Discrete Event Systems
- Introduction -

Lothar Thiele*
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zurich

Contents

1 Motivation

1.1 General Remarks

The reasons for the increasing interest in models and methods for discrete event systems have been well described by Christos Cassandras [1]:

> Over the past few decades, the rapid evolution of computer technology has brought about the proliferation of new dynamic systems, mostly ”man–made” and highly complex. Examples are all around us: computer networks; automated manufacturing systems; air traffic control systems; and highly integrated command, control, communication, and information (C^3I) systems. All activity in these systems is due to asynchronous occurrences of discrete events, some controlled (such as hitting a keyboard

key) and some not (such as a spontaneous equipment failure). This feature lends itself to the term discrete event system.

The mathematical arsenal at our disposal today (primarily, differential and difference equations) was developed over several centuries to model and analyze the time–driven processes one usually encounters in nature. The process of adapting this arsenal and developing a new one suitable for event–driven systems is only a few years old: The challenge is to develop new modeling frameworks, design tools, testing techniques, and systematic control procedures for this new generation of highly complex systems. New paradigms, based on combining mathematical techniques with processing experimental data, are slowly emerging. Moreover, the role of the computer itself as a tool for system design, analysis, and control is becoming critical in the development of these new paradigms.

The capabilities that discrete event systems have, or are intended to have, are extremely exciting. Their complexity, on the other hand, is overwhelming. Yet, the availability of computer–aided tools makes it possible and often cost–effective simply to build new systems, sometimes based on a "seat–of–the–pants" or trial–and–error approach alone, before we can fully figure out their properties, limitations, or overall implications. New models and methodologies are needed not only to enhance design procedures, but also to prevent failures (which can indeed be catastrophic at this level of complexity) and to deliver the full potential of these systems. ...

Historically, scientists and engineers have concentrated on studying and harnessing natural phenomena which are well modeled by the laws of gravity, classical and non-classical mechanics, physical chemistry, and so forth. In so doing, we typically deal with quantities such as the displacement, velocity, and acceleration of particles and rigid bodies, or the pressure, temperature, and flow rates of fluids and gases. These are "continuous variables" in the sense that they can take on any real value as time itself "continuously" evolves. Based on this fact, a vast body of mathematical tools and techniques has been developed to model, analyze, and control the systems around us. It is fair to say that the study of ordinary and partial differential equations currently provides the main infrastructure for system analysis and control.

But in the day–to–day life of our "man–made" and increasingly computer–dependent world, we notice two things. First, that 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 on a runway, how many telephone calls are active). And second that what drives many of the processes we use and depend on are instantaneous "events" such as the pushing of a button, hitting a keyboard key, or a traffic light turning green. In fact, much of the technology we have invented and rely on (especially where digital computers are involved) is event–driven: Communication networks, manufacturing facilities, or the execution of a computer program are typical examples.

Let us consider some examples which show the importance and proliferation of systems whose underlying model deals with discrete events:

Business Processes Business processes can be found in almost all organizations such as banking, insurance, government and industry. The term is very often used

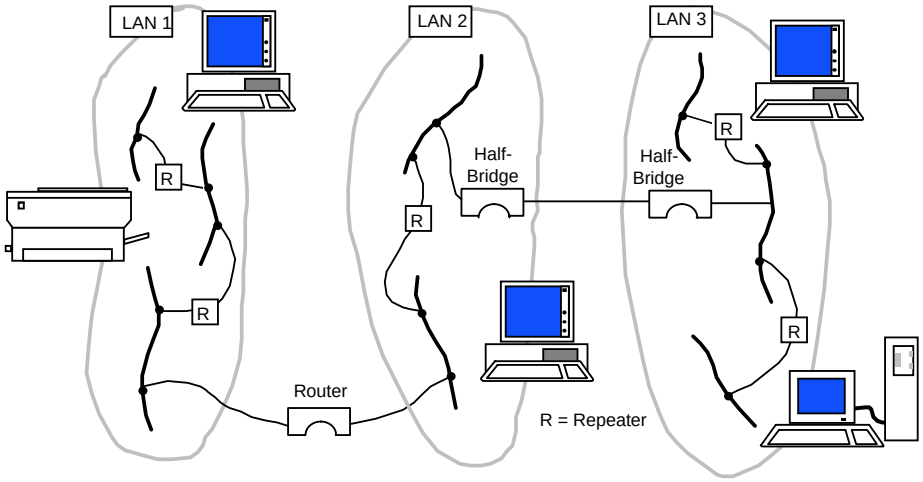

Figure 1: Example of a communication network.

to describe all the operational, logistical and functional activities which occur during e.g. accounting, budgeting, payroll, product design, development and sales. But of course, also the planning, scheduling, control and job setup belongs to the area of business processes.

For example, one may consider the processing of documents. Clearly, we are faced with discrete events of the type ‘a document arrived’ or ‘a document has been processed’. Communication between processes may be described as ‘passing documents’. In addition, it may be important to specify the time, which a document needs to be processed. Even in this small example, we find many of the elements of discrete event systems, for example queues of documents which need to be processed by some server or agent.

Nowadays, the use of information systems to analyze the business processes in a large organization for detecting inefficiencies, to set up new processes and to control them in operation is an emerging field. The models and methods applied here are based exactly on those foundations which will be described in later chapters. It is widely accepted that the increasing productivity in organizations stems to a great extend from these new methods for designing and optimizing business processes.

Communication Networks Communication networks are among the first areas where the theory of discrete event systems has been applied successfully, see e.g. Fig. 1. Events occurring at a particular component may be phone calls, the arrival of a file to be printed or the request to a web sever to deliver a certain page. Again, at places where there are resource conflicts we find queues which store incoming request until they can be served.

Typical questions which are solved using discrete event theory are the analysis of traffic on the communication links, the optimization of the network structure and choosing appropriate lengths of queues.

Production Processes Let us suppose we want to determine the performance of a machine which assembles a car. There are many robots arranged in a production line. A model of the whole system of course has many sub-components which are closely related to classical system theory. They describe for example the movements of robot arms or the algorithms which control the movement of the transport belt or other actors.

On the other hand, looking at the production process as a whole, one can see a lot of discrete type of actions going on. The produced car arrives at a assembling station, the robot picks up a wheel, it fixes the wheel, picks up screws and so on. Finally, the car leaves the station. All these events occur at some time instance and a certain time (determined by the necessary work to be performed) passes between successive events. In addition, the system is highly parallel, as all the assembling stations work simultaneously.

Nowadays, the design and optimization of production systems is done using computer aided design methods which are based on discrete event system theory. The complexity of the relation between subsystems and the inherent parallelism is overwhelming for an analysis and optimization by hand.

Computer Systems As has been described already, many processes occurring in communication systems are of discrete type. If we zoom into the architecture of a computer, we see again communication processes which are receiving and sending events. We see software tasks (for example from different users of a computer) which are sent in sequence to the processing unit, requests for using input/output devices such as a network controller, machine instructions entering the execution pipeline or those waiting in reservation stations for being accepted for processing. Zooming further into the circuit we can recognize Boolean signals changing from true to false. These discrete events are processed by gates and communicated to other components.

Here, as in all the other areas described above, one major task is the simulation of the overall system behavior. How to accomplish this task efficiently is another challenging question closely related to discrete event system theory.

Complex Software Systems Finally, if looking at recent concepts in the construction of large complex software systems, we can recognize the increasing importance of (discrete event) models. For example, a still simple software systems dealing with a user interface consisting of several windows, see Fig. 2. The user can enter information into these windows and the result of some computation is presented eventually within another one. A model which reflects the underlying concepts may consist of many logically distributed processes which are executed concurrently and which are communicating via events such as ”the user has made an input” or ”change the color of your window”.

The concept of software components (Java Beans, Microsoft DCOM) is based exactly on these ideas.

A common property of all the above application areas is that the computer plays two important roles here. On the one hand, he is used as a tool to analyze, design and optimize processes. But it is also part of the process, either as the process itself (computer systems, complex software systems, communication systems) or as part of the information, communication or control infrastructure (business processes, production

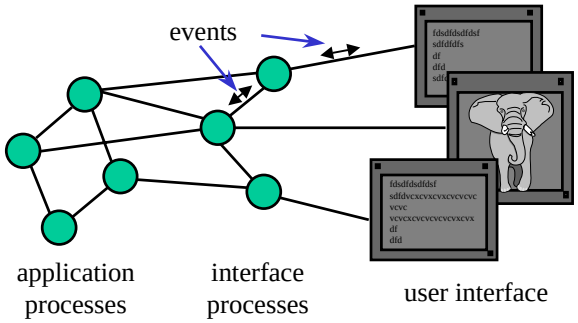

Figure 2: Example of a software system.

systems). This leads to a cycle, in which the computer is involved at many places: in the forward path as a part of the system under design and in the feedback path as part of the analysis and design phase.

