# Architecture for Predictable Systems

Mark Gerhardt
Doug Locke
TimeSys Corporation
Pittsburgh, PA
www.timesys.com

**Abstract**

*Predictable systems are systems in which correctness arguments consider both the appropriateness and the timeliness of delivered results. Such systems exhibit both statistical timing as well as bounded duration timing expectations. Such systems require suitable architectural techniques that do not preclude meeting the timing expectations. This paper provides a summary of some of these issues and the architectural concerns that surface when addressing timing constraints in performance critical systems. The currently well-known design methods are examined for their suitability to describe and record architectures with predictable performance. Relevant changes resulting in a suitable method for real-time object-oriented analysis to are suggested.*

## Introduction

A real-time system is one in which correctness depends on meeting time constraints. Correctness arguments, therefore, must reason about response time requirements as well as functional requirements. The timing requirements may be "Hard" duration limits between events or "Soft" statistical expectations. A real-time architecture must not ***preclude*** meeting response time constraints.

Object choices made while architecting a system always involve resource sharing and schedule contention that can ultimately result in timing failure. Many popular architectures and methodologies ***ignore*** response time concerns until it is too late to correct them economically.

## Levels of Timing Expectations

There are five categories of timing expectations. Each category provides a different degree of rigor regarding expected performance. More importantly, each category utilizes different infrastructure and communication techniques. The architectural structures and techniques appropriate for use in one of these categories are usually inappropriate for the others. The categories are presented in order of increasing knowledge about the internal mechanics of implementation. Note that both of the last two categories can guarantee bounded response time, but they have very different costs and fault characteristics.

- **Measured after the fact** – (quantitative indications) Quantitative measurements may or may not be repeatable. Ad hoc quantification and measurement is extremely misleading. For example, measuring the temperature for 3 days does not adequately prepare for prediction of tomorrow's temperature.
- **Repeatable Measured** Knowledge about the context in which measurements are taken adds to the confidence by which extrapolation and prediction can be done. Knowing the latitude, time of day, and date of the temperature measurement give increasing confidence to extrapolation of the three temperature measurements for the fourth day. In the case of a measurement capturing the execution time of the software application, one must consider the application in a context of background processes including spoolers, message handlers, and garbage collectors for meaningful extrapolation.
- **Statistically Predictable Architecture** Statistical characterizations indicate average response time and related standard deviation. Architectural techniques employed in building statistically predictable systems include queuing (usually first-in-first-out), asynchronous messaging, and reactive event handlers. Analysis techniques that are applicable include discrete event simulation via use cases and the application of queuing theory.
- **Analytically Guaranteed Bounded Latency** Latency is the duration between the occurrence of an event and the completion of the associated system response to that event. Architectural techniques resulting in systems that exhibit guaranteed bounded latency include Shared Resources. Shared resources are entities that are "locked" by clients. As such, they exhibit protected regions of use. In conjunction with Fixed Priority Scheduling, "real-time" O/S Kernels permit the use of analytical techniques such as rate monotonic analysis and resource arbitration policies such as priority inheritance. Such systems may be analyzed and may therefore be guaranteed to possess upper bounded latency response times to specified stimuli.
- **Deterministic** A deterministic architecture provides a priori knowledge about every state that a system will pass through over time in response to a specific stimulus. Techniques used to build deterministic architectures include centralized frame based schedulers (cyclic executives) and statically limited language subsets (eliminating constructs and concepts such as exceptions and their propagation, allocation, dynamic polymorphism resolution, and dynamic thread creation). The SPARK approach is an ex-

ample of such an approach from Praxis in the United Kingdom.

## Real-Time Properties

There is a difference in how we discuss real-time systems from how we discuss more conventional time-sharing systems. Time sharing systems have implicit expectations about fairness and concepts such as "round robin" scheduling disciplines. Common concepts such as throughput are often confused with real-time concepts such as deadlines. Consider the terminology shown in Table 1:

| | Time Sharing Systems | Real-time Systems |
|---|---|---|
| **Capacity** | High Throughput | Schedulability |
| **Responsivness** | Fast Average Response | Ensured Worst-Case Latency |
| **Overload** | Fairness | Stability |

Table 1. Terminology about Timing Concepts

***Schedulability*** is the ability of tasks to meet all hard deadlines.
***Latency*** is the worst-case system response time to events.
***Stability*** in overload means the system meets critical deadlines even if all deadlines cannot be met.

