Requirements Eng (2004) 9: 121–131
DOI 10.1007/s00766-004-0190-8

ORIGINAL ARTICLE

C. S. Shih · J. W. S. Liu

Acquiring and incorporating state-dependent timing requirements

Received: 3 July 2003 / Accepted: 15 August 2003 / Published online: 19 February 2004

Abstract Some real-time systems are designed to deliver services to objects that are controlled by external sources. Their services must be delivered on a timely basis, and the system fails when some services are delivered too late. In general, the timing requirements of the system may change when the states of the objects monitored by the system change. Such a system may fail if the timing requirements which it is designed to meet are erroneous. It may underutilize resources and consequently be costly or unreliable if the requirements are too stringent. Hence, one must identify how changes in object states call for changes in system requirements and how these changes should be incorporated into the design and implementation of the system. This paper first describes a methodology to determine timing requirements and to take into account requirement changes at runtime. The method is based on several timing requirement determination schemes. Simulation data show that these schemes are effective for applications such as mobile IP hand-offs. The paper then discusses how to incorporate this methodology in the system architecture and in the development process.

Keywords Real-time requirements · Requirement capture and update · Real-time software architecture

1 Introduction

A real-time system has time-critical jobs. The timing requirements of the system are often specified in terms of the deadlines of these jobs. The deadline of a job is the time instant by which the job is required to complete. The system fails if some job completes after its deadline.

Typically, the timing requirements of a system are derived from its high-level functional requirements. For example, a feedback controller must keep the motion of the plant it controls smooth and stable. This functional requirement in terms imposes timing requirements on the jobs that transmit sensor data, compute control laws, and send commands to actuators: They must complete within each sample period for the plant to operate smoothly. Several consecutive failures of in-time completion may result in runaway behavior of the plant.

A common practice is to impose on the system a set of fixed and known timing requirements (or a set for each operation mode of the system). Indeed, traditional real-time systems (e.g. [1, 2, 3, 4, 5, 6]) assume that the timing requirements are given and do not vary with time. This assumption is valid for systems such as a feedback controller. (Time parameters of such a controller are typically chosen by design. Hence, its timing requirements are known.) The assumption is sometimes not valid for real-time systems designed to deliver services to objects that are controlled by external sources. The timing requirements of the delivery services may depend on the state of the objects, and the states of the objects change with time. As a consequence, the timing requirements of the jobs in the system may change with time. By assuming that the deadline of every job is known and constant, a real-time system may underutilize its resources. More seriously, the system may fail to meet its high-level functional requirements because the constant deadlines do not correctly capture its high-level functional requirements.

An example is information delivery to on-board devices in an intelligent transport system. In such a system, users may subscribe to message delivery services to receive alert messages and traffic information. (Alert messages notify individual subscribers about unexpected traffic conditions, e.g. a traffic accident on the planned route of a subscriber.) The high-level functional requirement of the system is to deliver alert message(s)

C. S. Shih
Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, IL, 61801, USA

J. W. S. Liu (✉)
Microsoft Corporation, Redmond, WA, 98052, USA
E-mail: janeliu@microsoft.com

to each individual subscriber so that the subscriber can avoid the delay caused by unexpected obstacles and congestions.

Traffic information messages are transient and location dependent. As an example, a message on a traffic accident is relevant and important as long as the hazard and congestion caused by the accident persist. Once the accident scene is cleaned up and congestion cleared, the message no longer needs to be delivered. The message is location dependent in the sense that it is only relevant to subscribers who plan to travel to or through the accident site. The information carried by such a message is helpful only if the message is received by the subscriber before he reaches the last detour point between him and the accident site. The time that the subscriber will take to reach the last detour point depends on the state of the subscriber's vehicles, i.e. the speed and acceleration of the vehicle, its distance to the last detour point, etc. Consequently, the deadline for delivering the message is not a constant value but a function of the state of the vehicle. Without considering the state of every vehicle, the system may not deliver some messages in time. A late accident alert may cause the subscriber to be trapped in traffic.

Mobile IP hand-off is another example. As a device moves in a mobile IP network, its network connection must migrate from cell to cell. For this purpose, a hand-off request for a device is triggered whenever it moves across cell boundaries. Upon acknowledging the request, the new base station (BS) becomes the new agent of the mobile device. Packets for the device are then forwarded to the new BS. The high-level functional requirement of the system is to keep the mobile devices connected.

Each hand-off request should be completed by a deadline. In a network that supports hard hand-off, the deadline of a request is the time at which the device leaves the previous cell. When the network supports soft hand-off, the previous BS buffers packets for the device after the device moves out of its coverage area until the hand-off request to the new BS is acknowledged. In this case, the deadline is the time at which the previous BS must discard packets sent to the device to prevent network congestion. In both cases, the deadline for processing a hand-off request for a device depends on the motion of the device. As the speed and trajectory of a mobile device change, the deadline of its request changes accordingly. Traditionally, such changes are ignored in the process of determining low-level timing requirements of the system. As a result, the system does not meet some high-level functional requirements, despite that all jobs in it complete according to their chosen deadlines. For this particular application, the failure to process some hand-off requests in time leads to performance degradation, causing packet and connection losses.

We describe here a methodology for designing real-time systems which are able to track their state-dependent timing requirements. To characterize changes in timing requirements, we developed a new real-time task model, called the *state-dependent deadline model* [7]. Based on the new task model, we develop a set of deadline determination algorithms. These algorithms allow low-level timing requirements of real-time systems to be derived accurately from their high-level functional requirements. We propose a software architecture for real-time systems that incorporates requirement capture so that the system can track and meet its state-dependent timing requirements.

Following this introduction, Sect. 2 presents the related works. Section 3 describes the new state-dependent deadline model and defines the terms used here. Section 4 presents two approaches to acquiring the information required to determine the state-dependent requirements. The section also describes several deadline determination algorithms and their performance. Section 5 discusses how to incorporate requirement determination in the software architecture and design process. Section 6 summarizes this work.

