Design and Implementation of an Adaptive Dispatching Controller for Elevator Systems During Uppeak Traffic

David L. Pepyne and Christos G. Cassandras, *Fellow, IEEE*

***Abstract*— We design a dispatching controller for elevator systems during uppeak passenger traffic with the ability to adapt to changing operating conditions. The design of this controller is motivated by our previous paper where we proved that for a queuing model of the uppeak dispatching problem a *threshold* policy is optimal (in the sense of minimizing the average passenger waiting time) with threshold parameters that depend on the passenger arrival rate. The controller, which we call the concurrent estimation dispatching algorithm (CEDA), uses concurrent estimation techniques for discrete-event systems. The CEDA allows us to observe the elevator system while it operates under some arbitrary thresholds, and concurrently estimate, in an unobtrusive way, what the waiting time would have been had the system operated under a set of different thresholds. These concurrently estimated waiting times are used to adapt the operating thresholds to match the elevator service rate to a changing passenger arrival rate. Implementation issues relating to the limited state information provided by actual elevator systems are resolved in a way that maintains modest computational requirements and avoids the need for supplemental sensors beyond those already typically provided. Numerical performance results show the advantages of the CEDA over currently used dispatching algorithms for uppeak.**

***Index Terms*— Adaptive control, bulk-service queueing networks, concurrent estimation, discrete-event dynamic systems, optimization problems, perturbation analysis, queueing theory, thresholds, transportation systems.**

I. Introduction

THIS paper is a companion to our previous paper [15], where we proved that the structure of the optimal dispatching policy for elevator systems in uppeak traffic is a threshold-based policy with threshold parameters that change as a function of the passenger arrival rate. In this paper, we demonstrate how concurrent estimation techniques can be used to implement such a threshold dispatching policy in a realistic elevator system.

In our previous paper [15], we give a detailed introduction to the difficult problem of elevator dispatching (see also [17]). Briefly, passenger traffic in an office building can be described as combinations of the three basic components shown in Fig. 1 [16]. *Incoming* traffic represents passengers arriving only at the main lobby and traveling to destination floors up in the building; *outgoing* traffic represents passengers traveling down from the upper floors to the main lobby; and *interfloor* traffic is due to passengers moving randomly between floors other than the first. For example, the lunchtime traffic mode, which occurs in the middle of the day, can be described as a combination of outgoing traffic caused by workers going to lunch, and incoming traffic caused by workers returning from lunch. Similarly, the downpeak traffic mode, which occurs at the end of the day, consists almost exclusively of outgoing traffic caused by the workers as they leave the building. The uppeak traffic mode, which is the focus of this paper, occurs first thing in the morning, and is dominated by incoming traffic caused by the workers as they arrive and take the elevators up to their offices.

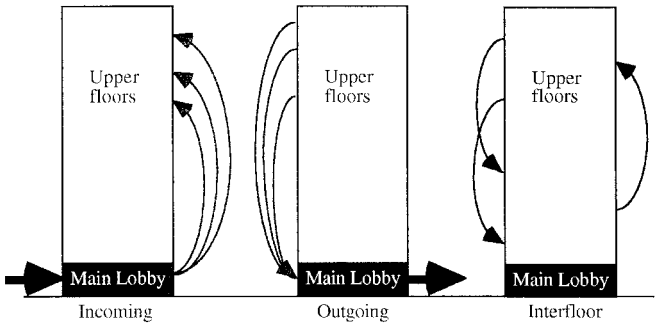

Fig. 1. Basic passenger traffic components in an office building.

During uppeak, passengers arrive only at the first floor, there is no interfloor or outgoing traffic. The elevators take the passengers up to their destinations, and then make an express run back down to the first floor to serve more passengers. In practice, most elevator systems employ what is called the *next car strategy* during the uppeak mode [2]. When using the next car strategy, only one car is loaded at a time. This car is referred to as the *designated next car* to be dispatched. All other cars which may be waiting at the first floor keep their doors closed or otherwise discourage passengers from entering (by dimming the lights inside the car, failing to signal the travel direction, and so forth). During the uppeak traffic mode, the dispatching question is reduced from deciding when and where to dispatch each elevator car, to one of simply deciding *when* to dispatch the designated next car. In many elevator systems, the number of passengers in a car can be estimated

Manuscript received September 30, 1996; revised August 14, 1997.

D. L. Pepyne is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003 USA.

C. G. Cassandras is with the Department of Manufacturing Engineering, Boston University, Boston, MA 02215 USA.

Publisher Item Identifier S 1063-6536(98)06134-X.

by weight sensors or by sensors which count passengers as they enter and exit the cars. Given that the number of passengers in a car can be determined, the simplest uppeak dispatching policy is a *threshold policy* with a threshold of one: dispatch the designated next car when the number of passengers inside is nonzero (letting θ denote the threshold parameter, we will refer to this as the $\theta = 1$ policy). Another uppeak dispatching policy, termed *half-capacity plus time-out,* dispatches an elevator when half its capacity is reached or when a timer, started when the first passenger enters the elevator, expires (usually a 20-s timer is used). For a more detailed discussion on these see [15]. The basic problem with both of these approaches is their *open-loop* nature. During uppeak, which lasts for about an hour each morning in a typical office building, the passenger arrival rate can double from one five-minute interval to the next. It is, therefore, difficult for open-loop dispatching policies to perform well over the entire hour-long uppeak period.

In [15] we developed a Markov decision problem (MDP) formulation of an elevator system in uppeak traffic, and showed that the solution that minimizes the average passenger waiting time at the lobby over an infinite horizon is a dynamic threshold policy where the threshold parameter is a function of the passenger arrival rate. Motivated by that work, we design in this paper an on-line, adaptive threshold-based dispatcher which adapts its threshold to the changing passenger arrival rate that occurs during the uppeak period. To evaluate the approach, we compare its performance to the two open-loop policies described above. To design the dispatching algorithm, we use *concurrent estimation* techniques (see [5]). Concurrent estimation allows us to observe the elevator system while it is operating under some threshold and estimate what the passenger waiting time would have been had we operated the system under all other admissible threshold values (without actually having to explicitly try them out). Using these estimated passenger waiting times, we adapt the threshold to the passenger arrival rate. Starting with arbitrary threshold settings, we have found that our strategy rapidly settles down to a set of thresholds that perform much better than the open-loop policies described previously.

The remainder of this paper is organized as follows. In Section II we describe the uppeak elevator dispatching problem and review previous results on optimal dispatching control that motivate the controller design presented in this paper. Sections III and IV describe the concurrent estimation methodology we adopt for designing an adaptive dispatching controller. Section V deals with the implementation issues involved in applying our dispatching control algorithm to an actual elevator system. Section VI gives performance results to demonstrate the efficacy of our algorithm compared to the state of the art. Finally, we conclude in Section VII.

II. Problem Formulation

For a typical office building, the uppeak traffic mode occurs in the morning when the building's occupants arrive for work [2]. The uppeak period lasts for about an hour, during which time virtually the entire population of the building will arrive at the main lobby and request elevator service. A typical variation in the arrival rate of incoming passengers during uppeak in such buildings is shown in Fig. 2. The start of the workday (say 9 A.M.) might be somewhere between 30–40 min into the uppeak period. The passenger arrival rate is low at the beginning of the period (8:20 A.M.) as the "early" workers are arriving. The arrival rate is very high just before 9:00 A.M. as most workers will try to arrive "just in time." At the end of the period (9:20 A.M.), the arrival rate tails off as the "late" workers finally arrive. We will return to the issue of modeling the passenger arrival process during the uppeak period in Sections IV-B and V-B.

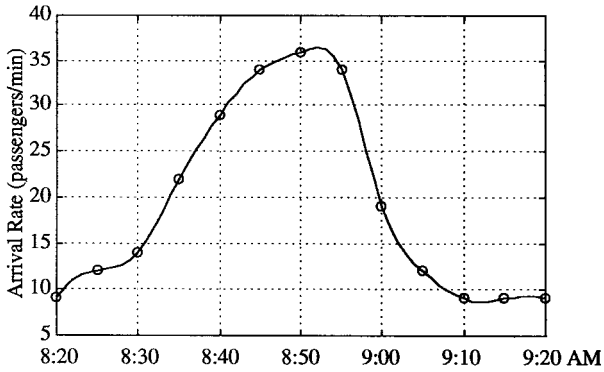

Fig. 2. Typical arrival rate of incoming passengers during the uppeak period.

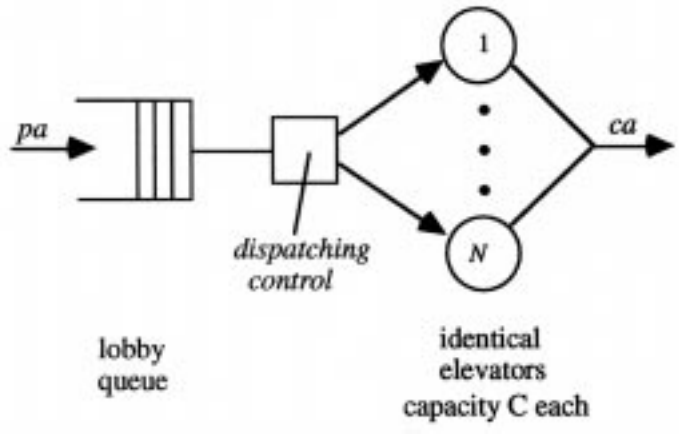

Fig. 3. Queueing model of an elevator system during uppeak.

