Chapter 7: PERTS: A Prototyping Environment for Real-Time Systems

J. W. S. Liu, C. L. Liu, J. L. Redondo, Z. Deng, T. S. Tia,

R. Bettati, J. Sun, A. Silberman, M. Storch, D. Hull

Department of Computer Science
University of Illinois, Urbana, Illinois 61801

1 Introduction

"Non-functional, or quality, aspects of large systems are often treated in an ad hoc manner—even when they are critical to the system's ultimate success. It is usually difficult to defend claims about a system's reliability or performance, for example, before large portions of the system have been implemented and tested." [SW93] This statement by Salasin and Waugh is especially true for real-time systems. Here by a ***real-time system***, we mean one in which a significant portion of all jobs are time-critical. The term ***job*** refers to a basic unit of work to be scheduled and allocated resources. A job may be a granule of computation, a unit of data transmission, a file access, or an I/O operation, and so on. Time-critical jobs have timing constraints. In the simplest form, the timing constraints of a job are specified in terms of its release time and deadline; the job's execution cannot begin until its ***release time*** and must be completed by its ***deadline***. The failure of a job to complete by its deadline is considered to be a timing fault, and a real-time system functions correctly only in the absence of timing faults. To validate a real-time system, its builder must demonstrate convincingly that all time-critical jobs always complete by their deadlines.

Traditionally, real-time systems are built by first focusing on their functional requirements and then validating the system's timing constraints by exhaustive simulation or testing. This approach is time consuming and costly. Even worse, exhaustive simulation and testing are reliable and feasible only for systems that use clock-driven, cyclic scheduling strategies. Consequently, almost all real-time systems that support critical applications are clock-driven. Such a system is difficult to maintain and extend. Because a small change in the application software or the underlying hardware and system software can produce unpredictable timing effects, the system must be tuned and tested exhaustively after every change.

An alternative to the clock-driven paradigm is the priority-driven approach to scheduling and resource management. A scheduling algorithm is ***priority-driven*** if it does not intentionally leave any processor idle. Almost all modern, event-driven scheduling algorithms, such as the FIFO, shortest-processing-time, earliest-deadline-first, and rate-monotonic algorithms, are priority-driven. Priority-driven algorithms are commonly used in non-real-time systems. These algorithms are supported by commonly used and standard operating systems (such as Posix Real-Time Extensions, Mach and Lynx OS) and programming languages (such as Ada9x).

Despite the fact that it is easier to maintain and enhance systems built on priority-driven strategies, the adoption of these strategies in real-time systems has been slow. The primary reason is the lack of reliable and tractable methods for validating timing constraints in systems based on these strategies. It is well known that priority-driven algorithms exhibit anomalous behaviors: The response time of a set of jobs can be larger when more processors are used execute them, when jobs have shorter execution times and few dependencies, and when jobs are released for execution earlier. When jobs have arbitrary release times and share nonpreemptable resources, scheduling anomalies can occur even when there is only one processor and jobs are preemptable. Unfortunately, variations in job execution time and resource requirements and jitters in job release times that can lead to anomalies are unavoidable. For this reason, priority-driven scheduling algorithms have not been used in real-time systems until recently.

Recent research on foundations of real-time systems has led to stable and responsive priority-driven scheduling strategies, as well as reliable and efficient methods for validating the timing properties of systems based on them. PERTS (Prototyping Environment for Real-Time Systems) is a system of software modules and tools built on these theoretical advances [VK91a] [VK91b]. The software modules provided by PERTS realize the existing and emerging real-time scheduling algorithms, task and resource assignment algorithms, and resource access control protocols. They are the building blocks of operating systems for time-critical applications. By providing a comprehensive system of design, analysis, validation, and evaluation tools together with these building blocks, PERTS supports the systematic and rigorous evaluation of new designs, experimentation with alternative scheduling and resource management strategies, and the analysis and validation of the resultant prototype system. By making research advances readily usable to system developers, PERTS will also serve as a technology transfer vehicle.

PERTS is similar to many real-time systems design and evaluation tools; the advanced algorithms and rigorous methods used in it distinguish PERTS from others. For example, the PERTS schedulability analysis system has the capability of Scheduler 1-2-3 [TM88], but is more versatile and powerful. The PERTS testbed can be configured to simulate a wide range of operating systems, hardware platforms, and scheduling hierarchies. Unlike integrated prototyping environments, such as CAPS [Luq89], PERTS does not provide a full range of prototyping tools. Rather, it focuses on providing rigorous tools and system building blocks that are not available in these systems.

Following this introduction, we describe briefly the key components of PERTS and the reference model that characterizes real-time systems and captures their timing constraints and resource requirements. The PERTS tools assume that the target system to be analyzed, validated, and evaluated is described in terms of this model. The capabilities and intended usages of PERTS in prototyping are then described. A key component is the **`schedulability analyzer`**. This unique set of analysis and validation tools addresses the difficult validation problems discussed in the subsequent section. These problems have been largely solved for systems based on the periodic-task model [LL73] [LW82] [SSL89] [LSD89] [SRL90] [Bak90]. The basic version of the PERTS schedulability analyzer makes use of these recent theoretical results. This basic system of tools is now available and is also described in the section. The following section describes the PERTS simulation environment. The last section discusses future directions.