2 Related works

In this paper, we address the problem of how to determine timing requirements for the scheduler to meet with the required response times of jobs. The variations arise from external causes that lead to changes in low-level requirements of the system. At a quick glance, one may relate this problem to control problems. This relationship is weak. In our problem, the scheduler (and the real-time system) has no control over the state changes of external objects and, hence, has no control over changes in timing requirements of jobs. It must either anticipate and estimate the changes and set scheduling goals accordingly, or observe and adapt to the changes and attempt to meet some continuously changing goals. In this sense, our problem resembles problems on overload handling: The scheduler has no control over which job may overrun its allocated time.

Past works on real-time scheduling assume that the relative deadlines or temporal distance constraints required of the system are time invariant. According to some scheduling algorithms, a scheduler may dynamically change relative deadlines of jobs as a way to tune system performance [1, 8, 9, 10]. For example, scheduling algorithms based on the elastic task model proposed by Buttazzo *et al.* [1] adjust the deadlines of real-time tasks based on the system workload. When the system is underloaded, the system decreases the periods of tasks to provide better quality of service (QoS). When the system is heavily loaded, the system increases periods of tasks to downgrade the QoS to the acceptable level and maintains the system load below the achievable utilization factor. Caccamo *et al.* [9] proposed the elastic feedback model and the associated task scheduling algorithms. Their goal is to optimize the performance of control applications in which jobs have widely varied execution times. According to their approach, the amount of resource reserved for each task is based on the average execution time, rather than the worst-case execution time. Whenever a job overruns, the system extends its deadline such that the utilization factor is no greater than the reserved utilization factor. Hence, each job can complete its work without jeopardizng other tasks in the system. Invariably, however, these studies assume that the scheduler has control over when and by much the deadlines are changed. Because of this fundamental difference, our work is only loosely related to these past works.

Our model and objective resemble the probabilistic models and deadline guarantees studied by the real-time community. An example is the model by Abeni and Buttazzo [11]. In their work, the arrival period and the execution times are given as some probability distributions, rather than the worst-case values. Hence, the deadline of every job in a task may be different but is constant over time.

This work is also related to the works dealing with requirement evolution [12, 13, 14]. Agile software development [12] develops a set of system design principles. The principles allow the developers to efficiently adapt requirement changes in the software development. Earlier works dealing with requirement evolution such as [13] and [14] are concerned with the change of requirements during the development phase of systems. However, in our work, we are concerned with the change of requirements in the operation phase of systems. Lutz and Mikulski [13] show that the requirements evolve in the design and operation phase of systems. In our problem, we focus on the methodology of designing a system to tolerate changes of timing requirements so there is no need to modify the system when the timing requirements change.

3 Background

3.1 Real-time jobs and tasks

Real-time systems are designed to complete jobs on a timely basis and, thus, to meet their high-level functional requirements. A *job* is an instance of computation, or the transmission of a data packet, or the retrieval of a file, and so on. In short, a job refers to a unit of work when it is not necessary for us to be specific about the work. We call jobs J_1, J_2, and so on. A real-time job must be completed by its deadline. The job may produce an incorrect result if it does not complete by its deadline; the system fails some high-level functional requirement(s) when this happens. A *task* is a sequence of jobs that have identical or similar characteristics and timing requirements.

Each job is characterized by its temporal parameters, including its release time, execution time and deadline. The *release time* of a job is the instant of time at which the job becomes available for execution: The job can be scheduled and executed at any time at or after its release time. We say that a job arrives when it becomes known to the scheduler. In general, the release time of a job is equal to or later than its arrival time. A job is said to be *eligible* in the time interval from its arrival time to the instant of time when the job completes. The *execution time* is the amount of time required to complete the execution of the job when it executes alone and has all the resources it requires. The *absolute deadline* of a job is the instant of time by which its execution is required to be completed. In many cases, it is more natural to state the timing requirement of a job in terms of its *response time*. We call the maximum allowable response time of a job its *relative deadline*. Except for where it is stated otherwise, by deadline we mean relative deadline and denote it by d. The deadline of a job is the primary timing requirement of the job, and we focus on this requirement hereafter.

3.2 State-dependent timing requirements

As stated earlier, the state-dependent deadline model intends to capture the time-varying nature of timing requirements of jobs in systems exemplified by the ones described in Sect. 1. In such a system, the deadlines of jobs depend on states of objects (e.g. vehicles) external to the real-time system. We call the states of external objects *object states* for short. As the states of some external objects change with time, the deadlines of jobs also change with time. We call a job whose deadline changes as object states change a *state-dependent deadline job*. Hereafter, we omit "state-dependent deadline" as long as there is no ambiguity.

The simplest case is when the external object states change in a predictable way for all times. Consequently, the relative deadline of every job is a known time function. We call this function the *deadline function* of the job. We use $d_i(t)$ to denote the relative deadline of job J_i at time t where t is the elapsed time since the release time of the job.

The assumption that all jobs have known deadline functions is often not valid. In general, we characterize the relative deadline of each job J_i at time t since the release of the job by a random process $\delta_i(t)$. The probability density function (PDF) of sample deadline d of $\delta_i(t)$[1] at time t is denoted by $g_i(t, d)$.

Figure 1 illustrates a possibility. The time origin is the release time of the job. A point (x, y) in this figure denotes that the relative deadline of the job at time x is y, where x is the elapsed time since the release time of the job. The shaded area shows the range of values of the job deadline at different time instants. The 45° line $y = t$ in the figure illustrates the passage of time. Suppose that when a subscriber requests for accident alert messages, there is an accident on his planned route and his travel time to the last detour point before the accident site is

[1] For each job J_i, $\delta_i(t)$ is an indexed set of random variables, where the indexed set is the set of positive real numbers. Hence, by definition, $\delta_i(t)$ is a random process. We do not assume any property of the random process.

