESIGN DESCRIPTION	EXAMPLE: PROGRAM CODE	EXAMPLE: PROGRAM CODE USING LANGUAGES CAPABLE OF REPRESENTING TIMING, SYNCHRONIZATION, AND CONCURRENCY (SUCH AS ADA AND MODULA-2)	EXAMPLE: WALK-THROUGHS, CODE INSPECTIONS
6 MACHINE REPRESENTATION	EXAMPLE: RAW DATA	EXAMPLE: MACHINE CODE	EXAMPLE: MAILBOXES, SEMAPHORES, INTERRUPT HANDLERS	EXAMPLE: SYSTEM PERFORMANCE
FUNCTIONING SYSTEM	EXAMPLE: AVAILABLE DATA	EXAMPLE: SYSTEM FUNCTIONALITY	EXAMPLE: TEMPORAL BEHAVIOR	EXAMPLE: BEHAVIOR VIS-A-VIS SPECIFICATIONS

tional description of the real-time system. These concepts lead us to propose a reinterpretation to Zachman's framework resulting in a "framework for real-time systems architecture." After presenting these changes, a brief examination of a few recent products and trends follows to show how they fit into and support this framework concept as it applies to real-time systems. This will serve to illustrate the applicability of the RTSA framework.

Key points for a real-time systems architectural framework. In order to develop a framework for real-time systems architecture (shown in Figure 2), the following key points summarize the reinterpretation of Zachman's work:

- The "Network" description column has been replaced by "Time." As noted earlier, because of the critical nature of the temporal behavior of real-time systems, the "Time" description of every participant's view must be described. Since common local and wide area networking and remote communications would probably be handled at the higher, operating system level, we feel justified in relabeling the "Network" column.
- "Machine Representation" has been inserted

Table 3 Different types of descriptions for the same product—as defined in RTSA framework

	Description I	Description II	Description III	Description IV
Orientation	Material	Function	Time	Verification
Focus	Structure	Transform	Dynamics	Testing/assurance
Description	WHAT the item is made of	HOW the item works	WHEN the events take place	Are we building the product RIGHT?
Example	Bill-of-materials	Functional specifications	Production schedule	Inspector's checklist
Descriptive model	Part-relationship-part	Input-process-output	Event-cycle-event	Construction-inspection-report
RT/S analog	Data model	Process model	Response time	Verification test plan
RT/S descriptive model	Entity-relationship-entity	Input-process-output	Event-cycle-event	Performance-verification-feedback

as another perspective, or view, of the system. Since the interrupt scheme, machine language implementation (to ensure test-and-set instruction availability, for example), performance predictability, and execution time are important to a real-time system, the "Machine Representation" row was added to the framework.

- Another column has been added to the RTSA framework to explicitly address the description of the real-time system in the context of system verification through testing. This column is referred to as "Verification."
- Various minor rewordings have been made to translate the information systems language to a real-time systems analog. Also, row numbers were added to facilitate references to the framework in this paper. Since the rows of the ISA framework mirror the system life cycle and remain virtually unchanged in the RTSA framework, no further discussion is warranted. However, since the columns have been significantly altered, Table 3 repeats Table 2 to further explain and support the four descriptions of the real-time system.

Implications to real-time systems development. The following subsections discuss how Zachman's framework, interpreted for real-time systems, can have significant benefits during the system definition and development process.

Provide solutions to critical real-time systems concerns. The rigorous structure and extensive integration that the RTSA framework brings to the design and development of real-time systems can help find solutions to problems of particular concern, such as timing, predictability, and deadlock. These types of problems are frequently caused by the complexity of the system. The RTSA framework provides a means to better plan and integrate the numerous design and implementation considerations.

This framework can help to develop an organization-specific architectural model to segregate and define the various areas that make up the overall architecture of a real-time system. Once an overall architectural model has been defined, it will contribute to a better understanding of design issues and the reasons for developing (or not developing) the various representations, and it will ensure that no aspect of the system is overlooked. For example, most well-known commercial operating systems are too bulky and all-purpose to be useful in real-time applications with stringent response time requirements. In addition, for custom computers, such as those used in many embedded applications, no commercial operating systems may be available. Hence, the real-time systems designer often must design a bare bones operating system or use one of the specialized real-time operating systems that are commercially available. Working with this framework for the architecture could provide a better design.