2 Key Components

Figure 1 shows a block diagram of PERTS. All PERTS software modules and tools are implemented in the C++ programming language and run under the X Window System.

Currently, PERTS contains an extendible library of reusable software modules and a comprehensive set of analysis, evaluation and validation tools. The software modules implement commonly-used and new scheduling algorithms and resource access control protocols, including the well-known algorithms for scheduling and managing resource access of periodic tasks [LL73] [LW82] [SRL90] [Bak90], servers for handling aperiodic jobs [SSL89], on-line and off-line algorithms for scheduling imprecise computations [LLS+94], and many other recently developed algorithms. The user can select and use a subset of them, together with an operating system kernel that allows external schedulers and resource managers, to assemble an effective run-time support system. PERTS schedulability analysis tools provide worst-case bounds and performance predictions of systems based on different workload models and scheduling paradigms. These tools allow the user to validate the timing constraints of the prototype system and provide feedback on

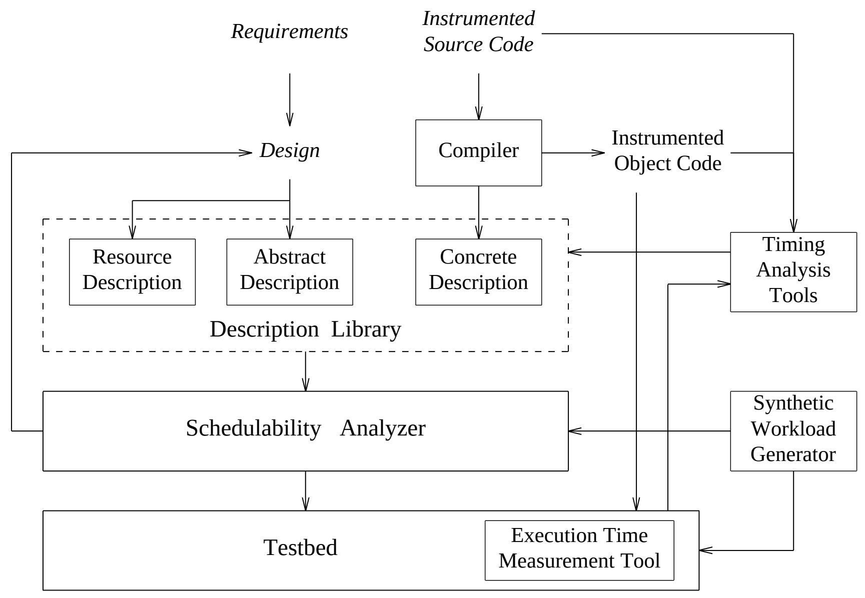

FIGURE 1: PERTS Tools and User Interface

its performance throughout the prototyping process.

Several aspects of real-time systems make an *a priori* schedulability analysis difficult or impossible. The arrival pattern of aperiodic tasks may be probabilistic and not adequately characterized before execution. Active resources in the system may be scheduled using different scheduling algorithms. Communication times may be nondeterministic, and so forth. In order to determine the timing properties of such systems, we turn to simulation methods. While simulation methods have been extensively used, most simulators are designed around a single scheduling paradigm and do not focus on real-time issues. PERTS provides an object-oriented simulation environment, which allows its users to easily construct discrete-event simulators of multi-paradigm, distributed real-time systems. This environment allows the experimental evaluation of alternatives in scheduling and resource management and the performance profiling of the overall system.

When it is completed, PERTS will also contain timing analysis and measurement tools. The timing tools can be used to extract from the annotated code its processing time and resource requirements, control and data dependencies, and timing constraints and, thus, the description of the part of the system implemented by

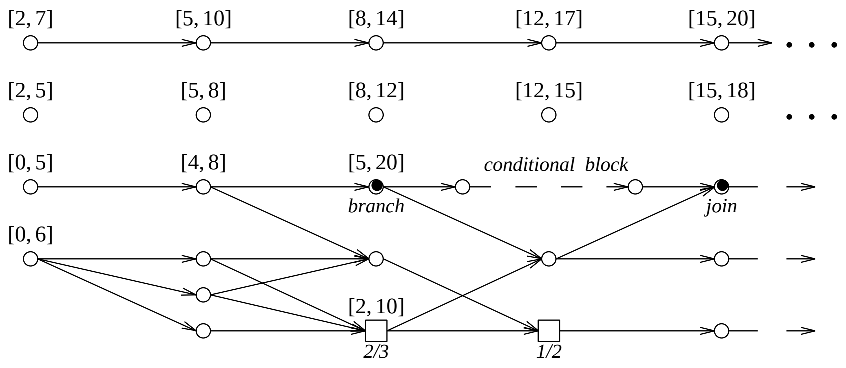

FIGURE 2: An Example of Task Graphs

the code.

