Article 6.43.27

DISCRETE EVENT SYSTEMS

Christos G. Cassandras, Dept. of Manufacturing Engineering, 15 St. Mary's St., Boston University, Brookline, MA 02446, USA

Keywords: Time-driven System, Event-driven System, Language, Automaton, Supervisory Control, Sample Path Analysis, Petri Net, Dioid Algebra, Simulation.

Contents:

Glossary:

Event-driven system: A dynamic system whose state changes only as a result of the asynchronous occurrence of various discrete events.

Time-driven system: A dynamic system whose state continuously changes as time evolves.

SUMMARY

Discrete Event Systems (DES) are characterized by the occurrence of discrete events asynchronously over time which are responsible for driving all dynamics. Such systems are ubiquitous in modern technological environments, ranging from communication networks and manufacturing to transportation and logistics. This class of event-driven dynamic systems is first compared to traditional time-driven systems described by differential (or difference) equations. We subsequently overview some of the major modeling frameworks for DES, based on automata, Petri nets, and dioid algebras. The use of these models is illustrated through queueing systems, a common class of DES.

1 Introduction

The term "discrete event system" was introduced in the early 1980s to identify an increasingly important class of dynamic systems in terms of their most critical feature: the fact that their behavior is governed by *discrete events* occurring asynchronously over time and solely responsible for generating state transitions. In between event occurrences, the state of such systems is unaffected. Examples of such behavior abound in technnological environments such as computer and communication networks, automated manufacturing systems, air traffic control systems, C^4I (command, control, communication, and information) systems, advanced monitoring and control systems in automobiles or large buildings, intelligent transportation systems, distributed software systems, and so forth. The operation of such environments is largely regulated by human-made rules for initiating or terminating activities and scheduling the use of resources through controlled events, such as hitting a keyboard key, turning a piece of equipment "on", or sending a message packet. In addition, there are numerous uncontrolled randomly occurring events, such as a spontaneous equipment failure or a packet loss, which may or may not be observable through sensors.

We shall henceforth use the acronym DES for "Discrete Event System", but must point out that the acronym DEDS, for "Discrete Event Dynamic System", is also commonly used to emphasize that it is the dynamics of such systems that render them particularly interesting.

The conceptual and practical challenges in the development of the DES field may be summarized as follows:

1. The types of variables involved in the description of a DES are both continuous and discrete, sometimes purely symbolic, i.e., non-numeric (as in describing the state of a piece of equipment as "on" or "off"). This renders traditional mathematical models based on differential (or difference) equations inadequate, and methods relying on the power of calculus are, consequently, of limited use. New modeling frameworks, analysis techniques, and control procedures are required to suit the structure of a DES.

2. Because of the asynchronous nature of events that cause state transitions in DES, it is neither natural nor efficient to use time as a synchronizing element driving the system dynamics (details on this issue are provided in Section 2). It is for this reason that DES are often referred to as *event-driven*, to contrast them to classical *time-driven* systems based on the laws of physics; in the latter, as time evolves state variables such as position, velocity, temperature, pressure, current, voltage, etc. also continuously evolve. In order to capture event-driven state dynamics, however, new mathematical models are necessary.

3. Uncertainties are inherent in the technological environments where DES are encountered. Therefore, the mathematical models used for DES and all associated methods for analysis and control must incorporate this element of uncertainty, sometimes by explicitly modeling nondeterministic behavior and often through the inclusion of appropriate stochastic model components.

4. Complexity is also inherent in DES of practical interest, usually manifesting itself in the form of combinatorially explosive state spaces. Purely analytical methods for DES design, analysis, and control have proved to be limited. A large part of the progress made in this field has relied on the development of new paradigms charcaterized by a combination of mathematical techniques and effective processing of experimental data.

The remainder of this article is organized as follows. In Section 2, we contrast time-driven and event-driven systems as a means of highlighting the characteristics of DES that motivate the development of the main modeling frameworks and methodologies for design and control of such systems. In Section 3, we describe the different levels of abstraction used for modeling DES, depending on the objectives of the analysis and the problems to be addressed. This will also help understand the nature of the subsequent three articles that focus on the supervisory control framework and on methodologies based on sample path analysis. Finally, in Section 4 we provide an overview of the main modeling approaches for DES. Without burdening the reader with excessive technical details, we compare different models for a class of DES that constitutes a basic building block in many applications.

2 Event-Driven and Time-Driven Systems

As previously mentioned, today's technological and increasingly computer-dependent world includes numerous examples of dynamic systems with the following two features. First, many of the quantities we deal with are "discrete", typically involving counting integer numbers (how many parts are in an inventory, how many planes are in a runway, how many telephone calls are active). Second, what drives many of the processes we use and depend on is a variety of instantaneous "events" such as the pushing of a button, hitting a keyboard key, or a traffic light turning green. In this section, we will explain how these features amount to fundamental differences between traditional dynamic systems modeled through differential (or difference) equations and the new class of DES.

Let us begin with the concept of "event". We do not attempt to formally define it, since it is a primitive concept with a good intuitive basis. We only wish to emphasize that an event should be thought of as occurring instantaneously and causing transitions from one system state value to another. An event may be identified with a specific action taken (e.g., somebody presses a button). It may be viewed as a spontaneous occurrence dictated by nature (e.g., a computer goes down for whatever reason too complicated to figure out) or it may be the result of several conditions which are suddenly all met (e.g., the fluid level in a tank exceeds a given value). For the purpose of developing a model for a DES, we will use the symbol e to denote an event. When considering a system affected by different types of events, we will assume we can define an *event set* E whose elements are all these events. Clearly, E is a discrete set.

Let us next concentrate on the nature of the state space of a system. In continuous-state systems the state generally changes as time changes. This is particularly evident in discrete-time models: the "clock" is what drives a typical sample path. With every "clock tick" the

state is expected to change, since *continuous* state variables *continuously* change with time. It is because of this property that we refer to such systems as *time-driven*. In this case, time is a natural independent variable which appears as the argument of all input, state, and output functions involved in modeling a system.

In DES, at least some of the state variables are discrete and their values change only at certain points in time through instantaneous transitions which we associate with "events". What is important is to specify the timing mechanism based on which events take place. Let us assume there exists a clock through which we will measure time, and consider two possibilities: (*i*) At every clock tick an event e is selected from the event set E (if no event takes place, we can think of a "null event" as being a member of E, whose property is that it causes no state change), and (*ii*) At various time instants (not necessarily known in advance and not necessarily coinciding with clock ticks), some event e "announces" that it is occurring. There is a fundamental difference between (*i*) and (*ii*) above. In (*i*), state transitions are *synchronized* by the clock: there is a clock tick, an event (or no event) is selected, the state changes, and the process repeats. Thus, the clock alone is responsible for any possible state transition. In (*ii*), every event $e \in E$ defines a distinct process through which the time instants when e occurs are determined. State transitions are the result of combining these *asynchronous* concurrent event processes. Moreover, these processes need not be independent of each other. The distinction between (*i*) and (*ii*) gives rise to the terms *time-driven* and *event-driven* systems respectively. Continuous-state systems are, by their nature, time-driven. However, in discrete-state systems this depends on whether state transitions are synchronized by a clock or occur asynchronously as in scheme (*ii*) above. Clearly, event-driven systems are more complicated to model and analyze, since there are several asynchronous event-timing mechanisms to be specified as part of our understanding of the system.