Categorize tools and techniques. A study of the various specifications and design techniques used in real-time systems makes it obvious that each technique has strengths and weaknesses that can often be confusing. This framework can classify the different methods of depicting the time and other dimensions of the real-time system, and therefore who in the development cycle would best utilize the tool or methodology. A detailed discussion of this topic can be found later in this paper.

Guide selection of tools and techniques. When the work is categorized into the cells of the framework, a system development group can use this understanding to select specific tools or techniques, or both, that best fit into the organizational structure of the group. For example, once a particular programming language is chosen, the tools that best support quick and accurate software construction using that language can be selected for the requirements analysis and design phases. When the roles, deliverables, and "hand-offs" of the group are defined, an implementation of CASE (computer-aided software engineering) tools may be possible as well.

Define deliverables and hand-offs for all participants. Once each cell in the framework is understood and the work that takes place within the cell is defined, all deliverables for that piece of the development process will be identified. When these deliverables are documented, the hand-offs between the cells are identified as well. This identification is critical to the project management of the development effort and provides measurable outputs for each participant.

Improve communications within individual project teams. The common understanding of the entire architectural framework by the "soup-to-nuts" project team will provide a common language for the architecture and improve dialog among team members. When given a firm baseline that can act as a reference point for the team, they cannot help but talk the same language and communicate better.

Facilitate communications within the real-time systems development community. Hopefully this framework will contribute to "establishing a stake in the ground" for real-time systems architecture—including requirements definition and the development structure—that can be referred to by members of the real-time systems community.

Analysis of current trends and products

This section offers a brief analysis of some methodologies and tools that support the architectural structure of the real-time systems environment. This cursory survey gives some significant examples from recent literature that fit into and help explain and support the real-time systems framework.

The Core method for real-time requirements. A 1992 paper by Faulk et al.[14] on the Core method addressed the issue of gathering requirements for real-time systems. The member companies of the Software Productivity Consortium develop large, mission-critical, real-time applications. They have identified requirements as the top priority problem in systems development. The board of directors of this group stated that "Requirements are incomplete, misunderstood, poorly defined, and change in ways that are difficult to manage." The Core (Consortium Requirements Engineering) method was developed to address this problem and is a single, coherent method for specifying real-time requirements.

Major features. This method uses the following techniques in its design:

- Integrates object-oriented and formal models
- Integrates graphical and formal specifications
- Permits nonalgorithmic specifications
- Provides a machine-like model

This method is focused solely on developing requirements for a real-time system with the following as some of its high-priority characteristics:

- Precise and testable system specifications
- Specifications that are easy to alter, and easily indicate ripple effects
- Comprehensible and practical presentation to all audiences
- Support for the representation of system bounds, interface, and context
- Definition of specifications allowed as a group of distinct and relatively independent parts
- Requirement that guidelines and examples of required input are included
- Definition of what makes a set of specifications congruous

Where it fits into the real-time systems framework. The Core method uses Stephen Mellor's real-time structured analysis approach[15] as one of the existing methods used in the meld of existing methods to form Core. Interestingly, this approach represents a system as a structure that can be viewed in three ways:

- Information—What information does the system use, and what are the relationships among pieces of information?

- Process—What are the functions of the the system, and what data and control information are exchanged among functions?
- Behavior pattern—What are the states of the system, and what events cause transitions among states?

To point out the obvious, these map directly to the "Data," "Function," and "Time" columns of the real-time framework shown in Figure 2. The Core method appears to be designed primarily to address the development of the owner's view (rows 1, 2, and possibly even 3 of the framework) of the requirements with much more structure than natural language. Without delving into its details, the Core methodology uses relatively nontechnical methods to capture the owner's and possibly the designer's perspectives. It provides the ability to capture requirements in a rigorous fashion, thereby enabling the designer, the builder, or both to directly transform these requirements to the next representation of the system and ultimately to the system itself.

Ready Systems' VRTXdesigner. In order to give programmers the ability to verify timing requirements, Ready Systems' VRTXdesigner provides for the verification of the underlying model by simulating the application, which must be running with the company's VRTX operating system's real-time scheduling mechanisms.[16] This product also provides programmers with the ability to monitor system response to stimulus, as well as concurrency, for the application skeleton.