Viewed as a DES, an elevator system during uppeak may be represented by the queueing model in Fig. 3. Incoming passengers arrive to a lobby queue where some dispatching control policy is used to decide how the elevators (also referred to as "cars") will be loaded (e.g., through the *next car strategy* described in the introduction). The passengers are served by N identical cars, each with a finite capacity of C passengers. The state space of this DES is given by $X = \{(y, z)\colon\ y = 0, 1, \cdots, z = 0, 1, \cdots, N\}$ where y is the length of the lobby queue and z is the number of cars waiting at the main lobby. The dynamics are driven by passenger arrival (*pa*) events, which occur when a passenger arrives at the lobby queue, and by car arrival (*ca*) events, which

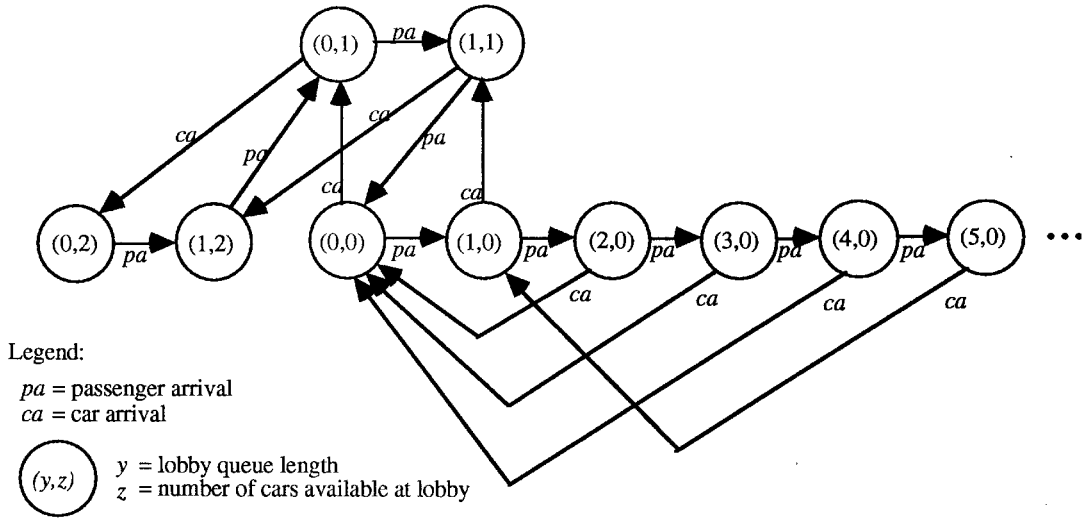

Fig. 4. State transition diagram for two-car elevator system operating under a threshold policy during uppeak.

occur when an elevator returns to the main lobby after serving passengers (for the detailed state transition structure see [15]). Control actions are taken only when an event occurs and they define a set $U_z = \{0, 1, \cdots, z\}$, where $u = 0$ implies that all available cars are held at the lobby, and $u = n$ implies that n cars ($n \leq z$) are allowed to be loaded and dispatched simultaneously.

A. Optimal Dispatching Control

For the system in Fig. 3 with Poisson passenger arrivals and exponentially distributed elevator service time (the time for an elevator to deliver a load of passengers and return to the lobby), the main result of [15] was to show that the optimal dispatching policy minimizing the average passenger waiting time is a *threshold-based policy*. Specifically, let the control action $u^*(y, z) \in \{0, 1, \cdots, z\}$ be the optimal number of cars to be dispatched when the lobby queue length is y and the number of available cars is z. Associated with each control action are z threshold parameters which we will denote by $\theta^*_{z,i}(\lambda, \mu)$, where $i = 1, \cdots, z$. These thresholds are functions of the passenger arrival rate λ and the elevator service rate μ, and are such that $(i-1)C < \theta^*_{z,i}(\lambda, \mu) \leq iC$, where C is the elevator capacity. Furthermore, the optimal number of cars to dispatch is given by (see [15])

$$u^*(y, z) = \begin{cases} z & y \geq \theta^*_{z,z} \\ z-1 & \theta^*_{z,z-1} \leq y < \theta^*_{z,z} \\ \vdots & \\ 0 & y < \theta^*_{z,1}. \end{cases} \quad (1)$$

In addition, it was also shown in [15] that the following relationship holds:

$$\theta^*_{z,i} = \theta^*_{z-1,i-1} + C, \quad z = 2, \cdots, N \; i = 2, \cdots, z. \quad (2)$$

Notice, however, that since $\theta^*_{z,1} < C$, the situation where more than one car is available and the queue length exceeds C is never encountered. Thus, in practice, only the N thresholds $\theta^*_{z,1}$, $z = 1, \cdots, N$ need to be determined. This fact has an important practical implication: instead of needing to know the lobby queue length y (which is difficult to measure), we only need to know the number of passengers inside the designated next car (which is much easier to measure).

1) Example: To illustrate the above threshold-based dispatching policy, consider the case of $N = 2$ cars with a capacity of $C = 4$ passengers each, and a threshold policy with $\theta^*_{1,1} = \theta^*_{2,1} = 2$. Then

$$u^*(y, 1) = \begin{cases} 1 & y \geq 2 \\ 0 & y < 2, \end{cases} \quad u^*(y, 2) = \begin{cases} 1 & y \geq 2 \\ 0 & y < 2. \end{cases}$$

Fig. 4 shows a state transition diagram for this system operating under the above dispatching policy.

The analysis in [15] did not provide a closed-form expression for the optimal threshold values. To implement the threshold policy, therefore, two problems remain: **P1**: For given λ and μ, we need to determine the threshold values; and **P2**: As λ changes throughout the uppeak period, a mechanism is needed for adapting the thresholds. Theoretically, these problems can be dealt with by solving a Markov chain corresponding to the system of Fig. 4 and evaluating the average passenger waiting time as a function of different thresholds to determine the values yielding the minimum average wait for a range of passenger arrival rates. This is clearly not a simple computational task, even if stationary solutions are desired; in our case, we will be interested in determining optimal thresholds over 5-min estimation intervals, so that the transient behavior of the Markov chain needs to be analyzed, an even more difficult task. In addition, the results provided by such analysis may still be inadequate because, in an actual elevator system, some of the modeling assumptions may not hold (e.g., the elevator service times may not be exponentially distributed).

We see the main value of the results in [15] as identifying the *structure* of the optimal policy. Motivated by the simple threshold-based structure, our objectives in the remainder of the paper are to 1) design an on-line approach for estimating optimal threshold values without having complete knowledge

of the state of the system and without being able to detect all events and 2) compare the performance of our dispatching controller to currently used policies; in particular, the $\theta = 1$ and the *half-capacity plus time-out* policies mentioned in the introduction.

III. Concurrent Estimation

In this section, we present a design approach based on "concurrent estimation" (see [5]) which allows us to estimate *on-line* the average passenger waiting time for any admissible dispatching threshold. The main idea is to observe the evolution of a sample path of an actual elevator system as it operates under some preselected thresholds. As the sample path evolves, observed data (e.g., event occurrences and their corresponding occurrence times) are processed to concurrently construct the set of sample paths that would have resulted if the system had operated under a set of different (hypothetical) dispatching thresholds. Using these "concurrently constructed" hypothetical sample paths, it is possible to "concurrently estimate" the corresponding average passenger waiting times. A simple scheme is then used to adjust the threshold used in the actual elevator system to the one that gives the best estimated waiting time. This cycle of concurrent sample path construction, waiting time estimation, and threshold adjustment is done continuously, aiming not only to identify the best thresholds for the present operating conditions, but also to adapt the thresholds to track changes in the operating conditions. As will be seen, this process of constructing sample paths does not interfere in any way with the normal operation of the actual elevator system.

To explain the principles of concurrent estimation, some notation, definitions, and background material will be presented first. We start by describing what is known as the "sample path constructability problem." Then, after reviewing the concept of a stochastic timed state automaton as a modeling framework for general DES, we describe the general procedure for constructing sample paths of DES. Concurrent estimation is then described as a solution to the sample path constructability problem. With this background, we present the general concurrent estimation scheme, and explain how the scheme is specialized to the uppeak dispatching problem.

A. Sample Path Constructability

Consider a DES and a finite discrete parameter set $\Theta = \{\theta_1, \cdots, \theta_m\}$, where each parameter $\theta_j \in \theta$, $j = 1, \cdots, m$ is in general vector valued. Suppose the sample path generated by the DES is a function of the parameter θ_j (e.g., the dispatching threshold), and designate the sample path generated under parameter θ_j by the sequence of pairs $\{c_k^j, t_k^j\}$, where $k = 1, 2, \cdots$ is an event-counting index, e_k is the kth event (e.g., a *pa* or *ca* event), and t_k is the occurrence time of the kth event (equivalently, a sample path can be defined by $\{x_k^j, t_k^j\}$, $k = 1, 2, \cdots$, where x_k is the state entered when the kth event occurs at time t_k). Now, assume that the DES is operating under θ_1 and that all events and event times e_k^1, t_k^1 for $k = 1, 2, \cdots$, are directly observable. The problem, then, is to use the observations of the sample path $\{c_k^1, t_k^1\}$ to construct the sample paths $\{c_k^j, t_k^j\}$, $k = 1, 2, \cdots$, for any θ_j, $j = 2, \cdots, m$, as shown in Fig. 5. This problem is referred to as the *sample path constructability problem* [7]. To obtain an on-line algorithm, we will perform this construction in real time while the observed sample path evolves. Moreover, we will perform the construction of all $m - 1$ sample paths for $j = 2, \cdots, m$ concurrently.

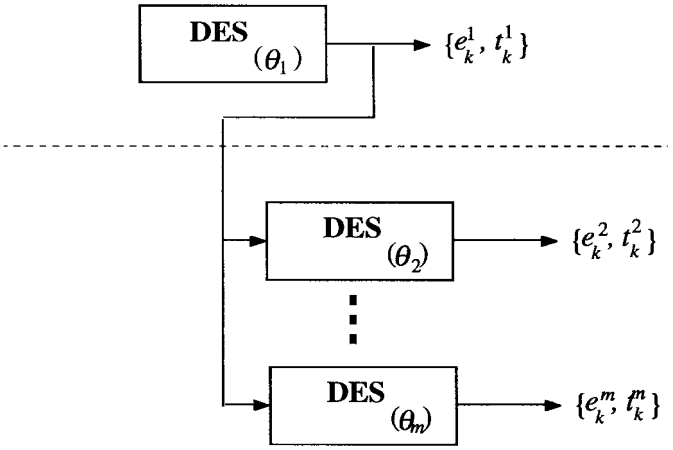

Fig. 5. The sample path constructability problem.