After being convinced that event driven systems play an enormous role in almost all areas of engineering, we should ask ourselves why we should deal with the theory of those systems. The following two sections are devoted to this question.

1.2 Elements of System Theory

Looking at the above examples, we can recognize one major source of complexity. Dealing with a software system consisting of a few easily understood communicating tasks is particularly simple. The same holds for a computer system which is composed of a few simple subsystems or a communication network just connecting two computers.

Therefore, the complexity stems from the fact, that there are several (or many) interacting subsystems. The overall system behavior is not only determined by the components individually but also from their interaction. Why does the computer system suddenly stop processing? Which sequence of internal events caused a certain malfunction? How should we choose the lengths of queues in a communication network such that no information gets lost during transmission? To answer these questions, we need to consider the system as a whole. One of the major achievements we expect from a system theory is the relation between microscopic and macroscopic behavior.

During system analysis we expect a path from the parts to the whole. For the design, i.e. synthesis and optimization of systems, we need a constructive path from the overall behavior to that of subsystems and their communication. In short, system theory should help us in understanding

$$\text{components} + \text{communication} = \text{system}$$

Classical system theory of time driven systems has developed a rich mathematical machinery to explore these dependencies. For example, if we have a linear time-invariant system, components can be described by their impulse response in the time domain or by their transfer function in the frequency domain. System composition is equivalent to convolution or multiplication, respectively. Another example is the analysis of electrical networks. The properties of individual network elements such as

resistors, capacitors or controlled sources are combined to the overall behavior of the network through methods of network analysis, e.g. graph representation of the network structure, trees or cycles, node admittance matrix or circuit impedance matrix. The interconnection of subnetworks which are modeled this way can be performed by linear algebra using port matrices as component descriptions. The resulting model of the whole network can then be used to analyze the behavior using well developed methods such as solving the corresponding (partial) differential equations.

The mentioned results (and many others too) are based on a few central terms:

- *time*,
- *signal*,
- *system*, transforming input to output signals
- *state* and state transition.

Interestingly, the same basic terms can be used while dealing with event driven systems, though with a completely different theory behind. Therefore, these basic terms will be introduced in Section 2.2.1 by means of the classical time driven system theory and extended to event driven systems. During the course of this book we will try to get useful answers to questions like the following ones:

- Which (sub)systems should be modeled as event driven systems, where should we use the theory of time driven systems?
- Suppose, we identify a subsystem as event driven. How can we model signals? Which model of computation should we use to characterize the behavior of the system, what are the alternatives ?
- How can we derive the overall system behavior from descriptions of its components, either event or time driven ?

Later on, we will see that another source of complexity is the heterogeneity of todays systems. Very often, we are faced with communicating components which are time driven and event driven, working in discrete or continuous time and are implemented electronically in hardware or software or even mechanically. Examples of this kind of system are embedded systems, see Section 2.6.1.

1.3 Purpose of System Theory

While dealing with these basic issues, we will be able to handle problems which occur in the analysis, design and optimization of event driven or mixed time/event driven systems. Therefore, another major focus of this book is to introduce methods and tools for this purpose and to get answers to questions of the following form:

- Can we say something about the correctness of a discrete event system, i.e. in comparison to the formal specification of their properties?
- How can we efficiently simulate the behavior of discrete event systems?
- How does the developed theory help us in designing and optimizing systems?

- What are the available (commercial) software tools which support the analysis or the design of discrete event systems? On which models and methods are they based?

Especially the last item again points out, that computers play a twofold role in discrete event systems. They are very often not only part of the system under consideration but they also are indispensable as tools for the computer aided analysis and design.

The purpose of the following section is to embed discrete event systems in the classical system theory which primarily deals with time driven systems and the corresponding signals.

2 Systems and Models

This section does not provide a formal treatment of discrete event systems but a gentle introduction into the subject.

There is no generally accepted definition of a system, but the following statement may help to understand the term and the context we are using it in.

Definition 2.1 *A **system** is a combination of components that act together to perform a function not possible with any of the individual parts [IEEE Standard Dictionary of Electrical and Electronic Terms].*

The major features in this definition are the 'components', the 'communication or interaction' between these components and finally the 'function' it is presumably to perform. On the other hand, identifying a system and its components in a particular situation is very much related to the given objectives.

Example 2.2 *For example, let us look at an automatic braking system in a car. The embedded electronics may consist of a microprocessor on which some software is being executed and of an integrated circuit which contains some dedicated hardware for processing the sensor data. From the electronic system point of view, the components are the memory, the microprocessor, the dedicated hardware components and the bus for communication. These units act together in order to perform the function of the embedded electronics. On the other hand, one may also consider the whole braking system and identify it components as the sensors, actuators and the embedded electronics. One may even consider the whole car as the system under consideration. One of its components may be the braking subsystem.*

Obviously, if we want to talk about systems and their components in a way that allows simulation, analysis and design, we need some means of representing their properties.

Definition 2.3 *A **model** is a formal description of a system or subsystem which covers selected information or knowledge.*

This description contains several aspects of a model. At first, it should be formal, i.e. it should contain the information in an unambiguous way, leaving no degree of freedom for interpretation. The next property of a model is the inherent abstraction by representing only selected information. Modeling and choosing the right model is a complex activity of abstracting information. It is driven by the perspective of the modelers and their particular goals.

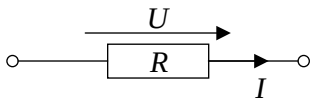

Figure 3: Model of a resistor.

Example 2.4 *In Fig. 3, the model of a resistor is shown. What is the selected information in this case? Obviously, the model only concerns part of the electrical behavior of a resistor, not its color, its weight or its temperature. In addition, only the stateless behavior (without inductance or capacitors) is represented here.*

Until now, a model has been described only by the information or data it contains. But, the term 'information' in the above definition points to another important property. The information not only concerns data which are representative for the system or subsystem. It is also necessary, to link these data to methods which allow the use and processing of these data. In this sense, the term model is closely related to abstract data types in computer science and software engineering: data and methods to process these data are bound together and cannot be separated.

Example 2.5 *The model according to Fig. 3 can be used to apply the usual analysis methods for electrical networks. In particular, Ohm's Law could be applied with* $U = RI$*. As network analysis is widely used and based on firm physical and mathematical ground, the model in Fig. 3 satisfies requirements for a (well behaved) model.*

In essence, the formal specification of systems or subsystems via models is a prerequisite for most of the activities found in engineering (as relevant for the topic of the book), such as

- checking system properties such as reliability, fault tolerance, testability, functionality and timeliness in an early stage of design,
- simulation for the purpose of validation, i.e. checking the behavior of the system for some inputs,
- verification of the behavior of a system by mathematical proof techniques,
- use of computer aided design methods for the design, optimization and analysis and
- documentation of the system and the design process.

From above discussion, it should be clear that suitable models of computation have to satisfy a number of requirements:

composability A model should provide well defined interfaces. This way, there is the possibility to interconnect models of subsystems and to determine a model covering the composed system. Remember one of the main purposes of a system theory, namely combining microscopic to macroscopic behavior.

simplicity In addition to their formal quality, models should represent the properties of a system in a concise way as they very often serve as a documentation as well.

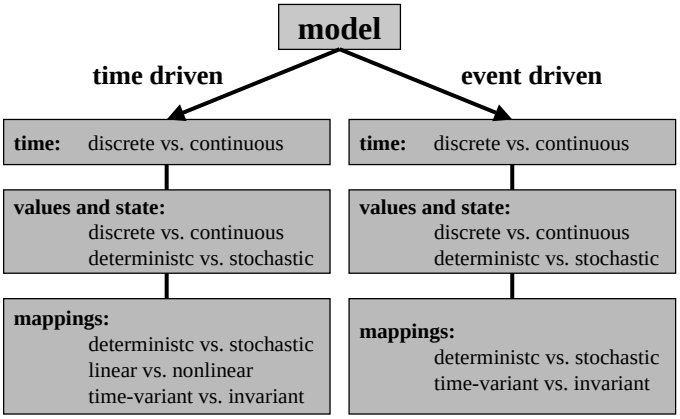

Figure 4: Classification of time driven and event driven models.

methods Models should support a rich class of methods which can be applied, for example simulation, formal verification of system properties (mathematical proof of correctness) and performance analysis.