Major features. VRTXdesigner is a top-down design tool, enabling users to graphically lay out their application modules with icons. Its simulated real-time operation gives the user the ability to check for conflicts, deadlocks, lockouts, starvations, processing bottlenecks, and timing requirements violations, including verification of critical timing deadlines, task preemption, and even CPU utilization. The application modules can utilize the capabilities of the VRTX operating system in the areas of scheduling and processing (including queues, mailboxes, semaphores, and event flags). The behavior of the external world can also be included in this simulation, either as a periodic function or statistically as a time distribution function. The simulation is a batch run and all results are saved. Users can then produce graphical time lines that show individual task execution. In later simulations, users can specify individual paths that they wish to monitor and subsequently see where the execution time is spent and how well the defined scheduling actually performs under real-time conditions with the VRTX OS kernel and the underlying hardware.

Where it fits into the real-time systems framework. This product fits primarily into the "Time" column, beginning in the design phase (rows 3 and 4). Although it does not actually assist in the initial design of the system, it certainly can be used to validate and improve the design. Its usefulness in verifying all timing aspects of the program code is invaluable and extends into an analysis of the execution time using machine language timings. It remains, however, a simulation tool, and in reality it does not fit into the architectural cells relating to the actual software construction (although depending on the development environment, it may). Fitting perfectly into the "Time" cell that relates to the design of the system, this product offers two extremely robust capabilities:

- Simulation of every aspect of the system design, even down to execution timings of individual tasks, thus allowing the designer to validate and improve the design
- Ability to inherently turn itself over to the actual programmer after the design is completed, before implementation and installation of the code begins

Unlike any popular programming language used in the development of information systems, this product clearly has a home in the framework for real-time systems architecture above the programming cells (row 5).

Dynamo: A time-based object-oriented model. Dynamo is another modeling process to include in a real-time software engineering architecture.[17] An object-oriented data model for real-time systems, Dynamo integrates time into the object-oriented model. The concept of time remains uniform across all aspects of the model, from object structure and behavior, to the execution model, to synchronization and concurrency control.

Major features. In Dynamo, a notion of quasi-real time is defined, which its authors claim keeps enough synchrony with real time to be meaningful, yet allows enough slack for the computer to do its work efficiently and reliably. Without prob-

Table 4 Real-time language requirements and where they fit into the real-time framework

Language Requirement	Framework Column	Rows in Addition to Detailed Representation View
Predictable execution time	Time, Verification	
Schedulability analysis	Time, Verification	Technology Model
Strong typing	Data	
Structured constructs	Data	
Modularity	All	Technology Model
Error handling	Function	Technology Model
Multiprogramming	Time	Technology Model
Process synchronization mechanisms	Time	Technology Model and Machine Representation
Ability to access hardware locations	All	Machine Representation
Direct interrupt handling	Function	Machine Representation
Language readability to allow for long-term maintenance	All	
Small, simple, and well-defined	All	Technology Model

ing the details of "quasi-real time" nor evaluating its validity, the authors give the following explanation of this view of time:[17] "If 'real-time' is the notion of everyday, human-oriented time, one may say that the computer-time is, by contrast, artificial, or 'virtual' time which abstracts some essential temporal relationships such as sequentiality of events. On the other hand, in a human-oriented interactive environment, real-time has the advantage of being more intuitive. Our effort to take advantage of the abstraction properties of virtual time while retaining a notion of real-time led to the notion of quasi-real time (qrt) . . . qrt is real-time with a built-in slack for accommodating events inside the computer that occur in unpredictable order and take up an unpredictable length of time. Since qrt is synchronized with the real-time clock at event boundaries, it is guaranteed to be 'close enough' to real-time." The authors explain that this concept of quasi-real time is one of the properties that their objects contain; the full list of properties follows:

- A unique object identifier
- A set of attributes
- A set of constraints
- A time stamp
- A quasi-real-time clock

In the object space, there can be several tuples with the same object identifier as long as they have different time stamps (containing different versions of the object in time). The quasi-real-time clock is used to link quasi-real time with real world time.