3 Reference Model

For the purpose of validating its timing constraints and evaluating its performance, it suffices to describe a real-time system by the workload performed by the system, the resources available to support the workload, and the algorithms used to allocate resources. The PERTS reference model of real-time systems characterizes each real-time system in this way: a system is defined by a task graph, a resource graph and a set of scheduling algorithms and resource access control protocols. The ***task graph*** describes the application system, called the ***task system***. The ***resource graph*** describes the physical and logical resources available to the task system. The scheduling and resource access control algorithms characterize the part of the operating system that allocates resources to the task system.

3.1 Task Graphs

A PERTS task graph is an extended precedence graph; the model represented by the graph is an integration of three well-known real-time workload models: the periodic-task model, the complex-job model, and the imprecise computation model [LLS+94]. The graph in Figure 2 shows an example. Each vertex in this graph, shown either as a circle or a square, represents a job. Edges in a task graph are directed; they specify data, temporal and control dependencies among jobs. There is an edge from *J* to *K* if the corresponding job *K* depends on the job *J* in some way. When there is an edge from *J* to *K*, the job *J* is an ***immediate predecessor*** of the job *K*, and the job *K* is an ***immediate successor*** of the job *J*.

Each job is defined by four sets of parameters: the temporal parameters, resource parameters, functional parameters, and interconnection parameters. The ***temporal parameters*** concern the job's temporal requirements and constraints. Maximum, average and minimum ***processing times*** are examples of temporal parameters, as are its release time and deadline. In Figure 2, the numbers in the square brackets above each vertex gives the release time and deadline of the job represented by the vertex. Other parameters are omitted for simplicity.

Traditional real-time applications, such as control-law computations and sensor data transmissions, are often characterized by the periodic-task model [LL73]. In this model, such computations and communications are ***periodic tasks***, each of which is a periodic sequence of identical jobs. A job is released at the beginning of every period and its deadline is some time instant at or before the end of the period. Figure 2 illustrates how periodic tasks are represented in the task-graph model. The chain at the top of the graph represents a periodic task whose phase (that is, the release time of its first job) is 2 and whose period is 3. Jobs in this task are dependent. Below the chain is another periodic task with the same phase and period. However, jobs in this task are independent. In the periodic-task model, some tasks are aperiodic. An ***aperiodic task*** is a stream of aperiodic jobs that are released and ready for execution sporadically; the jobs have the same statistical parameters: the same interarrival-time and execution-time distributions and the same response-time requirements. Aperiodic jobs model computations and communications that must be carried out in response to unexpected events, such as fault recovery, mode changes, and operator commands. The temporal parameters of jobs in aperiodic tasks are given in terms of their probability distributions.

The ***resource parameters*** indicate what resources a job requires to execute, as well as the intervals of time and numbers of units required. ***Functional parameters*** describe other relevant intrinsic properties of the job. Among the functional parameters are ***weight***, which denotes how important the job is, and ***laxity type***, which indicates qualitatively the relative cost of exceeding the deadline but still completing execution, versus the cost of not completing the execution if the execution cannot complete in time. The ***laxity function*** of a job gives us a quantitative measure of the value of completing the execution of the job at any particular time. These parameters tell us whether the deadline of the job is ***soft*** or ***hard***. It is not essential to make sure that every soft deadline is met. However, a short response time is usually a design goal. Also, a job may have one or more optional parts, defined by a set of optional intervals. Optional parts can be left unfinished. Optional parts allow us to model parts of a computation or data transmission that can be skipped under overload conditions at the expense of result quality, as well as multi-versioned jobs.

The ***interconnection parameters*** of a job give us a part of the information about how the job depends on other jobs. For example, the parameter ***out-type*** of a job specifies how many of its immediate successors must execute and whether the result of the job's execution has any bearing on which immediate successors execute. The parameter ***in-type*** of a job specifies how many of its immediate predecessors must be completed before it can begin execution. By default, the values of these parameters of every job are AND, meaning that a job can begin execution only when all of its immediate predecessors are complete and all of its immediate successors must be executed. By giving one or both of these parameters an OR value, we can represent conditional executions of jobs as exemplified by the subgraph in Figure 2 delimited by the two filled circles. These vertices mark the beginning and end of a conditional block that has two conditional branches. Either the upper branch, containing a chain of jobs, or the lower branch, containing only one job, is to be executed. Similarly, each of the square vertices represents a job which can begin execution as soon as some of its immediate predecessors complete. For example, the vertex marked 2/3 models a voter in a triple-redundant module; it and its successor can proceed as soon as two out of three replicated immediate predecessors complete. The immediate predecessors of the 1/2 vertex model two versions of a multi-version computation; only one of them needs to be completed for it to proceed.