A major class of methods is devoted to the design of systems through stepwise refinement. During the design process, the level of abstraction of subsystems is gradually reduced. For example, the design of a complex digital circuit may start from a coarse grained functional system description which is refined in the next step into components such as memory, microprocessor and communication system. The microprocessor may then be refined into its control and data path which subsequently are represented as networks of logical gates. This process of refinement can be continued until we reach the level of transistors or even that of semiconductor physics. Each of these abstraction layers has its own models. In order to be able to follow such a design process, models must support methods which transform them to the next lower level of abstraction.

For the purpose of system analysis, the inverse path should be supported by methods as well. Starting from interconnected models on one level of abstraction, one should be able to construct a coarse grain model which abstracts form details in the lower level. For example, given a microprocessor as a network of ten million transistors, we may be interested in simulating the execution of a long program. As the detailed consideration of all network elements through classical nonlinear network simulation takes much too long, we at first build a digital model abstracting from continuous signals in continuous time. This coarser model can be simulated much faster.

2.1 Classification

The purpose of this section is to embed discrete event systems into the well known classical system theory. The main purpose is to make clear the difference between a time driven and an event driven system modeling. According to Fig. 4, in both areas we are dealing with terms such as time, state, values and mappings/relations between state, input and output values. As well time driven as event driven models can be

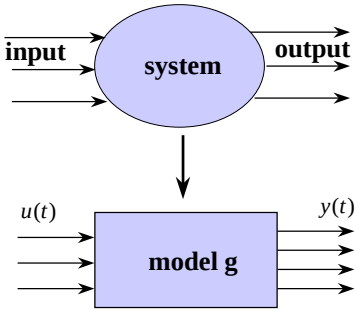

Figure 5: First step in modeling process.

classified using characteristics such as discrete vs. continuous, linear vs. nonlinear or deterministic vs. stochastic.

As we suppose that most of the readers a familiar with the basic concepts of classical system theory, some of the above terms will be introduced by means of time driven systems.

2.2 Input and output representations

2.2.1 Time Driven Systems

Fig. 5 shows the result of the first step in a typical modeling process for time driven systems: identifying input and output values which serve as stimulus and response. We also recognize the representation of the behavior of a system, namely in form of a function which maps input to output values.

In particular, if the system has n inputs and m outputs, they can be represented as functions mapping time to vectors of input and output values

$$u(t) = \begin{pmatrix} u_1(t) & \cdots & u_n(t) \end{pmatrix}^T \qquad y(t) = \begin{pmatrix} y_1(t) & \cdots & y_m(t) \end{pmatrix}^T$$

If these values are taken from the set $\mathbf{V}$ and the time value is taken from a set $\mathbf{T}$, we have $u : \mathbf{T} \to \mathbf{V}^n$ and $y : \mathbf{T} \to \mathbf{V}^m$. Very often, the set of values is that of real numbers $\mathbf{V} = \mathbf{R}$.

The mapping performed by a system with fixed inputs and outputs can now be represented as a function which maps the input function u to the output function y, in other words

$$y = g(u) \tag{1}$$

where $g : (\mathbf{T} \to \mathbf{V}^n) \to (\mathbf{T} \to \mathbf{V}^m)$.

Example 2.6 *The following example is a very simple system, namely a warehouse containing products. We are interested in the number of products stored in the warehouse depending on the delivered and removed items, see Fig. 6.*

The model we are using can be described as follows:

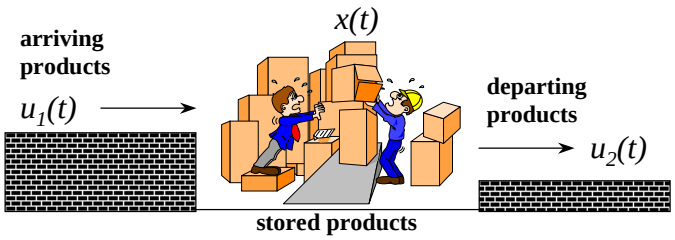

Figure 6: Model of a simple warehouse.

- *The input vector* $u(t) = \begin{pmatrix} u_1(t) & u_2(t) \end{pmatrix}$ *has two elements where*

$$u_1(t) = \begin{cases} 1 & \textit{if a product is being delivered at time } t \\ 0 & \textit{otherwise} \end{cases}$$

$$u_2(t) = \begin{cases} 1 & \textit{if a product is being removed at time } t \\ 0 & \textit{otherwise} \end{cases}$$

- *As we are interested in the number of stored products, we define as our output variable* $y(t) \in \mathbf{Z}_{\geq 0}$ *with*

$$y(t) = \textit{number of items stored in the warehouse at time } t$$

In order to determine the behavior of the warehouse system, i.e. the relation between the input and output values, we will make some further simplifying assumptions: (a) There is no limit on the number of stored items in the warehouse, (b) only one item is delivered or removed at a time and (c) delivering and removing take at least time δ*, (d) the time between two actions is at least* δ *(no simultaneous or overlapping loading or unloading) and (e) no action takes place before time* δ*.*

With these assumptions, we can determine the mapping between input and output variables as follows:

$$y(t) = \begin{cases} y(t-\gamma)+1 & \textit{if } u_1(t-\gamma) == 1 \wedge u_1(t) == 0 \\ y(t-\gamma)-1 & \textit{if } u_2(t-\gamma) == 1 \wedge u_2(t) == 0 \wedge y(t-\gamma) > 0 \\ y(t-\gamma) & \textit{otherwise} \end{cases} \qquad (2)$$

for some $\gamma \leq \delta$ *and* $t \geq \gamma$*. The initial state could be given as* $y(t) = 0$ *for* $0 \leq t < \gamma$*. The above equation defines the function g, see (1).*

Fig. 7 now shows output values $y(t)$ *for some typical input values* $u_1(t)$ *and* $u_2(t)$*.*

The above model is not elegant, obviously. For example, the introduction of the auxiliary variables δ *and* γ *is only an artifact of the time-driven modeling approach. In addition, this is a rather inefficient way of describing the behavior of the system if the time interval between two actions is much larger than* γ*.*

2.2.2 Event Driven Systems

The purpose of this section is to introduce discrete events as signals and discrete event systems as processes which operate on these signals.

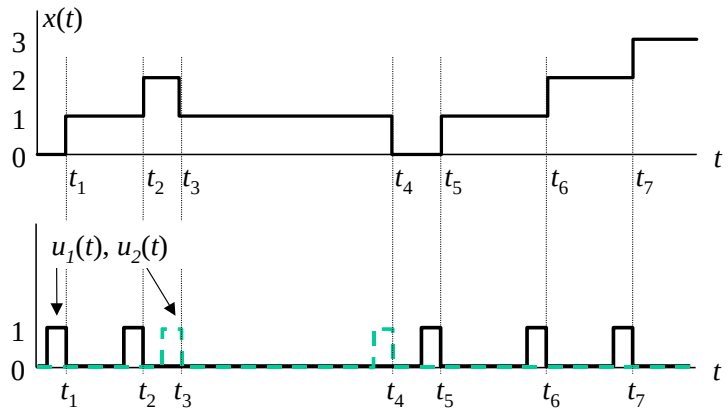

Figure 7: Typical behavior of the simple warehouse.

In comparison to time driven models, in event driven models the transition from one state to another is caused by events. Therefore, time is no longer a suitable independent variable in system modeling.

An event may be identified with actions such as 'a button has been pressed', 'a phone call arrived', 'a customer arrived' or 'a voltage of $100V$ has been exceeded'. As an event may happen at some specific time, we will very often bind events to time or other forms of tags. In other cases, we are only interested in the sequence of events, i.e. whether one event precedes another one. These uses of events leads to the following definition.