Where it fits into the real-time systems framework. Although Dynamo itself was not a full-fledged programming language at the time the paper by Bapa Rao et al.[17] was written, it clearly offers an object-oriented programming technique to be implemented in the software construction phase of the project. It therefore belongs in the "Technology Model" and "Detailed Representation" views (rows 4 and 5) of the framework. Since its purpose is to directly guide and contribute to the actual programming of the system, the user and logical designer would find little use for the model. The technique is object-oriented, so it already addresses both the "Data" and "Function" descriptions of the product. When the time dimension is added into Dynamo's object-oriented model, it begins to address all of the temporal behavior aspects of the system as well, and also covers the "Time" column.

Requirements of real-time programming languages. The impact of programming language constructs in the design and implementation of real-time systems is often overlooked. A thorough discussion of programming languages that were specifically designed for real-time use is beyond the scope of this paper. Some discussion of these topics can be found in Reference 18; the interested reader is also referred to References 19, 20, 21, and 22.

In Reference 23 Stoyenko sets forth requirements for a real-time programming language. In Table 4 each of these is related to the appropriate columns in the RTSA framework for real-time sys-

tems architecture; where appropriate, affected rows other than "Detailed Representation" (row 5—actual programming) are noted. As shown in this table, much of his programming language requirements relate to the temporal aspects of the programming construction. As also shown in Table 4, a significant majority of the requirements set forth by Stoyenko deal with the temporal behavior of the real-time systems architecture. Again, this framework helps to identify and categorize needs in this direction. A detailed case study using the Flex language (one of the languages included in Reference 22) presented the ability of the language to ensure temporal correctness as well as functional validity.[24] Its authors summarize their requirements of a real-time language as follows:

- Capacity to express different types of timing requirements
- Mechanism for run-time systems to enforce timing constraints
- Provision for ensuring the temporal correctness of the program

All three of the above language requirements relate to the temporal behavior of the real-time system, again supporting the "Time" aspect of the architecture of the real-time system. It also reaffirms this needed modification of the Zachman framework to make it applicable to the work of building real-time systems.

Specification and design techniques—where they fit into the framework. Many specification and design techniques are popular today, and each technique has strengths and weaknesses. The RTSA framework can help classify these methods of depicting the various dimensions of the real-time system, and therefore who in the development cycle would best utilize it and possibly which tools should be used for the development of that area of the architecture. Common methods include Petri nets, finite state automata (FSA), data flow diagrams, Warnier-Orr notation, structured English, and temporal logic. We cannot possibly survey these adequately here but the interested reader can consult the references. For example, References 25 and 26 provide a broad-based discussion of software specification, References 27 and 28 discuss the use of FSA, References 29 and 30 discuss Petri nets, References 31 and 32 discuss Warnier-Orr notation, References 12, 15, and 33 discuss data flow diagrams, References 24 and 27 discuss temporal logic, and Harel's state charts are described in References 34 and 35. Table 5 summarizes some of the more popular techniques, the advantages and disadvantages for each, and where they fit into the RTSA framework. For example, usually natural language description is the best way for the "owner" to represent requirements for the physical system. Pseudocode and programming design languages are slotted for the designer's view. Of course, one would not expect a programmer to utilize these tools as the sole method to implement the system. Rather, the deliverable from each of these tools would guide what is referred to here as the "logic builder" (possibly the programmer, possibly someone else) to next represent the requirements using techniques like finite state automata and Petri nets. Finally, these deliverables would be used by the programmer to actually "build" the program from this representation.

As shown, some of the techniques include the "Time" column. This would imply that in any requirements document for real-time systems, certain of these techniques must be utilized in order to include the time-related description of the system. These tools and techniques would be used to address system aspects such as timing, deadlock avoidance, and response time predictability. It should also be noted that some of the tools are applicable to rows 1 and 2 of the framework but only at higher levels of description. This is understandable but certainly not the purpose of the framework. That is, if the same tool or methodology can be used by the different participants in the development process (again: how the "owner" sees the product, how the "designer" designs it, and how the "builder" constructs it), all well and good. But according to the framework, these transitions are transformations from one representation to another, not simply the same representation with increased detail. Table 5 also points out the need of tools and methodologies at rows 1 and 2 of the framework to describe the "Time" aspects of the system. Human language and mathematical specifications can be used, but no other techniques apply to these early descriptions of the system. This ability to categorize and even assign tools and techniques to each area of the architecture is perhaps one of the most useful aspects of the framework. It can help guide choices of tools and methodologies, define deliverables and hand-offs, and serve as a model for the requirements and design efforts.