In view of this discussion, let us now turn our attention to mathematical models one can use for time-driven and event-driven systems. In the former case, the field of systems and control has based much of its success on the use of well-known differential-equation-based models, such as

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t), \qquad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{1}$$

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{2}$$

where (1) is a (vector) state equation with initial conditions specified, and (2) is a (vector) output equation. As is common in system theory, $\mathbf{x}(t)$ denotes the state of the system, $\mathbf{y}(t)$ is the output, and $\mathbf{u}(t)$ represents the input, often associated with controllable variables used to manipulate the state so as to attain a desired output. In order to use this type of model, the two key properties that the system must satisfy are: (*i*) it has a continuous state space, and (*ii*) the state transition mechanism is time-driven. The first property allows us to define the state by means of continuous variables, which can take on any real (or complex) value. It is because of this reason that this class of systems is often referred to as *Continuous Variable Dynamic Systems* (CVDS). Common physical quantities such as position, velocity, acceleration, temperature, pressure, flow, etc. fall in this category. Since we can naturally define time derivatives for these continuous variables, differential equation models like (1) can be used.

The second property points to the fact that the state generally changes as time changes. As a result, the time variable t (or some integer $k = 0, 1, 2, \ldots$ in discrete time) is a natural independent variable for modeling such systems.

In contrast to a CVDS, in a DES the state space is discrete (or at least includes several discrete variables) and the state transition mechanism is event-driven. From a modeling point of view, the latter property has the following implication. If we can identify a set of "events" any one of which can cause a state transition, then time no longer serves the purpose of driving such a system and may no longer be an appropriate independent variable.

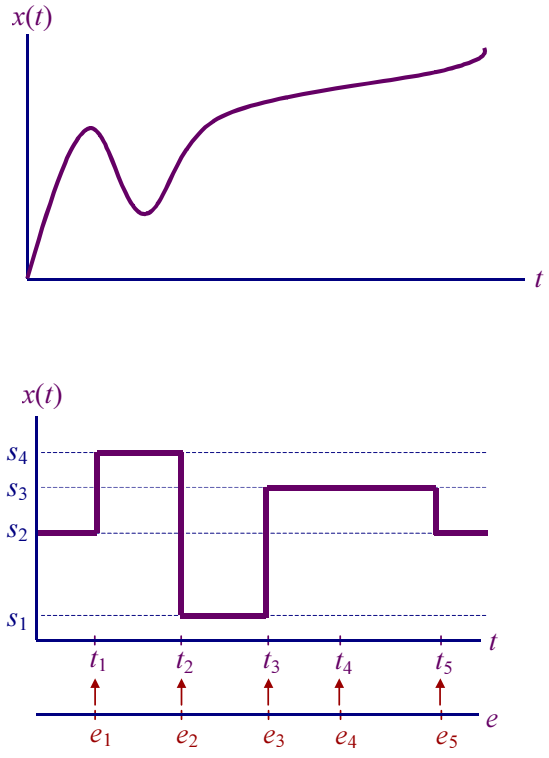

Figure 1: Comparison of time-driven and event-driven sample paths

These two characteristics are the ones used in defining a DES as a discrete-state, event-driven system, that is, we view a DES as one whose state evolution depends entirely on the occurrence of asynchronous discrete events over time. In fact, comparing state trajectories (sample paths) of CVDS and DES is useful in understanding the differences between the two and setting the stage for DES modeling frameworks. Thus, comparing typical sample paths from each of these system classes, as in Fig. 1, we observe the following:

- For the time-driven CVDS shown, the state space X is the set of real numbers $\mathbb{R}$, and

$x(t)$ can take any value from this set. The function $x(t)$ is the solution of a differential equation of the general form $\dot{x}(t) = f(x(t), u(t), t)$, where $u(t)$ is the input.

- For the event-driven DES, the state space is some discrete set $X = \{s_1, s_2, s_3, s_4\}$. The sample path can only jump from one state to another whenever an event occurs. Note that an event may take place, but not cause a state transition, as in the case of e_4. There is no immediately obvious analog to $\dot{x}(t) = f(x(t), u(t), t)$, i.e., no mechanism to specify how events might interact over time or how their time of occurrence might be determined. Thus, a large part of the early developments in the DES field has been devoted to the specification of an appropriate mathematical model containing the same expressive power as (1)-(2).

We should point out that discrete *event* systems should not be confused with discrete *time* systems. The class of discrete time systems contains both CVDS and DES. In other words, a DES may be modeled in continuous or in discrete time, just like a CVDS can.

3 Abstraction Levels in the Study of Discrete Event Systems

Let us return to the DES sample path shown in Fig. 1. Instead of plotting the piecewise constant function $x(t)$ as shown, it is often convenient to simply write the timed sequence of events

$$(e_1, t_1), (e_2, t_2), (e_3, t_3), (e_4, t_4), (e_5, t_5) \tag{3}$$

which contains the same information as the sample path depicted in Fig. 1. The first event is e_1 and it occurs at time $t = t_1$; the second event is e_2 and it occurs at time $t = t_2$, and so forth. When this notation is used, it is implicitly assumed that the initial state of the system, s_2 in this case, is known and that the system is "deterministic" in the sense that the next state after the occurrence of an event is unique. Thus, from the sequence of events in (3), we can recover the state of the system at any point in time and reconstruct the DES sample path in Fig. 1.

Consider the set of all possible timed sequences of events that a given system can ever execute. We call this set the *timed language* model of the system. The word "language" comes from the fact that we can think of the event E as an "alphabet" and of (finite) sequences of events as "words". We can further refine our model of the system if some statistical information is available about the set of sample paths of the system. Let us assume that probability distribution functions are available about the "lifetime" of each event type $e \in E$, that is, the elapsed time between successive occurrences of this particular e. We call a *stochastic timed language* a timed language together with associated probability distribution functions for the events. The stochastic timed language is then a model of the system that lists all possible sample paths together with relevant statistical information about them.

Stochastic timed language modeling is the most detailed in the sense that it contains event information in the form of event occurrences and their orderings, information about the exact

times at which the events occur (and not only their relative ordering), and statistical information about successive occurrences of events. If we omit the statistical information, then the corresponding timed language enumerates all the possible sample paths of the DES, with timing information. Finally, if we delete the timing information from a timed language we obtain an *untimed language*, or simply *language*, which is the set of all possible orderings of events that could happen in the given system. Deleting the timing information from a timed language means deleting the time of occurrence of each event in each timed sequence in the timed language. For example, the untimed sequence corresponding to the timed sequence of events in (3) is

$$\{e_1, e_2, e_3, e_4, e_5\}.$$

Languages, timed languages, and stochastic timed languages represent different levels of abstraction at which DES are modeled and studied: untimed (or logical), timed, and stochastic. The choice of the appropriate level of abstraction clearly depends on the objectives of the analysis. In many instances, we are interested in the "logical behavior" of the system, that is, in ensuring that a precise *ordering of events* takes place which satisfies a given set of specifications (e.g., first-come first-served in a job processing system). Or we may be interested in finding if a particular state (or set of states) of the system can be reached or not. In this context, the actual timing of events is not required, and it is sufficient to model only the untimed behavior of the system, that is, consider the language model of the system. *Supervisory control* is the term established for describing the systematic means (i.e., enabling or disabling events which are controllable) by which the logical behavior of a DES is regulated to achieve a given specification. This topic is further discussed in Article 6.43.27.2.

Next, we may become interested in *event timing* in order to answer questions such as: "How much time does the system spend at a particular state?" or "How soon can a particular state be reached?" or "Can this sequence of events be completed by a particular deadline?" These and related questions are often crucial parts of the design specifications. More generally, event timing is important in assessing the performance of a DES often measured through quantities such as *throughput* or *response time*. In these instances, we need to consider the timed language model of the system. The fact that different event processes are concurrent and often interdependent in complex ways presents great challenges both for modeling and analysis of timed DES. Moreover, since we cannot ignore the fact that DES frequently operate in a stochastic setting (e.g., the time when some equipment fails is unpredictable), an additional level of complexity is introduced, necessitating the development of probabilistic models and related analytical methodologies for design and performance analysis based on stochastic timed language models. *Sample path analysis* refers to the study of sample paths of DES, focusing on the extraction of information for the purpose of efficiently estimating performance sensitivities of the system and, ultimately, achieving on-line control and optimization (see "*Sample Path Analysis of Discrete Event Dynamic Systems*" for more details).