Together with interconnection parameters of jobs, parameters of edges between jobs completely specify how jobs depend each other. Specifically, the volume of communication data and the temporal distances between jobs are parameters of the edges connecting them.

3.2 Resource Graphs

Similarly, each resource (type) is defined by its parameters. Some parameters of a resource specify the constraints governing its usage, such as whether it is preemptable, whether it is reusable, etc. Other parameters give timing properties, such as its processing rate, context-switch time, etc. The user describes a system of resources by means of a resource graph. In this graph, each vertex represents a resource type whose attributes include the number of units. Vertices may be connected by two types of edges. There is an ***is-a-part-of edge*** for a vertex *X* to a vertex *Y* if the resource *Y* is a part of *X*. These edges connect vertices into rooted trees and give us information on the localities of the resources. For example, in the resource graph shown in the middle of Figure 3, there are four rooted trees whose vertices are shown as boxes. Each tree represents a computer or network whose components are represented by vertices in the tree. The other edges, shown as directed dashed edges in Figure 3, represent ***accessible-from*** relationships. They indicate which resources can be accessed remotely and by whom.

Some attributes of a job are specified by both its own parameters and the parameters of the resource(s) it requires. An example is ***execution time***, the actual

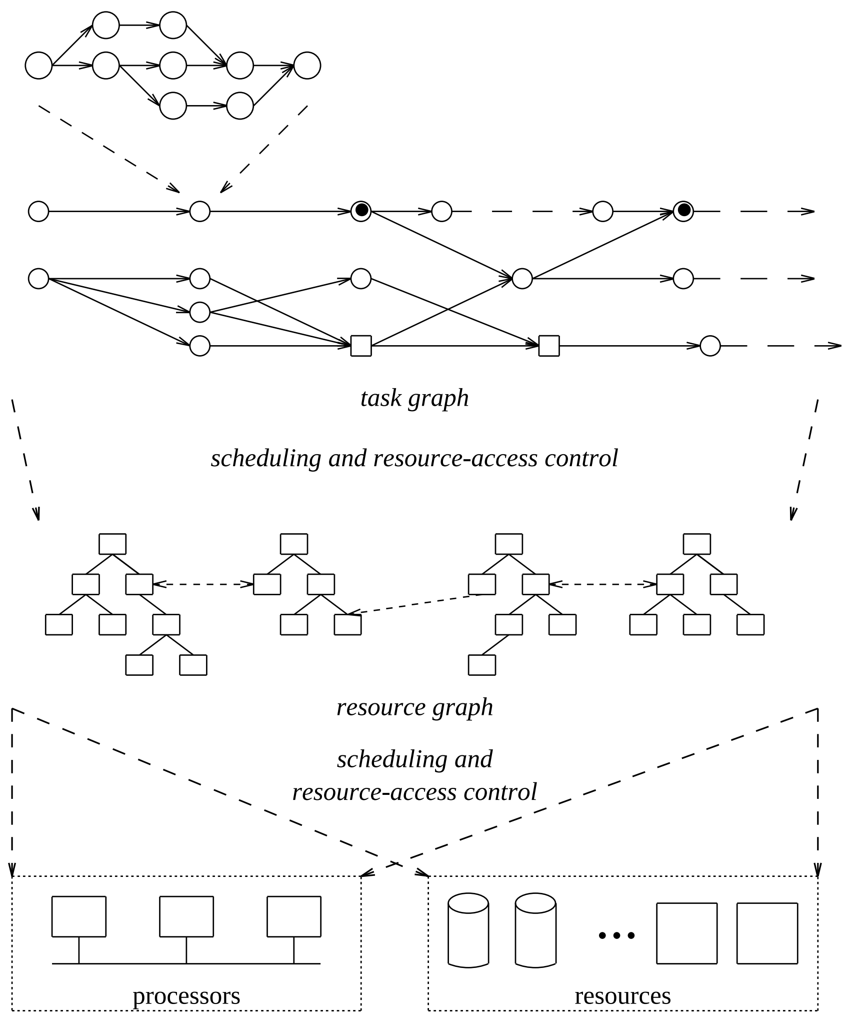

FIGURE 3: A Reference Model of Real-Time Systems

amount of time that the job executes before it completes. This time is a function of both the processing time of the job and the processing rate(s) of the processor(s) it requires. The former is a job parameter and is the execution time of the job when the processing rate of the processor on which it executes is 1. Hence the execution time is reduced by the factor 1/*x* when the processing rate is increased from 1 to *x*. The user may wish to investigate how the processing rate(s) of some resource(s) used by jobs affect their timing behavior. It will be easy to change the system description for this purpose. Another example is preemptability. We view a “nonpreemptable” value of the preemptability parameter of a job as an external constraint that the user wants imposed on the way the job is scheduled. Often, there is no intrinsic reason for a job to be nonpreemptable, but preemption may be costly. In this case, the user may say that the job is preemptable and rely on PERTS tools to determine when preemptive scheduling strategies are too costly and should not be used. On the other hand, whether a resource is preemptable is typically a functional property of the resource. For example, write locks and valid sequence numbers in a sliding window protocol are resources that must be used serially and, hence, are not preemptable. When a job requires a nonpreemptable resource, it is nonpreemptable on that resource.

