PERTS: A Prototyping Environment for Real-Time Systems†

Jane W. S. Liu, Kwei-Jay Lin and C. L. Liu
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

ABSTRACT

PERTS is a prototyping environment for real-time systems. It is being built incrementally and will contain basic building blocks of operating systems for time-critical applications, tools and performance models for the analysis, evaluation and measurement of real-time systems, and a simulation/emulation environment. It is designed to support the use and evaluation of new design approaches, experimentations with alternative system building blocks, and the analysis and performance profiling of prototype real-time systems.

I. INTRODUCTION

While existing approaches, techniques and tools for the design, prototyping and development of software systems are effective for many application domains, they often do not address the difficulties in building hard real-time computing systems. A *hard real-time computing system*, hereafter simply called a real-time system, is one in which most tasks have hard timing constraints. Here, the term *task* refers to a basic unit of work. A task may be a granule of computation, a unit of data transmission, a file access, or an I/O operation, etc. The simplest timing constraint imposed on a task is its *deadline*, the point in time by which the task is required to complete. The result produced by a task with a deadline is correct only if it is available by the deadline, in addition to being functionally correct. A late result is of little or no use. Applications supported by real-time systems include command and control, guidance and navigation, flight control, object identification, autonomous vehicle control, and intelligent manufacturing.

The approach that has been taken traditionally to construct real-time systems is to develop the application software first and then tune the application and the underlying system to make sure that all the timing constraints are met. This approach tends to produce brittle, difficult-to-modify and hard-to-maintain systems. Small changes in the application software, or in the underlying hardware and software support, can produce unpredictable timing effects that can be detected and corrected only through exhaustive testing and performance tuning. Consequently, it is costly to develop and validate new systems and to enhance, extend or port existing systems.

The lack of effective methods and tools for building robust and provably responsive real-time systems has motivated the recent research on the theoretical foundations of real-time computing [1,2]. A goal of this research is to find methods for predicting the timing behavior of the basic building blocks and the overall real-time systems built from them. Tools that support systematic construction and evaluation of real-time systems can be built based on these methods. Another goal is to develop integrated approaches to building real-time systems, as alternatives to the traditional approach. An integrated approach would begin with models and optimality criteria that explicitly account for the constraints and possibilities of trading off between various figures of merit, and then design the application and the underlying system to achieve the desired tradeoffs. Such an approach would lead to more flexible, easy-

† This work has been partially supported by ONR Contract Nos. NOOO14-89-J-1181, and N000-92-J-1815.

to-schedule programs, and resultant systems would degrade gracefully during overloads and failures.

This paper describes an ongoing project to build a prototyping environment, called PERTS (Prototyping Environment for Real-Time Systems), that aims at making recent and future theoretical results in real-time systems readily usable to practitioners. Specifically, PERTS will contain

(1) basic building blocks of the underlying support system for real-time applications — These reusable building blocks will realize existing and new scheduling algorithms, communication protocols and resource access control protocols.

(2) building blocks of flexible real-time programs and system software — These system components are based on the imprecise computation approach [3-5].

(3) tools and performance models for the analysis and evaluation of prototype real-time systems — The PERTS tools will provide worst-case bounds and performance predictions of systems based on different execution models and scheduling paradigms.

(4) a simulation/emulation environment — This environment will allow the experimental evaluation of alternatives in scheduling the target software system.

The rest of the paper is organized as follows. Section II describes the models of real-time systems on which PERTS components and tools are based. The capabilities of PERTS and its key components are presented in Section III. This project is compared with similar projects in Section IV. Section V is gives the current status of the project.

II. MODELS OF REAL-TIME SYSTEMS

Most of the workload models used to characterize real-time (software) systems are variations or extensions of the following basic deterministic model. The underlying system contains a number of identical processors. The software system **T**, called a *task system*, contains a number n of tasks. The maximum amount of processor time required by a task T_i to complete its execution is called its *processing time* τ_i. τ_i is assumed to be known. Tasks may have weights which tell us how important the tasks are relative to each other. Again, a task T_i may have a deadline d_i; we say that a task has no deadline if its deadline is infinite. In addition to its deadline, a task T_i may also have a release time r_i, the time instant after which the task is available to be scheduled and executed. The interval $[r_i, d_i]$ between its release time and deadline is its *feasible interval.*

A *schedule* of a task system **T** is an assignment of the processors to the tasks in **T**; a task is scheduled in a time interval on a processor if the processor is assigned to the task in the interval. In any valid schedule, every task is scheduled after its release time. Moreover, the total amount of processor time assigned to every task is equal to its processing time. A valid schedule is a *feasible schedule* if every task is scheduled in its feasible interval and, hence, completes by its deadline.