These different levels of abstraction are complementary, as they address different issues about the behavior of a DES. Indeed, the literature in DES is quite broad and varied as extensive research has been done on modeling, analysis, control, optimization, and simulation at all levels. Although the language-based approach to discrete event modeling is attractive in presenting

modeling issues and discussing system-theoretic properties of DES, it is by itself not convenient to address verification, controller synthesis, or performance issues; what is also needed is a convenient way of *representing* languages, timed languages, and stochastic timed languages. If a language (or timed language or stochastic timed language) is finite, we could always list all its elements, that is, all the possible sample paths that the system can execute. Unfortunately, this is rarely practical. Preferably, we would like to use *discrete event modeling formalisms* that would allow us to represent languages in a manner that highlights structural information about the system behavior and that is convenient to manipulate when addressing analysis and controller synthesis issues. Discrete event modeling formalisms can be untimed, timed, or stochastic, according to the level of abstraction of interest. In the next section, we provide a brief introduction to one of the major modeling formalisms, based on *automata*, which also forms the foundation for supervisory control and sample path analysis (further discussed in "*Supervisory Control of Discrete Event Systems*" and "*Sample Path Analysis of Discrete Event Dynamic Systems*"). Some more details on the manipulation of automata for the construction of system models from component models and comparisons with other modeling approaches are given in "*Modeling of Discrete Event Systems*". We will use automata to illustrate the construction of models for a common class of DES and contrast it to two other modeling frameworks.

4 Modeling Overview

The introduction to DES in the previous sections has served to point out the main characteristics of these systems. Two elements which have emerged as essential in defining a DES are: a discrete state space, which we denote by X, and a discrete event set, which we denote by E. We can now build on this basic understanding in order to develop some formal models for DES.

4.1 Automata

We already mentioned that the term "language" refers to the set of all possible event sequences that a given DES can execute. An *automaton* is a device that is capable of representing a language according to well-defined rules. We shall begin with the case of *deterministic* automata, and subsequently describe extensions to this concept.

Definition 1: A *Deterministic Automaton*, denoted by G, is a six-tuple

$$G = (X, E, f, \Gamma, x_0, X_m)$$

where

- X is the set of *states*
- E is the finite set of *events* associated with the transitions in G

- $f : X \times E \to X$ is the *transition function*: $f(x, e) = y$ means that there is a transition labeled by event e from state x to state y; in general, f is a *partial* function on its domain
- $\Gamma : X \to 2^E$ is the *active event function* (or feasible event function); $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined and it is called the *active event set* (or feasible event set) of G at x. The notation 2^E means the *power set* of E, i.e., the set of all subsets of E.
- x_0 is the *initial* state
- $X_m \subseteq X$ is the set of *marked states*.

The words *state machine* and *generator* (which explains the notation G) are also often used to describe the above object. Moreover, if X is a finite set (which we do not require in the definition above), we call G a *deterministic finite-state automaton*, often abbreviated as DFA. The automaton is said to be *deterministic* because f is a function over $X \times E$. In contrast, the transition structure of a *nondeterministic* automaton is defined by means of a relation over $X \times E \times X$ or, equivalently, a function from $X \times E$ to 2^X. Normally, the word "automaton" refers to a "deterministic automaton".

Regarding the inclusion of Γ in the definition of G, one of the reasons is so we can distinguish between events e that are feasible at x but cause no state transition, that is, $f(x, e) = x$, and events e' that are not feasible at x, that is, $f(x, e')$ is undefined. Finally, regarding the set X_m, its selection depends on the problem of interest; for instance, this set is chosen to designate states which, when entered, indicate that the system has completed some operation or task

The automaton G operates as follows. It starts in the initial state x_0 and upon the occurrence of an event $e \in \Gamma(x_0) \subseteq E$ it will make a transition to state $f(x_0, e) \in X$. This process then continues based on the transitions for which f is defined. Note that an event may occur without changing the state, i.e., it is possible that $f(x, e) = x$. It is also possible that two distinct events occur at a given state causing the exact same transition, i.e., for $a, b \in E$, $f(x, a) = f(x, b) = y$. What is interesting about the latter fact is that we may not be able to distinguish between events a and b by simply observing a transition from state x to state y.

For the sake of convenience, f is always extended from domain $X \times E$ to domain $X \times E^*$, where E^* is the set of *all* finite strings of elements of E, including the empty string (denoted by ε); the * operation is called the *Kleene-closure*. This is accomplished in the following recursive manner:

$$f(x, \varepsilon) := x$$
$$f(x, se) := f(f(x, s), e) \text{ for } s \in E^* \text{ and } e \in E.$$

The DFA setting is the basis of much of the supervisory control theory that has been developed to date and which is discussed in "*Supervisory Control of Discrete Event Systems*". Operations on automata that allow the construction of larger system models of interest from simpler component models are presented in "*Modeling of Discrete Event Systems*".

The automaton model above is also referred to as a *Generalized Semi-Markov Scheme* (abbreviated as GSMS) in the literature of stochastic processes. A GSMS is viewed as the basis for extending automata to incorporate an event timing structure and ultimately leads to *stochastic timed automata*, discussed next.

Let us start by considering an automaton G as defined above with a few minor changes: we allow for generally countable sets X and E, and we leave out of the definition any consideration for marked states. Thus, we begin with an automaton model (X, E, f, Γ, x_0). As it stands, this model is based on the premise that a given event sequence $\{e_1, e_2, \ldots\}$ is provided, so that, starting at state x_0, we can generate a state sequence $\{x_0, f(x_0, e_1), f(f(x_0, e_1), e_2), \ldots\}$. Note that if $\Gamma(x_0)$ includes several events, the fact that $e_1 \in \Gamma(x_0)$ is the particular event that occurs first is part of the input information necessary to operate the automaton. We extend our modeling setting to *timed* automata by incorporating a *clock structure* associated with the event set E which now becomes the input from which a specific event sequence can be deduced.

Definition 2: The *Clock Structure* (or *Timing Structure*) associated with an event set E is a set $\mathbf{V} = \{\mathbf{v}_i : i \in E\}$ of clock (or lifetime) sequences

$$\mathbf{v}_i = \{v_{i,1}, v_{i,2}, \ldots\}, \quad i \in E, \quad v_{i,k} \in \mathbb{R}^+, \quad k = 1, 2, \ldots$$

We can then provide the following definition:

Definition 3: A *Timed Automaton* is a six-tuple

$$(X, E, f, \Gamma, x_0, \mathbf{V})$$

where $\mathbf{V} = \{\mathbf{v}_i : i \in E\}$ is a clock structure and (X, E, f, Γ, x_0) is an automaton. The automaton generates a state sequence

$$x' = f(x, e') \tag{4}$$

driven by an event sequence $\{e_1, e_2, \ldots\}$ generated through

$$e' = \arg \max_{i \in \Gamma(x)} \{y_i\} \tag{5}$$

with the clock values y_i, $i \in E$, defined by

$$y_i' = \begin{cases} y_i - y^* & \text{if } i \neq e' \text{ and } i \in \Gamma(x) \\ v_{i,N_i+1} & \text{if } i = e' \text{ or } i \notin \Gamma(x) \end{cases} \qquad i \in \Gamma(x') \tag{6}$$

where the *interevent time* y^* is defined as

$$y^* = \min_{i \in \Gamma(x)} \{y_i\} \tag{7}$$

and the *event scores* N_i, $i \in E$, are defined by

$$N_i' = \begin{cases} N_i + 1 & \text{if } i = e' \text{ and } i \notin \Gamma(x) \\ N_i & \text{otherwise} \end{cases} \qquad i \in \Gamma(x') \tag{8}$$

In addition, initial conditions are: $y_i = v_{i,1}$ and $N_i = 1$ for all $i \in \Gamma(x_0)$. If $i \notin \Gamma(x_0)$, then y_i is undefined and $N_i = 0$.