Definition 2.7 *An **event** $e = (v,t) \in \mathbf{V} \times \mathbf{T}$ is a tuple of a value $v \in \mathbf{V}$ and a tag $t \in \mathbf{T}$. Here, $v(e)$ denotes the value of an event e and $t(e)$ its tag, i.e. $e = (v(e), t(e))$.*

The tag associated to an event can be used to specify for example the time instance when the event occurs or it can describe precedence relationships, see [2]. In the following, we will restrict ourselves to sets $\mathbf{T}$ whose elements are totally ordered, i.e. for any distinct $t \in \mathbf{T}$ and $t' \in \mathbf{T}$ we have either $t < t'$ or $t > t'$. Examples are $\mathbf{T} = \mathbf{R}$ or $\mathbf{T} = \mathbf{Z}_{\geq 0}$.

Now, we can specify the meaning of a signal in a discrete event system.

Definition 2.8 *A **signal** $s \in \mathbf{S}$ in a discrete event model is a set of events, e.g. $s = \{e_1, e_2, \ldots\}$ where $e_i = (v_i, t_i)$. We also define*

- *the set of events with tag t in signal s:*

$$\mathbf{E}(s,t) = \{e \in s : t(e) = t\}$$

- *the set of tags occurring in signals s:*

$$\mathbf{T}(s) = \{t : (v,t) \in s\}$$

- *the set of values occurring in signals s:*

$$\mathbf{V}(s) = \{v : (v,t) \in s\}$$

- *the set of events in s that have a tag which is larger than and closest to t*

$$\mathbf{next}(s,t) = \mathbf{E}(s,t_+)$$

where $t_+ > t$ and there is no $t_s \in \mathbf{E}(s,t)$ with $t_+ > t_s > t$.

Note that the function **next** is properly defined, if between any two time values $t_1, t_2 \in \mathbf{T}(s)$ there is only a finite number of time values. This is a property of discrete event systems and will be explained further in Section 2.3, see Def. 2.15.

As in the case of values in time driven systems, we very often are concerned with tuples of signals. For example, a model may have several inputs and several outputs. In this case we write $s = \begin{pmatrix} s_1 & \cdots & s_N \end{pmatrix} \in \mathbf{S}^N$.

Example 2.9 *Let us consider again the simple warehouse from Fig. 6 and Example 2.6. The set of values we could consider here is $\mathbf{V} = \{a, r\}$ with the meaning $a \equiv$ 'item arrived' and $r \equiv$ 'item removed'. If we choose the tag values t to denote the time when the event occurs we could describe the signal corresponding to the inputs u_1 and u_2 in Fig. 7 as*

$$s_1 = \{(a,t_1), (a,t_2), (a,t_5), (a,t_6), (a,t_7), (r,t_3), (r,t_4)\} \tag{3}$$

Suppose we add another value to our set, namely $o \equiv$ 'one item in warehouse exceeded'. If we put all occurring events into one signal, we have according to Fig. 7

$$s_2 = \{(a,t_1), (a,t_2), (a,t_5), (a,t_6), (a,t_7), (r,t_3), (r,t_4), (o,t_2), (o,t_6)\} \tag{4}$$

Of course, we could also have put the events with value o into another signal

$$s_3 = \{(o,t_2), (o,t_6)\}$$

and describe all occurring events with a tuple $s = \begin{pmatrix} s_1 & s_3 \end{pmatrix}$.

Following Def. 2.8, the set of events with tag t_2 in signal s_2 is

$$s_2(t_2) = \{(a,t_2), (o,t_2)\}$$

The set of tags occurring in s_2 is

$$\mathbf{T}(s_2) = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$$

and the set of values in s_2 is

$$\mathbf{V}(s_2) = \{a, r, o\}$$

A signal s in a discrete event system can therefore compared to a time dependent value $u(t)$ in a time driven system, see Fig 5. Here, $u(t)$ is a functions from time values t to values u. But there are some major differences which will be pointed out now:

- The tags in a signal need not to be related directly to the physical time, they may also denote just the sequence of events. In this case we may have tags $t \in \mathbf{Z}_{\geq 0}$. For example, if we are not interested in the time instance when items are delivered or removed, s_1 in Equ. 3 could be represented as the set

$$\{(a,1), (a,2), (a,5), (a,6), (a,7), (r,3), (r,4)\}$$

- In time driven systems, $u(t)$ is a function from time $t \in \mathbf{T}$ to a value. As we have seen in (4), this need not to be the case for a signal in an event driven setting. Here, to tag t_2 the two values *a* and *o* are associated. The same holds for t_5. The interpretation of this situation is obvious: at time instances t_2 and t_5, two events or values occurred simultaneously.

  Signals where no two events have the same tag are called *functional*. An example is shown in (3).

Looking at signals like in (3) we recognize that a representation in form of an ordered sequence may be helpful. In this case, the events of a signal are ordered according to the order in which they occurred.

Definition 2.10 *A **stream** $\sigma \in \Sigma$ in a discrete event model is a sequence of events $\sigma = (e_1, e_2, \ldots)$. We use $\sigma(i) = e_i$ with $1 \leq i \leq |\sigma|$. For all elements $i < j$ of σ, the following holds: event e_i happened earlier than e_j.*

Example 2.11 *The signal shown in (3) could be equivalently written as*

$$\sigma_1 = ((a,t_1),(a,t_2),(r,t_3),(r,t_4),(a,t_5),(a,t_6),(a,t_7)) \tag{5}$$

see also Fig. 7. This representation will also be used in this book if appropriate.

Now, we can describe how signals are processed. In (1) we have represented the mapping of a time driven system in form of a function *g* which maps an input function *u* into an output function *y*. In a similar way, we also define a process in an discrete event system.

Definition 2.12 *A **process** P maps a tuple of input signals $u = (u_1 \cdots u_{N_i}) \in \mathbf{S}^{N_i}$ into a tuple of output signals $y = (y_1 \cdots y_{N_o}) \in \mathbf{S}^{N_o}$, i.e. $P : \mathbf{S}^{N_i} \to \mathbf{S}^{N_o}$.*

A similar definition can also be given using the concept of streams instead of signals. Intuitively, the input signals of a process are not constrained by the process whereas the output signals are. There are many different ways of specifying the behavior (function) of a process. In the book we will learn more about different models such as finite state machines, Petri Nets and data flow graphs, just to name a few. Similar to the theory of time driven systems, many properties of processes can be identified such as causality or determinacy.

2.3 Time

The definitions of input and output values, signals and streams use a notation of time, in particular $t \in \mathbf{T}$ where $\mathbf{T}$ is totally ordered. Therefore, for any distinct $t \in \mathbf{T}$ and $t' \in \mathbf{T}$ we have either $t < t'$ or $t > t'$. Now, we can say something about the distinction between continuous time and discrete time systems.

In the following definitions, we use the term 'the set $\mathbf{T}$ of admissible time values'. This means that in order to specify the functionality of a system, we only need to refer to time values in $\mathbf{T}$.

Definition 2.13 *In a **continuous time system**, the set $\mathbf{T}$ of admissible time values is the set of real numbers, i.e. $\mathbf{T} = \mathbf{R}$.*

Therefore, between any two finite time values of a system in continuous time there is an infinite number of other admissible time values and any time value $t \in \mathbf{R}$ is admissible. Examples of systems in continuous time are models of classical mechanics and electronic circuits.

Definition 2.14 *In a **discrete time system**, the set* $\mathbf{T}$ *of admissible time values is the set of integers, i.e.* $\mathbf{T} = \mathbf{Z}$.

There are many examples of time-driven discrete time systems. Almost any digital computer we are using operates in a periodic discrete time fashion. Successive time values are separated by a fixed time interval, the period T. These clock ticks are generated by a clock generator and distributed to all other components. We also know time-driven systems which use the same concept of discrete time, e.g. systems dealing with sampled signals. Here, the classical system theory has a rich set of tools at hand, e.g. the so-called z-transform. Moreover, if we try to analyze a time driven system in continuous time numerically on a digital computer, we very often have to discretize the time axis and solve the equations iteratively, time step after time step. This way, differential equations are transformed into difference equations.

Unfortunately, we are faced with additional difficulties if we deal with discrete event systems. As mentioned already, in time driven systems, the time t is an independent variable that 'advances' independently of the system under consideration. In contrary, the advance of time in event driven systems is determined by the system itself. The distances between events may not be determined by a fixed clock period but may vary depending on events occurring in the system. The following definition guarantees the necessary advance of time, see also Def. 2.8:

Definition 2.15 *In a specific execution (or run) of a discrete event system, let* $\mathbf{T}_r$ *denote the set of tags* t *occurring in all signals* s*, i.e.* $\mathbf{T}_r = \bigcup_s \mathbf{T}(s)$*. For any execution and for any two time instances* $t_1, t_2 \in \mathbf{T}_r$ *there is only a finite number of other time instances between* t_1 *and* t_2.