The system may also contain a number of distinct resources. Each task may require some of these resources during its execution. We say that tasks requiring the same resource are in (*resource*) *conflict* with each other. A resource access control protocol governs the accesses of tasks to resources and resolves the conflicts among them.

Periodic-Task Model

Many real-time applications, such as control-law computations, radar signal processing, and voice/video transmissions, can be characterized by the classical *periodic-task model* [6]. In the periodic-task model, we model such computations and data transmissions as *period tasks*. The system **T** contains

n periodic tasks, each of which is a periodic sequence of requests for the same work. A request is released at the beginning of every period and its deadline is the end of the period. The processing time τ_i of T_i is the maximum amount of processor time required to complete every request in T_i.

In addition to periodic tasks, some tasks may arrive and become ready for execution at random instants. These tasks are *aperiodic*. Aperiodic tasks model computations and communications that must be carried out in response to unexpected events, such as requests for changing the operation mode, processing sporadic messages, etc. Aperiodic tasks usually do not have deadlines, and their processing times may be unknown. We want to complete each aperiodic task as soon as possible, while making sure that all deadlines of periodic tasks are met at all times.

Complex-Task Model

Real-time tasks that are not periodic are often characterized by the classical deterministic model. In this model, a task system **T** is a set of n tasks. These tasks may be dependent; data and control dependencies between tasks impose constraints on the order in which tasks are executed. We use a *precedence relation* $<$ over **T** to specify the constraints on their execution order. T_i is a *predecessor* of T_j (and T_j a *successor* of T_i), denoted as $T_i < T_j$, if T_j cannot begin execution until the execution of T_i completes. T_i is an *immediate predecessor* of T_j (and T_j is an *immediate successor of* T_i) if $T_i < T_j$ and there is no task T_k such that $T_i < T_k < T_j$. Two tasks T_i and T_j are *independent* when neither $T_i < T_j$ nor $T_j < T_i$. They can be executed in any order. We can use a directed graph $G = (\mathbf{T}, <)$, a *task graph*, to represent the task system **T** and the precedence constraints among tasks. There is a node in G for each task in **T**. There is an edge from T_i to T_j when T_i is an immediate predecessor of T_j. Figure 1 shows a task graph for example. Nodes of all shapes represent tasks. The numbers in the brackets above the tasks are the feasible intervals of the tasks. For simplicity, their other attributes, such as their processing times and resource requirements, are not shown.

We note that a periodic task in the periodic-task model can be modeled as an infinite chain of dependent tasks where the first task is the immediate predecessor of the second task, the second task is the immediate predecessor of the third task, and so on. Such a chain is shown in Figure 1; it represents a periodic task whose first request is released at time 2 and whose period is 3.

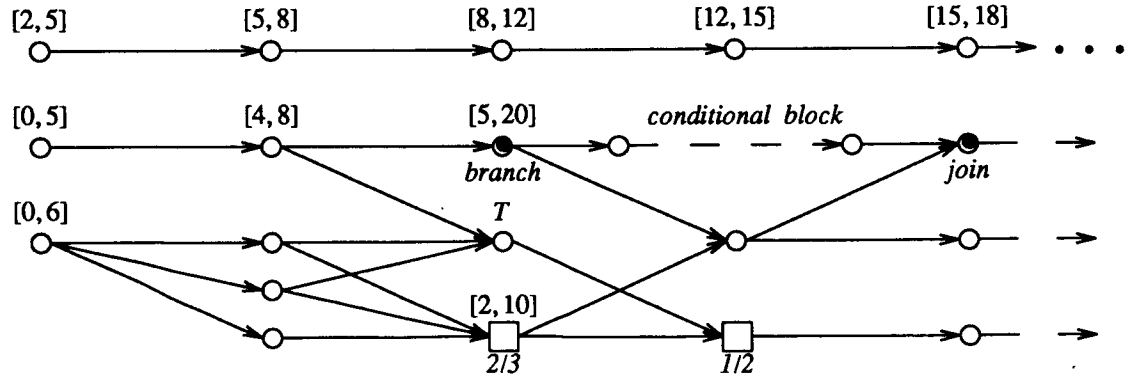

Figure 1 An Example of Task Graphs

Real-time applications sometimes contain redundant modules, carry out heuristic searches, use multiple versions, execute some tasks conditionally, etc. These applications cannot be conveniently characterized by the classical model. For this reason, we extended the classical model; the extensions include OR tasks [7] and conditional blocks [8].