3.3 Scheduling Hierarchy

The third element of the PERTS reference model is the set of algorithms and protocols used to map the task graph onto the resource graph. Figure 3 shows how this element completes the description of the system to be analyzed and validated.

Some resources, such as memory, are physical entities. Other resources, such as database locks and system calls, are logical entities. Logical resources are implemented by system software and, therefore, must be scheduled to execute on physical resources. The scheduling and resource access control algorithms for this purpose are typically different from the ones used for the application system. Also, a job may be divided into subjobs, and the resources allocated to the job by the operating system are in turn allocated to the subjobs. Figure 3 shows such a scheduling hierarchy. The PERTS simulation environment and tools are based on this view of the overall system. Using them to study the interaction between scheduling strategies used in the different levels of the scheduling hierarchy will be convenient.

4 Capabilities and Usage

Figure 4 illustrates one way PERTS may be used to the support the systematic migration from the initial design to the full implementation of a prototype real-time system. Each box in this figure represents one or more PERTS tools whose purposes are stated by the label of the box.

Specifically, the PERTS schedulability analyzer can serve as an interactive design tool. In the initial stage of the prototyping process, the architecture of the target application system is captured by an abstract task graph. At the abstract level, the task graph represents a family of possible configurations of the system. Job parameters and dependencies have only estimated values derived from the re-

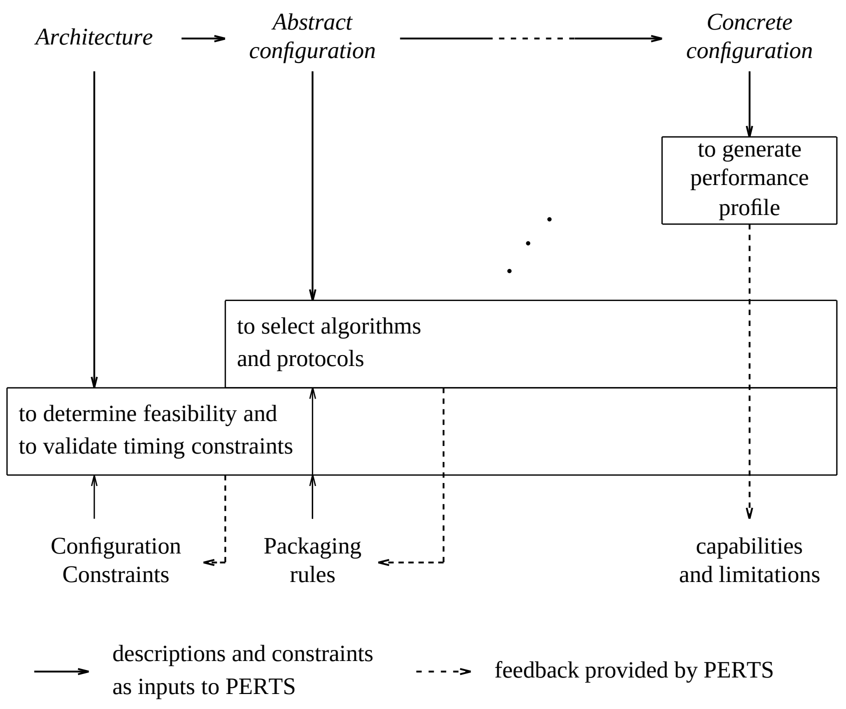
Architecture
Abstract configuration
Concrete configuration
to generate performance profile
to select algorithms and protocols
to determine feasibility and to validate timing constraints
Configuration Constraints
Packaging rules
capabilities and limitations
descriptions and constraints as inputs to PERTS
feedback provided by PERTS

FIGURE 4: Usage of PERTS Tools in Prototyping

quirements of the system. The schedulability analyzer can be used for many purposes: to determine whether the budgeted amounts of all resources are sufficient to achieve the required degree of responsiveness; to select computational algorithms from the available choices, which have different levels of result quality versus processing time and resource requirements; and to suggest values of job parameters, such as the periods of the periodic tasks, the sizes of servers for handling aperiodic jobs, the granularities of distributed modules, and alternative dependency relationships. In other words, the tool provides the feedback needed for the user to select a feasible configuration that can meet all timing constraints.

The schedulability analyzer will support the hierarchical approach to building large and complex real-time software on distributed and parallel hardware platforms. Examples of algorithms and tools for this purpose include modules for scheduling and validating jobs with end-to-end deadlines and for assignment of jobs to processors. For example, the job assignment module can help the designer find a feasible partition and assignment such that the jobs assigned to each processor can meet their individual deadlines and the overall task system can meet its end-to-end deadlines. When the task system does not have a feasible assignment, the schedulability analyzer can suggest possible changes to job and resource parameters that will help the user make the task system feasible.