Comparing (4) to the state equation (1) for time-driven systems, we see that the former can be viewed as the event-driven analog of the latter. However, the simplicity of (4) is deceptive: unless an event sequence is given (as in the case of Definition 1), determining the *triggering* event e' which is required to obtain the next state x' involves the combination of (5)-(8). Therefore, the analog of (1) as a "canonical" state equation for DES requires all five equations (5)-(8).

In Definition 3, the clock structure $\mathbf{V}$ is assumed to be fully specified in a deterministic sense. Let us now assume that the clock sequences $\mathbf{v}_i$, $i \in E$, are specified only as stochastic sequences. This means that we no longer have at our disposal real numbers $\{v_{i,1}, v_{i,2}, \ldots\}$ for each event i, but rather a distribution function, denoted by G_i, which describes the *random* clock sequence $\{V_{i,k}\} = \{V_{i,1}, V_{i,2}, \ldots\}$.

Definition 4: The *Stochastic Clock Structure* (or *Stochastic Timing Structure*) associated with an event set E is a set of distribution functions $G = \{G_i : i \in E\}$ characterizing the stochastic clock sequences

$$\{V_{i,k}\} = \{V_{i,1}, V_{i,2}, \ldots\}, \quad i \in E, \quad V_{i,k} \in \mathbb{R}^+, \quad k = 1, 2, \ldots$$

Most of the DES analysis based on stochastic clock structures asumes that each clock sequence consists of random variables which are independent and identically distributed (iid) and that all clock sequences are mutually independent. Thus, each $\{V_{i,k}\}$ is completely characterized by a distribution function $G_i(t) = P[V_i \leq t]$. There are, however, several ways in which a clock structure can be extended to include situations where elements of a sequence $\{V_{i,k}\}$ are correlated or two clock sequences are dependent on each other.

We can extend the definition of a timed automaton by viewing the state, event, and all event scores and clock values as random variables denoted respectively by X, E, N_i, and Y_i, $i \in \mathcal{E}$, where we use $\mathcal{E}$ to denote the event set and distinguish it from some event E which takes values $i \in \mathcal{E}$. Similarly, we use $\mathcal{X}$ to denote the state space and distinguish it from some state X which takes values $x \in \mathcal{X}$.

Definition 5: A *Stochastic Timed Automaton* is a six-tuple

$$(\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$$

where $\mathcal{X}$ is a countable *state space*; $\mathcal{E}$ is a countable *event set*; $\Gamma(x)$ is the *active event set* (or feasible event set); $p(x'; x, e')$ is a *state transition probability* defined for all $x, x' \in \mathcal{X}$, $e' \in \mathcal{E}$ and such that $p(x'; x, e') = 0$ for all $e' \notin \Gamma(x)$; p_0 is the probablity mass function $P[X_0 = x]$, $x \in \mathcal{X}$, of the initial state X_0; and G is a *stochastic clock structure*.

The automaton generates a stochastic state sequence $\{X_0, X_1, \ldots\}$ through a transition mechanism (based on observations $X = x$, $E' = e'$):

$$X' = x' \text{ with probability } p(x'; x, e') \tag{9}$$

and it is driven by a stochastic event sequence $\{E_1, E_2, \ldots\}$ generated through

$$E' = \arg \max_{i \in \Gamma(X)} \{Y_i\} \tag{10}$$

with the stochastic clock values Y_i, $i \in \mathcal{E}$, defined by

$$Y_i' = \begin{cases} Y_i - Y^* & \text{if } i \neq E' \text{ and } i \in \Gamma(X) \\ V_{i,N_i+1} & \text{if } i = E' \text{ or } i \notin \Gamma(X) \end{cases} \qquad i \in \Gamma(X') \tag{11}$$

where the *interevent time* Y^* is defined as

$$Y^* = \min_{i \in \Gamma(X)} \{Y_i\} \tag{12}$$

and the *event scores* N_i, $i \in \mathcal{E}$, are defined by

$$N_i' = \begin{cases} N_i + 1 & \text{if } i = E' \text{ and } i \notin \Gamma(X) \\ N_i & \text{otherwise} \end{cases} \qquad i \in \Gamma(X') \tag{13}$$

and $\{V_{i,k}\} \sim G_i$ (the notation $\sim$ denotes "with distribution"). In addition, initial conditions are: $X_0 \sim p_0(x)$, $Y_i = V_{i,1}$ and $N_i = 1$ if $i \in \Gamma(X_0)$. If $i \notin \Gamma(X_0)$, then Y_i is undefined and $N_i = 0$.

A simple interpretation of this elaborate definition is as follows. Given that the system is at some state X, the next event E' is the one with the smallest clock value among all feasible events $i \in \Gamma(X)$. The corresponding clock value, Y^*, is the interevent time between the occurrence of E and E', and it provides the amount by which time, T, moves forward:

$$T' = T + Y^*$$

Clock values for all events that remain active in state X' are decremented by Y^*, except for the triggering event E' and all newly activated events, which are assigned a new lifetime V_{i,N_i+1}. Event scores are incremented whenever a new lifetime is assigned to them. It is important to note that the "system clock" T is fully controlled by the occurrence of events, which cause it to move forward; if no event occurs, the system remains at the last state observed.

It is conceivable for two events to occur at the same time, in which case we need a priority scheme in order to overcome a possible ambiguity in the selection of the triggering event in (10). This is usually accomplished through a priority scheme over all events in $\mathcal{E}$. In practice, it is common to expect that every G_i in the clock structure is absolutely continuous over $[0, \infty)$ (so that its density function exists) and has a finite mean. This implies that two events can occur at exactly the same time only with probability 0.

A stochastic process $\{X(t)\}$ with state space $\mathcal{X}$ which is generated by a stochastic timed automaton $(\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$ is referred to as a *Generalized Semi-Markov Process* (GSMP). This process is used as the basis of much of the sample path analysis methods further discussed in Article 6.43.27.3.

4.1.1 Queueing Systems

In order to illustrate the use of automata in modeling DES, we consider here a particularly useful class of DES, known as *queueing systems*. These arise in numerous application domains from computer networks and manufacturing to logistics and transportation. The term "queueing" refers to a fact of life intrinsic in many of the most common systems we design and build: in order to use certain resources, we have to wait. For example, to use the resource "CPU", various "tasks" wait somewhere in a computer until they are given access to the CPU through potentially complex mechanisms. There are three basic elements that comprise a queueing system: (i) The *entities* that do the waiting in their quest for resources, generically referred to as *customers*, (ii) The *resources* for which the waiting is done, generically referred to as *servers*, and (iii) The space where the waiting is done, which we shall call a *queue*. Examples of customers are messages transmitted over some communication medium and production parts in a manufacturing process. Examples of servers are communication channels responsible for the transmission of messages and various types of machines used in manufacturing. Examples of visible queues are found in bank lobbies, bus stops, or warehouses. However, queues are also ever-present in communication networks or computer systems, where less tangible forms of customers, like telephone calls or executable tasks, are also allocated to waiting areas. In some cases, queues are also referred to as *buffers*.