In the classical model, a task with more than one immediate processor must wait until all its immediate processors have been completed before its execution can begin. We call such tasks *AND tasks*. An example is the task labeled T in Figure 1. All three of its immediate predecessors must be completed before T can begin execution. All other AND tasks are represented by unfilled circles. In some applications, a task may begin execution after one (or some) of its immediate predecessors is completed. Such a task is called an *OR task*. A task system containing AND and OR tasks is said to have *AND/OR precedence constraints*. Examples of OR tasks are the two square nodes at the bottom of the graph in Figure 1. The one labeled *2/3* can begin execution as soon as 2 of its 3 immediate predecessors complete. In a triple-redundant module, the voter can be modeled as a 2/3 OR task; it and its successors can proceed as soon as two out of its three replicated immediate predecessors complete. Similarly, we can model a two-version computation as the two immediate predecessors of a 1/2 OR task; only one of them needs to be completed before the OR task can begin execution.

In the classical model, all the immediate successors of a task must be executed; an outgoing edge from every node expresses an *AND constraint*. This model cannot characterize data-dependent, conditionally executed tasks. In the complex-task model, some outgoing edges express *OR constraints*. Only one of all the immediate successors of a task whose outgoing edges express OR constraints is to be executed. Such a task is called a *branch* node. In a meaningful task graph, there is a *join* node associated with each branch node. Each subgraph whose source node is an immediate successor of a branch node and whose sink node is an immediate predecessor of the corresponding join node is called a *conditional branch*. Here, by a source (or sink) node of a subgraph, we mean a node that has no predecessor (or successor) in the subgraph. The subgraph that begins from a branch node and ends at the associated join node is called a *conditional block*. Only one conditional branch in each conditional block is to be executed. An example is shown in Figure 1 where the conditional block has two conditional branches. Either the upper conditional branch, containing a chains of tasks, or the lower conditional branch, containing only one task, is to be executed.

Imprecise-Computation Model

For many real-time applications, it is better to have timely, approximate results than late exact results. A system that supports *imprecise computations* [3-5] attempts to produce usable approximate results when an overload or failure prevents an exact result from being produced in time. The system does so by trading off the quality of the results produced by the tasks for the amounts of processing times required to produce the results. To make this tradeoff possible, we structure every task in such a way that it can be logically decomposed into two parts: a mandatory part and an optional part. The mandatory part is the portion of the task that must be done in order to produce a result of an acceptable quality. This part must be completed before the deadline of the task. The optional part is the portion of the task that refines the result. The optional part, or a portion of it, can be left unfinished, if necessary, at the expense of the quality of the result produced by the task.

The imprecise-computation model captures this task structure. Each task T_i is decomposed into two tasks: the *mandatory* task M_i and the *optional* task O_i. Let m_i and o_i be the processing times of M_i and O_i, respectively. $m_i + o_i = \tau_i$. m_i is always bounded and known. On the other hand, o_i and, hence, τ_i can be unknown and unbounded. The release time and deadline of the tasks M_i and O_i are the same as that of

T_i, and O_i is the immediate successor of M_i. We note that the complex-task model is a special case of the imprecise computation model in which all tasks are entirely mandatory, that is, $o_i = 0$ for all tasks. Intelligent and incremental computations, known as anytime or sufficiently good computations in AI literature, can be also modeled as tasks that are entirely optional, that is, $m_i = 0$ for all tasks.

In a valid schedule of a system of imprecise tasks, the total amount of processor time assigned to each task is at least equal to m_i and at most equal to τ_i. A task is said to be *completed in the traditional sense* at an instant t when the total amount of processor time assigned to it becomes equal to its processing time at t. A mandatory task M_i is said to be completed when it is completed in the traditional sense. The optional task O_i may be terminated at any time, however, even if it is not completed at the time; no task is scheduled outside of its feasible interval. A task T_i is said to be completed in a schedule whenever its mandatory task is completed. When the total amount of processor time σ_i assigned to O_i in a schedule is equal to o_i, the *error* ε_i in the result produced by T_i (or simply the error of T_i) is zero. Otherwise, if σ_i is less than o_i, the error of T_i is equal to $E_i(\sigma_i)$, the *error function* of the task T_i. $E_i(\sigma_i)$ is typically a monotone non-increasing function of σ_i. In other words, the longer a task is allow to execute, the smaller the error in the result it produces.