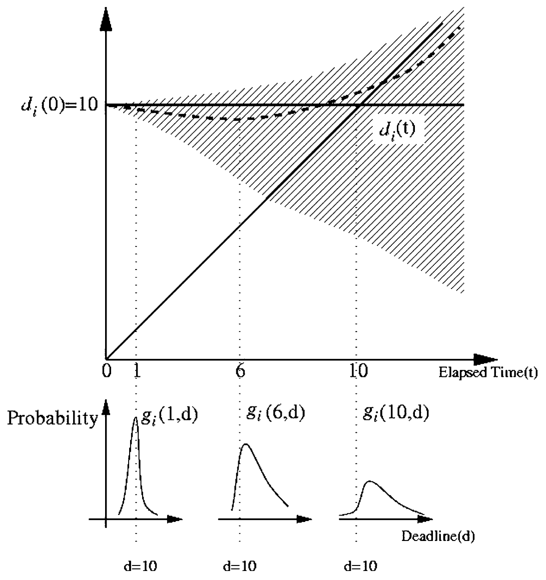

Fig. 1 Deadline random process

10 min. If he were to travel at a constant speed, the deadline function of his message delivery job is $d_i(t) = 10$, which is the horizontal line at 10 in the figure. Because he may speed up or, more likely, slow down, the actual deadline may be the sample deadline function depicted as the heavy dashed line in Fig. 1. (He first speeds up, causing the deadline to be shorter. The sample function $d_i(t)$ intersects $y = t$ at the instant when he reaches the last detour point. If he continues on the same route after he passes the last detour point, he slows down as he comes closer to the accident site and, eventually, he stops. Thus, the deadline sample function becomes parallel to $y = t$; the deadline is always smaller than the elapse time; and the alert message is late and of no help to him.) The scheduler does not know the value of this sample function. However, based on the initial location and speed of the subscriber, the system can retrieve, from its stored statistical data on travel time, a set of time-dependent histograms and use the set as an estimate of the probability density function $g_i(t, d)$ of the random process $\delta_i(t)$ at different values of t. To illustrate, Fig. 1 shows the PDF at three different values of time. In this case, the subscriber has a high probability of slowing down and a very small probability of speeding up as time progresses. Moreover, the uncertainty in his speed and time to the last detour point before the accident site grows with time.

3.3 Timing constraint

According to the traditional definition, a job meets its timing constraint (or the job is timely) if it completes by its deadline. This definition of timeliness needs to be generalized when the deadline changes with time. To illustrate, let us consider a job whose deadline function $d(t)$ is known and is as shown in Fig. 2. If the job

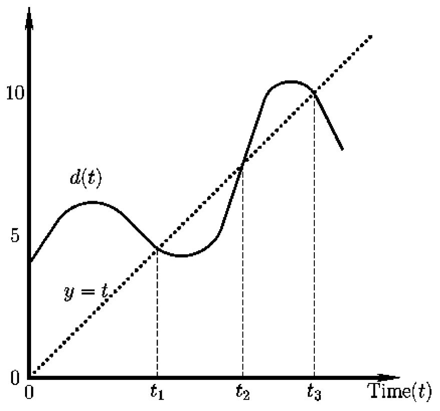

Fig. 2 Sample deadline function

completes at an instant in the time interval $(0, t_1]$ or $[t_2, t_3]$, its response time is less than its relative deadline. Hence, the job completes in time. In contrast, if the job completes in the interval (t_1, t_2), it misses its deadline because its response time is larger than its relative deadline. We say that there is a *deadline miss* at time x if $d(x)$ is less than the elapsed time x. A time interval during which the relative deadline is never less than the elapsed time is called a *feasible interval.* Therefore, intervals $(0, t_1]$ and $[t_2, t_3]$ are feasible intervals, while (t_1, t_2) is not.

An important special case is where the deadline function of every job has only one feasible interval and the interval begins at the release time of the job. This is true for the systems exemplified by the ones described in Sect. 1. We focus on this special case here. For some applications, multiple feasible intervals do exist as exemplified by the deadline function in Fig. 2. The case of multiple feasible intervals will be considered in our future work.

The following definition states the timing constraint of a job in the special case considered here, that is, every job has a known feasible interval of length T for some $T > 0$.

Definition 3.1. In-time completion A job J meets its timing constraint, or simply it completes in time, if and only if there is no deadline miss before it completes. In other words, $d(x) \geq x$ for $0 < x \leq t$ when the response time of the job is t.

Clearly, the response time t of a job that completes in time is within its feasible interval. When the relative deadlines of jobs are random processes, the system cannot provide deterministic guarantee of in-time completion. We use $P_{\text{InTime}}(t)$ to denote the probability that a job completes in time when its response time is t. The most natural generalization of Definition 3.1 is the one below.

Definition 3.2. Probability of in-time completion in the strict sense The probability of a job (with deadline given by a random process $\delta(y)$) completing in time is the probability that there is no deadline miss before the job completes. In other words, $P_{InTime}(t) = Pr[\delta(x) \geq x, \forall\, 0 < x \leq t]$.

For most systems, a weaker constraint according to the following definition suffices; this is a natural generalization of the traditional definition of probability of in-time completion.

Definition 3.3. Probability of in-time completion The probability of in-time completion of a job is the probability that its response time is no greater than its deadline at the time when the job completes. In other words, $P_{\text{InTime}}(t) = \Pr[\delta(t) \geq t]$, *where t is the response time of the job.*

This paper focusses on how to design a system so that the jobs in the system have a high probability of in-time completion according to Definition 3.3. We also want to achieve a high system utilization. *System utilization* is the fraction of time that the system is busy executing jobs in the system.

We assume that the execution time of every job is known when the job arrives. Without loss of generality, we assume that every job is released upon arrival (i.e. its release time equals its arrival time.) Upon the arrival of each job, the system conducts a schedulability analysis to check if the job can meet its timing constraint without adversely affecting eligible jobs in the system. A job is admitted to the system if its probability of in-time completion is no less than a given threshold. Otherwise, the system rejects the job.

4 Determining state-dependent deadlines

4.1 Deadline probability density function (PDF) estimation

For most applications, the deadline function is unknown when a job arrives. There are two approaches to acquiring the probability density function of the deadline random process: Using historical sample data and using simulated data.

4.1.1 Using historical sample data

Probability density functions of deadline functions can be estimated based on sample data when sufficient data are available. Some systems collect a massive amount of statistical data on the values of job deadlines as a function of the elapsed time under various circumstances. These data allow the system to estimate probability density function $g_i(t, d)$ of the deadline random process $\delta_i(t)$ for each job J_i. Such an estimate is obtained from sample values taken from a large ensemble of statistically identical jobs. In particular, the estimate of $g_i(t, d)$ is not obtained from sample values of job deadlines of an individual job.