PERTS can provide similar support in the later phases of software prototyping. For example, PERTS can be used to identify and choose a set of scheduling algorithms and resource access protocols. For this purpose, the system description needs to be more detailed; a more detailed task graph gives more accurate information about the timing and resource requirements of jobs and other relevant characteristics. Similarly, a more detailed resource graph gives more accurate information about the resources. PERTS will produce sample task assignments, schedules, memory layouts, etc., to provide the feedback needed in the iterative prototyping process. PERTS will also include program execution time analysis and measurement tools. In the later stages of development, as the source code of the target task system becomes available, these tools can be used to extract job parameters and dependencies from the code.

PERTS also provides a simulation environment which will allow a thorough evaluation of the target system. The most concrete description is the instrumented object code. This code can run, under the scheduling directives produced by the schedulability analyzer, in a simulated target environment provided by the testbed. The testbed will contain a workload generator capable of generating synthetic and trace-driven workloads to support the simulation of the embedded environment.

PERTS can also be used to ease the process of upgrading and re-engineering existing systems. The PERTS schedulability analysis and simulation tools can be used to identify where changes in software or hardware are likely to cause timing constraints to be violated, and to predict system performance for the changes proposed by the designer. By making it easier to ascertain the timing impact of modifications, PERTS can help to reduce re-engineering costs.

5 Schedualability Analysis and Validation

The PERTS schedulability analyzer uses, as much as possible, analytical techniques that are based firmly on scheduling theory. These techniques have an advantage over simulation and testing in terms of reliability and cost.

5.1 Challenges in Validating Timing Constraints

Figure 5 illustrates the difficulty and complexity we are likely to encounter when trying to validate timing constraints by means of simulation. The simple task system in this example contains 4 independent jobs, and the underlying system contains 2 identical processors. The ready times and deadlines are as listed in the table. The execution times of all the jobs are fixed except for job J_2. Its execution time can be any value in the range [2, 6]. The scheduling algorithm used is preemptive and priority-driven. A scheduling algorithm is ***priority-driven*** if it does not leave any resource idle intentionally. Such an algorithm can be implemented by assigning priorities to jobs and placing all jobs ready for execution in a queue ordered by their priorities. Whenever a resource is free, it is allocated to the job with the highest priority among all ready jobs. Almost all commonly used event-driven scheduling algorithms, such as the FIFO, LIFO, shortest-processing-time-first, earliest-deadline-first, and rate-monotonic algorithms are priority-driven. In this example, the priority order is J_1, J_2, J_3, J_4 with J_1 having the highest priority. A constraint is that jobs are *not migratable*. In other words, once a job begins execution on a processor, it is constrained to execute on that processor until completion. We want to validate that all deadlines can be met by simulating the system. A naive way is to simulate the system twice: when the execution time of J_2 has the maximum value 6 and when it has the minimum value 2. The results are the schedules shown in parts (a) and (b) of Figure 5. By examining these schedules, we would conclude that all jobs can complete by their deadlines. However, this simulation test does not give us full coverage. This fact is illustrated by the schedules in parts (c) and (d). The worst-case schedule is shown in (c); the completion time of J_4 is 21 when the execution time of J_2 is 3. If jitter in the completion time of J_4 is also important, we would need to know its earliest completion time. We note that in the best case, shown in (d), J_4 completes at time 15 when the execution time of J_2 is 5. To find the schedules in (c) and (d) by simulating the system, we need to exhaustively try all possible execution times of J_2.

The phenomenon illustrated by Figure 5 is known as a ***scheduling anomaly***, the unexpected behavior exhibited by priority-driven scheduling algorithms. Graham [Gra69] has shown that the completion time of a set of jobs can be later when more processors are used to execute them and when jobs have shorter execution times and fewer dependencies. When jobs have arbitrary release times and share nonpreemptable resources, scheduling anomalies can occur even when there is only one processor and the jobs are preemptable. These anomalies make it difficult to ensure full coverage in simulation whenever there are variations in job execution time and resource requirements, as well as jitters in job release times. Unfortunately, these variations are often unavoidable. For an arbitrary scheduling algorithm, there is no efficient way to find the worst-case completion time of each job. It is impractical to find the worst-case completion times of all jobs in a large sys-

	r_i	d_i	$[c_i^-, c_i^+]$
J_1	0	10	[5, 5]
J_2	0	10	[2, 6]
J_3	4	15	[8, 8]
J_4	0	20	[10, 10]