Note that any sample performance metric $L(\theta_j)$ (e.g., the average waiting time for some dispatching threshold θ_j) is obtained as a function of the corresponding sample path $\{c_k^j, t_k^j\}$, $k = 1, 2, \cdots$. The importance of the sample path constructability problem, therefore, becomes clear when placed in the context of the following basic optimization problem:

$$\text{find } \theta \in \Theta \text{ to minimize } J(\theta) = E[L(\theta)] \tag{3}$$

where we are careful to distinguish between $L(\theta)$, the performance obtained over a *specific sample path* of the system and $J(\theta)$, the *expectation over all possible sample paths.* The solution to the sample path constructability problem, if it exists, enables us to learn about the behavior of a DES under all possible parameter values in Θ from a single "trial," i.e., a single sample path obtained under one parameter value. Most importantly, if performance estimates $L(\theta_1), \cdots, L(\Theta_m)$ are all available at the conclusion of one trial, we can immediately select a candidate optimal parameter $\theta^* = \arg\min_{\theta \in \theta} L(\theta)$. This is potentially the true optimal choice, depending of course on the statistical accuracy of the estimates $L(\theta_1), \cdots, L(\theta_m)$ of $J(\theta_1), \cdots, J(\theta_m)$. In practice, the statistical properties of the DES and the size of the parameter set Θ may make it necessary to use an iterative process to ultimately identify the true optimal parameter value. If Θ is very large, for example, parameter space partitioning or random-search types of algorithms may need to be used (e.g., see [1], [9], and [18]). However, when the parameter set is small, it is possible to obtain all estimates $L(\theta_1), \cdots, L(\theta_m)$ concurrently, which gives rise to the term "concurrent estimation" associated with solving the sample path constructability problem and the basic optimization problem (3). In this case, much simpler and faster schemes may be used to identify the optimal parameter (e.g., see [4]). In practice, however, optimality is usually traded for

speed, and *ad hoc* schemes are often used to quickly identify a parameter which gives satisfactory, rather than optimal, performance.

B. Stochastic Timed State Automata

To explain the principles of concurrent sample path construction and estimation, it is useful to review how a single sample path is formally constructed for any DES. To do this we will make use of the *stochastic timed-state automaton* (e.g., see [3]), which provides a general framework for modeling DES.

We begin by reviewing the concept of a *state automaton*. A *state automaton* is defined by (E, X, Γ, f, x_0), where E is a countable *event set*, X is a countable *state space*, $\Gamma(x)$ is a set of *feasible* or *enabled* events, f is a *state transition function*, and $x_0 \in X$ is an *initial state*. The feasible event set $\Gamma(x) \subseteq E$ and is defined for all states $x \in X$. The feasible event set reflects the fact that it is not always physically possible for some events to occur. The state transition function $f(x, i)$, $f: X \times E \rightarrow X$, is defined only for the feasible events $i \in \Gamma(x)$; it is undefined for $i \notin \Gamma(x)$. It is also possible to replace the state transition function f by a state transition probability function $p(x'; x, i)$ representing the probability that the next state is x' given that the current state is x when event i occurs.

A *timed* state automaton $(E, X, \Gamma, f, x_0, V)$ is obtained when the model above is endowed with a *clock structure*, $V = \{\mathbf{v}_i, i \in E\}$. This clock structure associates with every event $i \in E$ a real-valued clock sequence $\mathbf{v}_i = \{v_i(1), v_i(2), \cdots\}$, where, $v_i(j)$ is the jth *lifetime* of event i. The jth lifetime is the amount of time between the instant when this event is enabled for the jth time and its next occurrence.

Finally, in a *stochastic* timed state automaton $(E, X, \Gamma, f, x_0, G)$, the clock structure V is replaced by a set of probability distribution functions $G = \{G_i, i \in E\}$. In this case, the clock sequences $\mathbf{v}_i = \{v_i(1), v_i(2), \cdots\}$ are random processes. For simplicity, we usually assume that the lifetimes $v_i(1), v_i(2), \cdots$ are i.i.d. random variables with distribution G_i. Thus, to generate a sample path of the system, whenever a lifetime for event i is needed we obtain a sample from G_i. The state sequence generated through this mechanism is a stochastic process known as a *generalized semi-Markov process* (GSMP) (see also [8] and [10]).

It will be helpful later on if we summarize the exact steps involved in generating a sample path of a stochastic timed state automaton. In addition to the state x of the underlying automaton, let us define two more state variables as follows. First, let us associate with each feasible event a state variable τ_i to denote the next occurrence time of event i. The usefulness of this variable, is that, given the occurrence times for each feasible event, the next event to occur, called the *triggering event*, is given by

$$e' = \arg\min_{i \in \Gamma(x)} \{\tau_i\}$$

and the time at which the event occurs is immediately given by $t' = \tau_{e'}$. The other state variable which we will find useful is the *score* of event i which we will denote by $s_{i,n}$: after n total events have been observed in a sample path, the score $s_{i,n}$ is the number of events of type $i \in E$ that have occurred. We can now construct a sample path of any DES modeled as a stochastic timed automaton as follows.

1) Sample Path Construction (SPC) Procedure:

Step 1: Given the current state x_k and next event times $\tau_{i,k}$ for all feasible events $i \in \Gamma(x_k)$, determine the next event time

$$t_{k+1} = \min_{i \in \Gamma(x_k)} \{\tau_{i,k}\}. \tag{4}$$

Step 2: Determine the triggering event

$$e_{k+1} = \arg\min_{i \in \Gamma(x_k)} \{\tau_{i,k}\}. \tag{5}$$

Step 3: Determine the next state

$$x_{k+1} = f(x_k, e_{k+1}). \tag{6}$$

Step 4: Update the next event times for all events $i \in \Gamma(x_{k+1})$

$$\tau_{i,k+1} = \begin{cases} \tau_{i,k} & \text{if } i \neq e_{k+1} \text{ and } i \in \Gamma(x_k) \\ t_{k+1} + v_i(s_{i,k}+1) & \text{if } i = e_{k+1} \text{ or } i \notin \Gamma(x_k). \end{cases} \tag{7}$$

Step 5: Update the event scores

$$s_{i,k+1} = \begin{cases} s_{i,k} + 1 & \text{if } i = e_{k+1} \\ s_{i,k} & \text{otherwise.} \end{cases} \tag{8}$$

Step 6: Increment k and continue from Step 1.

For some given state x_0, this procedure is initialized by setting $s_{i,0} = 0$ for all $i \in E$ and $\tau_{i,0} = v_i(1)$ for all $i \in \Gamma(x_0)$. For those familiar with DES simulation, the SPC procedure above is nothing more than the standard event scheduling simulation scheme. Some readers may have noticed that the formalism we have used here to define the GSMP is slightly different than the usual one (e.g., see [3], [8], and [10]), but it will prove more useful for our purposes as we will subsequently explain.

2) Example: Let us illustrate the SPC procedure for the two car elevator system operating under a threshold policy $\theta^*_{1,1} = \theta^*_{2,1} = 2$ whose state transition diagram was introduced in Fig. 4. In this case, the event set is $E = \{pa, ca\}$, and the state set is $X = \{(y, z): y = 0, 1, \cdots, z = 0, 1, 2\}$ where y is the length of the lobby queue and z is the number of cars waiting at the main lobby. Feasible events at each state are shown in Fig. 4 by the corresponding outgoing arrows. Note that we do not differentiate between distinct car arrivals because of the assumption that the cars are identical. Suppose that we do not have any information regarding the lifetime distributions of pa and ca events. Thus, we cannot generate the event lifetimes $v_i(1), v_i(2), \cdots$ needed in (7). If we are observing an actual elevator system, however, we can simply observe events e_{k+1} as they occur and record their occurrence times t_{k+1}. Since we know the state transition function for the system, we can then use (6) to update the state, and through (8) we can update the event scores. Again, in the absence of event lifetimes $v_i(1), v_i(2), \cdots$ we cannot explicitly update the next event times, but we can evaluate them through (7) after the events occur and their lifetimes have been determined.

At the beginning of our sample path construction, let us assume that the initial state is (0, 2). Referring to Fig. 4, the only feasible event in this state is pa and we have $\tau_{pa,0} = v_{pa}(1)$. Let us now see how the SPC procedure applies to the first few events in a typical sample path:

- *First Event* $(k = 1)$*:* Suppose this is observed at time t_1. Since only a pa event was feasible, this is necessarily a pa event, and, by (4), it is understood that $t_1 = v_{pa}(1)$. Applying (6) as specified through the state transition diagram in Fig. 4, the next state is (1, 2) and a second pa event becomes feasible. By (7) this next pa event will occur at time $\tau_{pa,1} = t_1 + v_{pa}(2)$, and by (8), we update the pa score from one to two.
- *Second Event* $(k = 2)$*:* Suppose this second pa event is observed at time t_2. Repeating the process, we must now have $t_2 = t_1 + v_{pa}(2)$, the new state is (0, 1), a third pa event becomes feasible, and since we have reached the dispatching threshold and a car was dispatched, a ca event also becomes feasible. By (7) the occurrence time for the third pa event will be $\tau_{pa,2} = t_2 + v_{pa}(3)$ and the occurrence time for the ca event will be $\tau_{ca,2} = t_2 + v_{ca}(1)$.
- *Third Event* $(k = 3)$*:* Suppose this is observed at time t_3. In this case, the event may have been either a pa or ca, since both were feasible at state (0, 1). Let us assume that the ca event occurred first; this implies, from (4), that $t_3 = t_2 + v_{ca}(1) < t_2 + v_{pa}(3)$. The ca event causes a transition into state (0, 2) where the feasible event set is reduced to $\{pa\}$, which from (7) will occur at time $\tau_{pa,3} = \tau_{pa,2}$.

The sample path constructed thus far is $\{(pa, t_1), (pa, t_2), (ca, t_3), \cdots\}$. As the sample path evolves, any performance metric of interest can be estimated as a function of $\{e_k, t_k\}$, $k = 1, 2, \cdots$. Sample path construction is terminated when the performance estimates have been obtained to some desired degree of statistical accuracy.

The example above serves to illustrate how to formally construct a sample path of a DES when event lifetimes are not available, but rather directly observed. This will facilitate the understanding of the concurrent estimation technique in the next section.

C. Concurrent Sample Path Construction

A considerable amount of work is currently being directed toward solving the sample path constructability problem. For DES in which all event processes are Markovian (memoryless), the standard clock method [19] and augmented system analysis [6] provide two very efficient solutions. Most recently, Cassandras and Panayiotou [5] have proposed a general-purpose approach for DES; while their approach is not as efficient as the above two, it is applicable to DES with arbitrary event lifetime distributions. We will review this approach next and then specialize it to the uppeak dispatching problem.

The starting point is to consider a given DES operating under a specific parameter value θ. Assume that all events and their occurrence times are observable. Now recall that associated with every event $i \in E$ is a real-valued clock sequence $v_i = \{v_i(1), v_i(2), \cdots\}$, where $v_i(j)$ is the jth lifetime of event i. Then, define

$$V_i(n) = \{v_i(1), \cdots, v_i(s_{i,n})\}, \qquad i \in E \tag{9}$$

to be the sequence of observed lifetimes of event i after n total events have been observed.