The above fact is trivially satisfied in the case of a discrete time system as $\mathbf{T}_r \subseteq \mathbf{Z}$.

2.4 State

In the theory of discrete event systems, the notation of a state plays an important role.

Definition 2.16 *The **state** of a model at time* t_0 *contains the information which is necessary to determine the output for all* $t \geq t_0$ *from this information and from the input for* $t \geq t_0$.

Remember, that in the case of a time-driven system, input and output are characterized by functions $u(t)$ and $y(t)$, respectively. Usually, the state information is represented as a vector similar to the input and output values, i.e.

$$x(t) = \begin{pmatrix} x_1(t) & \cdots & x_p(t) \end{pmatrix}^T$$

In the case of an event-driven system, the state x can be represented in many different forms. No general approach is known. Sometimes, the state is represented in form of a signal or stream, sometimes it is represented as a function $x(t)$.

Definition 2.17 *The set* $\mathbf{X}$ *of possible states of a system is called its **state space**.*

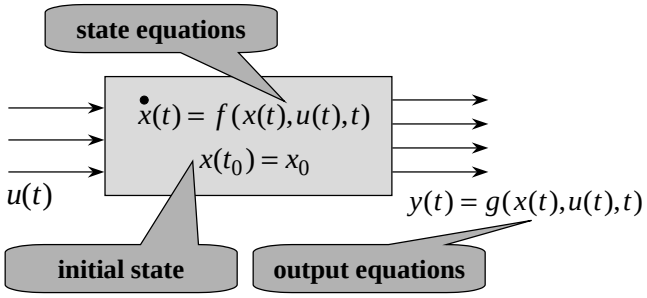

Figure 8: State space modeling of continuous time driven systems.

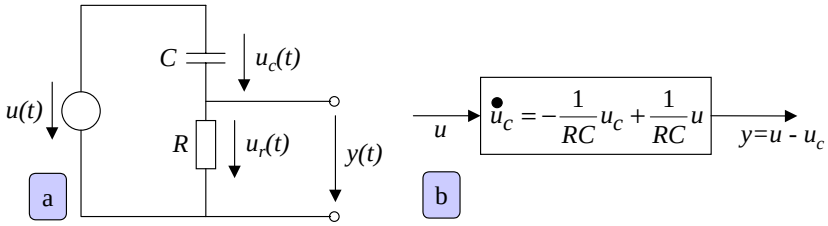

Figure 9: A simple electrical network (a) and its state space model (b).

Using the notation introduced so far, the modeling process for a time driven system yields suitable mathematical relations between input, output and state.

Definition 2.18 *Equations which determine the state x for all $t \geq t_0$, given and the input for all $t \geq t_0$ are called* ***state equations****.*

Definition 2.19 *Equations which determine the output for all $t \geq t_0$, given the state at t_0 and the input for all $t \geq t_0$ are called* ***output equations****.*

These definitions can be compared to the usual state space representation in system theory as shown in Fig. 8. In the case of time driven linear systems with continuous time, this modeling approach is one of the major approaches and has been extensively studied using a variety of techniques. For example, applying Laplace transform in order to obtain a set of algebraic equations instead of differential equations and representing the model in frequency and/or time domain.

Example 2.20 *Let us consider the very simple circuit shown in Fig. 9. We are interested in the relation between the input voltage $u(t)$ and the output voltage $y(t)$.*

Using simple techniques from network theory we obtain

$$i = C\dot{u}_c \qquad u_r = iR \qquad u = u_r + u_c \qquad y = u_r$$

A natural choice for the state variable is u_c. Note that from network theory it is known that usually voltages on capacities and currents through inductances are state

variables. We now obtain the state equation

$$\dot{u}_c = -\frac{1}{RC}u_c + \frac{1}{RC}u$$

and the output equation

$$y = u - u_c$$

Obviously, the considered circuit is time-invariant and can be modeled in continuous time.

We have been classifying systems according to their time domain into discrete time and continuous time. In a similar way, we could classify the admissible values of input and output variables and those of the states.

Definition 2.21 *In **continuous state** models, the state space* $\mathbf{X}$ *of admissible state vectors is connected. In a connected set* $\mathbf{X}$ *there do not exist two nonempty disjoint open subsets* $\mathbf{X}_1, \mathbf{X}_2 \subseteq \mathbf{X}$ *with* $\mathbf{X}_1 \cap \mathbf{X}_2 = \emptyset$ *and* $\mathbf{X}_1 \cup \mathbf{X}_2 = \mathbf{T}$.

*In **discrete state** models, the state space* $\mathbf{X}$ *is isomorphic to a subset of the integers, i.e. countable.*

Example 2.22 *The model of the circuit in Fig 9 uses the state variable* u_c. *The corresponding state space is continuous.*

Example 2.23 *Very often, the state space* $\mathbf{X}$ *of a discrete event system is also discrete, see Def. 2.21. But this needs not to be the case. For example, consider a remote sensor application. The sensor continuously tracks a certain position, velocity, acceleration, temperature, pressure, flow and so on. A user sends asynchronously a message to the sensor to store the current value. Another message from the user is used to retrieve the stored value. As values of events we have* $\mathbf{V} = \{$ *'store', 'retrieve'* $\}$ *but the state, i.e. the stored values, is not discrete but may be considered to be a continuous subset of the real numbers. Of course, during one run of the system, the set of used states is countable as only countably many events can occur. But the set from which these states are taken still is continuous. Therefore, we will refer to a model as a discrete event system whether* $\mathbf{X}$ *is countable, finite, or neither.*

Very often, the dynamic behavior of discrete state systems is simpler to visualize but much harder to analyze. Especially in the case of a finite state space, simple diagrams can grasp the behavior of a system in a graphical way, for example state transition diagrams. For continuous state systems, a lot of well developed algorithmic machinery involving tools and methods can be used for dealing with differential equations or difference equations. Moreover, if the model is linear, many generally applicable results are available.

In order to characterize the evolution of a model, very often a sample path is visualized. These sample paths are nothing else than a graphical representation of the state trajectory $x(t)$ for some subset of $\mathbf{T}$. The next Fig. 10 shows some representations of sample paths.

In these examples, we can recognize different representations of state trajectories and different classifications of state spaces $\mathbf{X}$:

- In Fig. 10a, two possible state trajectories for a single state variable $x(t)$ are shown. They both start at value $x_0 = x(t_0)$ and may be caused by two different input functions $u^1(t)$ and $u^2(t)$.

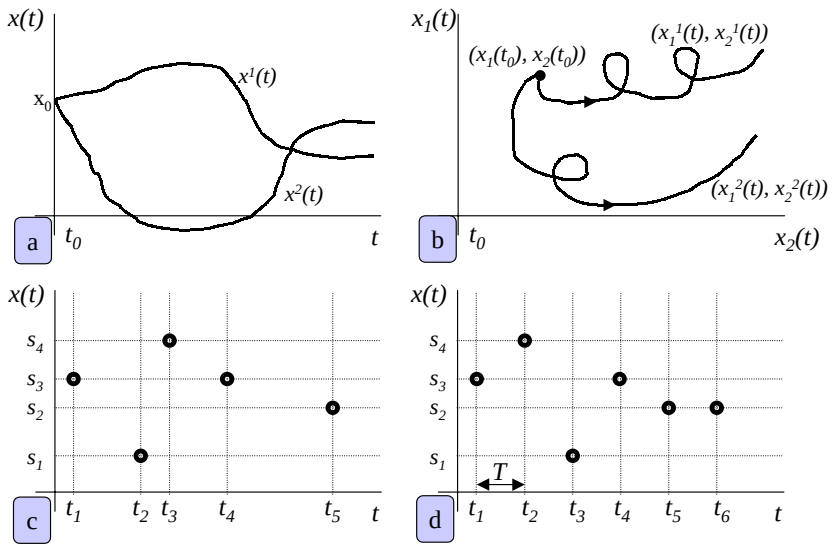

Figure 10: Sample paths (state trajectories).

- In Fig. 10b, two two-dimensional state trajectories are shown in form of parameterized curves $(x_1^1(t), x_2^1(t))$ caused by input $u^1(t)$ and $(x_1^2(t), x_2^2(t))$ caused by input $u^2(t)$.
- In contrary to Figs. 10a and b, the Figs. 10c and d show the evolution of a single state variable for discrete time systems.
- In Fig. 10d, we have a *periodic discrete time* representation which is characterized by equidistant time values, i.e. $t_j = t_i + T$ for time values t_i, t_j next to each other.
- In Figs. 10c and 10d, the state space $\mathbf{X}$ turns out not to be continuous but a discrete set.