The study of queueing systems is motivated by the simple fact that resources are not unlimited; if they were, no waiting would ever occur. This fact gives rise to obvious problems of resource allocation and related tradeoffs so that (a) customer needs are adequately satisfied, (b) resource access is provided in fair and efficient ways among different customers, and (c) the cost of designing and operating the system is maintained at acceptable levels.

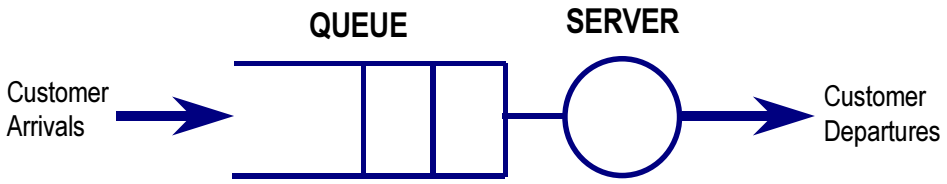

Figure 2: A simple queueing system

Graphically, a simple queueing system is represented as shown in Fig. 2. A circle represents a server, and an open box represents a queue preceding this server. The slots in the queue are meant to indicate waiting customers. Customers are thought of as *arriving* at the queue, and *departing* from the server. It is also assumed that the process of serving customers normally takes a strictly positive amount of time (otherwise there would be no waiting). Thus, a server may be thought of as a "delay block" which holds a customer for some amount of service time. The *capacity* of a queue is the maximum number of customers that can be accommodated in the actual queueing space. In many models, it is assumed that the capacity is "infinite", in the sense that the queueing space is so large that the likelihood of it ever being full is negligible.

The *queueing discipline* refers to the rule according to which the next customer to be served is selected from the queue. The simplest such rule is the First-In-First-Out (FIFO): customers are served in the precise order in which they arrive.

Viewed as a DES, the queueing system of Fig. 2 has an event set $\mathcal{E} = \{a, d\}$, where a denotes an "arrival" event, and d denotes a "departure" event. A natural state variable is the number of customers in queue, which we shall call *queue length*. By convention, the queue length at time t is allowed to include a customer in service at time t (i.e., $x(t) = 1$ means that a single customer is currently present and located at the server). Thus, the state space is the set of non-negative integers, $\mathcal{X} = \{0, 1, 2, \ldots\}$.

An automaton, as specified in Definition 1, can be used to model a simple queueing system by defining, in addition to the state space $\mathcal{X}$ and event set $\mathcal{E}$ above, the active event sets

$$\Gamma(x) = \{a, d\} \text{ for all } x > 0 \text{ and } \Gamma(0) = \{a\}$$

corresponding to the fact that no departure event d is feasible when the system is empty ($x = 0$). Finally, the state transition functions are given by

$$\begin{aligned} f(x, a) &= x + 1 \quad \text{for all } x \geq 0 \\ f(x, d) &= x - 1 \quad \text{if } x > 0 \end{aligned}$$

The initial state x_0 is chosen to be the initial number of customers of the system. A state transition diagram for this system is shown in Fig. 3. Note that the state space in this model is infinite, but countable; also, we have left X_m unspecified in the standard definition of a DFA.

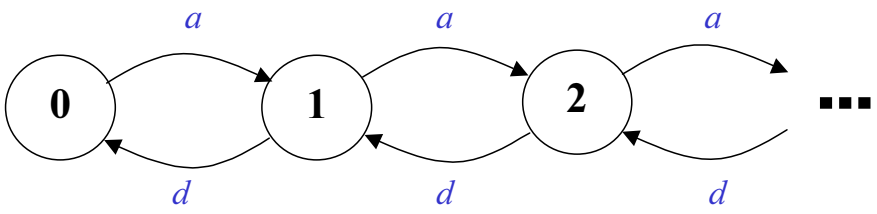

Figure 3: State transition diagram of a simple queueing system

If the abstraction level at which one is interested in studying this system is fully captured by the state transition diagram in Fig. 3, then there is no need for a more detailed model involving timing information. If, for instance, we are given an event sequence $\{e_1, e_2, \ldots, e_n\}$ and a question of interest is "will the system state ever reach a given level L during operation under this sequence?", then it is easy to see that such questions can be easily answered by inspection of the diagram in Fig. 3. As an example, if $\{a, a, a, d, a\}$ is given, $x_0 = 0$, and $L = 2$, we can immediately see that the state after the first two events is necessarily $x = 2$. On the other hand, if d events are not observable or can simply not be deterministically ordered in a full event sequence, then the same question cannot be answered without further information

which involves the timing of d events, such as information about the stochastic clock sequence $\{V_{d,k}\} = \{V_{d,1}, V_{d,2}, \ldots\}$ (see Definition 4).

Thus, if the abstraction level of interest involves timing issues in a queueing system, we turn our attention to a timed automaton model, as in Definition 3. If the arrival and departure processes cannot be deterministically specified, then a stochastic timed automaton model, as in Definition 5, is more suitable. In this case, we need to enrich the previous model by adding a stochastic clock structure, i.e., distributions G_a and G_d to characterize the arrival process and the departure process: $\{V_{a,k}\} \sim G_a$ and $\{V_{d,k}\} \sim G_d$. The state transition probabilities in this model are simple:

$$\begin{aligned} p(x+1; x, a) &= 1 \quad \text{for all } x = 0, 1, \ldots \\ p(x-1; x, d) &= 1 \quad \text{for all } x > 0 \end{aligned}$$

4.2 Petri Nets

An alternative modeling formalism for DES is provided by *Petri nets*, originating in the work of C. A. Petri in the early 1960s. Like an automaton, a Petri net is a device that manipulates events according to certain rules. One of its features is that it includes explicit conditions under which an event can be enabled. This representation is conveniently described graphically, at least for small systems, resulting in *Petri net graphs*. Petri net graphs are intuitive and capture a lot of structural information about the system. An automaton can always be represented as a Petri net; on the other hand, not all Petri nets can be represented as *finite-state* automata. Another motivation for considering Petri net models of DES is the body of analysis techniques that have been developed for studying them.

The process of defining a Petri net involves two steps. First, we define the Petri net graph, also called *Petri net structure*. Then we adjoin to this graph an initial state, a set of marked states, and a transition labeling function, resulting in the complete Petri net model, its associated dynamics, and the languages that it generates and marks.

Petri net graph. A Petri net graph has two types of nodes, *places* and *transitions*, and arcs connecting these. It is a *bipartite graph* in the sense that arcs cannot directly connect nodes of the same type; rather, arcs connect place nodes to transition nodes and transition nodes to place nodes. Events are associated with transition nodes. In order for a transition to occur, several conditions may have to be satisfied. Information related to these conditions is contained in place nodes. Some such places are viewed as the "input" to a transition; they are associated with the conditions required for this transition to occur. Other places are viewed as the output of a transition; they are associated with conditions that are affected by the occurrence of this transition. The precise definition of Petri net graph is as follows.

Definition 6. A *Petri net graph* is a weighted bipartite graph (P, T, A, w) where

- P is the finite set of *places* (one type of node in the graph)

- T is the finite set of *transitions* (the other type of node in the graph)
- $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs from places to transitions and from transitions to places in the graph
- $w : A \to \{1, 2, 3, \ldots\}$ is the *weight function* on the arcs.

We assume that (P, T, A, w) has no isolated places or transitions. When drawing Petri net graphs, we need to differentiate between the two types of nodes, places and transitions. The convention is to use circles to represent places and bars to represent transitions.