Reference Model of Real-Time Systems

Figure 2 shows a generic model of real-time systems. The software system is represented by a task graph. As stated earlier, the task graph gives the processing time and resource requirements of tasks, the timing constraints of each task, and the dependencies between tasks. Tasks are scheduled and allocated resources according to a set of scheduling algorithms and resource access control protocols. This set of algorithms and protocols is an explicit element of the reference model as shown in this figure.

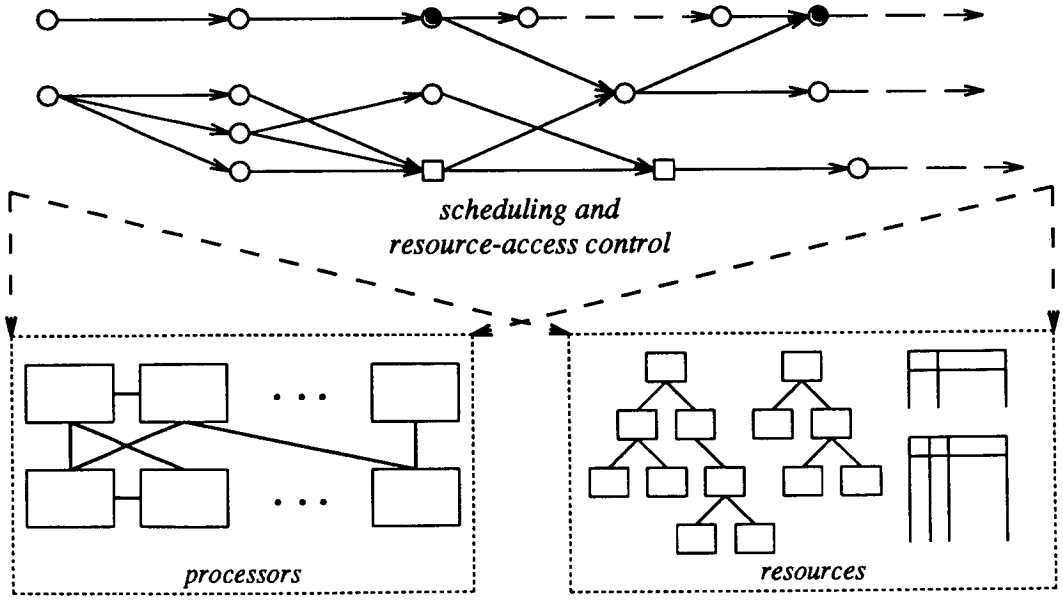

Figure 2. A Model of Real-Time Systems

The underlying hardware and run-time system is modeled as a set of processors and resources. Processors are entities that are typically modeled as servers in queuing models. Computers, I/O buses, communication networks and virtual connections are examples of processors. Resources are entities that are sometimes modeled as passive resources or passive queues. Memory pages, I/O buffers, semaphores, and valid message numbers in send and receive windows are examples of resources. It is not necessary for us to make a fine distinction between processors and resources. Rather, we characterize each processor or resource by a set of parameters. Some of these parameters specify the constraints governing its usage, such as whether it can be shared, whether it is reusable, etc. Other parameters give its timing properties, such as context switch time, acquisition time, etc.

III. PERTS COMPONENTS AND CAPABILITIES

Figure 3 shows the key components of PERTS. PERTS can be used to support the design of real-time systems. The design of a target task system is captured by its abstract description, which is a task graph. At the abstract level, estimated task parameters and dependencies in the task graph can be derived from the requirements of the system. During the design phase, the schedulability analysis system will serve as an interactive tool. This tool can be used for many purposes, including to determine whether sufficient amounts of all resources are available; to identify potential bottlenecks; to select computational algorithms from available choices with different levels of result quality versus resource requirements; and to provide suggestions on the choices of task parameters. The analysis tool and performance predictor can be used to identify where later changes in software or hardware are likely to lead to unpredictable timing effects. In this way, the schedulability analysis system can also help in the design of the test suite which will be needed later to test the target system.