2.5 Discrete Event Systems

After introducing all the necessary elements like time, signals and state, we define discrete event systems. Moreover, we will try to make clear the differences to time-driven systems.

Definition 2.24 *A **discrete event system** is an event-driven system, that is, its state evolution depends entirely on the occurrence of discrete events over time. If the admissible time instances are taken from a continuous or discrete set as defined in Defs. 2.13 and 2.14, it is a discrete event system in continuous time or discrete time, respectively. During the run of a discrete event system, between any two time instances there is only a finite number of other time instances, see Def. 2.15.*

Using the above discussion, we can summarize the major properties of discrete event systems (DES) as follows:

- A DES is event driven, i.e. the state transitions are caused by the occurrence of events.
- As in time driven systems, a DES model can be defined in continuous or discrete time depending on **T**.
- The state space of a DES can be either discrete or continuous depending on **X**. In a run of a DES only a countable number of different states can be accessed.
- During the run of a discrete event system, between any two time instances there is only a finite number of other time instances, i.e. i.e. $\mathbf{T}(s) \subseteq \mathbf{T}$ where s is the union of all signals, is order isomorphic to a subset of the integers.
- Ordering and timing of events can be represented in signals (based on sets) or in streams (based on ordered sequences). Processes can be represented as functions acting on signals or streams.

The purpose of the following examples is to make clear the distinction between time-driven and event-driven systems.

Example 2.25 *The simple warehouse described in Example 2.6 can be modeled as a discrete event system in continuous time. The event values* $\mathbf{V} = \{$ *'item arrived', 'item removed'* $\}$ *occur to arbitrary time instances. The admissible set of tags is* $\mathbf{T} = \mathbf{R}_{\geq 0}$. *The state evolution could be described as follows using the stream* σ_1 *in (5):*

$state = 0$; $pos = 1$; $\sigma_{out} = ()$;
do forever
  if $(|\sigma_1| \geq pos)$ **then**
    $e = \sigma_1(pos)$;
    if $(v(e) ==$ 'item arrived'$)$ **then**
      $state = state + 1$;
    if $((v(e) ==$ 'item removed'$) \wedge (state > 0))$ **then**
      $state = state - 1$;
    $\sigma_{out}(pos) = (state, t(e))$;
    $pos = pos + 1$;

Here, the output is also represented as a stream, i.e. σ_{out}*, see Fig 11. Elements are added to the stream using an update of the function* σ_{out} *at the current location pos. Using the stream* σ_1 *in (5) we obtain*

$$\sigma_{out} = ((1,t_1),(2,t_2),(1,t_3),(0,t_4),(1,t_5),(2,t_6),(3,t_7))$$

We could also use signals in order to describe the behavior of the process, see Def.2.8. In order to simplify the discussion, we suppose functional signals, i.e. no two events have the same tag. Remember that the function **next**(s,t) *returns the set of events that have the next larger tag, i.e.* $\tau > t$ *and there is no other event in signal s with a tag between t and* τ*. Now, we can define the behavior of the warehouse process using the following algorithm:*

$state = 0$; $\tau = 0$; $s_{out} = \emptyset$;
do forever
  if (**next**$(s_1, \tau) \neq \emptyset$) **then**

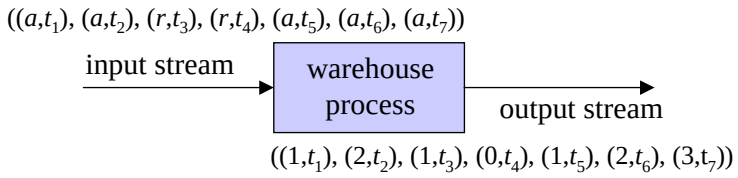

Figure 11: Discrete event model of a warehouse.

$\{(v,\tau)\} = \mathbf{next}(s_1,\tau)$;
if ($v ==$ 'item arrived') **then**
    $state = state + 1$;
if (($v ==$ 'item removed') $\wedge$ ($state > 0$)) **then**
    $state = state - 1$;
$s_{out} = s_{out} \cup \{(state,\tau)\}$;

With input s_1 from (3) we obtain the output

$$s_{out} = \{(1,t_1),(2,t_2),(1,t_3),(0,t_4),(1,t_5),(2,t_6),(3,t_7)\}$$

Now, let us compare this approach to the time-driven model in example 2.6. As has been mentioned already, the model in (2) has some deficiencies. Because of the introduction of the auxiliary variables δ and γ, it is a rather inefficient way of describing the behavior of the system if the time interval between two actions is much larger than γ.

This observation can be made quantitative. Let us suppose that we would like to simulate the above warehouse for a time interval $t \in [0,T]$. One approach would be to discretize the time with period γ. As a consequence, the time resolution is limited to γ and the state equation must be evaluated

$$N = \lfloor T/\gamma \rfloor \tag{6}$$

times. Obviously, a higher resolution necessitates a smaller period γ and a larger computation time as N grows.

In short, from the description of the system one would expect an event driven model. The warehouse can best be described as an discrete event system yielding a more elegant mathematical formulation and a more efficient simulation.

Example 2.26 *A program on a computer may be in one of several states, for example $\mathbf{X} = \{$ 'waiting for input', 'running', 'ready' $\}$ where 'waiting for input' denotes that the program is not executed currently as some input values are required. 'Ready' may denote the state that the program could be executed currently, but the computer is busy with some other program. In state 'running', the program is being executed. Moreover, let us suppose that the computer is driven by a periodic clock. In other words, the set of admissible tags is $\mathbf{T} = \mathbf{Z}_{\geq 0}$. As a consequence, the system can be considered to work in discrete time. But intuitively, the state changes are not caused by the advance of time but by the occurrence of events. For example, events causing a state change may be 'interrupt', 'program execution finished' or 'new program arrived for execution'. Time is not a natural independent variable in a system description. The above scenario describes a discrete event system in discrete time.*

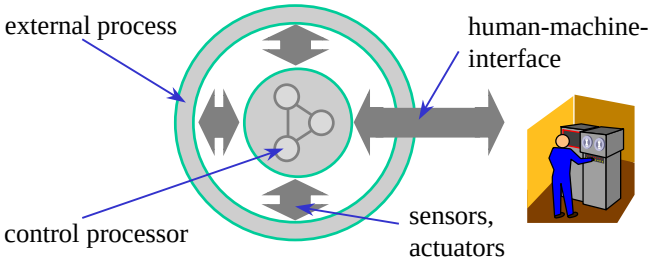

Figure 12: Visualization of an embedded system.

Finally, the next sections contain in some more detail examples of discrete event systems.

2.6 Examples of Discrete Event Systems

2.6.1 Embedded Systems

Especially in the area of information and communication systems, it is widely accepted, that the design of complex systems needs formal models and corresponding methods. As an example, let us consider the class of *embedded system*. In computer science and engineering, this term has come to denote a computer that is physically embedded within a larger system and of which the primary purpose is to maintain some property or relationship between the other components of the system in order to achieve the overall system objective [3]. Embedded computers are economically very important and nowadays they are used in almost all technical systems such as automobiles, transportation systems, home electronics, medical and communication devices, see Fig. 12.

Embedded systems have certain characteristics that greatly complicate the process of designing their hardware and software components. These characteristics are heterogeneity (hardware vs. software, sensors/actors vs. signal processing, analog vs. digital), real-time requirements, reactive behavior, and fault-tolerance and predictability for use in safety-critical applications.

Developers and engineers are faced with new problems when it comes to specifying, simulating, designing, and optimizing such complex systems. Because of the intricate relationship between an embedded computer and its environment, the design of embedded system is particularly driven by cost vs. benefit trade-offs. Examples of benefit metrics are reactiveness to events, performance in stream processing, power consumption, size of program and data memory, hardware cost and design time. Due to these constraints, implementations typically comprise as well software programmable components (e.g., processors, microcontrollers, digital signal processors), dedicated hardware components (e.g., ASICs), communication and memory subsystems.

General-purpose design methodologies, such as the object-oriented technology, fail or are at least not completely satisfying for embedded systems. Experience has shown that these methodologies with their single model of computation cannot meet the strong requirements in the embedded system domain.