Let the set of places be represented by $P = \{p_1, p_2, \ldots, p_n\}$, and the set of transitions be represented by $T = \{t_1, t_2, \ldots, t_m\}$. A typical arc is of the form (p_i, t_j) or (t_j, p_i), and the weight related to an arc is a positive integer. In describing a Petri net graph, it is convenient to use $I(t_j)$ to represent the set of input places to transition t_j. Similarly, $O(t_j)$ represents the set of output places from transition t_j. Thus, we have

$$I(t_j) = \{p_i \in P : (p_i, t_j) \in A\}, \qquad O(t_j) = \{p_i \in P : (t_j, p_i) \in A\} \ .$$

Similar notation can be used to describe input and output transitions for a given place p_i: $I(p_i)$ and $O(p_i)$.

Petri net dynamics. *Tokens* are assigned to places in a Petri net graph in order to indicate the fact that the condition described by that place is satisfied. The way in which tokens are assigned to a Petri net graph defines a *marking*. Formally, a *marking* x of a Petri net graph (P, T, A, w) is a function $x : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. Marking x defines row vector $\mathbf{x} = [x(p_1), x(p_2), \ldots, x(p_n)]$, where n is the number of places in the Petri net. The ith entry of this vector indicates the (non-negative integer) number of tokens in place p_i, $x(p_i) \in \mathbb{N}$. In Petri net graphs, a token is indicated by a dark dot positioned in the appropriate place.

The *state* of a Petri net is defined to be its marking row vector $\mathbf{x} = [x(p_1), x(p_2), \ldots, x(p_n)]$. Note that the number of tokens assigned to a place is an arbitrary non-negative integer, not necessarily bounded. It follows that the number of states we can have is, in general, infinite. Thus, the state space X of a Petri net with n places is defined by all n-dimensional vectors whose entries are non-negative integers, i.e., $X = \mathbb{N}^n$. While the term "marking" is more common than "state" in the Petri net literature, the term state is consistent with the role of state in system dynamics. Moreover, the term state avoids the potential confusion between marking in Petri net graphs and marking in the sense of *marked states* in automata.

The state transition mechanism of a Petri net is captured by the structure of its graph and by "moving" tokens from one place to another. A transition $t_j \in T$ in a Petri net is said to be *enabled* if

$$x(p_i) \geq w(p_i, t_j) \ \text{ for all } \ p_i \in I(t_j) \ .$$

In words, transition t_j in the Petri net is enabled when the number of tokens in p_i is at least as large as the weight of the arc connecting p_i to t_j, for all places p_i that are input to transition t_j.

When a transition is enabled, it can occur or *fire* (the term "firing" is standard in the Petri net literature). The *state transition function* of a Petri net is defined through the change in the state of the Petri net due to the firing of an enabled transition. The state transition function, $f : \mathbb{N}^n \times T \to \mathbb{N}^n$, of Petri net (P, T, A, w, x) is defined for transition $t_j \in T$ if and only if

$$x(p_i) \geq w(p_i, t_j) \quad \text{for all } p_i \in I(t_j) \ . \tag{14}$$

If $f(\mathbf{x}, t_j)$ is defined, then we set $\mathbf{x}' = f(\mathbf{x}, t_j)$ where

$$x'(p_i) = x(p_i) - w(p_i, t_j) + w(t_j, p_i), \quad i = 1, \ldots, n \ . \tag{15}$$

Condition (14) ensures that the state transition function is defined only for transitions that are enabled; an "enabled transition" is therefore equivalent to a "feasible event" in an automaton. But whereas in automata the state transition function was quite arbitrary, here the state transition function is based on the structure of the Petri net. Thus, the next state defined by equation (15) explicitly depends on the input and output places of a transition and on the weights of the arcs connecting these places to the transition. According to (15), if p_i is an input place of t_j, it loses as many tokens as the weight of the arc from p_i to t_j; if it is an output place of t_j, it gains as many tokens as the weight of the arc from t_j to p_i. Clearly, it is possible that p_i is both an input and output place of t_j, in which case equation (15) removes $w(p_i, t_j)$ tokens from p_i, and then immediately places $w(t_j, p_i)$ new tokens back in it.

In general, it is entirely possible that after several transition firings, the resulting state is $\mathbf{x} = [0, \ldots, 0]$, or that the number of tokens in one or more places grows arbitrarily large after an arbitrarily large number of transition firings. The latter phenomenon is a key difference with automata, where finite-state automata have only a finite number of states, by definition. In contrast, a finite Petri net graph may result in a Petri net with an unbounded number of states.

In order to illustrate the use of Petri nets and contrast them to automata, let us describe how the simple queueing system of Fig. 2 can be modeled through a Petri net. We begin by considering three events (transitions) driving the system: a represents a customer arrival, s represents the start of customer service, and c represents service completion and departure from the system. Thus, we define the transition set $T = \{a, s, c\}$. Transition a is spontaneous and requires no conditions (input places). On the other hand, transition s depends on two conditions: the presence of customers in the queue, and the server being idle. We represent these conditions through two input places for this transition, place Q (queue) and place I (idle server). Finally, transition c requires that the server be busy, so we introduce an input place B (busy server) for it. Thus, our place set is $P = \{Q, I, B\}$.

The complete Petri net graph for the queueing system of Fig. 2 is shown in Fig. 4. On the left side, no tokens are placed in Q, indicating that the queue is empty, and a token is placed in I, indicating that the server is idle. This defines the initial state $\mathbf{x}_0 = [0, 1, 0]$. Since transition a is always enabled, we can generate several possible sample paths. As an example, on the right side of Fig. 4 we show state $[2, 0, 1]$ resulting from the transition firing sequence $\{a, s, a, a, c, s, a\}$.

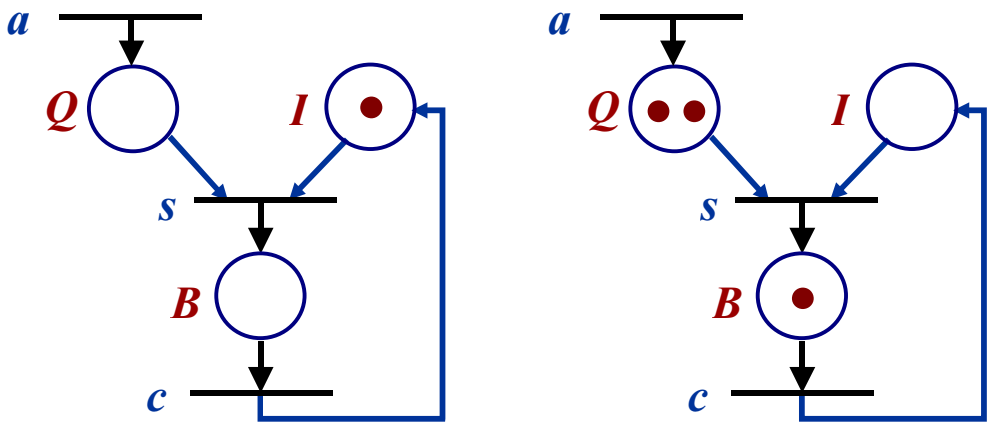

Figure 4: Petri net model of a simple queueing system

This state corresponds to two customers waiting in queue, while a third is in service (the first arrival in the sequence has already departed after transition c).

Note that a c transition always enables an s transition as long as there is a token in Q. If we assume that the enabled s transition then immediately fires, this is equivalent to the automaton model seen in Fig. 3 using only two events (arrivals a and departures d), but specifying a feasible event set $\Gamma(0) = \{a\}$.