The objective is to use the observed event lifetime sequences (9) and the SPC procedure to construct the hypothetical sample path that would result if the same DES were operating under some different parameter value $\hat{\theta} \neq \theta$. Let $k \leq n$ be the total number of events in this hypothetical sample path that we are constructing. Denote the corresponding event lifetime sequences in the constructed sample path by

$$\hat{V}_i(k) = \{v_i(1), \cdots, v_i(\hat{s}_{i,k})\}, \qquad i \in E \tag{10}$$

where $\hat{s}_{i,k}$ is the corresponding score of event i in the constructed sample path. Next define

$$\tilde{V}_i(n, k) = \{v_i(\hat{s}_{i,k} + 1), \cdots, v_i(s_{i,n})\}, \qquad i \in E \tag{11}$$

to be sequences of those observed event lifetimes which have not yet been used in the construction of the hypothetical sample path; that is, these sequences contain all event lifetimes which are in $V_i(n)$ but not in $\hat{V}_i(k)$. We associate with $\tilde{V}_i(n, k)$ a set

$$A(n, k) = \{i\colon i \in E, s_{i,n} > \hat{s}_{i,k}\} \tag{12}$$

consisting of the subset of events i for which $\tilde{V}_i(n, k)$ contains at least one element, i.e., there is at least one observed lifetime available that has not yet been used in the constructed sample path. This set $A(n, k)$ is referred to as the *available event set* because it contains the set of events whose lifetimes are available to be used to construct the hypothetical sample path after n observed events. Last, we define one more set as follows. Let $\hat{x}_k$ denote the state after k events on the constructed sample path, and let $\hat{e}_k$ be the triggering event at the $(k-1)$th state visited on this sample path. Then, define

$$M(n, k) = \Gamma(\hat{x}_k) - (\Gamma(\hat{x}_{k-1}) - \{\hat{e}_k\}). \tag{13}$$

That is, $M(n, k)$ contains all those events that are feasible in state $\hat{x}_k$ that were not feasible in state $\hat{x}_{k-1}$. Note that the triggering event $\hat{e}_k$ is also in this set if it happens that $\hat{e}_k \in \Gamma(\hat{x}_k)$. Intuitively, $M(n, k)$ consists of all those events whose occurrence times are missing from the perspective of the constructed sample path when it enters the state $\hat{x}_k$: the occurrence times for events in $\Gamma(\hat{x}_{k-1})$ are already known and remain available to be used in the sample path construction if they are still feasible; those events not feasible in $\hat{x}_{k-1}$ which have become feasible in the state $\hat{x}_k$, on the other hand, are *missing* as far as their occurrence times are concerned. We shall refer to $M(n, k)$ as the *missing event set* after n observed events.

It is clear from Steps 3.2) and 3.3) of the SPC procedure in the last section, that in order to continue sample path construction from state $\hat{x}_k$ in the hypothetical sample path, we must have lifetimes (equivalently, occurrence times) for all events in the feasible event set $\Gamma(\hat{x})$. A key result from [5] is a necessary and sufficient condition for sample path

construction: when event e_{n+1} is observed in the actual DES, construction of the hypothetical sample path can continue if and only if

$$M(n+1, k) \subseteq A(n+1, k) \tag{14}$$

otherwise, construction of the hypothetical sample path is "suspended" in state $\hat{x}_k$ until some future observed event causes (14) to be satisfied. Thus, with every observed event, condition (14) is checked: if it is satisfied, the SPC procedure is invoked to update the state of the constructed sample path; otherwise, construction is suspended at $\hat{x}_k$ until some future observed event causes (14) to be satisfied. Note that with every observed event the set $A(n, k)$ is updated and possibly enlarged, while the set $M(n, k)$ remains fixed, since it depends only on $\hat{x}_k$.

An explicit algorithm for constructing concurrent sample paths under parameter values $\hat{\theta} \neq \theta$ based on observed data from a sample path under θ is given in [5], where a detailed discussion of the conditions for which the approach is applicable may also be found. Briefly, the conditions for meaningful concurrent sample path construction are the following: 1) we assume that θ does not affect the event lifetime distributions in the DES, but only the state transition structure. If that is not the case, the algorithm described next requires certain modifications that we will not dwell on here; 2) changes in θ should not introduce new types of events into the event set E; and 3) the state transition structure of the observed system is assumed irreducible. More generally, there should be no state transition causing an event to become permanently disabled (since this implies that (14) may never be satisfied).

The algorithm in [5] is referred to as the time warping algorithm (TWA). We reproduce it here with minor changes to suit our purposes. In the algorithm, the operators $+$ and $-$ are applied to both scalars and sequences. When applied to scalars, they denote the usual addition and subtraction operations. When applied to sequences, $+$ indicates the addition of an element to the end of the sequence and $-$ indicates removal of the first element of the sequence.

Time Warping Algorithm (TWA):

1) INITIALIZE:
   Given x_0, $\hat{x}_0$
   Set $n = k = 0$, $t_n = \hat{t}_n = 0$; for all $i \in E$ set $s_{i,n} = \hat{s}_{i,0} = 0$; for all $i \in \Gamma(x_o)$ set $\tau_{i,0} = \nu_i(1)$
   Set $M(0, 0) = \Gamma(\hat{x}_0)$, $A(0, 0) = \emptyset$
2) WHEN EVENT e_{n+1} IS OBSERVED:
   2.1) Add the observed lifetime of e_{n+1} to $\tilde{V}_i(n+1, k)$

$$\tilde{V}_i(n+1, k) = \begin{cases} \tilde{V}_i(n, k) + v_i(s_{i,n}+1) & \text{if } i = e_{n+1} \\ \tilde{V}_i(n, k) & \text{otherwise.} \end{cases}$$

   2.2) Update available event set $A(n+1, k) = A(n, k) \cup e_{n+1}$.
   2.3) Update missing event set $M(n+1, k) = M(n, k)$.
   2.4) If $M(n+1, k) \subseteq A(n+1, k)$, then go to 3). Else, set $n \leftarrow n+1$ and repeat from 2).
3) TIME WARPING OPERATION:
   3.1) Obtain $\nu_i(\hat{s}_{i,k}+1)$ from $\tilde{V}_i(n+1, k)$ for all missing events $i \in M(n+1, k)$ and use the SPC procedure to determine $\hat{t}_{k+1}$, $\hat{e}_{k+1}$, $\hat{x}_{k+1}$, and $\hat{\tau}_{i,k+1}$, $\hat{s}_{i,k+1}$ for all i.
   3.2) Discard all used event lifetimes

$$\tilde{V}_i(n+1, k+1) = \tilde{V}_i(n+1, k) - v_i(\hat{s}_{i,k}+1)$$

   for all $i \in M(n+1, k)$.
   3.3) Update available event set

$$A(n+1, k+1) = A(n+1, k) - \{i: i \in M(n+1, k), \hat{s}_{i,k+1} = s_{i,n+1}\}.$$

   3.4) Update missing event set

$$M(n+1, k+1) = \Gamma(\hat{x}_{k+1}) - (\Gamma(\hat{x}_k) - \{\hat{e}_{k+1}\}).$$

   3.5) If $M(n+1, k+1) \subseteq A(n+1, k+1)$ then set $k \leftarrow k+1$ and go to 3.1). Else, set $k \leftarrow k+1$ and $n \leftarrow n+1$ and go to 2).

Note that the time warping operation [i.e., Steps 3.1)–3.5)] may result in several state updates in the constructed sample path in response to a single observed event in the actual DES: as long as the sample path construction condition (14) is satisfied, construction of the hypothetical sample path will proceed; otherwise, the constructed sample path's clock is stopped, while the observed system's clock keeps moving ahead. When the missing lifetimes become available and (14) is satisfied, the constructed sample path "instantaneously" processes as many events as possible causing its clock to "warp" forward. This process of moving backward in time to revisit suspended sample paths and then forward in time by one or more events lends itself to the term *time warping* [5].

It should be clear that by a simple modification to the TWA, any number of hypothetical sample paths can be concurrently constructed: instead of a single sample path under a parameter value $\hat{\theta}$, we can have many sample paths each indexed by $c = 1, 2, \cdots$ and each operating under a different parameter value θ_c. Computationally, the requirements of the TWA are minimal: adding/subtracting elements to sequences, simple arithmetic, and checking condition (14). It is usually the memory requirements that limit the number of concurrent sample paths that can be constructed, since the event lifetimes $\tilde{V}_i^c(n, k)$ need to be stored for each constructed sample path c. The advantage of simultaneously constructing many sample paths, however, lies in the fact that from the full state history generated for each constructed sample path, it is possible to evaluate and compare any desired performance measure of interest. In this way the TWA can be used to solve the sample path constructability problem and the optimization problem (3).

D. Specialization of the TWA to the Uppeak Elevator Dispatching Problem

Our objective is to observe an actual elevator system while it is operating under some arbitrary dispatching threshold during the uppeak traffic period and use the TWA to construct the hypothetical sample paths and estimate the passenger waiting times that would have resulted if the elevator system had been operating under the various thresholds $\theta_{z,1}$. There are N such thresholds, and each one may take any value in the set $\{1, \cdots, C\}$. Thus, a total of $K = C^N$ sample paths need to be constructed and the passenger waiting time under each needs to be estimated. Here we describe how we specialize the TWA to accomplish these objectives.

We begin by introducing some notation. First, let

$c =$ sample path index, with $c = 0$ denoting the observed sample path and $c = 1, \cdots, K$ denoting the cth constructed sample path.

For the uppeak dispatching problem we need to store observed lifetimes for two types of events, pa events and ca events (recall that we do not differentiate among cars since they are all identical). To do this we define two vectors.

V_{pa} A vector for storing observed pa event lifetimes.
V_{ca} A vector for storing observed ca event lifetimes.

Now we define the following.

$paIndex(0)$ An index into V_{pa} where the most recent pa lifetime observed in the actual elevator system is stored.
$caIndex(0)$ An index into V_{ca} where the most recent ca lifetime observed in the actual elevator system is stored.
$paIndex(c)$ The index into V_{pa} where the next pa lifetime will be obtained for constructing sample path $c = 1, \cdots, K$.
$caIndex(c)$ The index into V_{ca} where the next ca lifetime will be obtained for constructing sample path $c = 1, \cdots, K$.

Finally, we define two indicator functions for each constructed sample path.

1) $paFlag(c) = 1$ if a pa event is needed to continue sample path construction, zero otherwise.
2) $caFlag(c) = 1$ if a ca event is needed to continue sample path construction, zero otherwise.