## Current Design Practices and Processes

Current practices, processes, and tools use object subsystem abstractions that do functional encapsulation; i.e., the abstractions characterize services and interface-centric concerns. In order to successfully analyze a system's guaranteed performance and latency, the semantics of resource contention and usage not only must be categorized explicitly and made visible to potential clients, but this must be done surprisingly early in the development process.

Most popular object-oriented decomposition techniques and their accompanying graphical representations do not sufficiently address expression of concurrency, active objects, shared resources, and synchronization and contention semantics with sufficient precision to allow all architectural analysis. A simple example illustrates this point.

## Family Obligations – An Example

The idea of shared resources is familiar to most people. Consider the family car. When parents run errands, the errands must be sequenced because each errand requires the car. The inability to do an errand that a family member is ready to do, but cannot do because the car is unavailable, is regarded as ***blocking***. Most real-time systems fail to meet their performance deadlines because of excessive blocking rather than because of excessive CPU utilization. This has been well proven with numerous case histories.

Performance limitations from sharing a car are so well known that families purchase additional cars to improve performance. Consider therefore, a family with two cars. This family has three things to do: 1) purchase groceries, 2) take the kids to soccer, and 3) purchase hardware and fix the leaky toilet.

There are, therefore, three independent functional actions to perform, corresponding to numbers one through three above. The family has two cars and only two adults to drive the two cars. The family is also "socially correct" according to American suburban conventions. This means that the mother will take the kids to soccer (it seems few have heard of "Soccer Dads," but everyone has heard of "Soccer Moms") and will also purchase the groceries (supermarkets in the USA sell Cosmopolitan magazine rather than Esquire.) This means that the father (using the other car) will go to the hardware store, purchase the parts, and fix the toilet (Oh joy!)

The interesting implementation details relevant here are that the soccer field is on the other side of town and takes approximately 45 minutes driving each way. In addition, the supermarket is next door to the hardware store in a strip mall five minutes from home.

If there are performance constraints, for example, indicating that in addition to the above tasks, Mom also needs to spend five hours preparing dinner to entertain Dad's boss, this might limit the time allowable for soccer and groceries support to under one-hour. This would mandate a re-architecture of the system and its resources so that Dad was assigned to soccer and grocery tasks while Mom took care the toilet repair. (Consider this performance directed social change!)

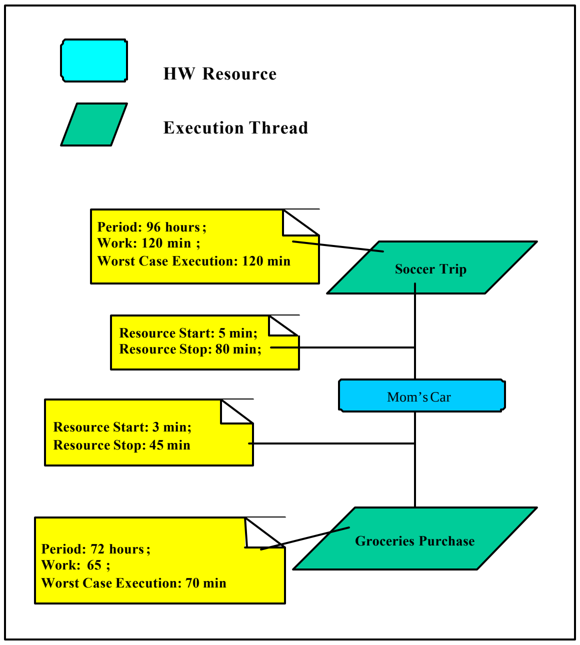
Current design methods and practices do not effectively represent this type of performance information. Work is currently underway within the Object Management Group for an enhanced standard extending the Unified Modeling Language (UML) and to effectively deal with real-time architecture and design issues.

## Sample Graphical Notation

The following figure provides a candidate graphical notation to express the desired semantics at the higher level of design regarding resources, and scheduling issues.

Fig 2. Early Graphical Design and Essential Timing Detail

At the highest level of architecture, there are essential details that must be considered and essential allocations of very high-level functionality to a set of predetermined and constrained resources. Reducing the time that it


HW Resource
Execution Thread
Period: 96 hours; Work: 120 min ; Worst Case Execution: 120 min
Soccer Trip
Resource Start: 5 min; Resource Stop: 80 min;
Mom's Car
Resource Start: 3 min; Resource Stop: 45 min
Groceries Purchase
Period: 72 hours; Work: 65 ; Worst Case Execution: 70 min

takes to purchase groceries (analogous to reducing computation time by speeding things up) has a fairly minimal effect in comparison to the amount of time Mom spends in traffic (analogous to blocking time vs. computation time issues).