Similar to timed automata, we can define timed Petri nets by introducing a clock structure, except that now a clock sequence $\mathbf{v}_j$ is associated with a transition t_j. A positive real number, $v_{j,k}$, assigned to t_j has the following meaning: when transition t_j is enabled for the kth time, it does not fire immediately, but incurs a firing delay given by $v_{j,k}$; during this delay, tokens are kept in the input places of t_j. Not all transitions are required to have firing delays. Some transitions may always fire as soon as they are enabled. Thus, we partition T into subsets T_0 and T_D, such that $T = T_0 \cup T_D$. T_0 is the set of transitions always incurring zero firing delay, and T_D is the set of transitions that generally incur some firing delay. The latter are called *timed transitions*.

Definition 7: The *Clock Structure* (or *Timing Structure*) associated with a set of timed transitions $T_D \subseteq T$ of a marked Petri net (P, T, A, w, x) is a set $\mathbf{V} = \{\mathbf{v}_j : t_j \in T_D\}$ of clock (or lifetime) sequences

$$\mathbf{v}_j = \{v_{j,1}, v_{j,2}, \ldots\},\ t_j \in T_D,\quad v_{j,k} \in \mathbb{R}^+,\quad k = 1, 2, \ldots$$

Graphically, transitions with no firing delay are still represented by bars, whereas timed transitions are represented by rectangles. The clock sequence associated with a timed transition is normally written next to the rectangle.

Definition 8: A *Timed Petri Net* is a six-tuple

$$(P, T, A, w, x, \mathbf{V})$$

where (P, T, A, w, x) is a marked Petri net and $\mathbf{V} = \{\mathbf{v}_j : t_j \in T_D\}$ is a clock structure.

Returning to the Petri net model of a queueing system, shown in Fig. 4, the timed transition set is $T_D = \{a, d\}$, corresponding to customer arrivals and to departures from the server. Transition s, on the other hand, incurs no firing delay, since service starts as soon as the server is idle and a customer is in the queue. The clock structure for this model consists of $\mathbf{v}_a = \{v_{a,1}, v_{a,2}, \ldots\}$ and $\mathbf{v}_d = \{v_{d,1}, v_{d,2}, \ldots\}$, which is identical to the clock structure for the timed automaton model presented earlier. The corresponding Petri net graph is shown in Fig. 5.

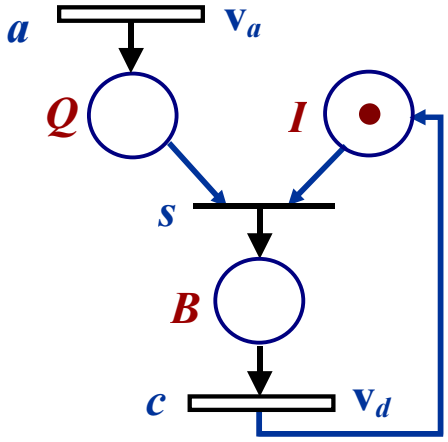

Figure 5: Timed Petri net model of a simple queueing system

It is now possible to derive simple state equations that characterize the event timing dynamics of this queueing system. Let us define a_k to be the kth arrival time of a customer (kth time when transition a fires), d_k to be the kth departure time (kth time when transition c fires), and s_k to be the kth service start time (kth time when transition s fires). Similarly, let $\pi_{Q,k}$ be the time when Q receives its kth token, $\pi_{I,k}$ the time when I receives its kth token, and $\pi_{B,k}$ the time when B receives its kth token. Assume an initial condition $\pi_{I,1} = 0$. By inspection of Fig. 5, we have the following relationships:

$$a_k = a_{k-1} + v_{a,k}, \qquad k = 1, 2, \ldots, \qquad a_0 = 0 \tag{16}$$

$$s_k = \max\{\pi_{Q,k}, \pi_{I,k}\}, \qquad k = 1, 2, \ldots \tag{17}$$

$$d_k = \pi_{B,k} + v_{d,k}, \qquad k = 1, 2, \ldots \tag{18}$$

and

$$\pi_{Q,k} = a_k, \qquad k = 1, 2, \ldots \tag{19}$$

$$\pi_{I,k} = d_{k-1}, \quad k = 2, 3, \ldots, \quad \pi_{I,1} = 0 \tag{20}$$

$$\pi_{B,k} = s_k, \qquad k = 1, 2, \ldots \tag{21}$$

Combining (16) through (21) to eliminate $\pi_{Q,k}$, $\pi_{I,k}$, and $\pi_{B,k}$, we get

$$s_k = \max\{a_k, d_{k-1}\}, \quad k = 1, 2, \ldots, \quad d_0 = 0 \tag{22}$$

$$d_k = s_k + v_{d,k}, \qquad k = 1, 2, \ldots \tag{23}$$

We can further eliminate s_k from (23) to obtain the following fundamental relationship for this simple queueing system:

$$d_k = \max\{a_k, d_{k-1}\} + v_{d,k}, \quad k = 1, 2, \ldots, \quad d_0 = 0 \tag{24}$$

This is a simple recursive relationship characterizing customer departure times. It captures the fact that the kth departure occurs $v_{d,k}$ time units after the $(k-1)$th departure, except when $a_k > d_{k-1}$. The latter case occurs when the departure at d_{k-1} leaves the queue empty; the server must then await the next arrival at time a_k, and generate the next departure at time $a_k + v_{d,k}$.

Rewriting (16) and (24) for $k = 2, 3, \ldots$ as follows:

$$a_k = a_{k-1} + v_{a,k}, \quad a_0 = 0 \tag{25}$$

$$d_k = \max\{a_{k-1} + v_{a,k}, d_{k-1}\} + v_{d,k}, \quad d_0 = 0 \tag{26}$$

we obtain a state space model reminiscent of (1) for time-driven systems. In this case, the queueing system model is driven by the clock sequences $\mathbf{v}_a$ and $\mathbf{v}_d$, viewed as input vectors, and its output consists of the arrival and departure time sequences $\{a_1, a_2, \ldots\}$ and $\{d_1, d_2, \ldots\}$ generated through the state equations (25) and (26). Clearly, $\mathbf{v}_a$ and $\mathbf{v}_d$ may also be replaced by stochastic clock sequences, leading to a *stochastic* timed Petri net model.

4.3 Dioid Algebras

Another modeling framework we will briefly describe in what follows is based on developing an algebra using two operations: $\min\{a, b\}$ (or $\max\{a, b\}$) for any real numbers a and b, and addition $(a+b)$. The motivation comes from the observation that the operations "min" and "+" are the only ones required to develop the timed automaton model in Definition 3. Similarly, the operations "max" and "+" are the only ones used in developing the timed Petri net models described in the previous section. The term "dioid" (meaning "two") refers to the fact that this algebra is based on two operations. The operations are formally named *addition* and *multiplication* and denoted by $\oplus$ and $\otimes$ respectively. However, their actual meaning (in terms of regular algebra) is different. For any two real numbers a and b, we define

$$Addition: \qquad a \oplus b \equiv \max\{a, b\} \tag{27}$$

$$Multiplication: \qquad a \otimes b \equiv a + b \tag{28}$$

This dioid algebra is also called a $(\max, +)$ algebra. The motivation for pursuing this modeling approach goes beyond the observation that "max" and "+" are the only operations of apparent importance in the study of DES. If we consider a standard linear discrete-time system, its state equation is of the form

$$\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k)$$

which involves (regular) multiplication $(\times)$ and addition $(+)$. It turns out that we can use a $(\max, +)$ algebra with DES, replacing the $(+, \times)$ algebra of conventional time-driven systems, and come up with a representation similar to the one above, thus paralleling to a considerable extent the analysis of classical time-driven linear systems.