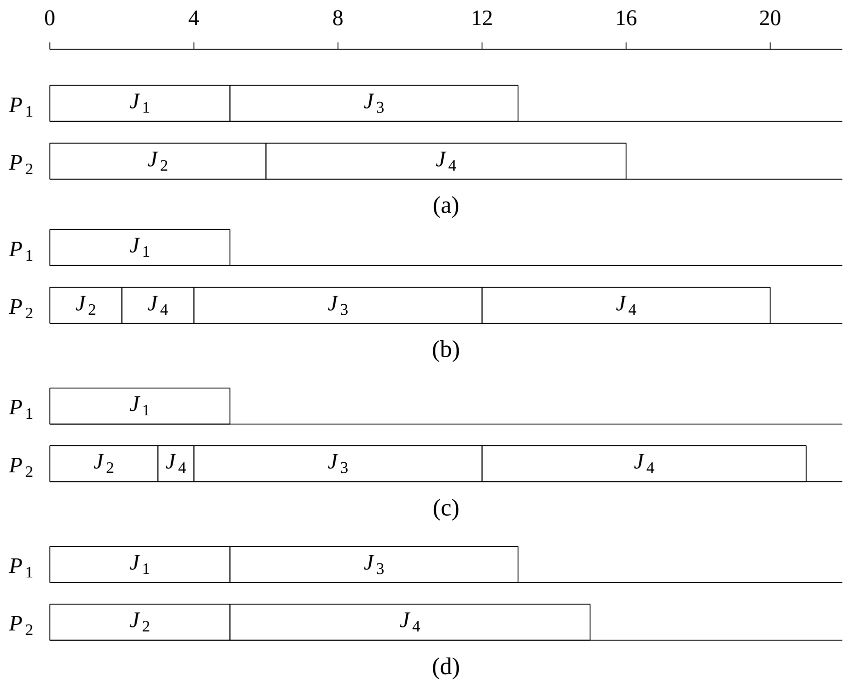

FIGURE 5: An Illustrative Example

tem by exhaustive simulation. Because of difficulties in validation and certification, modern event-driven scheduling algorithms have not been used in safety-critical real-time systems until recently, and then only in systems characterizable by workload models that support rigorous analytical methods.

Recent advances in scheduling theory have made the validation and certification of real-time systems that use certain priority-driven algorithms tractable. In particular, there are now rigorous analytical methods or efficient algorithms for bounding the worst-case completion times of jobs in systems that are characterized by the periodic-task model. We call the conditions that allow us to determine whether every deadline is met under all normal operating conditions ***schedulability conditions***. These conditions constitute the theoretical basis for the basic schedulability analyzer that is now available.

Although the known schedulability analysis methods and conditions currently used in PERTS are derived based on the periodic-task model, they in fact can be modified easily and applied to all systems in which jobs are statically bound to processors, as opposed to being dynamically dispatched to available processors. In particular, it is known that the periodic-task model and the schedulability tests based on the model are robust. Even when some of the assumptions of the model are not valid, the conclusions of schedulability tests in the framework of the model often remain correct. For example, if tasks are scheduled according to a fixed-priority-driven algorithm, it is not necessary for jobs in periodic tasks to be released periodically. A schedulability test based on known schedulability conditions assumes that the job being analyzed is released at a worst-case instant. Consequently, jitters that delay the release times of jobs but do not increase the incremental demand for processor time will not invalidate the conclusion that all periodic tasks are schedulable. It is also not necessary for the jobs to execute for exactly their worst-case execution times. Unlike the example in Figure 5, a schedulable conclusion obtained by using the worst-case execution times of all jobs remains true even when the execution times of some jobs are in fact shorter.

5.2 Features of the Basic Version

The version of PERTS [LRD+93] that is currently available supports arbitrary static-priority algorithms, including the rate-monotonic and deadline-monotonic algorithms [LL73] [LW82], as well as well-known dynamic-priority algorithms such as the earliest-deadline-first algorithm. The supported resource access control protocols include the non-preemptive critical section approach, the priority-ceiling protocol [SRL90], and the stack-based protocol [Bak90]. The priority-ceiling protocol has been extended to handle multiple units of resources. Aperiodic tasks can be scheduled according to a variety of approaches, including pure or persistant polling, and sporadic server [SSL89].

Each part of the interaction between the user and the schedulability analyzer is called a dialogue. The three dialogues are System Analysis, Node Analysis, and End-to-End Analysis. ***System Analysis*** is the main dialogue. Schedulability analysis of a multiprocessor system begins here. Its objectives are (1) to help the user to assign tasks to nodes and to partition resources among nodes and (2) to show the schedulability results of the complete system. The user may initiate a Node Analysis dialogue to analyze the tasks and resources assigned to a selected node.