This work is actually an extension of the work emanating from 1984 by R.J.A. Buhr about visual prototyping from a work by the same name. These considerations also represent the issues that are being currently dealt with within the Object Management Group in its work extending UML for real-time.

## Rate Monotonic Analysis

Rate Monotonic Analysis (RMA) consists of a set of techniques for analyzing and guaranteeing that the executable threads within a system – including periodic and aperiodic activities – will be completed before their required respective deadlines.

RMA is an analysis rather than a simulation or modeling technique. For complex systems, obtaining the scenario that embodies the worst-case performance stress upon a system is extremely difficult. Simulated execution of such a simulation case usually involves significant computing resources.

Both the average and the worst case response times are of interest for systems which manifest timing requirements. Average response time is usually obtained through simulation modeling or extensive laboratory measurements. Worst-case response time is obtained through application of Rate Monotonic Analysis. System design tradeoffs altering the worst case and average case response times can be done by changing not only execution times, but by altering the internal queuing, synchronization, and concurrency partitioning of the architecture as desired

## Hard & Soft Real-Time

To a first order, time constraints can be characterized in two categories. The first category consists of hard time constraints, for which the system must be carefully designed to never miss one. The implication of a hard-real-time response requirement is that missing such a requirement constitutes a failure to meet some part of the overall system requirements, and is thus logged as a system failure.

The second category consists of soft time constraints. A soft time constraint is one that must be met just like a hard time constraint, but missing one may not always be considered a system failure. Soft time constraints are usually characterized by constraints that can be missed infrequently, or which can be missed by small amounts, or both. There are also other definitions of soft time constraints, such as periodic computations that might be skipped occasionally.

Most real-time systems contain mixtures of hard and soft time constraints. For example, a system might have a requirement for controlling a radar transmitter, which is usually characterized as a hard time constraint. The same system might have response time constraints for operator actions, which are generally soft constraints.

## Timing Requirements

Interestingly, the requirements for meeting time constraints in most application systems are not obvious from the top-level description of the system. Instead, most time constraints are derived from other system requirements, such as accuracy, fidelity, fault-tolerance, or user interfaces.

For example, a robot might have an accuracy requirement for positioning an arm. This frequently results in a derived requirement for periodicity in measuring position; periodicity results in a time constraint for the resulting position computation. For another example, a requirement for fault tolerance will generally imply that a timeout or heartbeat mechanism must be used. The presence of a heartbeat or timeout in a system results in a hard-real-time timing requirement because a failure to complete an operation within the response requirement will result in an anomalous declaration of failure and recovery that is generally at least as dangerous as the original failure for which automatic recovery was required.

## RMA Fundamentals Overview

As previously mentioned, Rate Monotonic Analysis (RMA) consists of a set of techniques for analyzing and guaranteeing that the executable threads within a system will be completed before their required respective deadlines. These techniques are based on work originally performed by the Jet Propulsion Laboratory [1] that proved that a set of periodic tasks would always complete before the end of their periods as long as their total worst-case utilization never exceeds a specified bound that depends only upon the number of tasks, regardless of their phasing. Based on this theoretical result, it was shown that any set of periodic tasks whose total utilization is less than 69% would always complete before the end of their periods.

Subsequently, this basic result has been extended in many directions to handle task synchronization (mutual exclusion), deadlines that are not the same as task periods, arbitrary priority assignments, aperiodic tasks, and many other real-time system situations. The theory requires some basic system architecture information, including task periods (for periodic tasks), task deadlines, task priorities, execution time budgets, synchronization time budgets, and arbitration policies. Given this information, RMA analysis tools can determine whether all time constraints can be met, and if not, which constraints could be missed. The answer must then be interpreted to decide whether the architecture will produce acceptable timing results.

## Rate Monotonic Analysis Example

The following discussion of the family car utilizes the notation from "A Practitioner's Handbook to Rate Monotonic Analysis, Klein et al.", Kluwer 1993. The graphical rendering uses the TimeWiz tool provided by the TimeSys Corporation.