With the definitions above, we can obtain any of the sets involved in the TWA. Clearly, the vectors V_{pa} and V_{ca} correspond to V_i, $i \in E$. It is also easy to see that the set of pa event lifetimes already used in constructing sample path c is given by

$$\hat{V}_{pa}^c = \{V_{pa}(1), \cdots, V_{pa}(paIndex(c) - 1)\} \tag{15}$$

and the set of available pa event lifetimes which have not yet been used is given by

$$\tilde{V}_{pa}^c = \{V_{pa}(paIndex(c)), \cdots, V_{pa}(paIndex(0))\}. \tag{16}$$

The sets $\hat{V}_{ca}^c$ and $\tilde{V}_{ca}^c$ are formed similarly. In addition, the available event set is given by

$$\begin{aligned}
A(c) &= \{pa, ca\} && \text{if } paIndex(c) \le paIndex(0) \\
& && \text{and } caIndex(c) \le caIndex(0) \\
A(c) &= \{pa\} && \text{if } paIndex(c) \le paIndex(0) \\
& && \text{and } caIndex(c) > caIndex(0) \\
A(c) &= \{ca\} && \text{if } paIndex(c) > paIndex(0) \\
& && \text{and } caIndex(c) \le caIndex(0) \\
A(c) &= \emptyset && \text{if } paIndex(c) > paIndex(0) \\
& && \text{and } caIndex(c) > caIndex(0).
\end{aligned} \tag{17}$$

and the missing event set is given by

$$\begin{aligned}
M(c) &= \{pa, ca\} && \text{if } paFlag(c) = 1 \text{ and } caFlag(c) = 1 \\
M(c) &= \{pa\} && \text{if } paFlag(c) = 1 \text{ and } caFlag(c) = 0 \\
M(c) &= \{ca\} && \text{if } paFlag(c) = 0 \text{ and } caFlag(c) = 1 \\
M(c) &= \emptyset && \text{if } paFlag(c) = 0 \text{ and } caFlag(c) = 0.
\end{aligned} \tag{18}$$

As for the subset test in (14): The construction of sample path c must be suspended when

$$paFlag(c) = 1 \quad \text{and} \quad paIndex(c) > paIndex(0)$$

or when

$$caFlag(c) = 1 \quad \text{and} \quad caIndex(c) > caIndex(0). \tag{19}$$

In the first case, we need a pa lifetime and no pa lifetime is available (either because none have yet been observed to have occurred in the actual system, or because we have used them all up). In the second case, we need a ca lifetime, and no ca lifetime is available (again, either because none have yet been observed to have occurred in the actual system, or because we have used them all up).

In terms of the above definitions, we now specialize the TWA to the uppeak elevator dispatching problem. In what follows, we use $Y(0)$ and $Z(0)$ to denote the state variables for the observed system, i.e., the queue length at the first floor and the number of cars available to dispatch, respectively. $Y(c)$ and $Z(c)$ are used to denote the same state variables in the cth constructed sample path.

IV. Concurrent Estimation Dispatching Algorithm (CEDA)

1) Initialize:

$$\begin{aligned}
Y(0) = Y(c) &= 0 && \text{for all } c = 1, \cdots, K \\
Z(0) = Z(c) &= N && \text{for all } c = 1, \cdots, K.
\end{aligned}$$

Start observing when the lobby queue is empty, and all cars are parked at the first floor lobby:

$$\begin{aligned} V_{pa} &= V_{ca} = \emptyset \\ paIndex(0) &= caIndex(0) = 0 \\ paIndex(c) &= caIndex(c) = 1 \\ paFlag(c) &= 1 \\ caFlag(c) &= 0. \end{aligned}$$

No events have been observed yet. Each concurrent estimator is initially suspended; each one awaiting a pa event. No ca event is needed, since all cars are assumed initially available at first floor lobby:

$$\begin{aligned} paClock &= 0 \\ caClock(1) = \cdots &= caClock(N) = 0. \end{aligned}$$

The above are clocks for recording event lifetimes.

$$T(0) = T(c) = 0.$$

All clocks (observed and constructed sample paths) are initially zero.

2) When an Event is Observed in the Actual Elevator System:

1) Get the event type (pa or ca) and the time $T(0)$
   1.1) Based on the time $T(0)$, update the operating threshold. The threshold is updated every 5 min as described in Section V.

2.1) If event is pa:
   2.1.1) $V_{pa}(paIndex(0)+1) = T(0) - paClock$. Record pa lifetime and insert into V_{pa}.
   2.1.2) $paIndex(0) = paIndex(0) + 1$. Increment pointer in V_{pa}.
   2.1.3) $paClock = T(0)$. Start clock for next pa event lifetime.

2.2) If event is ca:
   2.2.1) Determine car index: $theCar$
   2.2.2) $V_{ca}(caIndex(0) + 1) = T(0) - caClock(theCar)$. Record ca lifetime and insert into V_{ca}.
   2.2.3) $caIndex(0) = caIndex(0) + 1$. Increment pointer in V_{ca}.

3) Dispatching policy decides if car should be dispatched in observed elevator system. Here we assume the designated next car is dispatched when the number of passengers inside exceeds a certain operating threshold.
   3.1) Determine car index: $theCar$
   3.2) $caClock(theCar) = T(0)$. Start clock for next ca event lifetime.
   3.3) Estimate the waiting time for those passengers who were served by the car that was just dispatched. Update the average passenger waiting time. The waiting time for a passenger is the time between a passenger's arrival for elevator service and the time when the elevator that serves the passenger actually departs.

3) Time Warping Operation:

4) For $c = 1, \cdots, K$. For each concurrent constructed sample path.

5.1) If $paFlag(c) = 1$. A pa event is needed to continue construction.
   5.1.1) If $paIndex(c) > paIndex(0)$, go to 4). Suspend construction of sample path.
   5.1.2) $\tau^c_{pa} = T(c) + V_{pa}(paIndex(c))$. Else, determine next pa event time.
   5.1.3) $paIndex(c) = paIndex(c) + 1$. Increment pointer in V_{pa}.
   5.1.4) $paFlag(c) = 0$. Reset the pa flag.

5.2) If $caFlag(c) = 1$. A ca event is needed to continue construction.
   5.2.1) If $caIndex(c) > caIndex(0)$, go to 4). Suspend construction of sample path.
   5.2.2) $\tau^c_{ca} = T(c) + V_{ca}(caIndex(c))$. Else, determine next ca event time.
   5.2.3) $caIndex(c) = caIndex(c) + 1$. Increment pointer in V_{ca}.
   5.2.4) $caFlag(c) = 0$. Reset ca flag.

6) Use the SPC procedure to determine next event e^c, and next event time, τ^c_e and set

$$T(c) = \tau^c_e.$$

   6.1) If event is pa:
      6.1.1) $Y(c) = Y(c) + 1$. Increment lobby queue length.
      6.1.2) $paFlag(c) = 1$. Set pa flag to indicate a pa lifetime is needed to continue sample path construction.
   6.2) If event is ca:
      6.2.1) $Z(c) = Z(c) + 1$. Increment number of cars at lobby.

7) If $Z(c) > 0$ and $Y(c) \geq \theta_{Z(c),1}(c)$. Check car availability and dispatching threshold.
   7.1) $Y(c) = Y(c) - \min(Y(c), C)$. Decrement the lobby queue by the number of passengers that are served when the car is dispatched. The car capacity is limited to C passengers.
   7.2) $Z(c) = Z(c) - 1$. Decrement the number of available cars.
   7.3) $caFlag(c) = 1$. Set ca flag to indicate a ca lifetime is needed to continue sample path construction.
   7.4) Update the average waiting time for this constructed sample path.

8) Go to Step 5.1). Continue sample path construction until suspended.

As the algorithm above is running, we are estimating the passenger waiting time for both the actual system and for each constructed sample path. In Step 3.3) we update an estimate of the passenger waiting time each time a car is dispatched in the actual system. Similarly, in Step 7.4) we update estimates of the passenger waiting time each time

a car is dispatched in a constructed sample path. We use these passenger waiting time estimates in Step 1.1) to adjust the operating threshold in the actual elevator system in an effort to improve the passenger waiting time performance. To reflect this fact, we refer to each constructed sample path as a concurrent estimator and the algorithm itself as the concurrent estimation dispatching algorithm (CEDA). Note that the output of CEDA is the set of all concurrently constructed sample paths, estimates of passenger waiting times for each constructed sample path, and an estimate of the actual passenger waiting time in the real elevator system. Most importantly, the algorithm generates the suggested operating threshold used by the dispatching controller to decide when each car should be dispatched.

V. Implementation Issues

To use the CEDA in a actual elevator system, some implementation issues need to be resolved. In the subsections that follow, we address issues relating to: computational requirements (Section V-A), pa event lifetimes (Section V-B), passenger waiting time estimation (Section V-C), ca event lifetimes (Section V-C), and threshold adaptation (Section V-D). A summary of the adjustments made to the CEDA for implementation are then given (Section V-E). We should emphasize that our objective throughout is to resolve each issue in the *simplest* way possible, and then compare the results of the simplest possible implementation to those obtained through the state-of-the-art dispatching schemes mentioned in the introduction.

A. Computational Requirements

As we have seen, an optimal policy is defined by C^N different thresholds, where C is the elevator capacity and N is the total number of elevators. Thus, for example, if we have four cars each with a capacity of 20 passengers, we would need $20^4 = 160\,000$ concurrent estimators to choose the best threshold. While it is certainly possible to concurrently construct 160 000 sample paths, given the necessary computer memory, our approach here is to ignore the fact that the threshold is a function of both the number of passengers in the designated next car and the number of empty cars waiting at the first floor [recall (1)] and simply choose the threshold only as a function of the number of passengers in the designated next car. While this obviously results in a suboptimal solution, experiments in our previous paper [15], show that the numerical values of the thresholds are not particularly sensitive to the number of empty cars waiting at the first floor. With this simplification, we only need C concurrent estimators, and our problem reduces to choosing the threshold $\theta \in \{1, \cdots, C\}$ that matches, as best as possible, the elevator service rate to the passenger arrival rate.