Table 5 Specification and design techniques

Technique	Advantages	Disadvantages	Framework Row/Column
Human language	Well known Can clarify descriptions	Ambiguous No code generation	All/All
Mathematical specification	Precise and unambiguous Promotes formal program-proving techniques Rigorous code optimization can be done	Can be cryptic Difficult to do Training in mathematical modeling not common Formal proofs error-prone	All/Data, Time, Verification
Flow chart	Widely used and understood Describes individual tasks well	Cannot depict multitasking Temporal behavior cannot be described Encourages GOTOs	Rows 1–4/Function, Verification
Structure chart	Widely used and understood Best for small/simple systems Clearly identifies function execution sequence Identifies recursion and repeated modules Encourages top-down design	Provides for no conditional branching Cannot describe concurrency or process interaction No way to show temporal behavior	Rows 1–4/Function
Pseudocode & programming design language	Better than using high-order language for specifications Close to a programming language Adaptable to formal program-proving techniques Some can handle concurrency	Still programming language in which user must be fluent Cost and maintenance of design tools can be high Errors can still be made in high-level abstractions	Rows 3–4/Data, Function, Time (depending on the tool), Verification
Finite state automaton	Widely used for state-driven systems Easy to develop Easy to generate code to implement Since based on mathematics, can be formally optimized Unambiguous Can depict concurrency	"Insideness" of modules cannot be shown No intertask communications Number of states can grow very large	Rows 3–4/Data (states only), Time, Verification
Data flow diagram	Widely used and understood Emphasizes flow of data De-emphasizes flow of control Useful in identifying concurrency Structure chart can be derived Can help partition system into hardware and software components	Difficult to show synchronization in flow	Rows 3–4/Data

System verification and testing. The crucial area of testing is part of most software development methodology and, therefore, must be included in an architectural framework as well. The behavior of the system must constantly be checked against the system requirements. The "Verification" column in the RTSA framework is designed to ensure that this need is met and that this test plan is robust and thorough enough to test the output of each stage, as well as the final product. Traditional testing methods can be generally applied to a real-time system, including black box and white

box testing. These techniques can be used by the unit author and by the independent test team to exercise each module and the overall system. The goal of these tests is to ensure that all system requirements, especially those concerning system response times, have been met. These techniques can also be applied at the subsystems and system level. For a more complete discussion of testing techniques see References 36, 37, or 38.

Summary and conclusions

This framework for real-time systems architecture can have the following uses in the world of real-time systems specification, design, development, implementation, and testing:

- Categorize and assign tools and techniques to each area of the architecture.
- Guide tool and methodology choices used by the various disciplines.
- Define deliverables for each contributor.
- Clarify hand-offs between participants.
- Assist in addressing problem areas of particular concern to real-time systems.
- Provide a structure in which a model can be developed for the entire software life cycle.
- Propose a baseline for discussion of systems architecture among the real-time systems community.
- Improve communications between participants and understanding of each other's individual architecture.

It is impossible to determine exactly what is missing; that is, which cells within the framework appear to be without popular tools, etc., in the real-time systems world. A thorough survey and analysis of existing tools, methodologies, simulation and performance evaluation environments, and programming languages with the purpose of "slotting" them into one of the cells of the framework would appear to contribute to the applicability of the real-time framework as presented. Undoubtedly this task would be exhausting. Unfortunately, the usefulness of the results would be questionable. Many would argue about the categorizations, and others would not be interested since so many of the products would be irrelevant to their environment. However, it would behoove any development organization to evaluate their own tools, etc., within the context of the RTSA framework for the purpose of identifying which areas of the framework are being addressed and which areas are not being addressed. In order to build a complex real-time system, an architectural framework such as that presented in this paper can serve to ensure accuracy and dependability. This work would also help the organization realize some of the benefits noted in the previous section, such as definition of deliverables, clarification of hand-offs and interfaces, and improved communications.

Once the "missing pieces" are identified (i.e., cells in the framework not included in the development methodology of an organization), how this impacts the systems architecture and ultimately the development of real-time systems must be decided on a case-by-case basis. One organization may consciously decide to omit or combine individual architectures, or work toward implementing a single methodology or mechanized tool, or both, to support the entire systems life-cycle architecture. Any approach is valid, and any approach can be successful. But the framework can support these conscious decisions and guide the individual model that is used for any real-time systems architecture.