The term **node** refers to a computer. During ***Node Analysis***, the analyzer proposes a server for each aperiodic task on the node, displays a short summary of schedulability results on all tasks that are bound to the node, and allows the user to initiate new dialogues. With these dialogues, the user can obtain detailed information about processor time usage, resource contention, and average response time. For example, the user may want to tune the average response time of some aperiodic tasks. This adjustment can be made in several ways, including changing the types of the servers and modifying the sizes of the servers. While choosing the types and parameters of the servers, the user may want more accurate estimations of the average response times. The ***Aperiodic Tasks*** dialogue allows the user to design a simulation experiment and start a simulation process in the background. Upon its completion, the user can visualize the simulation result, and after viewing the results, set the server's parameters to those used in the experiment. As another example, in the ***Time Demand*** dialogue, the analyzer presents the schedulability results graphically to help the user gain insight into why the task system is schedulable or not schedulable, how much slack time the tasks have, etc. If the task system is not schedulable, the user can ask the analyzer to propose changes to the job and resource parameters in order to make it schedulable.

Both Node-Analysis and System-Analysis dialogues offer a node-oriented view of the system under consideration. The user looks primarily at nodes and sees the tasks that are assigned to them. In contrast, a task-oriented view provides the user with a picture of the tasks in the system, together with the information about the node to which each task is assigned. In some cases a task-oriented view of the system is required. This is especially true in the analysis of multiprocessor and distributed systems in which tasks execute in turn on different nodes. In PERTS, the task-oriented view of the system is provided in the End-to-End Analysis dialogue. The ***End-to-End Analysis*** dialogue provides information on whether each job will always meet its end-to-end deadlines, and gives its worst-case completion time on each processor on which the job executes. The user can choose to provide the immediate release time and deadline on each of the processor. Alternatively, only the end-to-end release times and deadlines are given; the analyzer is left to assign the individual intermediate release times and deadlines.

6 Simulation Environment

The PERTS simulation environment is called DRTSS. Again, the overall goal for DRTSS is to allow the user to model the widest possible variety of real-time systems in a natural manner and at the desired level of detail. Specifically, it supports the simulation of complex distributed real-time systems with many different active resources such as CPUs, networks, disk drives, graphical displays, and so forth. These resources are typically scheduled according to different algorithms, and the granularity of units of work scheduled on them may vary over a wide range. DRTSS should be useful as a tool for exploring the interaction between scheduling algorithms on the various resources. DRTSS should also allow the user to account for the overhead of scheduling algorithms and the effects of their implementations in a realistic way. A factor in scheduling overhead is the time and resources used to carry out scheduling activities. The user should be able to simulate the strategy used by a scheduler to schedule its own scheduling activities.

DRTSS differs from most other simulation frameworks in that it does not take the traditional queueing theoretical view of the target system. Instead, the target system is modelled by PERTS task graphs and resource graphs

A DRTSS simulator consists of several objects. The ***driver*** controls the overall execution of the simulator. Each active resource has an associated ***microkernal*** that interfaces it to the rest of the simulator. Associated with each resource may be one or more ***scheduling algorithms*** that control the allocation of the resource to tasks. An **`event`** marks the occurrence of a scheduling-related activity.

A “software backplane” interface allows all manner of scheduling algorithms to plugged into a DRTSS simulator. The user can add custom scheduling algorithms and resource access control protocols that follow the protocol of the interface. PERTS also supplies most common modern scheduling algorithms and resource access control protocols, so users will not be required to implement these algorithms. Tasks that have been implemented (i.e., those tasks for which the user supplies executable code) can actually execute. In this way, DRTSS allows the emulation of parts of the software system.

The set of primitive events can be viewed directly. However, in most cases large volumes of low-level event information is neither required nor desired. Usually we are interested in counts of the occurrences, or probability distributions of the intervals between occurrences. We are also interested in detecting particular patterns of primitive events that have particular temporal relationships, which we call compound events. We are developing CELL, a language based on first-order logic, to facilitate the recognition of compound events and the extraction of pertinent timing information.

The output event list provides a clean interface between the simulator proper and output processing tools, such as analysis and display tools. The simulator and the event list processing tools communicate via Unix sockets. Using event list processing tools, we can graphically display the event list as a schedule, save it as a text file to examined by hand later, input it to an analysis program, etc. We plan to provide several performance analysis tools written in CELL.

As of this writing, DRTSS is in the early stages of implementation. It is being written in C++, will run in the Unix/X Window System environment, and will have an OSF/Motif based user interface, as do all modules of PERTS.

7 Future Work

We are implementing the components of PERTS incrementally in the C++ programming language. Again, the basic schedulability analyzer, together with graphical editors and a compiler needed to enter task and resource graphs graphically or textually, is now available. We are implementing the PERTS simulation environment. We also have been evaluating different approaches and methods for automatic extraction of information on processing time and resource usage of software modules written in annotated C++.

We are developing rigorous conditions and performance bounds that are theoretical underpinnings of reliable and efficient validation strategies for systems that do not conveniently fit in the framework of the periodic-task model. An example is a multiprocessor system in which each ready job is placed in a common queue and can be dispatched and scheduled on any available processor. Some of the preliminary results are described in [LH]. Future enhancement of the schedulability analyzer to predict the timing properties of general systems will be built on these conditions.

8 Acknowledgements

This work was partially supported by ONR Contract number N00014-J-92-1815.