To deal with the fact that the passenger arrival rate is changing throughout the uppeak period (as seen in Fig. 2), we will use the following strategy. We will partition the hour-long uppeak period into 12, 5-min long estimation intervals indexed by $p = 1, \cdots, 12$. We will use the CEDA to find the best threshold $\theta(p)$ to use during each interval. That is, for each interval p, we will observe the actual elevator system as it operates under $\theta(p)$, and we will construct C sample paths. At the end of interval p, we will estimate the waiting time for both the actual elevator system and for each of the C constructed sample paths. We will use these waiting time estimates to choose the threshold $\theta(p)$ to be used during interval p on the next day. If the passenger arrival profile has the same general statistics from one day to the next, this strategy is expected to work well.

B. pa Event Lifetimes

The CEDA requires pa event lifetimes for sample path construction. The problem here is that elevator systems usually do not have sensors to detect every passenger arrival event. Typical elevator systems, however, can obtain a reasonable estimate of the number of passengers inside a car when it is dispatched, since most elevator systems have weight sensors in each car or light beams which count passengers as they enter and exit the cars.

Given an estimate of the number of passengers inside a car at the time it is dispatched, we can use the rate at which passengers are being carried away from the first floor to estimate the passenger arrival rate. The estimate $\lambda_{est}(p, d)$ is computed for the dth time an elevator is dispatched during the pth 5-min estimation interval by dividing the total number of passengers *served so far* during interval p by the time since interval p began. The estimate is limited to a prespecified maximum value to deal with the large estimation errors that could otherwise result when a car is dispatched immediately after an interval change. Using this passenger arrival rate estimate, each time a pa event is needed for sample path construction during interval p, we will generate one according to a Poisson process at rate $\lambda_{est}(p, d)$ using a standard random variate generation technique (e.g., see [3]). The rate itself is assumed to vary according to a function such as that shown in Fig. 2, obtained from extensive empirical data collected in the elevator industry. The underlying Poisson assumption is also based on extensive empirical studies for elevator systems [11]. For the performance results contained in Section VI, it was important to adopt this widely used passenger arrival model in order to compare the CEDA with other dispatching algorithms used in the elevator industry.

C. Passenger Waiting Time Estimation

The threshold adaptation scheme we develop in Section V-E requires an estimate of the passenger waiting time in the actual elevator system. The passenger waiting time is measured from the instant a passenger arrives to the first floor lobby queue to the instant the car that serves the passenger is dispatched from the first floor. Although we know the time when a car is dispatched, we do not know the time each passenger arrived.

To estimate passenger waiting times, we will use the following strategy. Assume the first floor lobby queue is served first come first served (FCFS). Define the arrival time of the passenger at the front of the queue as $T_f(d)$. This will be the first passenger to enter a designated next car when

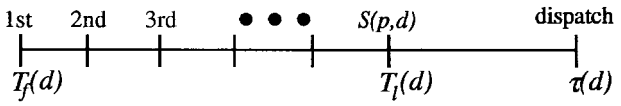

Fig. 6. Strategy for estimating passenger waiting times.

one becomes available for the dth time during interval p. The designated next car will serve all or part of the lobby queue. Define the arrival time of the last passenger who is able to load into the car as $T_l(d)$. As in Section V-B, assume the elevator system can determine the number of passengers in a car, and define $S(p, d)$ to be the number of passengers served the dth time an elevator is dispatched during interval p. Finally, define $\tau(d)$ to be the time the elevator is dispatched. Now partition the time interval $T_l(d) - T_f(d)$ uniformly as in Fig. 6. Then an estimate of the wait for the last passenger is $\tau(d) - T_l(d)$, for the second to last it is $\tau(d) - T_l(d) + (T_l(d) - T_f(d))/(S(p, d) - 1)$, for the third to last it is $\tau(d) - T_l(d) + 2(T_l(d) - T_f(d))/(S(p, d) - 1)$, and so on until finally the wait for the first passenger is estimated as $\tau(d) - T_f(d)$.

The total passenger wait the dth time an elevator is dispatched during interval p is, therefore, estimated as

$$W_a(p, d) = \sum_{k=0}^{S(p,d)-1} \left[(\tau(d) - T_l(d)) + k\left(\frac{T_l(d) - T_f(d)}{S(p, d) - 1} \right) \right]. \tag{20}$$

And the average passenger wait for interval p is estimated as

$$\overline{W}_a(p) = \frac{1}{\sum_d S(p, d)} \sum_d W_a(p, d). \tag{21}$$

Notice that we are estimating the average wait for those passengers who are *served* during interval p. In this extremely simplistic estimation procedure, we choose to disregard the fact that some of these passengers may have *arrived* during a previous interval. Clearly the strategy above assumes Poisson arrivals, and we justify it by noting again that Poisson arrivals have been shown to be a good model of passenger arrivals in elevator systems [11].

In (20) and (21), $\tau(d)$ and $S(p, d)$ can be directly measured. It is $T_f(d)$ and $T_l(d)$ that must be estimated. There are three dispatching situations that determine how these estimates are obtained.

Case I: The designated next car is already waiting at the first floor with its doors open when the first passenger arrives. In this case, $T_f(d)$ can be directly observed as the time the first passenger enters the car (detected by a weight sensor or a light beam in the car). In addition, we set $T_l(d) = \tau(d)$, since the car is expected to be dispatched upon arrival of the last passenger (i.e., the passenger that reaches the specified dispatching threshold, or the last passenger able to enter just before the doors close).

Case II: All cars are busy when the first passenger arrives. In this case, $T_f(d)$ can be directly observed as the time the first passenger pushes the elevator call button. The arrival time of the last passenger, however, is not known. The last passenger may have already been waiting when the next car became available, or the last passenger may have arrived and entered just as the doors closed and the car was dispatched. To estimate $T_l(d)$ we use our arrival rate estimate $\lambda_{est}(p, d)$ as follows:

$$T_l(d) = T_f(d) + \frac{S(p, d) - 1}{\lambda_{est}(p, d)}. \tag{22}$$

Here, of course, we must check that $T_l(d) < \tau(d)$. If not, we set $T_l(d) = \tau(d)$.

Case III: The $(d-1)$th dispatching event leaves part of the lobby queue behind because the car becomes full. In this case, when the dth dispatching occurs, we cannot observe either $T_f(d)$ or $T_l(d)$. What we will do in this case is the following: when the $(d-1)$th car departs full, we will estimate the arrival time of the first passenger in the lobby queue that remains as

$$T_f(d) = T_l(d-1) + \frac{1}{\lambda_{est}(p, d-1)}. \tag{23}$$

Then, when the dth dispatching occurs, we will proceed as in Case II above to estimate the arrival time of the last passenger.

Clearly, the waiting time estimation procedure above is quite crude. As already stated, however, our first goal is to examine whether our overall approach provides a significant performance improvement over state-of-the-art dispatching policies despite such crude estimation methods and rough approximations. Seeking more sophisticated methods is something we can pursue to provide further improvements as necessary.

Experimental results show that the quality of the waiting time estimate is best for Case I and worst for Case III. This is to be expected since the accuracy of the information on which the estimation is based, $T_f(d)$ and $T_l(d)$, becomes increasingly unreliable in going from Case I to Case III. Fortunately, Case III occurs only rarely (unless the arrival rate is very high in which case the dispatching threshold is $\theta = C$), so that the waiting time estimation error for a 5-min interval typically only amounts to a few seconds as shown in Fig. 7. The large estimation error during the ninth estimation interval is caused by the high arrival rate during the eighth interval: this high arrival rate leaves a large residual queue of passengers which do not get served until the ninth interval. Because $\lambda_{est}(p, d)$ is based on the number of passengers served during interval p, a large queue at the beginning of an interval biases the arrival rate estimate, which in turn biases the waiting time estimate for the Cases II and III dispatching situations during that interval.

D. ca Event Lifetimes

Although elevator systems have sensors for detecting ca events, there are two implementation issues we must resolve. The first issue concerns the passenger loading time and the second issue is a technical one concerning the TWA. Both of these issues can bias the waiting time estimates produced by the concurrent estimators.

Until now we have been assuming that the passenger loading time is negligible. That is, when constructing sample paths, an elevator returning to a lobby queue loads instantly and is dispatched immediately. This has the effect of decreasing the apparent service time, increasing the apparent handling

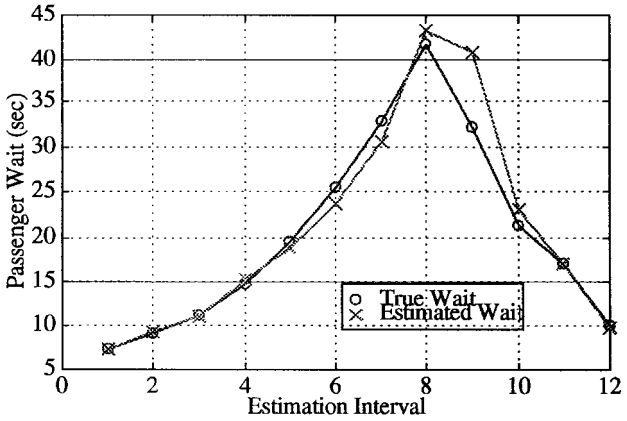

Fig. 7. Comparison between true wait and estimated wait for the uppeak arrival profile.

capacity, and changing the estimated optimal threshold. One way to deal with the passenger loading time is to introduce another event, a *passenger transfer event,* which occurs when a passenger enters an elevator at the first floor. Lifetimes for such events can be detected via weight sensors or counting devices in the cars. A simpler method, and the one we use, is to redefine the ca event lifetime to include the passenger loading time. That is, we will define the ca lifetime to be the time from when the first passenger enters the car up until the time that the car returns to the first floor after service. In this way we account for the passenger loading time without increasing the complexity of the dispatching algorithm.

Recall that the TWA algorithm, upon which the CEDA is based, requires all event lifetimes to be independent of the parameter θ, which in our case is the dispatching threshold. Generally, however, ca event lifetimes depend on the passenger load, and hence indirectly on the dispatching threshold. The most obvious way to deal with this issue is to keep a separate ca lifetime array for each passenger load. That is, instead of a single ca lifetime array, we would have C such arrays. Then when a ca lifetime is needed during sample path construction, we check the passenger load in the car being dispatched, and get the ca lifetime from the appropriate lifetime array. This approach, however, greatly increases the probability that sample path construction will get suspended at Step 5.2.1 of the CEDA. Another possible approach is to make observations of the system as it operates to find the mean ca lifetime as a function of the passenger load and use these means to appropriately scale the ca lifetimes used in the concurrent estimators. The drawback of this approach is the extensive data collection required.