As an example, we return to the intelligent transport system described in Sec. 1. Because the deadline of each message delivery job is the instant of time when the vehicle arrives the last detour point, we use the travel time of each vehicle to the last detour point as the deadline of the corresponding message delivery job. The travel time of a vehicle is determined by the speed of the vehicle, distance from the current location to the last detour point, traffic density, and so on. These variables collectively describe the state of the vehicle. An intelligent transport system typically collects a massive amount of statistical data on travel times of vehicles as a function of distance to accident sites under various traffic densities. Based on such data, the system can construct an estimate of the probability density function $g_i(t, d)$ of the deadline random process $\delta_i(t)$ for the message deliver job J_i of each subscriber when the subscriber first requests for service. Specifically, the estimate of $g_i(t, d)$ at elapsed time t may be a histogram on travel times of a large ensemble of statistically identical subscribers caught in similar situations, i.e. same remaining distance to the last detour point, same time of the day, same highway, similar traffic density, and so on.

4.1.2 Constructing PDFs by simulations

Another means of estimating the PDF is to simulate conditions that cause changes in deadline values, collect the observed deadline values in the simulation, and use them to estimate the PDF. A simulation model for this purpose should characterize changes in the states of external objects. By simulating changes in object states under various circumstances, the system can generate and collect statistical data on deadlines of statistically identical jobs. Thus, the system can obtain a histogram of the deadline values as a function of the elapsed time for each job.

As an example, the Random Walk Model commonly used in mobile network literature (e.g. in [15] and [16]) can be used to simulate movements of mobile devices in an unobstructed area. According to this model, the motion of each device is determined by three parameters: speed s, direction θ, and duration τ. For each movement, the device moves at speed s in direction θ for τ units of time. These three parameters are randomly generated with given means and standard deviations.

Figure 3 shows one example. The goal is to obtain histograms of deadlines of hand-off requests when the mobile device moves at a low speed. In this example, the speed of the mobile device is normally distributed with mean 14.4 km/hour and variance 0.9.[2] The direction of a movement is normally distributed using the current direction of a device as the mean and 30° as the variance. The duration of each movement is truncated-normally distributed with mean 5 s and variance 1 s. A hand-off request is triggered as soon as a mobile device enters the overlapped area of two adjacent mobile cells. Figure 3a and 3b shows the histogram of

[2] A negative speed means the device moves in the opposite direction.

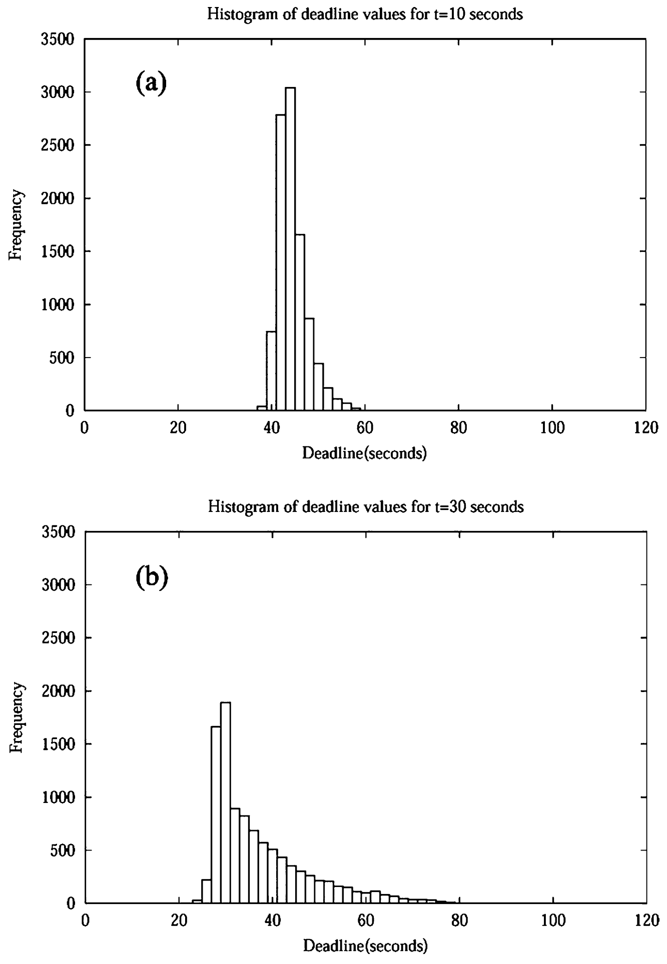

Fig. 3 Histograms of deadline values of hand-off requests

the deadline values when the elapsed time is 10 s and 30 s, respectively. When the elapsed time is 10 s, the job deadlines are within the range [38, 60]. As the elapsed time increases, the range of deadline values increases. In this example, the deadlines are in the range [20, 80] when the elapsed time is 30 s.

4.2 Deadline determination

For most applications, the exact shape of each PDF is not likely to be known or estimated accurately. The amount of statistical data needed to get a high degree of confidence in an estimated distribution function is typically formidably large. (It is even unlikely for an intelligent transport system to acquire the needed amount of data for this purpose.) However, statistical averages such as mean value and variance functions can be estimated reasonably accurately based on statistics taken from an ensemble of relatively small size. For this reason, the deadline determination algorithms discussed below make use only of ensemble averages and variances.

4.2.1 Deadline determination algorithms

There are two classes of deadline determination algorithms [7]: fixed deadline determination and dynamic deadline determination. A fixed deadline determination algorithm chooses a constant relative deadline for each job based on an estimated deadline function of the job. The scheduler uses the chosen deadline as the target, aiming at completing the job within the chosen relative deadline. Algorithms in this class include the worst-case deadline (WCD) algorithm, the mean reference deadline (MRD) algorithm, the mean reference deadline- α (MRD- α) algorithm, and the initial state mean reference deadline (ISMRD) algorithm. The WCD algorithm selects the minimal of all possible deadline values as the deadline of the job; this is traditionally how the job deadline is determined. The MRD, MRD- α, and ISMRD algorithms are similar in their way of selecting the job deadline. They compute the intersection of an estimated deadline function of the job and the reference timeline $y = t$, and use the deadline at the intersection as the deadline of the job. They differ in the way they compute the estimated deadline function. The MRD algorithm uses the mean function of the deadline random process as the estimated deadline function. The MRD- α algorithm shifts the mean deadline function by α times of the standard deviation function and uses the resulting function as the estimated deadline function. The ISMRD algorithm takes into account the object states when a job arrives. Instead of using the mean deadline function, the ISMRD algorithm uses as the estimated deadline function the mean deadline function conditioned on the given deadline value at the instant of time when the job arrives.