The schedulability analysis system will support the hierarchical approach to building large and complex, real-time software on distributed and parallel hardware platforms. Examples of algorithms that will be implemented for this purpose include algorithms for end-to-end scheduling of distributed tasks that have overall deadlines; algorithms for scheduling parallelizable tasks with deadlines on massively parallel systems; partitioning and assignment schemes for statically assigning tasks to processors; load balancing algorithms for dynamic adjustment of load conditions; and protocols for controlling concurrent access to resources and data transmissions. For example, the task partitioning and assignment module can help the designer to find a partition and assignment of the given task system so that the tasks assigned to each processor can meet their individual deadlines and the overall task system can meet its end-to-end deadlines. When the given task system does not have such an assignment, the analysis tool in the system can suggest possible changes to make such an assignment feasible. If a dynamic task assignment approach is chosen, the performance predicting tool can be used to determine whether the worst-case performance of the assignment is acceptable.

PERTS will provide similar support in later phases of software prototyping. In earlier development stages, PERTS can be used to identify and choose a set of operating system policies for task partitioning and assignment, load balancing, scheduling and resource management. In this case, the concrete description may simply be a more detailed task graph that gives more accurate information about the timing and resource usage characteristics of the tasks. PERTS will produce sample task assignments, schedules, memory layouts, etc. to provide the feedback needed in the iterative software prototyping process. PERTS will have program execution time analysis and measurement tools. In later stages, when some source code of the target task system becomes available, these tools can be used to extract task parameters and graph structures from the code. PERTS also provides a simulation environment for a thorough evaluation of the target system. The most concrete description is the instrumented object code

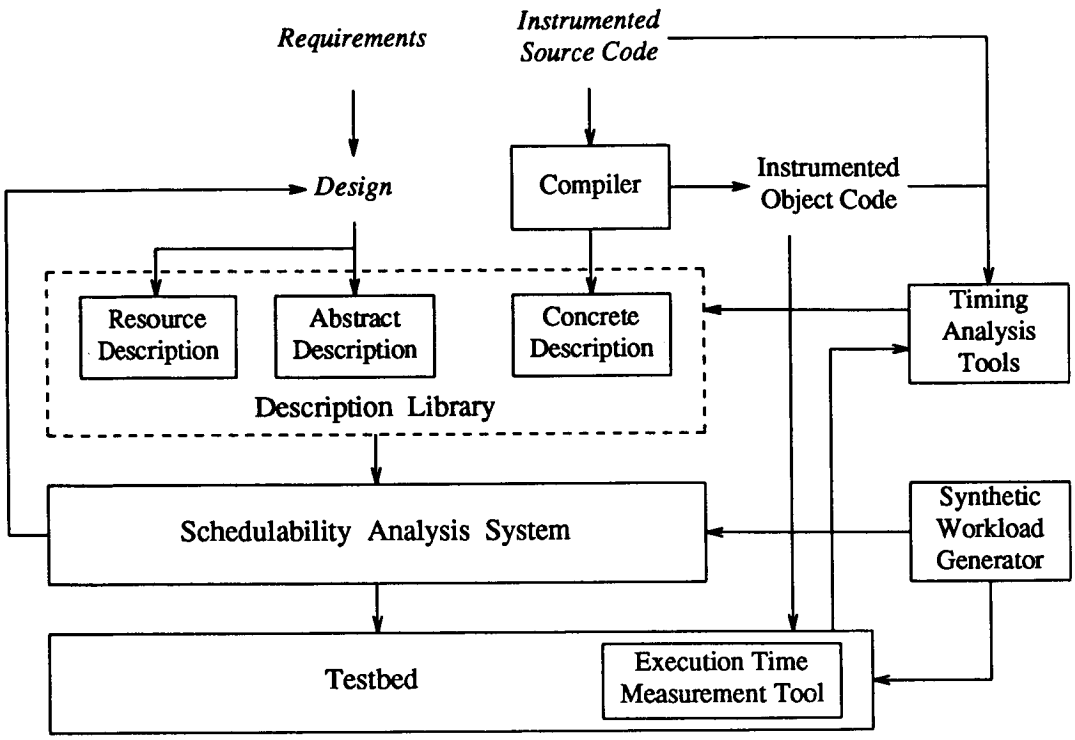

Figure 3. The Prototyping Environment of Real-Time Systems

generated by a cross language compiler. This code can be run, under the scheduling directives produced by the scheduling analysis system, in a simulated target environment provided by the testbed The testbed will contain a workload generator capable of generating synthetic or trace-driven workloads to support the simulation of the embedded environment.