We will illustrate the point above by considering, once again, the simple queueing system of Fig. 2. Our starting point is the set of equations (25)-(26) derived using a timed Petri net model. For simplicity, we will assume that

$$v_{a,k} = C_a, \qquad v_{d,k} = C_d \qquad \text{for all } k = 1, 2, \ldots$$

where C_a and C_d are given constants. Note that C_a represents a constant interarrival time, and C_d represents a constant service time for all customers. We will also assume that $C_a > C_d$, which is reasonable, since otherwise customers arrive faster than the service rate. We now rewrite (25)-(26) as follows:

$$\begin{aligned} a_{k+1} &= a_k + C_a, & a_1 &= C_a \\ d_k &= \max\{a_k, d_{k-1}\} + C_d, & d_0 &= 0 \end{aligned}$$

Using the $(\max, +)$ algebra, these relationships can also be expressed as

$$\begin{aligned} a_{k+1} &= (a_k \otimes C_a) \oplus (d_{k-1} \otimes -L) \\ d_k &= (a_k \otimes C_d) \oplus (d_{k-1} \otimes C_d) \end{aligned}$$

where $-L$ is any sufficiently small negative number so that $\max\{a_k + C_a, d_{k-1} - L\} = a_k + C_a$. In matrix notation, we have

$$\begin{bmatrix} a_{k+1} \\ d_k \end{bmatrix} = \begin{bmatrix} C_a & -L \\ C_d & C_d \end{bmatrix} \begin{bmatrix} a_k \\ d_{k-1} \end{bmatrix} \tag{29}$$

Defining

$$\mathbf{x}_k = \begin{bmatrix} a_{k+1} \\ d_k \end{bmatrix}, \qquad \mathbf{A} = \begin{bmatrix} C_a & -L \\ C_d & C_d \end{bmatrix}$$

we get

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k, \qquad \mathbf{x}_0 = \begin{bmatrix} C_a \\ 0 \end{bmatrix} \tag{30}$$

which in fact looks like a standard linear system model, except that the addition and multiplication operations are in the $(\max, +)$ algebra. We should emphasize, however, that while conceptually this offers an attractive representation of the queueing system's event timing dynamics, from a computational starndpoint one still has to confront the complexity of performing the "max" operation when numerical information is ultimately needed to analyze the system or to design controllers for its proper operation.

5 Control and Optimization of Discrete Event Systems

The various control and optimization methodologies developed to date for DES depend on the modeling level appropriate for the problem of interest.

Logical behavior. Issues such as ordering events according to some specification or ensuring the reachability of a particular state are normally addressed through the use of automata (see Definition 1) and Petri nets (see Definition 6). Supervisory control theory provides a systematic framework for formulating and solving problems of this type. In its simplest form, a *supervisor* is responsible for enabling or disabling a subset of events that are controllable, so as to achieve a desired behavior. The supervisor's actions follow the occurrence of those events that are in fact observable (generally, a subset of the event set of the DES). In this manner, a supervisor may be viewed as a feedback controller. Similar to classical control theory, two central concepts here are those of *controllability* and *observability*, which are further discussed in "*Supervisory Control of Discrete Event Systems*". The synthesis of supervisors is a particularly challenging problem, largely due to the inherent computational complexity resulting from the combinatorial growth of typical state spaces of DES. Some special classes of supervisor synthesis problems and solution approaches for them are discussed in "*Supervisory Control of Discrete Event Systems*".

Event timing. When timing issues are introduced, timed automata (see Definition 3) and timed Petri nets (see Definition 8) are invoked for modeling purposes. Supervisory control in this case becomes significantly more complicated. An important class of problems, however, does not involve the ordering of individual events, but rather the requirement that selected events occur within a given "time window" or with some desired periodic characteristics. Models based on the algebraic structure of timed Petri nets or the $(\max, +)$ algebra discussed in Section 4.3 provide convenient settings for formulating and solving such problems.

Performance analysis. As in classical control theory, one can define a performance (or cost) function intended to measure how "close" one can get to a desired behavior or simply as a convenient means for quantifying system behavior. This approach is particularly crucial in the study of stochastic DES, where the design of a system is often based on meeting specifications defined through appropriate performance metrics, such as the expected response time of tasks in a computer system or the throughput of a manufacturing process. Because of the complexity of DES dynamics, analytical expressions for such performance metrics in terms of controllable variables are seldom available. This has motivated the use of simulation and, more generally, the study of DES sample paths; these have proven to contain a surprising wealth of information for control purposes. The theory of *Perturbation Analysis* (PA) has provided a systematic way of estimating performance sensitivities with respect to system parameters for important classes of DES, including the queueing systems discussed in Section 4.1.1. What is noteworthy in this approach is that these sensitivity estimates can be extracted from a *single* sample path of a DES, resulting in a variety of efficient algorithms which can often be implemented on line. Perturbation analysis and related approaches are further discussed in "*Sample Path Analysis of Discrete Event Dynamic Systems*".

Simulation. Because of the aforementioned complexity of DES dynamics, simulation becomes

an essential part of DES performance analysis. Discrete-event simulation can be defined as a systematic way of generating sample paths of a DES by means of the timed automaton in Definition 3 or its stochastic counterpart in Definition 5. In the latter case, a computer pseudo-random number generator is responsible for the task of providing the elements of the clock sequences in the model. It should also be clear that the same process can be carried out using a Petri net model or one based on the dioid algebra setting.

Optimization. Optimization problems can be formulated in the context of both untimed and timed models of DES, as discussed in "*Supervisory Control of Discrete Event Systems*" and "*Sample Path Analysis of Discrete Event Dynamic Systems*", respectively. Moreover, such problems can be formulated both in a deterministic and a stochastic setting. In the latter case, the ability to efficiently estimate performance sensitivities with respect to controllable system parameters provides a powerful tool for stochastic gradient-based optimization (when one can define derivatives) or discrete optimization methods (in the frequent case where the controllable parameters of interest are discrete, as in turning a piece of equipment "on" or "off" or selecting the (integer) number of resources needed to execute certain tasks).

Hybrid Systems. It is worth mentioning that the combination of time-driven and event-driven dynamics gives rise to a *hybrid* system. Control and optimization methodologies for such systems are particularly challenging as they need to capture both aspects, often through appropriate decomposition approaches.

Bibliography

Baccelli, F., Cohen, G., Olsder, G.J., Quadrat, J.-P. (1992). *Synchronization and Linearity: An Algebra for Discrete Event Systems*, Wiley, Chichester, 1992. [Good reference for the dioid algebra approach to discrete event systems]

Cassandras, C.G., and Lafortune, S. (1999). *Introduction to Discrete Event Systems*, Kluwer Academic Publishers. [Textbook with a comprehensive coverage of all aspects of discrete event systems, including modeling frameworks, supervisory control, discrete event simulation, and sample paths analysis methods]

Glasserman, P., Yao, D.D. (1991). *Monotone Structure in Discrete-Event Systems*, Wiley. [This book emphasizes structural properties of discrete event systems that lend themselves to methods for performance analysis, control, and optimization]

Ho, Y.C. (Ed.). (1991). *Discrete Event Dynamic Systems: Analyzing Complexity and Performance in the Modern World*, IEEE Press. [A collection of papers reflecting different approaches to the modeling and analysis of discrete event systems, summarizing the first decade of research accomplishments in the field]

Ho, Y.C., Cao, X. (1991). *Perturbation Analysis of Discrete Event Dynamic Systems*, Kluwer Academic Publishers. [This book covers the most important class of sample-path-based methods for the performance analysis, control, and optimization of discrete event systems]

Peterson, J.L. (1981). *Petri Net Theory and the Modeling of Systems*, Prentice Hall. [Good textbook reference for Petri nets]