Dynamic determination algorithms, on the other hand, try to improve the choice of the relative deadline of each eligible job based on the job's past and current deadline values. The scheduler may adapt the scheduling strategy as the chosen relative deadlines of jobs change. Dynamic deadline determination algorithms assume that the past and current job deadlines are known. In other words, at elapsed time x, the values of deadline function $d(t)$ for $0 \leq t \leq x$ are known. Algorithms in this class include the conditional mean reference deadline (CMRD) algorithm and the conditional mean reference deadline- α (CMRD- α) algorithm. The CMRD algorithm computes the mean deadline function of the deadline random process conditional on the values of $d(t)$ for $0 \leq t \leq x$, and uses the conditional mean function as the estimated deadline function. It then chooses the intersection of the estimated deadline function and the reference timeline as the job deadline. To improve the probability of in-time completion, the CMRD- α algorithm shifts the conditional mean deadline function by α times of the standard deviation function.

The approach to deadline update should be chosen based on the accuracy of the probability density function and the overhead of updates. When the confidence interval of the mean deadline function is relatively small or the overhead is not negligible, the system may use fixed deadline determination algorithms. Otherwise, the dynamic deadline determination algorithms can provide better performance.

4.2.2 Relative merits

The algorithms described above were evaluated in a simulation experiment [7]. They were found effective in determining state-dependent job deadlines. As expected, the deadline miss rate can be reduced at the expense of processor utilization by determining job deadlines in a more conservative manner. These algorithms offer a systematic way to trade off the probability of in-time completion and system utilization.

In general, fixed deadline assignment works well when the deadline functions are known or can be predicted reasonably accurately. An obvious advantage is lower scheduling overhead. The usage of constant deadlines makes it possible for the scheduler to queue each job only once, upon its arrival, for example. For some types of jobs (e.g. message transmissions in some networks), reordering eligible jobs in the ready queue can be prohibitively expensive. Fixed deadline assignment may be the only option. Dynamic deadline assignment is recommended when changes of object states and requirements can be observed and updated with minimal overhead and the additional scheduling overhead can be tolerated.

Among the fixed deadline determination algorithms, the ISMRD algorithm outperforms the others in most simulated cases. The ISMRD algorithm estimates the job deadline with the minimal estimation error and achieves a higher system utilization in comparison with the traditional deadline determination algorithm, i.e. the WCD algorithm.

The dynamic deadline determination algorithms outperform the fixed deadline determination algorithms when the overhead of reordering jobs in the dispatch queue is negligible. The simulation results show that these algorithms can improve the system schedulability from 15% to 30% under different circumstances.

In summary, the deadline determination algorithms take into account the change of object states. The more accurate low-level timing requirements produced by them allow the system to meet its high-level functional requirements without underutilizing system resources.

5 Software design

To complete state-dependent deadline jobs in time, the developer must take into account the time-varying deadlines while designing such a system. In this section, we first present the process of accepting and executing state-dependent deadline jobs. Then, we present a general software architecture of such a system. Last, we discuss the implementation of an independent module, called *requirement engine*, which is designed to assign deadlines to jobs in the real-time system. (The requirement engine is independent in the sense that it does not use resources in the real-time system.)

Specifically, the real-time system contains jobs with known timing requirements as well as state-dependent deadline jobs. For the sake of clarity, we continue to treat the deadline of each state-dependent job as its only unknown timing requirement. In other words, all the timing requirements of the real-time system are known, except the deadlines of state-dependent deadline jobs. The function of the requirement engine is to determine these deadlines.

5.1 Admission control and scheduling

Figure 4 shows the admission control and deadline update procedure. In this figure, the solid lines and boxes denote the actions and operations in the real-time system; the dashed lines and boxes denote the actions and operations for the requirement engine. For the sake of concreteness, we assume that the real-time system and the requirement engine communicate via messages.

When a job arrives, the system checks whether the job has a valid known deadline. (We will return shortly to explain when the deadline of a job may be known but invalid.) If the job has a valid (known) deadline, it is subjected to acceptance test. As a part of an acceptance test, the system conducts a schedulability analysis to determine whether the newly arrived job can be scheduled to complete in time (with an acceptable probability) without adversely affecting the in-time completion of eligible jobs in the system. If the new and eligible jobs are not schedulable, the real-time system rejects the job. The system accepts the new job if it and eligible jobs are schedulable, i.e. they can all complete in time. The job is then inserted in the dispatch queue according to the scheduling algorithm used by the system. (The flow chart in Fig. 4 states that the new job is inserted in the dispatch queue according to its valid deadline. This statement may give the impression that eligible jobs are scheduled in the earliest-deadline-first (EDF) basis.[3] We do not mean to strict the choice of scheduling algorithms; the real-time system may use any good real-time scheduling algorithm, together with a suitable schedulability analysis scheme, that aims at meeting the timing requirements of all eligible jobs.)

When the new job is a state-dependent deadline job and does not have a valid deadline, the system sends a deadline determination request to the requirement engine. Upon receiving a request, the requirement engine computes and returns a valid deadline for the new job. Associated with each request is a timeout interval. If the timeout expires before the requirement engine returns a valid deadline for the new job, the system rejects the new

[3] This algorithm is optimal for scheduling jobs on one processor.