In the spirit of our goal to start with an implementation which is as simple as possible, we have simply neglected the influence of the passenger load altogether. This approximation allows us to work with a single ca lifetime array, and it reduces the chance that a concurrent estimator will become suspended waiting for a ca lifetime.

E. Threshold Adaptation

The goal of threshold adaptation is to choose the best threshold to use for each of the 12 5-min estimation intervals in an uppeak traffic period. The simplest scheme is to determine the index c of the best performing concurrent estimator during interval p and choose the threshold associated with that concurrent estimator. That is, if concurrent estimator c (which in our implementation uses a dispatching threshold of c) gives the best estimated waiting time for interval p, then set $\theta(p) = c$ as the threshold to be used during interval p on the next day. A potential problem with this approach is that the threshold that gives the best wait for the concurrent estimators may not be the one that gives the best wait for the actual elevator system. The reason has to do with the various approximations we use to generate pa and ca lifetimes for concurrent sample path construction. To deal with this potential problem, we propose a simple adaptation algorithm. To describe the algorithm, let $\theta_a^n(p)$ be the operating threshold that was used in the actual system on the nth day during interval p, and let $\overline{W}_a(p)$ [obtained using (21)] be the estimated average wait in the actual system during interval p. Let $\overline{W}_c(p)$ be the estimated wait for concurrent estimator c during interval p, and let $\overline{W}_c^*(p) = \min_c[\overline{W}_c(p)]$ be the best estimated wait over all of the concurrent estimators during interval p. Let $\theta_c^*(p) = \arg\min_c[\overline{W}_c(p)]$ be the threshold used by the best performing concurrent estimator. Finally, define $\Delta = |\overline{W}_a(p) - \overline{W}_c^*(p)|/\overline{W}_a(p)$ and adapt the threshold according to

$$\theta_a^{n+1}(p) \leftarrow \begin{cases} \theta_c^*(p) & \beta < \Delta \\ \theta_a^n(p) + \operatorname{sgn}(\theta_c^*(p) - \theta_a^n(p)) & \alpha < \Delta \le \beta \\ \theta_a^n(p) & \Delta \le \alpha. \end{cases} \tag{24}$$

Here α and β are scalars. These scalars were chosen with the following intention in mind. When the difference Δ between the best estimated wait and the actual wait is "small," we leave the threshold unchanged. When this difference is "large," we immediately switch to the best threshold given by our concurrent estimators. When Δ is in mid-range, we adjust the threshold in the appropriate direction by a small amount. As we will see in the next section, we obtained this type of behavior with the values $\alpha = 0.2$ and $\beta = 0.8$.

F. Implementation Adjustments

Here we summarize the adjustments made to specific steps of the CEDA for implementation purposes (each step below refers to the corresponding step with the same number shown in the detailed algorithm in Section IV).

Step 1.1: Check $T(0)$ to see if the time interval p has changed. If it has, update the threshold $\theta(p)$ [by either setting it to the threshold of the best performing concurrent estimator, or by using the adaptation algorithm (24)], and then set $p \leftarrow p+1$.

Step 2.1: If this pa event is for the "first" passenger (as defined in Section V-C), then determine the dispatching case and compute $T_f(d)$. When the first passenger arrives to an empty car, we have Case I, and $T_f(d)$ is the time the passenger enters the car. If the first passenger arrives when all the cars are busy, we have Case II, and $T_f(d)$ is the time the passenger pushes the elevator call button.

Step 3: Dispatch when the number of passengers inside the designated next car reaches the threshold $\theta(p)$. When a car is dispatched, determine the load $S(p, d)$, and update $\lambda_{est}(p, d)$ and $W(p, d)$. When a car departs full, we have dispatching Case III, and $T_f(d+1)$ is estimated using (23).

Step 4: Replace K by C for the number of constructed sample paths: one for each threshold $\theta \in \{1, \cdots, C\}$.

Steps 5.1.1–5.1.3: We will not suspend sample path construction to wait for passenger arrival events. Instead, when we need a pa lifetime for sample path construction, we will generate one according to a Poisson process at rate $\lambda_{est}(p, d)$.

Step 7: Dispatch in a concurrent estimator when $Y(c) \geq c$ and $Z(c) > 0$, i.e., the threshold for the cth concurrent estimator is c.

VI. Performance Results

In this section we provide explicit numerical results obtained by applying the CEDA to a detailed elevator system simulator developed at the University of Massachusetts in collaboration with the OTIS Elevator Co. The simulator models a ten-floor office building served by four elevator cars, each with a capacity of 20 passengers. Details regarding the simulator can be found in [12] and [13].

A. CEDA Performance Under a Fixed Passenger Arrival Rate

As a test of the accuracy of the CEDA, we performed the following experiment. We fixed the arrival rate of incoming passengers to 20 pass/min. Then we obtained a response curve describing the true average passenger waiting time for each of the 20 dispatching thresholds $\theta = 1, \cdots, 20$. We did this by performing 30 runs of one simulated hour apiece at each threshold. At the completion of each run, the true waiting time was determined (since the complete state information is available in the simulator, the true passenger wait can be determined). The waiting time for each threshold was then averaged over the 30 runs at that threshold. The results are shown in Fig. 8, where the best dispatching threshold for this arrival rate is seen to be $\theta = 9$ passengers with a wait of 14.56 s. Notice that the curve has one local minimum and one local maximum. Also notice the importance of choosing the correct threshold. For example, if we were to use the simple $\theta = 1$ policy the wait is 18.86 s, some 30% longer than the best. Alternatively, the half-capacity plus 20-s timeout policy discussed in the introduction gives a waiting time of 16.84 s, some 16% longer than the best. As seen from Fig. 8, this is worse than simply using a half-capacity policy which gives a wait of 14.91 s. The reason for the difference is that the 20 s timeout acts like a threshold of 6.67 passengers (6.67 = 20 s $\times$ 20 pass/min $\times$ 1/60 min/s).

Next we performed 30 additional runs for the same arrival rate of 20 incoming pass/min. Our interest here was to see what threshold the CEDA would choose. For the first run, the threshold was initialized to $\theta = 1$. For subsequent runs, the threshold changed as the CEDA adapted it to the 20 pass/min arrival rate. At the end of the first run, the threshold to be used for the second run was set to the threshold of the concurrent estimator that gave the best estimated wait for the first run. For subsequent runs, the threshold was tuned using (24).

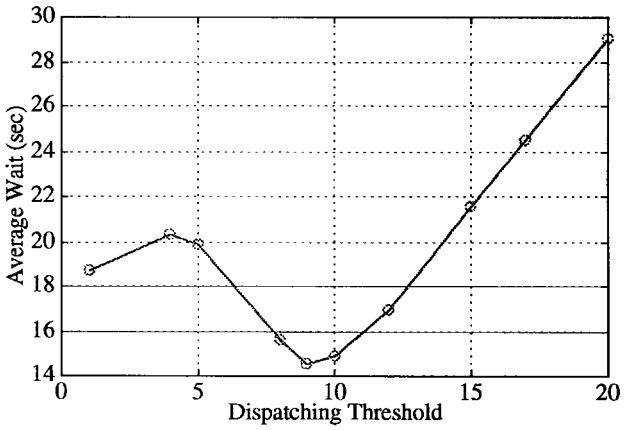

Fig. 8. Response curve describing the true average wait as a function of the dispatching threshold for a fixed arrival rate of 20 incoming pass/min (no outgoing or interfloor).

In comparison to the response curve obtained above using brute force simulation, we observed that the concurrent estimators did not give a particularly good estimate of the wait in the actual system. Those concurrent estimators operating at low thresholds tended to overestimate the wait, while those operating at higher thresholds tended to underestimate the wait. The reasons for these estimation errors can be traced back to inaccuracies in the arrival rate estimate which is used to generate pa events for sample path construction (Section V-B), and inaccuracies in the ca event lifetimes used to generate ca events for sample path construction (Section V-D). Despite the fact that the concurrent estimators are really not that accurate, however, the CEDA did converge to a mean threshold of $\theta = 9.26$ passengers, which is essentially equal to the true best threshold of $\theta = 9$ obtained by brute force simulation. This is a feature of algorithms, such as the CEDA, that are based on ordinal comparisons: although modeling errors may yield poor estimates of the performance of the actual system, the strategy that optimizes performance can be very robust with respect to such errors. The robustness of solutions to optimization problems with respect to modeling errors is not unusual (e.g., see [14]).

B. CEDA Performance in Uppeak Traffic

In this section, incoming passengers arrive to the first floor with a mean rate that varies as shown in Fig. 2. As mentioned in Section V-B, this passenger arrival model is widely used in the elevator industry and was adopted here for the sake of meaningful comparisons with other schemes (see Table I). There are no outgoing or interfloor passengers. This traffic simulates the uppeak traffic mode. Recall, the CEDA uses 12 different thresholds $\theta(p)$, one for each 5-min interval p. On the first day, the thresholds were arbitrarily initialized to $\theta(p) = 1$ for $p = 1, \cdots, 12$. At the end of the first day, the threshold to be used during interval p on the next day was set to the threshold of the concurrent estimator that gave the best estimated wait for that interval. On subsequent days, the

TABLE I
WAITING TIME PERFORMANCE FOR UPPEAK PASSENGER TRAFFIC

Policy	True average wait over entire hour-long uppeak period (sec.)	Remarks
$\theta = 1$	33.20	This policy tries to dispatch as soon as a passenger enters a car.
$\theta = 5$	34.50	
$\theta = 10$	36.33	This is a half-capacity policy.
$\theta = 15$	35.16	
$\theta = 20$	35.94	This policy dispatches when a car becomes full.
half capacity plus 20 second timeout	34.85	This is a commonly used uppeak dispatching policy.
CEDA	24.61	This is our Concurrent Estimation Dispatching Algorithm.

threshold adaptation scheme (Section V-E) was used to further adapt and fine tune the threshold. All results are averages over 30 runs of one simulated hour apiece.

Table I compares the performance of the CEDA to several open-loop threshold policies. It is interesting to notice that all of the open loop policies give essentially the same waiting time performance. The CEDA which adapts the threshold to the changing arrival rate, however, gives far superior performance—some 35% better than the best static, open-loop policy.

A better way to compare dispatching policies during the uppeak period is to examine the average wait during each 5-min interval as shown in Fig. 9. As expected, the $\theta = 1$ policy does well at the beginning and end of the uppeak period when the arrival rate is low. Similarly, the $\theta = 10$ policy does well for medium arrival rates, and the $\theta = 20$ does well for high arrival rates. No static, open-loop policy, however, does well for all intervals. In comparison, the CEDA, which uses a different threshold for each 5-min interval, does very well—essentially giving lower bound performance.