During the development of this paper, three areas for future work became apparent:

1. Include the hardware architecture in the RTSA framework where needed in order to more fully support the tight integration of the software and hardware systems.
2. Develop an entire framework for the area of system testing. According to the seventh rule of the ISA framework, since the logic of the framework is recursive, it may be an extraordinary benefit to the testing and quality control communities to develop test plans for each of the perspectives (rows) and descriptions (columns) of the system architecture. In order to preserve most of the ISA framework when developing the RTSA framework, this concept was not pursued.
3. Integrate the "Verification" column into the ISA framework. Although the testing of real-time systems requires more rigor and detail than necessary for an information system, the architecture of the information system would benefit as well. It could even be argued that business rules or policies exist that should be captured in this column, rather than residing in one of the others.

As technology continues to climb the price-performance curve, real-time systems—along with information systems—will continue to grow in scope and complexity. The method of construction of these systems will continue to gain in importance. This issue of architecture must be addressed in order to build these complex engineering products called "real-time systems." The framework for real-time systems architecture as presented in this paper can provide this structure.

Cited references

1. J. A. Zachman, "A Framework for Information Systems Architecture," *IBM Systems Journal* **26**, No. 3, 276–292 (1987).
2. R. McLeod, Jr., *Management Information Systems: A Study of Computer-Based Information Systems*, Macmillan Publishing Company, New York (1990).
3. D. Connor, *Information Systems Specification and Design Road Map*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1985).
4. G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger, "Developing Real-Time Tasks with Predictable Timing," *IEEE Software* **9**, No. 5, 35–45 (September 1992).
5. C. J. Paul, A. Acharya, B. Black, and J. K. Strosnider, "Reducing Problem-solving Variance to Improve Predictability," *Communications of the ACM* **34**, No. 8, 81–93 (August 1991).
6. A. D. Stoyenko, V. C. Hamacher, and R. C. Holt, "Analyzing Hard-Real-Time Programs for Guaranteed Schedulability," *IEEE Transactions on Software Engineering* **17**, No. 8, 737–750 (August 1991).
7. P. Gopinath, T. Bihari, and R. Gupta, "Compiler Support for Object-Oriented Real-Time Software," *IEEE Software* **9**, No. 5, 45–51 (September 1992).
8. T. Shephard and J. A. M. Gagne, "A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems," *IEEE Transactions on Software Engineering* **17**, No. 7, 669–677 (July 1991).
9. P. Gopinath, R. Ramnath, and K. Schwan, "Data Base Design for Real-Time Adaptations," *The Journal of Systems Software* **17**, No. 2, 155–168 (February 1992).
10. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Co., Reading, MA (1986).
11. P. A. Laplante, *Real-Time Systems Design and Analysis*, IEEE Press, Piscataway, NJ (1992).
12. T. DeMarco, *Structured Analysis and System Specification*, Prentice-Hall/Yourdon, Englewood Cliffs, NJ (1978).
13. J. F. Sowa and J. A. Zachman, "Extending and Formalizing the Framework for Information Systems Architecture," *IBM Systems Journal* **31**, No. 3, 590–616 (1992).
14. S. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr., "The Core Method for Real-Time Requirements," *IEEE Software* **9**, No. 5, 22–34 (September 1992).
15. P. T. Ward and S. J. Mellor, *Structured Development for Real-Time Systems*, Vol. I, II, III, Prentice-Hall/Yourdon, Englewood Cliffs, NJ (1986).
16. R. Weiss, "Real-Time System Simulator," *Electronic Engineering Times* (September 10, 1990).
17. K. V. Bapa Rao, A. Gafni, and G. Raeder, "Dynamo: A Time-based Object-oriented Model to Support Distributed Collaborative Development," *Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering* (May 1990), pp. 61–69.
18. A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages*, Addison-Wesley Publishing Co., Reading, MA (1990).
19. J. R. Allard and L. B. Hawkinson, "Real-Time Programming in Common LISP," *Communications of the ACM*, **35**, No. 9, 64–69 (September 1991).
20. A. D. Stoyenko and E. Kligerman, "Real-Time Euclid: A Language for Reliable Real-Time Systems," *IEEE Transactions on Software Engineering* **SE-12**, No. 9, 940–949 (September 1986).
21. F. Boussinot and R. DeSimmi, "The ESTEREL Language," *Proceedings of the IEEE* **79**, No. 9, 1293–1304 (September 1991).
22. *Proceedings of the IEEE* **79**, No. 9 (September 1991).
23. A. D. Stoyenko, "The Evolution and State-of-the-Art of Real-Time Languages," *The Journal of Systems and Software* **18**, 61–84 (April 1992).
24. K. B. Kenny and K.-J. Lin, "Building Flexible Real-Time Systems Using the Flex Language," *Computer* **24**, No. 5, 70–78 (May 1991).
25. I. Sommerville, *Software Engineering, 4th Edition*, Addison-Wesley Publishing Co., Reading, MA (1992).
26. C. Ghezzi, J. Mehdi, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1991).
27. Y. S. Ramakrishna, P. M. Melliar-Smith, L. E. Moser, L. K. Dillon, G. Kutty, "Really Visual Temporal Reasoning," *Proceedings of the Real-time Systems Symposium* (1993), pp. 262–273.
28. E. M. Clark, Jr., D. E. Long, and K. McMillen, "A Language for Computational Specification and Verification of Finite State Hardware Controllers," *Proceedings of the IEEE* **79**, No. 9, 1283–1292 (September 1991).
29. W. B. Joerg, "A Subclass of Petri Nets as a Design Abstraction for Parallel Architectures," *ACM Computer Architecture News* **18**, No. 4, 67–75 (December 1990).
30. N. G. Leveson and J. L. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Transactions on Software Engineering* **13**, No. 3, 386–397 (March 1987).
31. K. Orr, *Structured System Development*, Yourdon Press, Englewood Cliffs, NJ (1977).
32. J. D. Warnier, *Logical Construction of Programs*, Van Nostrand Reinhold, New York (1974).
33. D. J. Hatley and I. A. Pribhai, *Strategies for Real-Time System Specification*, Dorset House, New York (1987).
34. D. Harel, "On Visual Formalisms," *Communications of the ACM* **31**, No. 5, 514–530 (May 1988).
35. D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering* **16**, No. 4, 403–414 (April 1990).
36. B. Hetzel, *The Complete Guide to Software Testing, 2nd Edition*, QED Information Sciences Inc., Wellesley, MA (1988).
37. W. E. Howden, "Life-Cycle Software Validation," *Software Life-Cycle Management*, Infotech, Maidenhead, England (1980), pp. 101–116.
38. G. J. Myers, *Reliable Software Through Composite Design*, Van Nostrand Reinhold, New York (1975).