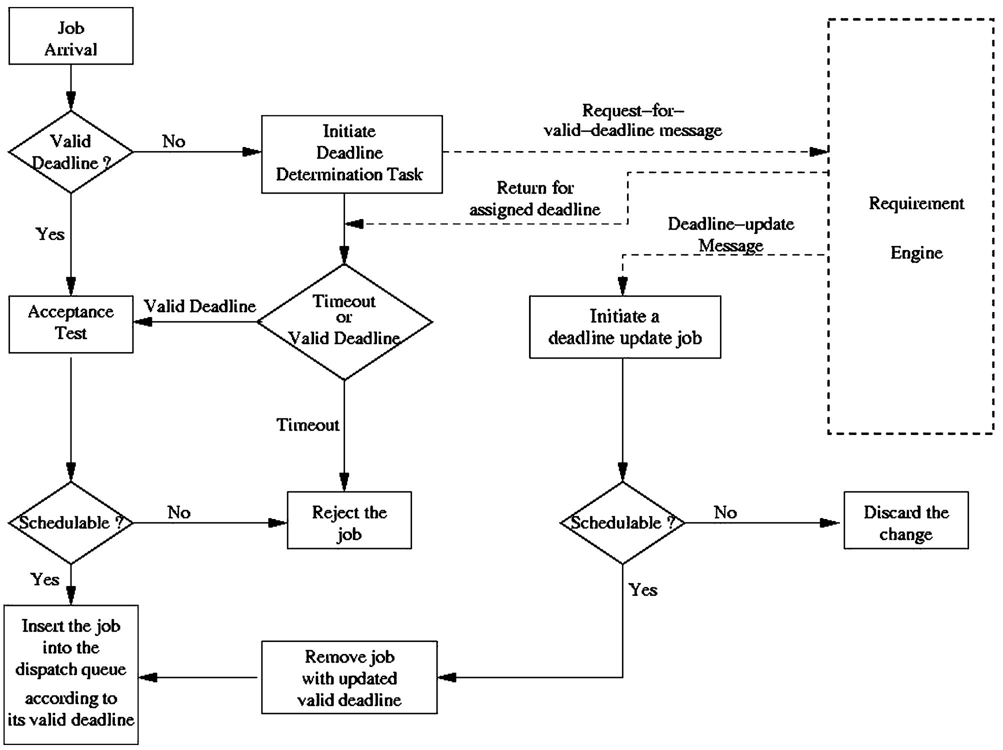
Job Arrival
Valid Deadline ?
No
Yes
Initiate Deadline Determination Task
Request–for–valid–deadline message
Return for assigned deadline
Deadline–update Message
Requirement Engine
Acceptance Test
Valid Deadline
Timeout or Valid Deadline
Timeout
Initiate a deadline update job
Schedulable ?
No
Reject the job
Yes
Insert the job into the dispatch queue according to its valid deadline
Remove job with updated valid deadline
Schedulable ?
No
Discard the change
Yes

Fig. 4 Flow chart of accepting state-dependent deadline jobs

job. If the requirement engine returns a valid deadline in time, the system proceeds with an acceptance test on the new job according to the valid deadline of the job, and accepts or rejects the job in the way described above. When a state-dependent deadline job is accepted, the system registers the job with the requirement engine if the engine uses a dynamic deadline determination algorithm. (This part is not shown in Fig. 4.)

When the requirement engine uses a dynamic deadline determination algorithm, the deadline of a state-dependent deadline job may change before it completes. When such a change occurs, the requirement engine initiates a deadline update job (which is a system job with a constant and short relative deadline.) In general, arbitrary deadline changes may overload the system and cause some jobs to miss their deadlines. Hence, the system allows changes only if a schedulability analysis of all eligible jobs with updated deadlines succeeds. Otherwise, the changes are ignored. Whenever the valid deadline of a job is changed, the job is removed from the dispatch queue and reinserted according to the new valid deadline.

When a job completes and dynamic deadline determination algorithms are used, the system sends a message to the requirement engine to unregister the job. The message stops the requirement engine from continuously updating object states and timing requirements of the completed job.

5.2 System architecture

As stated earlier, the requirement engine is independent from the real-time system in the sense that the engine does not compete for resources with jobs in the real-time system and its operations are not under the control of the real-time system scheduler. The interaction between the system and the requirement engine are accomplished by message passing. In fact, a requirement engine may serve more than one real-time system. In addition to the deadlines of state-dependent deadline jobs, it can be generalized to provide other time-varying requirements. The architecture described here allows these generalizations naturally.

Figure 5 shows the overall architecture. The upper portion of this figure shows the components of the real-time system; the lower portion shows the requirement engine. The thin and thick directed lines represent the interactions within the real-time system and to/from the system, respectively. In particular, the thick dashed lines represent the interactions between the real-time system and requirement engine.

We have already described the function of the admission controller and the queuing of admitted jobs in the dispatch queue. The real-time system also maintains a pending queue. Each state-dependent deadline job is placed in the pending queue by the admission controller when it waits for a valid deadline. The job is removed from the pending queue when the requirement engine assigns a valid deadline or when the timeout associated with the deadline determination request for the job expires, whichever occurs first. Figure 5 shows that the real-time system treats a job removed from the pending queue when its deadline is assigned as a new arrival with a valid deadline. On the other hand, when timeout expires, a job removed from the pending queue is rejected.