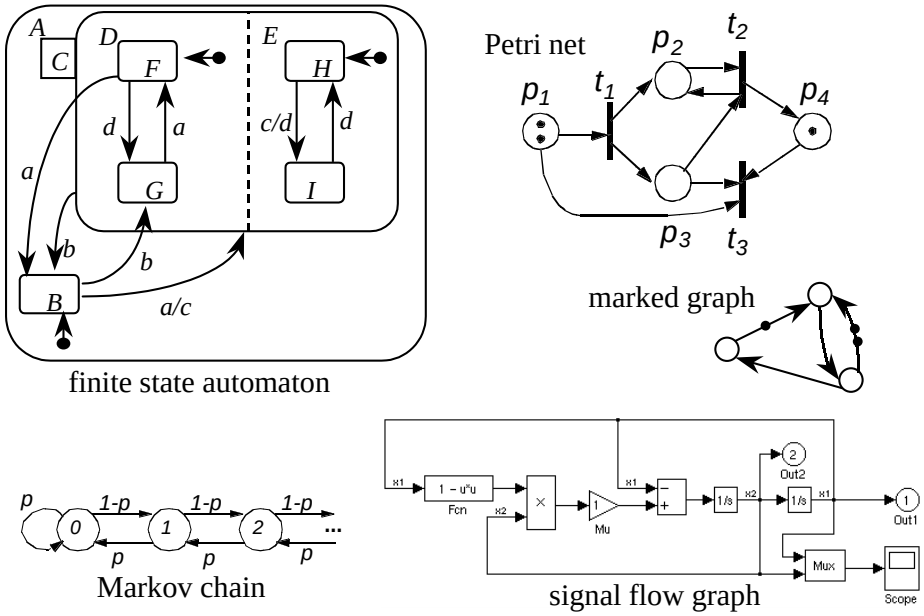

Figure 13: Widely used models of computation.

2.6.2 Models of Computation

For embedded systems, mostly a complementary model-based view is applied. In order to achieve optimal systems, different models of computation and different types of optimizations must be brought together. Engineers, for example, draw annotated diagrams of components (block diagrams) or states (finite state machines) to describe the high-level architecture or function of a system. In Fig. 13 some of the most popular models and graphical notations are summarized. Diagrams of this kind are used in many different disciplines such as computer architecture, dynamic systems, control theory, signal processing and communications. But the meaning of these diagrams is usually quite different.

It is one of the purposes of the book to explain the semantics of the models behind diagrams such as in Fig. 13. Besides 'classical' examples such as ordinary or partial differential equations we find

- Petri nets,
- data flow descriptions such as marked graphs, synchronous data flow graphs or Boolean data flow graphs,
- Kahn process networks,
- (stochastic) finite state machines (Markov chains),
- Harels hierarchical extended finite state machines,
- synchronous reactive models,

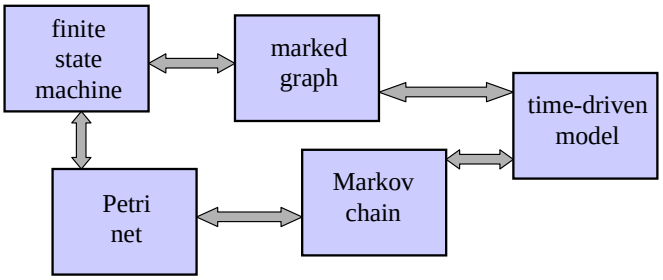

Figure 14: Communicating models of computation.

- communicating sequential processes (CSP),
- queuing systems and
- general discrete event models.

Complex systems can rarely be described and designed using a single model only. In many applications of embedded systems, the transformative domain (data processing, stream processing) and the reactive domain (reaction to discrete events, control flow) are tightly interwoven. Examples include mode and parameter control of dataflow processing systems, system configuration and initialization, e.g., in packet based transmission systems, wireless modems, etc. As different kinds of problem domains lead to different formalisms and models, they also capture domain specific knowledge of the design process, hardware/software architectures, and computation paradigms. Therefore, in contrary to purely time-driven systems, we find two main structuring principles for systems, see Fig. 14:

- The usual *vertical structuring* following different levels of abstraction, e.g. from a coarse grain specification to fine grain descriptions.
- A *horizontal structuring* on one level of abstraction. Here, several communicating components of a system are model on the basis of different models of computation.

As an advantage, these domain-specific models of computation enable the use of specialized methods within the design process, such as verification, scheduling with deadlines and throughput constraints, and efficient code generation.

On the other hand, we are faced with additional problems. For example, even if we understand the different models of computation in isolation, we need to relate the individual behavior to that of the whole system. In particular, the interfacing different models causes additional difficulties. For using state-of-the-art methods in system design, a basic foundation of the underlying models is indispensable in order to make best use of their capabilities, and to correctly understand their semantics and their communication capabilities for the purpose of system specification, implementation or analysis.

Further informations can be found in recent research projects, e.g. Ptolemy at UC Berkeley `http://ptolemy.eecs.berkeley.edu/` or Codesign and Moses at ETH, see `http://www.tik.ee.ethz.ch/ moses`. In addition, software engineering tools are coming to the marked leading to a software development process

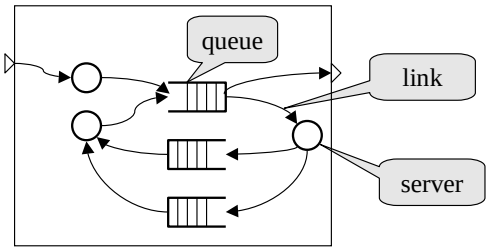

Figure 15: Example of a queuing network.

based on domain-specific models of computation, e.g. the DOME project at Honeywell `http://www.htc.honeywell.com/dome/index.htm`.

2.6.3 Queuing Systems

Queues are important components in many discrete event systems. They occur mainly, if we are faced with limited resources. As a consequence, entities have to wait until they can be served. Some examples of typical situations are as follows:

- People are waiting in lines (in restaurants, in bank lobbies, at bus stops, in supermarket, in front of a telephone) before they are served (server, bank teller, bus, supermarket checkout teller, telephone).
- Software tasks, transactions or jobs are waiting in a CPU (central processing unit) of a microprocessor or in one of the peripheral devices before being processed.
- Messages, packets or calls are waiting within routers or other switching equipment within a communication network for an available output line.
- Production parts in a manufacturing process are waiting in front of a machine which will process or handle them.
- Cars are waiting in front of a traffic light until an intersection can be used and the traffic light turns green.

If looking at these examples, we can recognize three important elements in a queuing systems, namely the *entities* which are waiting or being processed (people, tasks, messages, production parts, cars), the *resources* for which the waiting is done (bank teller, CPU, output line, production machine, intersection) and finally, the *space* where the waiting is done (queue in bank lobby, task queue in a processor, message queue in a router, queues of parts, queues of cars). The entities, resources and space are often called *customers*, *servers* and *queues*, respectively.

The following figures shows a typical representation of a queuing network consisting of servers, queues and links. These links visualize the paths on which customers flow through the network.

Obviously, this representation is quite abstract as many details are missing. Later on, we will learn about more formal specifications like Petri nets.

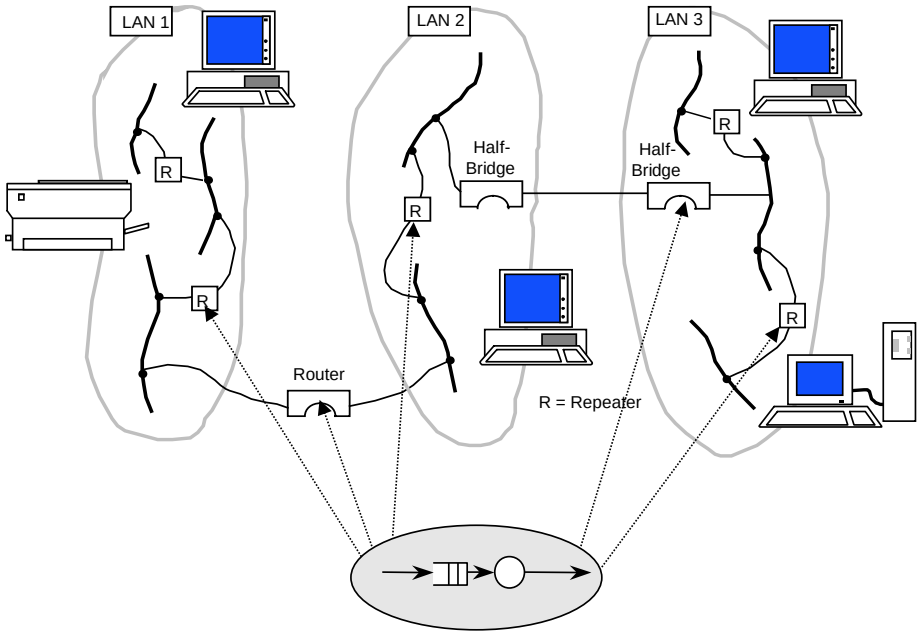

Figure 16: Example of a communication network.