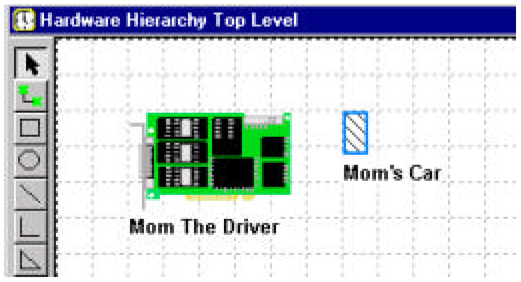


Fig 3 -Resource Diagram

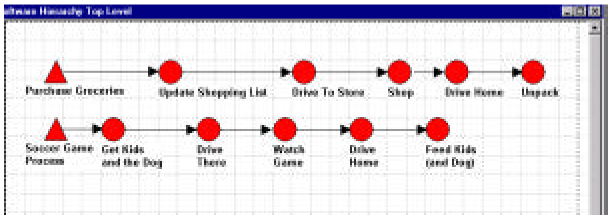


Fig 4 – The Software Behavior Diagram

[1] Liu & Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment*. JACM 20 (1):46, 1973.

Fig 3 shows the Resources specified by the architecture design – the active or schedulable resource (Mom, the driver), and Mom's car (a resource that can be used by only one driver at a time, or used in a locked and protected fashion.) These resources are utilized by the behavior of the system. This is depicted in the software diagram.

Each collection of linked triangles and circles represents a thread. Each element of the software diagram has properties associated with it.

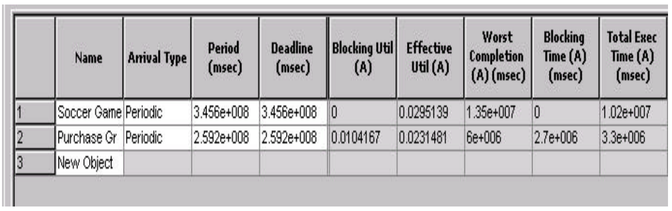
The Triangles are ***Triggers*** and indicate the invocation of a thread. They are concerned with things relevant to invocation of the thread. Circles represent ***Actions.*** Actions convey information about where they execute, how much they execute, and what resources they need in order to execute.

The following table summaries the relevant properties needed to describe the problem and analyze its performance. Note that the timing diagram in Fig 4 is a more precise definition of the problem than Fig 2. The need for the car is more precisely defined to be needed only by the four actions containing "drive" in their names. Each represents driving to or from a respective location. When Mom is watching the soccer game, she really does not need the car and it could be used for other purposes with no additional consequences to Mom's deadlines as long as it showed up at the soccer field by the end of the game.

| Symbol | Property | Meaning | Value |
|---|---|---|---|
| **Triangle** | Period | Repetition interval for execution | Groceries 72 Hrs<br>Soccer 96 Hrs |
| **Triangle** | Deadline | Needed completion Time | Groceries 72 Hrs<br>Soccer 96 Hrs |
| **Circle** | CPU Used | Which Schedulable Resource is Used | All Actions are done by Mom |
| **Circle** | Execution Time | How long the work takes | Update Shop List<br>Drive To Store<br>Shop<br>Drive Home<br>Get Kids<br>Drive To Soccer<br>Watch<br>Drive Home<br>Feed Kids |
| **Circle** | Priority | What the scheduling importance is | All Actions in Groceries Thread get higher priority than all actions in Purchase Groceries (Rate Monotonic Assignment) |

Table 5 – Timing Properties and Meanings

Analysis can now be done to guarantee meeting all deadlines. The TimeWiz tool generated the following:



| | Name | Arrival Type | Period (msec) | Deadline (msec) | Blocking Util (A) | Effective Util (A) | Worst Completion (A) (msec) | Blocking Time (A) (msec) | Total Exec Time (A) (msec) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Soccer Game | Periodic | 3.456e+008 | 3.456e+008 | 0 | 0.0295139 | 1.35e+007 | 0 | 1.02e+007 |
| 2 | Purchase Gr | Periodic | 2.592e+008 | 2.592e+008 | 0.0104167 | 0.0231481 | 6e+006 | 2.7e+006 | 3.3e+006 |
| 3 | New Object | | | | | | | | |

Fig 6 – Worst Case Analysis

Since all analyzed worst-case completion times are smaller than required deadlines, the system is analytically proven to be ***Schedulable***.