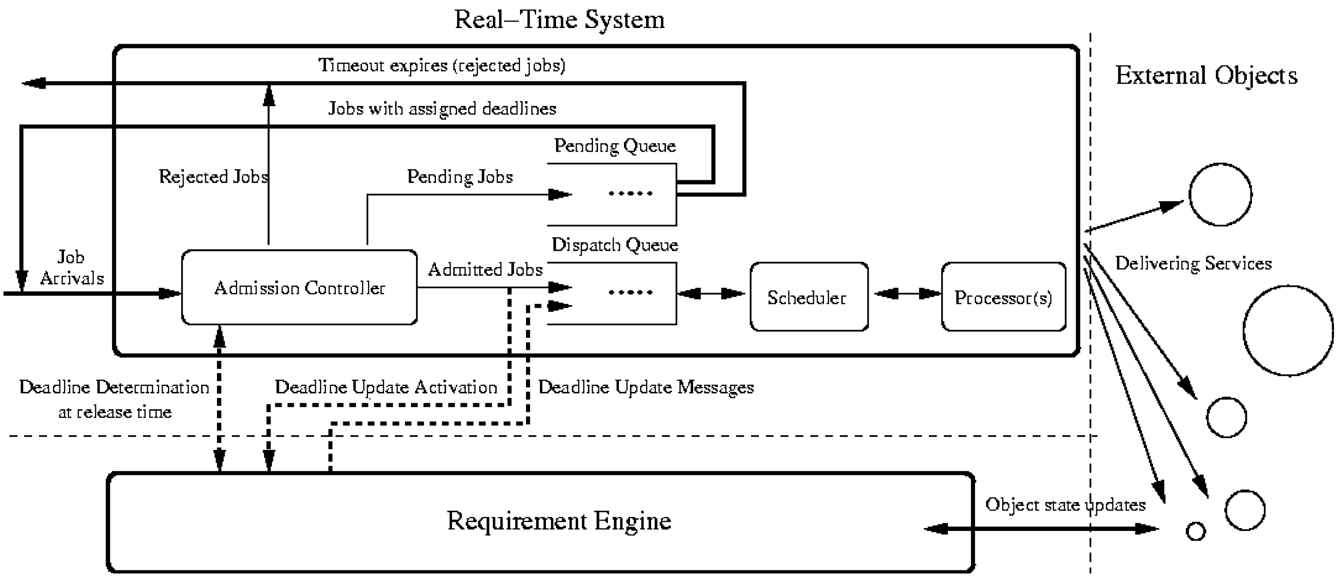

Fig. 5 System architecture

This figure also shows notifications from the requirement engine whenever the deadline of a registered job changes. When a job completes, the requirement engine is notified to unregister the completed job. These interactions were described earlier.

5.3 Requirement engine

The requirement engine consists of four modules: the job registration module, the statistical database, the object state update module, and the deadline determination module. These modules and their interactions are shown in Fig. 6. Each box and directed line represent a logic component and data flow between components, respectively. With these four modules in a requirement engine, the real-time system will be aware of the changes of time-varying deadlines during the runtime. Thus, the system can adapt to changes in deadlines of state-dependent deadline jobs to meet their high-level functional requirements.

The job registration module maintains records for registered jobs. In this architecture, the requirement engine supports both periodic time-triggered and sporadic event-triggered requirement updates. The job registration module is responsible for triggering deadline updates when the timing requirements are time-triggered periodically. The module also keeps track of the latest valid deadlines of all registered jobs.

The statistical database stores the statistical data on the deadline random processes. Such data include estimates of PDFs and statistical average and standard deviation functions of the random processes. The information is used by the engine in computations of valid deadlines.

The object state update module acquires the object states on behalf of the deadline determination module. Acquisitions can be either active or passive: The object state update module sometimes actively probes the states of the external objects and notifies the requirement determination module when non-negligible changes in object states occur. Indeed, the module actively probes the job states whenever it receives a request from the determination module. For instance, the object state update module in an intelligent transport system may actively probe the state of a vehicle periodically by sending a message to the global positioning system (GPS) device on-board the vehicle. Probing is said to be passive when the module receives updates of object states from external objects. For instance, in a mobile IP network, a mobile device may periodically send the "I-am-live" message to the base station. The signal-to-noise ratio of the message can be used to estimate the state of the mobile device.

The deadline determination module is an interface between the requirement engine and the real-time system. It accepts deadline determination requests from the real-time system and returns valid deadlines to the system. When a deadline determination request message for a state-dependent deadline job arrives, the job is first registered by the job registration module. (A state-dependent deadline job is unregistered when the job is rejected or completes in time.) Then, the module initiates a deadline determination task: Each instance of the task computes a valid deadline for the state-dependent deadline job. The deadline determination task first queries the object state update module for the object states of the job and selects an estimate of the PDF (or statistical average and standard deviation functions) of the deadline random process of the job. A valid deadline is then computed using one of the algorithms described in Sect. 4. The computed valid deadline is returned to the real-time system for use in schedulability analysis.

The timing parameters of the deadline determination task depend on the deadline determination algorithms used in the requirement engine. When a fixed deadline