In order to characterize a queuing system in more detail we need to specify many additional properties. For example, if two links are connected to the output of a single queue or server, we should now which of the possible paths are taken by a specific customer.

Moreover, there are many different *queuing disciplines*, i.e. rules which determine the next customer to be served. One widely used rule is called first-in first-out (FIFO) as the customers are served in the ordering of their arrival. But there are also other possibilities such as last-in-first-out (LIFO) where the customer are served in reverse order of their arrival.

In addition, queues very often have a limited *capacity* only. As in this case only a finite number of customers can wait in a queue we need to specify what happens with customers arriving after the queue is complete.

Other properties we may need to specify is the timing of events in a queuing networks a server may need some time to deal with a customer, i.e. to process an event.

A typical discrete event model of a queue involves events with values $\mathbf{V} = \{a, d\}$ where a denotes the arrival of a customer and d denotes its departure. A typical state variable x may denote the number of waiting customers (or the queue length), i.e. $\mathbf{X} = \{0, 1, 2, 3, \ldots\}$. In this sense, the model is very similar to the celebrated simple warehouse in Example 2.6. Finally, some examples of queuing systems are described in order to find out some typical analysis and design problems.

Communication Networks As can be seen in Fig. 16, servers in a communication network may be switching equipment which selects the correct routes for transmitted messages, communication media and of course all kind of other peripheral devices.

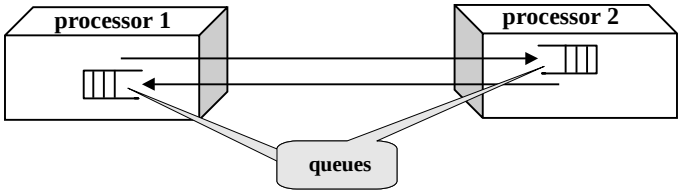

Figure 17: A point-to-point communication.

Even if only to computers are sending messages to each other via a dedicated point-to-point link, queues are involved in order to compensate for different execution speeds and reaction times of the devices, see Fig. 17.

Typical questions we are faced with are the determination of the latency or throughput of a network, the jitter which changes the time difference between subsequent messages and the number of lost messages. These quantities are very often determined through a formal analysis or extensive simulations.

On the other hand, the optimization of communication networks involves sophisticated control procedures which work as well locally on each server as well as globally over the whole network. The purpose of these protocols is to ensure a fair and efficient processing of messages such that latency and jitter are minimized while maximizing the throughput.

If the network structure and its components are not given beforehand, further questions concern the optimal allocation of resources such as communication lines and switching devices and the choice of an optimal network topology.

Recently, another requirement has gained lots of interest. It is expected that in many future communication networks, billing and accounting of used network services will play an important role. Maximizing the profit will be of primary concern in this new generation of protocols, as well for the users as for the network providers. Again, the underlying models are event driven and the complexity of the involved design problems prohibits a purely ad hoc approach.

Computer Systems Fig. 18 shows the block diagram of a typical computer system containing a processor (central processing unit), memory and peripheral devices. In almost all of these components, we find queues.

Events in such a system may be user interactions, generated software tasks, input/output requests, interrupts, cache misses or page faults. Even in the CPU we find may queues because of the inherent resource conflicts. For example, data to be stored in a second level cache or main memory are stored in a write buffer until the memory has time to serve the write request. In a modern superscalar processor architecture, in front of any execution unit there are buffers for instructions which have been fetched but whose corresponding execution unit is still busy with processing. Of course, in all the I/O controllers we find queues for requests which are waiting for being served.

Because of the complex interplay of all these queues and processing units, analyzing and optimizing a computer system is a complex task. Again, this complexity mainly stems from the *interaction* of many complex components and from various conflicting criteria such as price, performance and power consumption.

The following example taken from [1] shows a possible state representation of a

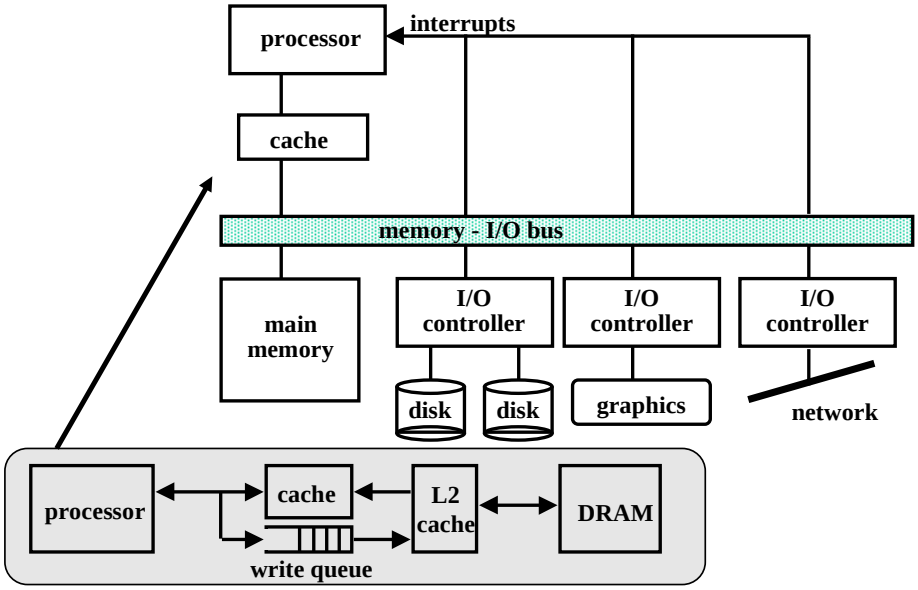

Figure 18: Block diagram of a computer system.

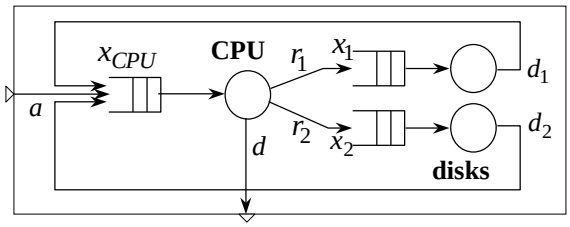

Figure 19: A computer as a queuing system.

very simple computer system.

Example 2.27 *Fig. 19 shows the model of a CPU, two disks D1 and D2 and several queues. Jobs arrive from the left input to the system and are queued before being processed. The CPU either delivers the results to the output or generates disk requests for D1 or D2. After the disks have found the necessary data, the jobs are again forwarded to the CPU.*

A possible modeling uses events with values from the set

$$\mathbf{V} = \{a, d, r_1, r_2, d_1, d_2\}$$

where

a, d	*denote the arrival or departure of a task*
r_1, r_2	*denote the arrival of requests for the disks D1 and D2*
d_1, d_2	*denote the departures of tasks from disks D1 and D2*

A possible state representation uses $x = \begin{pmatrix} x_{CPU} & x_1 & x_2 \end{pmatrix}^T$ *and the state space*

$$\mathbf{X} = \left\{ \begin{pmatrix} x_{CPU} & x_1 & x_2 \end{pmatrix} : x_{CPU}, x_1, x_2 \geq 0 \right\}$$

where the state variables x_{CPU}, x_1, x_2 *denote the lengths of the queues in front of the CPU, disk D1 and disk D2, respectively.*

References

[1] C. G. Cassandras. *Discrete Event Systems: Modelling and Performance Analysis*. Aksen Associates Inc. Publishers, Homewood, IL and Boston, MA, 1993. ISDN 0-256-11212-6.

[2] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. Technical Report UCB/MRL 97/11, University of California, Berkeley, march 1998.

[3] Anthony Ralston and Edwin D. Reilly. *Encyclopedia of Computer Science*. Chapman and Hall, London, 1993.