## Enhancing the Development Process to Deal with Timing Performance

The top-level concerns that take precedence when performance constraints must be met invert the traditional order and concerns within the design process. A non-traditional view of systems architecture development is needed. It is motivated by the requirement to produce an architecture capable of **predictably** meeting its time constraints in addition to all the traditional correctness, quality, and maintainability concerns.

Object-Oriented systems development and Object-Oriented programming both focus upon producing encapsulations and abstractions for system componentry. The means of encapsulating these abstractions is based upon an analysis of function and data interaction. The effects of the resulting architecture on the ability of a system to meet its performance expectations requires significant additional understanding well beyond the traditional understanding involving function sequences and their combined computational timing requirements.

Starting with the goal of guaranteed timing performance and the technique of Rate Monotonic Analysis, we define an enhanced architecture derivation process that will perform *simultaneous* activities resulting in three kinds of critical architectural decisions about system components:

- partitioning,
- allocating responsibility, and
- defining cooperation.

Traditional approaches and methods define a sequence for making these decisions. However, since the resulting system must meet time constraints, it is of paramount importance that these three decisions all be made simultaneously with collective insight.

Traditional approaches and methods define a sequence for making these decisions. However, since the resulting system must meet time constraints, it is of paramount importance that these three decisions all be made simultaneously with collective insight.

Classical design and early architecture decomposition efforts traditionally focus on domain analysis and functional cohesion to derive subsystem components. This domain analysis traditionally assigns objects to the nouns that appear during domain discovery.

Based on extensive experience in defining time-constrained architectures, however, the authors feel strongly that timing concerns must play an important role in the choice of system modules or objects. Concurrency modeling and its accompanying synchronization behavior become a dominant concern surprisingly early in the architecture development process whenever response time is a critical factor. Understanding the RMA technology and its implications drive us to construct system components that are explicitly constrained to serve *one* primary time constraint to the greatest extent possible. It can be readily shown that objects that serve more than one primary time constraint will introduce inefficiencies and performance degradation within the resulting system.

## How Performance Affects the Choice of Objects

In simple applications, each thread is assigned a priority based upon its rate of execution. (More sophisticated scheduling algorithms are also available that may be used to vary priority between actions.) The choice of threads therefore becomes an important decomposition consideration at high levels of abstraction.

Each thread should serve a single time constraint; otherwise some portion of that thread will be running at a non-optimal priority. At best, such non-optimal priorities result in bounded priority inversion that will adversely affect the overall system performance; at worst, as seen in several real-life examples, the system will fail in unpredictable ways.

Suppose, for example, that a system is being designed to handle data acquisition and tape file creation. The initial design might have a single aggregate object to handle both functions. The sampling rates required for data collection provide a time constraint, while the requirement to keep the tape "streaming" (not underrun the buffer) provides a second time constraint. Actual systems have been built for these functions by using a single larger thread to handle both functions and some additional ones. This approach could not meet performance constraints. Similar systems have demonstrated an inability to meet deadlines even though their overall CPU computation utilization is less than 20 percent!

The solution is to factor the composite object into separate objects, each satisfying a single time constraint. These objects are implemented as schedulable threads, with scheduling priority assigned rate monotonically.

A related performance concern involves the *cooperation* between objects. When something is produced by one object that is needed by another, the objects really do not execute independently, even though they may be characteriz ed as doing so. This is an example of a *precedence constraint*. If two objects are synchronized to depend upon each other's execution, the composite performance takes on the timing constraints of the slower object (longer frame rate) by blocking the faster object, forcing it to "slow down."

As an example, consider a vendor who makes and delivers pizza. A pizza is produced every six minutes, but is delivered in a very strange town. Each delivery takes 20 minutes, because social behavior in that town requires some minimal cordial social interaction. It is also understood that no pizza is thrown away. This is represented in Figure 6. (a relaxed notation is used here for simplicity)

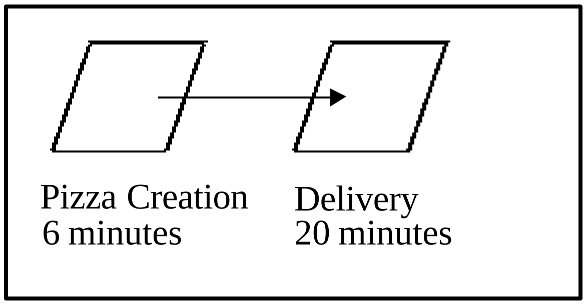


**Figure 6. Pizza Delivery**

The pizzas will stack up on the counter waiting for the delivery vehicle to return! In general, if the time constraints at both ends of the dependency are too dissimilar, the architecture must be adjusted to reduce blocking.