Table II below shows a typical evolution of the CEDA thresholds. As can be seen, a drastic threshold adjustment takes place immediately after the first day, with few changes occurring after about one week of operation. The rapid initial adjustment was achieved by setting the thresholds to those giving the best concurrently estimated wait. This put the thresholds close to their optimal values. Because of errors in the concurrent estimators, the threshold adaptation algorithm (24) was then needed to fine tune the thresholds. As expected,

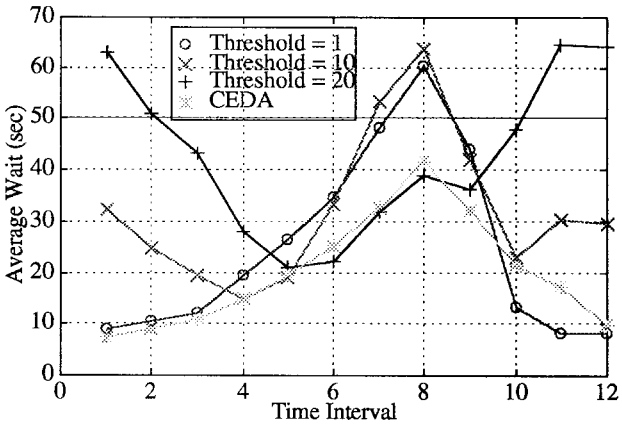

Fig. 9. True wait for each 5-min interval during the hour-long uppeak period.

where the arrival rate is low, the threshold is low, and where the arrival rate is high, the threshold is high. The thresholds for intervals 10 and 11, however, are higher than we would expect them to be (in comparison to the thresholds for intervals 1 and 2). This causes the somewhat poor waiting time performance during these intervals in Fig. 9. The reason can be traced to day eight, where, something about the sample path on that day caused the threshold for interval 10 to get set to 20 passengers (the car capacity). Subsequent correction was so gradual that the threshold remained too high for most of the 30 days. This in turn caused the average wait for interval ten to be high. Then, because the threshold during interval ten was too high, this resulted in residual queues at the end of the interval.

TABLE II
EVOLUTION OF THE CEDA THRESHOLDS

	interval											
	1	2	3	4	5	6	7	8	9	10	11	12
	mean passenger arrival rate (pass/min)											
	9	12	14	22	29	34	36	34	19	12	9	9
day	dispatching threshold											
1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	3	3	6	9	14	18	12	12	4	1	2
5	2	3	4	8	11	15	17	13	11	7	3	2
7	3	3	4	8	12	13	17	13	9	8	4	3
8	3	4	4	8	13	14	18	12	10	**20**	3	3
10	3	4	2	9	13	15	19	13	11	18	8	3
15	3	4	4	9	15	16	18	12	11	13	7	4
20	3	4	5	9	15	17	17	14	10	10	6	2
25	3	4	6	8	15	17	16	15	9	6	9	5
30	2	2	6	7	13	18	11	19	11	6	5	3

This biased the apparent passenger arrival rate for interval 11, causing an increase in the threshold and the waiting time for that interval as well. This behavior, however, actually serves well to illustrate the adaptive capability of our CEDA-based controller.

VII. CONCLUSION

In this paper an on-line adaptive dispatching control algorithm was designed for use in elevator systems during uppeak passenger traffic. The design of the algorithm was motivated by our previous paper [15] where we proved that for a queueing model of the uppeak dispatching problem the structure of the optimal dispatching control policy minimizing the average passenger waiting time is a threshold-based policy with threshold parameters that depend on the passenger arrival rate. The CEDA presented in Section IV is based on the TWA from [5]. The CEDA allows us to observe the elevator system, in a unobtrusive way, while it operates under some arbitrary thresholds and concurrently estimate what the waiting time would have been had we operated the system under a set of different thresholds. These concurrently estimated waiting times are then used to adapt the operating threshold to the changing passenger arrival rate. The implementation of our algorithm is simple and does not require anything more than most elevator systems can already supply, viz.: the ability to detect button push events and car arrival events, and the ability to provide a reasonable estimate of the number of passengers inside the elevator cars. In addition, our algorithm requires only modest memory, and the most complicated calculation is the generation of pa event lifetime samples from a Poisson process.

Conceptually, the CEDA is trying to adapt the operating threshold so that the elevator service rate tracks the passenger arrival rate. In this respect, the CEDA is similar to an algorithm from the literature called the rate matching algorithm (RMA) [13]. Like the CEDA, RMA partitions the uppeak period into 12 5-min long intervals. Then given historical information regarding the passenger arrival rate, the best threshold to use for each interval is determined off-line as the smallest threshold for which the elevator service rate (obtained by dividing the passenger load by the predicted service time) first exceeds the arrival rate. To predict the service time, RMA gives each floor a probability as a passenger destination and uses probability arguments, based on the number of passengers in the car, to predict how high the car will go and the number of stops it will make. Once the thresholds have been chosen, they are fixed, and RMA operates open-loop. As far as we know, however, RMA is not used in practice. The algorithms that are most often used seem to be static, open-loop policies similar to the half-capacity plus timeout policy described in the body of the paper. As we showed, the CEDA performs much better than any static, open-loop policy and is thus well

suited when the arrival rate profile has similar statistics from one day to the next.

ACKNOWLEDGMENT

The authors wish to thank Dr. Bruce A. Powell of the OTIS Elevator Company for his input in this work, including the traffic data that was used for the implementation results reported in Section VI.

REFERENCES

[1] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines.* Chichester, U.K.: Wiley, 1989.
[2] G. C. Barney and S. M. dos Santos, *Elevator Traffic Analysis Design and Control*, 2nd ed. London, U.K.: Peter Peregrinus, 1985.
[3] C. G. Cassandras, *Discrete-Event Systems: Modeling and Performance Analysis.* Boston, MA: Richard D. Irwin and Aksen Associates, 1993.
[4] C. G. Cassandras and J. Pan, "Parallel sample path generation for discrete-event systems and the traffic smoothing problem," *J. Discrete-Event Dynamic Syst.*, vol. 5, no. 2/3, pp. 187–217, 1995.
[5] C. G. Cassandras and C. Panayiotou, "Concurrent sample path estimation for discrete-event systems," in *Proc. 35th Conf. Decision and Contr.*, 1996, pp. 3332–3337.
[6] C. G. Cassandras and S. G. Strickland, "On-line sensitivity analysis of Markov chains," *IEEE Trans. Automat. Contr.*, vol. 34, pp. 76–86, 1989.
[7] ———, "Observable augmented systems for sensitivity analysis of Markov and semi-Markov processes," *IEEE Trans. Automat. Contr.*, vol. 34, pp. 1026–1037, 1989.
[8] P. Glasserman, *Gradient Estimation Via Perturbation Analysis.* Boston, MA: Kluwer, 1991.
[9] W. B. Gong, Y. C. Ho, and W. Zhai, "Stochastic comparison algorithm for discrete optimization with estimation," in *Proc. 31st IEEE Conf. Decision Contr.*, 1992, pp. 795–802.
[10] Y. C. Ho and X. R. Cao, *Perturbation Analysis of Discrete-Event Dynamic Systems.* Boston, MA: Kluwer, 1991.
[11] G. T. Hummet, T. D. Moser, and B. A. Powell, "Real time simulation of elevators," in *Winter Simulation Conf.*, Miami Beach, Dec. 4–6, 1978, pp. 393–402.
[12] J. Lewis, "An elevator simulator with a relative system response group control algorithm," Tech. Rep. CCS-88-102, Dept. Elect. Comput. Eng., Univ. Mass., Amherst, 1988.
[13] J. Lewis, "A dynamic load balancing approach to the control of multiserver polling systems with applications to elevator system dispatching," Doctoral dissertation, Dept. Elect. and Comput. Eng., Univ. Mass., Amherst, 1991.
[14] B. Mohanty and C. G. Cassandras, "The effect of model uncertainty on some optimal routing problems," *J. Optimization Theory and Applicat.*, vol. 77, pp. 257–290, 1993.
[15] D. L. Pepyne and C. G. Cassandras, "Optimal dispatching control for elevator systems during uppeak traffic," *IEEE Trans. Contr. Syst. Technol.*, vol. 5, pp. 629–643, 1997.
[16] M. L. Siikonen, "Elevator traffic simulation," *Simulation*, vol. 61, no. 4, pp. 257–267, 1993.
[17] G. R. Strakosch, *Vertical Transportation: Elevators and Escalators.* New York: Wiley, 1983.
[18] A. Torn and A. Zilinskas, *Global Optimization, Lecture Notes in Computer Science.* Berlin, Germany: Springer-Verlag, 1987.
[19] P. Vakili, "A standard clock technique for efficient simulation," *Operations Res. Lett.*, vol. 10, pp. 445–452, 1991.

David L. Pepyne received the B.S.E.E. degree from the University of Hartford, CT, in 1986. In 1991, he entered the University of Massachusetts, Amherst, where he is currently completing the Ph.D. degree.

From 1986 to 1990, he was a Flight Test Engineer with the U.S. Air Force at Edwards AFB, CA. His current research interests include the performance optimization of discrete-event and hybrid systems and nonlinear optimization techniques.

Christos G. Cassandras (S'82–M'82–SM'91–F'96) received the B.S. degree from Yale University, New Haven, CT, in 1977, the M.S.E.E degree from Stanford University, CA, in 1978, and the S.M. and Ph.D. degrees from Harvard University, Cambridge, MA, in 1979 and 1982, respectively.

From 1981 to 1984, he was with ITP Boston, Inc., where he worked on control systems for computer-integrated manufacturing. In 1984, he joined the faculty of the Department of Electrical and Computer Engineering, University of Massachusetts at Amherst, until 1996. He is currently Professor of Manufacturing Engineering and Professor of Electrical and Computer Engineering at Boston University. His research interests include discrete-event systems, stochastic optimization, computer simulation, and performance evaluation and control of computer networks and manufacturing systems. He is the author of more than 100 technical publications in these areas, including a textbook.

Dr. Cassandras is on the Board of Governors of the IEEE Control Systems Society and is Editor–in–Chief of the IEEE TRANSACTIONS ON AUTOMATIC CONTROL. He serves on several other editorial boards and has guest-edited for various journals. He was awarded a Lilly Fellowship in 1991. He is a member of Phi Beta Kappa and Tau Beta Pi.