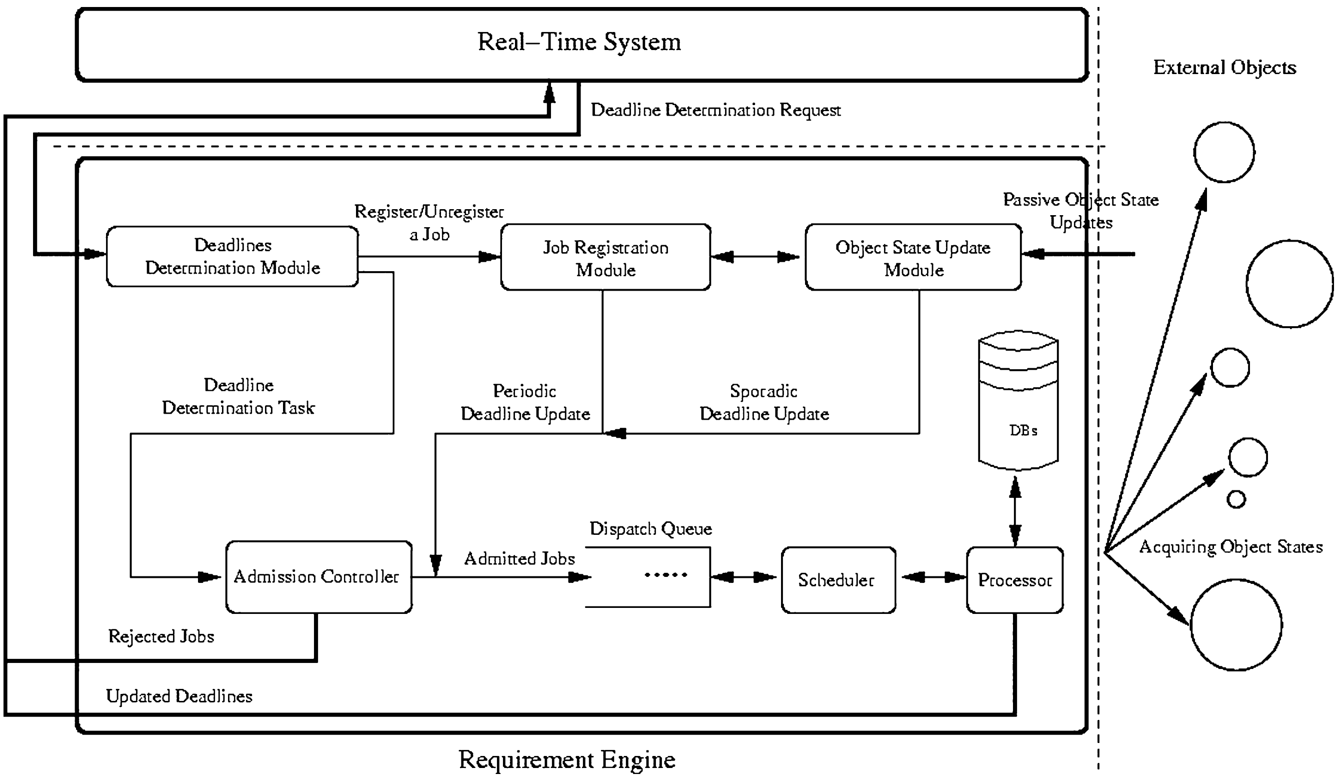

Fig. 6 Requirement engine

determination algorithm is used, the deadline determination module initiates an aperiodic task, which has only one instance. On the other hand, when a dynamic deadline determination algorithm is used, the module initiates a periodic or sporadic task. The task repeatedly recomputes and updates the valid deadline of each state-dependent deadline job until the job is unregistered from the system. When the system requests for periodic monitoring and update of the deadlines of some jobs, a periodic task is created for each job so that valid deadline of the job is updated once every constant time interval. Periodic update of valid deadlines allows the system to estimate accurately the amount of resources required to process the updates. (Hence, the admission controller will not falsely admit too many real-time jobs and overload the system.)

When the system chooses to have valid deadlines updated only when the deadlines may have changed more than some thresholds, the deadline determination module creates sporadic tasks to do this work. Sporadic update of valid deadlines allows the system to allocate the resources more effectively. Each instance of such a sporadic deadline determination task is initiated by a message from the object state update module when a valid deadline update becomes necessary (e.g., whenever object states have changed by a specific threshold.) A minimal inter-arrival time is set for every sporadic task so that the change of object states within such a time interval will never be large enough to trigger a valid deadline update. Overload caused by sporadic deadline updates can be avoided by using a sporadic server approach [4, 17] to allocate resources for the sporadic updates.

After a deadline determination task is created, the admission controller conducts a schedulability analysis to make sure that the new deadline determination task and other tasks in the engine will complete in time. The new task is accepted only if the schedulability analysis succeeds. Otherwise, the task is rejected and a failure of the deadline determination request is reported. Accepted deadline determination tasks are scheduled by some real-time scheduling algorithm such as the Rate-Monotonic or EDF algorithm to complete by their deadlines.

6 Summary

We presented here a method for designing real-time systems so that they can better meet their high-level functional requirements. In particular, the low-level timing requirements of a real-time system may change over time. A new task model is developed to characterize real-time jobs whose timing requirements may change over time. We described two approaches to acquiring statistical data on the deadline random process before the system starts. Given the statistical data, many determination algorithms can be used to determine the timing requirements of each job when the job arrives and before the job completes. We also propose an architectural framework for real-time systems that consider changes in timing requirements. A requirement engine is added to acquire object states and to determine timing requirements. The requirement engine allows the real-time system to track changes in its timing requirements and adapt the scheduling strategies it uses to meet the requirements.

Acknowledgements This work is supported in part by a grant from the MURI program N00014-01-0576, in part by ONR N0004-02-0102, and in part by Lockheed Martin Corporation 1-5-36137.

References

1. Buttazzo G, Lipari G, Abeni L (1998) Elastic task model for adaptive rate control. In: Proceedings of the IEEE real-time systems symposium, Dec 1998, pp 286–295
2. Liu CL, Layland J (1973) Scheduling algorithms for multi-programming in a hard real-time environment. J ACM 20:46–61
3. Han C, Lin K (1992) Scheduling distance-constrained real-time tasks. In: Proceedings of the IEEE real-time systems symposium, Dec 1992, pp 300–308
4. Sprunt B, Sha L, Lehoczky J (1989) Aperiodic task scheduling for hard-real-time systems. Real-time Syst J 1:27–60
5. Liu JWS, Shih WK, Lin KJ, Bettati R, Chung JY (1994) Imprecise computations. Proc IEEE 82:83–94
6. Tai TS, Deng Z, Shankarand M, Storch M, Sun J, Wu LC, Liu JWS (1995) Probabilistic performance guarantee for real-time tasks with varying computation times. In: Proceedings of the IEEE real-time technology and application symposium, 1995, pp 164–173
7. Shih CS, Liu JWS (2002) State-dependent deadline scheduling. In: Proceedings of the IEEE real-time systems symposium, Austin, TX, USA, 2002, pp 3–14
8. Stankovic JA, Lu C, Son SH, Tao G (1999) The case for feedback control real-time scheduling. In: Proceedings of the 11th Euromicro conference on real-time systems, 1999, pp 11–20
9. Caccamo M, Buttazzo G, Sha L (2000) Elastic feedback control. In: Proceedings of the 12th Euromicro conference on real-time systems, 2000, pp 121–128
10. Caccamo M, Buttazzo G, Sha L (2000) Capacity sharing for overrun control. In: Proceedings of the IEEE real-time systems symposium, Orlando, FL, USA, 2000, pp 295–304
11. Abeni L, Butazzo G (1999) QoS guarantee using probabilistic deadlines. In: Proceeding of the 11th Euromicro conference on real-time systems (ECRTS99), York, England, 1999, pp 242–251
12. Martin RC (2002) Agile software development, principles, patterns, and practices. Prentice Hall, Upper Saddle River, NJ
13. Lutz RR, Mikulski IC (2003) Resolving requirements discovery in testing and operations. In: Proceedings of the 11th IEEE international requirements engineering conference (RE'03), 2003, Monterey Bay, CA, USA, , pp 33–41
14. Dubois E, Pohl K (2003) RE 02: A major step toward a mature requirements engineering community. IEEE Software 20:14–15
15. Perkins CE, Wang KY (1999) Optimized smooth handoffs in Mobile IP. In: Proceedings of the IEEE international symposium on computers and communications, July 1999, pp 340–346
16. Lee KD, Kim S (1999) Traffic model and analysis for handoff performance in microcellular networks with directed retry. In: Proceedings of the IEEE TENCON'99, vol 1, 1999, pp 39–42
17. Ghazalie TM, Baker TP (1995) Aperiodic servers in a deadline scheduling environment. Real-Time Syst J 9:31–67