IV. RELATED WORK

PERTS is similar to many other real-time systems design tools in its intended use. These systems all intend to reduce the complexity in real-time system development. The advanced algorithms and tools available in PERTS distinguish it from the other systems. For example, Scheduler 1-2-3 [9] primarily deals with periodic tasks, mixed with randomly arriving aperiodic tasks, and priority-driven scheduling disciplines. Several systems similar to Scheduler 1-2-3 are also available. They support the design and construction of domain specific applications. PERTS, on the other hand, provides a much more versatile and powerful schedulability analysis system. The PERTS testbed can be configured to simulate a number of operating systems and hardware configurations.

PERTS differs from most existing and experimental real-time system prototyping and development systems, and complements them, both in their capabilities and intended use. Many such systems provide an integrated environment with a full range of tools for requirement tracing, program construction, software reuse, etc. The experimental system CAPS [10] is an example. PERTS is similar to CAPS in certain ways; for instance, both provide tools for analysis of real-time software. CAPS is a stand-alone prototyping environment. PERTS is not designed to be a substitute for CAPS or other computer-aided

software prototyping systems. Rather, PERTS will focus on providing powerful design and evaluation tools that are not available in these systems.

V. CURRENT STATUS

We are implementing the components of PERTS incrementally in C++. Several suites of scheduling algorithms are in various stages of completion. They are algorithms for scheduling periodic tasks, imprecise computations, tasks with end-to-end deadlines [11] and tasks with AND/OR precedence constraints [7], as well as algorithms for assigning tasks to processors. The suite of algorithms for scheduling periodic tasks is near completion. Components of this suite that have been implemented and tested include the basic rate-monotone algorithm and the earliest-deadline algorithm; priority-ceiling protocol and stack-based protocol for resource access control; servers for handling aperiodic requests; and mode change protocols [1]. A basic schedulability analysis system, containing tools based on the rate-monotone scheduling theory and worst-case performance analysis, has been designed.

We have designed a simple language for describing task graphs and have implemented a compiler for this language. A user can describe a task graph in terms of this language, and the compiler will produce the graph. We also have a preprocessor that automatically extracts task graphs from annotated C++ programs.

REFERENCES

1. Van Tilborg, A. M. and G. M. Koob, *Foundations of Real-Time Computing: Scheduling and Resource Management*, Kluwer Academic Publishers, 1991.
2. Van Tilborg, A. M. and G. M. Koob, *Foundations of Real-Time Computing: Formal Methods and Specifications*, Kluwer Academic Publishers, 1991.
3. Liu, J. W. S., K. J. Lin, C.L. Liu, and C. W. Gear, "Imprecise Computations," in *Mission Critical Operating Systems*, Edited by A. K. Agrawala, C. D. Gordon and P. Hwang, IOS Press, Amsterdam, 1992, pp. 159-169.
4. Liu, J. W. S., K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer*, May 1991, pp. 58-68.
5. Liu, J. W. S., K. J. Lin, and C. L. Liu, "Imprecise Computations: A Means to Provide Scheduling Flexibility and Enhance Dependability," to appear in *Readings on Real-Time Systems*, Edited by Y. Lee and M. Krishna, IEEE Press.
6. Liu, C. L. and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, January 1973, pp. 46-61.
7. Gillies, D. and J. W. S. Liu, "The Complexity of AND/OR Scheduling," *Proceedings of the 2nd IEEE Conference on Parallel and Distributed Processing*, Dallas, Texas, December 1990, pp. 394-401.
8. Kim, T., C. L. Liu and J. W. S. Liu, "A Scheduling Algorithm for Conditional Resource Sharing," *Proceedings of the IEEE International Conference on Computer-Aided-Design*, November 1991, pp. 84-87.
9. Tokuda, H. and C. W. Mercer, "A Real-Time Tool Set for the ARTS Kernel," *Proceedings of the 9th IEEE Real-Time Systems Symposium*, December 1988.
10. Luqi, "Software Evolution Through Rapid Prototyping," *IEEE Computer*, May 1989.
11. Bettati, R. and J. W. S. Liu, "End-to-End Scheduling to Meet Deadlines in Distributed Systems," *Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 1992.*