Accepted for publication July 11, 1994.

Daniel J. Schoch *AT&T Corporation, 1 Speedwell Avenue East, Morristown, New Jersey 07962 (electronic mail: dschoch@attmail.att.com).* Mr. Schoch is manager of information technology in the corporate human resources division of AT&T. He holds B.S. and M.S. degrees in computer science and has 20 years experience in applying technical solutions to a variety of business and technical needs, using mainframe-, minicomputer-, and personal computer-based configurations. Prior to joining AT&T, Mr. Schoch worked with real-time systems for Control Data Corporation and was responsible for both systems and applications programming. He later joined Citibank, NA as a technical specialist overseeing a regional data center and related new applications development. In addition to several nontechnical staff assignments, his AT&T career covers a variety of technical areas: applications analysis and programming, corporate data standards development, local area network installation and management, end-user technical support, and client/server software development.

Phillip A. Laplante *Department of Mathematics and Computer Science, Fairleigh Dickinson University, Madison, New Jersey 07940 (electronic mail: laplante@sun490.fdu.edu).* Dr. Laplante is Associate Professor and Chair in the Department of Mathematics and Computer Science and a visiting research scientist at the Real-Time Computing Laboratory of the New Jersey Institute of Technology. He holds a Ph.D. in computer science and a Professional Engineering license in the state of New Jersey. He has over 10 years experience designing real-time systems and was the lead software engineer in the design and implementation of a new generation of inertial measurement systems for the space shuttle. He has taught courses in real-time design throughout the world and has published widely on real-time systems, computer-aided design, image processing, and software engineering.

Reprint Order No. G321-5556.