A real example of this is a CPU and its local memory. The original architecture of almost all CPUs and memories looked like Figure 6, where the CPU either read or wrote to the memory. In the early days of computers, CPU instruction execution times and memory access times were similar.

In current systems, memory is much faster than the CPU, requiring a change to the architecture so that the CPU and memory (objects which now have very dissimilar timing constraints) do not *directly* require each other's services. Such direct access produces significant blocking in the architecture while one object waits for the other. In modern CPUs, caching as shown in figure 7 solves this problem.

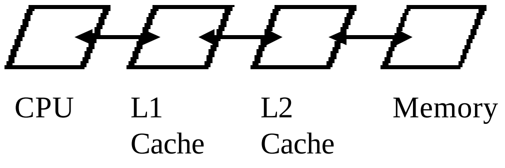


**Figure 7. Modern Cache Architecture**

**A Method for Generating Performance Driven Architectures**

The authors have been co-developers, trainers, and domain users of a design method tailored to produce real-time architectures; i.e., architectures which have non-trivial real-time constraints[2]. This method inverts the order of the activities traditionally performed in object-oriented or structured analysis. This is necessary so that the object decomposition and collaboration strategies that show up extremely early in the architecture derivation and commitment process reflect meaningful timing constraints and collaboration strategies that do not provide excessive blocking or priority inversion.

A layered recursive refinement development approach is assumed. The essential steps in the process are characterized as follows:

1. Initially partition the high-level domain-independent functionality into "first cut" objects characterized as black boxes.
2. Assign reactive responsibility (services provided to clients) for each object
3. Assign proactive responsibility (autonomous actions which are self invoked) to each object.
4. Characterize any shared resources required by the cooperation of reactive and proactive responsibilities within each object.
5. Formulate the arbitration policy of each object to synchronize and allocate resources between the reactive and proactive objects. This includes explicit use of r esources as well as characterizing synchronous and asynchronous communications behavior. (This will lead to explicit characterization of blocking.)
6. Connect the objects as required by their composite functionality and domain based cooperation.
7. Study each object and see if it attempts to satisfy a single time constraint. If more than one, further decompose the object until each object satisfies only one time constraint. Add resources and synchronizations as appropriate for the new partitioning.
8. Look at the connections between objects. When the timing constraints at each end of the connector are excessively dissimilar, introduce "timing transformers" in a manner similar to that done in the memory cache example. This provides minimal blocking between objects by adding new reactive objects whose primary purpose is to provide minimal blocking.

[2] Gerhardt, M., Locke C.D., Real Time Object Oriented Architecture Class, Lockheed-Martin, 1999.

## Back to the Family - Extensibility and Stability

The children will grow older and eventually begin to drive (ugh). They can undertake tasks of their own and, in fact, will incessantly attempt to get permission to use both cars for these tasks.

Schedulability analysis such as RMA not only produces a Boolean result about whether the system can meet all its deadlines (is schedulable), but when not all deadlines can be met, will indicate which ones can and cannot be met! Conversely, if all deadlines can be met, the analysis indicates the total ***effective utilization*** (including blocking) of the system. This gives an accurate indication of how much more work can be added to the system before deadlines are missed. This is just the sort of information that developers of product lines with long lifetimes (and significant extension to the original functionality) are seeking.

## Conclusions

This paper has demonstrated that the seemingly cryptic issues usually associated with hard real-time systems are quite common in everyday life. If we apply our instinct about these issues to real-time system architecture synthesis, achieving guaranteed performance becomes a realistic goal. These steps need to be accompanied by an improved design and decomposition process, adding attributes and behavior pertinent to characterizing utilization and existence of resources that may be contended for by simultaneous potential clients.

Once this notation and design discipline becomes institutionalized by a design team, technology like Rate Monotonic Analysis and commercial tools such as TimeWiz can be readily applied to eliminate the risk and cost of system performance behavior or anomaly.

Existing practices, notation, and design methods must be enhanced to address the inclusion of appropriate behavior and parametric information relating to timing performance. Subsequent scheduling and timing analysis requires such changes.